Datatypes

All values have one of the types: boolean number image text list function.
- in type contracts we also use “any” to refer to a value that could be of any of those types
- for each type there is a “type predicate” from any → boolean? that asks if a value has
  that type — the later section “Functions” is organized with these predicate as headings

Some functions are more specifically referred to as “unary”, “binary”, and/or “a predicate”.

The text/list with no characters/elements is also known as the “empty text/list”.

Definition Forms

(define variable-name value-expression)
- evaluate the value-expression, then the meaning of variable-name is that value

(define (function-name parameter-name ...) ; the part after “define” is known as the “header”
  body)
- the function-name is now a literal for a function that takes as many argument values as there
  are parameter-names, and substitutes those values into the body

Expression Forms

literal — a direct reference to a value of one of the six datatypes
variable-name — the value of a variable

(function argument-expression ...) — a function call
- the function is the name of a function in the language or defined function, or is anonymous
- evaluate each argument-expression to a literal value, then call the function
- if the function was defined or is anonymous: call it by substituting arguments into its body

Conditionals

(and condition, condition, ...)
(or condition, condition, ...)
- evaluate left-to-right, “delay” evaluation of later conditions, possibly “short-circuit”

(if [condition result] ; each grouped condition/else and result is known as an “if clause”
  ...     
  [else alternative-result])
- evaluate each condition in order, delay evaluation of later conditions and results

Anonymous Function

((fun parameter-name ...) ; the (fun parameter-name ...) part is the header
  body)
- a literal that directly refers to a function with those parameter-names and body
check

====

Documentation/Testing of a function f

Typically: (check (f a b c ...) literal-value)

Depending on the intent, any check involving f could be considered to document or test f.

Design for a function f

Partial : (check (f a b c ...) expression)
  • the expression is usually not a literal value, but not enough to be a Full Design

Partial or Full Design for a specific case/condition : (check (f a b c ...) expression)
  • when the function could handle the case with an if-clause or an or/and-condition, this
    is a Design for the result of the if-clause or condition of the or/and-condition

Full : (check (f a b c ...) expression)
  • where each of a b c ... appear in the expression, they appear as-is, unedited
  • then f must be definable as ...
    (define (f a b c ...)
     expression)
    ... after replacing a b c ... with parameter names in the header and body

step

====

Follow the evaluation described previously in “Definition Forms” and “Expression Forms”.

When showing (step original-expression) you do not need to (but are allowed to) show
“• Steps •”, the initial “○ original-expression”, or the bullets “•”.

If you wish to show the original expression again to underline the parts that will change,
you may underline those parts inside (step original-expression) to save writing.

There are specific steps to call map, combine, and repeats, described later in “Functions”.

Hiding

———

(step [hide f] expression)
• after the first step, don't show subsitution into the definition of f when calling f

(step [hide if-conditions] expression)
• in if-expressions: don't show conditions being evaluated, just show the result expression

(step [hide if-introduction] expression)
• when an if-expression would appear, just show the result expression
Functions
=======

identity : any → any  —— (identity v) → v

same? differ? : any any → boolean

function? : any → boolean

unary? binary? : function → boolean  —— whether the function is callable with one/two arguments

image? : any → boolean

width height : image → number

mirror flip : image → image  —— left-right for mirror, up-down for flip
counter clockwise : image → image  —— rotate 90° counter-clockwise / clockwise

rotate : image number → image  —— counter-clockwise

circle scale : image number → image

image-left image-right image-top image-bottom : image number → image

above beside overlay : image ... → image  —— join the images to produce one image

  • centered: top-to-bottom for above, left-to-right for beside, front-to-back for overlay

square star circle triangle : number → image , number text → image , number text text → image

  • the number specifies both width and height, except with triangle where it is just the width
  • the first text specifies the shading "outline" or "solid", which defaults to "outline"
  • the second text specifies the color, which defaults to "black"

rectangle oval : number number text text → image

  • the numbers specify width and height, text arguments are optional as with the other shapes

number? : any → boolean

zero? positive? negative? even? odd? : number → boolean

  • (zero? n) → (= n 0) , (positive? n) → (> n 0) , (negative? n) → (< n 0)
  • the predicates even?/odd? determine whether a whole number is/isn't a multiple of two

= ≤ > ≥ : number number ... → boolean  —— must be called with at least two arguments

maximum minimum : number ... → number  —— must be called with at least one argument

dec inc - : number → number  —— (dec n) → (- n 1) , (inc n) → (+ n 1) , (- n) → (* -1 n)

- / : number number → number  —— subtract or divide the two numbers

+ * : number ... → number  —— add up or multiply all of the numbers

  • may be called with no arguments: (+) → 0 , (*) → 1

random : number → number  —— (random n) → a random number from (range n)
boolean? : any → boolean

_________________________________________
not : boolean → boolean —— (not #true) → #false, (not #false) → #true

text? : any → boolean

_________________________________________

text-length : text → number —— the number of characters in the text
text-join : text ... → text —— all the characters in the texts, as one text

sub-text : text number number → text —— (sub-text t i j) → characters with index in (range i j)
character : text number → text —— (character t i) → (sub-text t i (inc i))

number→text : number → text —— the number written out as text
text→image : text number → image —— an image of the characters of the text, with given height

list? : any → boolean

_________________________________________
list : any ... → list —— the arguments gathered as elements in a list

length : list → number —— the number of elements in the list
join : list ... → list —— all the elements in the lists, as one list
• (join (list a₀ b₀ ...) (list a₁ b₁ ...) (list a₂ b₂ ...) ...) → (list a₀ b₀ ... a₁ b₁ ... a₂ b₂ ... ...)
• in particular: (join) → (list), (join l) → l

prepend : any list → list —— (prepend v (list a b c ...)) → (list v a b c ...)
append : list any → list —— (append (list a b c ...) v) → (list a b c ... v)

sub-list : list number number → list —— (sub-list l i j) → elements with index in (range i j)
element : list number → any —— the element of the list at a specific index
first second third fourth : list → any —— the element of the list at index 0 / 1 / 2 / 3

element? : any list → boolean —— whether a value is an element of a list

rest : list → list —— (rest (list a b c ...)) → (list b c ...)
reverse : list → list —— (reverse (list a b ... y z)) → (list z y ... b a)

range : number → list , number number → list , number number number → list
• (range start end by) → (list start (+ start by) (+ start by) ...) up to but excluding end
• (range start end) → (range start end 1)
• (range end end) → (range 0 end 1)

map : function list → list —— step as (map f (list a b c ...)) → (list (f a) (f b) (f c) ...)
combine : function list → any —— step as (combine f (list a b c ...)) → (f a b c ...)

repeats : function any number → list
• step as (repeats f seed n) → (list seed (f seed) (f (f seed)) ...) , with n elements

select : function list → list —— (select p? l) → list of elements from l for which p? is #true