CS 4700 Handouts - Scheme Quick Reference Sheet

1 Types

Scheme has literals of several primitive types. The literals are *self-evaluating* expressions, which means that they evaluate to the value that they represent.

```
3                   ; integer literal
-3                  ; integer literal
4.3                 ; floating point literal
4.65e2              ; floating point literal, scientific notation
#t                  ; boolean literal, true
#f                  ; boolean literal, false
"hello"             ; string literal
"hello\n"           ; string literal, note the newline
"I said \"hello\"" ; other backslash characters similar to C
```

Scheme has one composite type, a list. Lists (and strings) are allocated in the heap and automatically garbage collected, just like in Perl. So don't worry too much about the cost of creating new lists, use them freely, they are the primary data structure in Scheme. Note the 'quote' symbol in front of each list, it is actually a function call (see below).

```
'(3 4)              ; a two element list
'((4 3) 3)          ; lists can be nested
'()                 ; empty list
'(())               ; a one element list, the element is the empty list
'("hi" 43 ("joe"))  ; lists can be heterogeneous
```

Note that the list elements are **not** separated by commas. Type checking in Scheme is dynamic.

2 Built-in Functions

A function is a list that is evaluated. In general, a list will be evaluated, only in some contexts are lists **not** evaluated, such as when it is preceded by a 'quote'. Each function is in prefix notation.

(*function-name operand-1 operand-2 ... operand-N*)

Each function returns a value, and takes zero or more operands.

2.1 Arithmetic Functions

```
(max 34 5 7 38 6)  ==> 38         ; maximum of the operands
(min 3 5 5 330 4 -24)  ==> -24    ; minimum of the operands
(+ 3 4)  ==> 7                    ; addition
(* 4 4)  ==> 16                   ; multiplication
(- 3 4)  ==> -1                   ; subtraction
```

```
(- 3)  ==> -3                      ; unary subtraction
(magnitude -7)  ==> 7              ; absolute value
(magnitude 7)  ==> 7              ; absolute value
(quotient 35 7)  ==> 5            ; div
(quotient -35 7)  ==> -5          ; div
(quotient 35 -7)  ==> -5          ; div
(quotient -35 -7)  ==> 5          ; div
(modulo 13 4)  ==> 1             ; mod
(modulo -13 4)  ==> 3            ; mod
(remainder 13 4)  ==> 1          ; remainder
(remainder -13 4)  ==> -1        ; remainder
(gcd 32 4)  ==> 4               ; greatest common divisor
```

2.2   Comparison Functions

```
(= 2 3) ==> #f                     ; are all the integer operands equal?
(eqv? "abs" "abs") ==> #t          ; are all the operands equal?
(eqv? '(a) '(a)) ==> #t            ; are all the operands equal?
(< 2 3) ==> #t                     ; less than
(> 2 3) ==> #f                     ; greater than
(>= 2 3) ==> #f                    ; greater than or equal to
(or #f #t) ==> #t                  ; or
(not #f) ==> #t                    ; negation
(and #f #t) ==> #f                 ; and
(integer? 2) ==> #t                ; are all the operands integers?
(real? 3) ==> #f                   ; all reals?
(string? 3) ==> #f                 ; all strings?
(zero? 3 0) ==> #f                 ; all zero?
(positive? 3 4 5) ==> #t           ; all positive?
(negative? -3 4 -5) ==> #f         ; all negative?
(string= "2" "3") ==> #f           ; are two strings equal?
```

2.3   List Functions

These are by far the most important functions that we will be using.

```
(quote (a b))  ==> (a b)           ; quote suspends evaluation of the list
'(a b) ==> (a b)                   ; quote suspends evaluation of the list
(car '(a b)) ==> a                 ; get head of list (contents of address reg)
(car '()) ==> ERROR: car: Wrong type in arg1 ()  ; expects a list!
(car '((a) b)) ==> (a)             ; get head of list, could be a list!
(cdr '(a b)) ==> (b)               ; get rest of list (contents of address reg)
(cdr '()) ==> ERROR: car: Wrong type in arg1 ()  ; expects a list!
(car '((a) a (b))) ==> (a (b))     ; get rest of list
(cons 3 '())  ==> (3)              ; put element on head of list
(cons '(a) '(b c d))  ==> ((a) b c d)
(cons "a" '(b c))  ==> ("a" b c)
(list a 7 c)  ==> (a 7 c)          ; build a list from the operands
(list)  ==> ()                     ; build a list from the operands
(length '(a b c))  ==> 3           ; what is the length of a list?
(length '(a (b) (c d e)))  ==> 3   ; what is the length of a list?
(length '())  ==> 0                ; what is the length of a list?
```

```
(append '(x) '(y))  ==> (x y)            ; append one list to another
(append '(a) '(b c d))  ==> (a b c d)    ; append one list to another
(append '(a (b)) '((c)))  ==> (a (b) (c)) ; append one list to another
(reverse '(a b c))  ==> (c b a)          ; reverse a list
(reverse '(a (b c) d (e (f))))  ==> ((e (f)) d (b c) a) ; reverse a list
(null? '()) ==> #t               ; is the list empty?
(null? '(2)) ==> #f              ; is the list empty?
```

2.4   Miscellaneous Functions

While Scheme has several loop functions, only the following two functions are often useful.

```
(if (= 2 3) #t #f) ==> #f        ; an if function, returns result of evaluating
                                 ; one of the branches
(if (= 2 3) (+ 2 3) (- 2 3)) ==> -1
(if (>= 3 2)                     ; if functions can be nested
  (if (= 2 3) 'equal 'gt)
  'lt
)
(display "hello")                ; prints hello
(display "hello\n")              ; prints hello followed by a new-line
(display '(1 3))                 ; prints (1 3)
(exit)                           ; leave the scheme interpreter
```

3   Binding Names to Functions or Values

In Scheme each *name* is *bound* to a *meaning*. The name is a sequence of letters but could include numbers or special characters such as a question mark or hyphen. The meaning is usually either a function or literal. The binding between a name and a function is dynamic, so the meaning of a name can change during the lifetime of a program. The *define* function binds a name to a meaning. There are two flavours of *define*. The first binds a name to a value.

```
(define x 23)            ; bind name x to literal 23
x ==> 23                 ; dereference x
(+ x 4) ==> 27           ; names are dereferenced where-ever they appear
(x) ==> ERROR            ; 23 is not a function-name!
(define x '())           ; bind name x to the empty list
(append x x) ==> ()
(define y "hello")       ; bind name y to a string literal
(define y (+ 4 5))       ; bind name y to the result of evaluating
                         ; the function (+ 4 5), i.e., to 9
(define y '(+ 4 5))      ; bind name y to the list (+ 4 5)
(define y (+ 4 x))       ; bind name y to the result of (+ 4 x)
                         ; will result in an ERROR since x means ()
```

The second flavour of *define* binds a name to a function, and is how we will define new functions. In this form of define, enclose the name being defined in brackets. Add parameter names as necessary within the brackets. The scope of each parameter is local to the function definition.

```
(define (x) 23)          ; bind function-name x to evaluate to 23
(x) ==> 23               ; call the function named x
x ==> #<CLOSURE () 23>   ; dereference x
(define (x) (+ 4 5))     ; bind function-name x to the function (+ 4 5)
```

3

```
(x) ==> 9                       ; call the function named x
(define (add x y) (+ x y))      ; define a binary add function
(add 4 5) ==> 9                 ; call the add function, add 4 to 5 results in 9
(define (add x y z) (+ x (+ y z)))  ; define a ternary add function
(add 4 5 7) ==> 16              ; call the add function
```

Note that both operands could be lists, but neither is "evaluated." So this is another context, like within a quote function, in which lists are not evaluated as functions. The second flavour of bind is actually just syntactic sugar for binding the name to an *anonymous function*. An anonymous function is a function that does not have a name. The *lambda* function can be used to create an anonymous function.

```
(lambda (x) (+ x 1)) ==> #<CLOSURE (x) (+ x 1)>  ; create a function that
                                                 ; adds 1 to its one operand

((lambda (x) (+ x 1)) 4) ==> 5      ; use the anonymous function
(define succ (lambda (x) (+ x 1))) ; bind the name succ
(succ 4) ==> 5                      ; dereference succ to get function
(define (succ x) (+ x 1))          ; bind the name succ
(succ 4) ==> 5                      ; dereference succ to get function
```

So in general the second flavour of bind is just syntactic sugar for the first flavour of bind. Always remember that names are usually dereferenced whenever they are used. An exception is when the name is bound in a define function.

3.1 Important Function Details

Here are some important things to remember about functions.

- In general, functions have *eager* evaluation, which means that the parameters are evaluated prior to invoking the function call; however, there are exceptions. For instance the *if* and *cond* functions have *lazy* evaluation, which means that the function is called and the operands are evaluated as needed.

- Functions are first class values. They can be passed as parameters and returned from functions.

- Parameter passing is by positional correspondence. If the wrong number of arguments are passed, an error will be generated.

4