Scheme

Programming Language Essentials
1st edition
Chapter 1.1 Simple Expressions

more examples at
Format

; this is a comment
(NoT #F)

not case-sensitive
whitespace separates words
special characters can be (in) identifiers
some special characters are significant
minus, period, and digits cannot start identifiers
Literal
denotes “itself”

2
"string"
#t
#f
#\a
#\space
Variable
denotes the value of its binding
many standard bindings

x
very-long-name
name2
+
null?
Procedure Call

(operator operand₁ operand₂ ... )

terms evaluated prior to call
unknown order

(* (+ 2 3) 4)
((inc 2) 3)

higher-order: returns procedure
Definition

(define variable expression)

special form, controls evaluation
binds value of expression to variable
has no result value
rebinding is allowed
Conditional Evaluation

\((\text{if test-exp then-exp else-exp})\)

evaluates test-exp depending on result evaluates one of the others

\((\text{if (zero? 5) 1 (+ 1 2)})\)
Scheme

Programming Language Essentials
1st edition
Chapter 1.2 Data Types

more examples at
Data Type

set of values
procedures operating on the values
representation: internally and externally

dynamic type checking — Scheme
static type checking — C
a little bit of both — Java
Number

2 -5
(+ 1/3 4.5 5)
+ - * /

number?
=

<= >= < >

(some) arithmetic and comparison procedures take arbitrarily many arguments
Boolean

#t
#f

boolean?

eq?

not

(eq? (boolean? #f) (not #f))

result of comparisons and predicates
argument of conditionals
Character

\x  \space  \newline
char?
char=?
char<?
(char->integer  \space) ; not ASCII...
char->alphabetic?
char->numeric?
char-whitespace?
String

"this is a string"

string?

string-length

string-append

equal?

string->number  number->string

string->list  (string #\a #\b)

string->symbol  (symbol->string 'a)

(string-ref "abc" 0)
Symbol

identifier as a value

(quote x)
'x
(define x 'x)
symbol?
(eq? x 'x)

literals (numbers, booleans, strings, characters) are self-quoting
List

ordered sequence of elements

'(1 2 3) ; list value
'()       ; empty list
(+ 3 4)   ; procedure call/special form

(list 'a 1 '()) ; '(a 1 ())
(cons 'a '())   ; '(a)
(cons '() '())  ; '(())
(append '(a b) '(c d)) ; '(a b c d)
List (2)

(car '(a b c)) ; 'a
(cdr '(a b c)) ; '(b c)
(cadr '(a b c)) ; 'b
(cddr '(a b c)) ; '(c)
(caddr '(a b c)) ; 'c

null?
eq?
equal?
Pair

(cons 1 2)  ; '(1 . 2)
(cons 1 '()) ; '(1) ≡ '(1 . ())

pair?
eq?
equal?

cons cell with car and cdr fields
list consists of pairs linked through cdr and
terminated with ()
improper list is not terminated with ()
Shared Pairs

(define b '(3))
(define c b)
(define d (cons 2 b))
(define e (cons 2 c))

b and c print alike and are eq?
d and e print alike and are equal? but not eq?
(cdr d) and (cdr e) are both eq? to b and c
Vector

sequence of arbitrary values with efficient selection can be nested

(vector value₀ value₁ value₂ ... )
'#/value₀ value₁ value₂ ... )
vector?
vector-length
(vector-ref '#(0 1 2) 0)
vector->list
list->vector
Scheme

Programming Language Essentials
1st edition
Chapter 1.3 Procedures

more examples at
Procedure

first class value — can be passed to procedures and stored in data structures

(procedure? car) ; #t
(procedure? 'car) ; #f
(procedure? (car (list cdr)) ; #t

(((if (procedure? procedure?) cdr car)
  (cons car cdr))
  '(car in the car))
apply

executes a procedure value for a list of values as arguments

(apply + '(1 2))

(define abc '(a b c))
(apply cons (cdr abc))

(apply apply (list procedure? (list apply)))
**lambda**

creates an (anonymous) procedure value
names in list are bound to argument values
local to body of procedure

$$((\text{lambda } (n) (+ n 2)) \ 4)$$

(define add2 (lambda (n) (+ n 2)))
(add2 4)
map

(define map
  (lambda (proc list)
    (if (null? list)
        ()
        (cons (proc (car list))
            (map proc (cdr list))))
  )
)

(map (lambda (n) (+ n 2)) '(1 2 3 4))

[built-in to combine many list arguments]
andmap

(define andmap
  (lambda (pred? list)
    (if (null? list) #f
      (if (not (pred? (car list))) #f
        (if (= 1 (length list)) #t
          (andmap pred? (cdr list))))))
)

(andmap null? '())

(andmap null? '(()))

(andmap number? '(1 2 3))
compose

(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x))
    )))

(((compose car cdr) '(a b c d))
((compose list (compose cdr cdr))
  '(a b c d))

}
Currying

(define sub
  (lambda (x1)
    (lambda (x2)
      (- x1 x2)
    )))

(((sub 1) 2)

sub works like – but takes one argument at a time
Reverse Currying

(define sub-from
  (lambda (x)
    (lambda (y)
      (- y x)
    )))

((sub-from 2) 1)

when creating a curried procedure, arguments can be used in any order
Variable-Arity Procedure

any number of arguments can be received as a list

(define plus
  (lambda x
    (if (null? (cdr x)) ; one argument?
      (lambda (y) (+ (car x) y))
      (apply + x) ; more then one argument
    )))

(plus 1 2)
((plus 1) 2)