L7: Equality and identity tests. Input and output. Repetition through iteration

Mircea Marin
mircea.marin@e-uvt.ro

April 2020
Equality and identity

Every value is represented by an object in the memory of the computer.

- Two values are **equal** when they are semantically the same. For example, the integer 1 is equal to the floating-point number 1.0, because they represent the same quantity.
- Two values are **identical** when they are represented by the same object in the memory of the computer.
  - identical values are always equal.
  - some equal values are not identical. For example 1 is not identical to 1.0 because they have different representations in the memory of the computer.
Interned values. Identity tests

Some types have **interned values**: the interpreter creates a single object for a value of that type.

- Integers, floating-point numbers, booleans, and symbols are interned values. Strings are usually not interned.
- \((eq? \, v_1 \, v_2)\) returns \(\#t\) if \(v_1, \, v_2\) are identical, and \(\#f\) otherwise.

```scheme
> (define-values (x y z t u v w) (values 1 1 1.0 'a 'a 1.0 "a"))
> (list (eq? x y) (eq? x z) (eq? t u) (eq? z v) (eq? u w))
'(#t #f #t #t #f)
```

The internal representation of these values in computer memory is

There is only one object created for the values 1, 1.0, and 'a.
Equality tests

\texttt{eqv?} and \texttt{=} 

- \texttt{eqv?} is like \texttt{eq?} but does the right thing when comparing numbers. \texttt{eqv?} returns \texttt{#t} if its arguments are \texttt{eq?} or if its arguments are numbers that have the same value. \texttt{eqv?} does not convert integers to floats when comparing integers and floats.

- \texttt{=} is used to compare numbers. Like Java, C, C++, \texttt{=} in Racket converts one type to another before performing comparison.

\begin{verbatim}
> (eqv? 10 10) #t
> (eqv? 10.0 10.0) #t
> (eqv? 10.0 10) ; no conversion between types #f
> (= 0 0.0) #t
\end{verbatim}
Equality and identity
Input and output
Repetition through iteration

Equality tests for composite values

In general, two composite values are equal? if they have the same structure and their corresponding components are equal?

- equal? is defined recursively

Example

```scheme
> (equal? 0 0.0)
#f

> (equal? '(1 2 3) '(1 2 3))  > (equal? '(#1 2 3) '(#1 2 3))
#t                      #t

> (equal? '(1 . 2) '(1 . 2))
#t

> (eq? '(1 2 3) '(1 2 3))    > (eqv? '(1 2 3) '(1 2 3))
#f                        #f

> (struct emp (name salary) #:transparent)
> (equal? (emp "john" 3400) (emp "john" 3400))
#t

> (eq? (emp "john" 3400) (emp "john" 3400))
#f
```
Input and output

Ports

We can read or write data from system resources of many kinds, including files, the terminal, TCP connections, in-memory strings, etc.

- Access to a system resource is provided through a port:
  - data can be read or written sequentially, as a stream of bytes, without requiring the data to be consumed or produced all at once

- An input port produces bytes
- An output port consumes bytes
- Some ports are both input ports and output ports
- The bytes produced by input ports can be read by input operations, and the bytes consumed by output ports are produced by output operations.
Input/output operations

The I/O operations are designed to read/write values of various types: bytes, characters, numbers, strings, etc.

- The simplest operations are byte-based
- All other I/O operations make use of an encoder or decoder which knows how to represent a value as a sequence of bytes.

Basic remarks

- To read/write data from/to a system resource, we must get a port associated with it.
- When a port is associated with a file, network connection, or other system resource, it must be closed explicitly via `close-input-port` or `close-output-port` to release the resources associated with it. Afterwards, any attempt to read or write data through that port will raise an `exn:fail` exception.
An input port is associated with a stream from which we can read of peek data.

- **reading** removes a datum from the input stream
- **peeking** leaves the input stream unchanged

These operations are normally **blocking**: they do not complete until data is available from the port.

- Non-blocking variants of read and peek operations are also available.
Default ports

- Most I/O functions have optional arguments for the input or output ports, whose default values are the current input port or current output port. There is also a special output port where error messages are written, called the current error port. The current input/output/error ports do not have literal representations, but they can be obtained by calling the nullary functions current-input-port, current-output-port, and current-error-port.

- By default, the current input/output/error ports are associated with the interactions area of DrRacket.
We can read a value from a port \textit{in} with the function call

\begin{verbatim}
(read in)
\end{verbatim}

1. If \textit{in} is the current input port, it can be omitted. Thus, (read) is the same as (read (current-input-port)).

2. When \textit{in} is the current input port, the function call is blocking: It opens an input area where the user is expected to type in the unquoted form of a value, and to press \texttt{ENTER}. The returned value of this function call is the value obtained by quoting the given input.

**Example**

> (read)

(a b c)

After pressing \texttt{ENTER}, the returned value will be \texttt{'(a b c)}.
(read-line in mode)

reads a string of characters from port in until, but not including, an end-of-line or end-of-file separator. If no characters are read before an end-of-file is encountered, the function returns the value \texttt{eof}.

the value of \textit{mode} indicates how to detect an end-of-line:

- \textquote{linefeed} breaks lines on linefeed characters.
- \textquote{return} breaks lines on return characters.
- \textquote{return-linefeed} breaks lines on return-linefeed sequence of characters.
- \textquote{any} breaks lines on any of a return character, linefeed character, or return-linefeed combination. If a return character is followed by a linefeed character, the two are treated as a combination.
- \textquote{any-one} breaks lines in any of the previous three cases.

The default value of \textit{in} is the current input port; the default value of \textit{mode} is \textquote{linefeed}.
A built-in value can be printed in three ways:

1. with `print`, which prints a value like the REPL loop
2. with `write`, whose printed form can be passed to `read` to produce the value back
3. with `display`, which does not show the double quote delimiters of string literals as well as the initial quote character of quoted literals, otherwise it behaves like `write`.

These functions are called as follows:

```lisp
(print  datum  out)
(write  datum  out)
(display datum  out)
```

where the argument `out` is optional.

- The default value of `out` is the current output port.
- All these function calls return `#<void>` as result.
Writing operations

> (print 1/2)  
1/2  
> (print \#x)  
\#x  
> (print "hello")  
"hello"  
> (print '("i" pd))  
'("i" pd)  
> (print +)  
#<procedure:+>  

> (write 1/2)  
1/2  
> (write \#x)  
\#x  
> (write "hello")  
"hello"  
> (write '("i" pd))  
'("i" pd)  
> (write +)  
#<procedure:+>  

> (display 1/2)  
1/2  
> (display \#x)  
\#x  
> (display "hello")  
"hello"  
> (display '("i" pd))  
'("i" pd)  
> (display +)  
#<procedure:+>  

Remark: The printed information is a side effect of the function call. For example:

> (void? (write "abc"))  
"abc"#t

displays "abc" as side effect of the function call (write "abc"), followed by #t, which is the printed form of the value returned by (void? ...).
Other I/O operations

- **println**, **writeln**, **displayln**
- **printf** supports simple formatting of data and text. In the format string supplied to **printf**, `~a` displays the next argument, `~s` writes the next argument, and `~v` prints the next argument.

```racket
> (define (deliver who when what)
   (printf "Items ~a for shopper ~s: ~v" who when what))
> (deliver '("list") '("John") '("milk"))
```

Items (list) for shopper ("John"): '("milk")

See also:
- The **Racket** Guide, Chapter 8.
- The **Racket** Reference, Chapter 13.
For every positive integer $k$, the predefined operation \((\text{random } k)\) returns a random exact integer in the range 0 to $k - 1$.

\(\text{random}\) is not a mathematical function because it may return different results when called the same input argument.

Consider the following function definition:

\[
(\text{define} \ (\text{make-oracle})) \newline
\ (\text{let} \ ([[n \ (\text{random} \ 1024)]]) \newline
\ (\text{lambda} \ (k)) \newline
\ (\text{cond} \ [[(< \ k \ n) \ "larger"] \newline
\ [ (> \ k \ n) \ "smaller"] \newline
\ [(eq? k n) \ "won!"])))
\]
The effect of evaluating `(define f (make-oracle 1024))` in an environment $E$:

After evaluating `(define f (make-oracle 1024))`:

The interpreter chooses a random number $\nu$ between 0 and 1023 and stores it in the local variable $n$, which is visible only from the body of function $f$. 

---

M. Marin
An interactive game
Random numbers and an oracle

You are asked to play the following game: Given a newly created oracle function \( f \), find the value of \( n \) by performing at most ten function calls \( (f \ k) \) where \( 0 \leq k \leq 1023 \). You win when your function call \( (f \ k) \) returns "won!", and this happens when you guess the value of \( n \) for \( k \).

```
(define (start-play)
  (define f (make-oracle))
  (define (play [tries 10]) ; you have 10 tries
    (cond [(= tries 0) "failed!" ] ; you ran out of tries
          [#t (display "Guess:"))
            (define answer (f (read)))
            (displayln answer)
            (unless (equal? answer "won!"
                                    (play (- tries 1)) ; continue playing
                    ))])
  (play))
> (start-play)
```
What is iteration?

**Iteration**: type of repetition that, like recursion, involves repeating a task a certain number of times, or for every element in a list, or until a condition is met.

Usually, iteration changes the values of some loop variables before repeating the same kind of computation, until a condition is met.

- We can not change the values of variables in FP
  - we can not perform iterative computations
  - **but** we can simulate them by tail-recursive function calls.

**Iteration versus recursion**

- Iterative computations are desirable because they are very efficient: they are fast and consume little memory.

- Recursive computations **tend to be less efficient**: every recursive call consumes memory and time to extend the environment with new frames; but **tail-recursive function calls have the same efficiency as iterative computations**.
do

(\(\text{do}\) \((\left[\text{var}_1 \text{ initial\_value}_1 \text{ update\_expr}_1\right] \ldots \left[\text{var}_n \text{ initial\_value}_n \text{ update\_expr}_n\right]\))
\((\text{test exit\_body})\)
\(\text{body}\))

- First, all local variables \(\text{var}_1, \ldots, \text{var}_n\) are bound to the values of \(\text{init\_value}_1, \ldots, \text{init\_value}_n\), like in the \(\text{let*}\) form.
- Next, \(\text{test}\) is evaluated.
  - If its value is non-\(#f\), then \(\text{exit\_body}\) is evaluated and its value is returned.
  - If its value is \(#f\), then \(\text{body}\) is evaluated. Next, \(\text{var}_1, \ldots, \text{var}_n\) are bound to the values of \(\text{update\_expr}_1, \ldots, \text{update\_expr}_n\), and the process is repeated by evaluating \(\text{test}\) again.

**Remarks**
- If \(\text{exit\_body}\) is omitted, the \(#<\text{void}>\) value is returned.
- If \(\text{update\_expr}_i\) is omitted, \(\text{var}_i\) keeps its initial value.
Examples

(define (fact-iter n) ; iterative implem. of fact with do
  (do ([i n (- i 1)]
      [result 1 (* result i)])
      (= i 0) result)
  ; empty body: nothing to do
))

(define (reverse-iter lst) ; iterative implem. of reverse with do
  (do ([acc null (cons (car l) acc)]
      [l lst (cdr l)])
      (null? l) acc)))

(define (map-iter f lst) ; iterative implem. of map with do
  (do ([l (reverse lst) (cdr l)]
      [result ’() (cons (f (car l)) result)])
      (null? l) result)))
do
A tail-recursive implementation

(define (do-iter x_1 ... x_n)
  (let ([var_1 x_1]
         ... [var_n x_n])
    (cond [test exit_body]
          [#t body (do-iter update_expr_1 ... update_expr_n)]))

Remark: The computation of
(do ([var_1 initial_value_1] ... [var_n initial_value_n]
    (test exit_body)
    body)
  body)

is equivalent with the computation of
(do-iter initial_value_1 ... initial_value_n)
do
A tail-recursive implementation

```
(define (do-iter \(x_1 \ldots x_n\))
  (let ([var_1 x_1]
        \ldots
        [var_n x_n])
    (cond [test exit_body]
          [#t body (do-iter updateExpr_1 \ldots updateExpr_n)]))
```

**Remark:** The computation of

```
(do ([var_1 initialValue_1 updateExpr_1]
     \ldots
     [var_n initialValue_n updateExpr_n])
   (test exit_body)
   body)
```

is equivalent with the computation of

```
(do-iter initialValue_1 \ldots initialValue_n)
```
Applications

Repeating actions a number of times

\[
\text{(do ([counter 1 (+ counter 1)])}
\]
\[
\text{((> counter num\_times) exit\_body)}
\]
\[
\text{body)}
\]

or

\[
\text{(do ([counter num\_times (- counter 1)])}
\]
\[
\text{((= counter 0) exit\_body)}
\]
\[
\text{body)}
\]

Example (Print all powers of } a, \text{ from } 1 \text{ to } num\_times)\)

\[
\text{(define (print\_powers a num\_times)}
\]
\[
\text{(do ([n 1 (+ n 1)])}
\]
\[
\text{((> n num\_times))}
\]
\[
\text{(println (expt a n)))}
\]

M. Marin
Applications
Repeating an action for every element in a list lst

\[
\text{(do (\text{[current_list lst (cdr current_list)]})
  ((null? current_list) exit_body)
  body)}
\]

Example (Check if a list consists of numbers only)

\[
\text{(define (number_list? lst)
  (do (\text{[current_list lst (cdr current_list)]})
    ((or (null? current_list)
        (not (number? (car current_list))))
     (if (null? current_list)
       #t
       #f)))}
\]
Applications
Repeating an action for every element in a list \( \text{lst} \)

\[
\text{(do ( \text{current_list lst (cdr current_list)} \))}
\]

((null? \text{current_list}) exit_body)
body)

Example (Check if a list consists of numbers only)

(define (number_list? lst)
  (do ( \text{current_list lst (cdr current_list)} \))
    ((or (null? \text{current_list})
      (not (number? (car current_list))))
      (if (null? \text{current_list})
        #t
        #f))))

Remark: this is an iteration with multiple exit cases:
- the exit cases are combined in an \text{or} in the test of the \text{do}
- the return value is determined by verifying the specific condition that caused the exit
do
Mistakes to avoid

- The return value of a do loop is that of its exit_body.
- The body of a do loop is only used for side-effects, like printing out values:
  - this is different from the do loop in other languages, like Pascal or C.

Example (A buggy implementation of number_list?)

```scheme
(define (buggy_number_list? lst)
  (do ([current_list lst (cdr current_list)])
      ((null? current_list) #t)
      (if (not (number? (car current_list))) #f)))

> (buggy_number_list? '(1 2 three))
#t
```
A worked-out example
List sorting by insertion

Given a list \texttt{lst} and a comparison function \texttt{cmp} such that
\[
(cmp \ x \ y) \text{ yields } \#t \text{ if } x \text{ is less or equal to } y, \text{ and } \#f \text{ otherwise}
\]

Define \(\text{sort-iter} \ cmp \ \texttt{lst}\) which returns \texttt{lst} with its elements arranged in increasing order w.r.t. \texttt{cmp}.

**Main idea:** use two loop variables:

- \texttt{crt}, initially \texttt{lst}, from which we repeatedly remove its first element
- the sorted list \texttt{answer}, initially \texttt{null}, into which we repeatedly insert (\texttt{car \ crt}) in the proper position.

and stop when \texttt{crt} becomes \texttt{null}

\(\Rightarrow\) \texttt{answer} will be the sorted version of \texttt{lst}.
A worked-out example
List sorting by insertion (contd.)

\[
\begin{align*}
\text{(define (sort-iter \texttt{cmp} \texttt{lst})} & \\
\text{ (do ([crt \texttt{lst} (cdr \texttt{crt})]} & \\
\text{ [answer null (insert-iter (car \texttt{crt}) answer \texttt{cmp})]])} & \\
\text{ ((null? \texttt{crt}) answer))} & 
\end{align*}
\]
A worked-out example
List sorting by insertion (contd.)

(define (sort-iter cmp lst)
  (do ([crt lst (cdr crt)]
       [answer null (insert-iter (car crt) answer cmp)])
    ((null? crt) answer)))

; inserts x in sorted-list at proper position
(define (insert-iter x sorted-list cmp)
  (do ([lst1 '() (cons (car lst2) lst1)]
       [lst2 sorted-list (cdr lst2)])
    ((or (null? lst2)
         (cmp x (car lst2))
         (append (reverse lst1) (cons x lst2))))))
A worked-out example
An iterative definition of (insert-iter x sorted-lst cmp)

**Main idea:** use two loop variables:

- **lst2**, initially sorted-lst, from which we repeatedly remove its first element, if it is greater than \( x \)
- **lst1**, initially empty, to which we add (with `cons`) the first element of **lst2**, if it is greater than \( x \)

⇒ the proper position is detected when

- **lst2** becomes ’()  
  or when
- \((\text{cmp } x (\text{car } \text{lst2}))\) yields #t
Main idea: use two loop variables:

- **lst2**, initially sorted-lst, from which we repeatedly remove its first element, if it is greater than **x**
- **lst1**, initially empty, to which we add (with **cons**) the first element of **lst2**, if it is greater than **x**

⇒ the proper position is detected when

- **lst2** becomes ’()  

  **or** when

  - (cmp **x** (car **lst2**)) yields #t

In both cases, we place **x** between the reverse of **lst1** and **lst2**.
**Main idea:** define an iterative function for every loop.

**Example (Find an employee in a company)**

\[
\langle \text{company} \rangle ::= (\text{list } \langle \text{division} \rangle \ldots \langle \text{division} \rangle) \\
\langle \text{division} \rangle ::= (\text{list } \langle \text{div-name} \rangle \langle \text{department} \rangle \ldots \langle \text{department} \rangle) \\
\langle \text{department} \rangle ::= (\text{list } \langle \text{dept-name} \rangle \langle \text{emp-name} \rangle \ldots \langle \text{emp-name} \rangle)
\]

The following is an instance of a company:

```
'(far-east (engineering gino bill)  
     (advertising betty kim alice))
(eastern (health janet)  
     (technical eric robin))
(western (engineering stephen egon)  
     (investment brian)))
```
Main idea: define an iterative function for every loop.

Example (Find an employee in a company)

\[
\langle \text{company} \rangle ::= (\text{list} \langle \text{division} \rangle \ldots \langle \text{division} \rangle)
\]
\[
\langle \text{division} \rangle ::= (\text{list} \langle \text{div-name} \rangle \langle \text{department} \rangle \ldots \langle \text{department} \rangle)
\]
\[
\langle \text{department} \rangle ::= (\text{list} \langle \text{dept-name} \rangle \langle \text{emp-name} \rangle \ldots \langle \text{emp-name} \rangle)
\]

The following is an instance of a company:

'((far-east (engineering gino bill)
   (advertising betty kim alice))
 (eastern (health janet)
   (technical eric robin))
 (western (engineering stephen egon)
   (investment brian)))

Define (\texttt{find-emp person company}) which returns

the list of division name and department name where person occurs in company,

and \texttt{f} if person is not an employee of company.
Nested loops using iteration
Example continued

> (define comp '((far-east (engineering gino bill)
    (advertising betty kim alice))
  (eastern (health janet)
    (technical eric robin))
  (western (engineering stephen egon)
    (investment brian))))

> (find-emp 'eric comp)
'(eastern technical)

person can be looked up in comp by two nested loops:

Outer loop: for each div ∈ comp: call (find-dept div person) to detect if there is dept ∈ div such that person ∈ dept

Inner loop: the function call (find-dept div person) checks if there is dept ∈ div such that person ∈ dept
  ▶ if yes, return the name of dept
  ▶ otherwise, return #f
Auxiliary functions:

; return name of division
(define (div-name division)
  (car division))

; return list of departments of division
(define (dept-list division)
  (cdr division))

; return name of department
(define (dept-name department)
  (car department))

; return list of employee names in department
(define (employees department)
  (cdr department))
Nested loops using iteration
Example continued

(define (find-emp person division-list)
  ; outer loop
  (do ([divs division-list (cdr divs)])
    ((or (null? divs) (find-dept (dept-list (car divs)) person))
      (if (null? divs)
        #f
        (list (div-name (car divs))
          (find-dept (dept-list (car divs)) person)))))

(define (find-dept company person)
  ; inner loop
  (do ([d-list company (cdr d-list)])
    ((or (null? d-list)
      (member person (employees (car d-list))))
      (if (null? d-list)
        #f
        (dept-name (car d-list))))))