This document is a listing of the functions/macros in Scheme which will be useful in CS 135.

Basic Scheme Commands

- and
- append
- apply
- car
- cadr
- cdr
- cdar
- cond
- cons
- define
- display
- equal?
- if
- lambda
- length
- list
- map
- max
- min
- or
- not
- reverse
- Comparsion Functions (=, >, <, >=, <=)

and

`and` takes a variable number of arguments and returns whether all of them are true.

```
; True and True and True is True
> (and #t #t #t)
#t

; True and False is False
```

```
> (and (= 0 0) (> 7 100))
#f
```

Note: `and` expands into a series of `if` branches when the compiler reads the code. It cannot be used as a function with functions like map, filter and reduce.

append

Takes two or more lists and returns the result of concatenating those lists together.

```
> (append '(1 2 3) '(4 5 6))
'(1 2 3 4 5 6)

> ; Append multiple lists together
> (append '(a b) '(c) '(d e f g) '(h i))
'(a b c d e f g h i)
```

case

`case` is a macro which provides similar/identical functionality to switch/case in C-style languages.

```
> (define (message-for-grade grade)
     (case grade
        ('(A) "Excellent!")
        ('(B) "Not bad! Keep up the good work.")
        ('(C) "There's always a curve.")
        ('(D) "Did you study?")
        ('(F) "Maybe this just isn't your thing")))
> (message-for-grade 'B)
"Not bad! Keep up the good work."
```

cond

`cond` is a nice macro that helps abstract away heavily nested `if` expressions. It's used like how Python's `elif` keyword is used to avoid nested if statements.

```
;; Compare two numbers `x` and `y`, returns:
;;    0  if they are equal
;;    1  if `x` is greater than `y`
;;   -1  if `x` is less than `y`
;;
> (define (compare x y)
    (cond
```

```
      ((> x y) 1)
      ((< x y) -1)
      (else 0)))

;; 5 > 3, so 1
> (compare 5 3)
1
```

cons

```
(cons element list)
```

Add an element to the beginning of a list.

```
> (cons 1 '(2 3))
'(1 2 3)
```

define

```
(define name value)

(define (fn-name args ...)
   ... function body ...)
```

`define` is the way to set variables and to define functions.

```
; Define a variable "a" with the value 10
> (define a 10)
> a
10

; Define a function that returns twice the length of the list "numbers"
> (define (double-len numbers)
     (* 2 (length numbers)))

> (double-length '(a b c d))
8
```

`define`

display

```
(display string)
```

`display` is the equivalent of `print` in Python or `System.out.println` in Java except it does not print the new line by default (you need to add `\n` to the end of the string literal for the new line).

```
;; The "\n" is for the newline
> (display "Hello, World!\n")
Hello, World!
```

equal?

`(equal? x y)`

`equal?` returns whether or not two values are equal or not. It is generally better to use `equal?` than `=` because `=` can only check numeric values, and `equal?` uses more advanced definitions of equality.

```
> (equal? 2 2)
#t
> (equal? '(2 3) '(5 6))
#f

> (= '(1 2 3) '(1 2 3))
#f
> (equal? '(1 2 3) '(1 2 3))
#t
```

if

`(if condition then else)`

Equivalent of an if statement. If the condition is true, evaluates to the `then` branch, otherwise evaluates to the `else` clause

```
> (if (= 2 (/ 4 2))
    (display "They're equal\n")
    (display "They're not equal\n"))
They're equal

; You can nest if expressions
> (define (compare x y)
    (if (> x y)
      1
      (if (< x y)
        -1
        0)))
```

```
;; 10 > 2 ==> 1
> (compare 10 2)
1
```

lambda

```
(lambda (args ...) ... function body ...)
```

`lambda` creates in-line functions. It is especially useful when used with functions like map, filter, and reduce

```
> ; Here we pass a function that adds 2 to a number
> (define (add-2-to-each lst)
    (map (lambda (x) (+ x 2)) lst))

> (add-2-to-each '(1 20 13 401 5 108 71))
'(3 22 15 403 7 110 73)
```

list

```
(list v ...)
```

`list` takes multiple values and returns a linked list containing those

```
> (list 1 'a 2.3 3 4.5e+3)
'(1 a 2.3 3 4.5e+3)
```

map

```
(map proc list)
```

`map` takes a function and a list and returns a new list of the function applied to

```
> ; (abs x) - the absolute value of x
> (map abs '(-1 2 -3 4 5 -6))
'(1 2 3 4 5 6)
```

or

`or` takes zero or more arguments and returns if any of them are true.

```
; False or True is True
> (or (equal? '(1 2) 'a) (> 34 5))
#t

; No arguments means none of them are true.
```

```
> (or)
#f
```

Note: `or` expands into a series of `if` branches when the compiler reads the code. It cannot be used as a function with functions like map, filter and reduce.

Comparison Functions

Note: Regular comparison functions only work on numbers (integers and floats).

Symbol	Sample Usage
=	`(= 1 1)`
>	`(> 100 3)`
>=	`(>= 10 2)`
<	`(< 5 7)`
<=	`(<= 4.5 6.7)`

equal?

See equal?

Functions We Will Define

- filter
- reduce
- Cartesian/Cross Product
- Set functions

filter

```
(filter pred lst)
```

`filter` is a powerful function for processing data. It takes a list and returns a new list only containing

reduce

```
(reduce f init lst)
```

`reduce` is a core function in functional programming. It is useful for aggregation functions (functions that compute based on a collection of values rather than a single value), such as `sum`, `product`, and the union of sets.

```
; For each element in the list, ignore the value and increment the length by 1
> (define (length lst)
    (reduce (lambda (len _) (+ len 1)) 0 lst))

> ; The sum of '(1 2 3 4 5 6 7) is the same as
> ; (+ (+ (+ (+ (+ (+ (+ 0 1) 2) 3) 4) 5) 6) 7)
> (define (sum numbers)
    (reduce + 0 numbers))
```

cross-product

```
(cross-product s t)
```

`cross-product` takes two sets A and B and returns `{(a, b) | a in A, b in B}`

Set functions

element?

```
(element? item list)
```

Takes an item and a list and returns whether or not the element is present in the list

```
> (element? 5 '(1 2 3 4 5))
#t

> (element? 'lieb '(buildings we still have))
#f
```

intersection

```
(intersection s t)
```

Given two sets, returns the set of elements which are in BOTH sets.

```
> (intersection '(1 2 3) '(4 3 2))
'(2 3)
> (intersection '(r i p) '(l i e b))
'(i)
```

```
> (intersection '() '())
'()
```

make-set

`(make-set list)`

Takes a list and removes all duplicate elements.

```
> (make-set '(0 0 0))
'(0)
> (make-set '(1 2 3 2 4 5))
'(1 2 3 4 5)
```

Note: There are multiple ways to implement `make-set` and some of them don't preserve the order of the set (which when working with sets is not necessary)

set-equal?

`(set-equal? s t)`

Determines whether two sets are equivalent (i.e. A = B => every element in A is in B and every element in B is in A)

```
> (set-equal? '() '())
#t

> (set-equal? '(a b c) '(a b c))
#t

> (set-equal? '(1 2 3 4) '(4 3 1 2))
#t

> (set-equal? '(1 2 3) '(1 2 4))
#f
```

subset?

`(subset? s t)`

Returns true if `s` is a subset of `t`. I.e., if every element of `s` is in `t`.

```
> (subset? '() '())
#t
> (subset? '(1 2 3) '(1 2 3 4 5))
#t
> (subset? '(115 284 385) '(115 146 284 135 385 392))
```

```
#t
> (subset? '(-2 0 2) '(-1 1 3))
#f
> (subset? '(-1 1 2) '(-1 1 3 5 7))
#f
```

union

`(union s t)`

Given two sets, return a list representing the set which contains all of the elements from each set EXACTLY once.

```
> (union '(1 2 3) '(4 5 6))
'(1 2 3 4 5 6)

> (union '(1 2 3) '(1 2))
'(1 2)

> (union '(1 1 1) ())
'(1)
```

Predicates

A predicate is a function/procedure/method that returns a boolean (true/false) value based on its inputs. Here is a list of common predicates built into the Racket module eopl that we use as the language for all of our labs.

Predicate	Description
`(boolean? x)`	`x` is a boolean (`#t` or `#f`)
`(even? x)`	`x` is an even number
`(integer? x)`	`x` is an integer (whole number)
`(list? x)`	`x` is a list
`(negative? x)`	`x` is negative (`x < 0`)
`(null? x)`	`x == null` OR `x == '()`
`(number? x)`	`x` is a number
`(odd? x)`	`x` is an odd number
`(positive? x)`	`x` is positive (`x > 0`)
`(string? x)`	`x` is a string
`(symbol? x)`	`x` is a symbol
`(zero? x)`	`x == 0`

For more information on any of the commands built-in to EOPL, see

https://docs.racket-lang.org/eopl/