```
; Warmup what does Scheme display? 15 min (11:55)

 '(print hello)
; expect (print hello)

 `(one ,(+ 1 1) three ,(+ 3 1))
; expect (one 2 three 4)

 (cons 'print (cons 'hello nil))
; expect (print hello)

 (list 'and (+ 3 2) '(- 2 1) #f)
; expect (and 5 (- 2 1) #f)

 (append (list 2) '(3))
; expect (2 3)

 `,(print hi)
; expect Error: unknown identifier: hi

 `,(print '(print hi))
 ; expect (print hi)

 `(define (square ,((lambda () 'x))) (* x x))
; expect (define (square (x) (* x x)))


; Demo
(define-macro (value-of expression with bindings)
      ; (cons 'let (cons bindings (cons expression nil)))
      (list 'let bindings expression)
      ;`(let ,bindings ,expression)
   )
; (value-of (+ y (* x 2)) with ((x 2) (y 3)))
; expect 7

; Q0
; Write a macro that creates a one argument lambda with expr as the body.
; Expr should expect the formal parameter to be always named x.
(define-macro (fl expr)
  `(lambda (x) ,expr)
  ; (cons 'lambda (cons (list 'x) (cons expr nil)))
  ; (list 'lambda (list 'x) expr)
```

```
  )
; ((fl (+ 2 x) 3)
; expect 5


; Q1
; Write a macro that creates a one argument function called name
; with expr as the body. The expression should expect the formal
; parameter to always be named x.
(define-macro (fd name expr)
  (list 'define (list name 'x) expr)
;  `(define (,name x) ,expr)
;   (cons 'define (cons (list name 'x) (cons expr nil)))
  )
; (fd square (* x x))
; (define (square x) (* x x))


; Q2
; Write a macro that takes in a list of subexpressions
; and evaluates them in reverse order.
(define (reverse lst)
    (cond ((null? lst) nil)
          (else (append (reverse (cdr lst)) (list (car lst))))
          )
  )

(define-macro (backwards expressions)
      (cons 'begin (reverse expressions))
      ;`(begin ,(reverse expressions)) <-- Wrong (good demo why)
    )
; (backwards (
;   (print 'third)
;   (print 'second)
;   (print 'first)
;  ))

; expect first
; expect second
; expect third


; Q3
```

```
; Write a macro that takes in a list of condition-value pairs evaluating the
; pair which condition is False. Don't worry about handling an "else".
; Hint don't use a quasi-quote.
(define-macro (inverse-cond conditions)
    (cons
      'cond
      (map
        (lambda (pair) (cons (not (car pair)) (cdr pair)))
;        (lambda (pair) (cons (not-else (car pair)) (cdr pair)))
        conditions)
     )
  )

; (inverse-cond
;   ((#t 'wrong)
;    ('test 'wrong)
;    (#f 'correct!))

; expect correct!

Q4
; Let's now make this work for else
(define (not-else expr)
  (if (eq? expr 'else)
      #t
      (not expr)
   )
  )

; (inverse-cond
;   ((#t 'wrong)
;    ('test 'wrong)
;    (else 'correct!)
;    )
;  )

; expected 'correct!
```