Racket Exam Practice Cheat Sheet

Question 1 - Higher-Order Functions

```
(define (myzip x y)
  (map cons x y))

(myzip '(1 2 3) '(a b c))
; => '((1 . a) (2 . b) (3 . c))
```

Key Points:
- Higher-order functions take functions as arguments or return them.
- map applies a function to each element of a list.
- cons pairs elements: (x . y).
- Analogy: pairing shoes from two boxes.

Question 2 - Tail Recursion

```
(define (revt x y)
  (if (null? x) y (revt (rest x) (cons (first x) y))))

(define (myrev x) (revt x null))

(myrev '(1 2 3))
; => '(3 2 1)
```

Key Points:
- Tail recursion: recursive call is last operation.
- Uses an accumulator for efficiency.
- Analogy: moving books from one stack to another.

Question 3 - Substitution Model of Evaluation

```
(define (sumf f n)
  (if (= n 0) 0 (+ (f n) (sumf f (- n 1)))))

(define (nth-power n) (lambda (x) (power x n)))
(define (power a n) (if (= n 0) 1 (* a (power a (- n 1)))))

(sumf (nth-power 3) 2)
; => 9
```

Key Points:
- Substitute function calls with definitions step-by-step.
- Example: 2^3 + 1^3 = 9.
- Dynamic scope would give a different result.

Question 4 - Environment Model of Evaluation

```
(define (make-adder x)
  (lambda (y) (+ x y)))

(define add1 (make-adder 1))
```

```
(define x 2)
(add1 2)
; Lexical: 3, Dynamic: 4
```

Key Points:
- Lexical scoping: look where function was defined.
- Dynamic scoping: look where function is called.
- Environment tracks variable bindings.

Question 5 - Streams

```
(require racket/stream)

(define (stream-enumerate-interval a b)
  (if (> a b) empty-stream
      (stream-cons a (stream-enumerate-interval (+ a 1) b))))

(stream-fold + 0 (stream-map (charfun prime?) (stream-enumerate-interval 2 100)))
```

Key Points:
- Streams are lazy lists: values calculated on demand.
- Useful for infinite sequences or large data.
- Analogy: water faucet - only runs when needed.

Question 6 - Evaluation Orders

```
(define (square x) (* x x))
(define (sum-of-squares x y) (+ (square x) (square y)))
(define (f a) (sum-of-squares (+ a 1) (* a 2)))

(f 5)
; Applicative: 136, Normal: 136, Memoization: 136 (less repeated work)
```

Key Points:
- Applicative (call by value): evaluate arguments first.
- Normal (call by name): delay evaluation, may repeat work.
- Memoization: cache results, avoid recomputation.
- Analogy: prep ingredients first (applicative) vs calculate as you cook (normal).