Dr. Jozo J. Dujmović

SCHEME REFERENCE CARD

ARITHMETIC OPERATORS AND FUNCTIONS

(+ <expr1> <expr2> ...)	<expr1> + <expr2> + ...	
(- <expr1> <expr2> ...)	<expr1> - <expr2> - ...	
(add1 <expr>)	<expr> + 1	[M-]
(1+ <expr>)	<expr> + 1	
(sub1 <expr>)	<expr> - 1	[M-]
(-1+ <expr>)	<expr> - 1	
(minus <expr>)	Change the sign of <expr>	[M-]
(* <expr1> <expr2> ...)	<expr1> * <expr2> * ...	
(/ <expr1> <expr2> ...)	<expr1> / <expr2> / ...	
(expt <expr1> <expr2>)	Raises <expr1> to the power <expr2>	
(float <expr>)	Convert <expr> to floating point	
pi	A variable initially set to 3.1415927	[M-]
(quotient <expr1> <expr2>)	Integer division <expr1> / <expr2>	
(remainder <exp1> <exp2>)	(sign(<exp1>))(fraction(<expr1>/<expr2>)	
(ceiling <expr>)	Smallest integer >= <expr>	
(floor <expr>)	Larger integer <= <expr>	
(round <expr>)	Nearest integer of <expr>	
(truncate <expression>)	Integer part of <expression>	
(modulo <expression1> <expression2>)	<expression1> modulo <expresson2>	
(gcd <expr1> <expr2>)	The greatest common divisor	
(lcm <expr1> <expr2>)	The least common multiple	

LIBRARY FUNCTIONS

(abs <expr>)	Absolute value of <expr>
(sqrt <expr>)	Square root of nonnegative <expr>
(max <expr1> <expr2> ...)	Maximum of <expr1>, <expr2>, ...
(min <expr1> <expr2> ...)	Minimum of <expr1>, <expr2>, ...
(exp <expr>)	The exponential of <expr> (base = e)
(log <expr>)	Natural logarithm of <expr>
(sin <expr>)	The trigonometric sine of <expr> in radians
(cos <expr>)	The trigonometric cosine of <expr> in radians
(tan <expr>)	The trigonometric tangent of <expr> in radians
(asin <expr>)	The arc sine of <expr>
(acos <expr>)	The arc cosine of <expr>
(atan <expr>)	The arc tangent of <expr>

RELATIONAL OPERATORS

(= <expr1> <expr2>)	True if <expr1> = <expr2> (numerical expr.)
(<> <expr1> <expr2>)	True if <expr1> <> <expr2>
(< <expr1> <expr2>)	True if <expr1> < <expr2>
(<= <expr1> <expr2>)	True if <expr1> <= <expr2>
(> <expr1> <expr2>)	True if <expr1> > <expr2>
(>= <expr1> <expr2>)	True if <expr1> >= <expr2>

RECOGNIZERS

(number? <obj>)	True if <obj> is a number	
(boolean? <obj>)	True if <obj> is Boolean	
(char? <obj>)	True if <obj> is a character	
(string? <obj>)	True if <obj> is a string	
(symbol? <obj>)	True if <obj> is a symbol	
(list? <obj>)	True if <obj> is a list	[T-]
(integer? <obj>)	True if <obj> is an integer	
(float? <obj>)	True if <obj> is floating point	
(real? <obj>)	True if <obj> is a real number	
(rational? <obj>)	True if <obj> is a rational number	[MT-]
(atom? <obj>)	True if <obj> is an atom	[M-]
(pair? <obj>)	True if <obj> is a dotted pair	
(null? <obj>)	True if <obj> is an empty list	
(eof-object? <obj>)	True if <obj> is end-of-file marker	
(vector? <obj>)	True if <obj> is a vector	
(procedure? <obj>)	True if <obj> is a procedure (function)	

TESTS

(zero? <expr>)	True if <expr> = 0
(positive? <expr>)	True if <expr> > 0
(negative? <expr>)	True if <expr> < 0
(even? <integer>)	True if <integer> is even
(odd? <integer>)	True if <integer> is odd

EQUIVALENCE PREDICATES

(eq? <obj1> <obj2>)	True if <obj1> is pointer-identical to <obj2>, i.e. bound to the same memory location (this test of physical sameness can be implementation-dependent)
(equal? <obj1> <obj2>)	True if <obj1> and <obj2> have the same type and value (this test causes evaluation of expr.)
(eqv? <obj1> <obj2>)	True if <obj1> and <obj2> are equal numeric values (same or dif. type), strings, and other atoms

LOGICAL VALUES AND OPERATORS

#t #T t T		True
( ) #f #F NIL		False
TRUE		A variable that some Schemes initially set to #T
FALSE		A variable that some Schemes initially set to #F
(not <expr>)		Negation of <expr>; (procedure? not) yields #T
(and <expr> ...)	[(and) yields #T]	Short-circuit conjunction of any # of <expr>...
(or <expr> ...)	[(or) yields #F]	Short-circuit disjunction of any # of <expr>...
		(**and** and **or** don't have the procedure status [recognized by procedure?])

CHARACTERS

`#\a #\B #\newline #\space #\tab #\TAB`	Characters (printable and non printable)
`(char->integer <char>)`	Decimal ASCII value of <char>
`(integer->char <integer>)`	Return a character corresponding to the ASCII code <integer>
`(char->upcase <char>)`	Return the upper case version of <char>
`(char->downcase <char>)`	Return the lower case version of <char>
`(char-alphabetic? <char>)`	True if <char> is an alphabetic character
`(char-numeric? <char>)`	True if <char> is a numeric character
`(char-whitespace? <char>)`	True if <char> is space, newline, tab, page, return
`(char-upper-case? <char>)`	True if <char> belongs to {A, B, ..., Z}
`(char-lower-case? <char>)`	True if <char> belongs to {a, b, ..., z}
`(char=? <char1> <char2>)`	True if <char1> = <char2> (case sensitive)
`(char<? <char1> <char2>)`	True if <char1> < <char2> (case sensitive)
`(char<=? <char1> <char2>)`	True if <char1> <= <char2> (case sensitive)
`(char>? <char1> <char2>)`	True if <char1> > <char2> (case sensitive)
`(char>=? <char1> <char2>)`	True if <char1> >= <char2> (case sensitive)
`(char-ci=? <char1> <char2>)`	True if <char1> = <char2> (case insensitive)
`(char-ci<? <char1> <char2>)`	True if <char1> < <char2> (case insensitive)
`(char-ci<=? <char1> <char2>)`	True if <char1> <= <char2> (case insensitive)
`(char-ci>? <char1> <char2>)`	True if <char1> > <char2> (case insensitive)
`(char-ci>=? <char1> <char2>)`	True if <char1> >= <char2> (case insensitive)

STRINGS

`"<character string>"`	A string of characters
`(make-string <length> <char>)`	String containing <length> copies of <char>
`(string-length <string>)`	Length of string (number of elements)
`(string-append <string1> <string2>...)`	Append all strings in the sequence
`(string-copy <string>)`	A new copy of an existing <string>
`(define <var> (string-copy <string>))`	<var> is bound to a new copy of <string>
`(define <variable> <var>)`	<variable> and <var> are synonyms that point to the same data object <string>
`(string-fill! <string> <char>)`	Fills an existing <string> with <char>'s
`(string-set! <string> <index> <char>)`	string[index]:=char; 0 <= index <= length-1
`(string-ref <string> <index>)`	Returns string[index]; 0<=index<=length-1
`(substring <string> <start> <end>)`	Extracts elements string[start]..string[end-1]; the length of substring = end - start
`(string->symbol <string>)`	Create a symbol whose name is <string>
`(symbol->string <symbol>)`	Create a string whose value is "<symbol>"
`(string->list <string>)`	Convert <string> to list of characters
`(list->string <list>)`	Convert a character <list> to string
`(string->number <string>)`	ASCII to decimal number conv. of <string> [T-]
`(number->string <number>)`	Dec. number to ASCII string conversion [T-]
`(string->number <string> <base-of-number>)`	ASCII string to number conversion
`(number->string <number> <base-of-number>`	Number to ASCII string conversion
`(string-null? <string>)`	True if <string> is an empty string
`(string=? <string1> <string2>)`	True if <string1> = <string2> (case sensitive)

(string<? <string1> <string2>)	True if <string1> < <string2> (case sensitive)
(string<=? <string1> <string2>)	True if <string1> <= <string2> (case sensitive)
(string>? <string1> <string2>)	True if <string1> >= <string2> (case sensitive)
(string-ci=? <string1> <string2>)	True if <string1> = <string2> (case insensitive)
(string-ci<? <string1> <string2>)	True if <string1> < <string2> (case insensitive)
(string-ci<=? <string1> <string2>)	True if <string1> <= <string2> (case insensitive)
(string-ci>? <string1> <string2>)	True if <string1> > <string2> (case insensitive)
(string-ci>=? <string1> <string2>)	True if <string1> >= <string2> (case insensitive)

PAIRS AND LISTS

quote <obj> abbreviated: '<obj>	Return the given <obj> unevaluated
(car <pair>)	Return the first element of pair (or the head of list)
(cdr <pair>)	Return the second element of pair or the tail of list
'( )	Empty list; both (car '()) and (cdr '()) return '() [M-]
(cxxxxr <list>)	Combination of up to 4 **car** and **cdr** (x = {a | d})
(cons <obj1> <obj2>)	Make a dotted pair (<obj1> . <obj2>)
(list <obj1> <obj2> ...)	Make a list of objects
'(<obj1> . (<obj2> . (<obj3> . ())))	Proper list '(<obj1> <obj2> <obj3>)
'(<obj1> . (<obj2> . <obj3>))	Improper list '(<obj1> <obj2> . <obj3>)
(length <list>)	Length of a list. (length <non-list>) => 0.
(append <list1> <list2> ...)	Append lists
(reverse <list>)	Return a list with reversed order of elements
(list-ref <list> <index>)	Returns the zero-based element <index> from the <list>, assuming 0 <= index <= length-1. If index<0 it returns the car element, and if index>=length it returns ().
(member <elem> <list>)	Returns a tail-sublist from <elem> to the end of list, or () if <elem> is not in the <list>.
(set-car! <list> <expr>)	Replaces the head of <list> with the value of <expr> (the returned value is implementation-dep.)
(set-cdr! <list> <expr>)	Replaces the tail of <list> with the value of <expr> (the value of <expr> is usually a list)

VECTORS

(vector <elem1> <elem2> ...)	Create a vector containing given elements
(define v '#(<elem1> <elem2> ...))	Definition of vector (' is sometimes omitted)
(make-vector <length>)	Vector having <length> unspecified elements
(make-vector <length> <elem>)	Vector containing <length> copies of <elem>
(vector-length <vector>)	Returns the length of <vector>
(vector-ref <vector> <index>)	Returns vector[index]; 0 <= index <= length-1
(vector-set! <vector> <index> <value>)	vector[index] := value
(vector->list <vector>)	Vector to list conversion
(list->vector <list>)	List to vector conversion

DEFINITIONS AND BINDINGS

(define <variable>)	Define <variable> without binding.
(define <variable> <expr>)	Define <variable> and assign it the value of <expr>. The returned value is the symbol

	<variable> (e.g. (symbol? (define a 1)) returns #T)
(set! <variable> <expression>)	<variable> := <expression> ; <variable> must be previously defined. The returned value is unspecified (some implementations, including **pcs**, return the value of <expression>).
(let ((<var1> <expr1>) ...) <expression1> ... **)**	Local bindings var1:=expr1,... and then the application of these values to the expression1, expression2 ... (the returned value is the value of the last expression)
(let* ((<var1> <expr1>) ...) <expression1> ... **)**	A version of **let** where variables are bound in sequence from the first to the last (different from **let**)
(letrec ((<var1> <expr1>) ...) <expression1> ... **)**	A version of **let** allowing for mutual recursion

FUNCTIONS

(lambda (<arg>...) <expr> ...**)**	Anonymous function definition (lambda expression)
(define <fun-name> <lambda expr>)	Function definition (binding fname and lambda expr.)
(define (<fun-name> <arg>...) <expr>...**)**	Abbreviated form of function definition.
((lambda (<var>...) <body>) <expr>...)	Equivalent to (let ((<var> <expr>)...) <body>) <var> are local varibles
(define <fname> (lambda <atom> <expr>...))	Function of an arbitrary number of arg's
(define(<fname>.<atom>)<expr>...)	Abbreviated form of the above func. of any # of arg.
(<fname> <arg1> <arg2> ...)	<atom> is bound to the list (<arg1> <arg2>...)
(map <function> <list>)	Apply <function> sequentially to each element of the <list> and return the list of resulting values
(andmap <function> <list>)	Application of **and** to the results of **map** [MT-]
(for-each <function> <list>)	Left-to-right application of function to list elements for side effect(s) only (the returned value is implementation-specific)
(apply <function> <list>)	Exec. <function> using list elements as arguments
(eval <expr>)	Evaluation of <expr>

CONTROL STRUCTURES

(**begin** <expr1> <expr2> ...)	Evaluate all expressions and return the value of the last expression in the sequence.
(**begin0** <expr1> <expr2> ...)	Evaluate all expressions and return the value of <expr1>
(**if** <condition> <expr1> <expr2>)	Conditional execution: it returns the value of <expr1> in cases where <condition>=#T. If <condition>=#F then the returned value is <expr2>; in cases where <expr2> is not specified the result of **if** expression is unspecified (some implementations, including pcs, return **( )**).
(**cond**	
(<test1> <expr11> <expr12> ...)	Conditional execution: evaluate tests and execute
(<test2> <expr21> <expr22> ...)	either the sequence of expressions following the first
. . . . .	that is nonnull, or the expressions in the last line.
(<testN> <exprN1> <exprN2> ...)	It returns the value of the last executed expression.

(**else** <expr1> <expr2> ...))	Else clause can be omitted (but in such a case the result of **cond** for all false tests is unspecified.)
(**case** <expr> (<atom1> <expr11> <expr12> ...) (<atom2> <expr21> <expr22> ...) . . . . . . (<atomN> <exprN1> <exprN2> ...) (**else** <expr1> <expr2> ...))	Conditional execution: evaluate <expr> and compare its value with <atom1>, <atom2>, ... until the eqv? comparison returns true. Evaluate all expressions in the selected list and return the value of the last executed expression.
(**case** <expr> (<atom-list1> <expr11> <expr12> ...) (<atom-list2> <expr21> <expr22> ...) . . . . . . (<atom-listN> <exprN1> <exprN2> ...) (**else** <expr1> <expr2> ...))	Conditional execution: evaluate <expr> and compare its value with <atom-list1>, <atom-list2>, ... until the memv? comparison returns true. Evaluate all expressions in the selected list and return the value of the last executed expression.
(**do** ((<variable> <initial-value> <update>) ...) (<termination-test> <expression> ...) <statement> ...)	Initial values are bound to corresponding variables in unspecified order. If the termination test fails, the sequence of statements (the body of the loop) is executed. After the execution the *do variables* are updated in unspecified order. The update is optional (can be omitted in some cases). If the termination-test is true then the termination expressions are evaluated from first to last and the last value is returned (if expressions are not specified then the returned value of do loop is the value of the <termination-test>).
(exit)	Close the transcript file and return to the level of the operating system.

INPUT AND OUTPUT

(transcript-on "<filename>")	Open for append (create if nonexistent) and begin echoing the terminal interaction to <filename>.
(transcript-off)	Close the transcript file
(load "<filename>")	Read and evaluate the contents of <filename>.
(read)	*Read* operator returns a scanned value from the keyboard.
(display <expr>)	Display the value of a single expression (in the case of **pcs** the cursor remains in the same line).
the-non-printing-object	Same as (display "")
(define inport (open-input-file "fname1"))	Definition of input and output ports associated with
(define outport (open-output-file "fname2"))	files named "fname1" and "fname2" respectively
(read inport)	Scan and return next lexical element from input port
(display <expr> outport)	Append the output string to file defined by outport
(close-input-port <port name>)	Close input file related to <port name>
(close-output-port <port name>)	Close output file related to <port name>
(newline)	New line (LF+CR)
(writeln <expr> ...)	Print a sequence of expressions using **display** followed by a **newline** [M-]

```
(writeln (eval (read)))              Read-evaluate-print sequence
(display (eval (read)))              Read-evaluate-print sequence
(write <expr>)                       Write expression in machine-readable form, e.g. strings
                                     have double quotes (display and writeln use the human-
                                     readable form)
(pp <procedure>)                     Pretty print the specified procedure (in the case of
                                     pcs it must be preceded by setting the debug mode)

(define (WL lst)                     Display of list elements (in cases without writeln)
  (cond ((null? lst) (display ""))
  (else (display (car lst)) (display " ") (WL (cdr lst)))))

(define W (lambda lst (WL lst)))     Display of a sequence of arguments (W <arg>...)
```

INTERNALS

```
scheme.ini                           A file in the current working directory that is loaded
                                     and executed after the start of pcs (TI PC Scheme)
patch.pcs                            A file in the Scheme home directory that is loaded and
                                     executed after the start of pcs (TI PC Scheme). It is used
                                     to customize the initialization of Scheme.
(edwin)                              Invoke the pcs EDWIN editor
PCS-DEGUB-MODE                       An environmental pcs variable that must be set to
(set! PCS-DEBUG-MODE true)           #T to enable full tracing, inspecting, and pp

[Inspect] ?                          TI Scheme Inspect Commands
   ?       -- display this command summary
   !       -- reinitialize INSPECT
 ctrl-A -- display All environment frame bindings
 ctrl-B -- display procedure call Backtrace
 ctrl-C -- display Current environment frame bindings
 ctrl-D -- move Down to callee's stack frame
 ctrl-E -- Edit variable binding
 ctrl-G -- Go  (resume execution)
 ctrl-I -- evaluate one expression and Inspect the result
 ctrl-L -- List current procedure
 ctrl-M -- repeat the breakpoint Message
 ctrl-P -- move to Parent environment's frame
 ctrl-Q -- Quit  (RESET to top level)
 ctrl-R -- Return from BREAK with a value
 ctrl-S -- move to Son environment's frame
 ctrl-U -- move Up to caller's stack frame
 ctrl-V -- eValuate one expression in current environment
 ctrl-W -- (Where) Display current stack frame
To enter `ctrl-A', press both `CTRL' and `A'.
[Inspect] Quit
[8] (exit)
```

NOTE: Some of the presented functions are implementation-dependent and may be unavailable in some Scheme implementations. Similarly, some Scheme implementations may have additional functions and features not included in this Reference Card. Features that are not available using Texas Instruments ***pcs*** *are denoted by* ***[T-]***, and those not available using MIT Scheme (installed on Libra) are denoted by ***[M-]***.