№9.

1) ООП в Scheme выполняется с использованием замыканий и функций высшего порядка для симуляции классов, методов и инкапсуляции. Scheme не имеет встроенной поддержки ООП в традиционном смысле, но его гибкость позволяет имитировать ООП-концепции

Пример описания класса "Транспортное средство" в Scheme и Racket:

Scheme:

```
(define (make-vehicle color max-speed)
  (let ((cur-speed 0))
    (define (accelerate amount)
      (set! cur-speed (min max-speed (+ cur-speed amount)))
      (if (> cur-speed max-speed)
          (set! cur-speed max-speed)))
   (define (decelerate amount)
      (set! cur-speed (max 0 (- cur-speed amount))))
   (define (get-speed)
       cur-speed)
   (define (get-color)
      color)
  (lambda (message)
   (cond ((eq? message 'accelerate) accelerate)
         ((eq? message 'decelerate) decelerate)
        ((eq? message 'speed) get-speed)
        ((eq? message 'color) get-color)
        (else (error "Unknown request"))))))
```

Racket

```
#lang racket
(define vehicle-class%
  (class object%
    (init-field color max-speed)
    (define cur-speed 0)
   (define/public (accelerate amount)
     (set! cur-speed (min max-speed (+ cur-speed amount)))
     (if (> cur-speed max-speed)
        (set! cur-speed max-speed)))
  (define/public (decelerate amount)
    (set! cur-speed (max 0 (- cur-speed amount))))
 (define/public (get-speed) cur-speed)
 (define/public (get-color) color)
 (super-new)))
```

Преимущества использования ООП по сравнению с программированием в рамках других парадигм.

1) Читаемость и многословность: ООП-синтаксис обычно более компактен и более привычен разработчикам, знакомым с ООП-языками

2) Поддержка на уровне языка: наследование, полиморфизм, инкапсуляция

3) Библиотеки, стандартные классы и средства

2) Рекурсивные вычислительные процессы вызывают сами себя, сохраняя контекст каждой вызова. Итеративные процессы воспроизводят состояние вычислений без увеличения контекста вызова – с помощью циклов или хвостовой рекурсии.
Приведём пример рекурсивной функции и итеративной функции, вычисляющих сумму всех нечётных чисел от 1 до n, делящихся на 13.

рекурсивный

```
(define (sum-odd-multiple-of-13-rec n)
  (if (= n 1)
      (if (= (remainder n 13) 0) n 0)
      (if (and (odd? n) (= (remainder n 13) 0))
          (+ n (sum-odd-multiple-of-13-rec (- n 2)))
           (sum-odd-multiple-of-13-rec (- n 1)))
     )
  )
)
```

итеративный

```
(define (sum-odd-multiple-of-13-iter n)
  ~~(define (iter counter~~
  (let iter ((counter n) (sum 0))
    (if (= counter 0)
       sum
      (if (and (odd? counter) (= (remainder counter 13) 0))
        (iter (- counter 2) (+ counter sum))
        (iter (- counter 1) sum)
      )
    )
  ))
```

С прагматической точки зрения итеративный процесс обычно предпочтительнее, потому что он более эффективен по ресурсоёмкости (не накапливает цепочек вызовов) и часто производителен.

3) Макросы в Scheme используются для написания кода, который может изменять или расширять синтаксис языка. Макрос syntax-rules позволяет создавать синтаксические расширения, которые выполняют подмену кода на этапе компиляции.
Пример макроса – синтаксическая конструкция unless:

```
(define-syntax unless
  (syntax-rules ()
   ((unless test body ...)
    (if (not test)
      (begin body ...)))))
```

Использование макросов оправдано, когда необходимо оптимизировать повторяющиеся шаблоны кода, добавить новые синтаксические конструкции или упростить сложные структуры вызовов.

Использование макросов не оправдано, когда:
1) Это приводит к сложному и трудному для понимания коду
2) Задачу можно решить с помощью функций.
Макросы мощные, но могут стать источником сложности и неочевидных ошибок, поэтому их следует использовать осторожно.