(Séquence 3.2

factorielle

Définition en Scheme de `n!`

```
;;; fact : nat -> nat
;;; (fact n) rend la factorielle de n
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1))) ) )
```

La spécification permet d'écrire `(fact 0)`, `(fact 3)` mais pas `(fact -1)` car -1 est un entier relatif pas un entier naturel.

Autre définition (plus sûre ?)

```
;;; fact : int -> nat
;;; ERREUR si n n'est pas positif ou nul
;;; (fact n) rend la factorielle de n
(define (fact n)
  (if (or (not (integer? n)) (negative? n))
      (erreur 'fact "attend un entier naturel")
      (if (= n 0)
          1
          (* n (fact (- n 1))) ) ) )
```

La fonction rend toujours un résultat, pour `(fact 5)`, `(fact -5)`, ou même `(fact 2.5)` qui n'est même pas un entier relatif mais (techniquement) un flottant.
Inconvénient : le test supplémentaire est fait à chaque appel récursif

Autre définition améliorée

Tester la positivité en dehors de la récursion :

```
;;; factorielle : int  -> nat
;;; ERREUR si n n'est pas positif ou nul
;;; (factorielle n) rend la factorielle de n
(define (factorielle n)
  (if (negative? n)
      (erreur 'factorielle "attend un entier naturel")
      (fact n) ))
```

```
;;; fact: nat -> nat
;;; (fact n) rend la factorielle de n
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1))) ) )
```

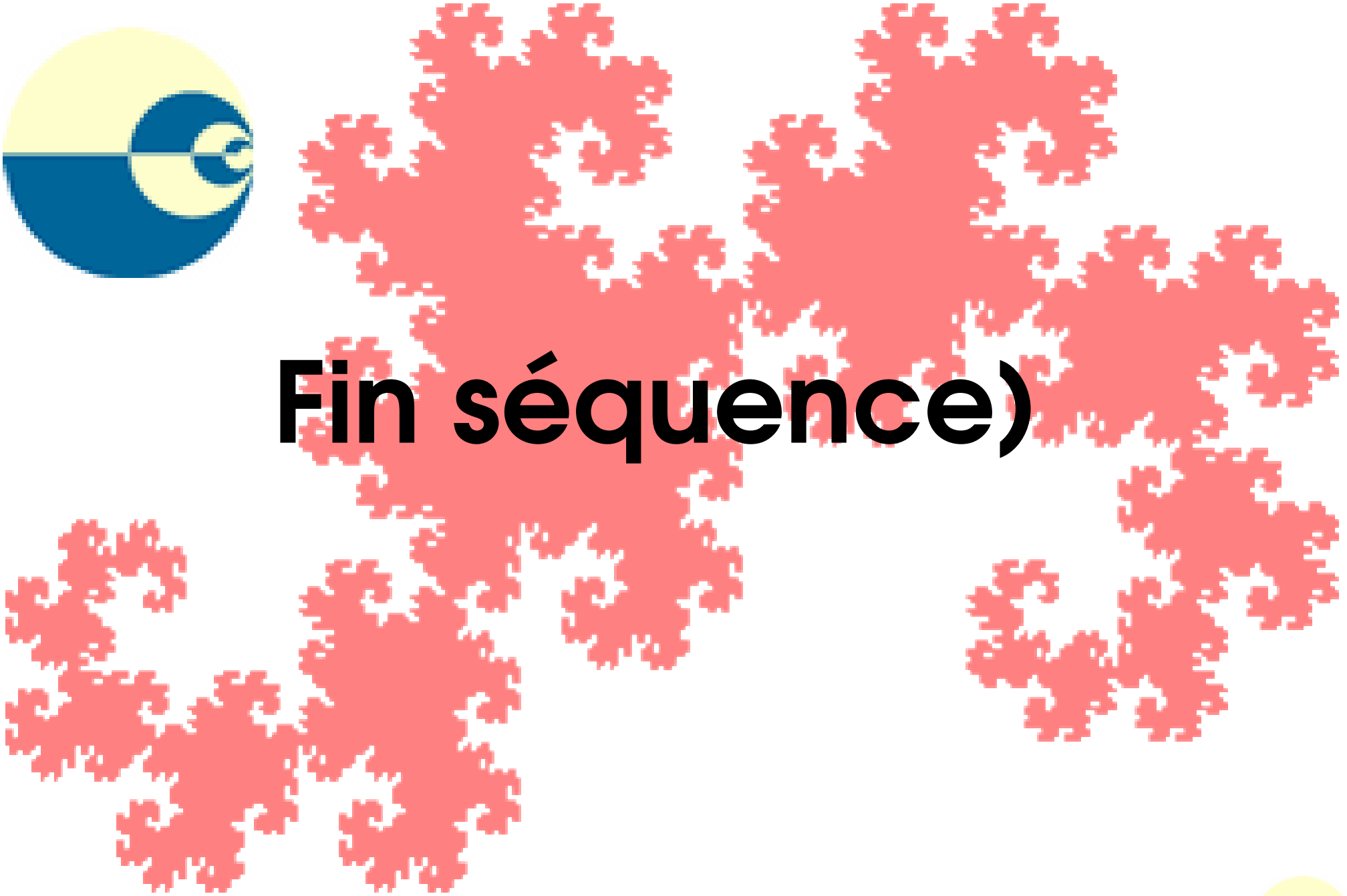
Autre définition améliorée plus sûre !

Afin de ne pas permettre d'appel direct à `fact` :

```
;;; factorielle: int  -> nat
;;; ERREUR si n n'est pas positif ou nul
;;; (factorielle n) rend la factorielle de n
(define (factorielle n)
  ;; fact: nat -> nat
  ;; (fact n) rend la factorielle de n
  (define (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1))) ) )

  (if (negative? n)
      (erreur 'factorielle
              "attend un entier naturel" )
      (fact n) ) )
```

La portée de `fact` est restreinte au corps de `factorielle`.



Fin séquence)