(Séquence 2.2

Booléens et connecteurs

Condition

Les expressions conditionnelles sont construites avec les opérations logiques de négation, de conjonction et de disjonction.

- Négation `not` : une *fonction prédéfinie*, une fonction prédéfinie
- Conjonction `and` : une *forme spéciale*
- Disjonction `or` : une *forme spéciale*

```
;;; Autre définition encore de valeur-absolue
(define (valeur-absolue x)
  (if (not (negative? x ) )
      x
      (- x) ) )
```

`and` et `or` : des formes spéciales

Syntaxe (les règles de grammaire)

<conjonction> → **(and** *<expression>**)

<disjonction> → **(or** *<expression>**)

Évaluation (gauche vers droite) de `and` et `or` :

- Pour `and`, on continue d'évaluer les expressions tant qu'elles valent Vrai.
- Pour `or`, on continue d'évaluer les expressions jusqu'à en trouver une Vraie.

Dans les deux cas les calculs s'arrêtent lorsque la valeur finale est déterminée. La valeur finale est celle de la dernière expression évaluée. Ainsi

```
(and (< 1 2) (> 5 6) (= 1 1)) → #f
(and (< 1 2) (< 5 6)) → #t
(or  (< 1 2) (> 5 6)) → #t
(or  (< 2 1) (> 5 6) (< 1 1)) → #f
```

Vrais et faux

En Scheme, tout ce qui n'est pas faux (c'est-à-dire `#f`) est vrai ! Autrement dit, toutes les valeurs sont considérées comme représentant vrai mais une seule valeur représente faux qui est `#f`.

```
;;; a-un-double: Nombre + string -> Nombre + String
;;; (a-un-double v) retourne la valeur v doublée si v es
;;; nombre ou une chaine. Retourne faux sinon.
(define (a-un-double v)
  (or (and (number? v)
           (* 2 v) )
      (and (string? v)
           (string-append v v) ) ) )
(a-un-double 3)         → 6
(a-un-double "cou")     → "coucou"
(a-un-double #t)        → #f
```

D'ailleurs, la fonction `a-un-double` est appelée un ***semi-prédicat***.

Prédicat et semi-prédicat

Si l'on veut un résultat qui ne soit pas seulement vrai ou faux mais qui soit l'un des deux booléens possibles alors on peut utiliser le prédicat `boolify` (qui est souvent utilisé dans des tests).

```
;;; boolify: Valeur -> bool
;;; (boolify valeur) prend un valeur et renvoie le boolé
;;; sa valeur logique.
(define (boolify v)
   (and v #t) )
(boolify 3)          → #t
(boolify "cou")      → #t
(boolify #t)         → #t
```

Plutôt que `(and v #t)`, on peut aussi écrire `(not (not v))`.

Rappel carte de référence

```
;;; number?: Valeur -> bool
        (number? 32.45)          → #t
        (number? "essai")        → #f

;;; positive?: Nombre -> bool
        (positive? 32.45)        → #t
        (positive? "essai")      ERREUR
```

Ce que l'on veut écrire :

```
;;; nombre-positif?: Valeur -> bool
        (nombre-positif? 32.45)   → #t
        (nombre-positif? "essai") → #f
```

Fonction `nombre-positif?` (essai 1)

```
;;;                    Une définition fausse:
(define (nombre-positif? v)                          FAUX
   (and (positive? v) (number? v)) )

(nombre-positif? 32.45)      → #t

(nombre-positif? "essai")                            ERREUR
```

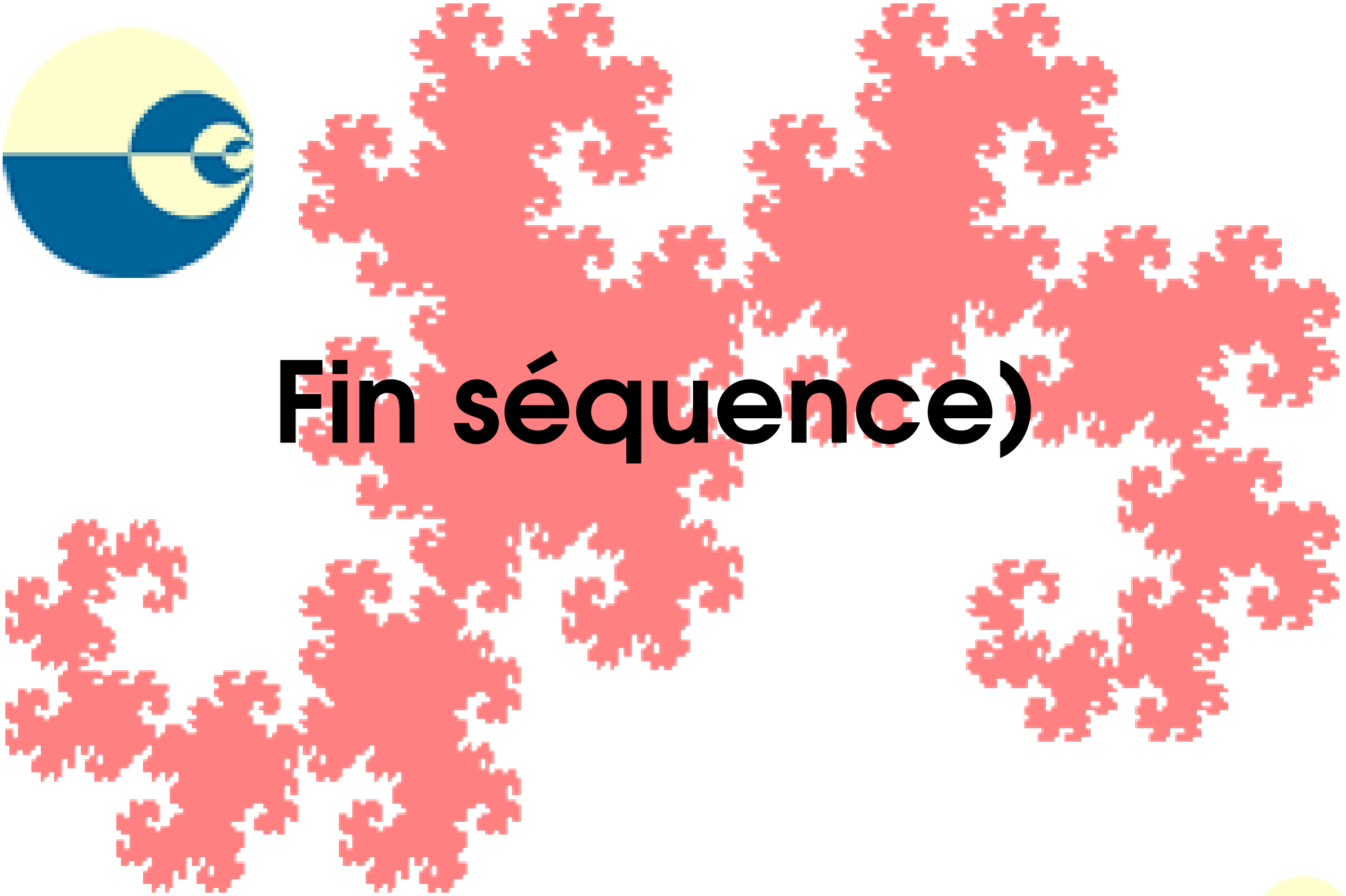
Fonction `nombre-positif?` révisée

Une définition correcte:

```
;;;
(define (nombre-positif? v)
   (and (number? v) (positive? v)) )

(nombre-positif? 32.45)    → #t

(nombre-positif? "essai") → #f
```



Fin séquence)