UNIVERSITY OF ALBERTA
AUGUSTANA CAMPUS

Augustana Computing Science 370

Scheme Summary

What You Are Allowed To Use

No other pre-defined functions may be used, though you may write your own functions.

1. Data types: atom and list

2. Special symbols (not case sensitive in our version (R5RS), but is in R6RS):
   a. Boolean: **`#t`** (else) and **#f**
   b. Characters: **`#\a`**, **`#\b`** … **`#\Z`**
   c. Strings: in double quotes

3. Basic functions:
   a. **`quote`**
   b. **`car`**
   c. **`cdr`**
   d. **`c _ _ _ _ r`**, where each _ is either "a" or "d"
   e. **`cons`**
   f. **`cond`**
   g. **`list`**
   h. **`append`**
   i. **`length`**
   j. **`reverse`**
   k. **`member`**
   l. **`map`**

4. Boolean functions:
   a. **`boolean?`** — #t or #f
   b. **`pair?`** — '(a b c) and '(a.b), but not '()
   c. **`list?`** — '(a b c) and '(), but not '(a.b)
   d. atom? – not defined in DrRacket's Scheme, but you could define it as:
      ```
      (define (atom? x)
        (not (pair? x)))
      ```
      Assuming the empty list is both a list and an atom.
   e. **`symbol?`**
   f. **`number?`**
   g. **`char?`** — literals written with #\ prefix, followed by the character, Unicode code, or special character descriptor (#\tab, #\linefeed, #\newline, #\space, etc.)
   h. **`string?`** — sequence of characters enclosed in double quotes (e.g., "Hello\n")
   i. **`null?`**
   j. **`eq?`**

k. **equal?**
l. **and**, **or**, **not**
m. **char=?**
n. **string=?**
o. **negative?**

5. Arithmetic functions
   a. **+**, **-**
   b. *****, **/**, mod (DrRacket's Scheme uses "**modulo**" for mod)
   c. **=**, **<**, **>**, **<=**, **>=**
   d. **random** (DrRacket's Scheme requires an import. Put the following line at the top of your file:

      ```
      (#%require (only racket/base random))
      ```
   e. **min**, **max**
   f. **sqrt**, **exp**, **log**, **abs**
   g. **exact->inexact**
   h. **inexact->exact**

6. Definitions — for data and functions, to associate a name with a value. You may only use them for functions in your assignment; you may use them on data for testing purposes.
   a. E.g. (**define** (functionName formalParams) body)
   b. E.g. (**define** functionName (lambda (formalParams) body))
   c. E.g. (define dataName 2) do NOT use this in your code, just for testing

   Note that assignments, such as (set! variable expression), should not be used in your formal program, but may prove valuable in testing. Assignments break the functional style.

7. I/O stuff:
   a. **symbol->string**
   b. **string->symbol**
   c. **string->list**
   d. **list->string**
   e. **char->integer**
   f. **integer->char**
   g. **read** – returns an atom
   h. **read-char**
   i. **peek-char**
   j. **display**
   k. **newline**

8. Special functions:
   a. **apply**
   b. eval – does not work under DrRacket's Scheme
      [You can make it work is a weird way by including a 2nd parameter:

e.g. > (eval '(+ 1 2) (scheme-report-environment 5))
3

But I recommend that you know what eval does, but do not use it in your programs.]
Note that: (apply + ' (3 4 5)) is the same as (eval (cons '+ ' (3 4 5))) in normal Scheme. In Dr.Racket's Scheme eval does NOT work without a 2nd parameter.

I wrote the following that you may use if you want a readLine function, which will give you input as a string, versus as an atom. Feel free to include them into your program, just attribute them to me (Rosanna Heise).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; readLine() --> line (as String)
;;
;; Read one line from standard input, not including the newline
;; but eliminating it.  This is wrapper for the recursive method
;; that does the work (readLoop).
;;
(define (readLine)
    (readLoop (read-char (current-input-port)) '())) ;do wait for one char

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; readLoop(currentCharacter line) --> line (as String)
;;
;; This recursive method reads a character at a time from the
;; current input port (assuming Scheme's "Interaction Window")
;; until it finds the newline (i.e. enter).  It builds the characters
;; into a string which is returned at the end.  Newline is not part
;; of the string, but is eliminated from the input
;;
(define (readLoop curChar line)
  (cond
    ((char=? #\newline curChar) (list->string line))
    (#t (readLoop (read-char (current-input-port))
                  (append line (list curChar))))))
```