Scheme Quick Reference Guide

Data types

- **numbers**
  - `42`
  - `2/3`
  - `-6`
  - `3.14159`
  - `3+2i`
- **symbols**
  - `x`
  - `apple`
  - `a-rather-long-symbol-name`
  - `greater-than-42?`
- **booleans**
  - `#t`
  - `#f`
- **strings**
  - `"Hello, world!"`
- **lists**
  - `(a b c)`
  - `(1 2 3 4)`
  - `((a b c) apple (#t (7 -2.5 pi)))`
  - `()`
- **vectors**
  - `#(a b c)`
  - `#(1 2 three 3.14)`
  - `#(#(a b c) #(1 2 3))`

Numeric data manipulation

- **`+ - * / sqrt random`**
  - `(+ 2 3)` evaluates to `5`
  - `(/ 2 3)` evaluates to `2/3`
  - `(/ 2.0 3.0)` evaluates to `0.6666666666666`
  - `(* (+ 2 3) 10)` evaluates to `50`
  - `(sqrt 2)` evaluates to `1.4142135623730951`
  - `(random 100)` may give `0, 16, 99, 42,` etc. (but never `100`)

Symbolic data manipulation

- **`quote`**
  - `(quote apple)` or `'apple` evaluates to `apple`
  - `(quote (a b c))` or `'(a b c)` evaluates to `(a b c)`
  - `(quote (+ 2 3))` or `'(+ 2 3)` evaluates to `(+ 2 3)`

- **cons car cdr list**
  - `(list 'a 'b 3)` evaluates to `(a b 3)`
  - `(list (list 1 2 3) 'x)` evaluates to `((1 2 3) x)`
  - `(cons 'x (list 'y 'z))` evaluates to `(x y z)`
  - `(cons 'x '(y z))` evaluates to `(x y z)`
  - `(cons 'x '())` evaluates to `(x)`
  - `(car '(a b c d))` evaluates to `a`
  - `(cdr '(a b c d))` evaluates to `(b c d)`
  - `(cdr '((a b) c d))` evaluates to `(c d)`

Predicate functions

- **number? symbol? boolean? null?**
  - `(number? 3)` evaluates to `#t`
  - `(symbol? 'apple)` evaluates to `#t`
  - `(boolean? #f)` evaluates to `#t`
  - `(null? '())` evaluates to `#t`
- **equal?**
  - `(equal? '(a b c) '(a b c))` evaluates to `#t`
  - `(equal? 'x 4)` evaluates to `#f`
  - `(equal? 'apple 'apple)` evaluates to `#t`
- **= < > <= >=**
  - `(= 2 3)` evaluates to `#f`
  - `(< 2 3)` evaluates to `#t`

Logical operators

- **and or not**
  - `(and (= 2 2) (> 5 1))` gives `#t`
  - `(or (= 2 3) (number? 'apple) (symbol? 3))` gives `#f`
  - `(and (not (= 1 2)) (not (null? '(a b c))))` gives `#t`

Input/output

- **display newline read printf**
  - `(display "Hello, world!")` prints out `Hello, world!` as a side-effect, without a carriage return, and evaluates to an undefined "void" value
  - `(newline)` generates a carriage return as a side-effect and evaluates to an undefined "void" value
  - `(read)` waits for the user to enter a *complete* Scheme expression (symbol, number, list, boolean, etc.) at the keyboard, then returns that value
  - `(printf "The value ~a is bound to x~%" x)` prints out
    `The value 10 is bound to x`
    as a side-effect, assuming that `x`'s value is 10, and evaluates to an undefined "void" value. The `~%` code generates a carriage return.

Conditionals

- **(if** ***test-expr then-exp else-exp*****)**
  - `(if (= 2 3) 'yes 'nope)` evaluates to `nope`
  - `(* 10 (if (number? 5) 2 'foo))` evaluates to `20`
  - `(if (if (= 5 5) #f #t) 'yes 'nope)` evaluates to `nope`

- **(cond**
  (***test-expr1 then-exp1*****)**
  (***test-expr2 then-exp2*****)**
  ...
  (***test-exprN then-expN*****)**
  (else ***else-exp*****))**

  - ```
    (cond
      ((= 2 3) 'yes)
      (else 'nope))
    ```
    evaluates to `nope`

  - ```
    (* 10
       (cond
         ((number? 5) 2)
         (else 'foo)))
    ```
    evaluates to `20`

  - ```
    (cond
      ((cond
         ((= 5 5) #f)
         (else #t))
       'yes)
      (else 'nope))
    ```
    evaluates to `nope`

Local variables

- **(let ((var1** ***expression1*****)**
  (var2 ***expression2*****)**
  ...
  (varN ***expressionN*****))**
  body-expr**)**

  - ```
    (let ((x (+ 2 3))
          (y (* 5 2)))
      (list x y))
    ```
    evaluates to `(5 10)`

  - ```
    (let ((a (* 10 (+ 2 3))))
      (+ a a))
    ```
    evaluates to `100`

  - ```
    (let ((x 100))
      (let ((y (* x 2)))
        (+ x y)))
    ```
    evaluates to `300`

Defining functions

- **(define** ***name***
  (lambda (*param1 param2 ...***)**
  body-expr**))**

  - `(define fun`
    `(lambda (a b)`
    `(* 10 (+ a b))))`

  - `(define factorial`
    `(lambda (n)`
    `(if (= n 0)`
    `1`
    `(* n (factorial (- n 1))))))`

  - `(fun 1 2)` evaluates to `30`
  - `(factorial 4)` evaluates to `24`

Local recursive functions

- **(letrec ((fun1** ***lambda-expression1*****)**
  (fun2 ***lambda-expression2*****)**
  ...
  (funN ***lambda-expressionN*****))**
  body-expr**)**

  - `(letrec ((odd?`
    `(lambda (n)`
    `(if (= n 0) #f (even? (- n 1)))))`
    `(even?`
    `(lambda (n)`
    `(if (= n 0) #t (odd? (- n 1))))))`
    `(odd? 42))`

    evaluates to `#f`