CSE 341: *Programming Languages*

Section AC with Nate Yazdani

agenda

- review: **`eval`**, **`quote`**, and **`quasiquote`**
- overview of some Ruby features
  - arrays
  - blocks
  - ranges
  - hashes
  - reflection

`eval`, **`quote`**, and **`quasiquote`**

- syntactically, Racket code can be thought of as a (possibly nested) list of tokens (*e.g.*, numbers, strings, and symbols)

- **`quote`**-ing a parenthesized expression gives you that list

- **`eval`** interprets such a list as Racket syntax for execution

- **`quasiquote`**-ing lets you **`unquote`** to evaluate *before* quoting a subexpression

`eval`, **`quote`**, and **`quasiquote`**

- … et code can be thought of as a … st of tokens (*e.g.*, numbers, strings,

“identifier values”

‘*e* same as **(quote** *e***)**

- **`quote`** … gives you that list

could also build your own lists

- **`eval`** interprets such a list as Racket syntax for e…

`*e* same as **(quasiquote** *e***)**

- **`quasiquote`**-ing lets you **`unquote`** to evaluate *before* quoting a subexpression

,*e* same as **(unquote** *e***)**

quotation

```
(define x 5)
(define y 7)

(+ 1 (* x y)) ; 36
(quote (+ 1 (* x y))) ; (list ‘+ 1 (list ‘* ‘x ‘y))
(eval (quote (+ 1 (* x y)))) ; 36

(+ x y #t) ; error!
(quote (+ x y #t)) ; (list ‘+ ‘x ‘y #t)

(+ x (* y 2)) ; 19
(quasiquote (+ x (unquote (* y 2)))) ; (list ‘+ ‘x 14)
```

Ruby

arrays

- most common data structure in Ruby
- comes with lots of built-in functionality
- dynamically typed, may store “heterogeneous” elements
- compared to other languages, Ruby arrays are
  - more permissive (fewer operations are errors)
  - more flexible
  - less efficient

arrays

- most common data structure in Ruby
- comes with lots of built-in functionality
- dynamically typed, may store “heterogeneous” elements

both good and bad

- compared to other languages, Ruby arrays are
  - more permissive (fewer operations are errors)
  - more flexible
  - less efficient

array operations

- length: a`.size` is the number of elements stored in a

- indexing:
  - if $i \geq 0$, then a`[`i`]` is the element stored at index i
  - if $i < 0$, then a`[`i`]` is a`[`a`.size + `i`]`

- construction:
  - `[`v_0`, `$\ldots$`, `v_n`]` is an array literal
  - `Array.new(`n`)` returns an n-element array of `nil`
  - `Array.new(`n`, `v`)` returns an n-element array of the result of v
  - `Array.new(`n`) { `e` }` returns an n-element array of the result of e for each position
  - `Array.new(`n`) { |`i`| `e` }` constructs an n-element array with the result of e for each position with the index bound to name i

array operations

- append: **`a + b = [a[0], a[1], …, b[0], b[1], …]`**

- add or remove from the back (*i.e.,* a**`[-1]`**):
  - a**`.push`** v adds v to the back of the array a
  - a**`.pop`** removes and returns the element at the back of the array a

- add or remove from the front (*i.e.,* a**`[0]`**):
  - a**`.shift`** removes and returns the element at the front of the array a, shifting other indices down by 1
  - a**`.unshift`** v adds v to the front of the array a, shifting all indices up by 1

arrays as stacks/queues

- push: a**`.push`** v
- pop: a**`.pop`**
- enqueue: a**`.push`** v
- dequeue: a**`.unshift`**

arrays as tuples

- a tuple (*e.g.*, in SML) stores a fixed number of values of different types

- in Ruby, an array serves that purpose just fine:
  `[true, “whoop whoop”, 42]`

arrays as sets

- set union: a_1 `|` a_2 returns an array of the distinct elements in either or both of a_1 and a_2
- set intersection: a_1 `&` a_2 returns an array of the distinct elements in both a_1 and a_2
- set difference: a_1 `-` a_2 returns an array of the distinct elements in a_1 but not in a_2

array slices

- an array slice constructs a new array from an interval of another

- a`[`i, n`]` is a slice of the array a from i to $i + n$ - 1

- similar syntax to update an array interval all at once
  - a`[`i, n`] = [`v_i, …, $v_{i+n\text{-}1}$`]`
  - *not* the same as creating a slice and then assigning that!

blocks

- similar to closures in some ways
  - has lexical scope
  - passed to method calls

- different in others
  - can’t store in a variable
  - might receive only some arguments (**`nil`** default)

```
object.method($v_0$, …, $v_n$) { |$x_0$, …, $x_n$| $e$ }

object.method($v_0$, …, $v_n$) do |$x_0$, …, $x_n$|
  $e$
end
```

iterators

- in Ruby, **`for`** and **`while`** loops are rarely used

- instead, call an *iterator* with a block for your “loop body”

```ruby
a = [1, 2, 3, 4]
a.map { |x| x * x } # [1, 4, 9, 16]
a.each { |x| puts x } # prints 1 to 4
a.inject(0) { |n, x| n + x } # 10
a.select { |x| x > 2 } # [1, 2]
a.any? { |x| x > 2 } # true
a.all? { |x| x > 2 } # false
```

iterators

don’t iterators kinda sound like higher-order functions?

- in Ruby, **for** and ed
- instead, call an *iterator* with a block for your “loop body”

by default, **`a.any?`** and **`a.all?`** checks if any/all elements are “true,” which in Ruby means neither **`false`** nor **`nil`**

```
a = [1, 2, 3, 4]
a.
a.                                          4
a.
a.
a.any? { |x| x > 2 } # true
a.all? { |x| x > 2 } # false
```

ranges

- a *range* is an efficient representation of a sequence of contiguous integers

- literal: i**`..`**j

- array conversion: r**`.to_a`**

- in some ways, can iterate over ranges like arrays, *e.g.*, r**`.map`**, r**`.each`**, and r**`.inject`**

hashes

- a *hash* (sometimes called a *dictionary*) uniquely maps some set of keys (h**`.keys`**) to values (h**`.values`**)

- literal: `{` k_1 `=>` v_1`,` …`,` k_n `=>` v_n `}`

- lookup: h`[`k`]`

- update: h`[`k`]` `=` v

- removal: h**`.delete`**`(`k`)`

- iteration: h**`.each`** `{` `|`k`,`v`|` e `}`

symbols

- like in Racket, a *symbol* is a “special string” that is more efficient to use after initial creation

- when Ruby code uses the same “constant string” frequently, then symbols are typically preferred

- literal: **`:woo`**, **`:woot_woot`**,
    - not **`:woot-woot`**, though

S

you can think of symbols as strings cached by the language runtime

- like in Racket, a *symbol* is a “special string” that is more efficient to use after initial creation

- when Ruby code uses the same “constant string” frequently, then symbols are typically preferred

- literal: **`:woo`**, **`:woot_woot`**,
  - not **`:woot-woot`**, though

duck typing

- in Ruby (much like Python), “duck typing” is a pervasive programming philosophy leveraging dynamic typing

- this practice roughly corresponds to using permissive, informal interfaces, so you can make one class (*e.g.*, **`Range`**) behave like another (*e.g.*, **`Array`**)

- can also check the actual class (*o*.**`class`**) and even get a list of supported methods (*o*.**`methods`**)

duck

“if it looks like a duck and quacks like a duck…”

- in Ruby (much like Python), “duck typing” is a pervasive programming philosophy leveraging dynamic typing

- this practice roughly corresponds to using permissive, informal interfaces, so you can make one class (*e.g.*, **`Range`**) behave like another (*e.g.*, **`Array`**)

- can also check the actual class (*o*.**`class`**) and even get a list of supported methods (*o*.**`methods`**) quick demo

Ruby exercises

write a Ruby method **`squares`** taking two arguments (say, a and b) and returning a hash mapping each integer i in $[a, b)$ to its square i^2

write a Ruby method **`print_hash`** to print out a hash `{` k_1 `=>` v_1`,` $\ldots$`,` k_n `=>` v_n `}` like the following:

k_1**:** v_1

$\ldots$

k_n**:** v_n