

Lab 2: Composite datatypes: pairs, lists, and vectors. Local definitions

Mircea Marin
mircea.marin@e-uvv.ro

March 1, 2021

Expressions

- All functional programming languages, including Racket, compute by evaluating expressions
- There are four kinds of expressions:
 - **values**: they evaluate to themselves.
 - **variables**: they are names given to values. The evaluation of a variable yields its value.
 - **function calls**: they are evaluated **strictly**:
 - first, all the argument of the function call are reduced to values, from left to right
 - next, the body of the called function is evaluated, with the function arguments instantiated with the values passed to the function call.
 - **special forms**, like
 - `(if test e1 e2)`
 - `(define var expr)`
 - ...

Every special form has its own rule(s) of evaluation.

Values

Remember that ...

- Values are **atomic** (e.g., numbers, strings, booleans, symbols) or **composite**
- A composite value is a value produced by putting together other kinds of values.
- A datatype whose elements are composite is a **composite datatype**
- The composite datatypes of Racket include: pairs, lists, vectors, hash tables, etc.

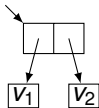
Composite datatypes

Every composite datatype has:

- **recognizers** = boolean functions that recognize values of that type.
- **constructors** = functions that build a composite value from component values
- **selectors** = functions that extract component values from a composite value
- **utility functions** = useful functions that operate on//with composite values
- A specific **internal representation** that affects the efficiency of the operations on them

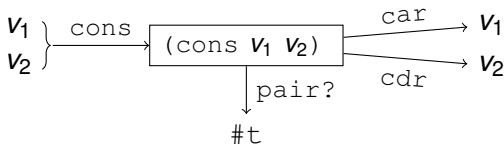
Pairs

- The simplest container of two values
 - **constructor**: `(cons V1 V2)`
 - **internal representation**: a **cons-cell** that stores pointers to the internal representations of v_1 and v_2



- `(cons? p)`: returns `#t` if the value of p is a pair, and `#f` otherwise.
- **selectors**
 - `(car p)`: returns the first component of pair p
 - `(cdr p)`: returns the second component of pair p

Diagrammatically, these operations behave as follows:



Operations on pairs

Examples

```
> (define p (cons 1 "a"))      > (define q (cons 'a 'b))
> p                            > q
'(1 . "a")                    '(a . b)
> (pair? p)                   > (car q)
#t                             'a
> (car p)
1
> (cdr p)
"a"
```

Remark

We can nest pairs to any depth to store many values in a single structure:

```
> (cons (cons (1 'a) "abc"))
'((1 . a) . "abc")
> (cons (cons 'a 1) (cons 'b (cons #t "c")))
'((a . 1) b #t . "c")
```

The printed form of pairs

RACKET applies repeatedly two rules to reduce the number of quote characters(') and parentheses in the printed forms:

rule 1: `(cons v1 v2)` is replaced by

`' (w1 . w2)`

where w_1, w_2 are the printed forms of v_1, v_2 from which we remove the preceding quote, if any.

There is space before and after the dot character in the printed form.

rule 2: Whenever there is a dot character before a parenthesised expression, remove the dot character and the open/close parentheses.

Printed form of pairs

Example

```
> (cons (cons 'a 1) (cons 'b (cons #t (cons "c" 'd))))  
'((a . 1) b #t "c" . d)
```

The printed form is obtained as follows:

- Apply rule 1 to reduce the number of quote characters \Rightarrow the form `'((a . 1) . (b . (#t . ("c" . d))))`
- Apply repeatedly rule 2 to eliminate dots and open/close parentheses:

```
'((a . 1) . (b . (#t . ("c" . d)))) $\rightarrow$ '((a . 1) b . (#t . ("c" . d)))  
'((a . 1) b . (#t . ("c" . d))) $\rightarrow$ '((a . 1) . (#t "c" . d))  
'((a . 1) . (#t "c" . d)) $\rightarrow$ '((a . 1) b #t "c" . d)
```

The final form is the printed form:

```
'((a . 1) b #t "c" . d)
```


We can input directly the printed forms, which are usually much shorter to write than combinations of nested `cons`-es:

Example

Instead of `(cons (cons 'v11 'v12) (cons 'v21 'v22))`
we can type `'((v11 . v12) v21 . v22):`

```
> (define p '((v11 . v12) v21 . v22))
> p
'(v11 . v12) v21 . v22)
> (car p)                > (cdr p)
'(v11 . v12)            '(v21 . v22)
> (car (car p))        > (cdr (car p))
'v11                    'v12
> (car (cdr p))        > (cdr (cdr p))
'v21                    'v22
```

Selectors for nested pairs

The selection of an element deep in a nested pair is cumbersome:

```
> (define p '(a ((x . y) . c) d))
```

To select the **second** of the **first** of the **first** of the **second** component of `p`, we must type

```
> (cdr (car (car (cdr p))))  
'y
```

We can use the shorthand selector `cdaadr`:

```
> (cdaadr p)  
'y
```

Other shorthand selectors: `cx1...xpr` where $x_1, \dots, x_p \in \{a, d\}$ and $1 \leq p \leq 4$ (max. 4 nestings)

Lists

Constructors and internal representation

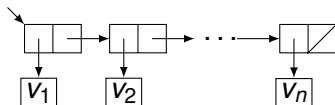
A **recursive** datatype with two **constructors**:

- `null`: the empty list
- `(cons v l)`: the list produced by placing the value v in front of list l .

If $n \geq 1$, the list of values v_1, v_2, \dots, v_n is

`(cons v1 (cons v2 ... (cons vn null) ...))`

with the **internal representation**



REMARK: The internal representation of a list with n values v_1, \dots, v_n consists of n `cons`-cells linked by pointers.

Printed form of lists

```
> null  
'() ; the printed form of the empty list
```

All non-empty lists are pairs, and their printed form is computed like for pairs.

Example

```
> (cons 'a  
      (cons 'b  
            (cons 'c (cons (cons 'd null)  
                          null))))  
'(a b c (d))
```

This printed form is obtained by applying repeatedly rule 2 to the form '(a . (b . (c . ((d . ()) . ())))

Lists

Other constructors and selectors

A simpler **constructor** for the list of values v_1, v_2, \dots, v_n :

```
> (list v1 v2 ... vn)
```

Selectors:

- (car *lst*) selects the first element of the non-empty list *lst*
- (cdr *lst*) selects the tail of the non-empty list *lst*
- (list-ref *lst* *k*) selects the element at position *k* of *lst*

NOTE: The elements are indexed starting from position 0.

Example

```
> (list 'a #t "bc" 'd)      > null
'a #t "bc" 'd              '()
> (list '() 'a '(b c))     > (list-ref '(1 2 3) 0)
'(() a (b c))              1
> (list-ref '(1 (2) 3) 1)
'(2)
```

List recognizers

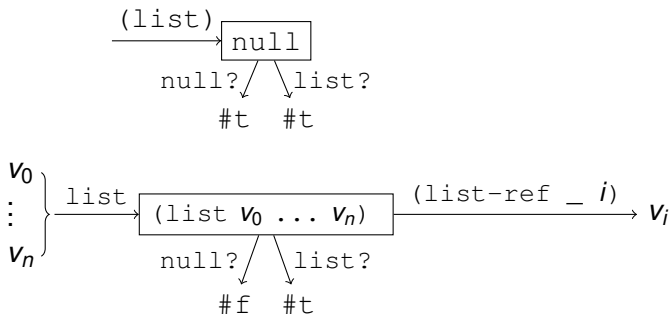
- `(list? lst)` recognizes if *lst* is a list.
- `(null? lst)` recognizes if *lst* is the empty list.

Example

```
> (define lst '(a b c d))
> (list? lst)
#t
> (car lst)
'a
> (cdr lst)
'(b c d)
> (list-ref lst 0)
'a
> (list-ref lst 1)
'b
```

List operations

Diagrammatic representation of their behavior



Utility functions on lists

(**length** *lst*) returns the length (=number of elements) of *lst*

```
> (length ' (1 2 (3 4)))      > (length ' ())  
3                             0
```

(**append** *lst*₁ ... *lst*_{*n*}) returns the list produced by joining lists *lst*₁, ..., *lst*_{*n*}, one after another.

```
> (append ' (1 2 3) ' (a b c))  
' (1 2 3 a b c)
```

(**reverse** *lst*) returns the list *lst* with the elements in reverse order:

```
> (reverse ' (1 2 3))  
' (3 2 1)
```


Operations on lists (1)

apply and filter

- If f is a function and lst is a list with component values v_1, \dots, v_n in this order, then
(`apply` f lst)
returns the value of the function call $(f\ v_1\ \dots\ v_n)$.
- If p is a boolean function and lst is a list, then
(`filter` p lst)
returns the sublist of lst with elements v for which $(p\ v)$ is true.

Examples

```
> (apply + '(4 5 6))      ; compute 4+5+6
15
> (filter symbol? '(1 2 a #t "abc" (3 4) b))
'(a b)
> (filter number? '(1 2 a #t "abc" (3 4) b))
'(1 2)
```

Operations on lists (2)

map

If f is a function and lst is a list with component values v_1, \dots, v_n in this order, then

```
(apply f lst)
```

returns the list of values w_1, \dots, w_n where every w_i is the value of $(f v_i)$

Example

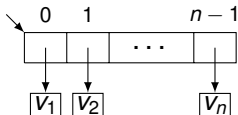
```
> (map (lambda (x) (+ x 1)) '(1 2 3 4))  
'(2 3 4 5)  
> (map list? '(1 2 () (3 4) (a . b)))  
'(#f #f #t #t #f)
```

Vectors

A composite datatype of a fixed number of values.

Constructors:

- `(vector v0 v1 ... vn-1)`
constructs a vector with n component values, indexed from 0 to $n - 1$, and internal representation



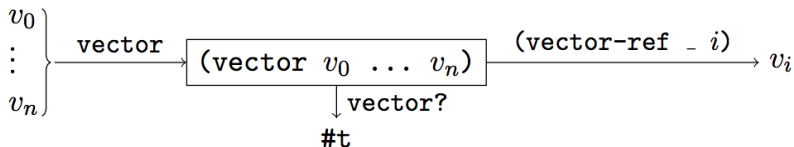
- `(make-vector n v)`
returns a new vector with n elements, all equal to v .

Recognizer: `vector?`

Selectors: `(vector-ref vec i)`

returns the component value with index i of the vector `vec`.

Operations on vectors



Example

```
> (define vec (vector "a" '(1 . 2) '(a b)))  
> (vector? vec)  
#t  
> (vector-ref vec 1)  
'(1 . 2)  
> (vector-ref vec 2)  
'(a b)  
> (vector-length vec) ; compute the length of vec  
3
```

Printed form of vectors

The printed form of a vector with component values v_0, v_1, \dots, v_n is

```
'#(w0 w1 ... wn)
```

where w_i is the printed form of v_i from which we remove the preceding quote character, if any.

Examples

```
> (vector 'a #t '(a . b) '(1 2 3))  
'#(a #t (a . b) (1 2 3))  
> (vector 'a (vector 1 2) (vector) "abc")  
'#(a #(1 2) #() "abc")  
> (make-vector 3 '(1 2))  
'#((1 2) (1 2) (1 2))
```

The printed forms of vectors are also valid input forms:

```
> '#(1 2 3)      > (vector? '#(1 2 3))  
'#(1 2 3)      #t
```

The `void` datatype

Consists of only one value, `' #<void>`:

- The recognizer is `void?`
- Attempts to input `' #<void>` directly will raise a syntax error:

```
> ' #<void>  
read: bad syntax `#<'
```

- We can obtain `' #<void>` indirectly, as the value of the function call `(void)`:

```
> (list 1 (void) 'a)  
' (1 #<void> a)  
> (void? (void))  
#t
```

- Usually, `' #void` is not printed

```
> (void) ; nothing is printed
```

Equality in RACKET: `eq?`, `eqv?` and `equal?`

There are many notions of object equality. The weakest notion is **structural equality**: the objects are not the same (in computer memory), but one can be replaced by the other in an expression, without causing any difference. The strongest notion of equality is **identity**: two objects are identical if they refer to the same object in computer memory. Racket has several predicates to test equality:

- `(eq? e1 e2)` yields `#t` if `e1` and `e2` evaluate to identical values, and `#f` otherwise.

```
> (eq? 1 1)      > (eq? 2 (+ 1 1))  > (eq? 1 1.0)
#t                #t                #f
```

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

Equality in RACKET: `eq?`, `eqv?` and `equal?`

`equal?` is especially useful when comparing compound values, such as lists.

- In general, `equal?` returns true if its arguments have the same structure. Formally, we can define `equal?` recursively. 'item `equal?` returns `#t` if its arguments are `eqv?`, or if its arguments are lists whose corresponding elements are `equal?`; and otherwise false.
- Two objects that are `eq` are both `eqv?` and `equal?`. Two objects that are `eqv?` are `equal?`, but not necessarily `eq?`.
- Two objects that are `equal?` are not necessarily `eqv?` or `eq?`.

Equality predicates

Examples

```
> (eq? "abc" "abc")
#t
> (eq? "abc" (symbol->string 'abc))
#f
> (eq? "abc" (keyword->string '#:abc))
#f
> (eq? 10 10)
#t
; (generally, but implementation-dependent)
> (eq? (/ 1.0 3.0) (/ 1.0 3.0))
#f
; (generally, but implementation-dependent)
> (eqv? 10 10)
#t
> (eqv? 10.0 10.0)
#t
> (eqv? 10.0 10) ; no conversion between types
#f
> (equal? 0 0.0)
#f
> (= 0 0.0)
#t
> (equal? "abc" (symbol->string 'abc))
#t
> (equal? "abc" (keyword->string '#:abc))
#t
```

Definitions in RACKET

Remember that:

- `(define name expr)`
is a special form which assigns name *name* to the value of *expr*.
- `(lambda (x1 ... xn) body)`
is a special form with the intended reading “the function which, for input arguments *x₁, ..., x_n*, computes the value of *body*.”
When evaluated, it creates a function value.

Racket also has special forms `let` and `let*` to define local variables:

```
(let ([var1 expr1]
      ...
      [varn exprn])
  expr)
```

```
(let* ([var1 expr1]
       ...
       [varn exprn])
  expr)
```

The `let` form

```
(let ([var1 expr1]  
      ...  
      [varn exprn])  
  block)
```

is evaluated as follows:

- 1 $expr_1, \dots, expr_n$ are evaluated to values v_1, \dots, v_n .
- 2 The definitions $var_1 = v_1, \dots, var_n = v_n$ are made local to *block*.
- 3 *block* is evaluated, and its value is returned as final result.

Remark

This special form is equivalent to

```
((lambda (var1 ... varn) body) expr1 ... exprn)
```

The let form

Examples

```
> (let ([x 5])
     (let ([x 2]           ; binds x to 2
           [y x])         ; binds y to the value of the outer x, which is 5
         (+ x y)))       ; computes the value of 2+5
```

7

```
> (let ([x 5])
     (let ([x 2]           ; binds x to 2
           [y x])         ; binds y to the value of the outer x, which is 5
       (define x 1)      ; this binding shadows the outer binding of x to 2
       (+ x y)))         ; computes the value of 1+5
```

6

□

The `let*` form

```
(let* ([var1 expr1]  
      ...  
      [varn exprn])  
  block)
```

- Similar with the `let` form, but with the following difference:
 - The scope of every local definition
[*var_i* *expr_i*]
is *expr_{i+1}*, ..., *expr_n*, and *block*.

Remark

This special form is equivalent to

```
(... (lambda (var1)  
      ...  
      (lambda (varn) body) exprn) ... expr1)
```

The `let*` form

Example

```
> (let* ([x 1] ; binds x to 1
         [y (+ x 1)] ; binds y to the value of (+ x 1), which is 2
         (+ y x)) ; computes the value of 2+1
```

3

Note that the following expression can not be evaluated

```
> (let ([x 1] ; binds x to 1
       [y (+ x 1)]) ; x is undefined here
     (+ y x))
```

x: unbound identifier in module in: x

Sections

- 3.8: Pairs and Lists
- 3.9: Vectors
- 3.12: Void and Undefined

from the *Racket Guide*