INF5171-20 Quiz no. 1
26 septembre 2017

Nom	

Soit les méthodes `foo` et `bar` ci-bas, définies dans une **extension** de la classe `Array`.

Indiquez ce qui sera affiché par `irb` pour chaque expression ci-bas

Note : `[*1..5] == [1, 2, 3, 4, 5]`.

---

1. Méthodes de la classe `Array`

```ruby
def foo
  res = Array.new(size)

  PRuby.pcall(
    0...size,
    lambda do |k|
      res[k] = yield(self[k])
    end
  )

  res
end
```

```ruby
def bar_ij( i, j )
  if i == j
    return yield(self[i]) ? 1 : 0
  end

  m = (i + j) / 2
  res1 = PRuby.future do
    bar_ij(i, m) { |x| yield(x) }
  end
  res2 = bar_ij(m+1, j) { |x| yield(x) }

  res1.value + res2
end

def bar
  return 0 if empty?

  bar_ij(0, size-1) { |x| yield(x) }
end
```

```ruby
#a.
>> [].foo { 1 }
=> []

# b.
>> [*1..5].foo { |x| x }
=> [1, 2, 3, 4, 5]

# c.
>> [*1..5].foo(&:even?)
=> [false, true, false, true, false]
```

```ruby
#d.
>> [].bar { 1 }
=> 0

# e.
>> [*1..5].bar { |x| x }
=> 5

# f.
>> [*1..5].bar(&:even?)
=> 2
```

2. Méthodes de la classe `Array` (bis)

Pour chaque variante ci-bas des méthodes `foo`, `bar` ou `bar_ij`, indiquez *i*) si cette variante permettra de produire le bon résultat et si oui *ii*) s'il s'agit d'une approche «intéressante».

a.
```ruby
def foo
  map { |x| PRuby.future { yield(x) } }.map(&:value)
end
```

i) oui; *ii*) pas très intéressante, car un très grand nombre de *threads* sont créés, avec une granularité très (trop!) fine ☹

b.
```ruby
def foo
  map { |x| PRuby.future { yield(x) }.value }
end
```

i) oui; *ii*) pas intéressante… car **strictement séquentielle** : on crée un future, mais à chaque fois on bloque tout de suite dessus, donc cela ne génère aucun parallélisme ☹

c.
```ruby
def foo
  res = Array.new
  (0...size).peach { |k| res << yield(self[k]) }

  res
end
```

i) non, car les ajouts dans `res` avec «<<» se feront dans un ordre arbitraire; *ii*) ☹

d.
```ruby
def foo
  res = Array.new
  (0...size).peach { |k| res[k] = yield(self[k]) }

  res
end
```

i) non : la réallocation dynamique du tableau pour augmenter sa taille pourrait ne pas fonctionner correctement en présence de *threads* multiples; *ii*) ☹

e.
```
def bar_ij( i, j )
  return yield(self[i]) ? 1 : 0 if i == j

  m = (i + j) / 2
  PRuby.pcall( -> { res1 = bar_ij(i,   m) { |x| yield(x) } },
               -> { res2 = bar_ij(m+1, j) { |x| yield(x) } } )

  res1 + res2
end
```

i) non, va générer une erreur car `res1` et `res2` ne sont pas définies dans le corps de `bar_ij`, donc leur affectation dans les blocs sera strictement locale (i.e., pas de capture); *ii*) ☹

f.
```
def bar
  nb = 0
  peach { |x| nb += 1 if yield(x) }

  nb
end
```

i) non : la variable `nb` est modifiée de façon non-atomique, donc il y situation de compétition; *ii*) ☹

g.
```
def bar
  pmap { |x| yield(x) ? 1 : 0 }.preduce(0, &:+)
end
```

i) oui; *ii*) plus ou moins : nécessite deux passes et la création d'une collection intermédiaire, or l'argument `final_reduce` de `preduce` permet de faire le même calcul, mais en une seule passe

```
def bar
  preduce( 0, final_reduce: ->(x, y) { x + y } ) do |s, x|
    yield(x) ? s + 1 : s
  end
end
```