Part 1

1. (apply max (map abs (list -5 10 365 -20)))

number

2. (define x 10)
(local [(define x "Northwestern University")
(define (help x) x)]
(help x))

string

3. (define val true)
(unless val 5)

void or (void)

4. (define z (list "winter" "is" "great"))
(if (= (length z) 3)
(cons "summer" (rest z)))

exception

5. (define r (list "a" "b" "c"))
(define res empty)
(begin (for-each
(lambda (s)
(set! res (cons (string-append s s)
res)))
r)
res)

(listof string)

6. (apply + (map furniture-price
(list my-table my-chair)))

number

7. sell!

furniture, string → void

8. (map (sell! f "ian")
(list my-table my-chair))

exception

9. (chair-color my-chair)

string

10. (begin (set-table-price! my-table 500)
(table-price my-table))

exception

Part 2

Question 1 Score:

Interaction	Desired-output

(define (list-contents pn)
  (cond [(empty? pn) empty]
  [else (append (list-contents (person-node-left pn))
  (person-ssn (person-node-person pn))
  (list-contents (person-node-right pn)))]))
(list-contents ian-node)
	> (list 4 7)
	
	Actual output (exception):
	append: expects a list, given 7

Fix: add a list or a cons to the second append item (list (person-ssn (person-node-person pn)))

```
(define (list-contents pn)
  (cond [(empty? pn)   empty]
        [else (append (list-contents (person-node-left pn))
                      (list (person-ssn (person-node-person pn)))
                      (list-contents (person-node-right pn)))]))
```

OR

```
(define (list-contents pn)
  (cond [(empty? pn)   empty]
        [else (append (list-contents (person-node-left pn))
                      (cons (person-ssn (person-node-person pn))
                      (list-contents (person-node-right pn)))]))
```

Question 2 Score:

Interaction	Desired-output

(define (find-max pn)
  (cond [(empty? pn) 0]
  [else (max (find-max (person-node-left pn))
  (find-max (person-node-right pn)))]))
(find-max ian-node)
	> 7
	
	Actual output: 0

Fix: change (find-max (person-node-left pn)) to (person-ssn (person-node-person pn))
*** if they don't remove the recursion on the left but add the (person-ssn (person-node-person pn)), it's still correct*

```
(define (find-max pn)
  (cond [(empty? pn) 0]
        [else (max ~~(find-max (person-node-left pn))~~
                   (person-ssn (person-node-person pn))
                   (find-max (person-node-right pn)))]))
```

EECS 111 Exam 2 Solutions

Question 3 Score:

Interaction	Desired-output
(define s 0) (map (lambda (x) (set! s (+ s x))) (list 1 2 3 4))	> 10 Actual output: (list (void) (void) (void) (void))

Fix: wrap the map in a begin and add a s to end. They are also welcome to change map to for-each, but that alone won't solve it.

```
(begin (map (lambda (x)
              (set! s (+ s x)))
            (list 1 2 3 4))
       s)
```

ALSO CORRECT:

```
(begin (for-each
          (lambda (x)
            (set! s (+ s x)))
          (list 1 2 3 4))
       s)
```

Question 4 Score:

Interaction	Desired-output
(define (sumlist lst) (local [(define sum 0) (define remaining lst) (define (sum-help) (cond [(empty? remaining) sum] [else (begin (set! remaining (rest remaining)) (set! sum (+ sum (first remaining))) (sum-help))]))] (sum-help))) (sumlist (list 1 2 3))	> 6 Actual output: first: expects a non-empty list; given: '()

Fix: swap order of set! lines

```
(define (sumlist lst)
  (local [(define sum 0)
          (define remaining lst)
          (define (sum-help)
            (cond [(empty? remaining) sum]
                  [else (begin (set! sum (+ sum (first remaining)))
                               (set! remaining (rest remaining))
                               (sum-help))]))]
    (sum-help)))
```

Question 5 Score:

Interaction	Desired-output
(define odds empty) (define lst2 (list 1 2 3 4 5 6 7 8 9)) (begin (for-each (lambda (x) (when (odd? x) (set! odds (cons x odds)))) lst2) odds)	> (list 1 3 5 7 9) Actual output: (list 9 7 5 3 1)

Fix: change (cons x odds) to (append odds (list x)) or call reverse on odds or lst2

```
(begin (for-each (lambda (x)
                   (when (odd? x)
                     (set! odds (append odds (list x))))) lst2)
       odds)

OR

(begin (for-each (lambda (x)
                   (when (odd? x)
                     (set! odds (cons x odds)))) (reverse lst2))
       odds)
OR

(begin (for-each (lambda (x)
                   (when (odd? x)
                     (set! odds (cons x odds)))) lst2)
       (reverse odds))
```