MOOC
Programmation récursive

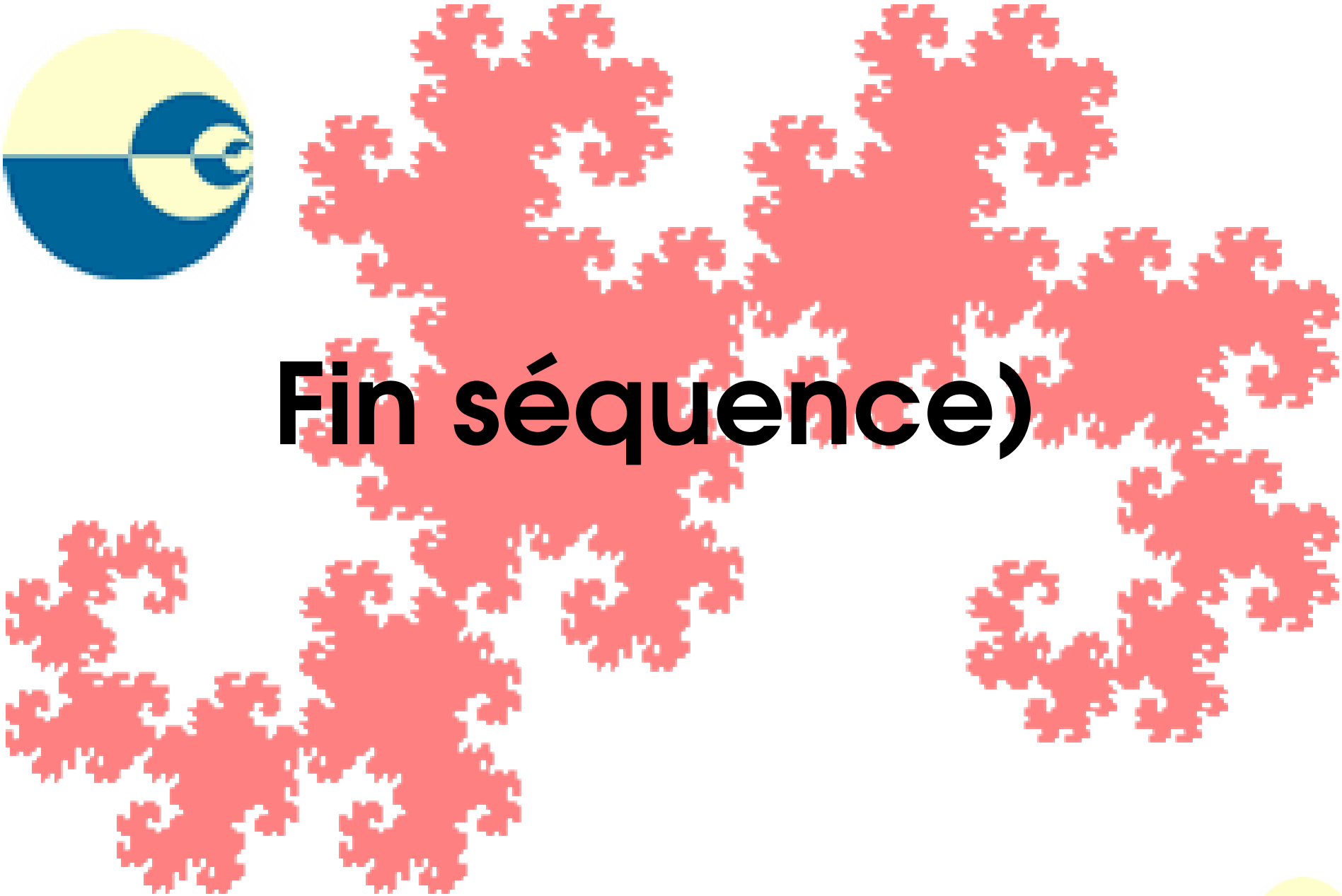
Christian Queinnec
Professeur à l'UPMC

Remerciements

Remerciements à Anne Brygoo, Titou Durand, Karine Heydemann, Pascal Manoury, Christophe Marsala, Frédéric Peschanski, Michèle Soria qui ont animé et enrichi ce cours pendant toutes ces années.

Les vidéos sont dues à Thomas Baspeyras et Daniel Tanasijevic du Centre de Production Multimédia de l'UPMC. Quelques-unes ont été tronçonnées en séquences plus courtes par mes soins.

Certains éléments de l'iconographie utilisée dans ce document ont été réalisés par Magnus Emil Liisberg Helding. Le logo du MOOC est dû à Anne Brygoo.



Fin séquence)

(Séquence 0.0

Objectifs généraux du MOOC

Objectifs du MOOC

- Initiation à la programmation récursive (en Scheme),
- Lecture et écriture de fonctions récursives,
  - Récursion sur les entiers
  - Récursion sur les listes
  - Récursion sur les arbres
- Fonctionnement d'un évaluateur d'expressions qui prend un texte et produit un résultat

Compétences

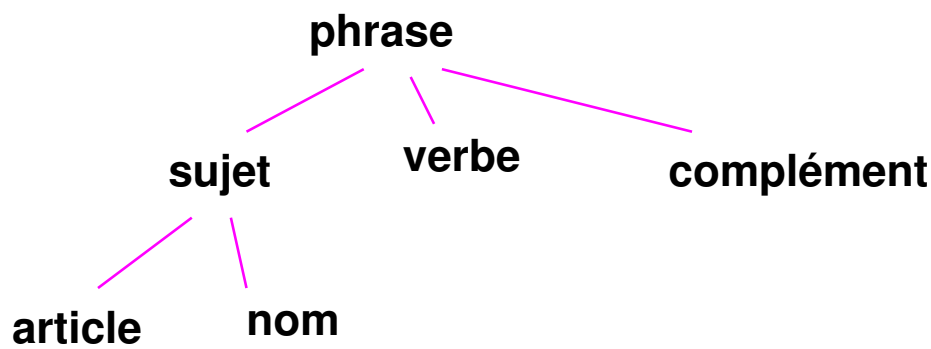
- Lecture, écriture d'algorithmes récursifs
- Connaissance de structures de données de base
- Compréhension du processus d'évaluation d'un programme du texte à sa valeur

Compétences

- Lecture, écriture d'algorithmes récursifs
- Connaissance de structures de données de base
- Compréhension du processus d'évaluation d'un programme du texte à sa valeur

3!

(a abaisser abajoue ...)



Déroulement

- Saison 1 (cours 1-4) :
  - fondements linguistiques
  - récursion sur entiers puis listes

Déroulement

- Saison 1 (cours 1-4) :
  - fondements linguistiques
  - récursion sur entiers puis listes
- Saison 2 (cours 5-8) :
  - récursion sur arbres binaires,
  - arbres binaires de recherche,
  - arbres généraux

Déroulement

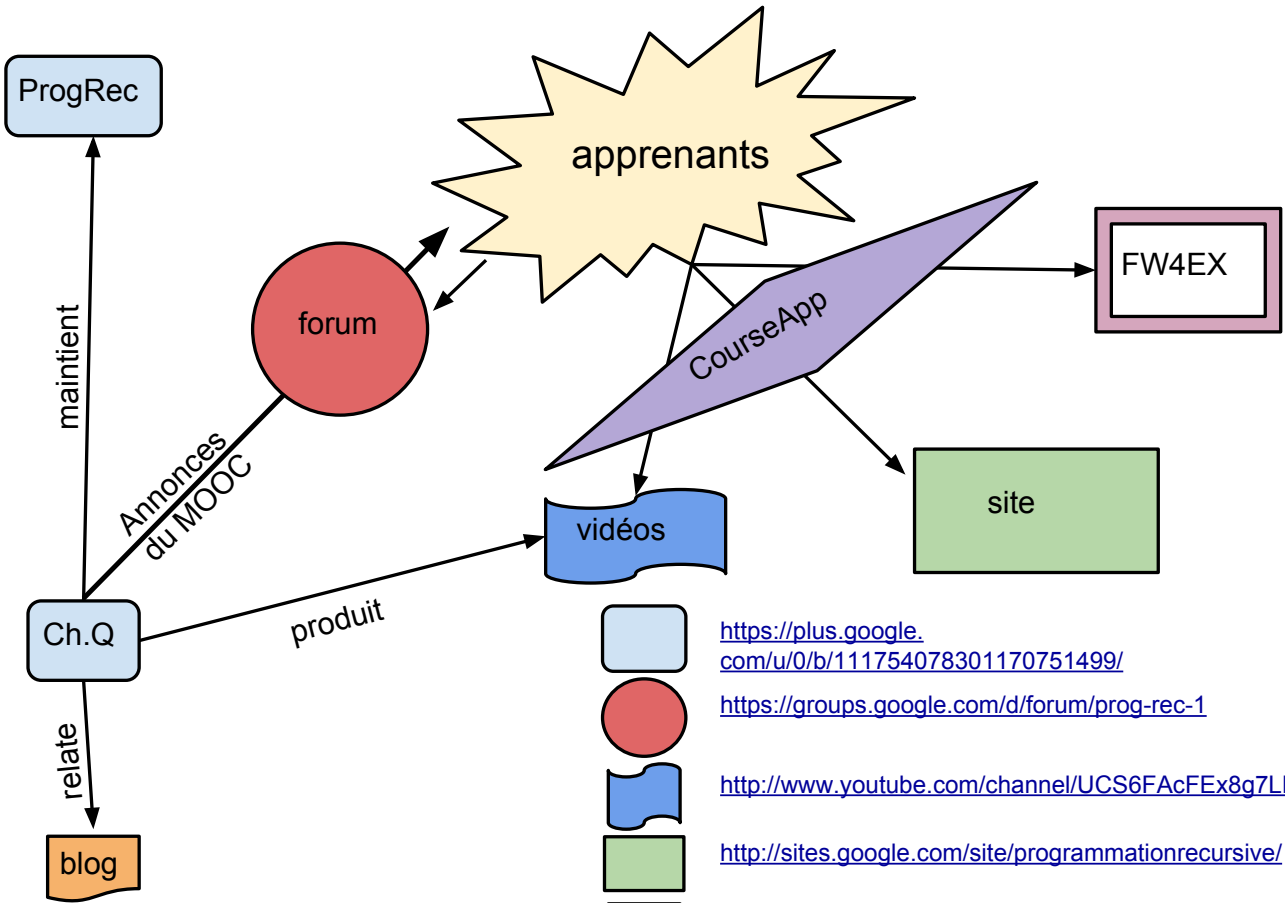
- Saison 1 (cours 1-4) :
  - fondements linguistiques
  - récursion sur entiers puis listes
- Saison 2 (cours 5-8) :
  - récursion sur arbres binaires,
  - arbres binaires de recherche,
  - arbres généraux
- Saison 3 (cours 9-10) :
  - S-expressions et programmes
  - processus d'évaluation

Moyens

- le cours, les vidéos et bien d'autres documents
- les exercices à correction automatisée
- le forum où s'entraider :
  - Questions/remarques sur les cours
  - Questions/remarques sur l'organisation, l'infrastructure

Moyens

La constellation du MOOC Programmation récursive



https://plus.google.com/u/0/b/111754078301170751499/

https://groups.google.com/d/forum/prog-rec-1

http://www.youtube.com/channel/UCS6FAcFEx8g7LNapQfz_lig

http://sites.google.com/site/programmationrecursive/

http://programmation-recursive.blogspot.fr/

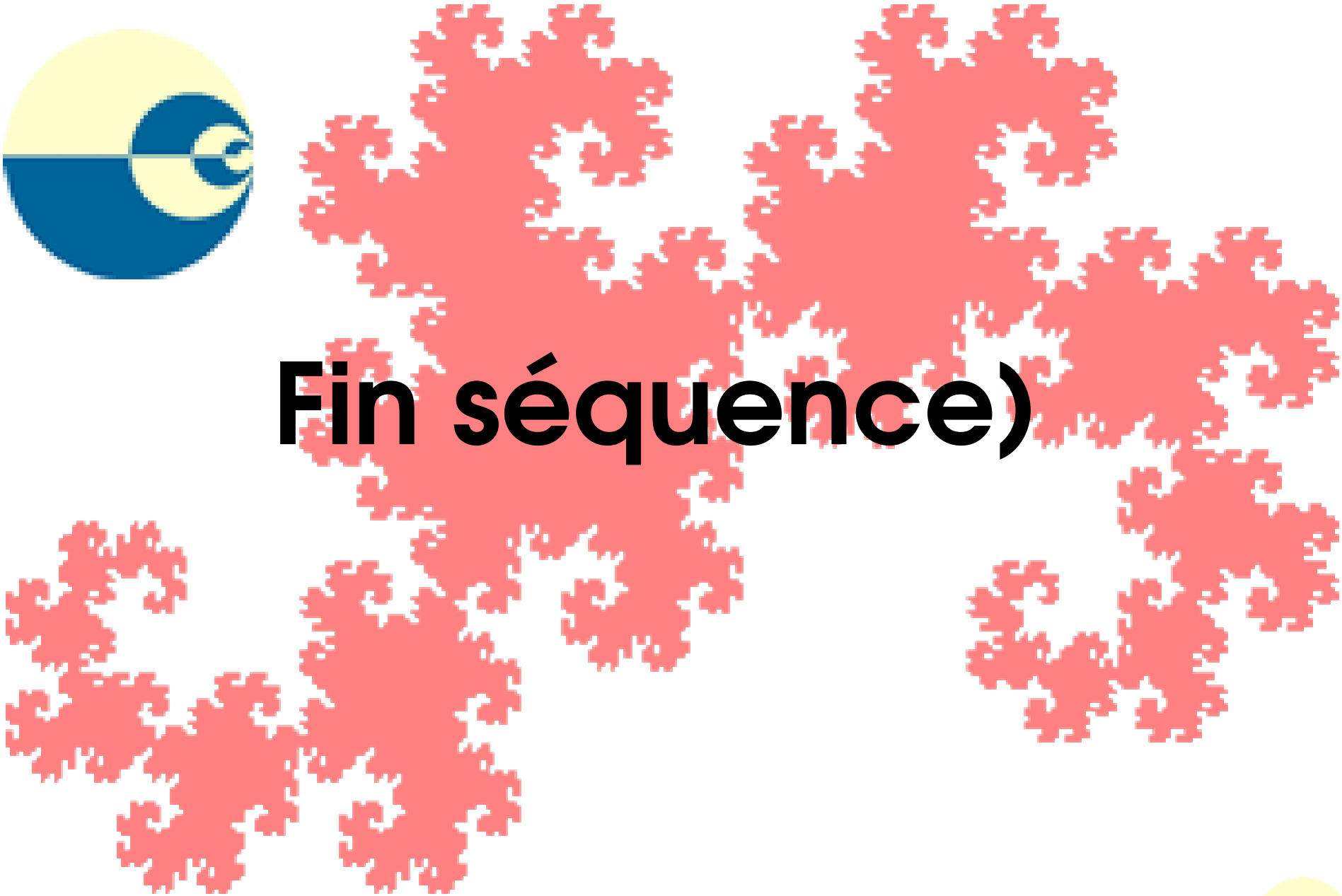
http://programmation-recursive-1.appspot.com/

http://mooc-li101-2014fev.fw4ex.org/

2014 février 03

Recommandations

- C'est la seconde édition du MOOC Programmation récursive
- Qui comporte de nouvelles caractéristiques expérimentales
- en quête de propositions d'amélioration



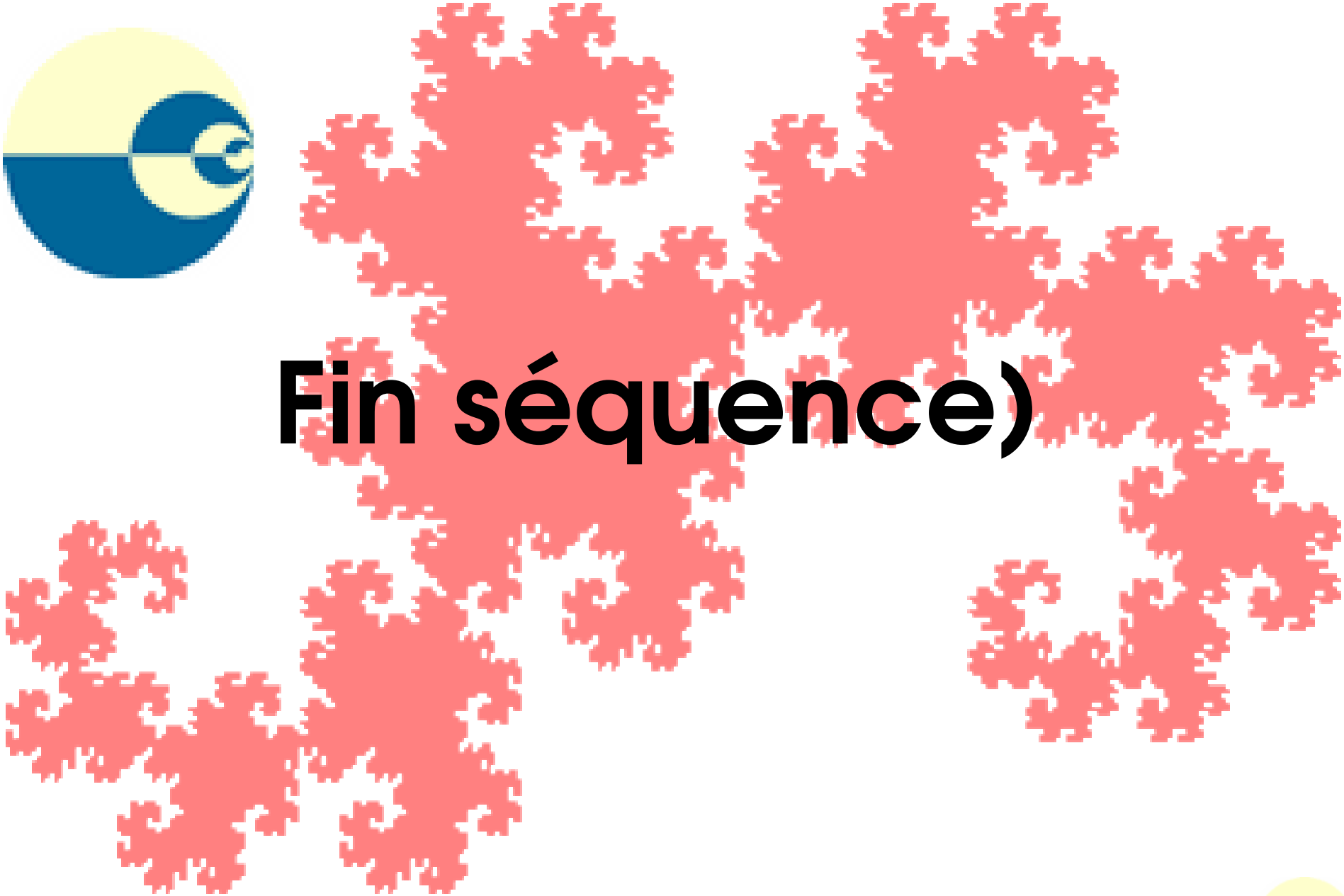
Fin séquence)

(Séquence 1.0

= Plan semaine 1

Plan semaine 1

- Étude des expressions
  - Lecture-écriture d'une expression
  - Différentes écritures linéaires
  - Écriture des expressions en Scheme
- Premiers pas en Scheme
  - Mécanismes d'un langage
  - Application de fonction
  - Définition de fonction : `define`
  - Alternative : `if`
- Premiers exercices en Scheme



Fin séquence)

(Séquence 1.1

Syntaxe

Expression

- En mathématique ou en informatique

  *Une **expression** est une formule exprimant une façon de calculer une valeur.*

- Comment écrire des formules ?
- Comment calculer des formules ?

Écriture d'une expression

$$3x^2 + 2x + 4$$

Caractéristiques sous-entendues :

- pas de signe pour la multiplication
- Écriture 2D
- Ordre de priorité des opérateurs sous-entendus
- avec toutes les parenthèses et tous les opérateurs :

$$(3 * (x\hat{}2)) + ((2 * x) + 4)$$

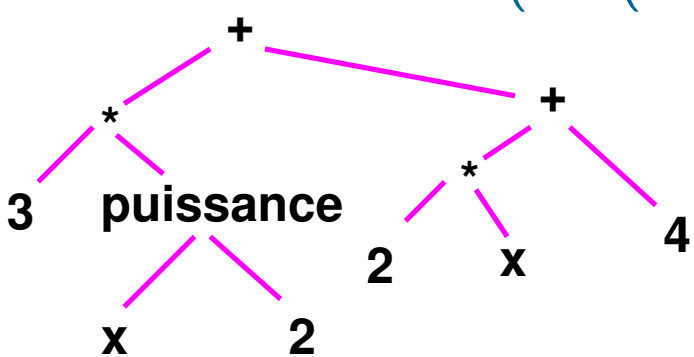
Écriture d'une expression

$$3x^2 + 2x + 4$$

Caractéristiques sous-entendues :

- pas de signe pour la multiplication
- Écriture 2D
- Ordre de priorité des opérateurs sous-entendus
- avec toutes les parenthèses et tous les opérateurs :

$$(3 * (x\hat{}2)) + ((2 * x) + 4)$$



Syntaxe

Attention aux ambiguïtés :

- a + b*c peut se comprendre comme :
  - (a + b)*c
  - a + (b*c)
- a - b + c peut se comprendre comme :
  - (a - b) + c
  - a - (b + c)
- a / b / c peut se comprendre comme :
  - (a / b) / c
  - a / (b / c)

Différentes écritures linéaires

- en **préfixe**, l'opérateur précède ses opérandes

OP a b

Exemple : $\log\ e$

- en **infixe**, un opérateur binaire se situe entre ses arguments

a OP b

Exemple : $x + 3$ *Exemple :* $A \to B$

- en **suffixe**, l'opérateur suit ses opérandes

a b OP

Exemple : f' *Exemple :* $n!$

Une expression doit être non ambiguë

Lemme de Lukasiewicz : si l'on connait l'**arité** (c'est-à-dire le nombre d'opérandes) des opérateurs, on peut retrouver l'expression à partir de l'une des notations postfixe et préfixe.

3 2 + 5 *

Une expression doit être non ambiguë

Lemme de Lukasiewicz : si l'on connait l'**arité** (c'est-à-dire le nombre d'opérandes) des opérateurs, on peut retrouver l'expression à partir de l'une des notations postfixe et préfixe.

$$3\ 2\ +\ 5\ * \equiv 5\ 5\ *$$

Une expression doit être non ambiguë

Lemme de Lukasiewicz : si l'on connait l'**arité** (c'est-à-dire le nombre d'opérandes) des opérateurs, on peut retrouver l'expression à partir de l'une des notations postfixe et préfixe.

$$3\ 2\ +\ 5\ *\ \equiv\ 5\ 5\ *\ \equiv\ 25$$

Les deux notations **postfixe** et **préfixe** représentent de façon non-ambiguë l'expression.

$$*\ -\ 5\ 1\ 3$$

Une expression doit être non ambiguë

Lemme de Lukasiewicz : si l'on connait l'**arité** (c'est-à-dire le nombre d'opérandes) des opérateurs, on peut retrouver l'expression à partir de l'une des notations postfixe et préfixe.

$$3\ 2\ +\ 5\ *\ \equiv\ 5\ 5\ *\ \equiv\ 25$$

Les deux notations **postfixe** et **préfixe** représentent de façon non-ambiguë l'expression.

$$*\ -\ 5\ 1\ 3\ \equiv\ *\ 4\ 3$$

Une expression doit être non ambiguë

Lemme de Lukasiewicz : si l'on connait l'**arité** (c'est-à-dire le nombre d'opérandes) des opérateurs, on peut retrouver l'expression à partir de l'une des notations postfixe et préfixe.

$$3\ 2\ +\ 5\ *\ \equiv\ 5\ 5\ *\ \equiv\ 25$$

Les deux notations **postfixe** et **préfixe** représentent de façon non-ambiguë l'expression.

$$*\ -\ 5\ 1\ 3\ \equiv\ *\ 4\ 3\ \equiv\ 12$$

En revanche, la notation **infixe** est ambiguë et il est nécessaire pour la rendre non-ambiguë d'utiliser des parenthèses.

Expressions Scheme

Caractéristiques de l'écriture des expressions en Scheme :

- écriture préfixe,
- complètement parenthésée,
- les parenthèses entourent toute l'expression,
- les séparateurs sont les espaces et les retours à la ligne.

```
6/2 + 5*6
```

Expressions Scheme

Caractéristiques de l'écriture des expressions en Scheme :

- écriture préfixe,
- complètement parenthésée,
- les parenthèses entourent toute l'expression,
- les séparateurs sont les espaces et les retours à la ligne.

$$6/2 + 5*6 \equiv \text{(+ (/ 6 2) (* 5 6))}$$

Expressions Scheme

Caractéristiques de l'écriture des expressions en Scheme :

- écriture préfixe,
- complètement parenthésée,
- les parenthèses entourent toute l'expression,
- les séparateurs sont les espaces et les retours à la ligne.

$$6/2 + 5*6 \equiv \text{(+ (/ 6 2) (* 5 6))}$$

$$6/2 + 5*6*|-2|$$

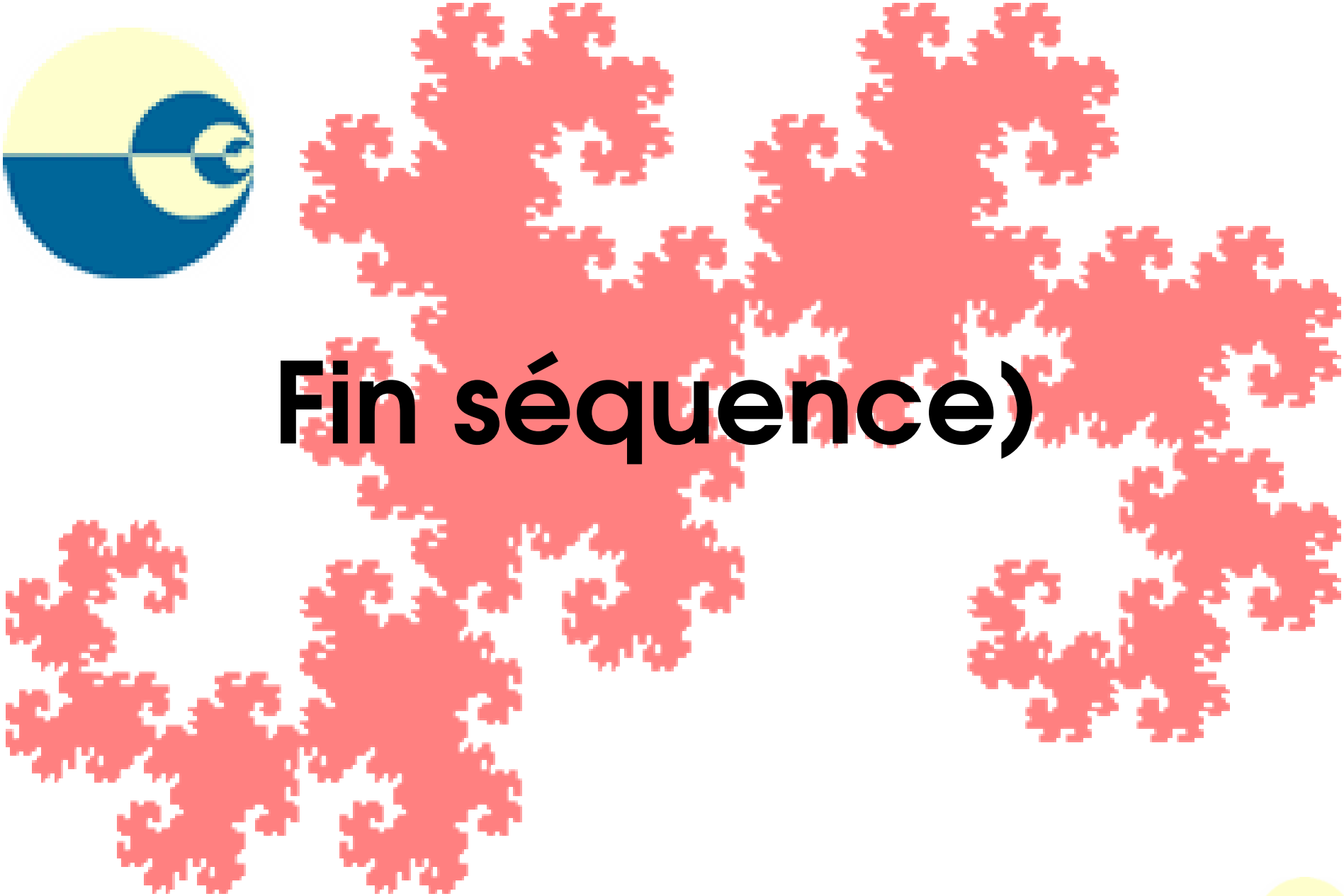
Expressions Scheme

Caractéristiques de l'écriture des expressions en Scheme :

- écriture préfixe,
- complètement parenthésée,
- les parenthèses entourent toute l'expression,
- les séparateurs sont les espaces et les retours à la ligne.

$$6/2 + 5*6 \equiv (+ (/ 6\ 2)\ (*\ 5\ 6))$$

$$6/2 + 5*6*|-2| \equiv (+ (/ 6\ 2)\ (*\ 5\ 6\ (abs\ -2)))$$



Fin séquence)

(Séquence 1.2

Langage

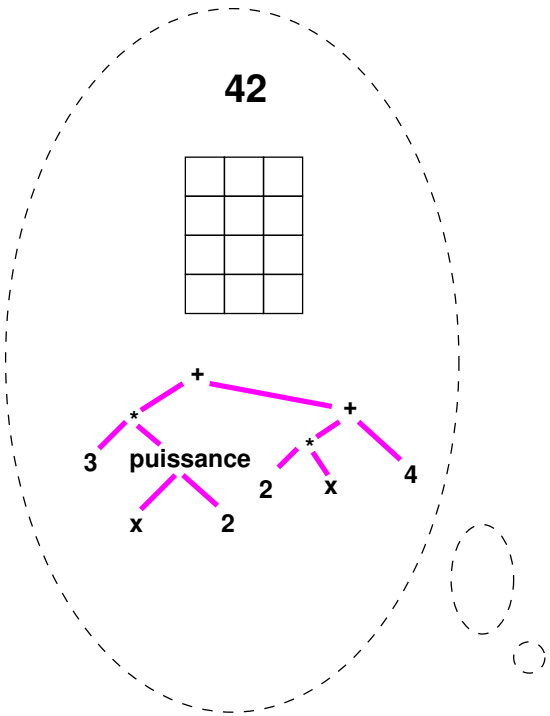
Premiers pas en Scheme

Quelques concepts d'un langage

- variable
- fonction
- forme spéciale (ou mots clés)
  - La définition de fonction : `define`
  - L'alternative : `if`

Point de vue

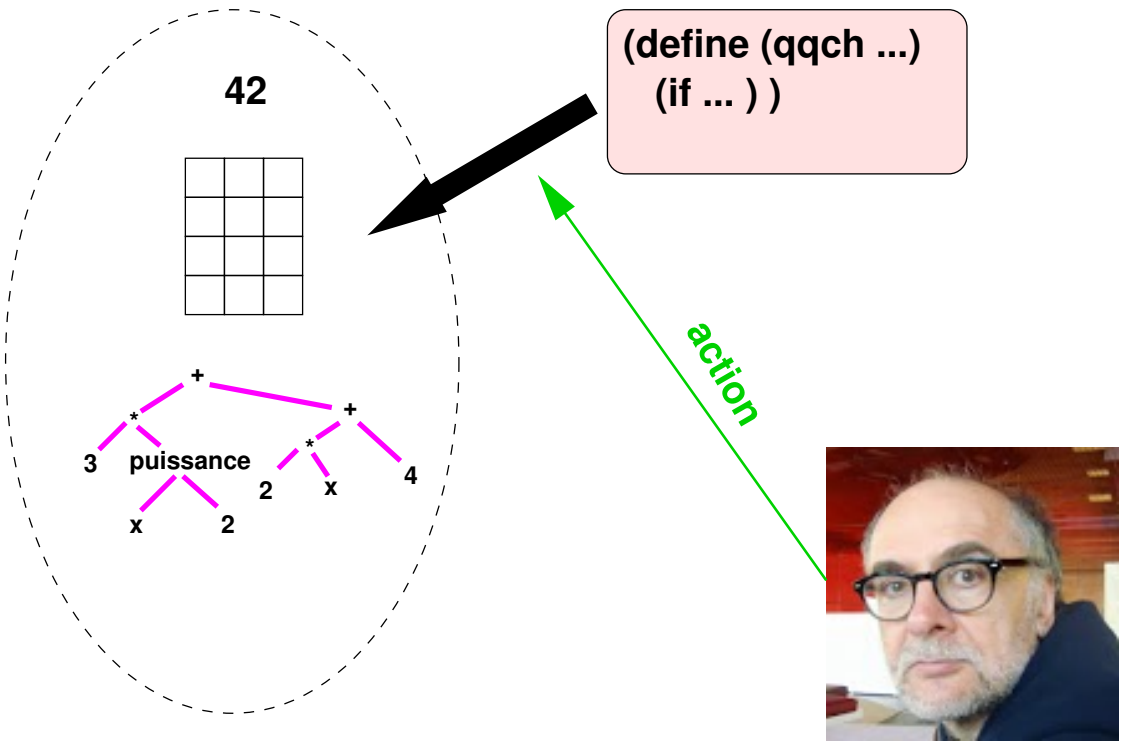
Pour construire quelque chose que l'on imagine (nombre, texte, matrice, arbre, etc.) on écrit des programmes :




```
(define (qqch ...)
  (if ... ) )
```

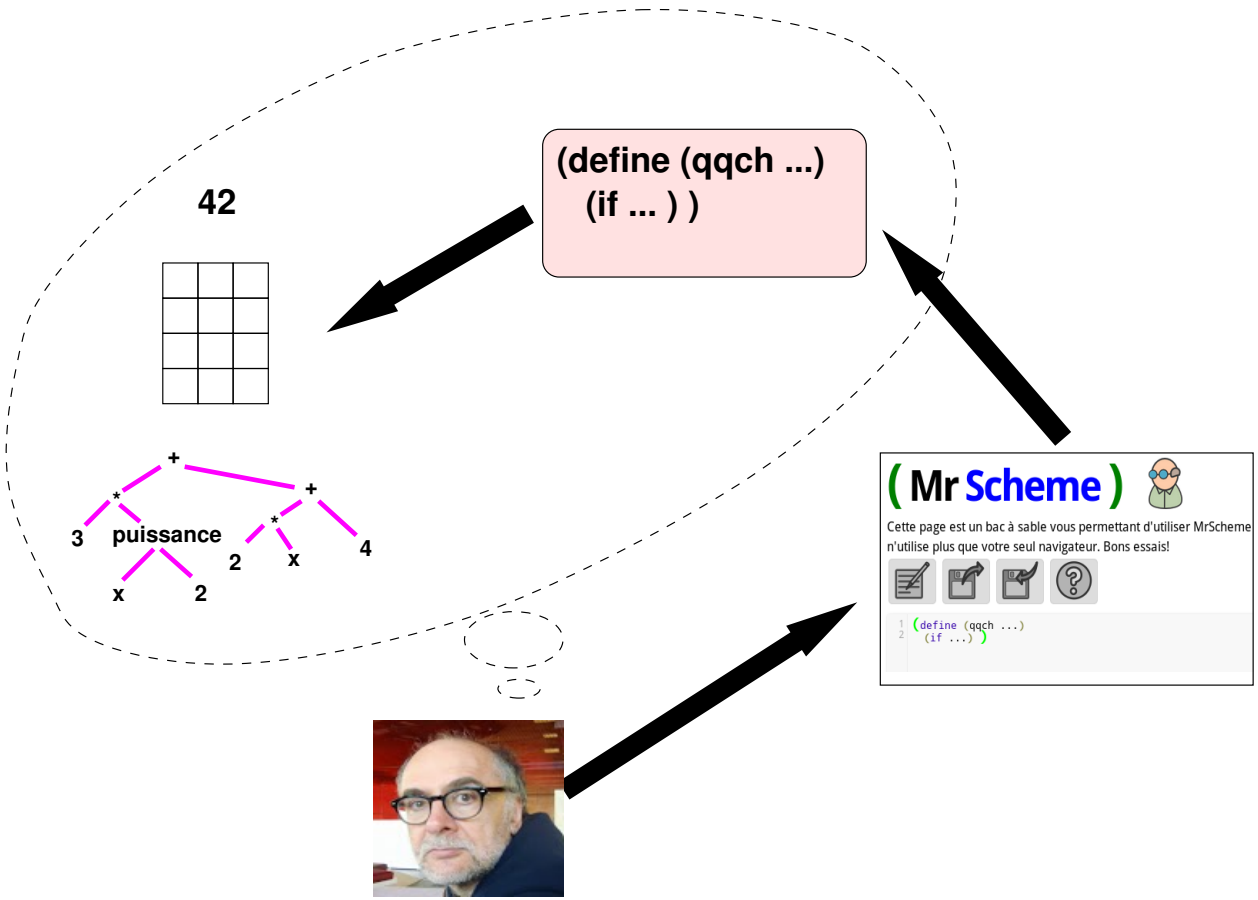
Point de vue

Pour lancer l'exécution du programme, on passe par un navigateur, un système d'exploitation (encore des programmes)



Point de vue

Mais un programme est lui-même un objet construit à l'aide d'un environnement de développement (encore des programmes)



Les mécanismes d'un langage

Le texte d'un programme permet de

- demander à l'ordinateur le résultat d'un calcul
- noter et d'organiser ses idées
- partager des connaissances avec d'autres personnes

Pour cela, un langage de programmation permet de

- manipuler des concepts prédéfinis (données ou traitement)
- construire de nouveaux concepts par composition

À votre disposition

Pour programmer, vous disposez

- d'objets ***primitifs***,
- d'une bibliothèque de fonctions qui vous permet, dès le début, de bénéficier du travail d'autres personnes
  - primitives
  - prédéfinies mais non primitives
- d'une grammaire qui vous donne les règles syntaxiques de construction
- et d'une carte de référence décrivant la sémantique des fonctions prédéfinies.

Objets primitifs

Le langage permet de manipuler :

- des constantes entières,

```
42, -5
```

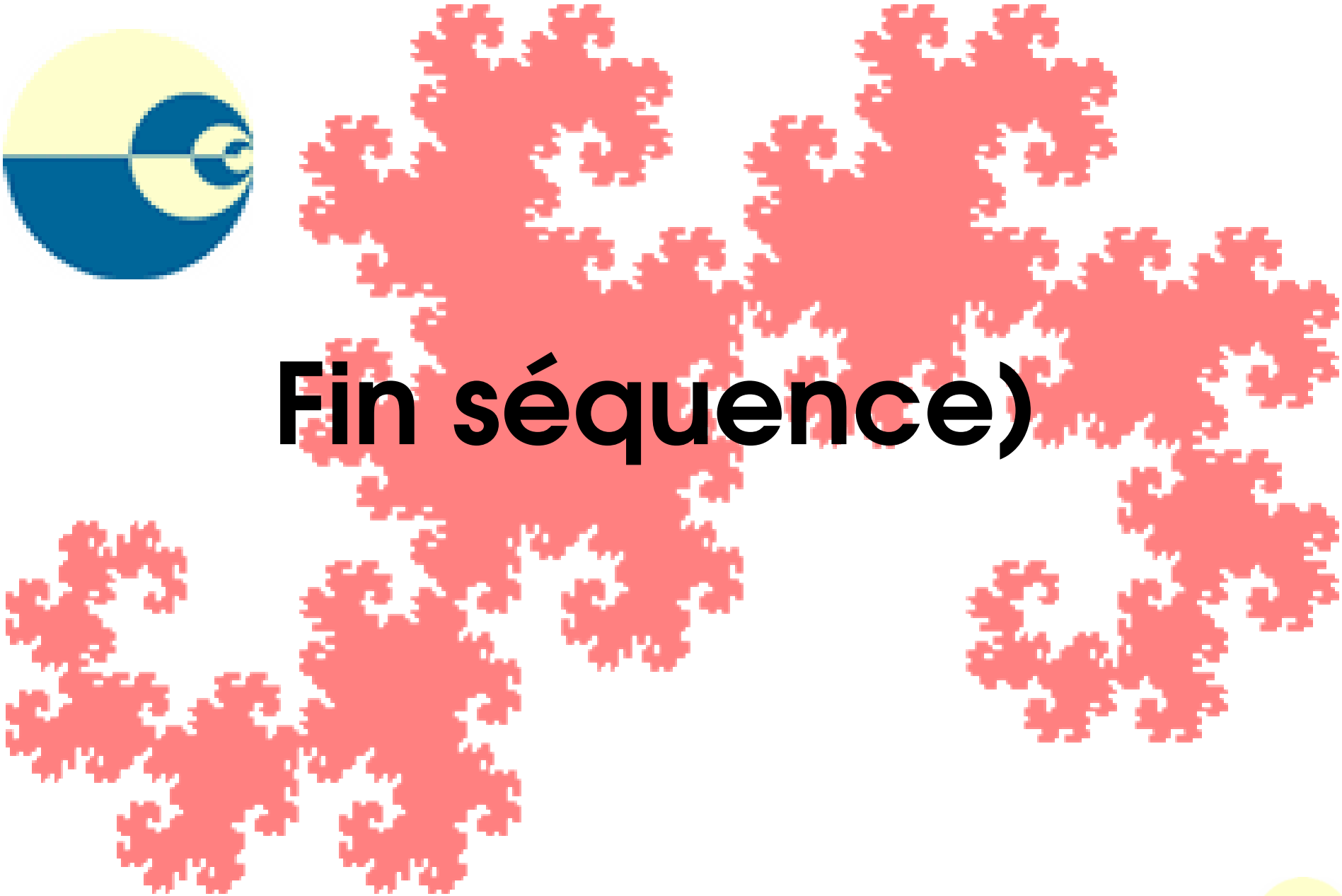
- des constantes flottantes,

```
2.03, -3.14e+3
```

- les valeurs booléennes, `#t` pour *true* et `#f` pour *false*
- les constantes chaînes de caractères,

```
"Je suis une chaîne!"
```

et tout le reste sera construit.



Fin séquence)

(Séquence 1.3

Bibliothèque

Fonctions prédéfinies

Pour pouvoir utiliser une fonction prédéfinie, il faut connaître sa ***spécification*** :

- son nom,
- son type,
- ce qu'elle calcule,
- la signification de ses arguments.

Exemple de fonction prédéfinie (dans carte de référence) :

```
;;; quotient: int*int -> int
;;; (quotient n1 n2) retourne le quotient de la
;;; division euclidienne de n1 par n2
;;; HYPOTHÈSE: n2 non nul
```

Ainsi `(quotient 17 3)` vaut 5 (le reste est 2).

Prédicat

Un ***prédicat*** est une fonction qui a, pour résultat, un booléen

```
;;; number?: Valeur -> bool

;;; positive?: Nombre -> bool
```

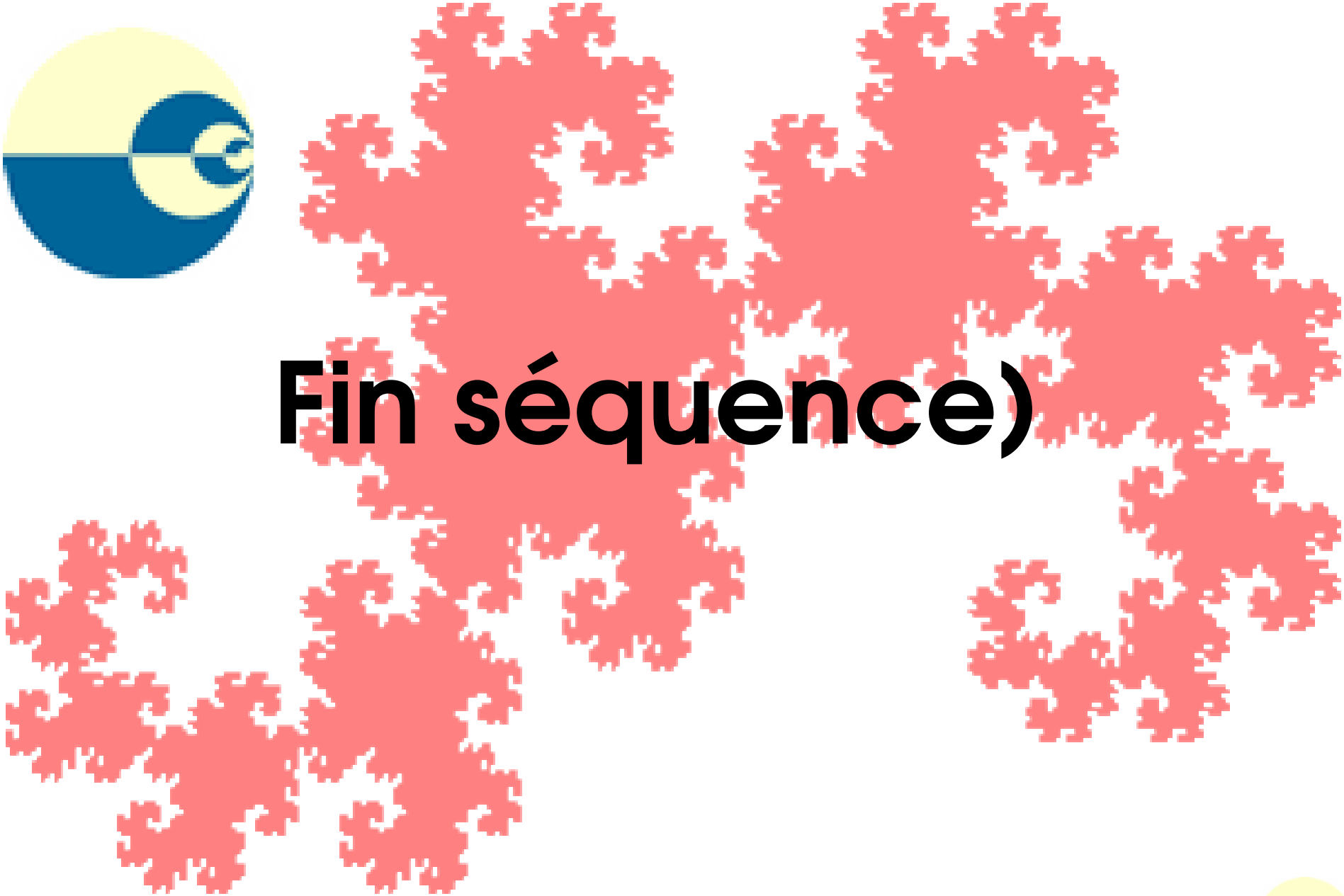
Ainsi `(number? -4)` comme `(number? 3.14)` est vrai mais pas `(number? "blabla")`.

ATTENTION ! `(positive? -3)` est faux
mais `(positive? "blabla")` est erroné !

Contenu de la bibliothèque

Tout ce qui est dans la carte de référence

- fonctions booléennes
- prédicats de reconnaissance
- prédicats sur nombres
- fonctions arithmétiques
- fonctions sur chaînes de caractères
- fonctions sur listes
- fonctions sur arbres binaires
- fonctions sur arbres généraux



Fin séquence)

(Séquence 1.4

Grammaire

Grammaire

Grammaire : ensemble des règles à suivre pour composer un texte

Une expression est **syntaxiquement** correcte si elle respecte la grammaire

Toute expression syntaxiquement correcte n'est pas forcément **sémantiquement** correcte !

Erreur de syntaxe : `(define pi)`
Erreur de type : `(+ 1 "un")`
Erreur téléologique : `(+ 1 2 10)` au lieu de `(+ 12 10)`

Règles de composition

<programme> → *<expression-ou-définition>**
<expression-ou-définition> → *<expression>*
<expression-ou-définition> → *<définition>*
<expression> → *<constante>*
<variable>
<forme-spéciale>
<application>

L'étoile marque une répétition quelconque, un signe « + » marque une répétition quelconque mais non vide.

Application de fonction

<application> → **(** *<fonction>* *<argument>** **)**
<fonction> → *<expression>*
<argument> → *<expression>*

et par exemple :

```
(+ (quotient 7 2) (* 5 6 2) 8 (abs -3))

(positive? (abs -3))
```

Définition de fonction

La définition de fonction permet :

- de définir de nouvelles fonctions

$$x \to x^2$$

- de nommer ces nouvelles fonctions.

$$\text{carré} : x \to x^2$$

- pour pouvoir ensuite les appliquer

$$\text{carré}(a+1)$$

Le pourquoi de cette grammaire

Ainsi définir que carré$(x) = x^2$ s'écrit :

- On enserre entre parenthèses :

```
( carré(x) = x^2 )
```

- On place un mot en tête pour indiquer ce que c'est :

```
(define carré(x) = x^2 )
```

- On convertit en polonaise parenthésée préfixée

```
(define (carré x) = (* x x))
```

- On élimine le bruit syntaxique

```
(define (carré x) (* x x))
```

- et finalement, l'on obtient :

```
;;; carre: Nombre -> Nombre
;;; (carre x) rend le carré du nombre x
(define (carre x)
   (* x x) )
```

Grammaire des définitions

<définition> $\rightarrow$
(define (*<nom-fonction>* *<variable>** **)** *<corps>* **)**
<corps> $\rightarrow$
<définition>* *<expression>*

Quatre étapes pour une définition

1. Donner la spécification de la fonction
2. Inventer la composition d'opérations menant au résultat cherché : l'algorithme,
3. Écrire l'algorithme en une expression syntaxiquement correcte
4. Tester la fonction en automatisant les tests avec `verifier`

Ainsi

```
;;; carre: Nombre -> Nombre
;;; rend le carré d'un nombre
(define (carre x)
   (* x x) )
(verifier carre
   (carre 1)    => 1
   (carre -2)   => 4
   (carre 2.1) => 4.41 )
```

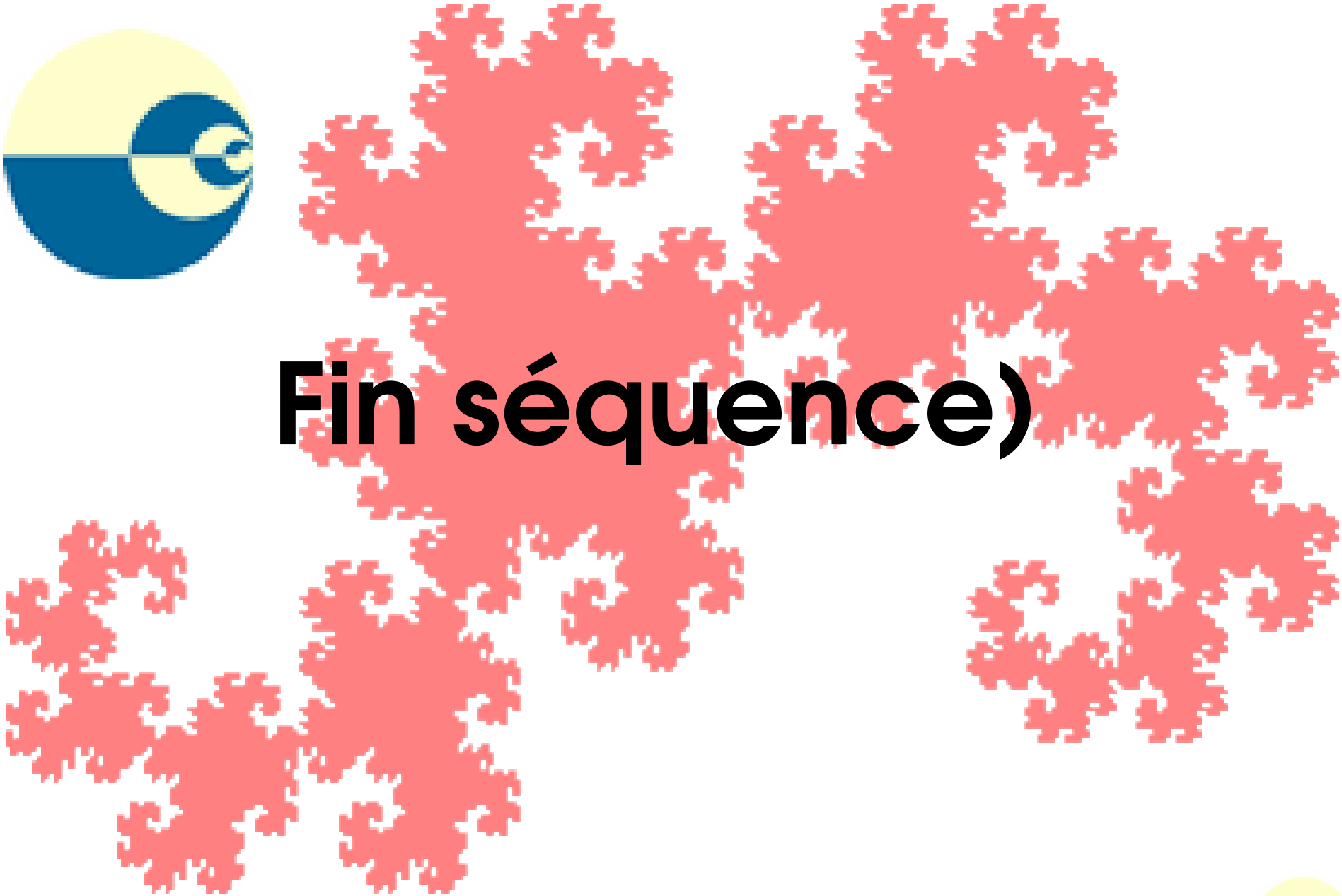
Alternative

<alternative> → **(if** *<condition>* *<consequence>* *<alternant>* **)**
<condition> → *<expression>*
<conséquence> → *<expression>*
<alternant> → *<expression>*

Une **condition** est une expression ayant pour valeur

- soit vrai c'est-à-dire `#t` pour *true*
- soit faux c'est-à-dire `#f` pour *false*

Ainsi `(if (positive? 3) "oui" "non")` vaut `"oui"`



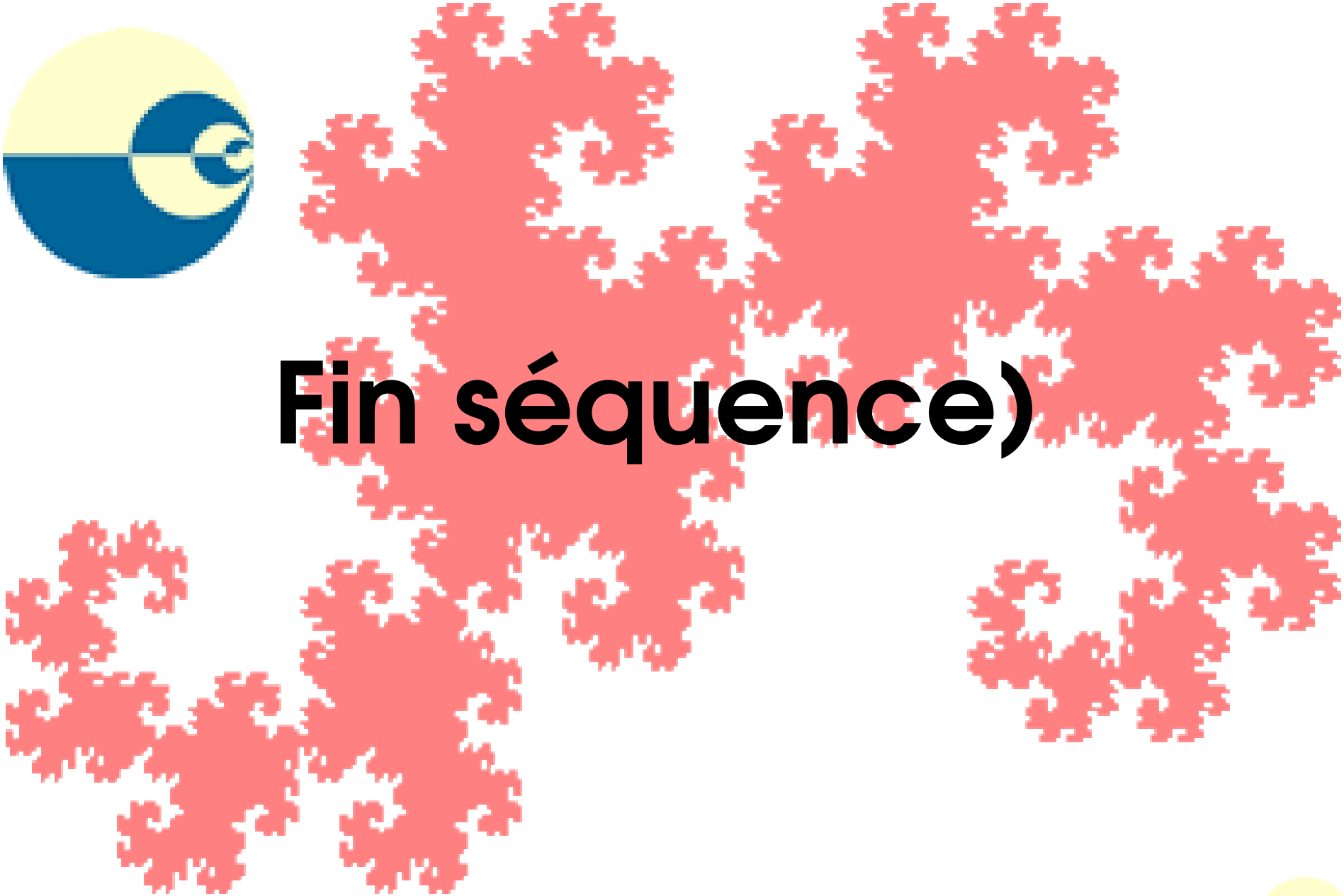
Fin séquence)

(Séquence 1.5

Résumé

Résumé

- Vous pouvez maintenant écrire des fonctions simples
- celles proposées dans la première liste d'exercices, soumettez vos solutions au correcteur automatique.
- Refaites, dans le bac à sable, de tête, sans regarder ces transparents, la fonction `carré`
- avec ses tests !
- Familiarisez-vous avec la carte de référence



Fin séquence)

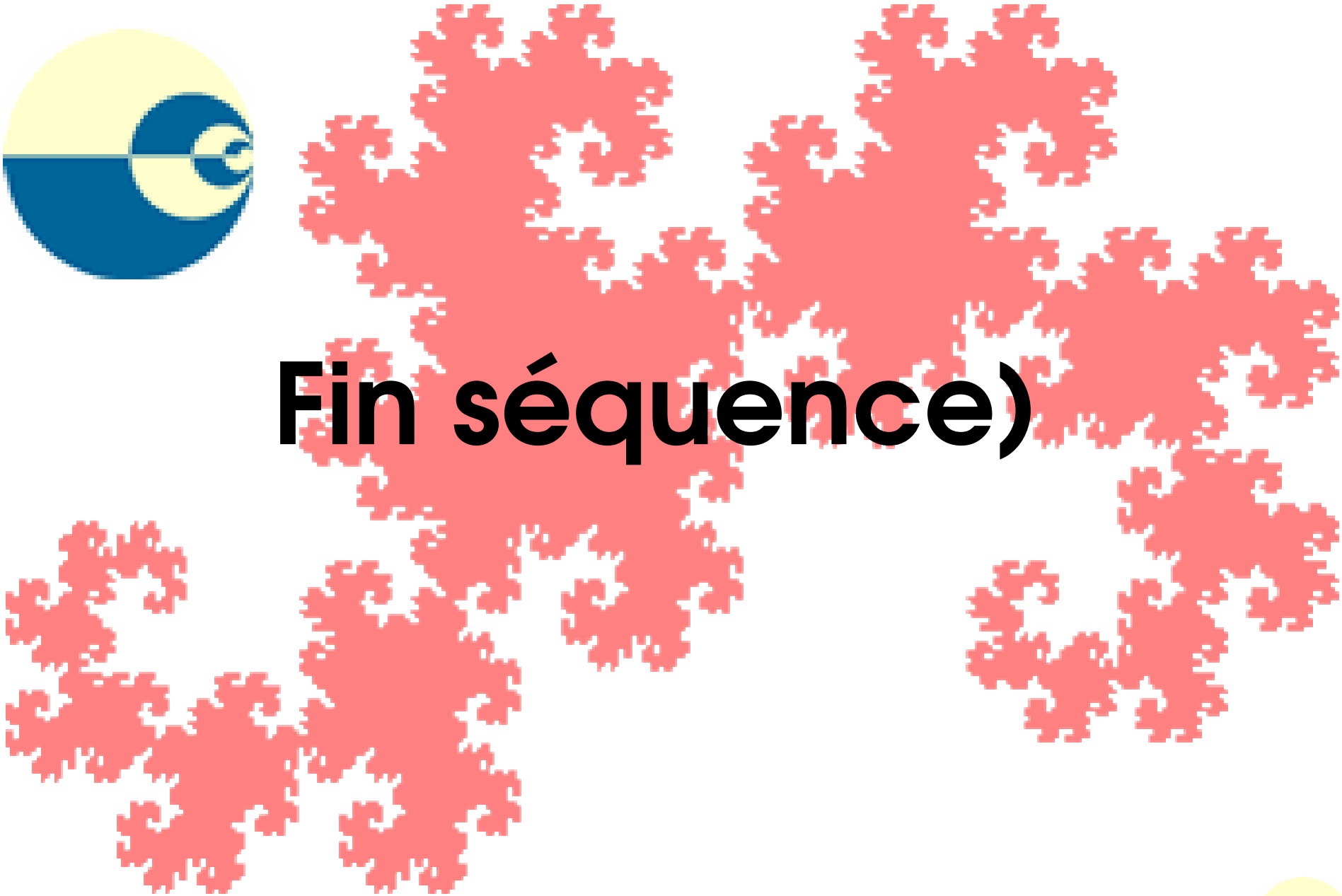
(Séquence 2.0

= Plan semaine 2

Plan semaine 2

Fin des éléments linguistiques de base

- Commentaire
- Spécification et définition d'une fonction
- Évaluation d'une application ou d'une forme spéciale
- Booléens et connecteurs
- Bloc lexical : forme spéciale `let`
- Erreur et hypothèse



Fin séquence)

(Séquence 2.1

Spécification et commentaire

Écriture de programmes

- Différence entre **grammaire** et **convention d'écriture**
- Voir la charte pour l'écriture de programmes en Scheme

Commentaires

Les commentaires sont faits pour les humains et sont ignorés des machines.
Règles d'usage pour l'emploi des commentaires :

```
;;; la spécification d'une fonction

   ;; la spécification d'une fonction interne
   ;; ou d'un passage compliqué

      ; commentaire local sur ce qui suit
      ; ou commentaire local en bout de ligne
```

Principe : on ne commente pas ce que l'on fait mais on commente pourquoi on l'a fait de cette manière !

Spécification

Rappel : Pour pouvoir utiliser une fonction prédéfinie, il faut connaître sa *spécification* :

- son nom,
- son type,
- ce qu'elle calcule,
- la signification de ses variables.

Quand on définit une nouvelle fonction il faut donner sa spécification pour pouvoir l'utiliser (l'appliquer) correctement.

Fonction : spécification et définition

1. **Spécification** (pour l'humain donc en commentaire)
   1.1 **Signature** :
   - nom de la fonction
   - type des arguments
   - type du résultat

   1.2 **Sémantique**
   - le **QUOI**
   - ce que la fonction fait (texte en français)
2. **Définition** (pour le calcul)
   - le **COMMENT**
   - implantation, calcul effectué (code en Scheme)

Multiples définitions

On peut écrire plusieurs définitions pour une même spécification.

```
;;; moyenne3: Nombre * Nombre * Nombre -> Nombre
;;; (moyenne3 x y z) rend la moyenne arithmétique
;;; de x, y et z
```

```
;;;                Une première définition:
(define (moyenne3 x y z)
  (/ (+ x y z)
     3))
```

```
;;;                Une autre définition:
(define (moyenne3 x y z)
  (+ (/ x 3)
     (/ y 3)
     (/ z 3)))
```

Multiples spécifications

On peut écrire plusieurs spécifications pour une même définition.

```
;;;              Une première spécification:
;;; additionTableur: Nombre * Nombre -> Nombre
;;; (additionTableur x y) additionne x et y
```

```
;;;              Une seconde spécification:
;;; additionTableur: Valeur * Valeur -> Nombre
;;; (additionTableur x y) additionne x et y si
;;; ce sont des nombres, rend zero sinon.

(define (additionTableur x y)
  (if (and (number? x) (number? y))
      (+ x y)
      0 ) )
```

Application de fonction

1. Appliquer la fonction avec des arguments particuliers

```
(moyenne3 13 18 14)
(moyenne3 (+ 4 5) 5 (- 8 1))
```

2. Évaluer une application
   - évaluer chacun des arguments,
   - puis évaluer l'application de la fonction aux valeurs obtenues.

```
(moyenne3 (+ 4 5)
          (moyenne3 5 4 6)
          (- 8 1) )
```

```
(moyenne3 (+ 4 5) (/ 5 0)(- 8 1))
```

ERREUR

Fonctions et formes spéciales

Une expression Scheme est une composition d'applications

- **Évaluation d'une application**
  - évaluer chacun des arguments,
  - évaluer l'application de la fonction aux valeurs obtenues.
- **sauf pour les formes spéciales : évaluation spéciale**
  - la forme spéciale `define` : *Définition*
  - la forme spéciale `if` : *Alternative*
  - les formes spéciales `and` et `or` : *Connecteurs*
  - la forme spéciale `let` : *Bloc lexical*

La syntaxe d'une alternative

Règle de grammaire :

<alternative> $\rightarrow$
(if *<condition>* *<conséquence>* *<alternant>* **)**

La *condition* est une expression ayant pour valeur

- soit vrai `#t`
- soit faux `#f`

La *conséquence* et l'*alternant* sont des expressions.

Convention d'écriture (differente de la syntaxe)

```
(if (positive? x)          ; condition
    x                      ; conséquence
    (- x) )                ; alternant
```

Évaluation d'une alternative

Tout comme la définition, l'alternative est une ***forme spéciale***, elle a une ***règle d'évaluation*** particulière.

- la condition est évaluée (retourne Vrai ou Faux)
- selon le résultat,
  - la conséquence est évaluée
  - ou l'alternant est évalué
- et devient la valeur de l'alternative.

Remarque : `define`, `if` ne sont donc pas des fonctions !

Exemple d'alternative

Fonction `valeur-absolue` :

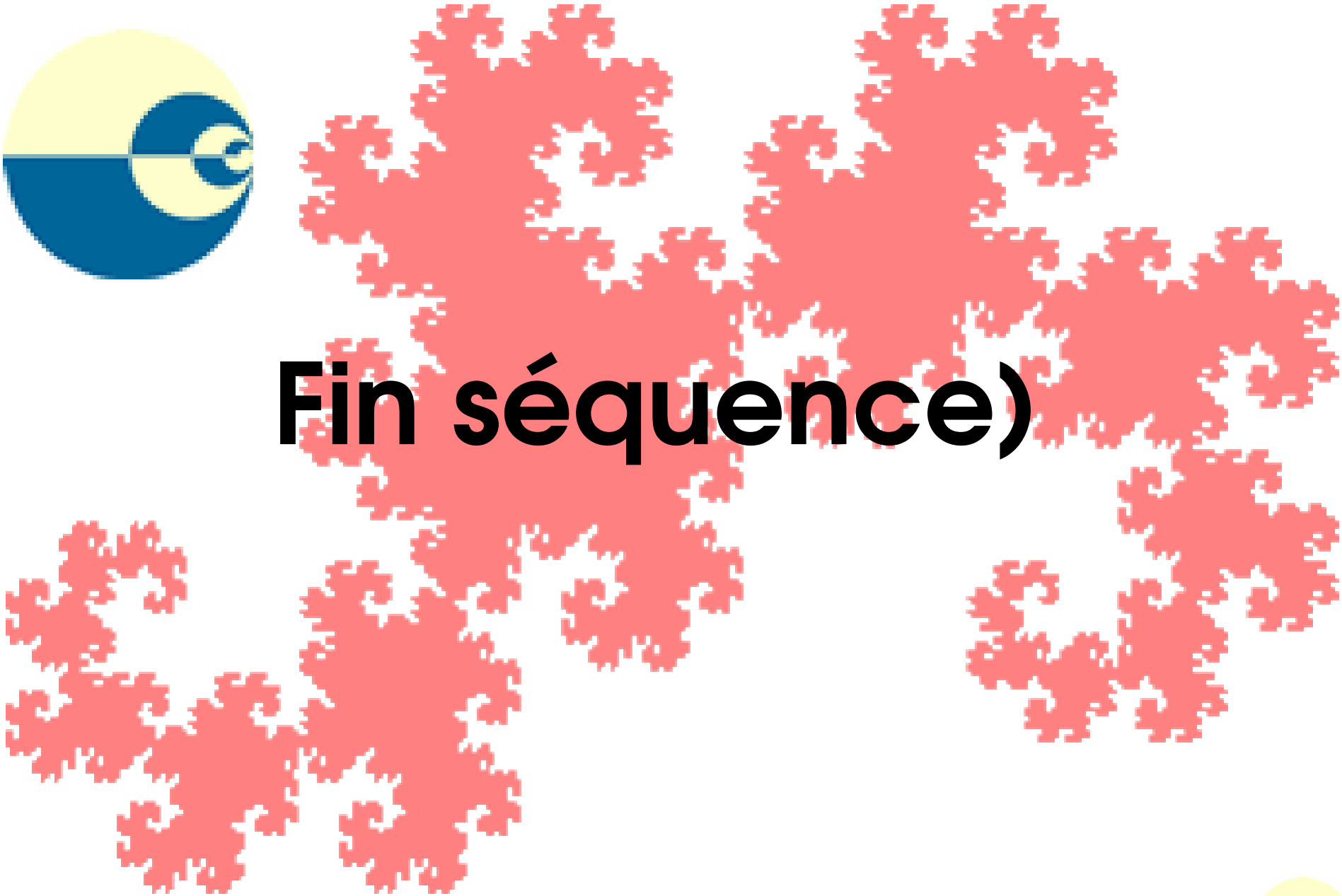
```
;;;          Spécification:
;;; valeur-absolue: Nombre -> Nombre
;;; (valeur-absolue x) rend la valeur absolue de x
```

```
;;;          Une première définition:
(define (valeur-absolue x)
  (if (>= x 0)
      x
      (- x) ) )
```

```
;;;          Une autre définition:
(define (valeur-absolue x)
  (if (negative? x )
      (- x)
      x ) )
```

Exemple 2 (suite)

```
;;;          Encore une définition:
(define (valeur-absolue x)
  ((if (>= x 0) + -) 0
                     x ) )

;;;          Applications:
(valeur-absolue (+ 3 2))
(valeur-absolue -5)
(valeur-absolue (- 5))
```



Fin séquence)

(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 :

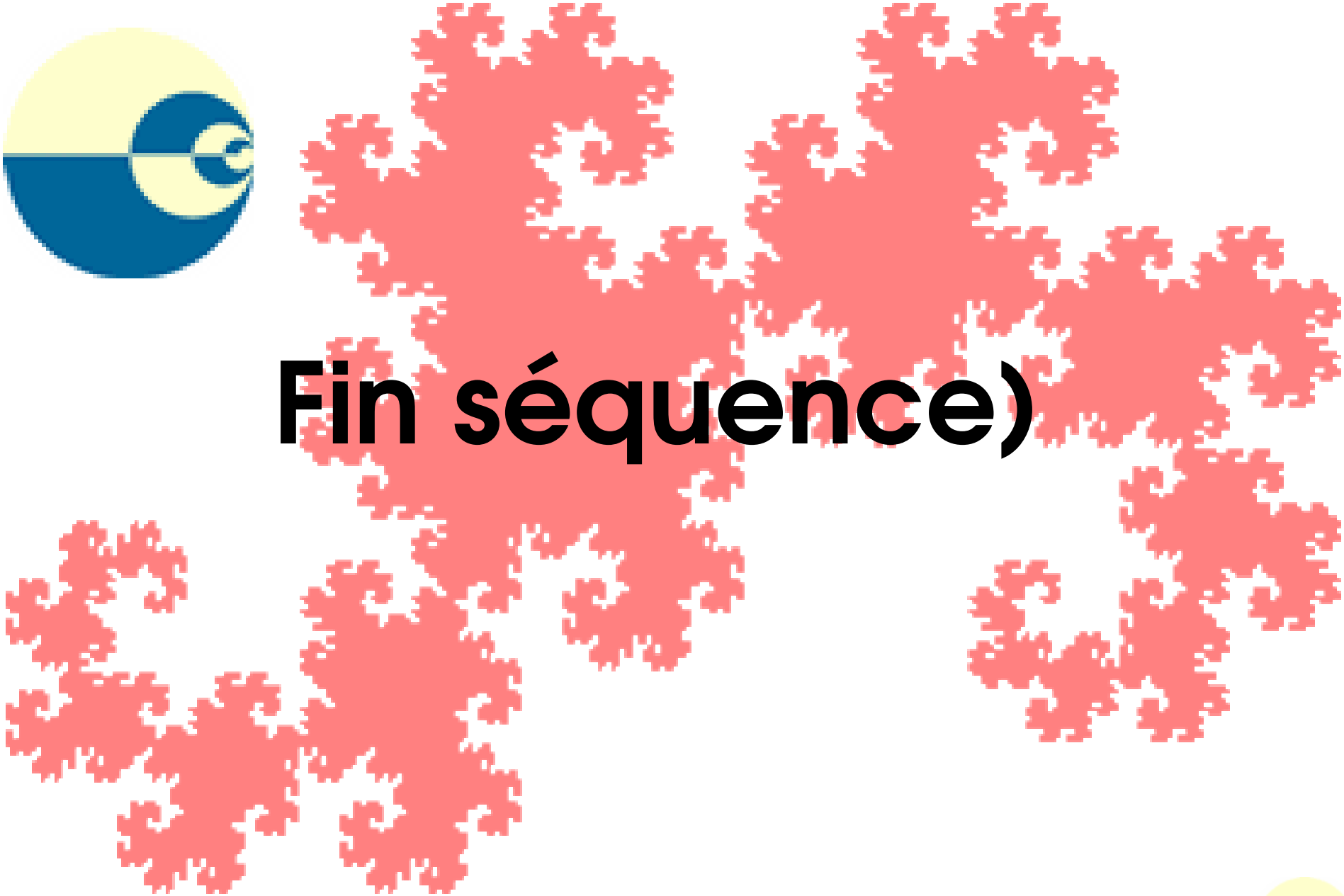
```
;;; nombre-positif?: Valeur -> bool
        (nombre-positif? 32.45)   → #t
        (nombre-positif? "essai") → #f
```

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)

(Séquence 2.3

Bloc lexical

Le bloc `let`

Théorème :
Étant donné un triangle quelconque de côtés a, b et c, son aire est égale à la racine carrée de $s(s-a)(s-b)(s-c)$ *où s est le demi-périmètre.*

On utilise 4 fois la quantité *s*, on souhaite donc la calculer une fois et l'utiliser 4 fois.

```
;;; aire-triangle:
;;;       Nombre * Nombre * Nombre -> Nombre
;;; (aire-triangle a b c) rend l'aire d'un triangle
;;; de côtés a, b et c
(define (aire-triangle a b c)
  (let ((s (/ (+ a b c ) 2)))
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

La syntaxe d'un `let`

<*bloc*> → **(let (** <*liaison*>* **)** <*corps*> **)**
<*liaison*> → **(** <*variable*> <*expression*> **)**
<*corps*> → <*expression*>

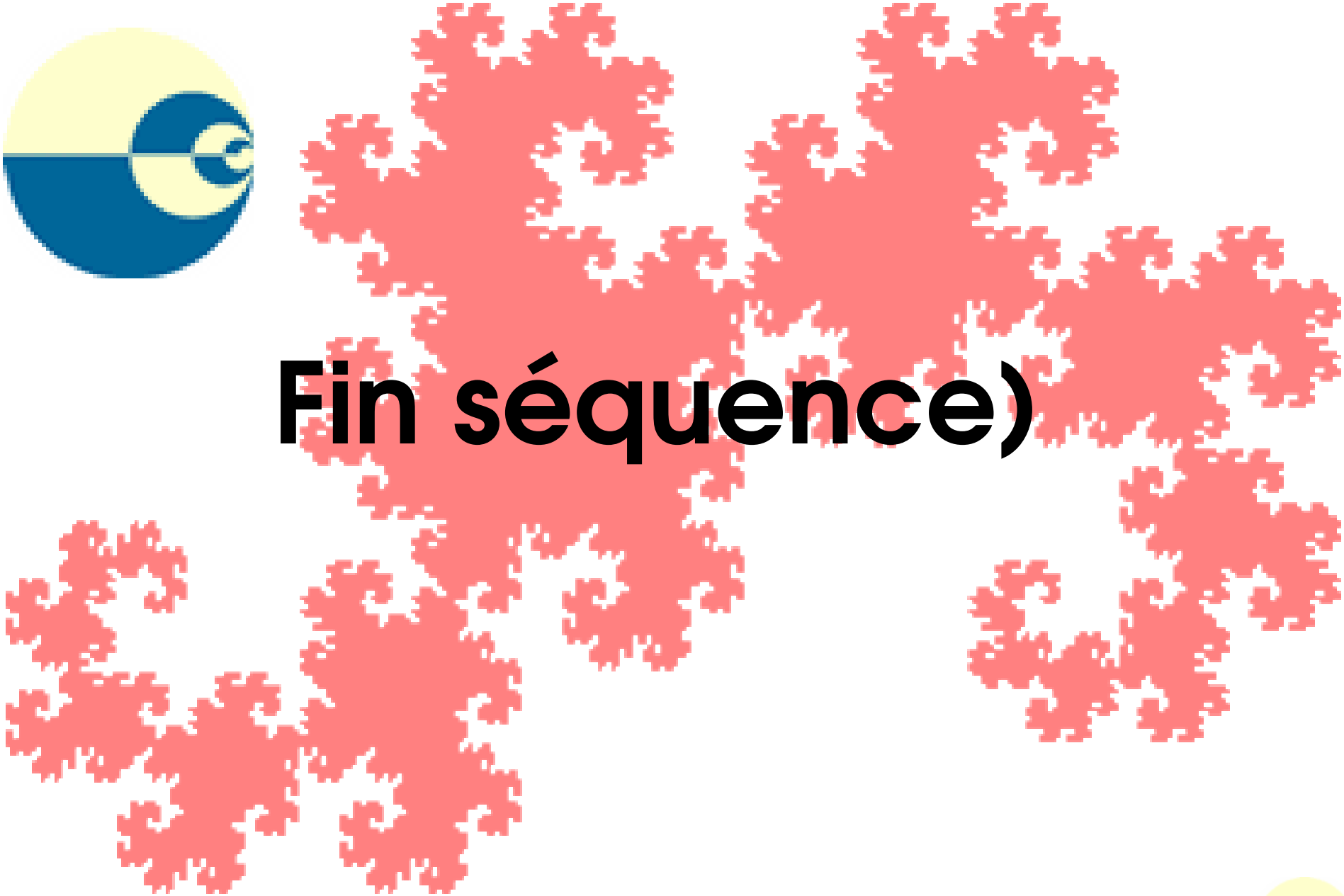
L'écriture d'un `let` :

```
(let ((var1 expr1)
      (var2 expr2)
      ...
      (varN exprN) )
  corps )
```

L'évaluation d'un `let`

- **évaluation** des expressions *expr1*, . . . , *exprN*
- **création** des variables locales *var1*, . . . , *varN*
- **enrichissement** de l'environnement courant en associant à chaque variable *varI* la valeur de *exprI*
- **évaluation** du *corps* dans cet environnement.

Environnement : l'ensemble des variables que l'on peut utiliser.
Portée d'une variable : la zone textuelle où l'on peut utiliser cette variable.
La portée des variables d'un `let` est le corps de ce `let`.



Fin séquence)

(Séquence 2.4

Erreur et hypothèse

Détection d'erreur

```
;;; aire-couronne : Nombre * Nombre -> Nombre
;;; ERREUR lorsque r1 < r2
;;; (aire-couronne r1 r2) rend l'aire de la couronne
;;; de rayon extérieur r1 et de rayon intérieur r2
;;; HYPOTHÈSE: r1 et r2 positifs
(define (aire-couronne r1 r2)
  (if (< r1 r2)
      (erreur 'aire-couronne
              "rayon extérieur (" r1 ") <"
              "rayon intérieur (" r2 ")")
      (- (aire-disque r1) (aire-disque r2))))
```

La détection d'erreur a donc un coût.

La notion d'hypothèse

```
;;; aire-couronne-sans : Nombre * Nombre -> Nombre
;;; (aire-couronne-sans r1 r2) rend l'aire de la
;;; couronne de rayon extérieur r1 et de rayon
;;; intérieur r2
;;; HYPOTHÈSES: r1 et r2 positifs et r1 >= r2
(define (aire-couronne-sans r1 r2)
  (- (aire-disque r1) (aire-disque r2)))
```

La preuve que les hypothèses sont respectées est maintenant à la charge de l'appelant.

Vocabulaire pour une application

Dans l'application `(* (+ 1 2) 3)` :

- l'expression en ***position fonctionnelle*** est `*`
- la fonction est la multiplication (la multiplication est la valeur de la variable `*`)
- les ***paramètres d'appel*** sont les expressions `(+ 1 2)` et `3`
- les ***arguments*** de la multiplication sont `3` et `3`

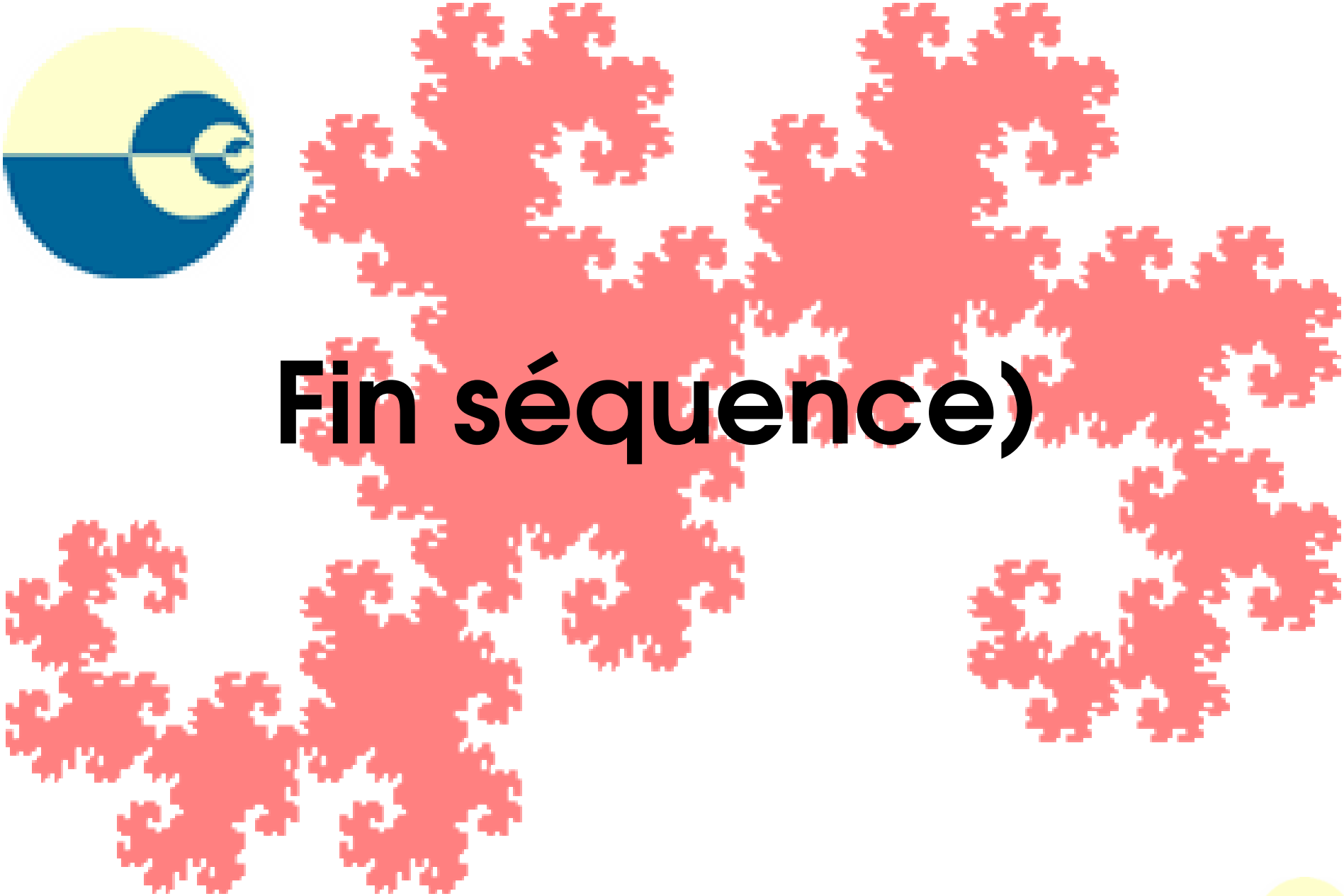
Vocabulaire pour une définition

Dans la définition

```
(define (produit x y)
   (* x y) )
```

- `x` et `y` sont les ***variables***
- les ***arguments*** de l'appel à `produit` sont les valeurs de ces variables.

De nombreux abus de langage sont toutefois faits.



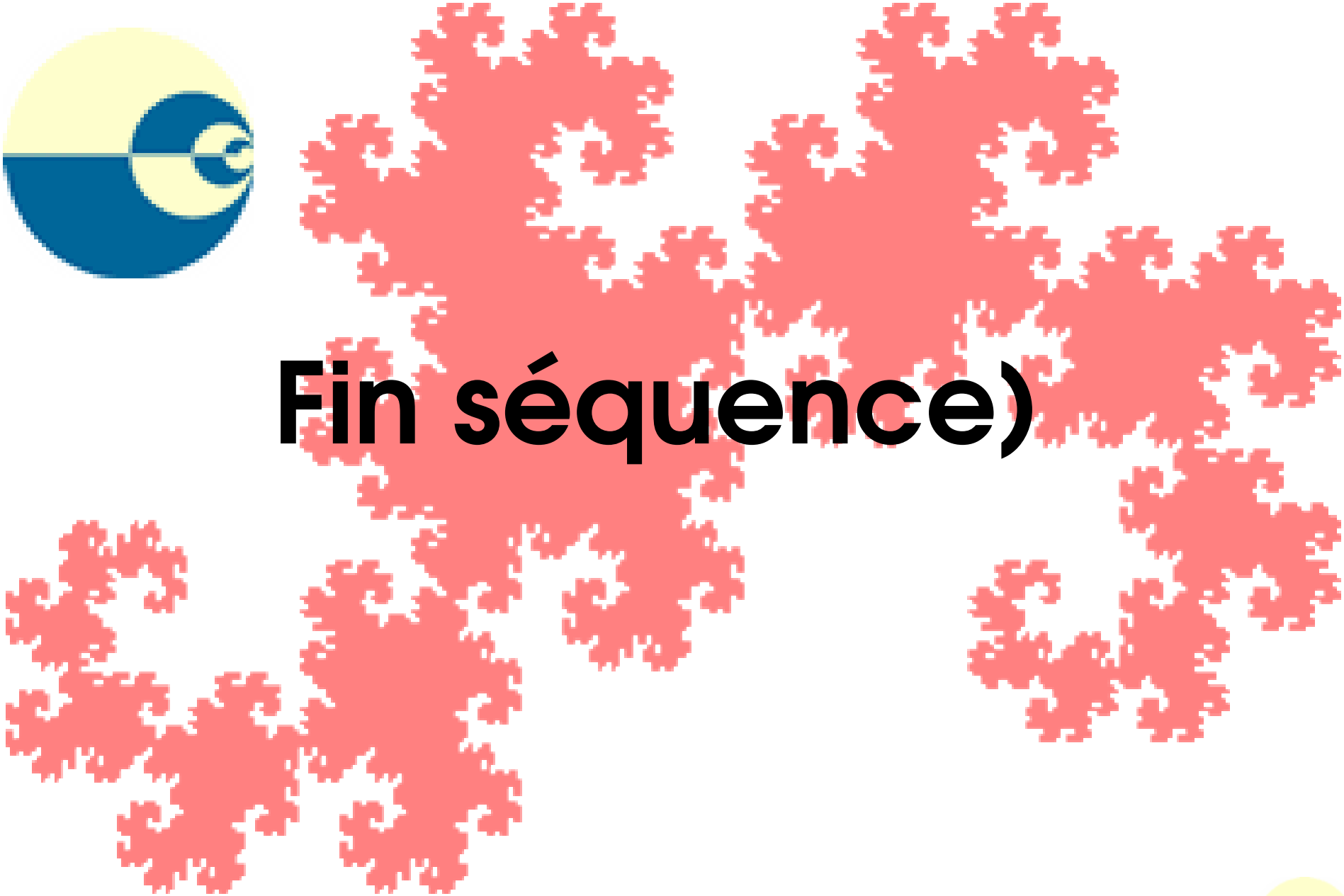
Fin séquence)

(Séquence 2.5

Résumé

Résumé

- Toutes les connaissances linguistiques de base de Scheme sont maintenant connues (sauf une : la citation).
- Il ne reste plus que la cinquantaine de fonctions de la carte de référence.
- Vous savez manipuler des conditions booléennes
- et nommer, afin de réutiliser, le résultat de calculs partiels.
- Enfin, vous avez acquis le vocabulaire de base (utile pour poser des questions précises)
- Vous comprenez les spécifications, leur structure et la notion de contrat qu'elles véhiculent



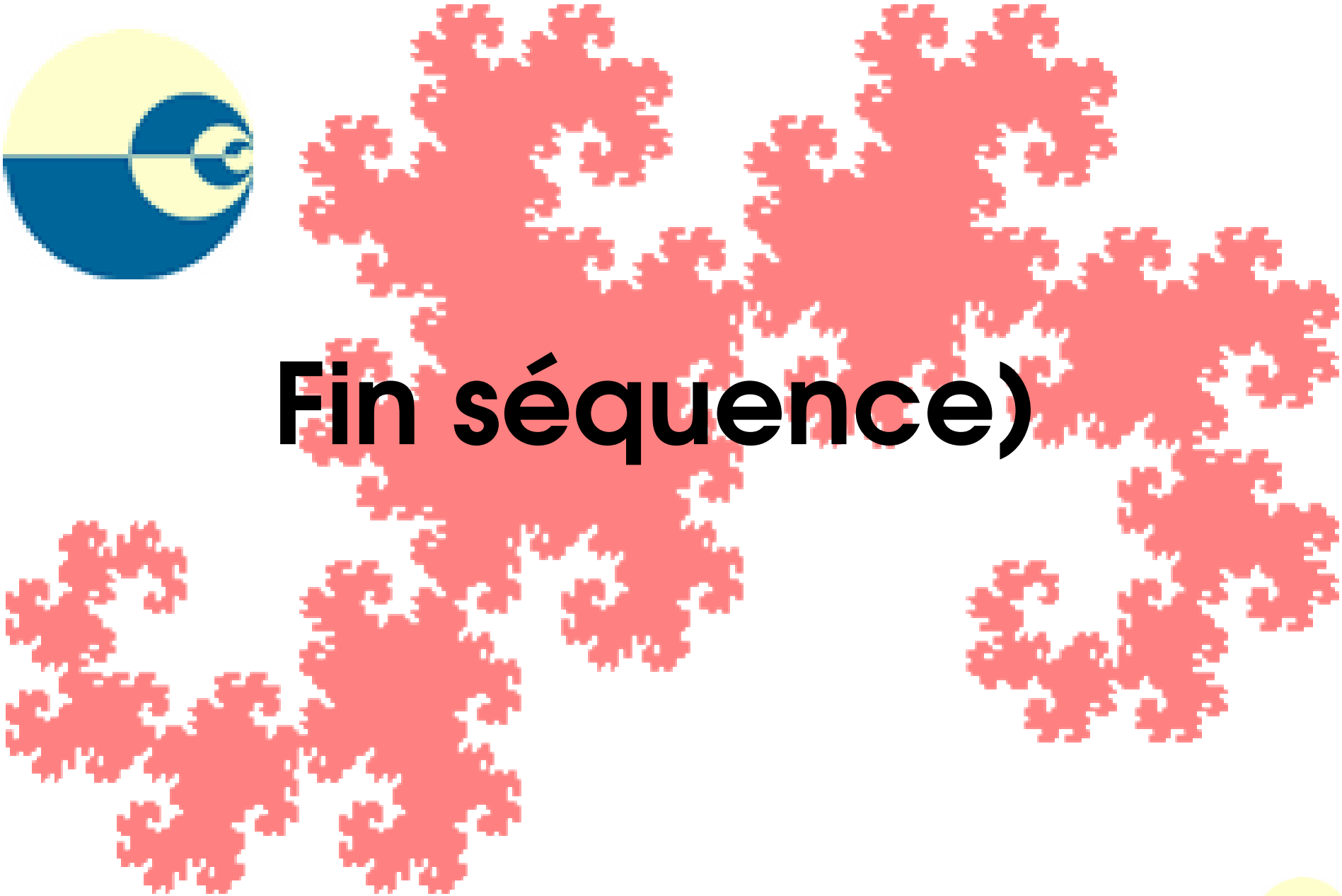
Fin séquence)

(Séquence 3.0

= Plan semaine 3

Plan semaine 3

1. Récursion sur les entiers naturels
   - Exemples : $n!$, x^n
   - Principes de la récursion
   - Définition d'une fonction récursive
   - Évaluation par substitution
2. Variations autour de la factorielle
3. Variations autour de la puissance



Fin séquence)

(Séquence 3.1

Récursion sur entiers naturels

Définition récursive : factorielle

Spécification **informelle** de la factorielle :
$n! = 1 * 2 * 3 * \ldots * n$

Définition récursive : factorielle

Spécification **informelle** de la factorielle :
$n! = 1 * 2 * 3 * \ldots * n$

Définition ambigüe : a-t-on $3! = 1 * 2 * 3 * 3$?

Définition récursive : factorielle

Spécification **informelle** de la factorielle :
$n! = 1 * 2 * 3 * \ldots * n$

Définition ambigüe : a-t-on $3! = 1 * 2 * 3 * 3$?

Définition **récursive** de factorielle n :

$$\begin{aligned} n! &= 1 \quad \text{pour} \quad n = 0 \\ n! &= n * (n-1)! \quad \text{pour} \quad n \geq 1 \end{aligned}$$

Définition récursive : puissance

Spécification **informelle** de la puissance :
$x^n = x * x * x * \ldots * x$

Définition récursive : puissance

Spécification **informelle** de la puissance :
$x^n = x * x * x * \ldots * x$

Définition **récursive** de x puissance n :

$$\begin{aligned} x^0 &= 1 \\ x^n &= x * x^{n-1} \qquad \text{pour} \qquad n \geq 1 \end{aligned}$$

Principes

- **Décomposition** : $f(n) = \ldots f(p) \ldots$
  Exprimer $f(n)$ en fonction de $f(p)$ avec **p<n**
- **Cas de base** : $f(0) = \ldots$
  Donner la(les) valeur(s) de f pour la(les) valeur(s) de base

Principes

- **Décomposition** : $f(n) = \ldots f(p) \ldots$
  Exprimer $f(n)$ en fonction de $f(p)$ avec **p<n**
- **Cas de base** : $f(0) = \ldots$
  Donner la(les) valeur(s) de f pour la(les) valeur(s) de base

Fonctionne car les entiers naturels forment un « domaine bien fondé » où il ne peut y avoir de suite infiniment décroissante.

Définition en 3 cas

Une autre définition de la puissance en trois cas :

$$\begin{aligned} x^0 &= 1 \\ x^{2n} &= (x^n)^2 \quad \text{pour} \quad n \geq 1 \\ x^{2n+1} &= x * (x^n)^2 \quad \text{pour} \quad n \geq 1 \end{aligned}$$

Double récursion

Les nombres de Fibonacci sont définis par une double récursion :

$$
\begin{aligned}
fib(0) &= 1 \\
fib(1) &= 1 \\
fib(n) &= fib(n-1) + fib(n-2) \qquad \text{pour} \qquad n \geq 2
\end{aligned}
$$

Récursion enveloppée

La fonction 91 de McCarthy :

$$\begin{aligned} f(n) &= n - 10 \quad \text{pour} \quad n > 100 \\ f(n) &= f(f(n+11)) \quad \text{pour} \quad n \leq 100 \end{aligned}$$

En Scheme

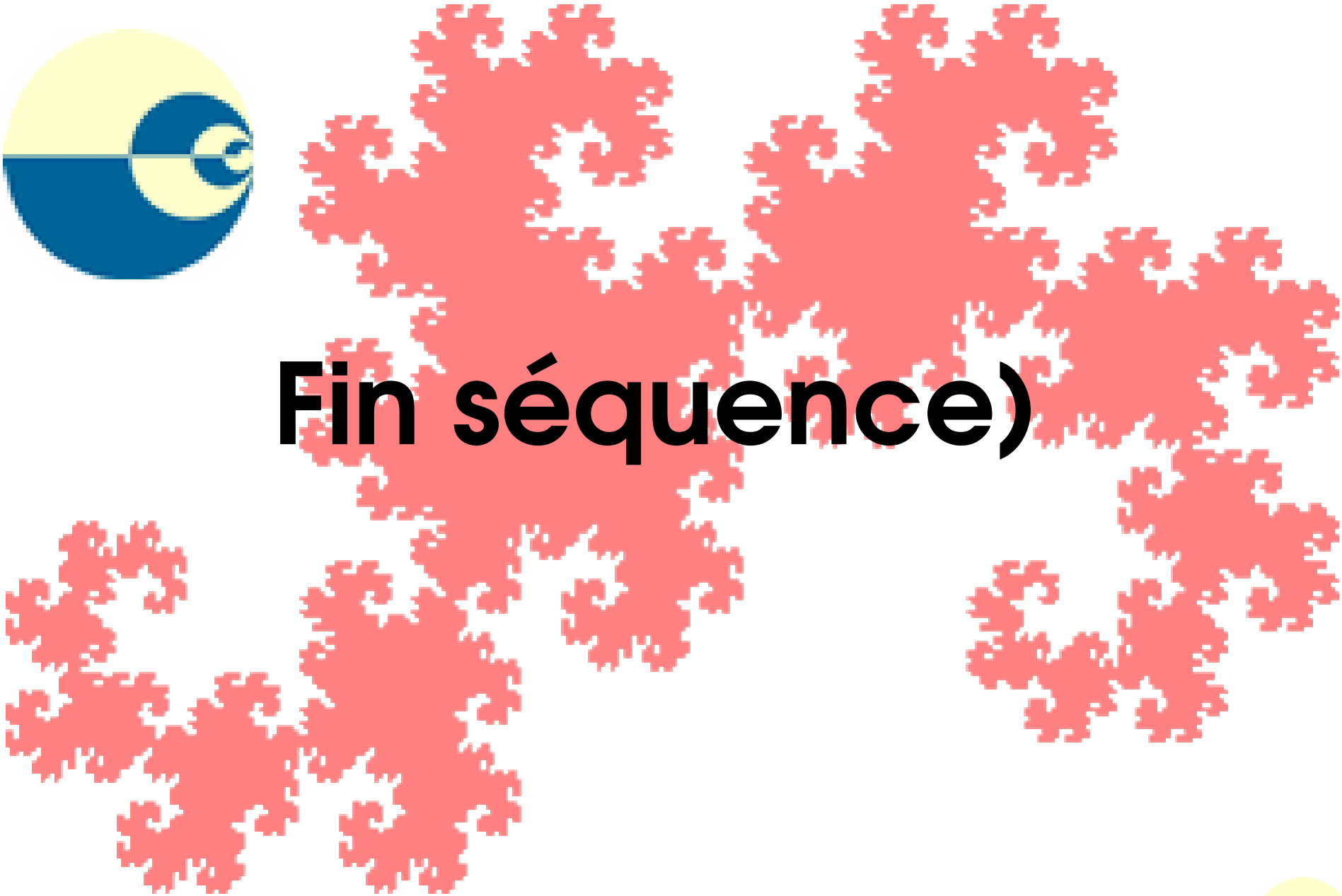
- Définition (courante) d'une fonction récursive sur entier naturel (c'est-à-dire les nombres 0, 1, 2, etc.) :

```
(define (f n)
  (if (> n 0)
      expression fonction de f(n − 1)
      cas de base
  ))
```

- Évaluation de la factorielle

```
(f 3) ≡ (* 3 (f 2)) ≡ (* 3 (* 2 (f 1)))
≡ (* 3 (* 2 (* 1 (f 0))))
≡ (* 3 (* 2 (* 1 1)))
≡ (* 3 (* 2 1))
≡ (* 3 2)
≡ 6
```

```
(f 0)  cas de base: arrêt des appels récursifs
```



Fin séquence)

(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) ))
```

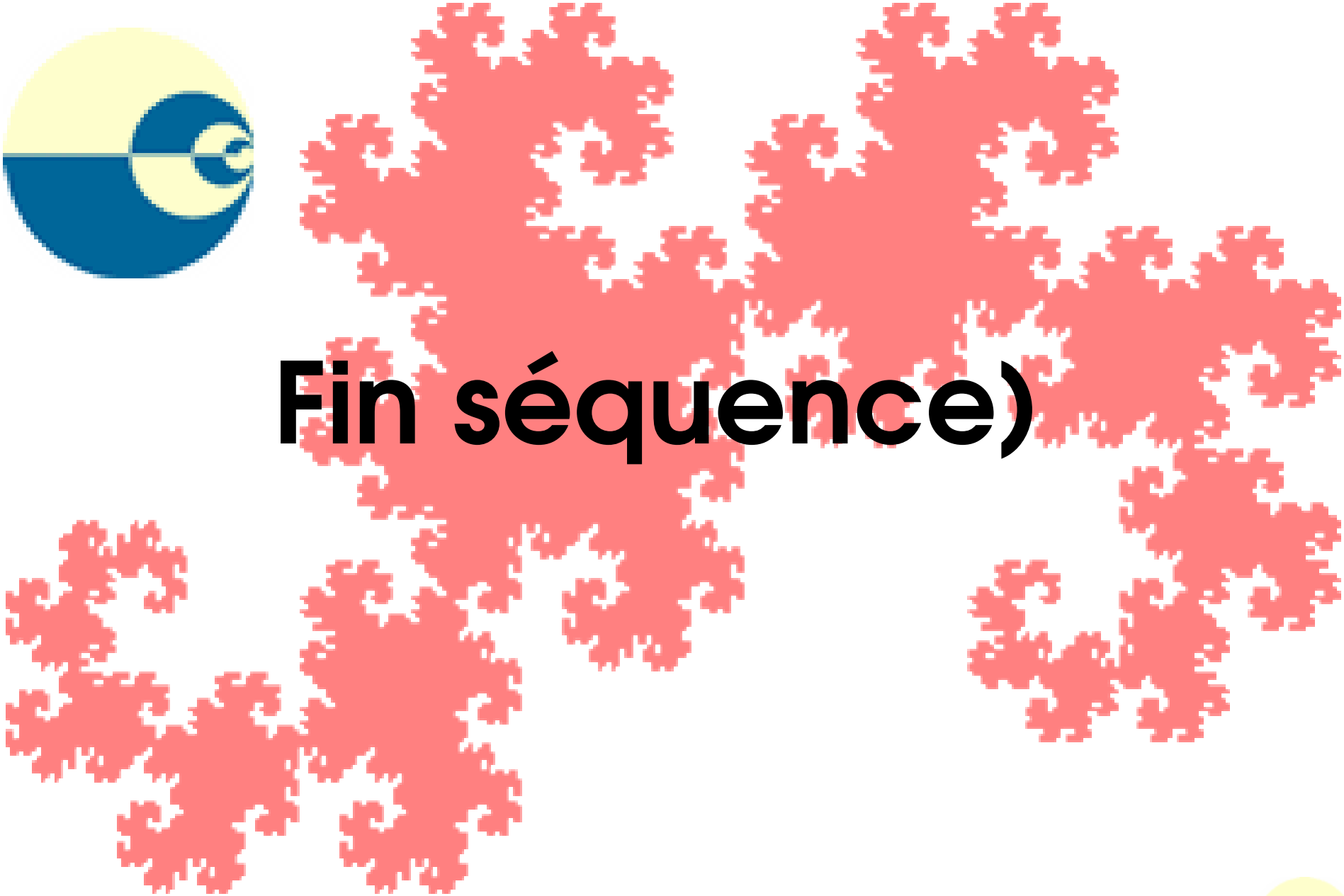
```
;;; fact: nat -> nat
;;; (fact n) rend la factorielle de n
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1))) ) )
```

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)

(Séquence 3.3

puissance

Définitions de la puissance

Définition **récursive** de x puissance n :

$$\begin{aligned} x^0 &= 1 \\ x^n &= x * x^{n-1} \qquad \text{pour} \qquad n \geq 1 \end{aligned}$$

Définitions de la puissance

Définition **récursive** de x puissance n :

$$\begin{aligned} x^0 &= 1 \\ x^n &= x * x^{n-1} \qquad \text{pour} \qquad n \geq 1 \end{aligned}$$

Une autre définition récursive de la puissance :

$$\begin{aligned} x^0 &= 1 \qquad \text{pour} \qquad n = 0 \\ x^{2n} &= (x^n)^2 \qquad \text{pour} \qquad n \geq 1 \\ x^{2n+1} &= x * x^{2n} \qquad \text{pour} \qquad n \geq 1 \end{aligned}$$

mais on peut aussi écrire :

$$x^{2n} = (x^2)^n \qquad \text{pour} \qquad n \geq 1$$

La fonction `puissance`

Définition **récursive** de x puissance n :

$$\begin{aligned} x^0 &= 1 \\ x^n &= x * x^{n-1} \qquad \text{pour} \qquad n \geq 1 \end{aligned}$$

```
;;; puissance : Nombre * nat -> Nombre
;;; (puissance x n) rend x a la puissance n
(define (puissance x n)
  (if (> n 0)
      (* x (puissance x (- n 1)))
      1 ) )
```

Lenteur ?

Une autre définition de la puissance calquée sur la définition :

$$\begin{aligned} x^0 &= 1 \quad \text{pour} \quad n = 0 \\ x^{2n} &= (x^n)^2 \quad \text{pour} \quad n \geq 1 \\ x^{2n+1} &= x * x^{2n} \quad \text{pour} \quad n \geq 1 \end{aligned}$$

```
;;; puissanceLent : Nombre * nat -> Nombre
;;; HYPOTHESE n est un entier positif ou nul
;;; (puissanceLent x n) rend x a la puissance n
(define (puissanceLent x n)
  (if (= n 0)
      1
      (if (even? n)
          (* (puissanceLent x (quotient n 2))
             (puissanceLent x (quotient n 2)))
          (* x
             (puissanceLent x (quotient n 2))
             (puissanceLent x (quotient n 2))))))
```

Puissance

Le calcul de $x^{n/2}$ est fait 2 fois à chaque appel récursif

```
|(puissanceLent 2 6)
| (puissanceLent 2 3)
| |(puissanceLent 2 1)
| | (puissanceLent 2 0)
| | 1
| | (puissanceLent 2 0)
| | 1
| |2
| |(puissanceLent 2 1)
| | (puissanceLent 2 0)
| | 1
| | (puissanceLent 2 0)
| | 1
| |2
| 8
```

suite↗

```
| (puissanceLent 2 3)
| |(puissanceLent 2 1)
| | (puissanceLent 2 0)
| | 1
| | (puissanceLent 2 0)
| | 1
| |2
| |(puissanceLent 2 1)
| | (puissanceLent 2 0)
| | 1
| | (puissanceLent 2 0)
| | 1
| |2
| 8
|64
```

Un autre essai de définition de la puissance en deux fonctions :

```
;;; carre : Nombre -> Nombre
;;; (carre y) rend le carre de y
(define (carre y)
  (* y y) )
```

```
;; puissanceBis a la meme specification que puissance
(define (puissanceBis x n)
  (if (= n 0)
      1
      (if (even? n)
          (carre (puissanceBis x (quotient n  2)))
          (* (carre (puissanceBis x (quotient n 2)))
             x ) ) ) )
```

et trace de `puissanceBis`

```
|(puissanceBis 2 6)
| (puissanceBis 2 3)
| |(puissanceBis 2 1)
| | (puissanceBis 2 0)
| | 1
| | (carre 1)
| | 1
| |2
| |(carre 2)
| |4
| 8
| (carre 8)
| 64
|64
```

Et, stylistiquement, avec une fonction interne

```
;;; puissanceBisBis : Nombre * nat ->  Nombre
;;; (puissanceBisBis x n) rend x a la puissance n
(define (puissanceBisBis x n)
  ;; carre : Nombre -> Nombre
  ;; (carre y) rend le carre de y
  (define (carre y)
    (* y y) )
  (if (= n 0)
      1
      (if (even? n)
          (carre (puissanceBisBis
                  x (quotient n  2)))
          (* x (carre (puissanceBisBis
                       x (quotient n 2)))))))
```

Puissance encore

L'imbrication d'alternative consomme de la marge gauche et des parenthèses. On peut aussi utiliser une nouvelle forme spéciale `cond` et écrire :

```
;;; puissance : Nombre * nat ->  Nombre
;;; (puissance x n) rend x a la puissance n
(define (puissance x n)
  ;; carre : Nombre -> Nombre
  ;; (carre y) rend le carre de y
  (define (carre y)
    (* y y) )
  (cond
   ((= n 0) 1)
   ((even? n)
    (carre (puissance x (quotient n  2))) )
   (else
    (* x (carre (puissance x (quotient n 2))))) ) )
```

Puissance

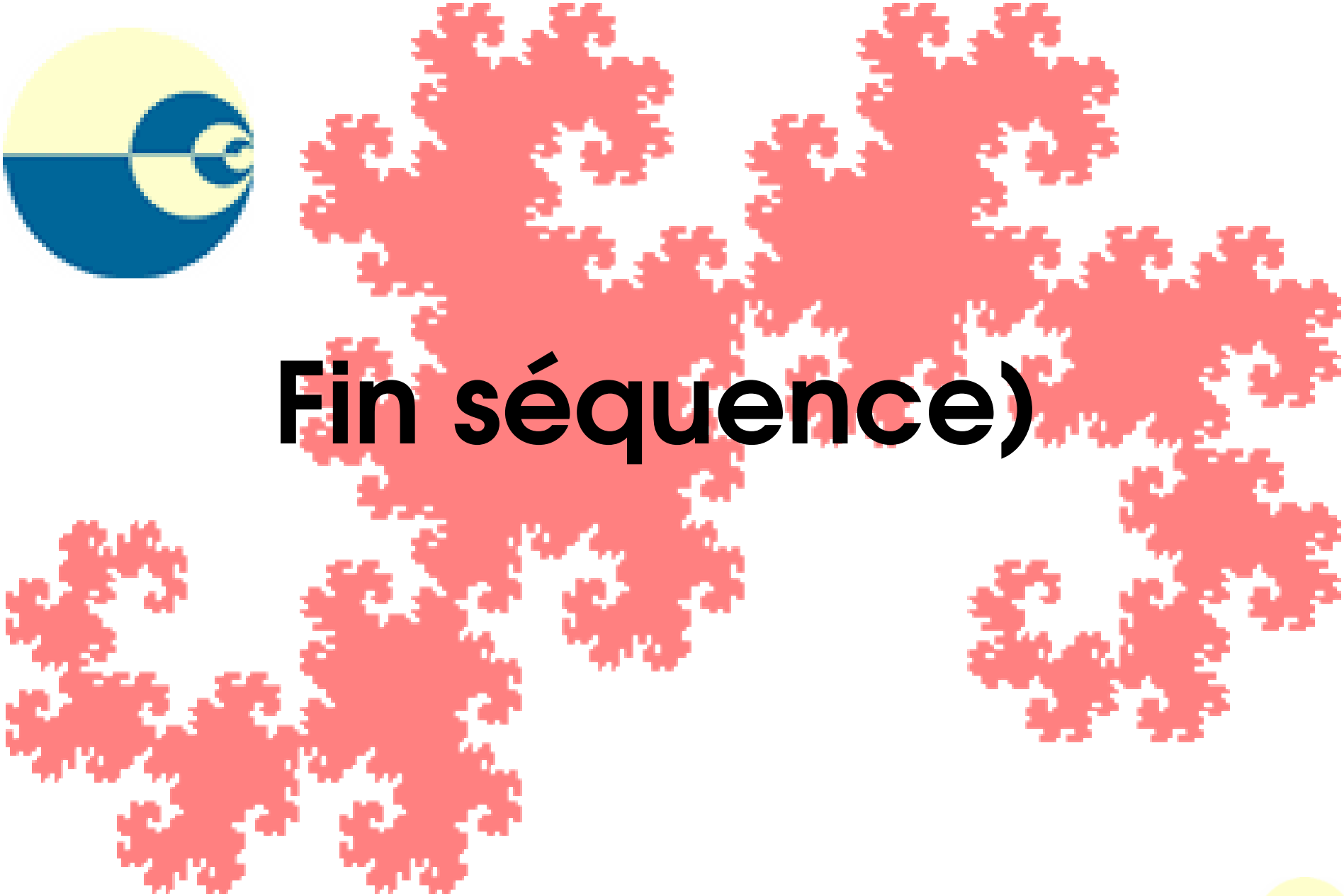
Et encore une autre définition où l'on met en facteur le calcul de $x^{n/2}$

```
(define (puissanceTer x n)
  (if (= n 0)
      1
      (let ((P (puissanceTer x (quotient n  2))))
        (if (even? n)
            (* P P)
            (* x P P) ) ) ) )
```

```
|(puissanceTer 2 6)
| (puissanceTer 2 3)
| |(puissanceTer 2 1)
| | (puissanceTer 2 0)
| | 1
| |2
| 8
|64
```

Complexité de la puissance

- La ***complexité*** s'intéresse au nombre d'opérations qu'il faut effectuer pour un calcul.
- Pour la puissance, la mesure sera le nombre de multiplications à réaliser.
- Avec `puissanceTer`, on divise par deux, à chaque fois, l'argument. Si l'on part d'un entier n, il faut environ $log_2(n)$ divisions par deux pour l'amener à 1 car $n = 2^{log_2(n)}$.
- Il faut donc environ 20 multiplications pour élever un nombre à la puissance 1000 (en fait 17). Il en faut 19 pour élever un nombre à la puissance 959 mais seulement 14 pour 960.
- Comparer avec les 1000 multiplications que nécessite le premier schéma récursif naïf.



Fin séquence)

(Séquence 3.4

Interlude : `cond` et `let*`

Interlude : la forme spéciale `cond`

L'alternative est un choix entre deux possibilités. La forme spéciale `cond` est un choix multiple.

```
(cond ( cond1 expr1 )
      ( cond2 expr2 )
      ...
      ( condn exprn )
      (else exprn+1 ) )
      ≡
(if cond1
    expr1
    (if cond2
        expr2
        ...
            (if condn
                exprn
                exprn+1 ) ... ) )
```

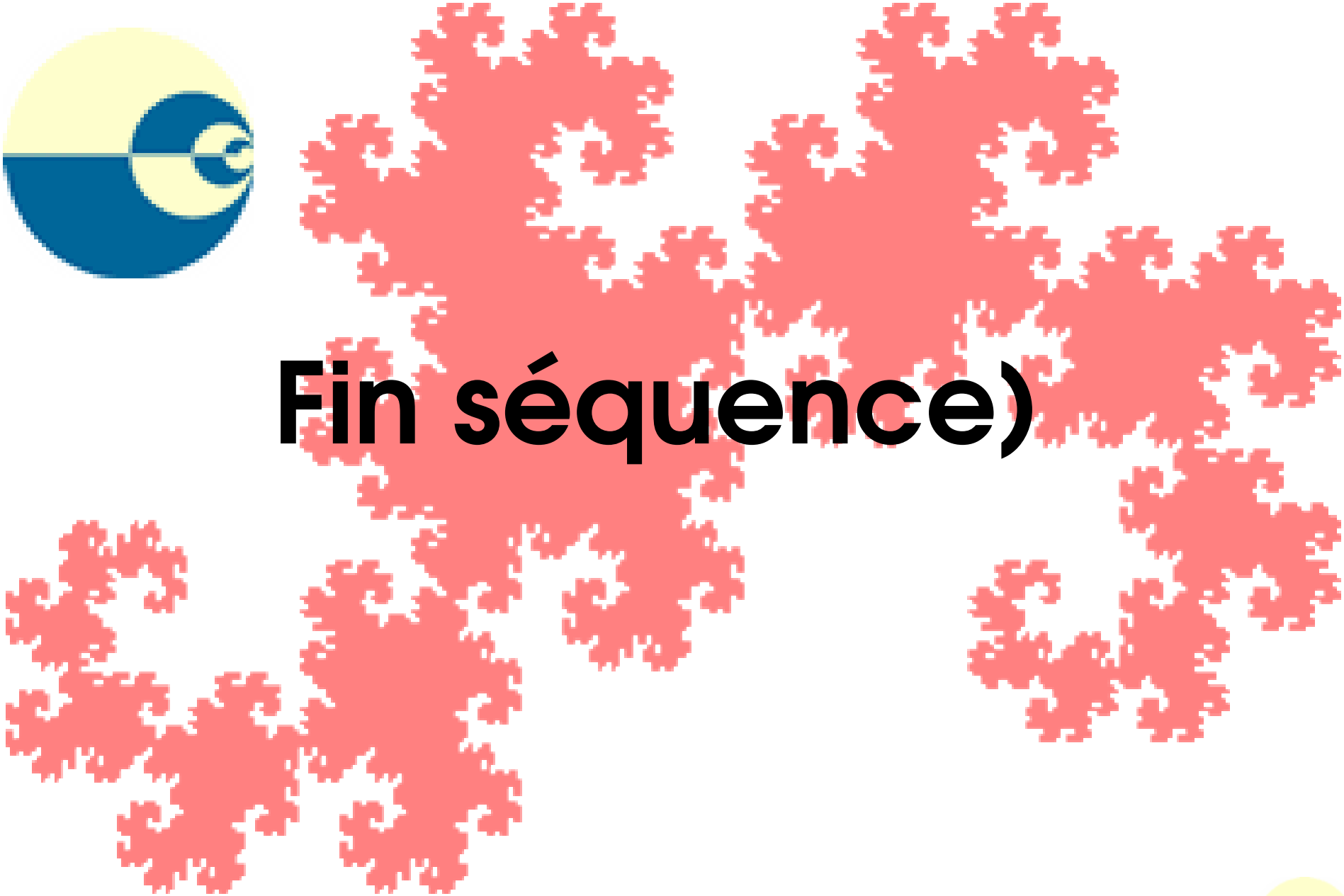
Interlude : la forme spéciale `let*`

Toujours pour gagner de la marge gauche :

```
(let* (( var1 expr1 )
       ( var2 expr2 )
       ...
       ( varn exprn ) )
  corps )
      ≡
(let (( var1 expr1 ))
  (let (( var2 expr2 ))
    ...
    (let (( varn exprn ))
      corps ) ... ))
```

Interlude algébrique

```
(if α (f β) (f γ))
   ≡ (f (if α β γ))


(if α (β arg) (γ arg))
   ≡ ((if α β γ) arg)
```



Fin séquence)

(Séquence 3.5

Récurrence et récursion

Récurrence et récursion

- Qu'est-ce que le principe de récurrence ?
- Comparaison avec la récursion

Récurrence

- La récurrence (ou principe d'induction) est une technique de preuve mathématique sur l'ensemble des entiers naturels.
- Les entiers naturels ont deux constructeurs 0 et successeur. Par convention successeur(0) = 1, successeur(1) = 2, etc.
- Dans ce monde, comment prouver que

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

Exemples sur des petits nombres

$$\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$$

$$\sum_{i=1}^{2} i = 1 + 2 = \frac{2(2+1)}{2} = 3$$

$$\sum_{i=1}^{3} i = 1 + 2 + 3 = \frac{3(3+1)}{2} = 6$$

Induction

Supposons que la formule soit vraie jusqu'à n alors

$$\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1) = \frac{n(n+1)}{2} + (n+1)$$

$$= \frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2}$$

QED

Récursion

- Que vaut $\sum_{i=1}^{n} i$?
- Décomposition récursive :

$$\sum_{i=1}^{1} i = 1$$

$$\forall n > 0, \sum_{i=1}^{n+1} i = (n+1) + \sum_{i=1}^{n} i$$

Récursion en Scheme

```
;;; somme: nat -> nat
;;; (somme n) calcule la somme des entiers naturels
;;; de 1 à n
(define (somme n)
  (if (> n 1)
      (+ n (somme (- n 1)))
      n ) )
```

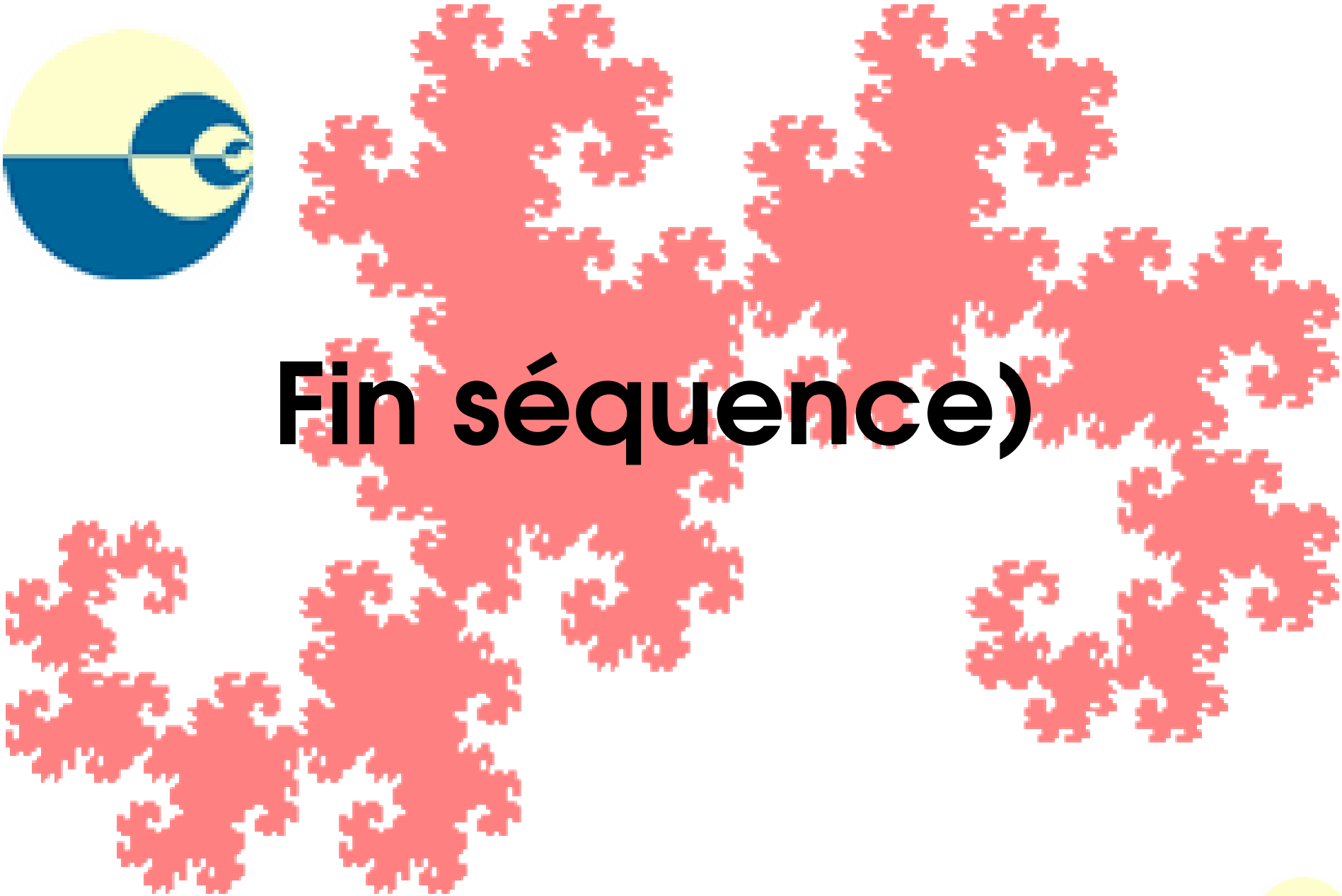
Coût linéaire (en nombre d'additions).

Récurrence et récursion

- Preuve d'une formule
- par induction à partir des cas de base
- Ne calcule pas !

- Calcul de valeurs
- par décomposition en problèmes plus simples et de même nature
- Ne trouve pas la formule !

- Pas toujours de formule !
  Quelle serait celle correspondant à $\sum_{i=1}^{n} log^3(i)$?
- Mais la formule est bien plus efficace !



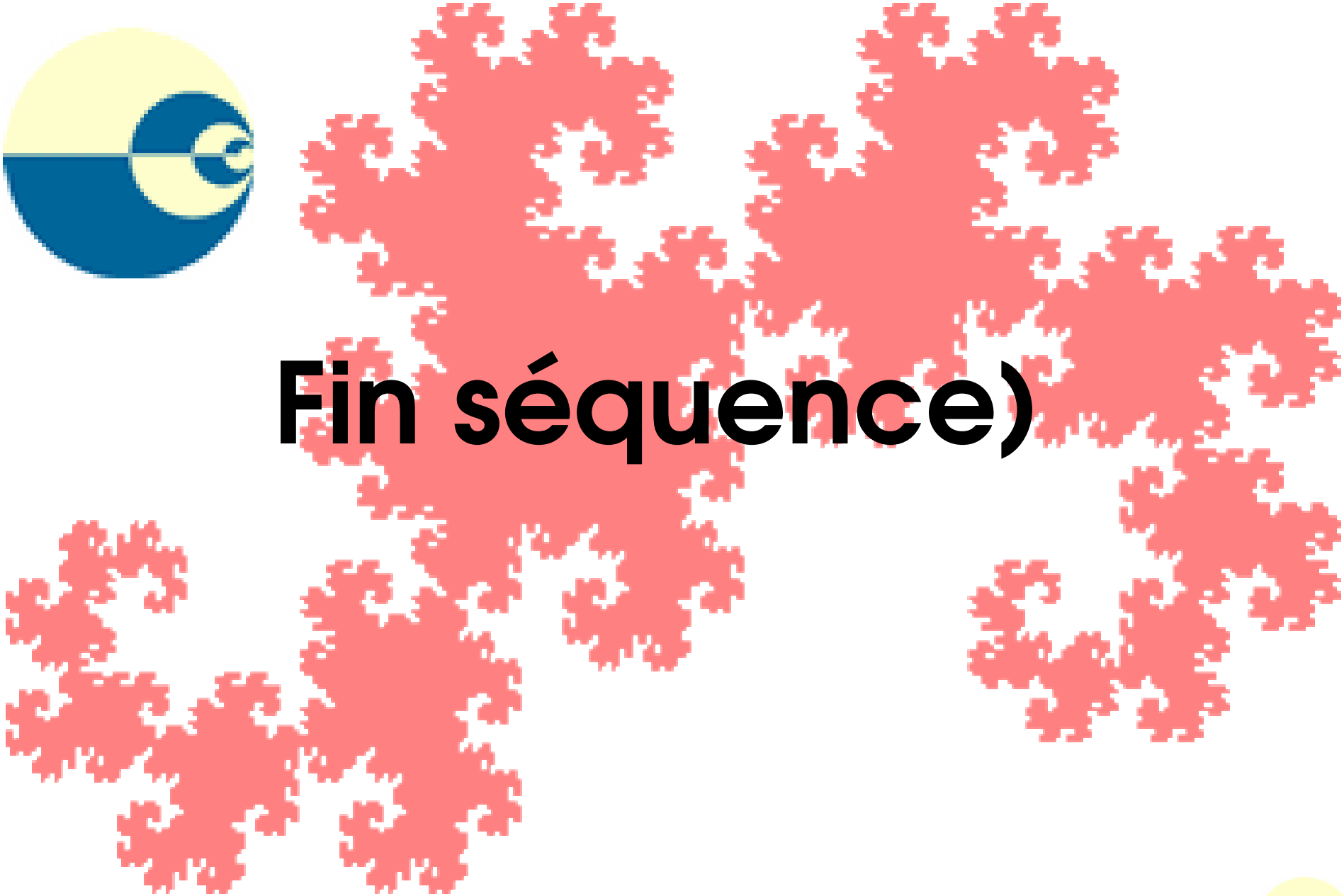
Fin séquence)

(Séquence 3.6

Résumé

Résumé

- Vous savez maintenant comment fonctionnent les récursions sur des entiers naturels.
- Enfin vous savez que certaines récursions sont plus efficaces que d'autres.



Fin séquence)

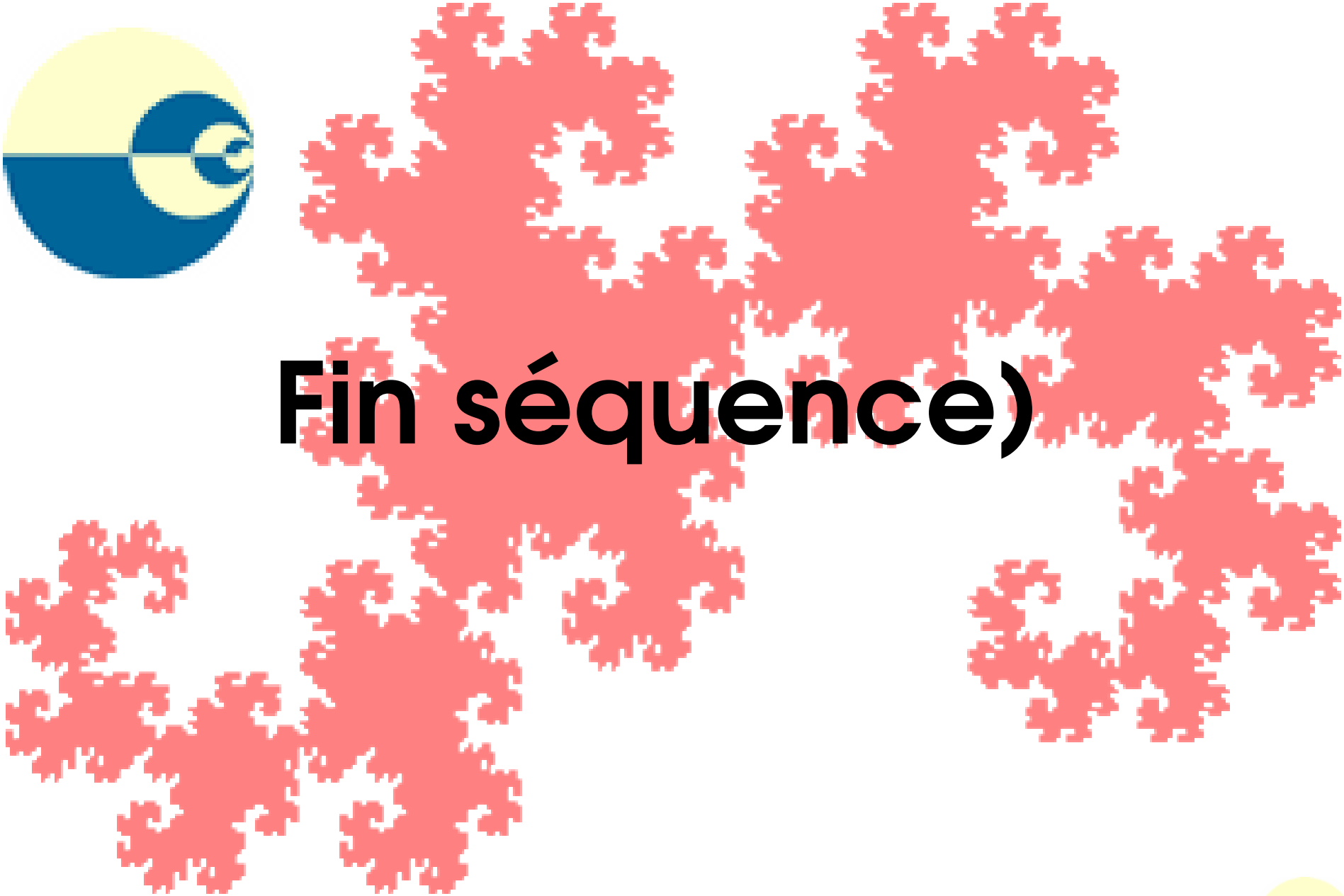
(Séquence 4.0

= Plan semaine 4

Plan semaine 4

Les listes et la récursion sur les listes

- Définition d'une liste
- Primitives sur les listes
  - Constructeurs
  - Accesseurs
  - Reconnaisseurs
- Définition de fonctions récursives sur les listes
  - somme des termes d'une liste
  - longueur d'une liste
  - typage



Fin séquence)

(Séquence 4.1

Récursion sur listes

Définition d'une liste

Une ***liste*** est une ***structure de données*** qui regroupe une séquence d'éléments de même type.
En termes plus informatiques, une liste est une ***collection homogène ordonnée***.
Le type d'une telle liste se note : **LISTE(α)**

- Quelques listes :

```
(10 12 16 14)                LISTE[nat]
("ma" "me" "mes")            LISTE[string]
((10 12) (16 14 -1))         LISTE[LISTE[int]]
```

- La liste vide représentée par : `()` et se prononce « nil »

Structure de données

Trois familles de fonctions pour manipuler toute structure de données :

- Les ***constructeurs*** permettent de construire une donnée structurée
- Les ***accesseurs*** permettent d'accéder aux composants d'une donnée structurée
- Les ***reconnaisseurs*** permettent de reconnaître la nature d'une donnée structurée

Primitives relatives aux listes

- **Constructeurs** pour construire une liste : `list`, `cons`
- **Accesseurs** pour accéder aux parties d'une liste : `car`, `cdr` (prononcer « coudeur »)
- **Reconnaisseur** (prédicat) pour savoir si une valeur est une liste non vide : `pair?`

Deux constructeurs `list` et `cons`

La fonction `list` (cf. carte de référence)

```
;;; list: alpha * ... -> LISTE[alpha]
;;; (list v...) cree une liste dont les termes
;;; sont les arguments. (list) rend la liste vide
```

`list` est une fonction ***n-aire***, elle prend un nombre quelconque, non fixé, d'arguments

LISTE(int)

```
(list 1 2 (+ 2 1) (/ 8 2))
      → (1 2 3 4)
```

LISTE(string)

```
(list "je" "tu" (string-append "el" "le") "il")
      → ("je" "tu" "elle" "il")
```

LISTE(bool)

```
(list (= 2 3) (not #f) (> 2 3))
      → (#f #t #f)
```

Constructeur `cons`

Une liste est (récursivement) définie comme :

- la liste vide
- ou une liste non vide c'est-à-dire constituée :
  - d'un premier terme
  - et d'autres termes qui forment aussi une liste (vide ou pas).

```
;;; cons: alpha * LISTE[alpha] -> LISTE[alpha]
;;; (cons v L) rend la liste dont le premier élément
;;; est v et dont les éléments suivants sont les
;;; éléments de la liste L.
```

```
(cons (+ 5 5) (list 20 30 40))
      → (10 20 30 40)
(cons 10 (cons 20 (cons 30 (cons 40 (list)))))
      → (10 20 30 40)
```

Les accesseurs

```
;;; car: LISTE[alpha] -> alpha
;;; (car L) rend le premier élément de la liste L
;;; ERREUR lorsque L n'est pas une liste non vide

;;; cdr: LISTE[alpha] -> LISTE[alpha]
;;; (cdr l) rend la liste des termes de L sauf son
;;; premier élément.
;;; ERREUR lorsque L n'est pas une liste non vide

(car (list 10 20 30 40))
       → 10
(cdr (list 10 20 30 40))
       → (20 30 40)
```

Propriétés algébriques

- Pour toute liste L et toute valeur v

```
(car (cons v L)) ≡ v
(cdr (cons v L)) ≡ L
```

- Pour toute liste **non vide** L

```
(cons (car L) (cdr L)) ≡ L
```

Le reconnaisseur `pair?`

Pour savoir si une valeur est une liste non vide : le prédicat `pair?`

```
;;; pair?: valeur -> bool
;;; (pair? v) rend vrai ssi v a un car et un cdr,
;;; c'est-à-dire ssi v est une liste non vide.
```

```
(pair? (list 10 20 30 40))
      → #t
(pair? (list))
      → #f
```

Remarque super-importante

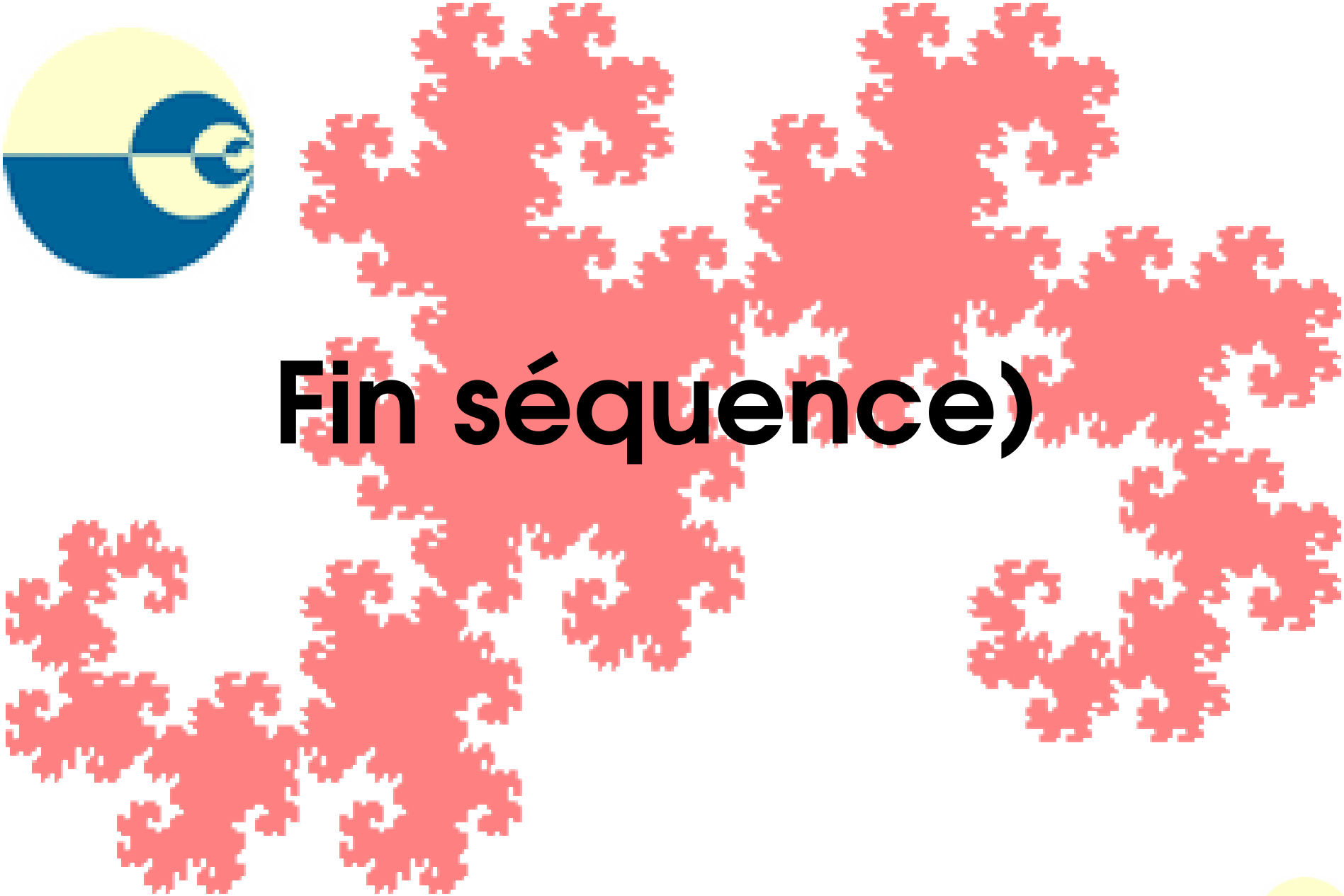
Jamais `car` (ou `cdr`) ne prendras
sans que de `pair?` ne t'assureras !

Si l'on sait que `L` vérifie `pair?` alors on peut directement écrire

```
... (car L) ...
```

sinon on doit écrire :

```
(if (pair? L)
    ... (car L) ...
    ... )
```



Fin séquence)

(Séquence 4.2

Somme des termes d'une liste

Somme des termes d'une liste

Sa spécification :

```
;;; somme: LISTE[Nombre] -> Nombre
;;; (somme L) rend la somme des éléments de L,
;;; rend 0 pour la liste vide
```

Somme des termes d'une liste II

- Lorsque la liste donnée n'est pas vide : la somme de ses éléments est égale au premier élement `(car L)` **plus** la somme des éléments du `cdr` de la liste

```
(somme (list e1 e2 ...en))
  ≡ (+ e1 (somme (list e2 ...en)))
```

- Lorsque la liste donnée est vide : la somme de ses éléments est égale à 0, par convention

```
(somme (list)) ≡ 0
```

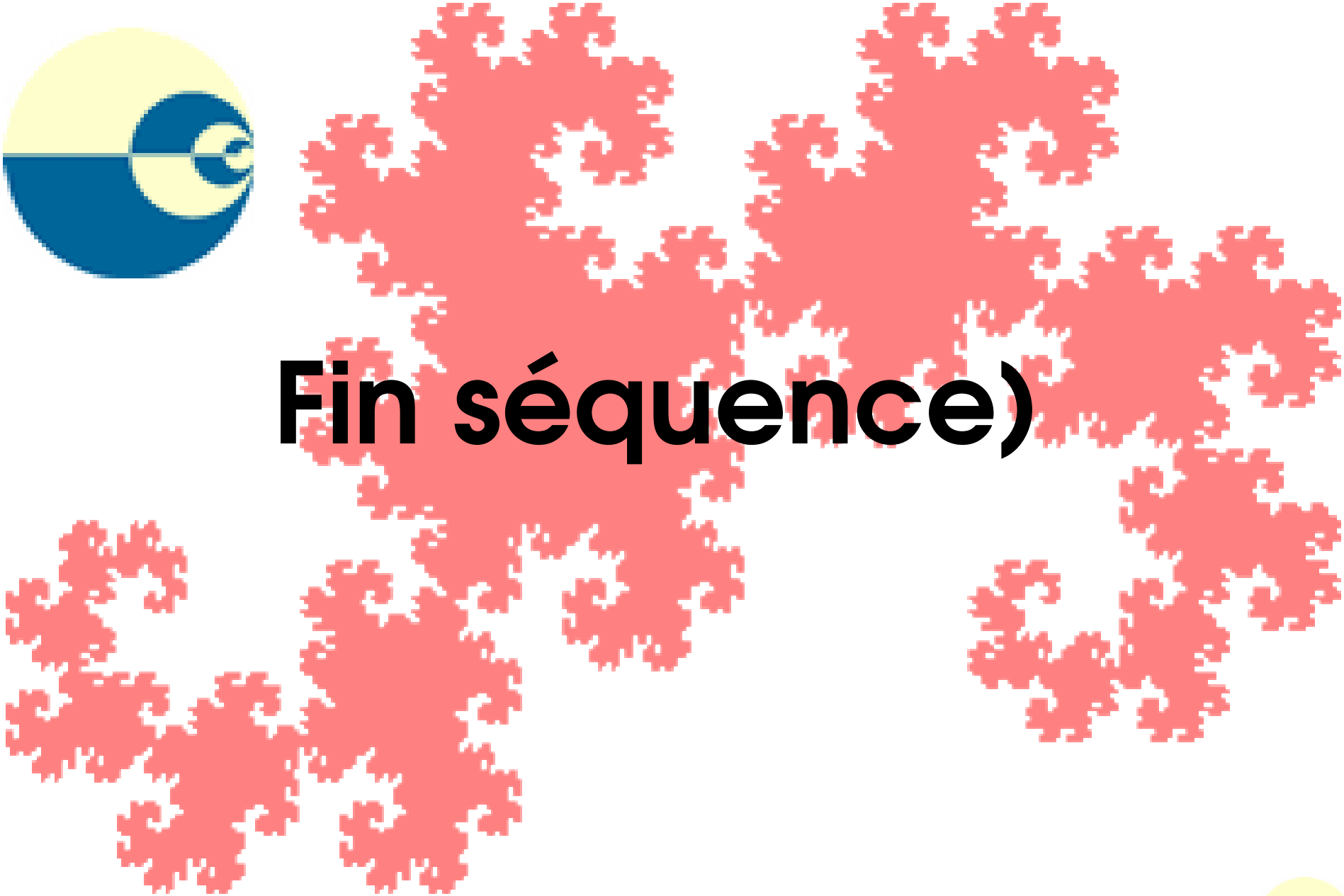
Une définition Scheme de somme

```
;;; somme: LISTE[Nombre] -> Nombre
;;; (somme L) rend la somme des éléments de L,
;;; rend 0 pour la liste vide
(define (somme L)
  (if (pair? L)
      (+ (car L) (somme (cdr L)))
      0 ) )
```

Trace d'une évaluation

On évalue `(somme (list 1 4 6 20))`

```
|(somme (1 4 6 20))
| (somme (4 6 20))
| |(somme (6 20))
| | (somme (20))
| | |(somme ())
| | |0
| | 20
| |26
| 30
|31
```



Fin séquence)

(Séquence 4.3

Longueur d'une liste

Longueur d'une liste

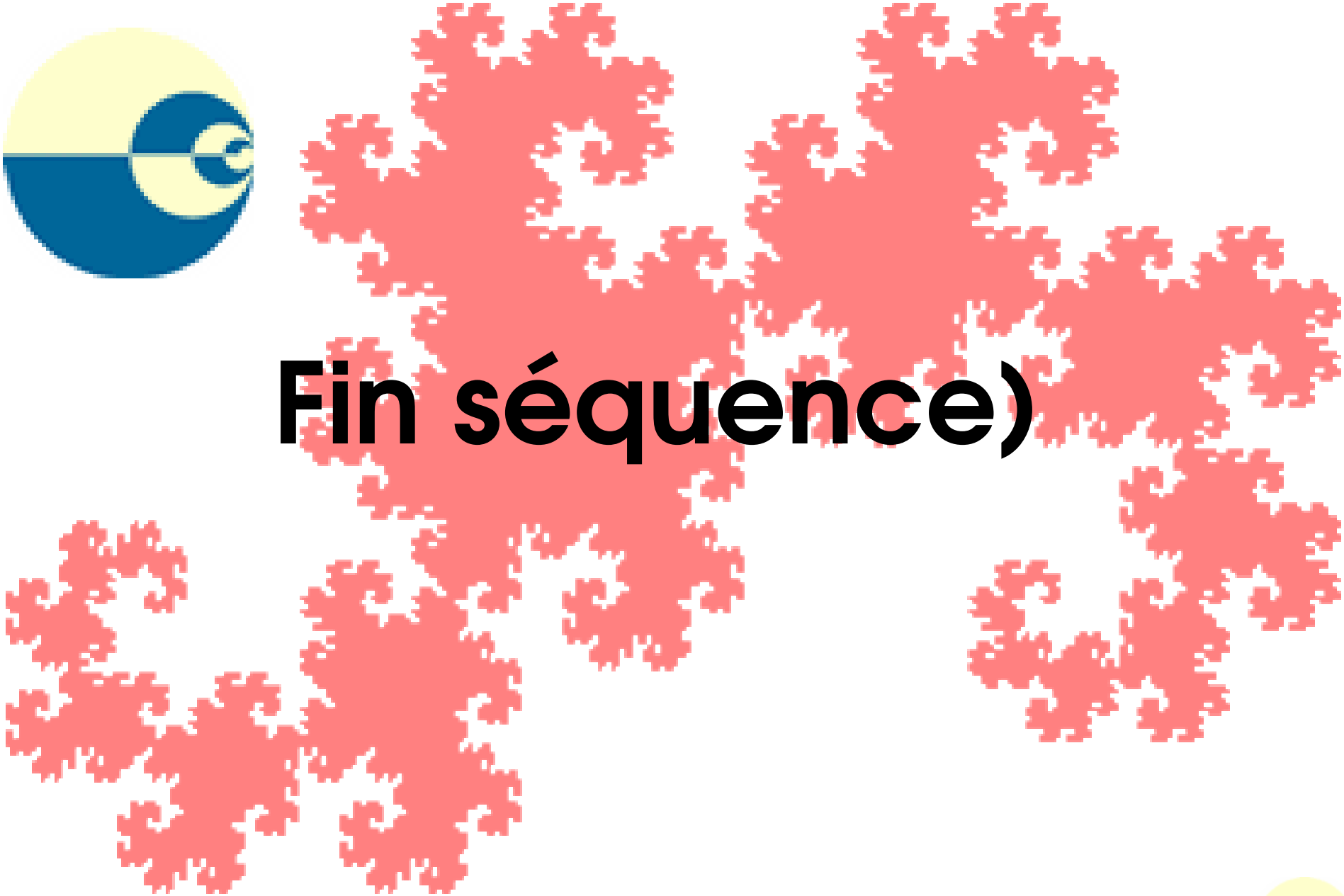
- Lorsque la liste donnée n'est pas vide :

```
(longueur (list e1 e2 ...en))
  ≡ (+ 1 (longueur (list e2 ...en)))
```

- Lorsque la liste donnée est vide : sa longueur est 0

```
(longueur (list)) ≡ 0
```

```
;;; longueur: LISTE[alpha] -> nat
;;; (longueur L) rend la longueur de la liste donnée
(define (longueur L)
  (if (pair? L)
      (+ 1 (longueur (cdr L)))
      0 ))
```



Fin séquence)

(Séquence 4.4

Typage

Typage d'une fonction

Le ***typage*** vérifie la **cohérence** entre

- Les types qui sont précisés dans la spécification
- Ce qui sera calculé par le code de la définition

Cf. équations aux dimensions en physique

Typage d'une fonction mystere001

```
(define (carre x)
  (* x x) )

carre: ???    -> ???
```

```
(define (mystere001 X)
  (if (pair? X)
      (+ (carre (car X)) (mystere001 (cdr X)))
      0 ) )

mystere001: ???            ->  ???
```

Typage d'une fonction mystere001

```
(define (carre x)
  (* x x) )

carre: ???    -> ???
carre: Nombre -> Nombre (du fait de *)
```

```
(define (mystere001 X)
  (if (pair? X)
      (+ (carre (car X)) (mystere001 (cdr X)))
      0 ) )

mystere001: ???            ->  ???
```

Typage d'une fonction mystere001

```
(define (carre x)
  (* x x) )

carre: ???    -> ???
carre: Nombre -> Nombre (du fait de *)
```

```
(define (mystere001 X)
  (if (pair? X)
      (+ (carre (car X)) (mystere001 (cdr X)))
      0 ) )

mystere001: ???              ->  ???
mystere001: ???              -> Nombre (car 0 et +)
```

Typage d'une fonction mystere001

```
(define (carre x)
  (* x x) )

carre: ???    -> ???
carre: Nombre -> Nombre (du fait de *)
```

```
(define (mystere001 X)
  (if (pair? X)
      (+ (carre (car X)) (mystere001 (cdr X)))
      0 ) )

mystere001: ???              ->  ???
mystere001: ???              -> Nombre (car 0 et +)
mystere001: LISTE[???]       -> Nombre (car car et cdr)
```

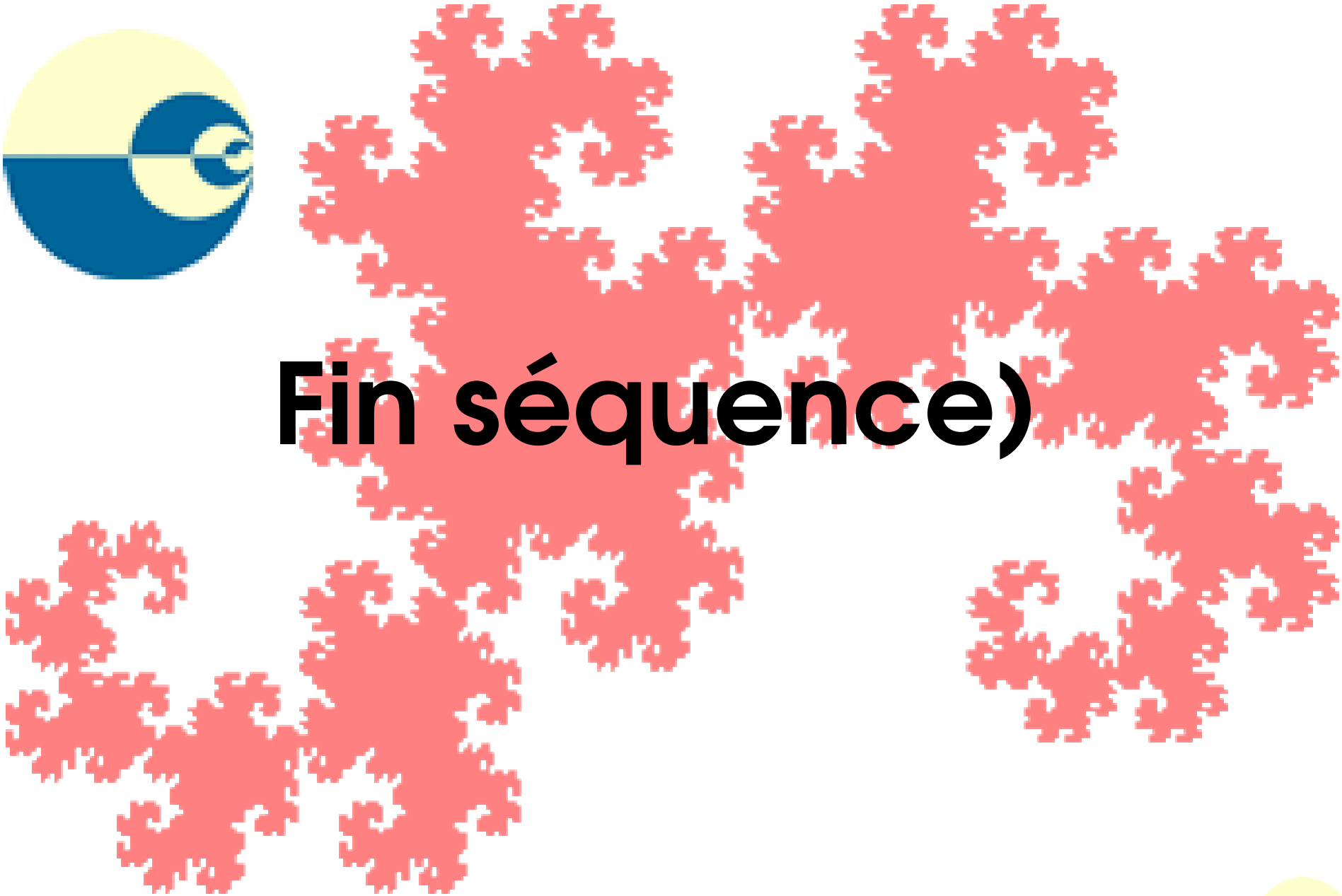
Typage d'une fonction mystere001

```
(define (carre x)
  (* x x) )

carre: ???    -> ???
carre: Nombre -> Nombre (du fait de *)
```

```
(define (mystere001 X)
  (if (pair? X)
      (+ (carre (car X)) (mystere001 (cdr X)))
      0 ) )

mystere001: ???              ->  ???
mystere001: ???              -> Nombre (car 0 et +)
mystere001: LISTE[???]       -> Nombre (car car et cdr)
mystere001: LISTE[Nombre]    -> Nombre (car carre)
```

Somme des carrés d'une liste

```
;;; carre: Nombre -> Nombre
;;; (carre n) rend le carré du nombre n
(define (carre n)
  (* n n) )

;;; somme-carres: LISTE[Nombre] -> Nombre
;;; (somme-carres L) rend la somme des carrés des
;;; éléments de L et rend 0 pour la liste vide
(define (somme-carres L)
  (if (pair? L)
      (+ (carre (car L)) (somme-carres (cdr L)))
      0 ) )
```



Fin séquence)

(Séquence 4.5

Schéma de récursion

Schéma usuel de récursion

Une liste est :

- soit vide
- soit constituée d'un premier *élément* `(car L)` et d'une *liste* restante `(cdr L)`

```
;;; fRec: LISTE[alpha] -> ...
(define (fRec L)
  (if (pair? L)
      (combinaison
        (car L)
        (fRec (cdr L)) )
      cas-liste-vide ) )
```

Méthode d'écriture

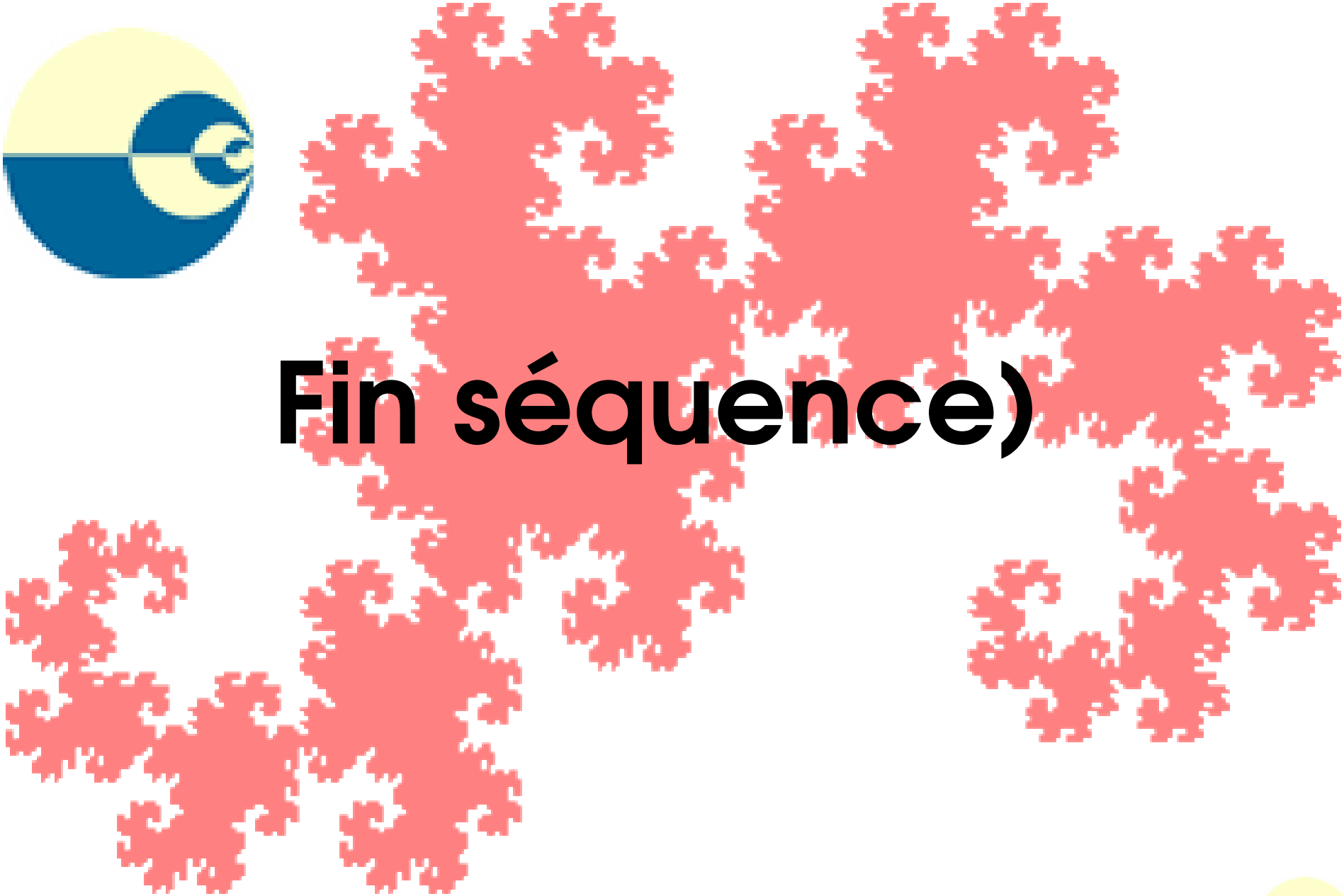
Pour écrire une fonction récursive, il est très fortement recommandé de suivre la méthode suivante :

1. décrire le calcul à partir du calcul sur des éléments « plus petits » (relation de récurrence),
2. déterminer les valeurs dites « valeurs de base » pour lesquelles :
   - 2.1 la relation de récurrence n'est pas définie,
   - 2.2 les valeurs d'appel de la fonction récursive, pas **strictement** « plus petites » que la donnée.
3. déterminer les valeurs de la fonction pour les valeurs de base.

Arrêt de récursion

Pour que les récursions s'achèvent, il faut

1. une « diminution » dans l'argument de l'appel récursif
   - Si la récursivité se fait sur les entiers naturels, la valeur de l'argument de l'appel récursif doit diminuer (par exemple $n-1$, ou $n/2$, ou ...)
   - Si la récursivité se fait sur une liste, la longueur de la liste passée en argument de l'appel récursif doit être plus courte (par exemple `(cdr L)`).
2. un test d'arrêt sur les valeurs de base



Fin séquence)

(Séquence 4.6

Exemples commentés

Quelques exemples illustrant

- Récursion sur des listes (append, ajout-en-fin, somme-cumulee)
- Complexité

Buts

Les spécifications des fonctions à écrire :

```
;;; ajout-en-fin: alpha * LISTE[alpha]
;;;                    -> LISTE[alpha]
;;; (ajout-en-fin x L ) rend la liste obtenue en
;;; ajoutant x à la fin de la liste L
```

```
;;; append: LISTE[alpha] * LISTE[alpha]
;;;                    -> LISTE[alpha]
;;; (append L1 L2) rend la concaténation de L1 et
;;; de L2
```

Ajout suffixe

```
;;; ajout-en-fin: alpha * LISTE[alpha]
;;;         -> LISTE[alpha]
;;; (ajout-en-fin x L ) rend la liste obtenue en
;;; ajoutant x à la fin de la liste L

(define (ajout-en-fin x L)
  (if (pair? L)
      (cons (car L)
            (ajout-en-fin x (cdr L)) )
      (list x) ) )
```

Quelquefois nommée `snoc` !

Complexité de l'ajout suffixe

On mesure ici le nombre d'appels à `cons` (qui consomme de la mémoire)

```
(ajout-en-fin 4 (list 1 2 3) )

|(ajout-en-fin 4 (1 2 3))
|  (ajout-en-fin 4 (2 3))
|  |(ajout-en-fin 4 (3))
|  |  (ajout-en-fin 4 ())
|  |  (4)
|  |(3 4)
|  (2 3 4)
|(1 2 3 4)
```

Si `L` a n termes, `snoc` (c'est-à-dire `ajout-en-fin`) effectue $n+1$ appels à `cons`. Le coût est donc ***linéaire*** alors que le coût de `cons` est constant (mais pas nul !).

Concaténation de listes

```
;;; append: LISTE[alpha] * LISTE[alpha]
;;;                     -> LISTE[alpha]
;;; (append L1 L2) rend la concaténation de L1 et
;;; de L2

(define (append L1 L2)
  (if (pair? L1)
      (cons (car L1)
            (append (cdr L1) L2))
      L2))
```

Complexité de la concaténation de listes

```
(append (list 1 2 3) (list 5 6 7 8))

|(append (1 2 3) (5 6 7 8))
| (append (2 3) (5 6 7 8))
| |(append (3) (5 6 7 8))
| | (append () (5 6 7 8))
| | (5 6 7 8)
| |(3 5 6 7 8)
| (2 3 5 6 7 8)
|(1 2 3 5 6 7 8)
```

Si `L1` a *n* termes, `append` effectue *n* appels à `cons`. Le coût est donc **linéaire** sur le premier argument (indépendamment de la longueur du second argument).

Concaténation de listes (variante)

et si la récursion portait plutôt sur `L2` ?

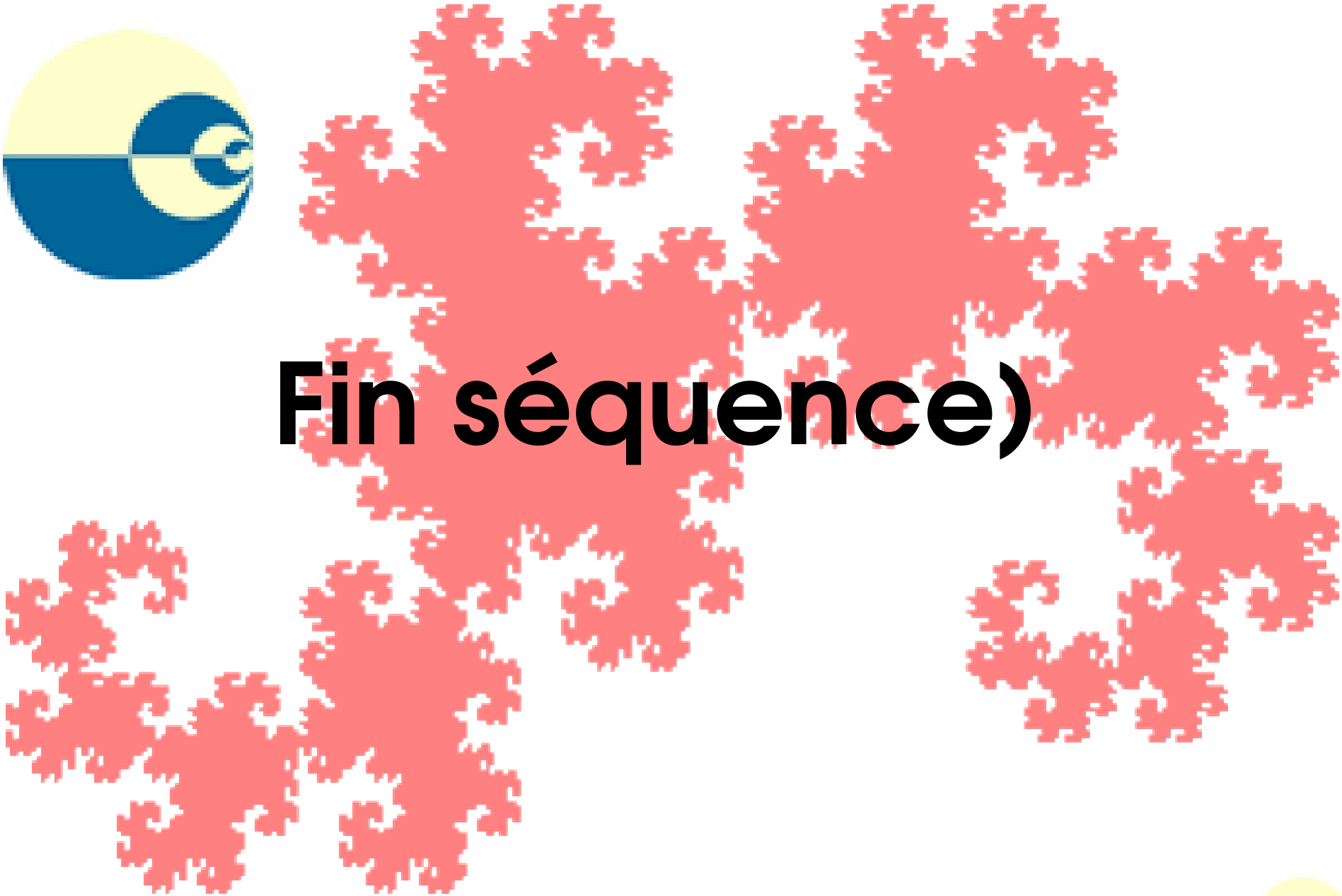
```
(define (appendr L1 L2)
  (if (pair? L2)
      (appendr (ajout-en-fin (car L2) L1)
               (cdr L2) )
      L1 ) )
```

Complexité de la concaténation de listes (variante)

La trace est trop longue pour être montrée. Vous pouvez l'obtenir vous-même avec MrScheme.

Si `L1` et `L2` ont n termes alors `appendr` effectue n appels à `ajout-en-fin` sur des listes de taille n puis $n+1$ puis $n+2$, etc. Le coût est donc, en nombre de `cons`, $\Sigma_{i=1}^{n}(n+i)$ c'est-à-dire de l'ordre de n^2 donc ***quadratique***.

Il faut donc faire attention au sens selon lequel on traite les listes !



Fin séquence)

(Séquence 4.7

somme-cumulee

Un exemple plus complexe

```
;;; somme-cumulee: LISTE[Nombre] -> LISTE[Nombre]
;;; (somme-cumulee L) rend la liste dont le premier
;;; élément est égal à la somme des éléments de L,
;;; dont le deuxième élément est égal à la somme
;;; des éléments de (cdr L) ... dont le dernier
;;; élément est égal au dernier élément de L.

;;; chaque élément de la liste résultat est égal à
;;; la somme des éléments qui le suivent (au sens
;;; large) dans la liste initiale

(somme-cumulee (list 4))
   → (4)
(somme-cumulee (list 3 4))
   → (7 4)
(somme-cumulee (list 2 3 4))
   → (9 7 4)
```

Une définition à proscrire

```
(define (somme-cumulee L)                    A NE PAS IMITER
  (if (pair? L)
      (cons (somme L)
            (somme-cumulee (cdr L)))
      (list)))
```

Double balayage de la liste `L` !
Si `L` comporte n termes, `somme` a une complexité linéaire (en nombre d'appels à `cdr`). Donc `somme-cumulee` consomme $n + (n-1) + (n-2) + \ldots + 1$ appels à `cdr` de l'ordre de n^2 appels donc une complexité quadratique.

Une autre définition à proscrire

```
(define (somme-cumulee L)
  (if (pair? L)
      (if (pair? (cdr L))
          (cons (+ (car L)
                   (car (somme-cumulee (cdr L))))
                (somme-cumulee (cdr L)))
          L)
      (list)))
```

A NE PAS IMITER

Nommer pour éviter des recalculs !

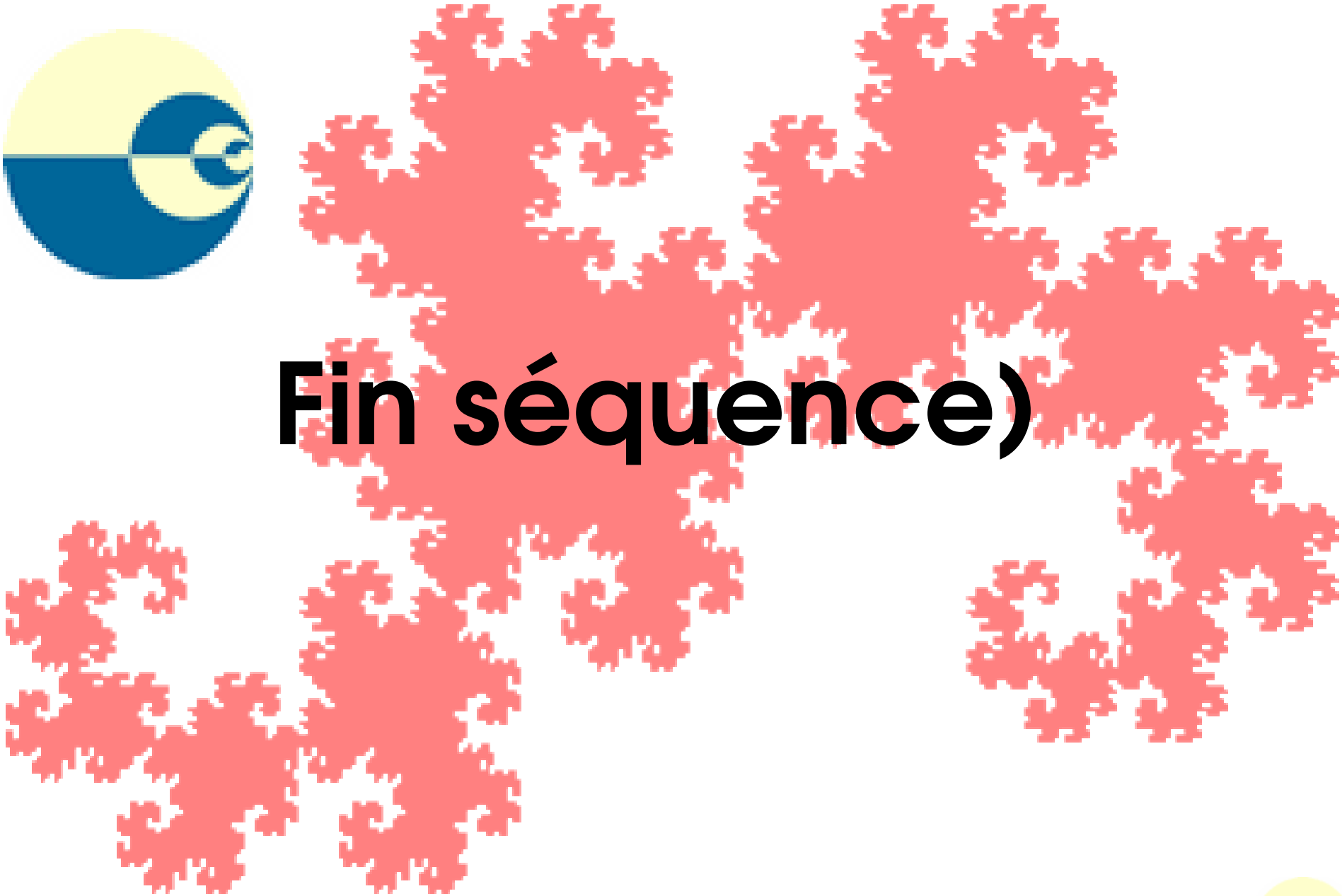
Une première définition

```
(define (somme-cumulee L)
  (if (pair? L)
      (if (pair? (cdr L))
          (let ((reste-fait (somme-cumulee (cdr L))))
            (cons (+ (car L) (car reste-fait))
                  reste-fait))
          L)
      (list)))
```

Définition interne pour éviter des tests

```
(define (somme-cumulee L)
  ;; sc-non-vide: LISTE[Nombre] -> LISTE[Nombre]
  ;; (sc-non-vide L) == (somme-cumulee L)
  ;; HYPOTHÈSE: L non vide
  (define (sc-non-vide L)
    (if (pair? (cdr L))
        (let ((reste-fait (sc-non-vide (cdr L))))
          (cons (+ (car L) (car reste-fait))
                reste-fait))
        L))
  (if (pair? L)
      (sc-non-vide L)
      L))
```

Deux fois moins d'appels à `pair?`.



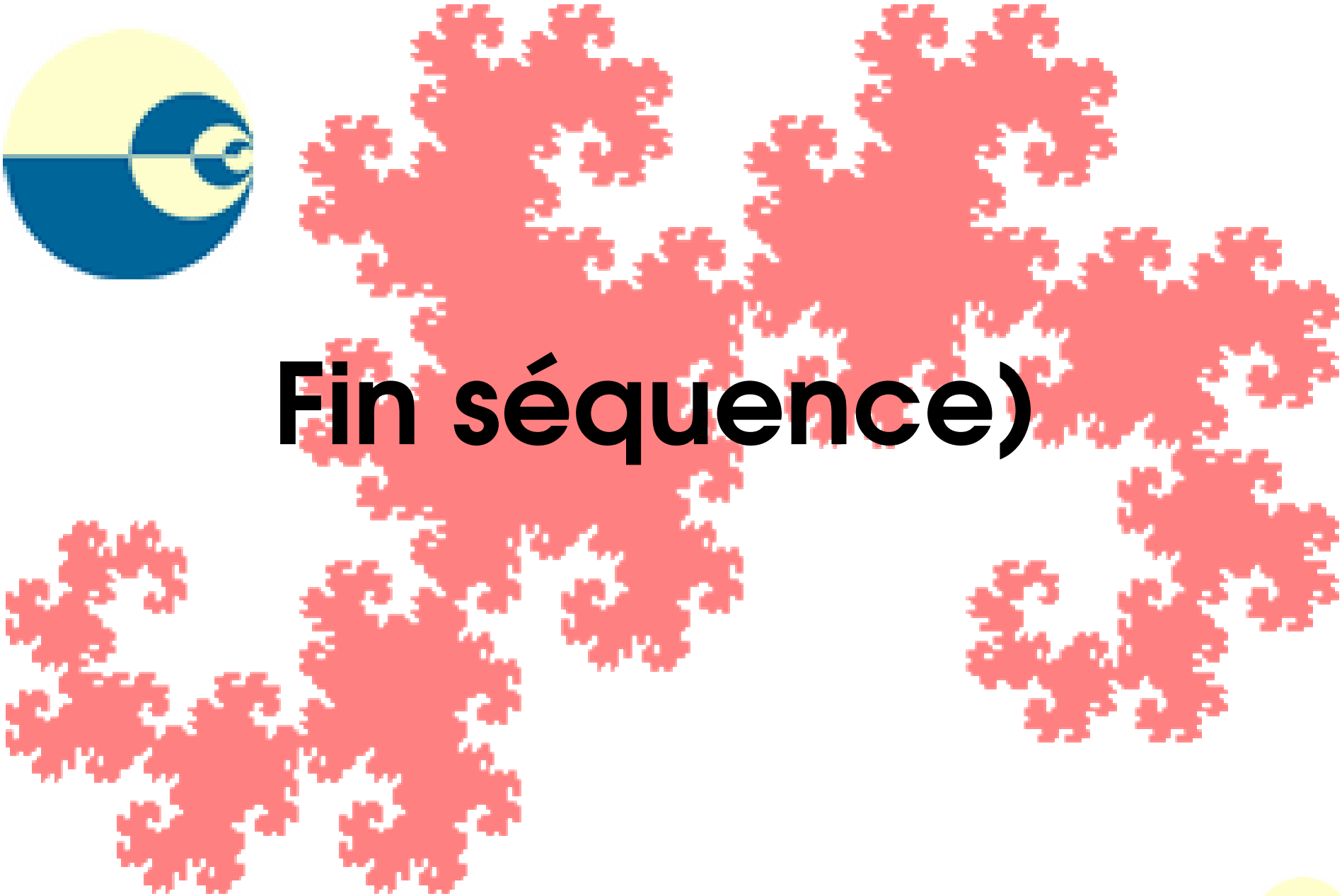
Fin séquence)

(Séquence 4.8

Résumé

Résumé

- Vous avez découvert votre première structure de donnée : la liste
- qui peut contenir un nombre quelconque d'éléments.
- Vous savez les construire, les démembrer et reconnaître si elles sont vides ou pas.
- Vous savez maintenant comment fonctionnent les récursions sur les listes.
- Vous savez vérifier le bon typage des fonctions que vous écrivez.



Fin séquence)

(Séquence 5.0

= Plan semaine 5

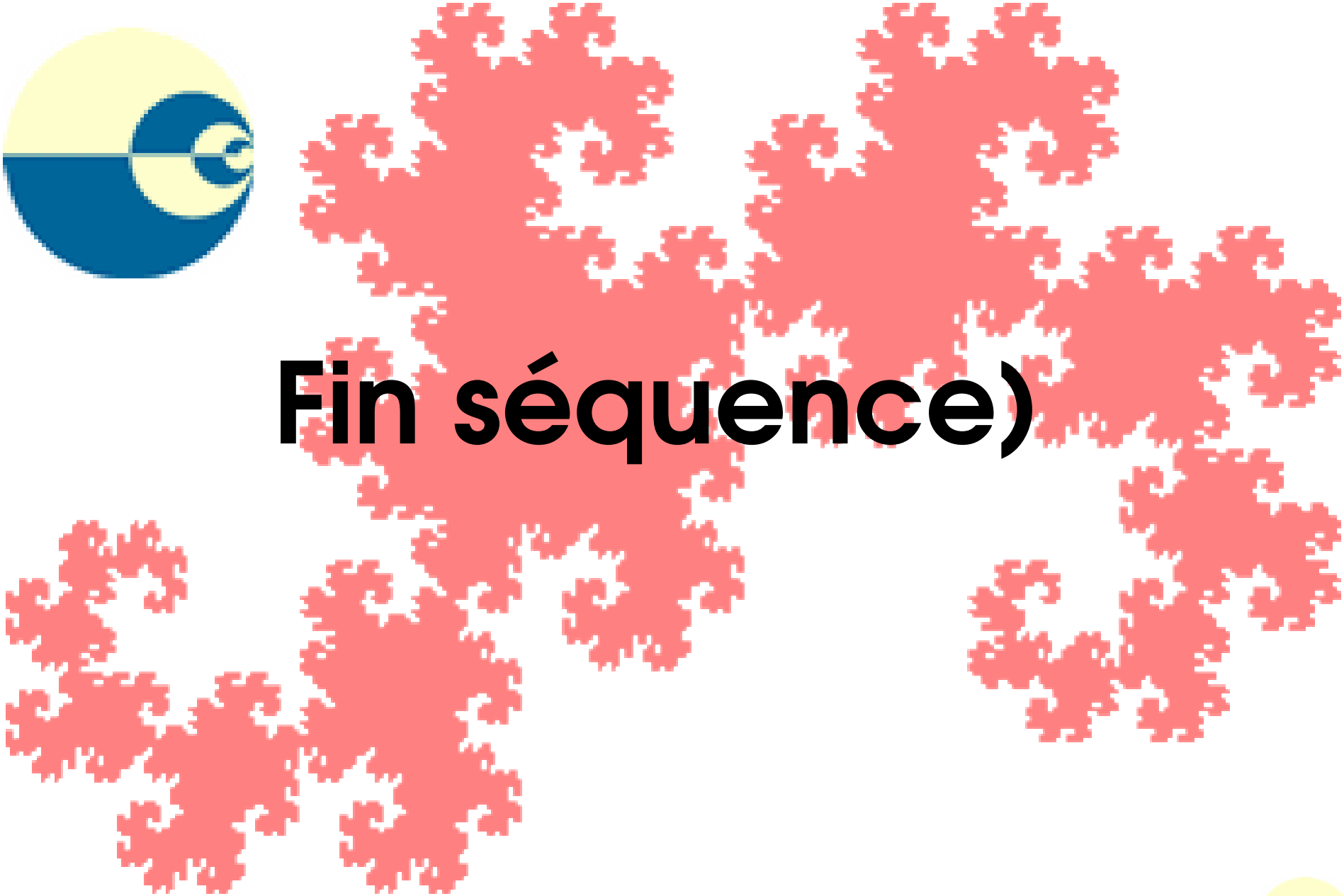
Plan semaine 5

Fonctionnelles sur les listes

- La fonctionnelle `map`
- La fonctionnelle `filtre`
- La fonctionnelle `reduce`

Fonctionnelles

On nomme ***fonctionnelle***, une fonction qui prend une fonction en argument ou renvoie une fonction en résultat.



Fin séquence)

(Séquence 5.1

map

Appliquer une fonction à une liste

- Présentation de plusieurs fonctions différentes
  - `liste-carres`
  - `liste-racines-carrees`
  - `liste-positive?`
- Le schéma récursif commun
- Passer la fonction en paramètre : la fonctionnelle `map`

Fonction `liste-carres`

```
;;; liste-carres: LISTE[Nombre] -> LISTE[Nombre]
;;; (liste-carres L) rend la liste des carrés
;;; des éléments de L.
(define (liste-carres L)
  (if (pair? L)
      (cons (carre (car L))
            (liste-carres (cdr L)) )
      (list) ) )
```

```
(liste-carres (list 1 2 3 4 5 6))
   → (1 4 9 16 25 36)
```

Fonction `liste-racines-carrees`

```
;;; liste-racines-carrees:
;;;      LISTE[Nombre] -> LISTE[Nombre]
;;; (liste-racines-carrees L) rend la liste des
;;; racines carrées des éléments de L.
;;; HYPOTHÈSE: les nombres de L sont positifs
(define (liste-racines-carrees  L)
  (if (pair? L)
      (cons (sqrt (car L))
            (liste-racines-carrees (cdr L)) )
      (list)))
```

```
(liste-racines-carrees (list 1 4 9 16 25))
   → (1 2 3 4 5)
```

Fonction `liste-positive?`

```
;;; liste-positive?: LISTE[Nombre] -> LISTE[bool]
;;; (liste- positive? L) rend la liste des booléens
;;; qui indiquent pour chaque élément de L s'il est
;;; positif.
(define (liste-positive?  L)
  (if (pair? L)
      (cons (positive? (car L))
            (liste-positive? (cdr L)) )
      (list)))
```

```
(liste-positive? (list 5 -9 0 4 8 -7))
  → (#t #f #f #t #t #f)
```

Vers un schéma de fonction

```
(define (liste-carres L)
  (if (pair? L)
      (cons (carre (car L))
            (liste-carres (cdr L)) )
      (list) ) )

(define (liste-racines-carrees  L)
  (if (pair? L)
      (cons (sqrt (car L))
            (liste-racines-carrees (cdr L)) )
      (list)))

(define (liste-positive?  L)
  (if (pair? L)
      (cons (positive? (car L))
            (liste-positive? (cdr L)) )
      (list)))
```

Un schéma de fonction récursive

Les trois définitions précédentes suivent le même **schéma récursif** :

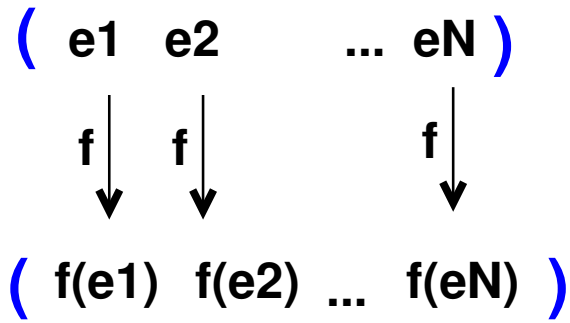
```
(define (fn-sur-liste L)
  (if (pair? L)
      (cons (fn-sur-elem (car L))
            (fn-sur-liste (cdr L)) )
      (list) ) )
```

La seule différence est la fonction *fn-sur-elem* à appliquer.
Passer la fonction *fn-sur-elem* en argument, c'est ***abstraire*** !

Une fonctionnelle `map`

(cf. carte de référence)

```
;;; map: (alpha -> beta)
;;;      * LISTE[alpha] -> LISTE[beta]
;;; (map fn L) rend la liste dont les éléments
;;; résultent de l'application de la fonction fn
;;; aux éléments de L
(define (map fn L)
  (if (pair? L)
      (cons (fn (car L)) (map fn (cdr L)))
      (list)))
```



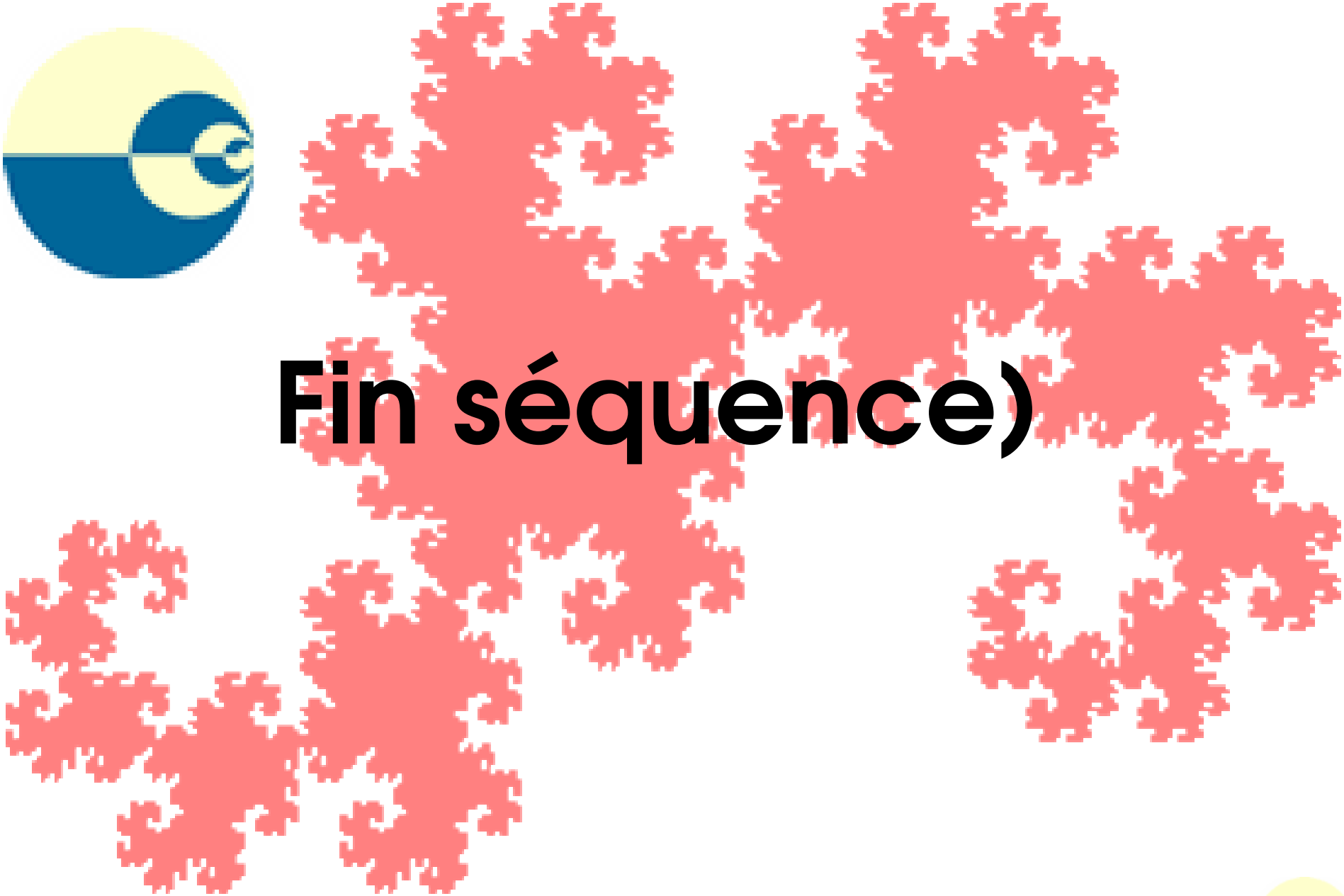
Applications de la fonction `map`

```
(liste-carres (list 1 2 3 4))
  ≡ (map carre (list 1 2 3 4))

(liste-racines-carrees (list 16 25 36))
  ≡ (map sqrt (list 16 25 36))
```

```
(map f (list e1 e2 ... eN))
  ≡ (list (f e1) (f e2) ... (f eN))
```

fonction interne et `map`

```
;;; verif-entre-bornes :
;;;  Nombre * Nombre * LISTE[Nombre] -> LISTE[bool]
;;; (verif-entre-bornes borneInf borneSup L) rend
;;; une liste de booléens vérifiant si les termes
;;; de L sont entre borneInf et borneSup

(define (verif-entre-bornes borneInf borneSup L)
  ;; entre-bornes? : Nombre -> bool
  ;; (entre-bornes? x) vérifie si x est entre
  ;; borneInf et borneSup
  (define (entre-bornes-inf-sup? x)
    (and (<= borneInf x) (<= x borneSup)))

  (map entre-bornes-inf-sup? L))
```



Fin séquence)

(Séquence 5.2

filtre

Filtrer une liste par un prédicat

Un autre problème : filtrer les éléments d'une liste par un prédicat *pred ?*

- soit *L1* une liste d'éléments de type *alpha*,
- et *pred ?* un prédicat : *alpha* $\rightarrow$ *bool* ;
- on applique *pred ?* sur chaque élément de la liste *L1*
- pour obtenir une liste *L2* qui contient UNIQUEMENT les éléments vérifiant *pred ?*

```
;;; filtre:
;;;  (alpha -> bool) * LISTE[alpha] -> LISTE[alpha]
;;; (filtre pred? L) rend la liste des éléments de
;;; L qui vérifient le prédicat pred?
```

La fonction `filtre-pairs`

```
;;; filtre-pairs: LISTE[int] -> LISTE[int]
;;; (filtre-pairs L) retourne la liste des éléments
;;; pairs de L.
(define (filtre-pairs L)
  (if (pair? L)
      (if (even? (car L))
          (cons (car L) (filtre-pairs (cdr L)))
          (filtre-pairs (cdr L)))
      (list)))
```

```
(filtre-pairs (list 1 2 3 5 8 6))
   → (2 8 6)
```

La fonction `filtre-impairs`

```
;;; filtre-impairs: LISTE[int] -> LISTE[int]
;;; (filtre-impairs L) retourne la liste des éléments
;;; impairs de L.
(define (filtre-impairs L)
  (if (pair? L)
      (if (odd? (car L))
          (cons (car L) (filtre-impairs (cdr L)))
          (filtre-impairs (cdr L)))
      (list)))
```

```
(filtre-impairs (list 1 2 3 5 8 6))
  → (1 3 5)
```

Vers un schéma de fonction

```
(define (filtre-pairs L)
  (if (pair? L)
      (if (even? (car L))
          (cons (car L) (filtre-pairs (cdr L)))
          (filtre-pairs (cdr L)))
      (list)))

(define (filtre-impairs L)
  (if (pair? L)
      (if (odd? (car L))
          (cons (car L) (filtre-impairs (cdr L)))
          (filtre-impairs (cdr L)))
      (list)))
```

Schéma de fonction `filtre`

Les deux définitions précédentes suivent le même schéma récursif :

```
(define (schema-filtre-pred L)
  (if (pair? L)
      (if (pred-sur-elem? (car L))
          (cons (car L)
                (schema-filtre-pred (cdr L)))
          (schema-filtre-pred (cdr L)))
      (list)))
```

Il reste à abstraire vis-à-vis du prédicat *pred-sur-elem ?*

La fonctionnelle `filtre`

```
;;; filtre:
;;;  (alpha -> bool) * LISTE[alpha] -> LISTE[alpha]
;;; (filtre pred? L) rend la liste des éléments de
;;; L qui vérifient le prédicat pred?
(define (filtre pred? L)
  (if (pair? L)
      (if (pred? (car L))
          (cons (car L) (filtre pred? (cdr L)))
          (filtre pred? (cdr L)))
      (list)))
```

```
(filtre even? (list 1 2 3 4 5))
  → (2 4)
(filtre odd? (list 1 2 3 4 5))
  → (1 3 5)
(filtre integer? (list 2 2.5 3 3.5 4))
  → (2 3 4)
```

Points communs entre `map` et `filtre`

```
;;; map:
;;;  (alpha -> beta) * LISTE[alpha] -> LISTE[beta]
;;; filtre:
;;;  (alpha -> bool) * LISTE[alpha] -> LISTE[alpha]
```

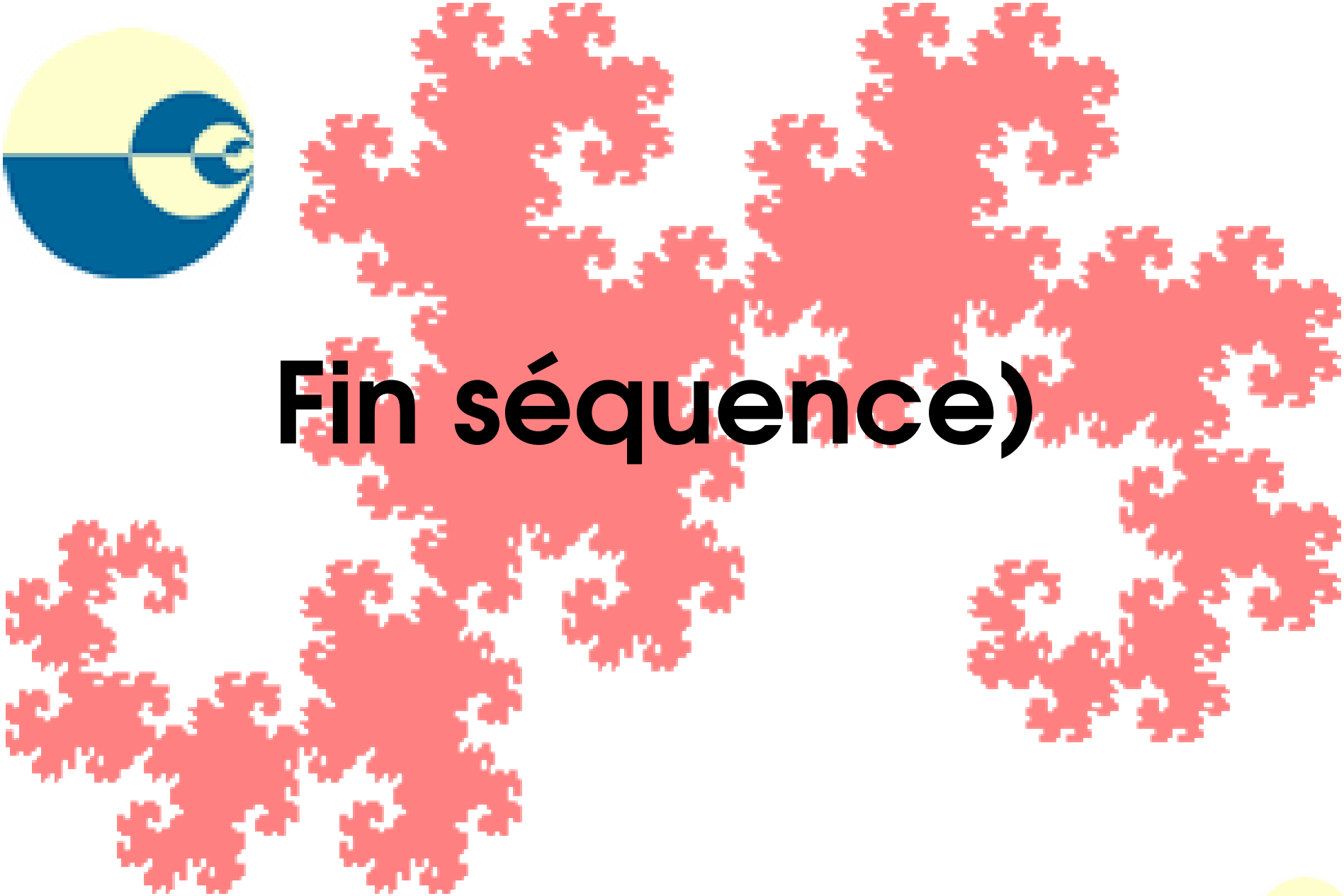
- Ce sont des fonctionnelles (elles reçoivent en argument une fonction)
- La fonction passée en premier argument a comme type de données le type des éléments de la liste de second argument
- Elles rendent une liste

Différences entre `map` et `filtre`

```
;;; map:
;;;  (alpha -> beta) * LISTE[alpha] -> LISTE[beta]
;;; filtre:
;;;  (alpha -> bool) * LISTE[alpha] -> LISTE[alpha]
```

- Le type du résultat de la fonction passée en argument : quelconque pour `map` et booléen pour `filtre`
- La longueur de la liste donnée et de la liste résultat (même longueur pour `map`)
- Les éléments de la liste résultat de filtre sont des éléments de sa liste donnée.

```
(map integer? (list 2 2.5 3 3.5 4))
  → (#t #f #t #f #t)
(filtre integer? (list 2 2.5 3 3.5 4))
  → (2 3 4)
```

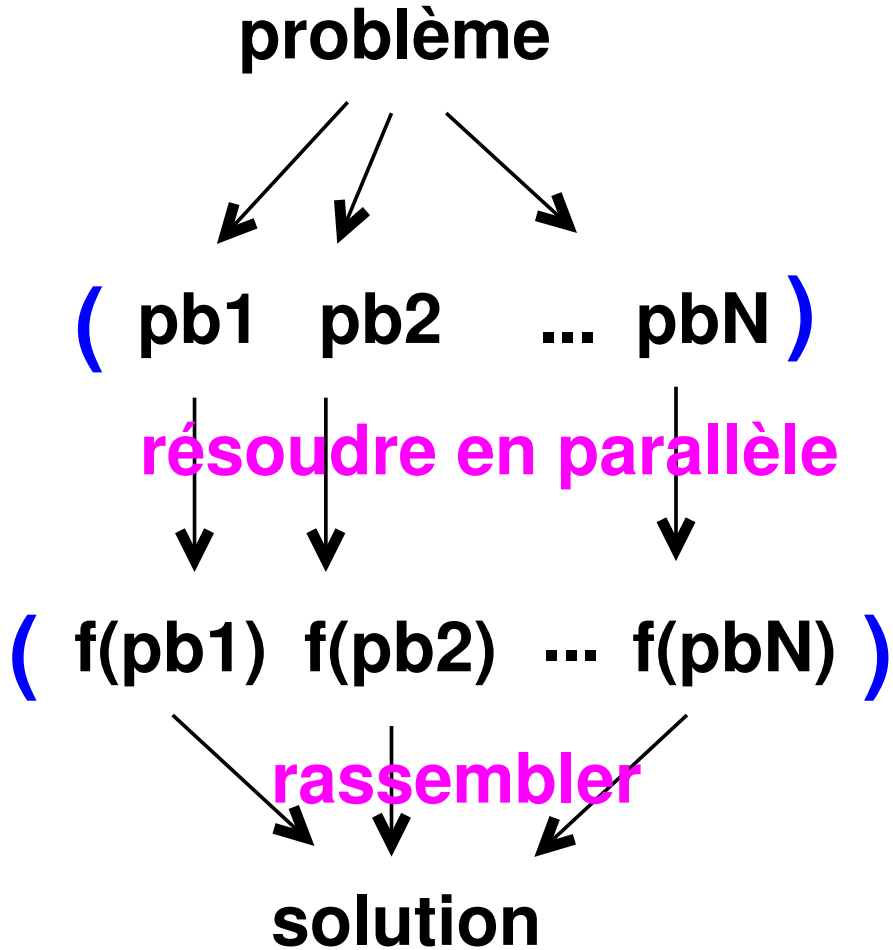


Fin séquence)

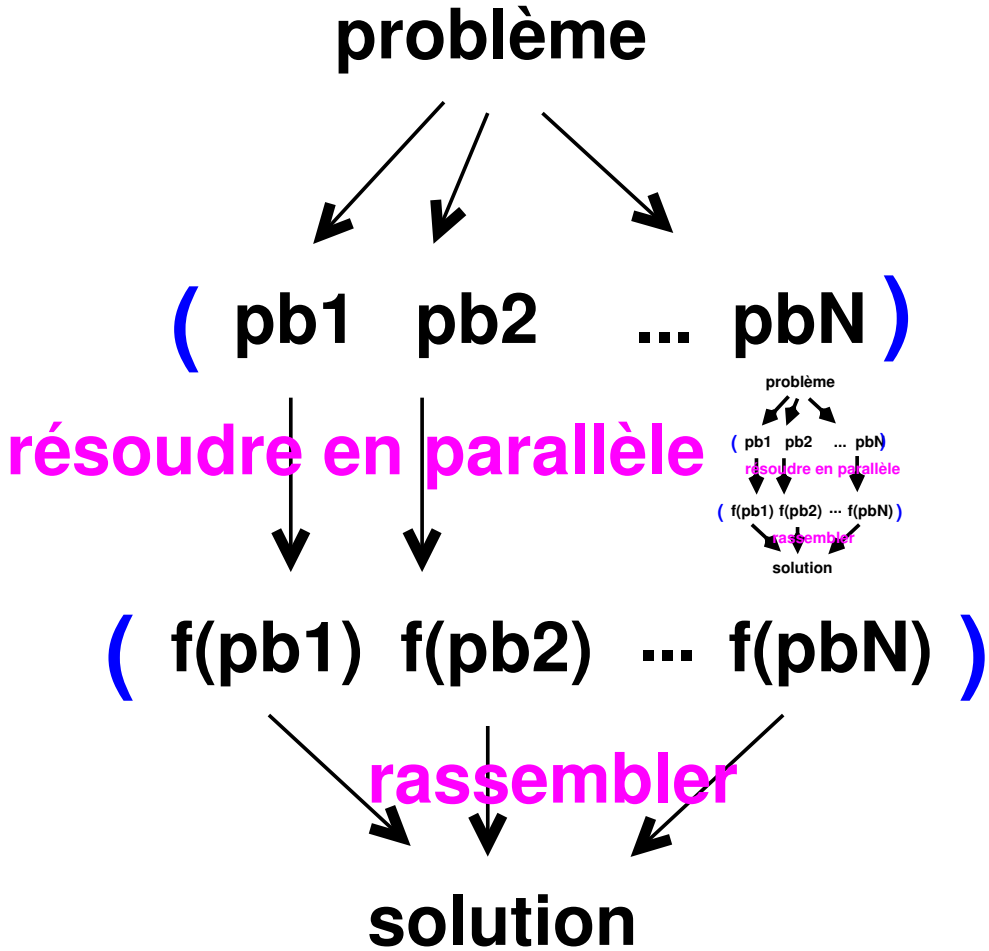
(Séquence 5.3

reduce

map-reduce


problème
(pb1 pb2 ... pbN)
résoudre en parallèle
(f(pb1) f(pb2) ... f(pbN))
rassembler
solution

map-reduce


problème
(pb1 pb2 ... pbN)
résoudre en parallèle
problème
(pb1 pb2 ... pbN)
résoudre en parallèle
(f(pb1) f(pb2) ··· f(pbN))
rassembler
solution
(f(pb1) f(pb2) ... f(pbN))
rassembler
solution

Combiner les éléments d'une liste

Combiner entre eux les éléments d'une liste, à l'aide d'une fonction binaire *fn*

- soit *L1* une liste d'éléments de type *alpha*,
- et *fn* une fonction : $alpha * beta \rightarrow beta$;
- on applique *fn* sur chaque élément de la liste *L1* (en démarrant avec un élément de base de type *beta*),
- pour obtenir un élément de type *beta*.

```
;;; reduce: (alpha * beta -> beta)
;;;         * beta * LISTE[alpha] -> beta
;;; (reduce fn base L) rend le résultat de
;;;       fn(e1, fn(e2, ... fn(en, base) ...))
```

La fonctionnelle `reduce`

Passer en paramètres : l'opérateur binaire et l'élément de base

```
;;; reduce: (alpha * beta -> beta)
;;;         * beta * LISTE[alpha] -> beta
;;; (reduce fn base L) rend
;;;   fn(e1, fn(e2, ... fn(en, base) ...))
(define (reduce fn base L)
  (if (pair? L)
      (fn (car L)
          (reduce fn base (cdr L)))
      base ) )
```

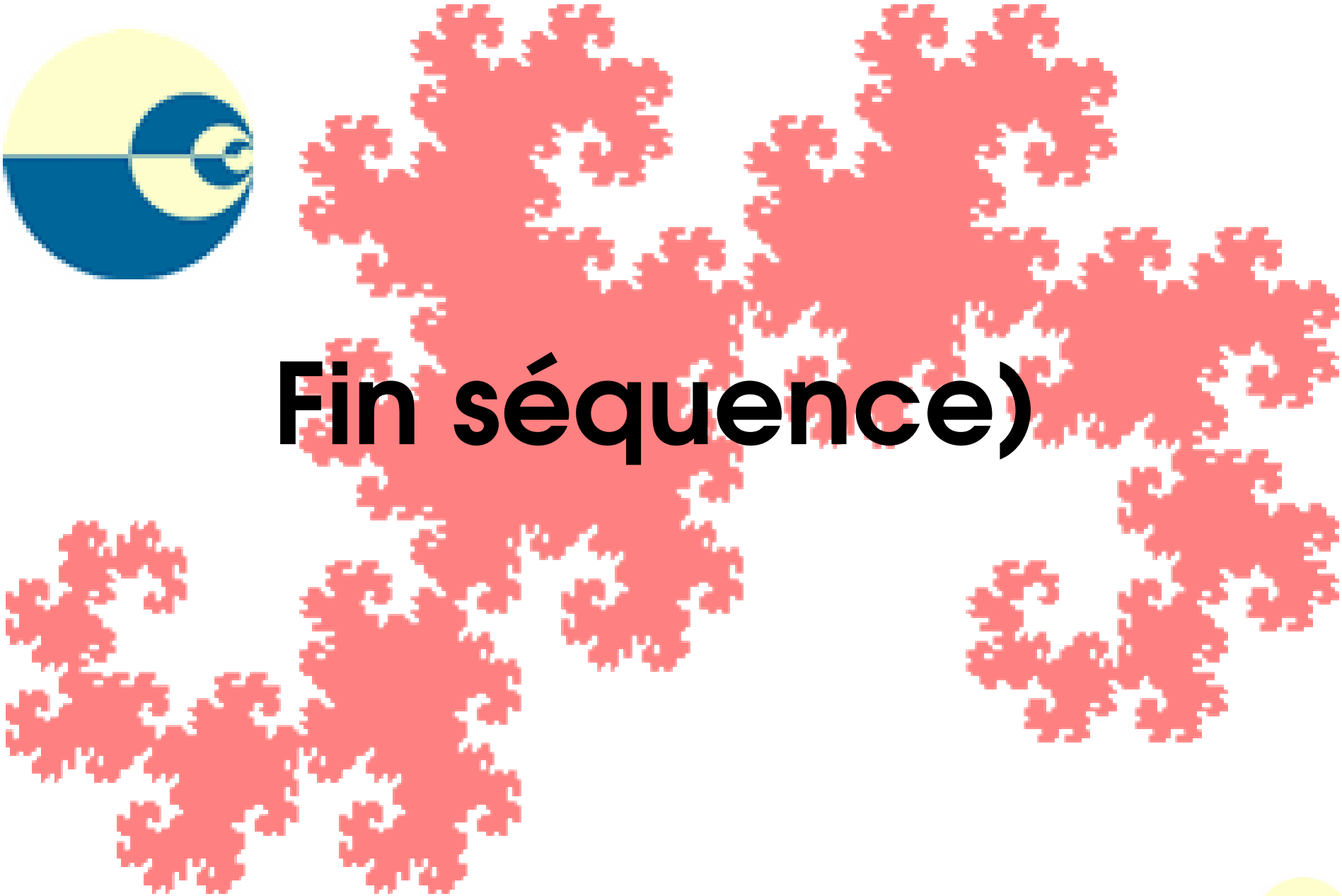
Applications `reduce`

```
(somme (list 1 2 3 4 5)) ≡
  (reduce + 0 (list 1 2 3 4 5))   → 15
  (+ 1 (+ 2 (+ 3 (+ 4 (+ 5 0)))))

(factorielle 5)  ≡
  (reduce * 1 (list 1 2 3 4 5))   → 120
  (* 1 (* 2 (* 3 (* 4 (* 5 1)))))

(somme-carres (list 1 2 3 4 5)) ≡
  (reduce + 0 (map carre (list 1 2 3 4 5))) → 55
  (+ (carre 1) (+ (carre 2) ... (+ (carre 5) 0) ...))
```

Associativité et `reduce`

Attention à l'ordre des opérations : $a - (b - c)$ n'est pas $(a - b) - c$!

```
(reduce - 0 (list 30 20 10 5))      →15

(reduce - 0 (list 30 20 10 5))      ≡
  (- 30 (- 20 (- 10 (- 5 0))))
```



Fin séquence)

(Séquence 5.4

Schéma général de récursion

méta-schéma de récursion

Le schéma de récursion usuel sur les listes est :

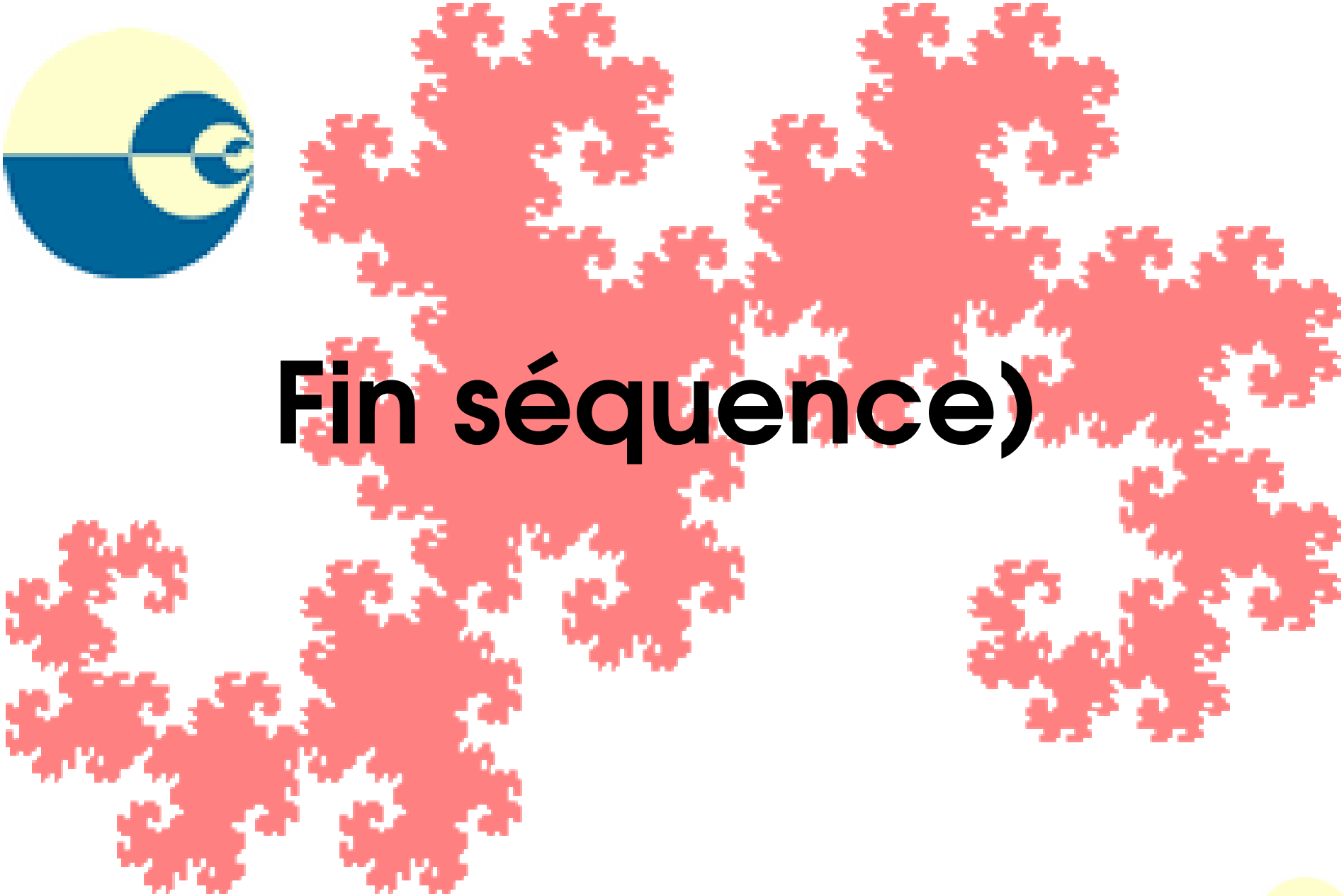
```
;;; fRec: LISTE[alpha] -> ...
(define (fRec L)
  (if (pair? L)
      (combinaison
        (car L)
        (fRec (cdr L)) )
      cas-liste-vide ) )
```

On peut l'abstraire en

```
(define (schema combinaison cas-liste-vide)
  (define (frec L)
    (if (pair? L)
        (combinaison (car L) (frec (cdr L)))
        cas-liste-vide ) )
  frec )
```

Applications de `schema`

```
(define (c0 f)
  (define (c01 e r)
     (cons (f e) r) )
  c01 )
(map f L) ≡ ((schema (c0 f) (list)) L)

(define (c1 pred)
  (define (c11 e r)
    (if (pred e) (cons e r) r) ) )
(filtre pred L) ≡ ((schema (c1 pred) (list)) L)

(somme L) ≡ ((schema + 0) L)

(define (c2 e r) (+ 1 r))
(longueur L) ≡ ((schema c2 0) L)
```



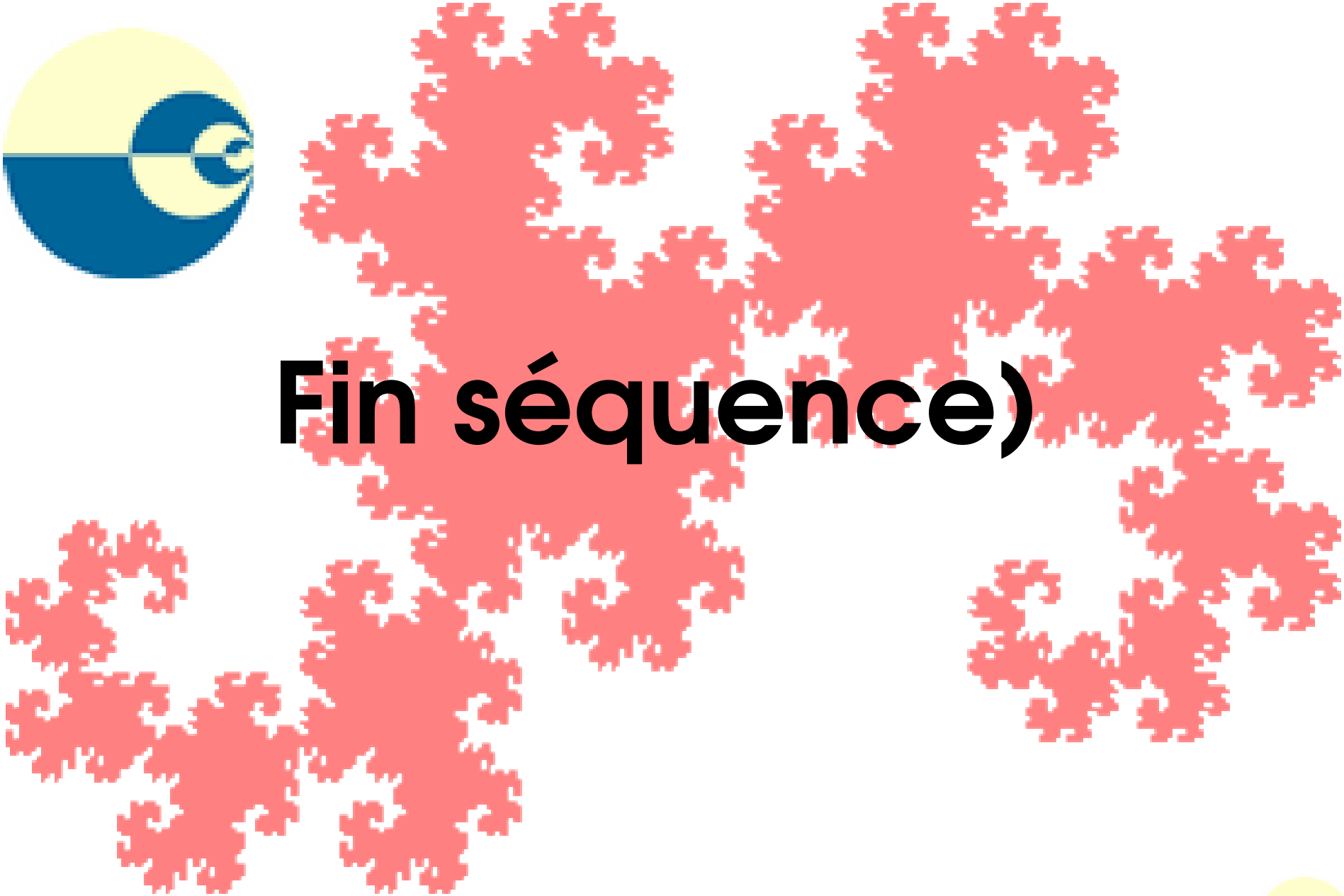
Fin séquence)

(Séquence 5.5

Résumé

Résumé

- Vous voyez maintenant comment sont factorisés des schémas de calcul récurrents dans des fonctionnelles.
- Plus besoin d'écrire certaines fonctions récursives, utilisez les fonctionnelles adéquates.



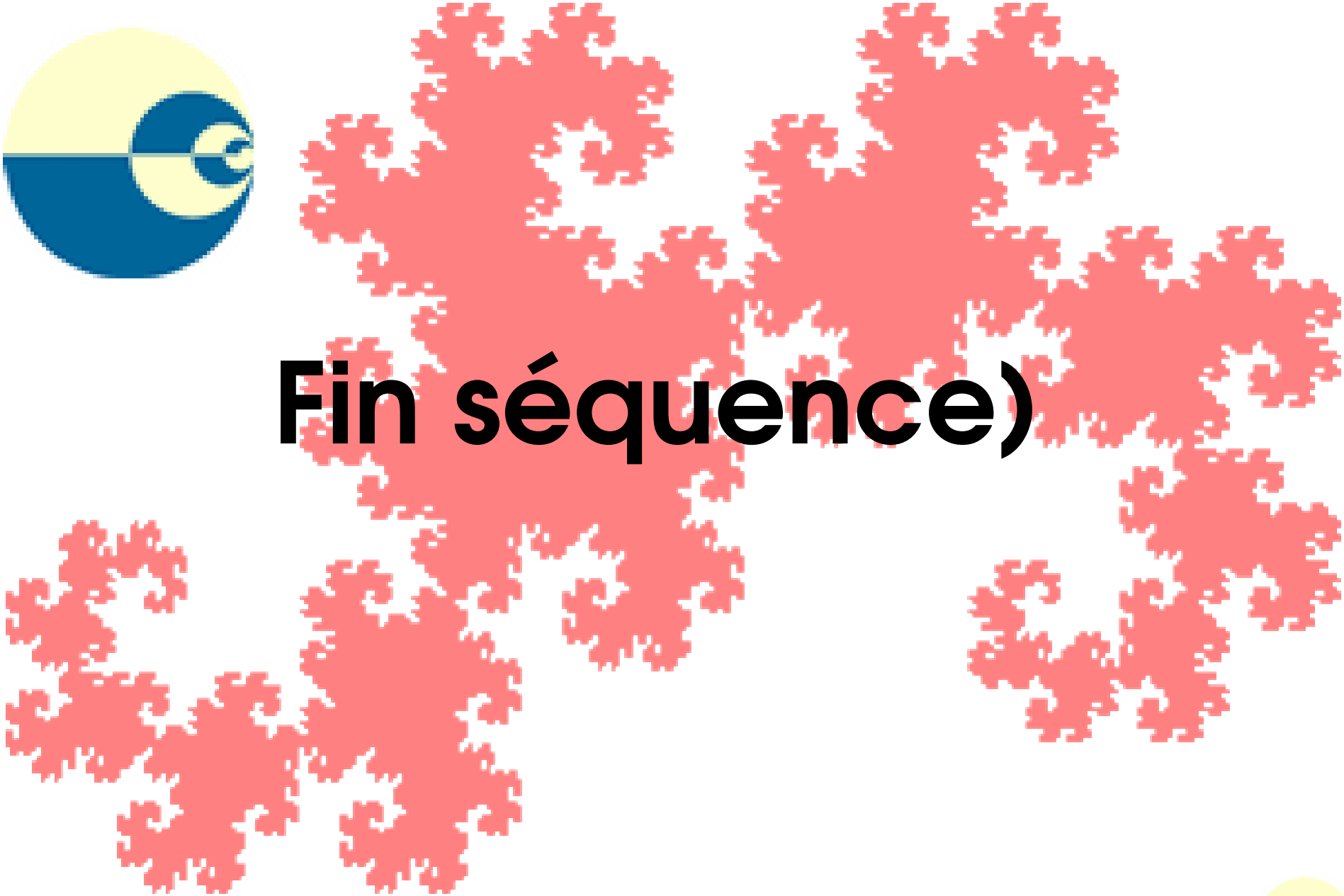
Fin séquence)

(Séquence 6.0

= Plan semaine 6

Plan semaine 6

- La citation
- La notion de liste d'associations
- La notion de barrière d'abstraction
- La notion de n-uplet



Fin séquence)

(Séquence 6.1

Citation

Listes

En Scheme, les programmes sont représentés par des ***S-expressions*** (ou expressions symboliques).
Les parenthèses `(` ... `)` servent aussi à noter en Scheme :

- des applications fonctionnelles,
- des formes spéciales (`define`, `if`, `and`, `or`, `let`, ...).

Comment faire apparaître une liste dans un programme ?

Expression et valeur d'une expression

Faire la différence entre :

- **expression** : `(list 1 2 3)`
- **valeur** : `(1 2 3)` la liste formée des éléments 1, 2 et 3

Notations : `(list 1 2 3)` $\equiv$ `(cons 1 (list 2 3))`
$\rightarrow$ `(1 2 3)`

Comment exprimer (*citer*) la **valeur** de l'**expression** `(list 1 2 3)` ?

Syntaxe de la citation

- Syntaxe de la citation

  (**quote** *exp*) **ou** '*exp*

- La citation est une **forme spéciale** : on n'évalue pas le paramètre mais on le retourne tel quel. La citation d'une expression signifie « J'ai pour valeur ce qui suit »

La citation

- la citation d'une constante est la constante elle-même
  `(quote 2)` ≡ `'2` ≡ `2` → `2`
- la citation d'un symbole : `(quote a)` ≡ `'a` → `a`
- la citation d'une liste est la liste des citations de ses éléments

```
(quote (e1 e2 e3)) ≡ '(e1 e2 e3)
                   ≡ (list 'e1 'e2 'e3)
```

```
(quote ()) ≡ '() ≡ (list)
```

Exemples :

```
(cons 'a '()) → (a)
(quote (a1 a2)) ≡ '(a1 a2) → (a1 a2)
(quote (1 2 3)) ≡ '(1 2 3) → (1 2 3)
```

Distinguer variable et symbole

Dans vos programmes Scheme, vous avez utilisé des ***symboles***

- comme mots clef : `if`, `and`, `or`, `let`, `define`
- comme identificateurs : `*`, `map`, `n`, ...

On peut manipuler les symboles pour eux-mêmes.

Remarques I

```
(let ((x 42))
  ...
  (display (list 'x '= x))    ; Imprime (x = 42)
  ... )
```

- NOTA1 : La « fonction » `display` rend, bien sûr, un résultat mais la norme de Scheme (IEEE 1178-1990) prescrit que ce résultat n'est pas spécifié. En conséquence, il ne faut jamais utiliser ce résultat puisque l'on ignore ce qu'il peut bien être.

Remarques II

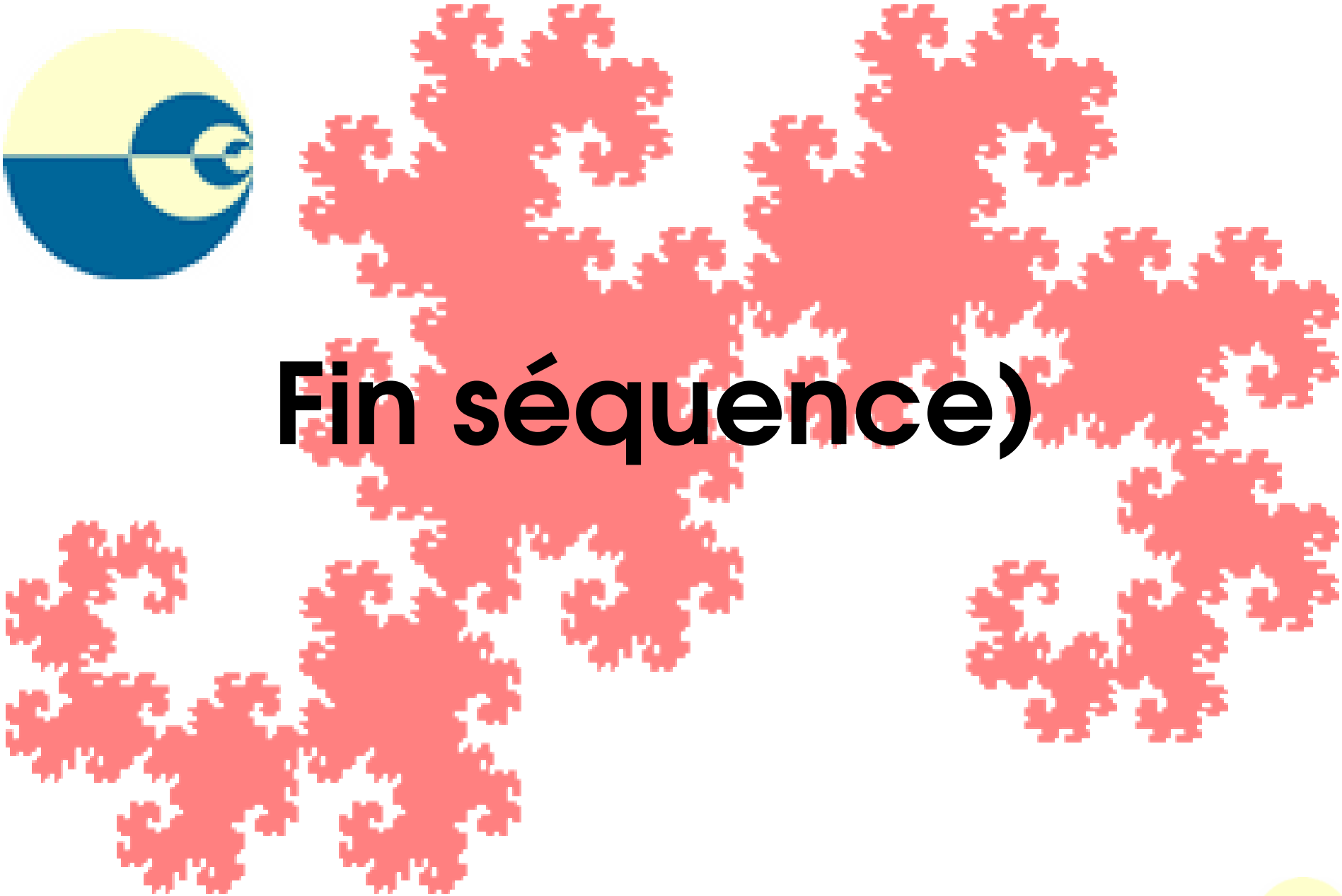
- NOTA2 : La « fonction » `display` imprime à l'écran son argument : c'est, en jargon, un « effet de bord » (traduction littérale de *side-effect* qui est en fait un effet secondaire). Si l'on veut imprimer une valeur (pour mettre au point son programme) et retourner cette valeur, il faut utiliser une séquence dont le mot-clé est `begin` et ainsi écrire :

```
(begin
  (display (list 'x '= x)) ; imprimer la valeur
  x )                      ; retourner la valeur
```

Remarques III

- NOTA3 : Fort heureusement, on a rarement besoin de ce mot-clé car il est implicite dans le corps des fonctions et le corps des blocs locaux. Ainsi peut-on écrire directement :

```
(define (foo x)
  (display (list 'x '= x))
  x )
```



Fin séquence)

(Séquence 6.2

Liste d'associations

Les listes d'associations

- Association
- Liste d'associations
- (constructeur) Ajout d'une association
- (sélecteur) Recherche d'une association
- (sélecteur) Recherche d'une valeur

Une association

Définition : étant donnés deux types ***Clef*** et ***Valeur***, une ***association*** est un élément de type `COUPLE[Clef Valeur]`, représentée par une liste de 2 éléments.

```
; association de type COUPLE[string string]
        ("chat" "cat")
; associations de type COUPLE[Nat Nat]
        (2003512    192)
        (2003513    567)
; associations de type COUPLE[Nat string]
        (192   "Virginie")
        (567   "Paul")
```

Autre exemple d'association

Seulement pour ceux qui désirent approfondir la notion de citation.

```
; associations de type COUPLE[symbole fonction]
          (list  '+    +)
          (list  '*    *)
          (list  '^    puissance)
```

`(list '+ +)` $\rightarrow$ `(+ #<primitive:+>)`

`'('+ +)` $\rightarrow$ `('+ +)` *ou* `((`**quote** `+) +)`

Une liste d'association

Définition : ***une liste d'associations*** est une liste dont chaque terme est une **association (clef valeur)**

```
; LISTE[COUPLE[string string]]
(list (list "chat"  "cat")
      (list "chien" "dog"))
   ≡
(list '("chat"  "cat")
      '("chien" "dog"))
  ≡
'(("chat"  "cat")
  ("chien" "dog"))
            →  (("chat" "cat")("chien" "dog"))
```

Ajout dans une liste d'associations

Le constructeur de la liste d'associations vide est `(list)`
Ce constructeur ajoute une association en tête d'une ***Aliste***

```
;;; ajout : alpha * beta * LISTE[COUPLE[alpha beta]]
;;;                 -> LISTE[COUPLE[alpha beta]]
;;; (ajout cle valeur a-liste) rend la liste d'asso-
;;; -ciation obtenue en ajoutant l'association
;;; (cle valeur) en tete de a-liste.
(define (ajout cle valeur a-liste)
  (cons (list cle valeur)
        a-liste) )
```

```
(let ((mon-dico '(("souris"  "mouse")
                  ("chat"    "cat")
                  ("chien"   "dog")
                  ("fromage" "cheese"))))
  (ajout "chat" "tabby" mon-dico))
```

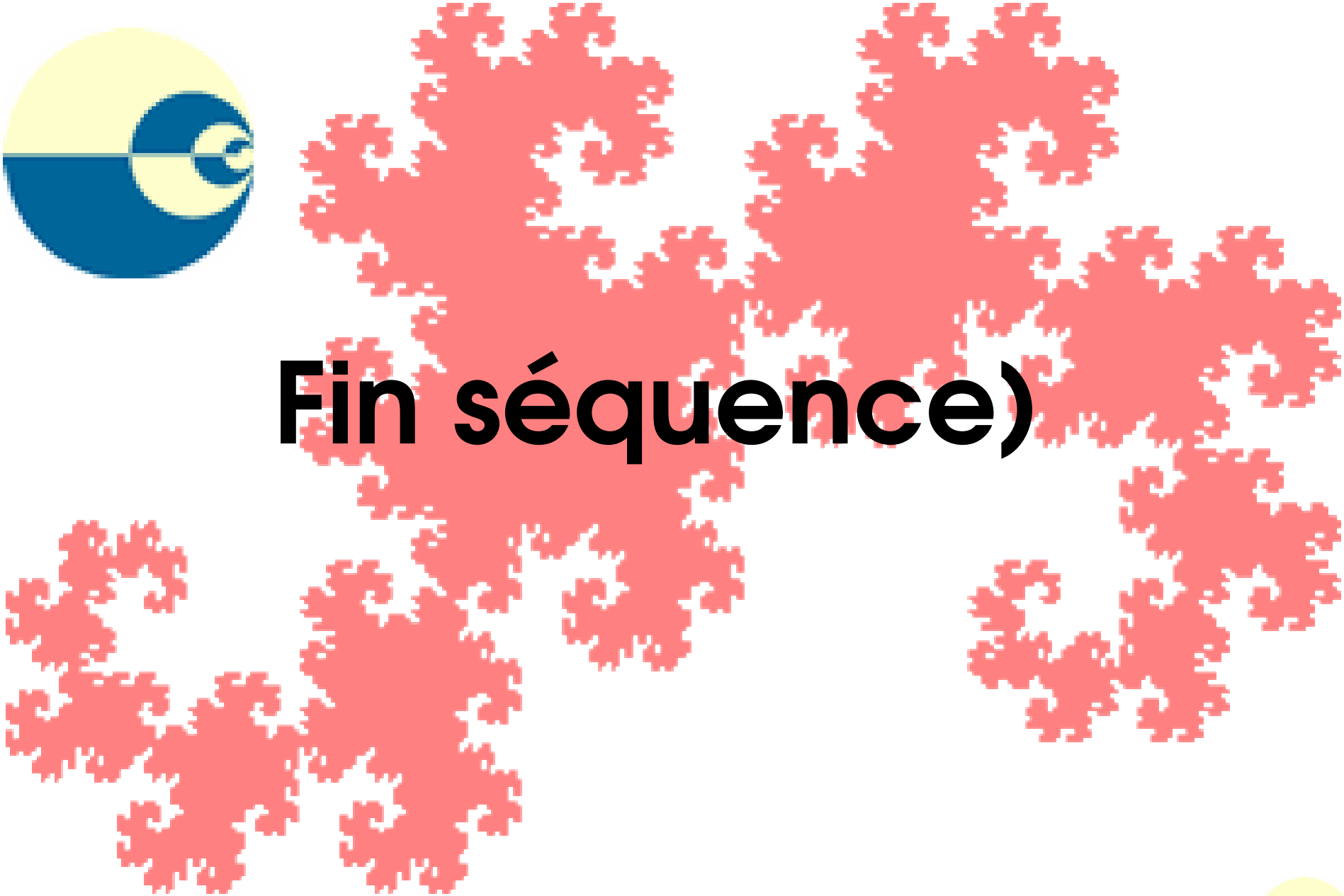
Recherche

(cf. carte de référence)

```
;;; assoc:
;;;   alpha * LISTE[COUPLE[a b]] -> COUPLE[a b] + #f
;;; (assoc cle aliste) rend la 1ère association de
;;; aliste dont le 1er élément est égal à cle. Rend
;;; la valeur #f en cas d'échec.
```

- Rend la première association de la liste (raison d'efficacité) et donc la plus récente (et la plus à gauche)
- C'est un ***semi-prédicat***, elle rend :
  - `#f` lorsque l'association n'existe pas,
  - et sinon la valeur `Vrai` sous la forme de l'association elle-même.

Rappelons qu'en Scheme tout ce qui n'est pas `#f` est Vrai. Une association vaut donc toujours vrai.



Fin séquence)

(Séquence 6.3

Usage de Aliste

Abbréviations

```
;;; cadr: LISTE[alpha] -> alpha
;;; ERREUR lorsque la liste donnée a moins de 2 éléments
;;; (cadr L) rend le deuxième élément de la liste donnée

;;; cddr: LISTE[alpha] -> LISTE[alpha]
;;; ERREUR lorsque la liste donnée a moins de deux éléme
;;; (cddr L) rend la liste L sans ses 2 premiers élément
```

```
(cadr L) ≡ (car (cdr L))

(cddr L) ≡ (cdr (cdr L))

(cadr '(3 5 2 5))          →          5

(cddr '(3 5 2 5))}         →          (2  5)

(cadddr '(2 3 5 6 8))      →          6

(cddddr '(2 3 5 6 8))      →          (8)
```

```
(define (mon-dico)
  '(("chat"    "tabby")
    ("souris"  "mouse")
    ("chat"    "cat")
    ("chien"   "dog")
    ("fromage" "cheese")))

(define (traduire dico)
  (define (wordfor mot)
    (let ((a (assoc mot dico)))
      (if a (cadr a) mot) ) )
  wordfor )

(map (traduire mon-dico)
     '(le chien poursuit le chat) )
   →  (le dog poursuit le tabby)
```

Définition de `assoc`

`assoc` est une fonction prédéfinie en Scheme.
Elle pourrait se définir comme :

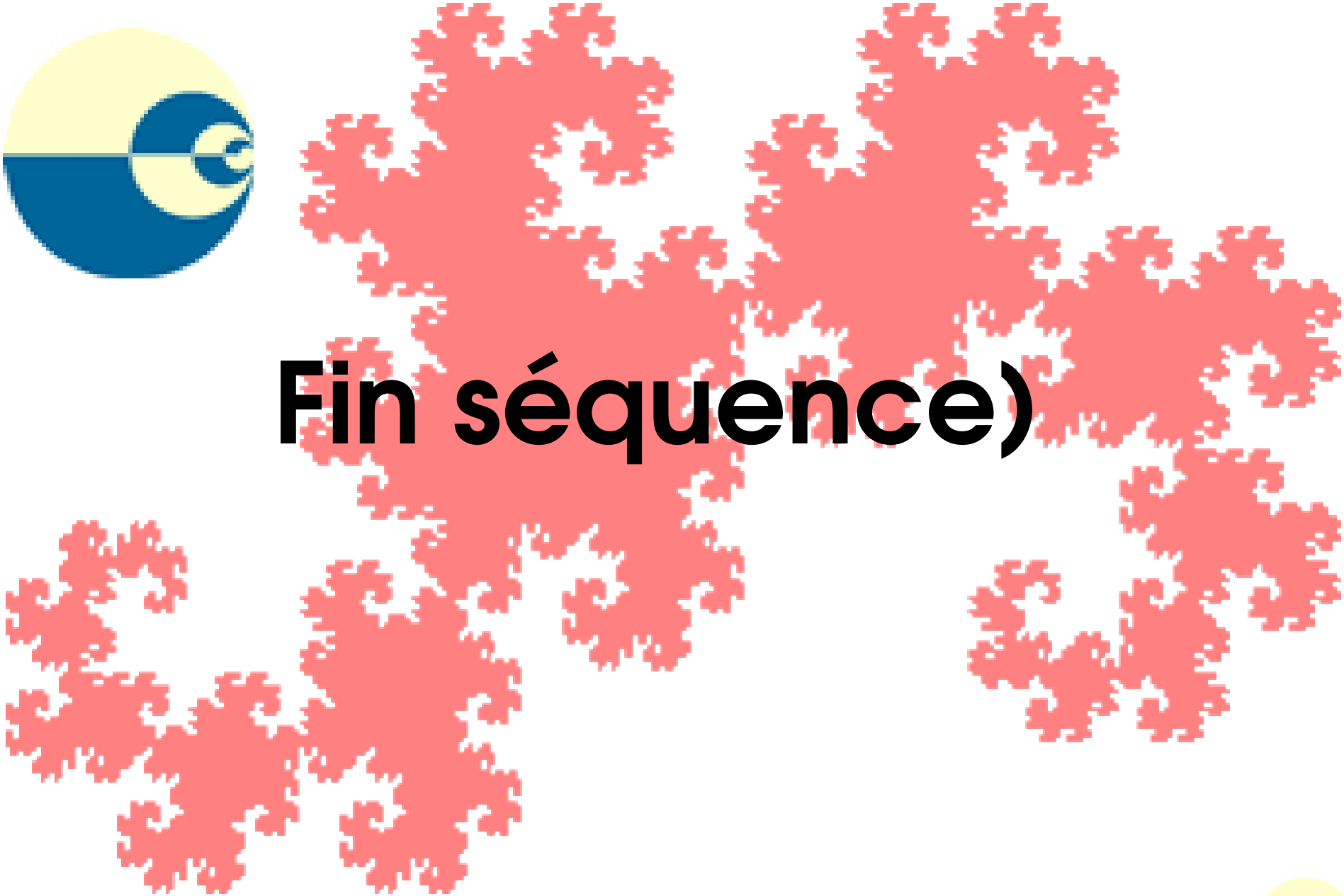
```
(define (assoc cle a-liste)
  (if (pair? a-liste)
      (if (equal? cle (caar a-liste))
          (car a-liste)
          (assoc cle (cdr a-liste)))
      #f))
```

Fonction `valeur-de`

La fonction `valeur-de` donne la valeur associée à une clé.

```
;;; valeur-de: alpha * LISTE[COUPLE[alpha beta]]
;;;                   -> beta + #f
;;; (valeur-de clef aliste) rend la valeur de la
;;; 1ère association de aliste dont le 1er élément
;;; est égal à clef. Retourne Faux en cas d'échec.
(define (valeur-de cle a-liste)
  (let ((couple (assoc cle a-liste)))
    (if couple
        (cadr couple)
        #f) ) )
```

Rappel : toute valeur différente de `#f` est équivalente à `Vrai`.



Fin séquence)

(Séquence 6.4

Barrière d'abstraction

Barrière d'abstraction

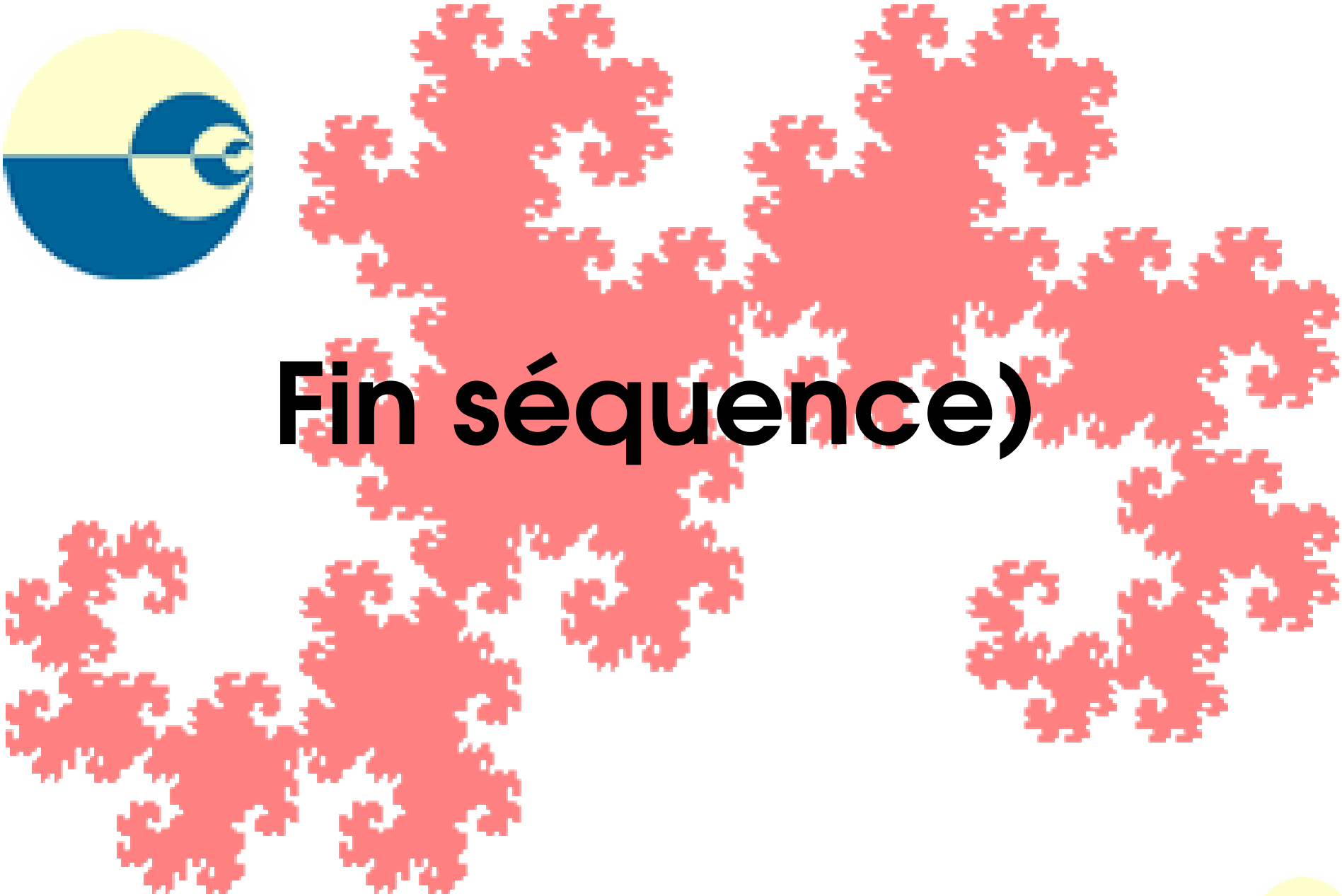
Une ***barrière d'abstraction*** est un ensemble de fonctions qui permet de manipuler des concepts sans se soucier de l'implantation de ces concepts.
Peu importe, pour les **manipuler**, de connaître leur **implantation** !

Notion de NUPLET

Un *n-**uplet*** est une **structure de données** qui comporte un nombre fixé d'éléments, de types a priori différents.

Le **type** `NUPLET[alpha beta gamma delta]` représente une structure de 4 éléments, de types `alpha`, `beta`, `gamma` et `delta`.

et, bien sûr,
`COUPLE[alpha beta]` = `NUPLET[alpha beta]`



Fin séquence)

(Séquence 6.5

Arbres

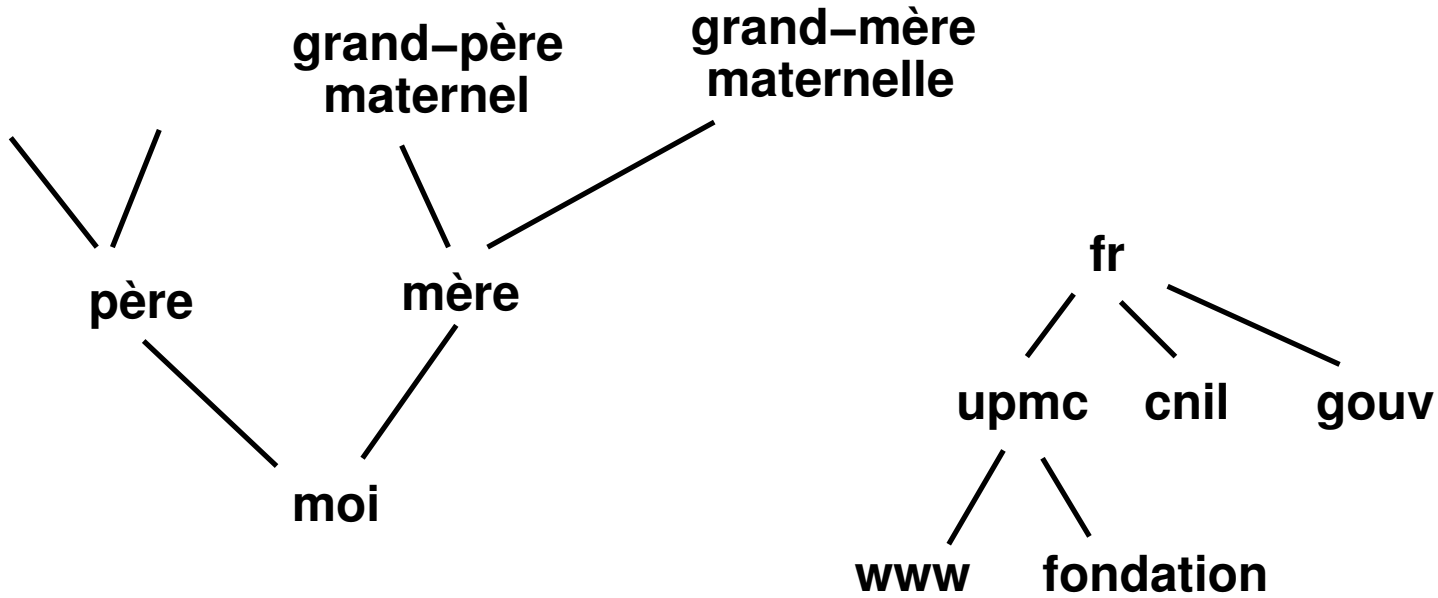
Arbres

- Exemples d'arbres
- Définition d'un arbre binaire
- Barrière d'abstraction des arbres binaires
- Schéma de récursion des arbres binaires
- Exemples : nombre de nœuds, profondeur, liste préfixe
- Affichage sous forme d'un paragraphe

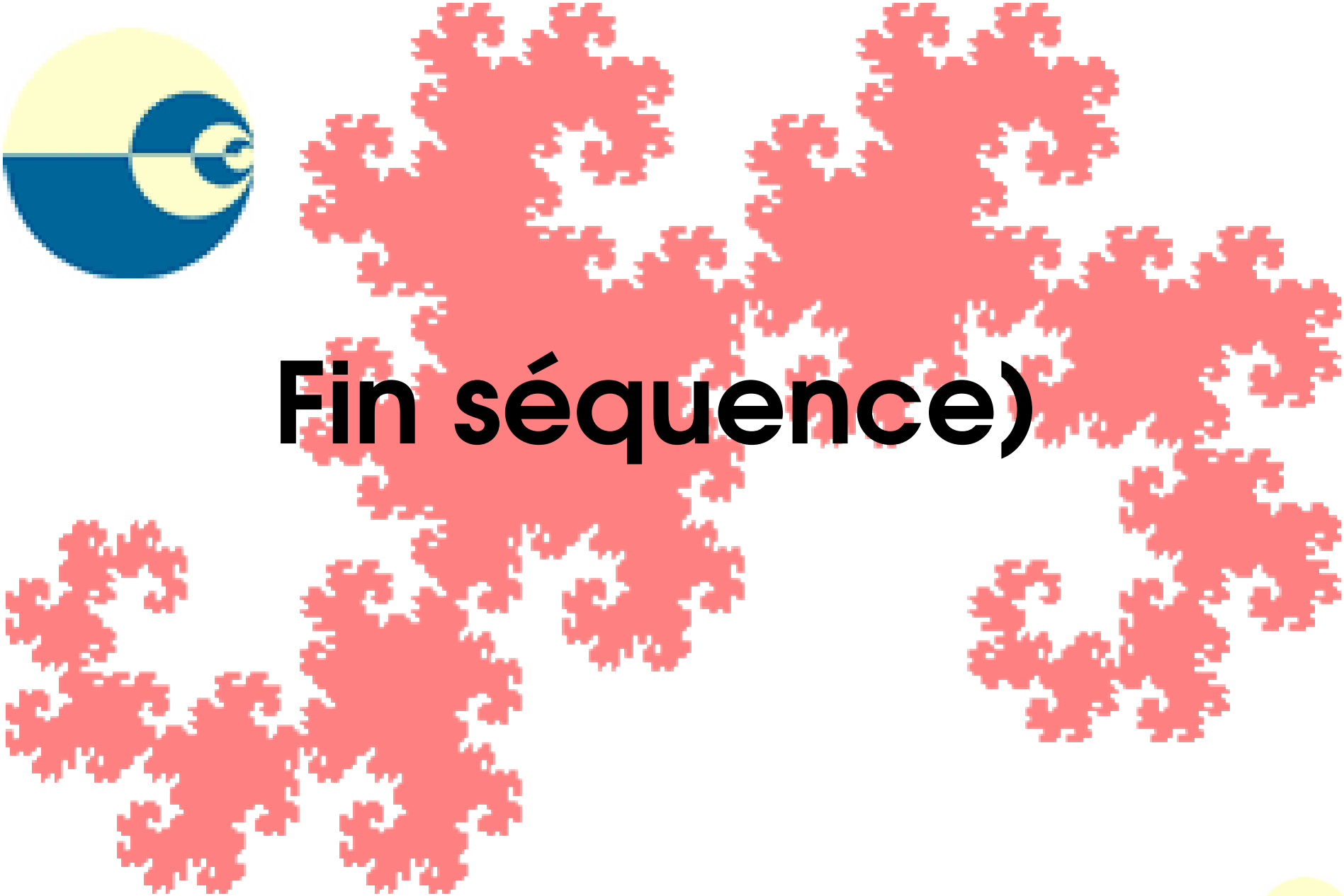
Des arbres

- Arbres binaires
  - Arbre généalogique des ascendants
  - Arbre représentant une expression arithmétique simple
- Arbres généraux
  - Arbre généalogique de la descendance
  - Table des matières d'un document
  - Arbre des fichiers
  - Noms d'ordinateur : *machine.domaine.pays*
  - Organigramme d'une société

Des arbres



Vocabulaire : nœud, étiquette, père, ascendant immédiat, fils, fille, descendant immédiat, branche, feuille, sous-arbre ...



Fin séquence)

(Séquence 6.6

Arbres binaires

Définition d'un arbre binaire

Un ***arbre binaire*** est une **structure de données** qui permet de représenter des éléments de même type, ordonnés hiérarchiquement.
Le type d'un tel arbre se note : **ArbreBinaire(α)**
Dans un arbre binaire non vide, chaque nœud porte une information (étiquette de type α), et a exactement 2 descendants immédiats.

Définition récursive d'un arbre binaire

Définition récursive : Un arbre binaire de type **ArbreBinaire(α)** est

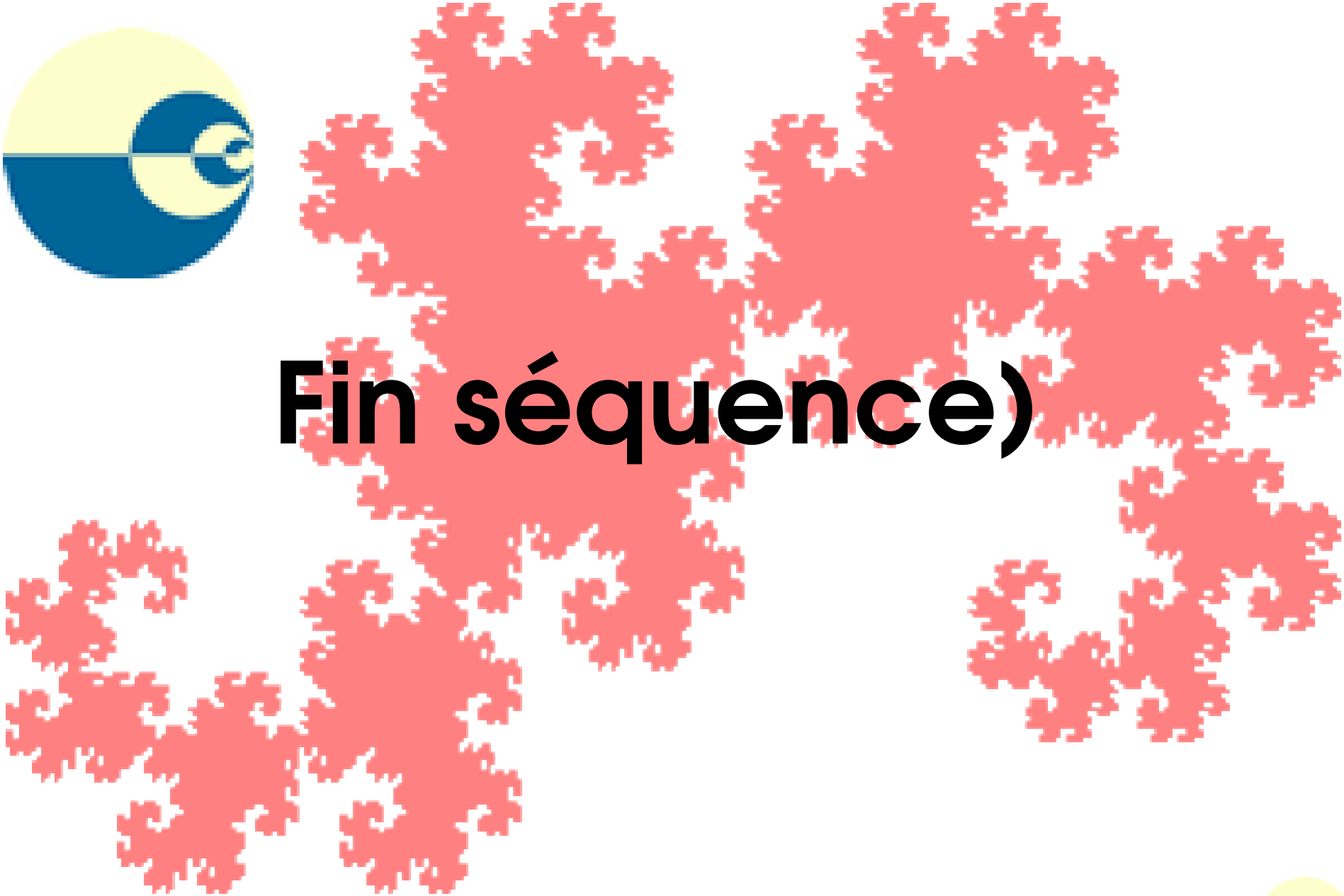
- soit vide
- soit formé
  - d'un nœud (portant une étiquette de type α)
  - d'un sous-arbre gauche de type **ArbreBinaire(α)**
  - d'un sous-arbre droit de type **ArbreBinaire(α)**

Barrière d'abstraction

La barrière d'abstraction des arbres binaires doit contenir :

- **Constructeurs** pour construire un arbre binaire : `ab-vide`, `ab-noeud`
- **Accesseurs** pour accéder aux parties d'un arbre binaire : `ab-etiquette`, `ab-gauche` et `ab-droit`
- **Reconnaisseur** pour déterminer si un arbre binaire est non vide `ab-noeud?`

1. On manipule les arbres binaires uniquement à travers leur **barrière d'abstraction**.
2. Barrière d'abstraction => on ne connait que la **spécification** des fonctions. (Plus tard implantation)



Fin séquence)

(Séquence 6.7

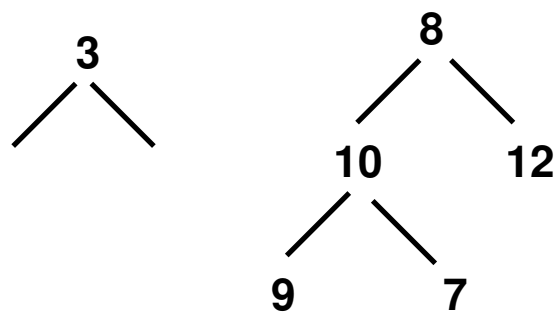
Spécifications des arbres binaires

Spécification des constructeurs

```
;;; ab-vide:  -> ArbreBinaire[α]
;;; (ab-vide) rend l'arbre binaire vide.

;;; ab-noeud: α
;;;          * ArbreBinaire[α]
;;;          * ArbreBinaire[α]
;;;      -> ArbreBinaire[α]
;;; (ab-noeud e B1 B2) rend l'arbre binaire formé
;;; d'une racine d'étiquette e, d'un sous-arbre
;;; gauche B1 et d'un sous-arbre droit B2.
```

Exemples : arbres binaires de nombres

```
(ab-noeud 3 (ab-vide) (ab-vide))

(let* ((b0 (ab-vide))
       (b9 (ab-noeud 9 b0 b0))
       (b7 (ab-noeud 7 b0 b0))
       (b10 (ab-noeud 10 b9 b7))
       (b12 (ab-noeud 12 b0 b0)) )
   (ab-noeud 8 b10 b12) )
```



Un arbre est un objet opaque qui ne se manipule que via la barrière d'abstraction.

Spécification des accesseurs

```
;;; ab-etiquette: ArbreBinaire[α] -> α
;;; (ab-etiquette B) rend l'étiquette de la racine
;;; de l'arbre B
;;; ERREUR lorsque B ne satisfait pas ab-noeud?

;;; ab-gauche: ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-gauche B) rend le sous-arbre gauche de B
;;; ERREUR lorsque B ne satisfait pas ab-noeud?

;;; ab-droit: ArbreBinaire[α] -> ArbreBinaire[α]
;;; (ab-droit B) rend le sous-arbre droit de B
;;; ERREUR lorsque B ne satisfait pas ab-noeud?
```

Propriétés algébriques

- Pour tout couple d'arbres binaires G et D, et toute valeur v

```
(ab-etiquette (ab-noeud v G D))      → v
(ab-gauche    (ab-noeud v G D))      → G
(ab-droit     (ab-noeud v G D))      → D
```

- Pour tout arbre binaire non vide B

```
(ab-noeud (ab-etiquette B)
          (ab-gauche B)
          (ab-droit B))      → B
```

Spécification du reconnaisseur

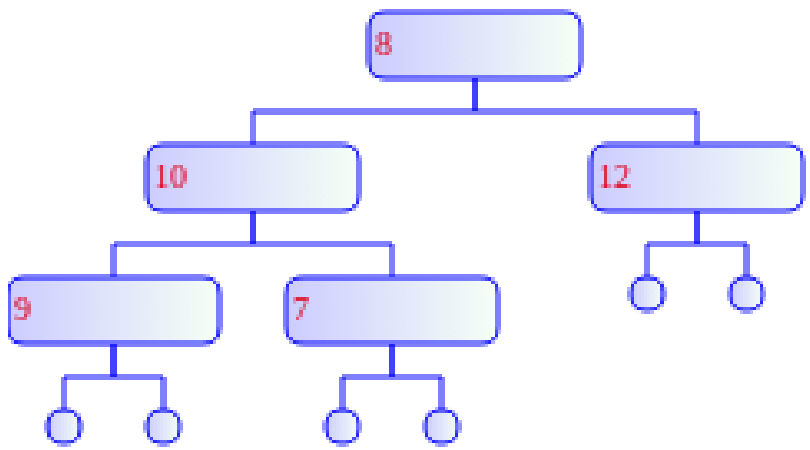
```
;;; ab-noeud?: ArbreBinaire[α] -> bool
;;; (ab-noeud? B) rend vrai ssi B n'est pas
;;; l'arbre vide.
```

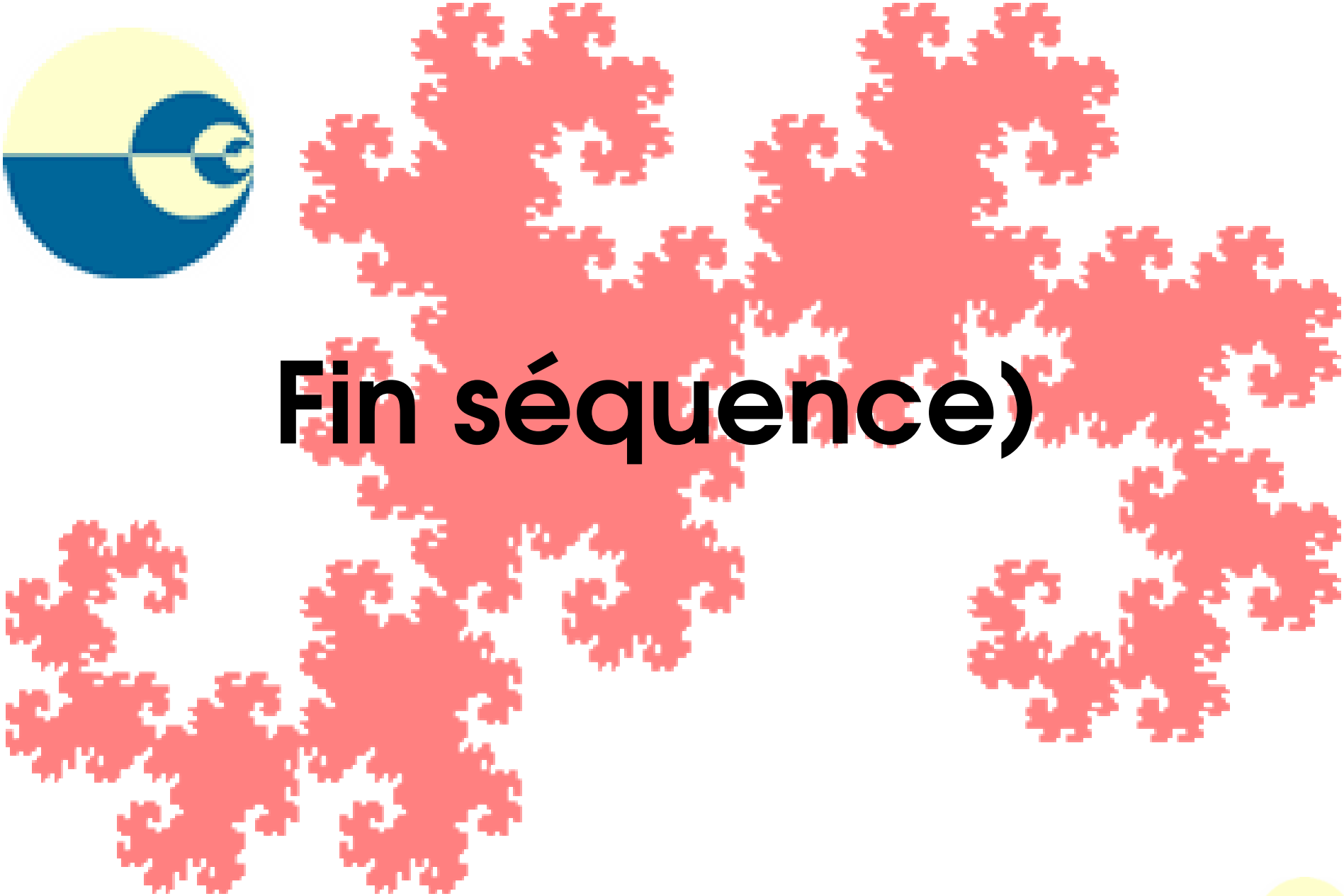
Propriété : Pour tout arbre binaire B,
`(ab-vide? B)` = `(not (ab-noeud? B))`

Affichage d'arbre binaire

La barrière d'abstraction contient aussi une fonction d'affichage, qui permet de « visualiser » les objets arbres binaires par des S-expressions. En MrScheme :

```
(ab-affiche (let* ((b0 (ab-vide))
                   (b9 (ab-noeud 9 b0 b0))
                   (b7 (ab-noeud 7 b0 b0))
                   (b10 (ab-noeud 10 b9 b7))
                   (b12 (ab-noeud 12 b0 b0)) )
              (ab-noeud 8 b10 b12) ))
```





Fin séquence)

(Séquence 6.8

Récursion sur arbres binaires

Récursion sur les arbres binaires

Un arbre binaire est :

- soit vide
- soit constitué d'une étiquette, d'un sous-arbre gauche et d'un sous-arbre droit

```
;;; fRec: ArbreBinaire[alpha] -> ...
(define (fRec B)
  (if (ab-noeud? B)
      (combinaison (ab-etiquette B)
                   (fRec (ab-gauche B))
                   (fRec (ab-droit B)))
      cas-arbre-vide ) )
```

Nombre de noeuds

```
;;; nombre-noeuds: ArbreBinaire[α] -> nat
;;; (nombre-noeuds B) rend le nombre de noeuds de B
(define (nombre-noeuds B)
  (if (ab-noeud? B)
      (+ 1
         (nombre-noeuds (ab-gauche B))
         (nombre-noeuds (ab-droit B)))
      0))
```

Somme des étiquettes

```
;;; somme-arbre:  ArbreBinaire[Nombre] -> Nombre
;;; (somme-arbre B) rend la somme des étiquettes de B
(define (somme-arbre B)
  (if (ab-noeud? B)
      (+ (ab-etiquette B)
         (somme-arbre (ab-gauche B))
         (somme-arbre (ab-droit B)))
      0))
```

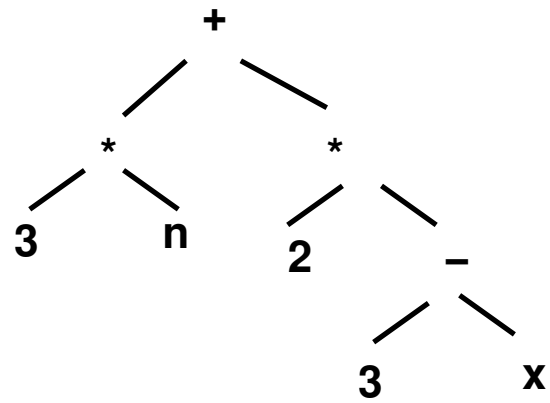
Profondeur

Définition récursive de la profondeur d'un arbre :

- la profondeur de l'arbre vide est 0,
- la profondeur d'un arbre non vide est égale au maximum des profondeurs de ses sous-arbres immédiats, augmenté de 1.

```
;;; profondeur:  ArbreBinaire[α] -> nat
;;; (profondeur B) rend la profondeur de B
(define (profondeur B)
  (if (ab-noeud? B)
      (+ 1
         (max (profondeur (ab-gauche B))
              (profondeur (ab-droit B))))
      0))
```

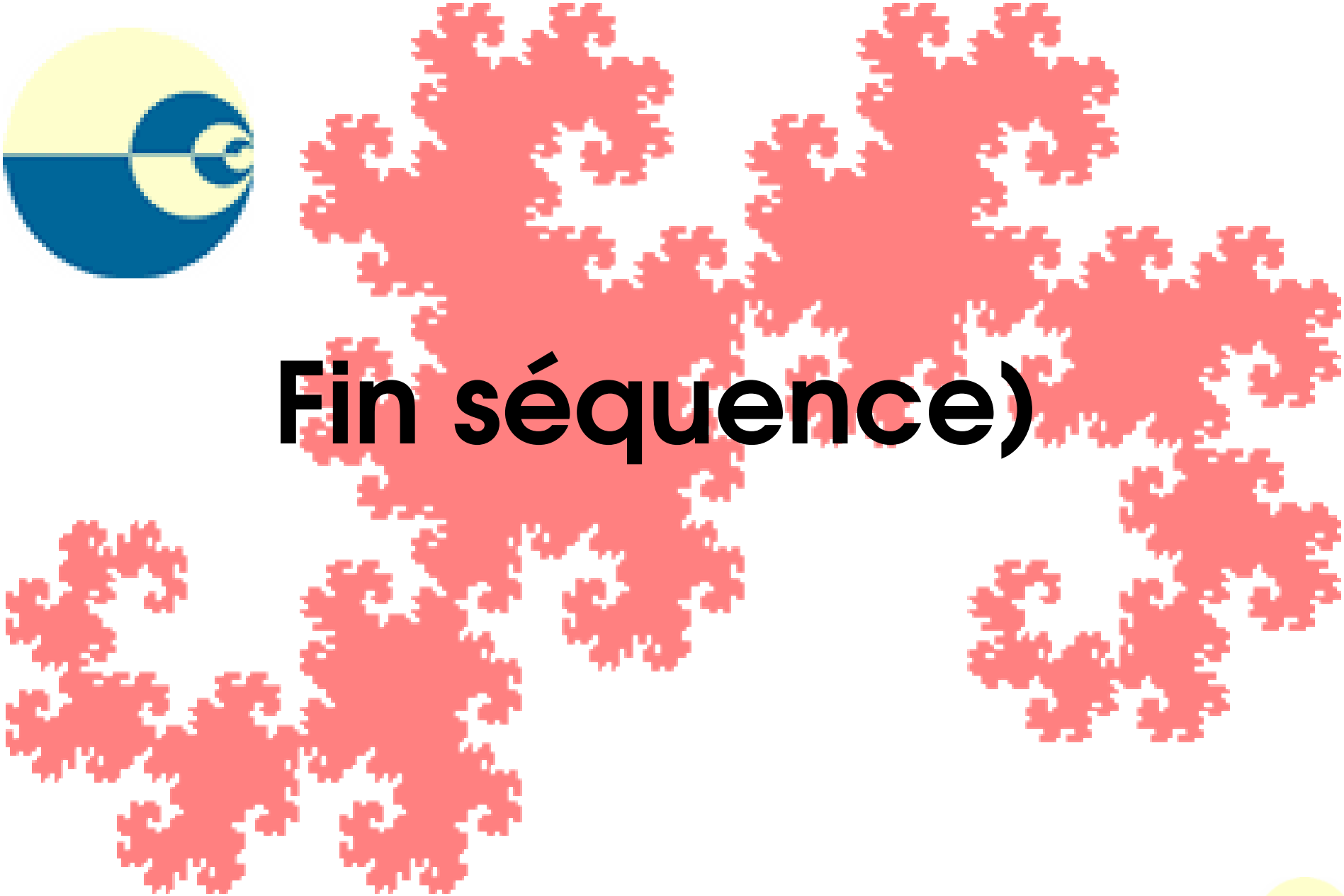
Différentes écritures d'une exp. arithm.



- infixe parenthésée : `((3 * n) + (2 * (3 - x)))`
- préfixe Scheme : `(+ (* 3 n) (* 2 (- 3 x)))`
- polonaise préfixe : `+ * 3 n * 2 - 3 x`
- liste préfixe : `(+ * 3 n * 2 - 3 x)`
- polonaise postfixe : `3 n * 2 3 x - * +`

Liste préfixe

```
;;; liste-pref:  ArbreBinaire[α] -> LISTE[α]
;;; (liste-pref B) rend la liste préfixée de B
(define (liste-pref B)
  (if (ab-noeud? B)
      (cons (ab-etiquette B)
            (append (liste-pref (ab-gauche B))
                    (liste-pref (ab-droit B))))
      (list)))
```



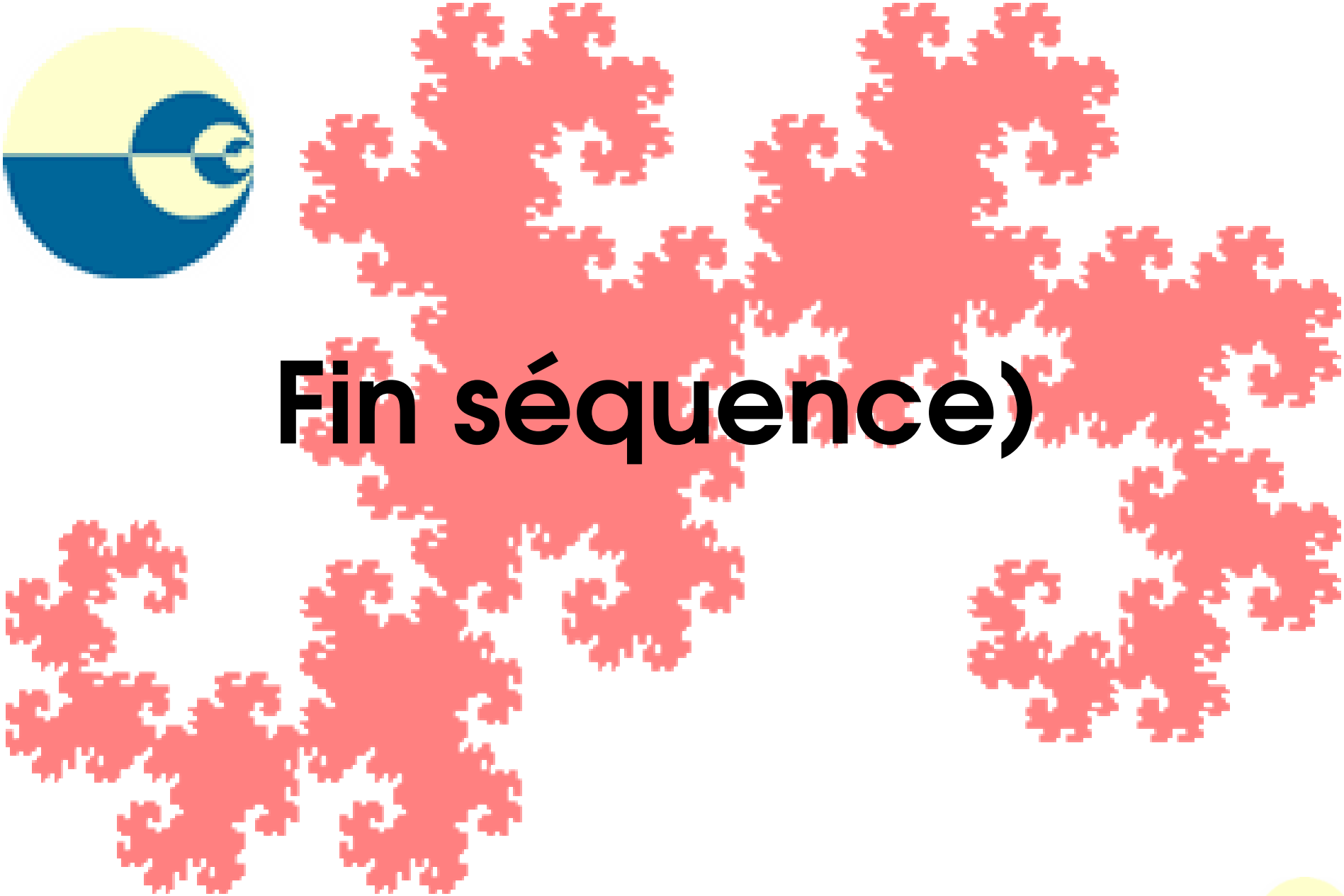
Fin séquence)

(Séquence 6.9

Résumé

Résumé

- la barrière d'abstraction est un contrat entre l'implémenteur et l'utilisateur.
- Elle doit être stable afin de permettre des développements indépendants.



Fin séquence)

(Séquence 7.0

= Plan semaine 7

Plan semaine 7

- Définition d'un arbre binaire de recherche
- Efficacité
- Recherche et ajout d'un élément
- Suppression d'un élément

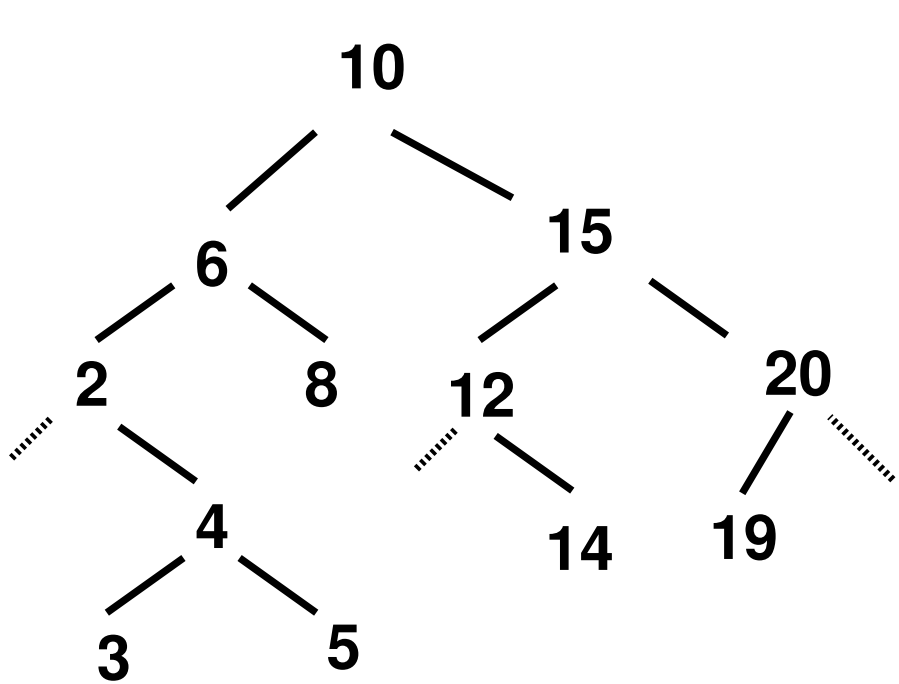
Arbre binaire de Recherche

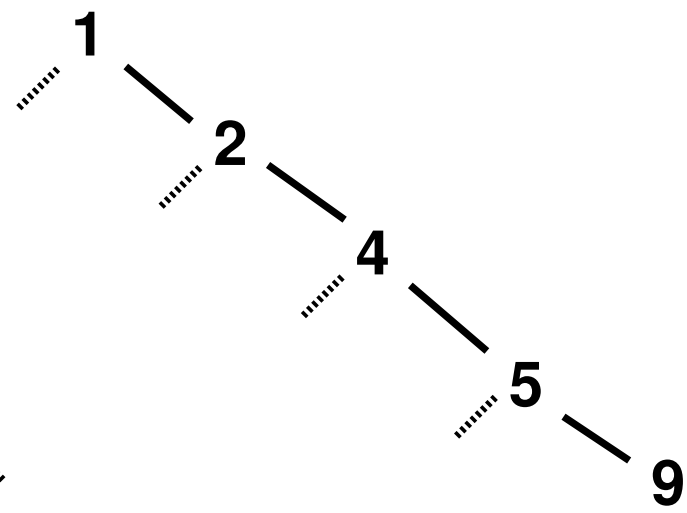
Définition : Un arbre binaire de recherche (ou ABR) est un arbre binaire tel que

- l'étiquette de la racine est
  - supérieure à toutes les étiquettes du sous-arbre gauche
  - inférieure à toutes les étiquettes du sous-arbre droit
- les sous-arbres gauche et droit de la racine sont aussi des arbres binaires de recherche.

Propriété IC : Soit A un arbre binaire, A est un arbre binaire de recherche si, et seulement si, la liste **infixe** des étiquettes de A est en ordre **croissant**.
On notera le type `ArbreBinRecherche`

Exemples

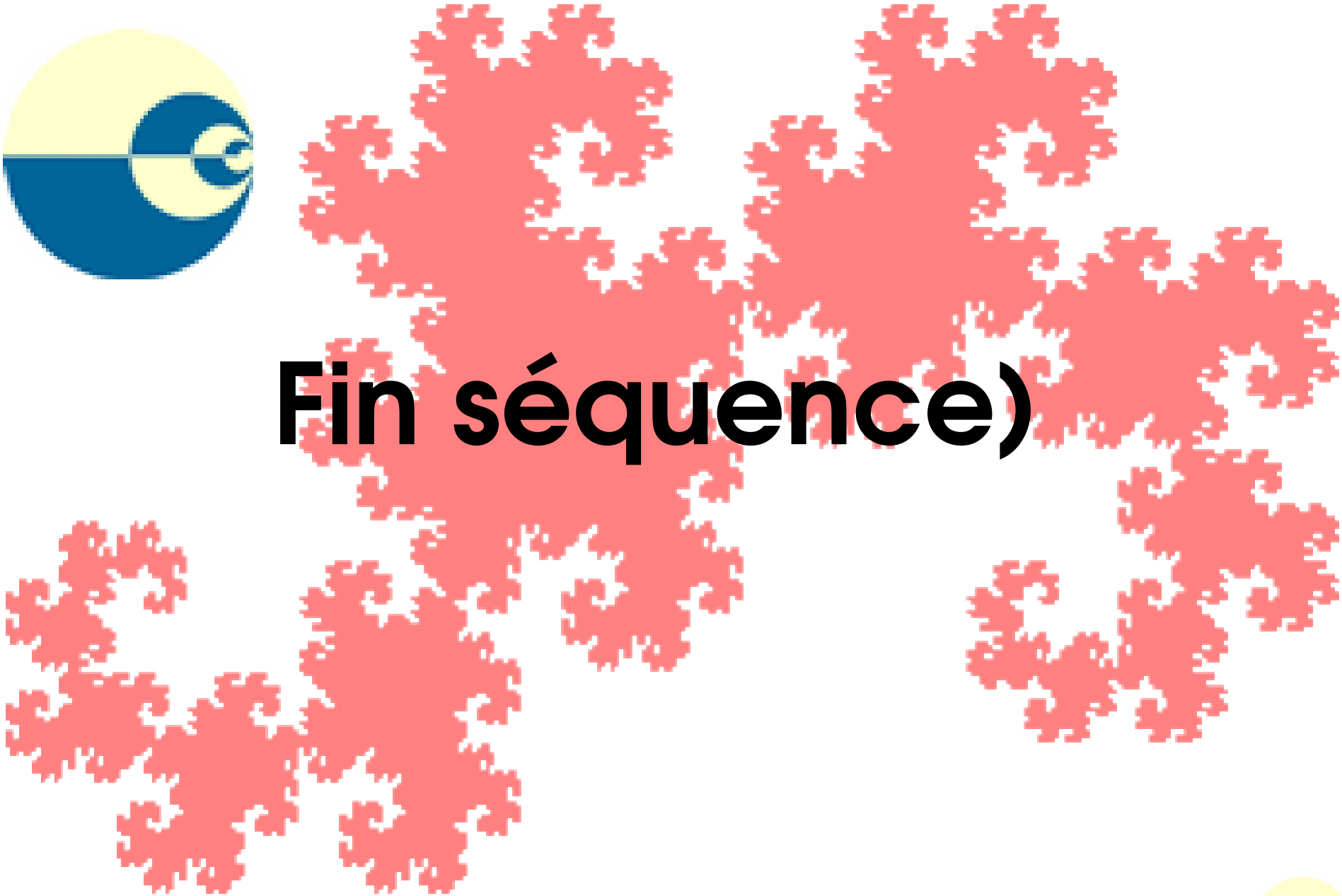




Liste infixe

```
;;; liste-infixe :  ArbreBinaire[α] -> LISTE[α]
;;; (liste-infixe B) rend la liste infixe des
;;; étiquettes de B
(define (liste-infixe B)
  (if (ab-noeud? B)
      (append (liste-infixe (ab-gauche B))
              (list (ab-etiquette B))
              (liste-infixe (ab-droit B)))
      (list)))
```

Liste infixe : 2 3 4 5 6 8 10 12 14 15 19 20



Fin séquence)

(Séquence 7.1

Barrière d'abstraction des ABR

Barrière d'abstraction des ABR

Représenter un ensemble d'éléments (ordre total), sur lequel on effectue les opérations suivantes :

- recherche d'un élément
- ajout d'un élément
- suppression d'un élément

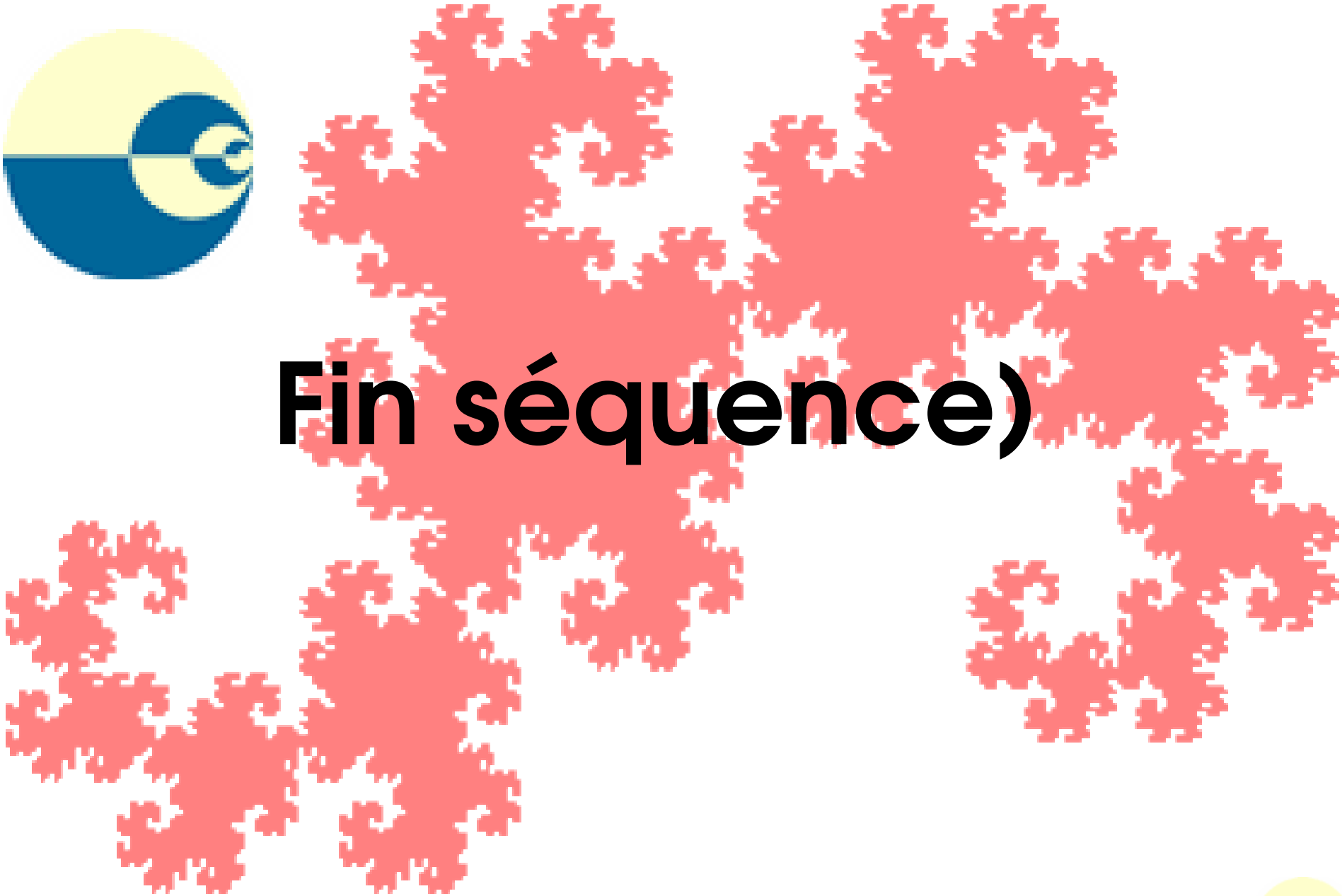
Fonctions pour ABR

On utilise préfixe `abr-` pour ces fonctions qui exploitent la barrière d'abstraction des arbres binaires.

```
ab-vide:                          -> ABR
abr-recherche: Nombre * ABR -> ABR + #f
abr-ajout:     Nombre * ABR -> ABR
abr-moins:     Nombre * ABR -> ABR
```

Quelques propriétés algébriques

```
(abr-recherche x (abr-ajout x ABR)) ≡ #t
(abr-recherche x (abr-moins x ABR)) →  #f
```



Fin séquence)

(Séquence 7.2

Récursion sur ABR

Principe des algorithmes

Une *unique comparaison* avec l'étiquette de l'arbre permet d'aiguiller le traitement

- *soit* dans le sous-arbre gauche
- *soit* dans le sous-arbre droit

Le principe d'aiguillage permet de « laisser tomber » un sous-arbre entier (gauche ou droit) et de recommencer récursivement le traitement sur *un seul* sous-arbre.

Schéma de fonction

```
;;; f-abr: Nombre * ArbreBinRecherche -> ...
(define (f-abr x ABR)
  (if (ab-noeud? ABR)
      (let ((e (ab-etiquette ABR)))
        (cond
          ((= x e) ...)
          ((< x e) (combinaison
                     (f-abr x (ab-gauche ABR))))
          (else (combinaison
                  (f-abr x (ab-droit ABR)))) ) )
      cas-arbre-vide ))
```

Efficacité

Si chaque sous-arbre contient à peu près la moitié des nœuds de l'arbre entier (arbre équilibré), alors le nombre de noeuds restant à examiner est *divisé par deux* à chaque fois : **dichotomie**

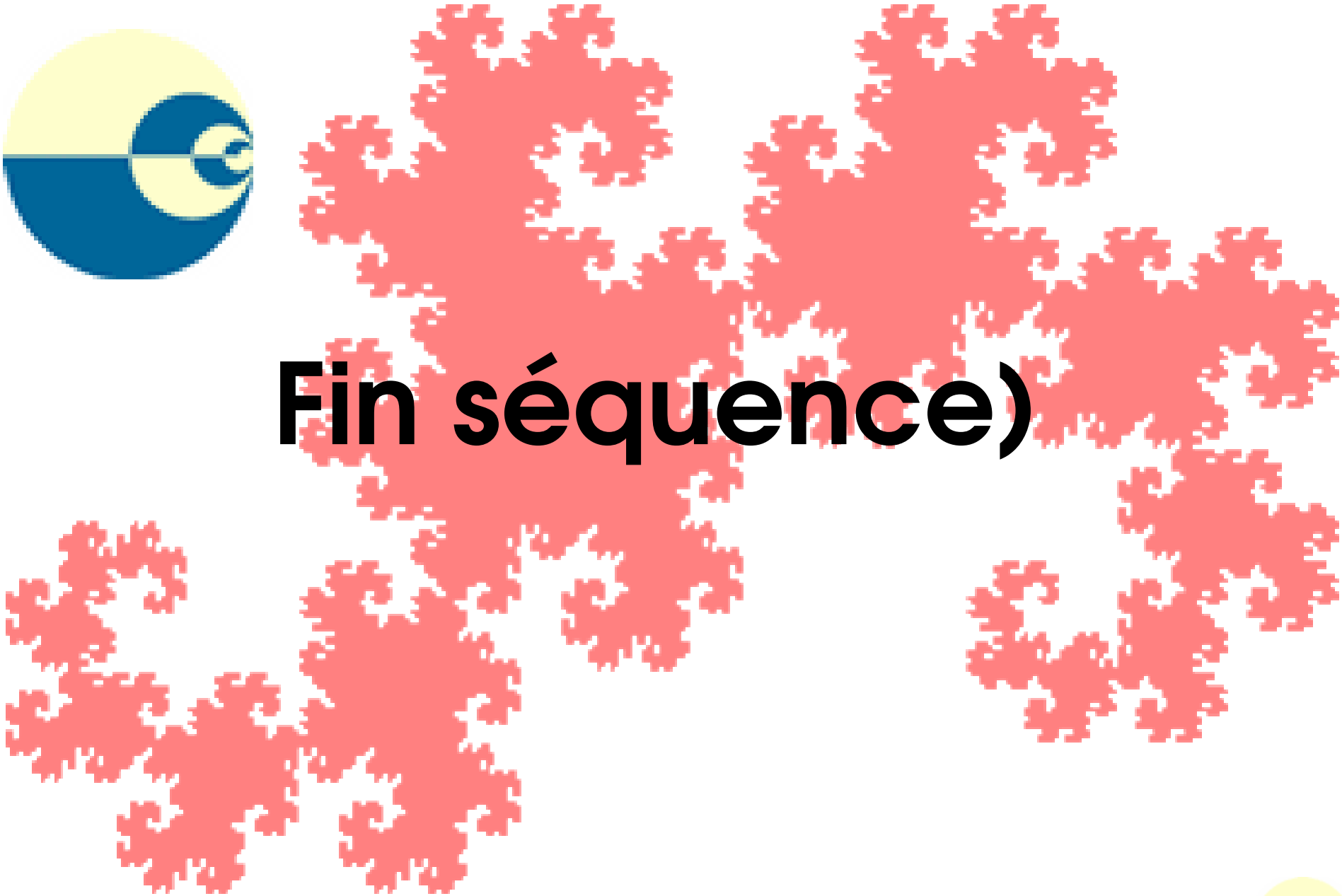
$$n \to n/2 \to n/4 \to \ldots \to n/2^p \Longrightarrow p \approx \log_2 n$$

Dichotomie $\to$ temps d'exécution logarithmique $0(\log n)$

MAIS si l'arbre est dégénéré le nombre de noeuds restant à examiner est seulement *diminué de 1* à chaque fois

$$n \to n-1 \to n-2 \to n-3 \ldots$$

$\to$ temps d'exécution linéaire $0(n)$



Fin séquence)

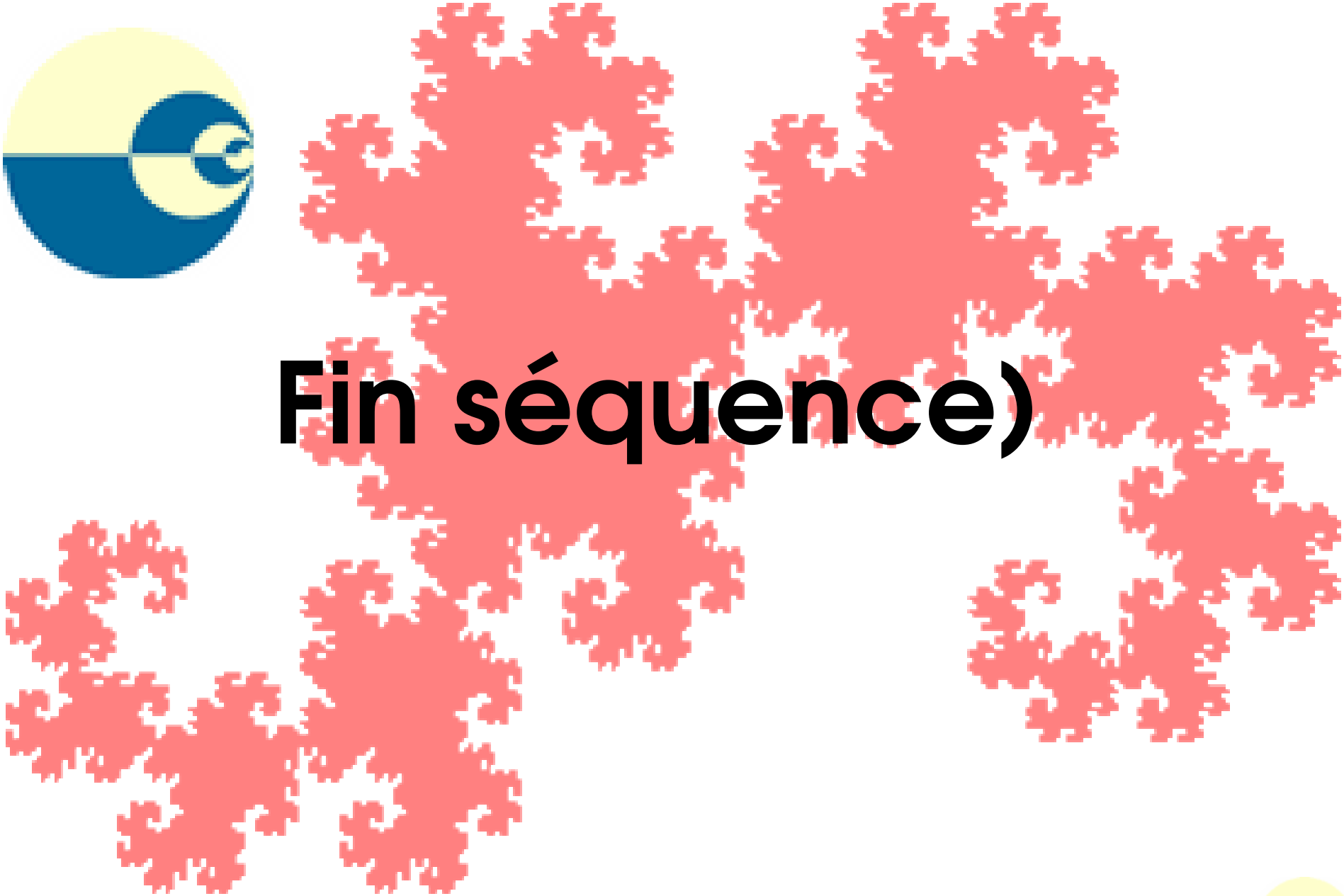
(Séquence 7.3

Recherche d'un élément dans un ABR

Recherche d'un élément

La fonction de recherche est un semi-prédicat

```
;;; abr-recherche: Nombre * ArbreBinRecherche
;;;     -> ArbreBinRecherche + #f
;;; (abr-recherche x ABR) rend l'arbre de racine x,
;;; lorsque x est dans ABR et renvoie #f si x
;;; n'apparaît pas dans ABR
(define (abr-recherche x ABR)
  (if (ab-noeud? ABR)
      (let ((e (ab-etiquette  ABR)))
        (cond ((= x e) ABR)
              ((< x e) (abr-recherche
                        x (ab-gauche ABR)))
              (else (abr-recherche
                     x (ab-droit ABR))) ) )
      #f))
```



Fin séquence)

(Séquence 7.4

Ajout d'un élément

Ajout d'un élément

Principe : ajouter « à l'endroit où » la recherche s'est terminée en échec
Remarque : la fonction d'ajout rend comme résultat un arbre binaire de recherche qui est (re)construit au fur et à mesure (constructeur `ab-noeud`)

Fonction d'ajout

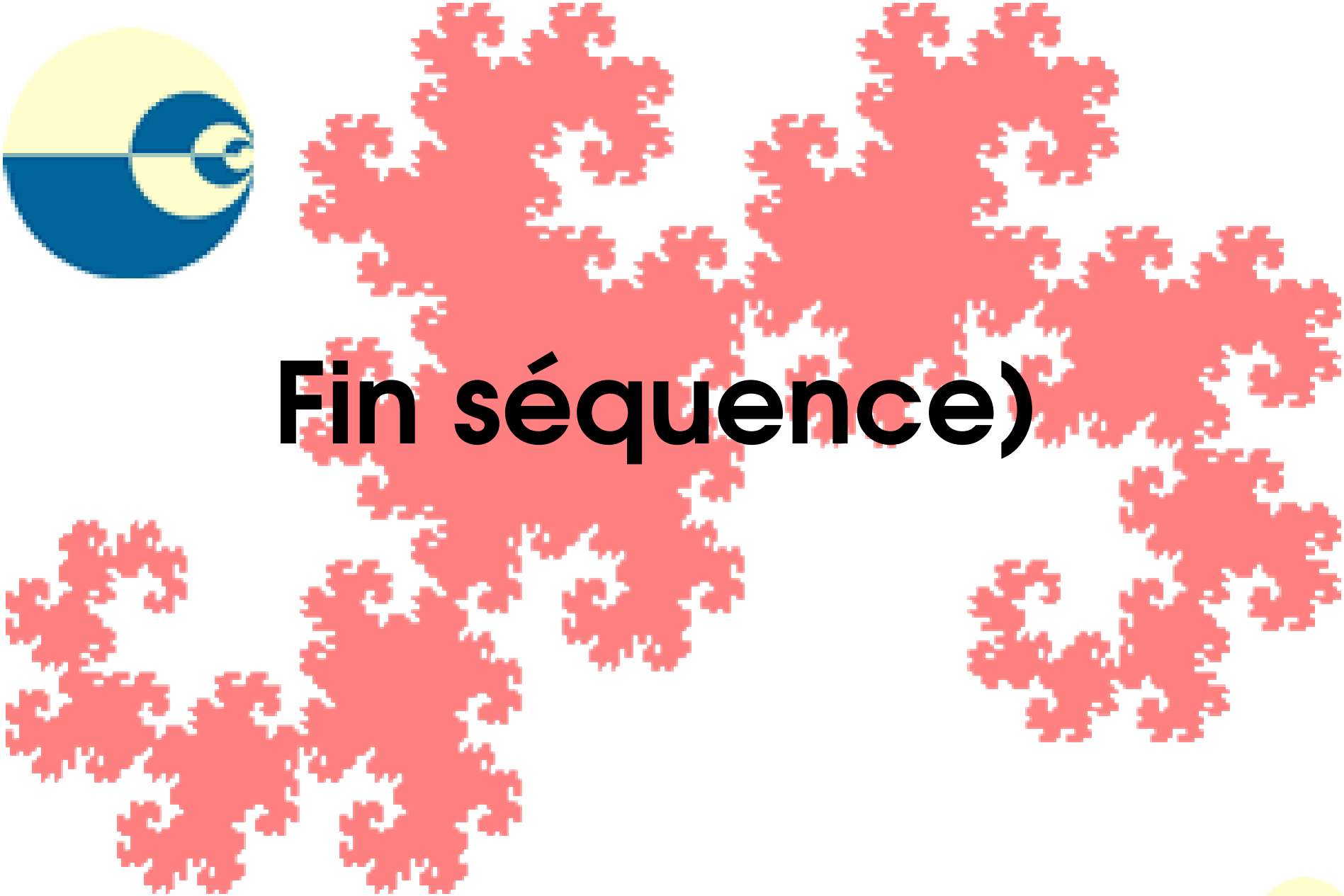
```
;;; abr-ajout: Nombre * ArbreBinRecherche
;;;     -> ArbreBinRecherche
;;; (abr-ajout x ABR) rend l'arbre ABR lorque x est
;;; dans ABR et sinon rend un arbre bin. de rech.
;;; qui contient x et toutes les étiquettes d'ABR
(define (abr-ajout x ABR)
  (if (ab-noeud? ABR)
      (let ((e (ab-etiquette  ABR)))
        (cond
          ((= x e) ABR)
          ((< x e)
           (ab-noeud e (abr-ajout x (ab-gauche ABR))
                       (ab-droit ABR)))
          (else (ab-noeud
                 e
                 (ab-gauche ABR)
                 (abr-ajout x (ab-droit ABR))))))
      (ab-noeud x (ab-vide) (ab-vide))))
```

Suppression d'un élément

C'est la fonction la plus compliquée. Voir le code commenté.

Perspectives

- D'autres implantations avec des propriétés plus fortes comme les AVL (Georgii Adelson-Velsky et Evguenii Landis, 1962) : arbres automatiquement quasi-équilibrés (la différence des profondeurs entre fils gauche et droit est au plus de 1).
- Introduction d'aléa pour prévenir (statistiquement) la fabrication d'arbres dégénérés.



Fin séquence)

(Séquence 7.5

Suppression d'un élément

Suppression d'un élément

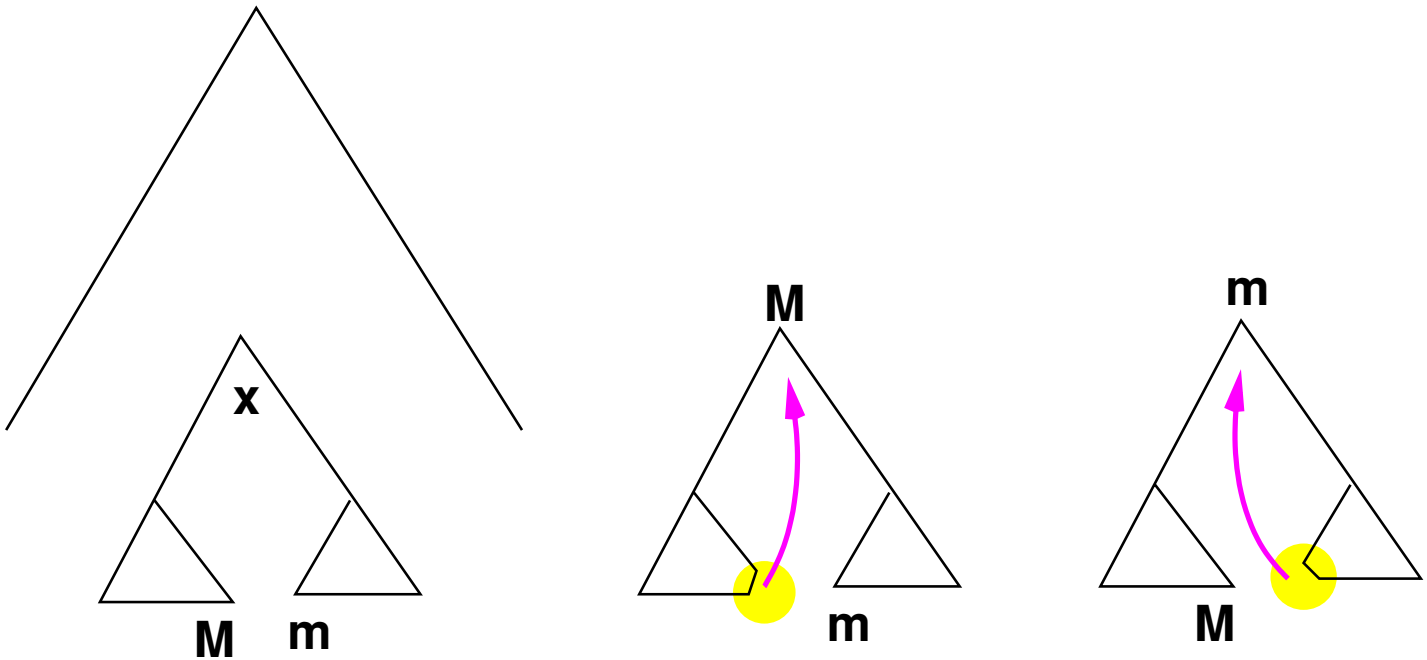
Principe :

- rechercher le sous-arbre dont l'élément à supprimer est la racine
- effectuer la suppression
  - la suppression est simple si l'élément est une feuille
  - la suppression est plus complexe sinon : modifier le sous-arbre en remplaçant la racine par son « précédent » dans le sous-arbre. *« précédent »* $\equiv$ *le plus grand des éléments inférieurs à la racine*
    - les éléments inférieurs forment le sous-arbre gauche
    - le plus grand des inférieurs est au bout de la branche droite

Remarque : autre façon pour supprimer la racine d'un ABR : la remplacer par son « suivant » (le plus petit des éléments supérieurs à la racine)

Suppression d'un élément

Pour supprimer l'élément *x*



Fonction de suppression

```
;;; abr-moins: Nombre * ArbreBinRecherche
;;;     -> ArbreBinRecherche
;;; (abr-moins x ABR) rend ABR lorque x n'y est pas
;;; et sinon rend un arbre bin. de rech. qui contient
;;; toutes les étiquettes de ABR sauf x
(define (abr-moins x ABR)
  (if (ab-noeud? ABR)
      (let ((e (ab-etiquette ABR)))
        (cond
          ((= x e) (moins-racine ABR)) ; supprimer racin
          ((< x e) (ab-noeud
                    e
                    (abr-moins x (ab-gauche ABR))
                    (ab-droit ABR)))
          (else (ab-noeud
                 e
                 (ab-gauche ABR)
                 (abr-moins x (ab-droit ABR))))))
      (ab-vide)))
```

Fonction de suppression de la racine

```
;;; moins-racine: ArbreBinRecherche -> ArbreBinRecherche
;;; (moins-racine ABR) rend l'arbre bin. de rech. qui
;;; contient toutes les étiquettes qui apparaissent
;;; dans ABR hormis l'étiquette de sa racine.
;;; HYPOTHÈSE: ABR non vide
(define (moins-racine ABR)
  (cond ((not (ab-noeud? (ab-gauche ABR)))
         (ab-droit ABR))
        ((not (ab-noeud? (ab-droit ABR)))
         (ab-gauche ABR))
        (else
          défaire le ss-ab-g -> max + ABR privé du max )))
```

Le max d'un arbre binaire de recherche

Cette fonction renvoie le max d'un arbre binaire de recherche

```
(define (max-abr ABR)
  (if (ab-noeud? (ab-droit ABR))
      (max-abr (ab-droit ABR))
      (ab-etiquette ABR)))
```

Arbre binaire de recherche sans son max

Cette fonction renvoie un ABR

```
(define (abr-sauf-max ABR)
  (if (ab-noeud? (ab-droit ABR))
      (ab-noeud (ab-etiquette ABR)
                (ab-gauche ABR)
                (abr-sauf-max (ab-droit ABR)))
      (ab-gauche ABR)))
```

Le max ET de l'arbre privé du max

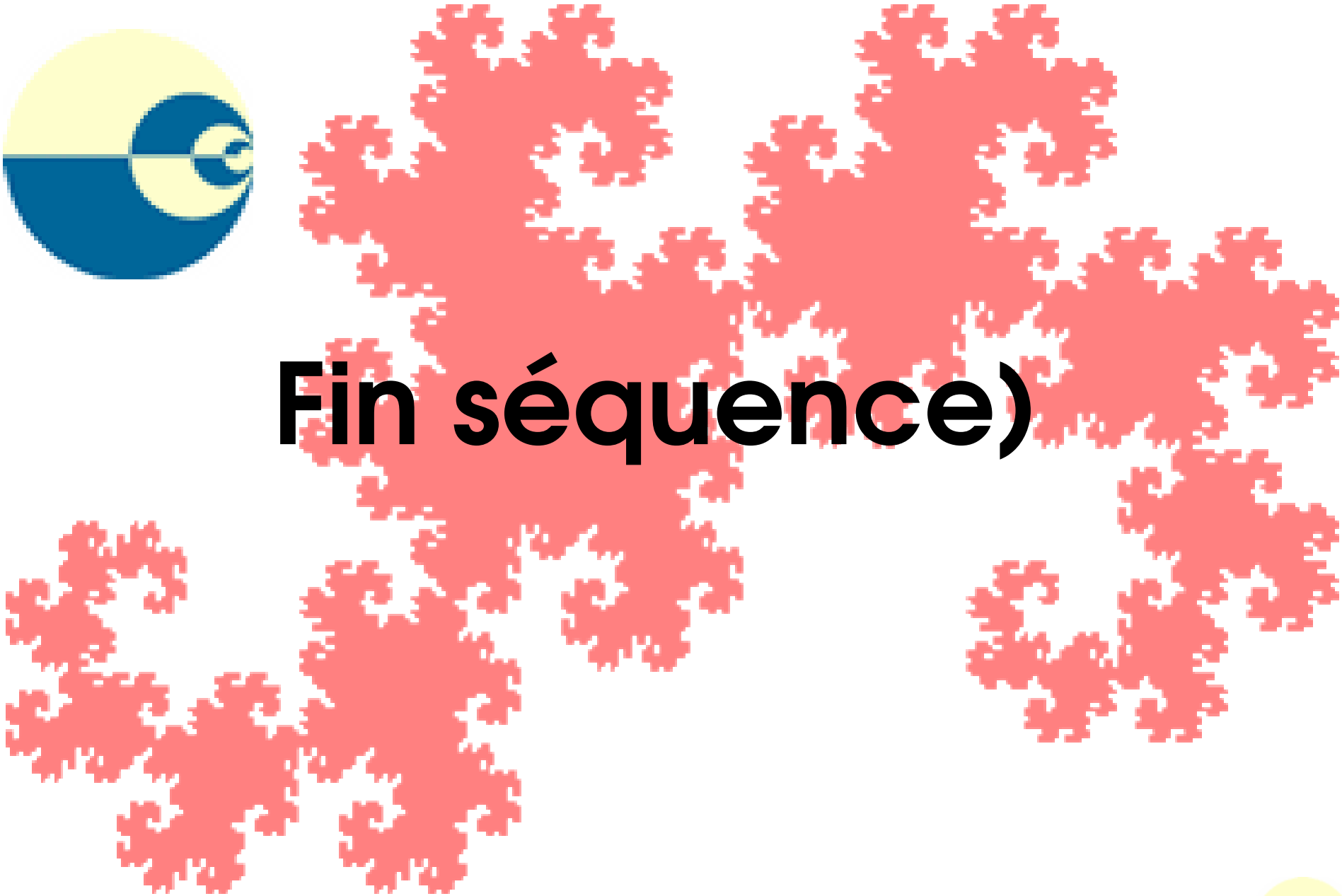
```
(define (max-sauf-max ABR)
  (if (ab-noeud? (ab-droit ABR))
      (let ((m-sm-ss-ab-d (max-sauf-max (ab-droit ABR))))
        (list (car m-sm-ss-ab-d)
              (ab-noeud (ab-etiquette ABR)
                        (ab-gauche ABR)
                        (cadr m-sm-ss-ab-d))))
      (list (ab-etiquette ABR)
            (ab-gauche ABR) ) ) )
```

Fonction de suppression de la racine

```
(define (moins-racine ABR)
  (cond ((not (ab-noeud? (ab-gauche ABR)))
         (ab-droit ABR))
        ((not (ab-noeud? (ab-droit ABR)))
         (ab-gauche ABR))
        (else ; defaire le ss-ab-g -->  max + ABR prive
         (let ((m-sm-ss-ab-g (max-sauf-max (ab-gauche AB
           (ab-noeud (car m-sm-ss-ab-g)
                     (cadr m-sm-ss-ab-g)
                     (ab-droit ABR)))))))
```

Perspectives

- D'autres implantations avec des propriétés plus fortes comme les AVL (Georgii Adelson-Velsky et Evguenii Landis, 1962) : arbres automatiquement quasi-équilibrés (la différence des profondeurs entre fils gauche et droit est au plus de 1).
- Introduction d'aléa pour prévenir (statistiquement) la fabrication d'arbres dégénérés.



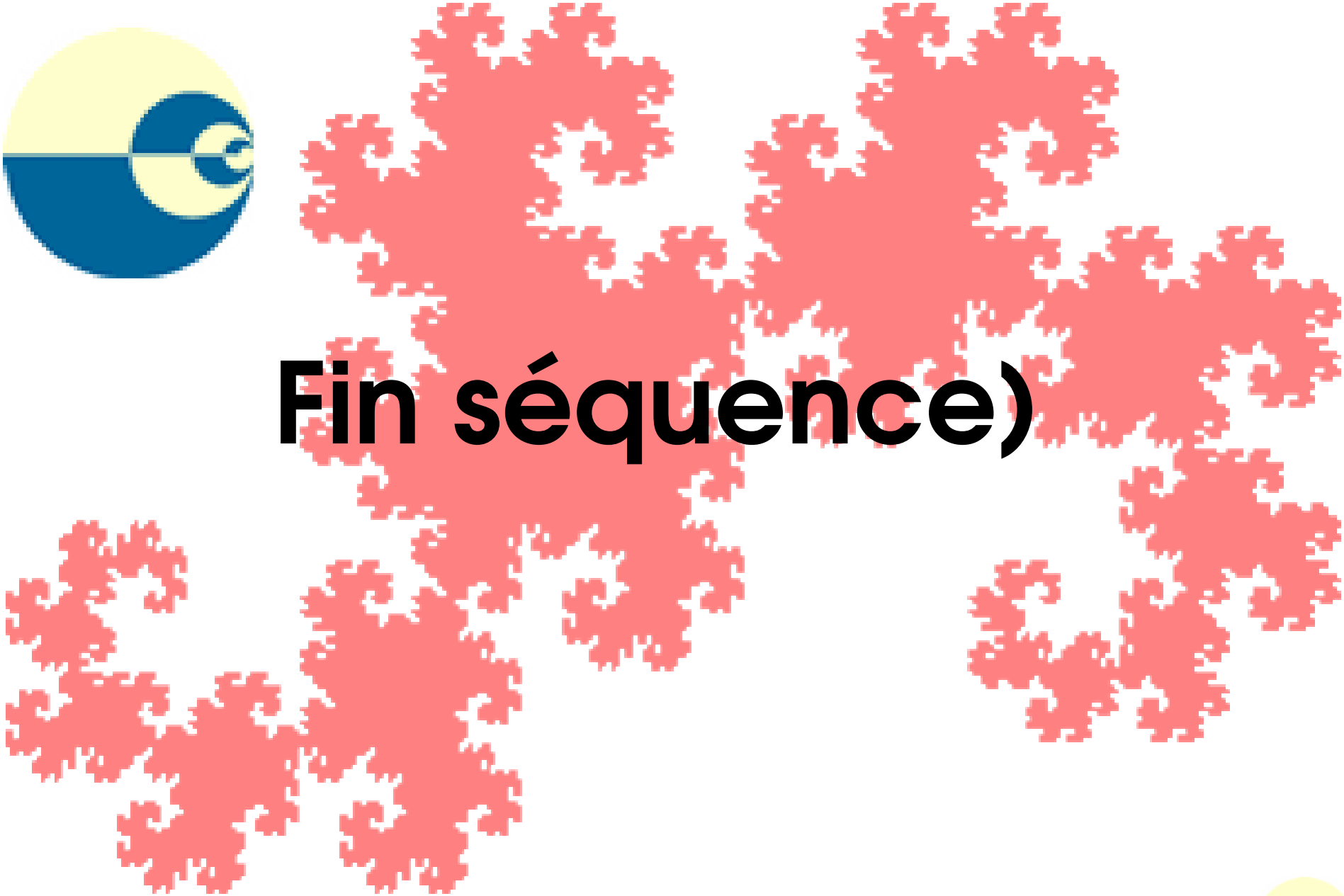
Fin séquence)

(Séquence 7.6

Résumé

Résumé

- Au cours de cette semaine, vous avez découvert un emploi efficace d'arbres binaires : les arbres binaires de recherche associés à un algorithme de recherche **dichotomique**.



Fin séquence)

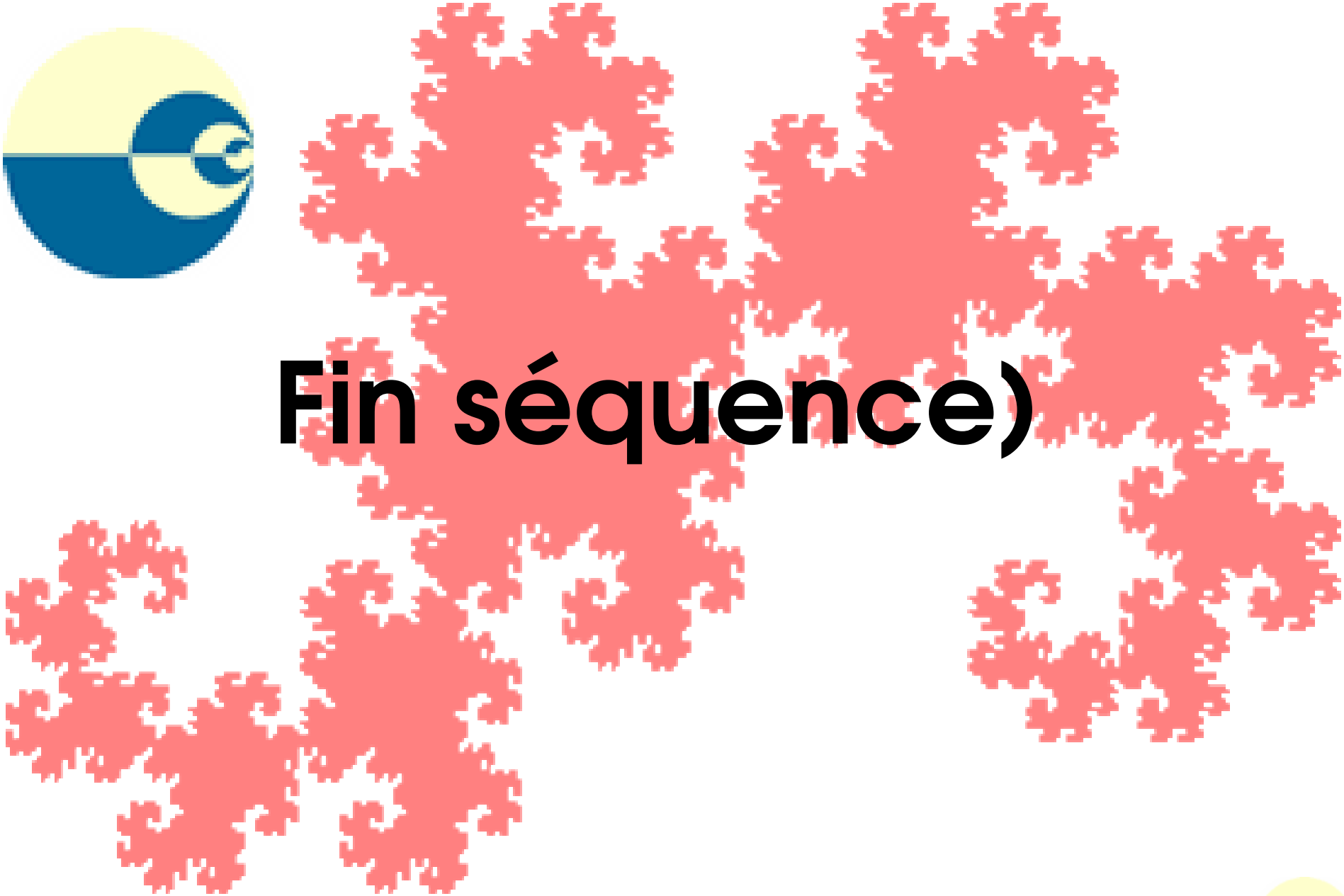
(Séquence 8.0

= Plan semaine 8

Plan semaine 8

Structures arborescentes : arbres généraux

- Définition et exemples
- Barrière d'abstraction des arbres généraux
- Schéma de récursion sur les arbres généraux
- Exemples : nombre de nœuds, profondeur, liste préfixe, affichage



Fin séquence)

(Séquence 8.1

Arbres généraux

Définition d'un arbre général

Dans un arbre général, chaque nœud porte une information (étiquette de type α) et a un nombre quelconque de descendants immédiats.
Le type est noté **ArbreGeneral(α)**

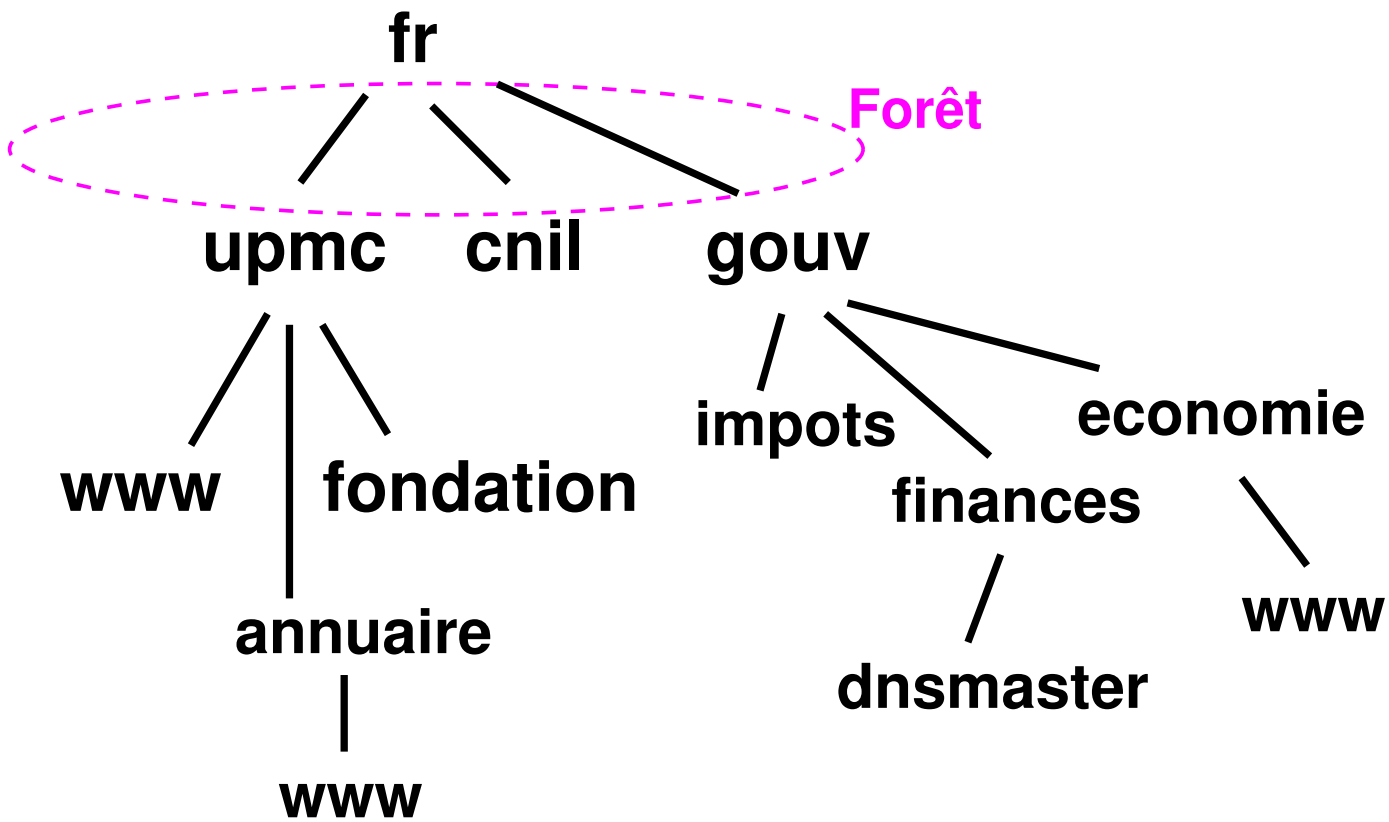
Définition d'un arbre général

Définition (récursive) Un arbre général est formé

- d'un nœud (portant une étiquette de type α)
- et d'une forêt (liste de sous-arbres) de type **LISTE(ArbreGeneral(α))**

Il n'y a pas d'arbre général vide, mais une forêt peut être vide

Exemple



Barrière d'abstraction des arbres généraux

- **Constructeur** pour construire un arbre général : `ag-noeud`
- **Accesseurs** pour accéder aux parties d'un arbre général : `ag-etiquette` et `ag-foret`
- **Reconnaisseur** inutile, puisqu'il n'y a qu'un seul constructeur

Remarque : les forêts sont des listes donc sont manipulées avec les primitives sur les listes

Spécification du constructeur

```
;;; ag-noeud: α * Foret[α] -> ArbreGeneral[α]
;;; avec Foret[α] = LISTE[ArbreGeneral[α]]
;;; (ag-noeud e foret) rend l'arbre formé de la
;;; racine d'étiquette e et, comme sous-arbres
;;; immédiats, les arbres de la forêt foret.
```

Exemples : arbres généraux de nombres

```
(ag-noeud 3 (list))    → #<object>
```

et construit l'arbre avec un unique nœud d'étiquette 3.

Spécification des accesseurs

```
;;; ag-etiquette: ArbreGeneral[α] -> α
;;; (ag-etiquette g) rend l'étiquette de la racine
;;; de l'arbre g.

;;; ag-foret: ArbreGeneral[α] -> Foret[α]
;;; avec Foret[α] = LISTE[ArbreGeneral[α]]
;;; (ag-foret g) rend la forêt des sous-arbres
;;; immédiats de g.
```

Propriétés algébriques

- Pour toute forêt d'arbres généraux F, et toute valeur v

```
(ag-etiquette (ag-noeud v F)) ≡ v
(ag-foret (ag-noeud v F))     ≡ F
```

- Pour tout arbre général G

```
(ag-noeud (ag-etiquette G)(ag-foret G)) ≡ G
```

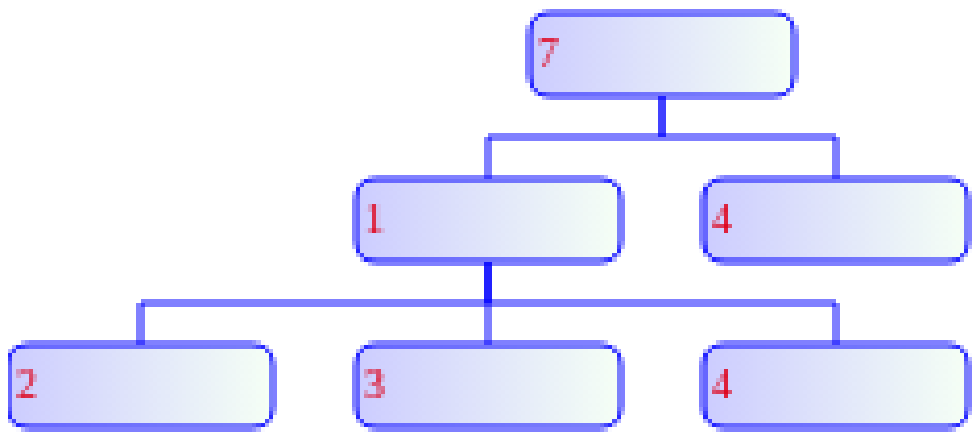
Reconnaisseur dérivé

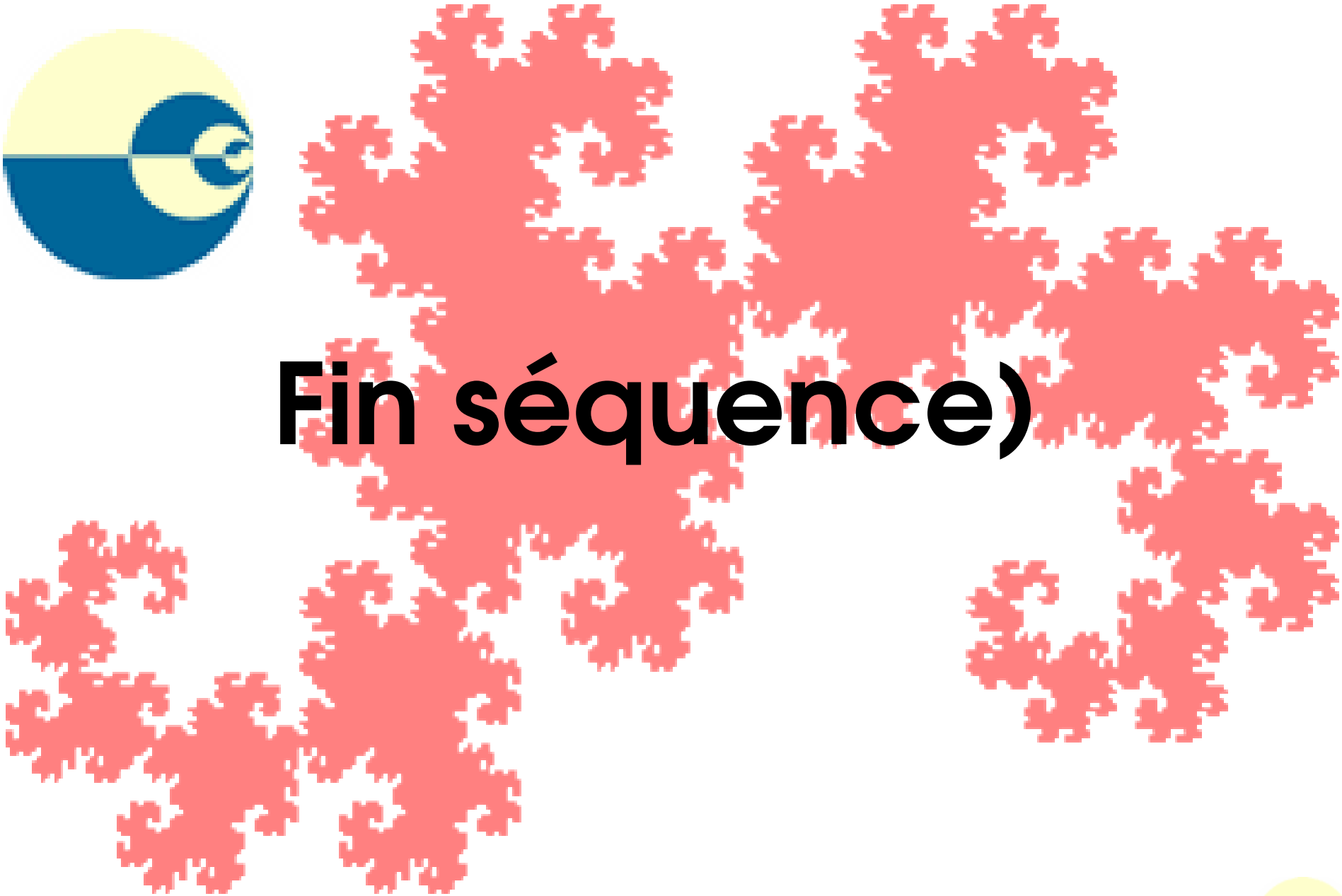
Définir le prédicat `ag-feuille?`

```
;;; ag-feuille?: ArbreGeneral[α] -> bool
;;; (ag-feuille? G) rend vrai ssi G est une feuille
(define (ag-feuille? G)
  (not (pair? (ag-foret G))) )
```

Une fonction d'affichage

```
(let* ((g1 (ag-noeud 2 '()))
       (g2 (ag-noeud 3 '()))
       (g3 (ag-noeud 4 '()))
       (g4 (ag-noeud 1 (list g1 g2 g3)))
       (g5 (ag-noeud 7 (list g4 g3))))
  (ag-affiche g5))
```





Fin séquence)

(Séquence 8.2

Récursion sur les arbres généraux

Récursion sur les arbres généraux

Un arbre général est constitué :

- d'une étiquette
- et d'une forêt (liste) d'arbres généraux

Traitement d'un arbre :

- Pour traiter un arbre, il faut traiter la forêt de ses descendants directs.
- Pour traiter une forêt, il faut traiter chaque arbre de cette liste.

C'est une **récursion croisée**.

Les itérateurs `map` et `reduce` sont utiles.

Schéma récursif sur les arbres généraux

```
;;; ArbreRec: ArbreGeneral[α] -> β
(define (ArbreRec G)
   (combinaison1 (ag-etiquette G)
                 (ForetRec (ag-foret G)) ) )

;;; ForetRec: LISTE[ArbreGeneral[α]] -> β
(define (ForetRec F)
   (if (pair? F)
      (combinaison2 (ArbreRec (car F))
                    (ForetRec (cdr F)) )
      base ) )
```

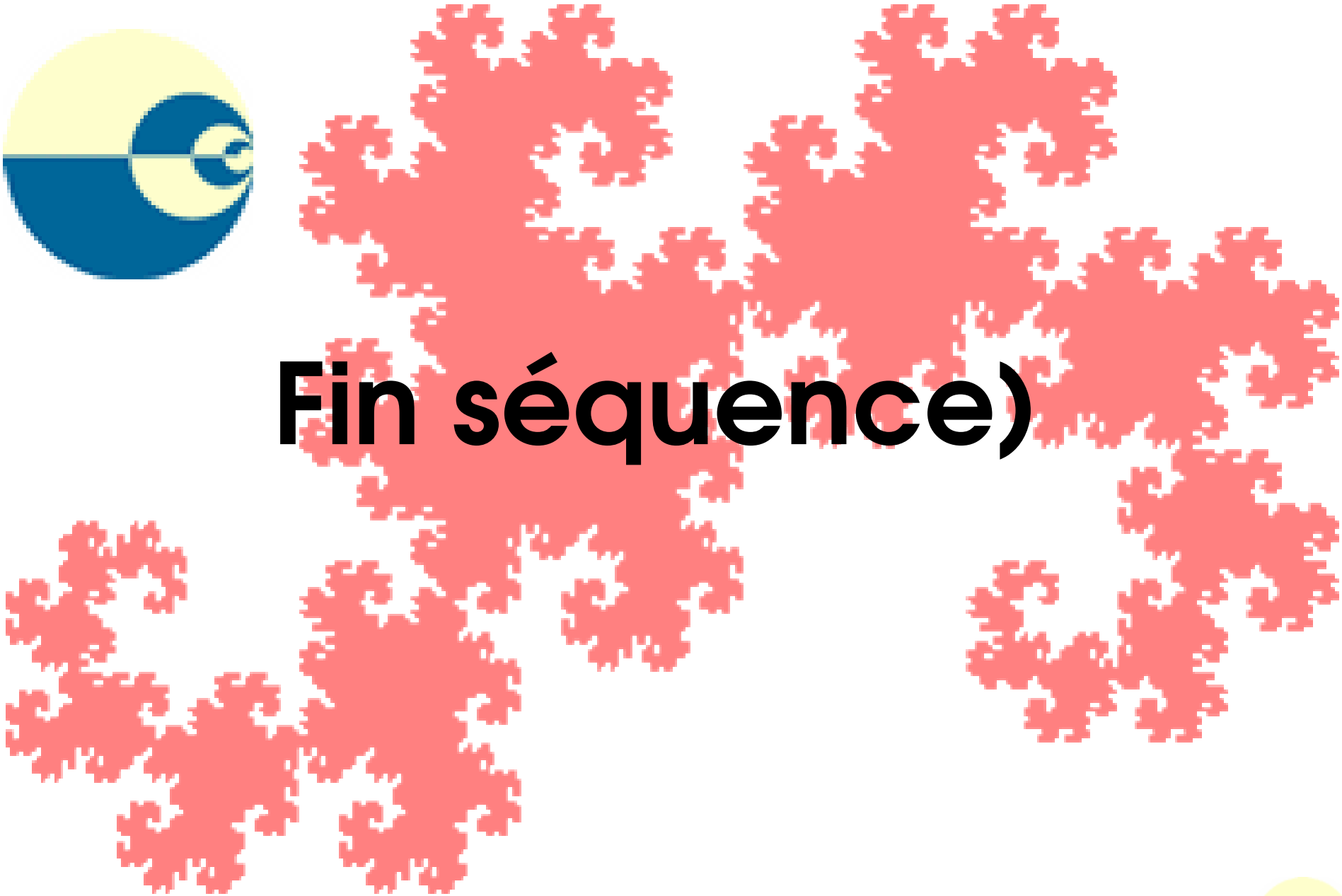
Schéma récursif sur les arbres généraux

Avec usage des itérateurs `map` et `reduce` :

```
;;; ForetRec: LISTE[ArbreGeneral[α]] -> β
(define (ForetRec F)
  (reduce combinaison2 base (map ArbreRec F)) )
```

et même en une seule fonction:

```
;;; ArbreRec: ArbreGeneral[α] -> β
(define (ArbreRec G)
  (combinaison1 (ag-etiquette G)
                (reduce combinaison2
                        base
                        (map ArbreRec (ag-foret G)) ) ) )
```



Fin séquence)

(Séquence 8.3

Exemples

Nombre de nœuds

```
;;; nombre-noeuds-arbre: ArbreGeneral[α] -> nat
;;; (nombre-noeuds-arbre G) rend le nombre de
;;; noeuds de G
(define (nombre-noeuds-arbre G)
  ;; nombre-noeuds-foret: Foret[α] -> nat
  ;; (nombre-noeuds-foret F) rend le nombre de
  ;; noeuds de F
  (define (nombre-noeuds-foret F)
    (if (pair? F)
        (+ (nombre-noeuds-arbre (car F))
           (nombre-noeuds-foret (cdr F)))
        0 ) )
  (+ 1 (nombre-noeuds-foret (ag-foret G))) )
```

Avec un `map`

```
;;; nombre-noeuds-arbre: ArbreGeneral[α] -> nat
;;; (nombre-noeuds-arbre G) rend le nombre de
;;; noeuds de G
(define (nombre-noeuds-arbre G)
  ;; nombre-noeuds-foret: Foret[α] -> nat
  ;; (nombre-noeuds-foret F) rend le nombre de
  ;; noeuds de F
  (define (nombre-noeuds-foret F)
    (reduce + 0 (map nombre-noeuds-arbre F)))
  (+ 1 (nombre-noeuds-foret (ag-foret G))))
```

ou encore:

```
(define (nombre-noeuds-arbre G)
  (reduce + 1 (map nombre-noeuds-arbre (ag-foret G))))
```

Profondeur

```
;;; ag-profondeur: ArbreGeneral[α] -> nat
;;; (ag-profondeur G) rend la profondeur de
;;; l'arbre G
(define (ag-profondeur G)
  ;; profondeurForet: Foret[α] -> nat
  ;; (profondeurForet F) rend la profondeur de F
  (define (profondeurForet F)
    (if (pair? F)
        (max (ag-profondeur (car F))
             (profondeurForet (cdr F)))
        0))
  (+ 1 (profondeurForet (ag-foret G))))
```

ou avec des itérateurs:

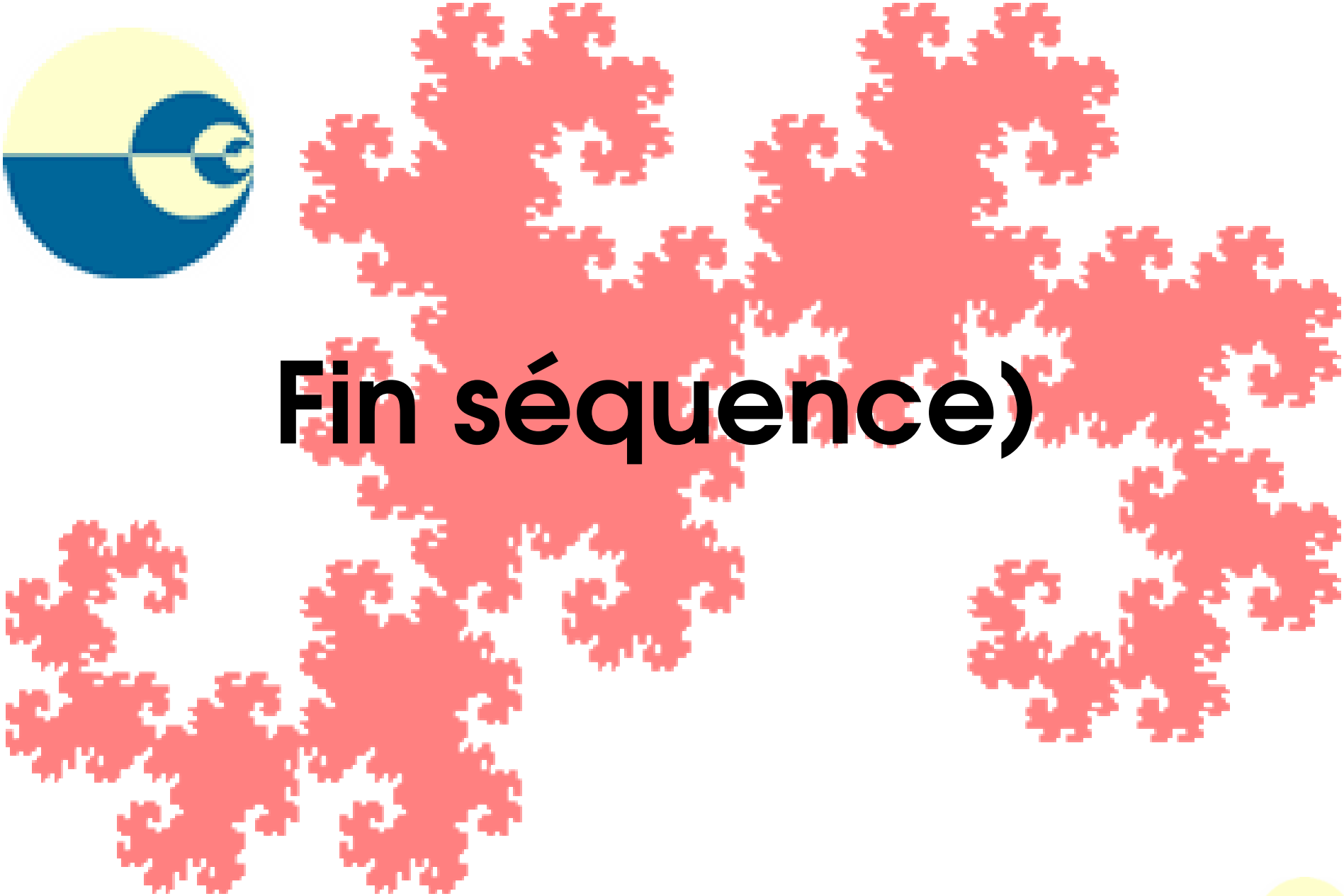
```
(define (ag-profondeur G)
  (+ 1 (reduce max 0 (map ag-profondeur (ag-foret G)))))
```

Liste préfixe

```
;;; ag-liste-prefixe: ArbreGeneral[α] -> LISTE[α]
;;; (ag-liste-prefixe G) rend la liste préfixe
;;; des étiquettes de l'arbre G
(define (ag-liste-prefixe G)
  ;; liste-prefixe-foret: Foret[α] -> LISTE[α]
  ;; rend la concaténation des listes préfixes
  ;; des étiquettes des arbres de F
  (define (liste-prefixe-foret F)
    (if (pair? F)
        (append (ag-liste-prefixe (car F))
                (liste-prefixe-foret (cdr F)))
        '() ) )
  (cons (ag-etiquette G)
        (liste-prefixe-foret (ag-foret G)) ) )
```

Avec des itérateurs

```
(define (ag-liste-prefixe G)
  (cons (ag-etiquette G)
        (reduce append
                '()
                (map ag-liste-prefixe (ag-foret G)))))
```



Fin séquence)

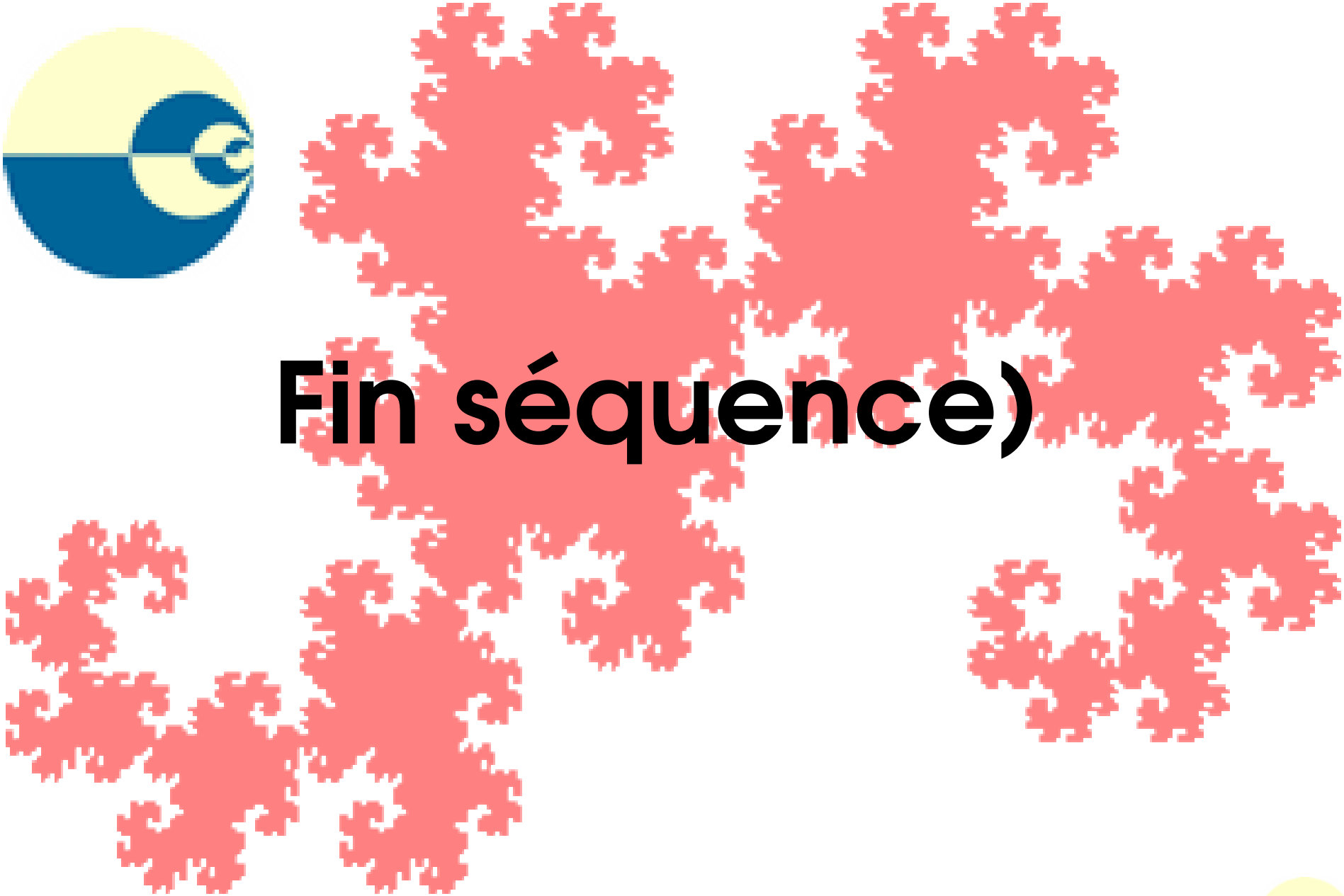
(Séquence 9.0

= Plan semaine 9

Plan semaine 9

Arbres généraux et S-expressions

- Définition d'une S-expression
- Évaluation d'une expression arithmétique

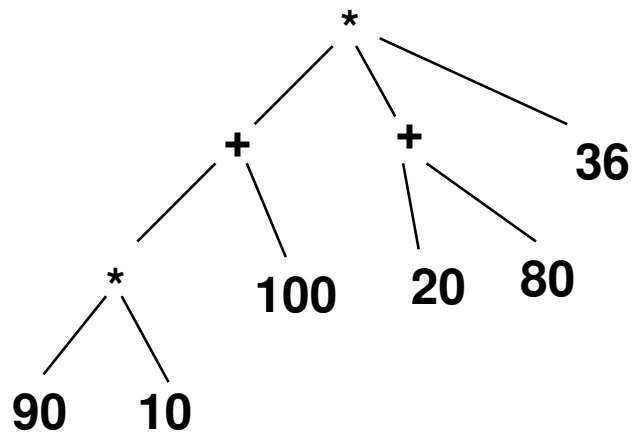


Fin séquence)

(Séquence 9.1

S-expression

S-expressions




```
(* (+ (* 90 10) 100)
   (+ 20 80)
   36)
```

L'expression Scheme ci-dessus ressemble à une liste (de quatre éléments) de type **LISTE(α)** mais les éléments de la liste ne sont pas de même type. On peut aussi voir cette expression comme un arbre général.
C'est une ***S-expression*** (pour *expression symbolique*)

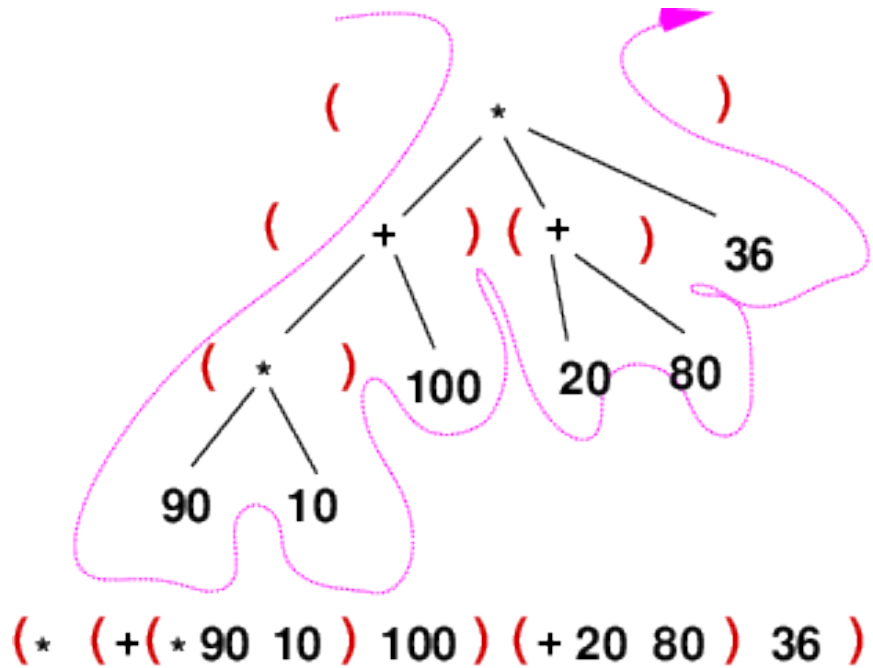
Syntaxe des S-expressions

<Sexpression> → <atome>
(<Sexpression>*)
<atome> → <Nombre>
<bool>
<string>
<Symbole>

Les ***lexèmes*** <Nombre>, <bool>, <string> et <Symbole> sont définis dans la carte de référence.

Arbres généraux et S-expressions

Un arbre général peut être représenté par une S-expression dont le premier élément est l'étiquette de la racine et dont le reste est constitué de la forêt de ses sous-arbres immédiats.



Implantation des AG par S-expressions

```
(define (ag-noeud e F)
  (cons e F))

(define (ag-etiquette ag)
  (car ag))

(define (ag-foret ag)
  (cdr ag))
```

Pourquoi alors avoir une barrière d'abstraction pour les arbres généraux ?

- Pour se concentrer sur la récursion dans les arbres généraux,
- parce qu'il y a d'autres implantations pour les arbres

Autre implantation des AG

```
(define (ag-noeud e F)
  (list "*AG*" e F))

(define (ag-etiquette ag)
  (if (and (pair? ag)
           (equal? (car ag) "*AG*") )
      (cadr ag)
      (erreur 'ag-etiquette "Pas un AG" ag) ) )

(define (ag-foret ag)
  (if (and (pair? ag)
           (equal? (car ag) "*AG*") )
      (caddr ag)
      (erreur 'ag-etiquette "Pas un AG" ag) ) )
```

Implantation par les S-expressions

Pourquoi des S-expressions ?

Implantation par les S-expressions

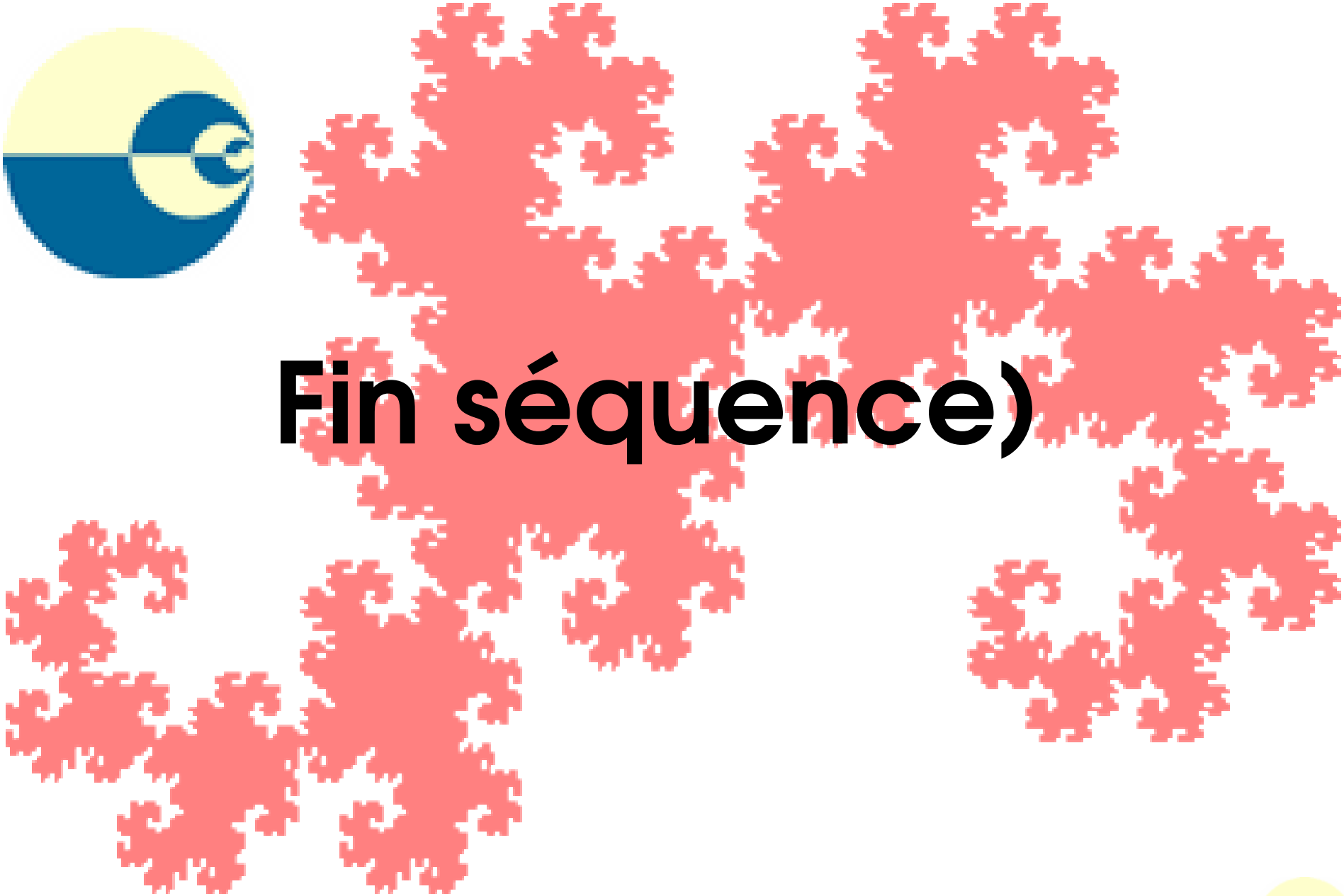
Pourquoi des S-expressions ?

- Parce qu'elles sont simples à citer.

```
(ag-noeud '+
          (ag-noeud '* (ag-noeud 90 (list))
                       (ag-noeud 10 (list)) )
          (ag-noeud 100 (list)) )
```

s'écrit directement :

```
'(+ (* 90 10) 100)
```



Fin séquence)

(Séquence 9.2

Évaluation d'expression arithmétique

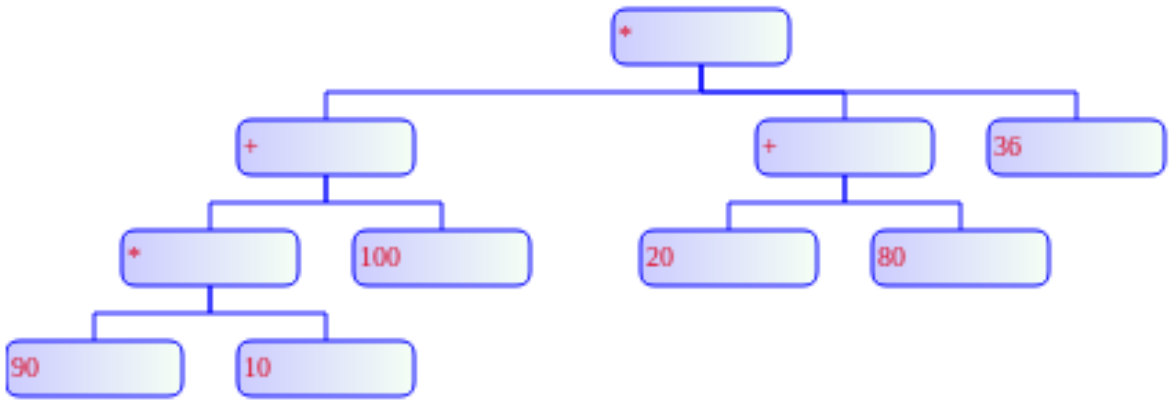
Expression arithmétique

Le but est d'évaluer des expressions arithmétiques, bien formées, dotées des seuls opérateurs `+` et `*`

- constante c'est-à-dire sans variable
  `(((90 * 10) + 100) * (20 + 80) * 36)`
- ou dans un **environnement** avec variables.
  `((90 * x) + y)`

Représentation expression arithmétique

- Visualisation et représentation par un arbre général (type `ExprArbre`)

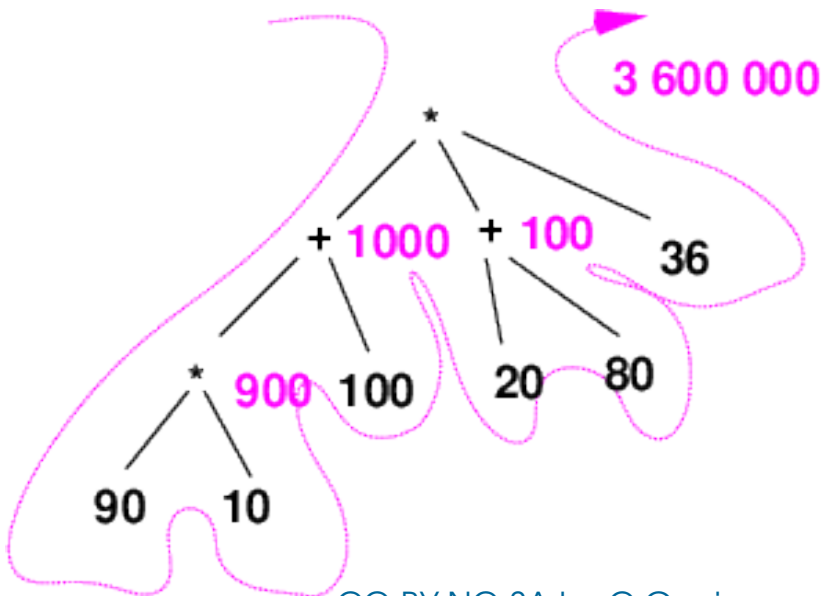


- Représentation par une S-expression (type `S-Expr`)

```
(* (+ (* 90 10) 100) (+ 20 80) 36)
(+ (* 90 x) y)
```

Définition de `evaluationExprArbre`

```
;;; evaluationExprArbre: ExprArbre-> Nombre
;;; (evaluationExprArbre G) rend la valeur de
;;; l'expression arithmetique représentée par G
(define (evaluationExprArbre G)
  (if (ag-feuille? G)
      (ag-etiquette G)
      (let ((args (map evaluationExprArbre
                       (ag-foret G))))
        (application (operation (ag-etiquette G))
                     args ))))
```



Définition de `operation`

```
;;; operation: Symbole -> Opération
;;; (operation operateur) rend la fonction
;;; correspondant au symbole operateur
(define (operation operateur)
  (cond ((equal? operateur '+) +)
        ((equal? operateur '*) *)
        (else (erreur 'operation
                      operateur
                      "pas défini"))))
```

Définition de `application`

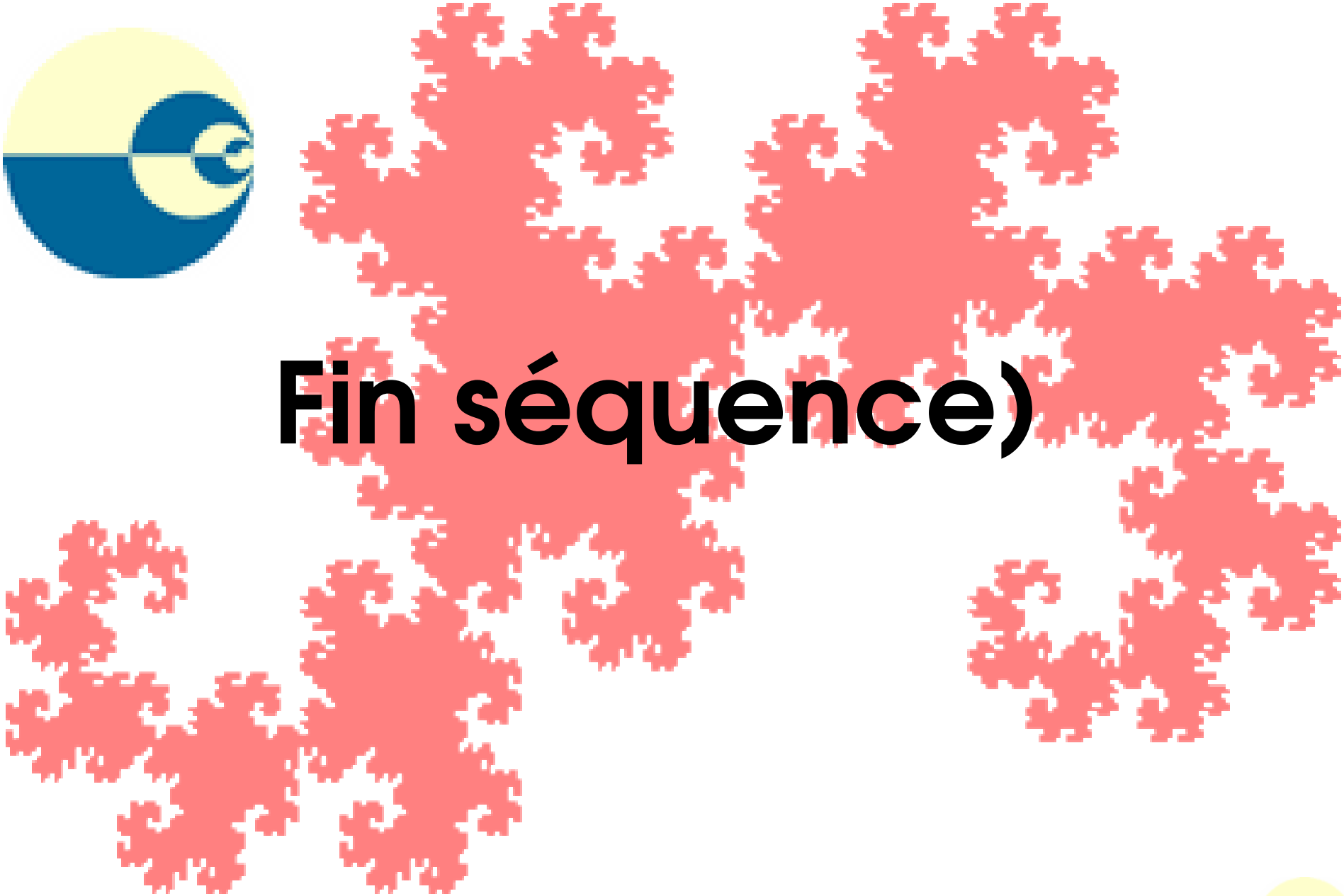
```
;;; application: Opération * LISTE[Nombre] -> Nombre
;;; (application op args) rend le résultat de
;;; l'application de l'opération op à la liste
;;; d'arguments args
;;; HYPOTHÈSE: args non vide
(define (application op args)
  (if (pair? (cdr args))
      (op (car args) (application op (cdr args)))
      (op (car args))))
```

Définition de `evaluation-sexpr`

Avec des Sexpressions plutôt que des arbres généraux, cela donne :

```
;;; evaluation-sexpr: S-Expr -> Nombre
;;; (evaluation-sexpr E) rend la valeur de
;;; l'expression arithmétique E
(define (evaluation-sexpr E)
  (if (pair? E)
      (let ((args (map evaluation-sexpr (cdr E))))
        (application (operation (car E)) args))
      E))
```

```
(evaluation-sexpr '(* (+ (* 90 10) 100) (+ 20 80) 36))
  → 3600000
```



Fin séquence)

(Séquence 9.3

Évaluation dans un environnement

Évaluation dans un environnement

L'environnement est représenté par une liste d'associations

```
Environnement ≡ LISTE[COUPLE[Symbole Nombre]]
```

```
(define (MonEnv) '((x 3) (y 4) (z 5)))

(valeur-de 'y (MonEnv)) → 4
```

La fonction d'évaluation doit donc avoir 2 arguments :

- l'expression elle même
- et la liste d'associations représentant l'environnement.

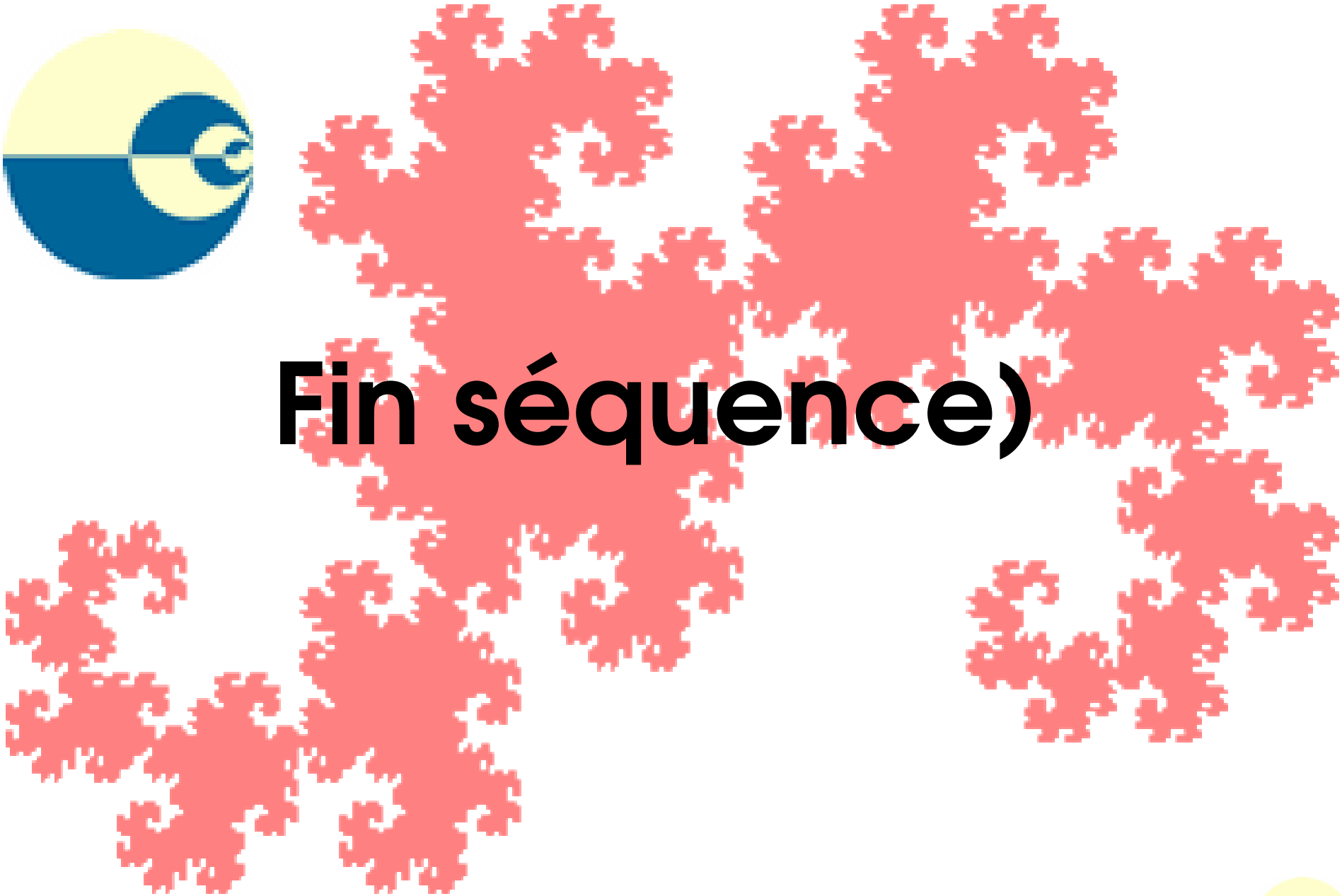
Spécification et application

```
;;; evaluation-expr: S-Expr
;;;     * LISTE[COUPLE[Symbole Nombre]] -> Nombre
;;; (evaluation-expr E env) rend la valeur de
;;; l'expression arithmétique E dans
;;; l'environnement env
```

```
(define (MonEnv) '((x 3) (y 4) (z 5)))

(evaluation-expr '(+ x (* 3 y 5) (+ 1 2 y (* z 10)))
                 (MonEnv))
  → 120
```

Définition de `evaluation-expr`

```
(define (evaluation-expr E env)
  ;; evaluation-expr-env: S-Expr -> Nombre
  ;; (evaluation-expr-env E): même sémantique
  ;; que evaluation-expr mais fonction rendue
  ;; unaire pour utilisation dans un map
  (define (evaluation-expr-env E)
    (evaluation-expr E env))

  (if (pair? E)
      (let ((args (map evaluation-expr-env
                       (cdr E) )))
        (application (operation (car E)) args))
      (if (symbol? E)
          (valeur-de E env)
          E)))
```



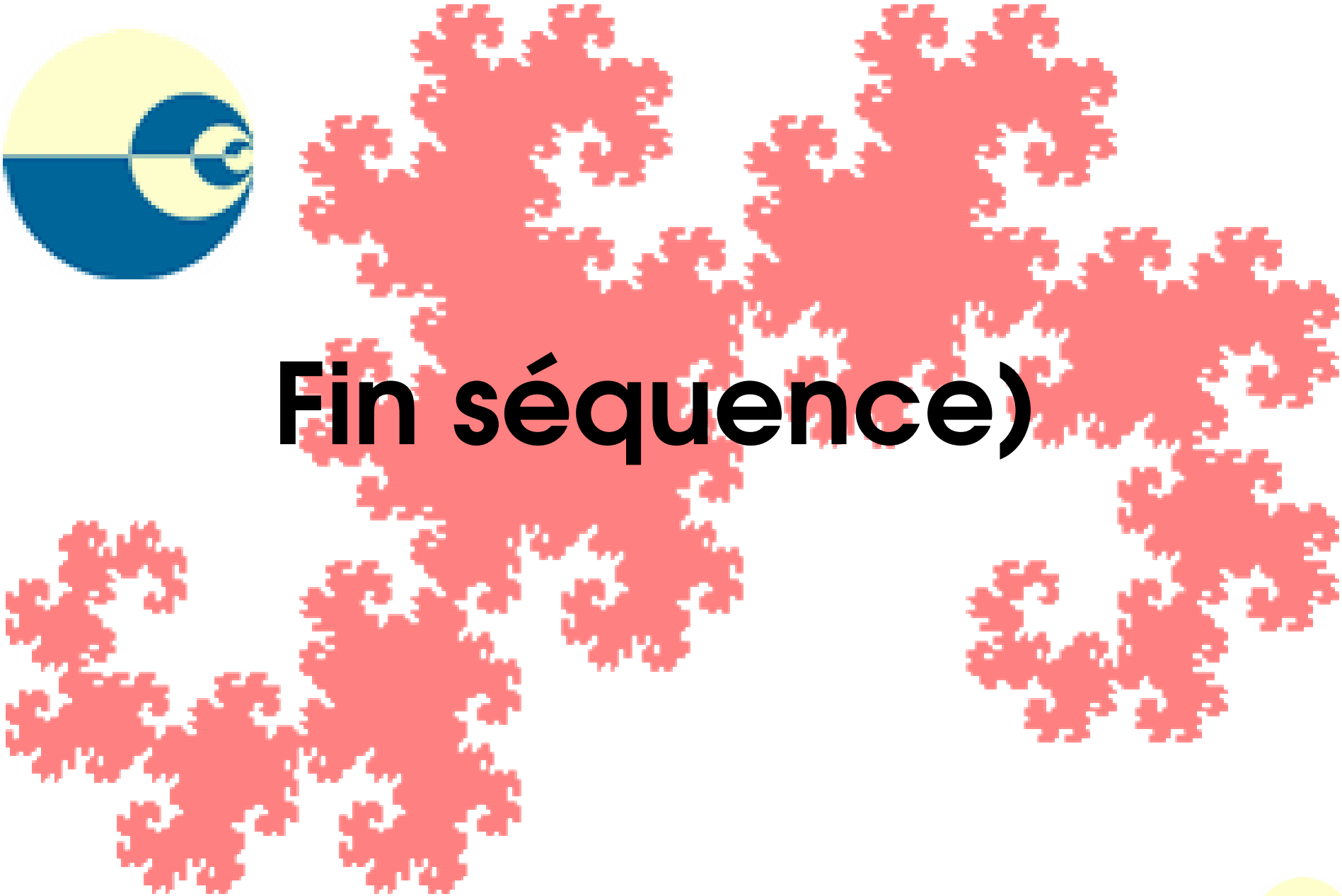
Fin séquence)

(Séquence 10.0

= Plan semaine 10

Plan semaine 10

- Langages et Scheme
- Processus d'évaluation
- Conclusions



Fin séquence)

(Séquence 10.1

Langages et Scheme

Langages

- Système d'écriture (syntaxe/sémantique)
  - mots-clés
  - fonctions (bibliothèques)

Langages

- Système d'écriture (syntaxe/sémantique)
  - mots-clés
  - fonctions (bibliothèques)
- Complet au sens de Turing
  - régularité
  - puissance
  - complétude
  - confort
  - précision
  - efficacité

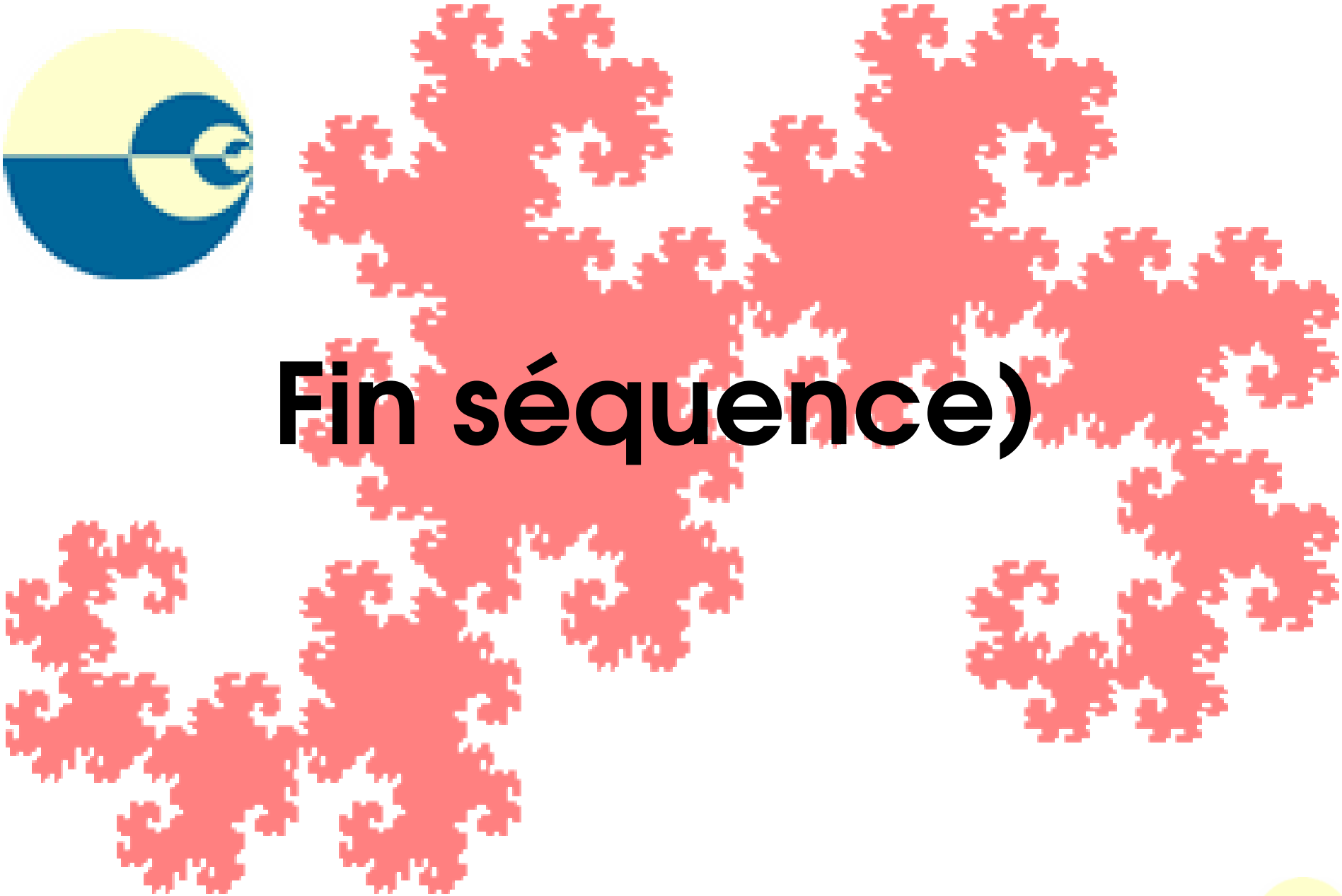
Langages

- Système d'écriture (syntaxe/sémantique)
  - mots-clés
  - fonctions (bibliothèques)
- Complet au sens de Turing
  - régularité
  - puissance
  - complétude
  - confort
  - précision
  - efficacité
- Niche écologique

Langages

- Système d'écriture (syntaxe/sémantique)
  - mots-clés
  - fonctions (bibliothèques)
- Complet au sens de Turing
  - régularité
  - puissance
  - complétude
  - confort
  - précision
  - efficacité
- Niche écologique

La perfection n'est pas quand on ne peut plus rien ajouter mais quand on ne peut plus rien retirer !



Fin séquence)

(Séquence 10.2

Scheme en quelques mots

Scheme

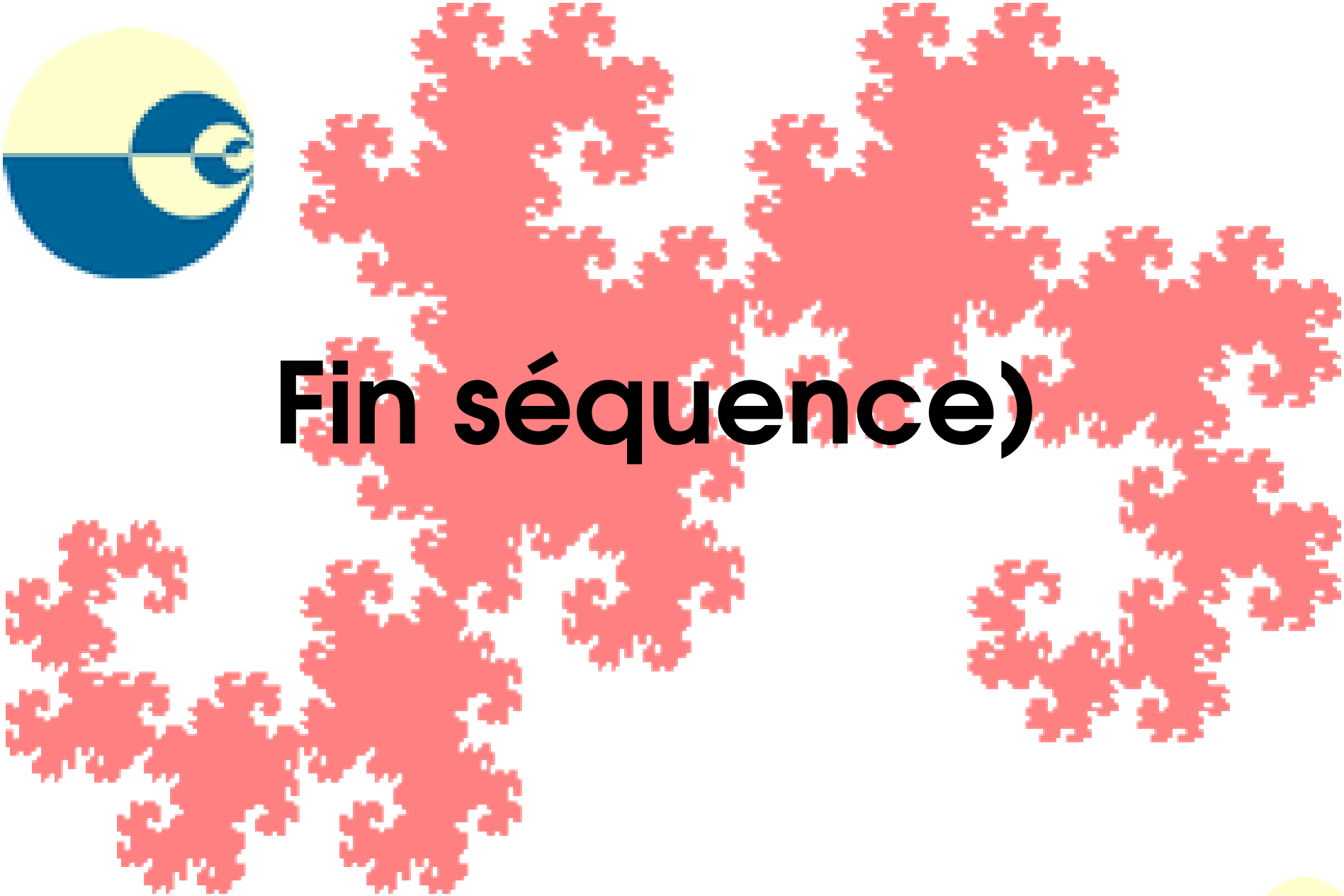
- variable, application
- mots-clés (ou formes spéciales) : `define`, `if`, `let`, `quote`
- mots-clés dérivés : `let*`, `and`, `or`, `cond`
- fonctions : une cinquantaine dans la carte de référence
- composabilité (attention à la place des `define`)
- récursion, structures de données récursives

Le reste de Scheme

Toutefois la moitié de Scheme resterait encore à étudier

- concepts : fonctions anonymes, affectation et données modifiables, tour des nombres, continuations, macros
- données : vecteurs, flux, exceptions
- styles : programmation par objets, par flots, paresseuse, dirigée par les données, par passage de continuation, etc.

Voir les 50 pages du standard IEEE aussi dite R4RS.



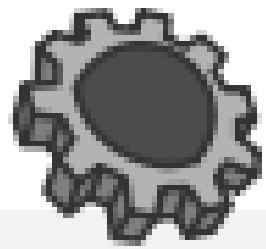
Fin séquence)

(Séquence 10.3

Processus d'évaluation

Évaluation

```
(define (f ...) ...)
(verifier f
   ... => ... )
(define (g ...) ...)
(verifier g
   ... => ... )
```



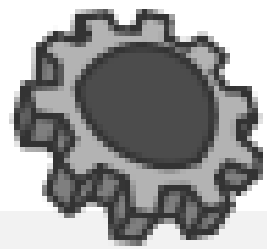
Bouton

```
;;; MrScheme: string -> Valeur
;;; (MrScheme chaîne) calcule la valeur du
;;; programme écrit dans chaîne.
```

Conditionnement

Regroupement en une seule S-expression :

```
(let ()
  (define (f ...) ...)
  (define (g ...) ...)
  (verifier f
     ... => ... )
  (verifier g
     ... => ... ) )
```



Bouton

```
;;; MrScheme: S-expression -> Valeur
;;; (MrScheme sexp) calcule la valeur du
;;; programme représenté par sexp.
```

Auto-évaluation

On nommera cette fonction `valeur`. C'est l'***interprète*** du langage de ce MOOC. Ainsi

```
(valeur '(+ 2 3))  →5
```

Auto-auto-évaluation

La fonction `valeur` est écrite dans le langage même que reconnaît `valeur` :

```
(valeur '(+ 2 3))  →5

(valeur '(let ()
           (define (valeur p) ...)
           (valeur '(+ 2 3)) ))  →5
```

Auto-auto-auto-évaluation

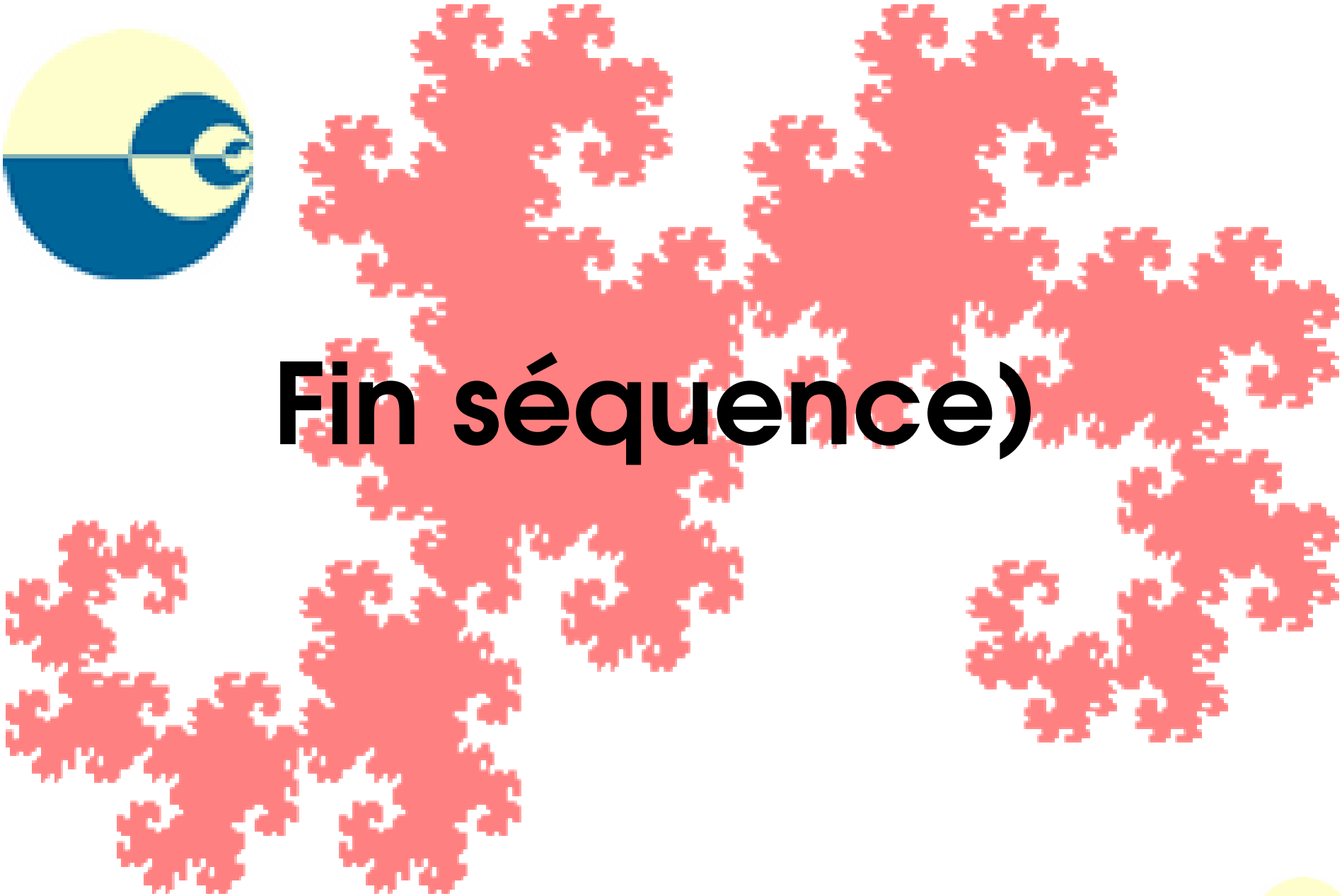
et même :

```
(valeur '(+ 2 3))  →5

(valeur '(let ()
              (define (valeur p) ...)
              (valeur '(+ 2 3)) ))  →5

(valeur
  '(let ()
     (define (valeur p) ...)
     (valeur '(let ()
                (define (valeur p) ...)
                (valeur '(+ 2 3)) )) ))  →5
```

La fonction `valeur`

- la base de tout interprète ou compilateur
- la base de tout metteur au point
- la possibilité d'engendrer des programmes
- une autre façon de comprendre Scheme (nommée `eval`)

Le code correspondant est l'objet de la leçon de cette dernière semaine.



Fin séquence)

(Séquence 10.4

Conclusions

Conclusions sur l'évaluateur

L'évaluateur est écrit dans le Scheme que vous connaissez :

- À un détail près (l'usage d'un vecteur et de `vector-set!` pour l'implantation des fonctions récursives)
- tout le reste n'est que parcours de listes ou d'arbres généraux (programmes, environnement)
- Scheme est le seul langage normalisé dont l'interprète a une taille étudiable.

Points abordés

- L'informatique est une science (mais aussi une technique)
- Bases des langages de programmation
- Principes de programmation (récursion, test, barrière d'abstraction)
- Structures de données (liste, arbre)
- Base d'algorithmique (dichotomie, parcours arborescent)
- Structure d'un évaluateur

Futur

La suite avec le MOOC « Programmation typée »

- typage
- modularité

Mais, si vous êtes maintenant mordu de langages de programmation ou tout simplement mordu de Scheme alors, pour en savoir beaucoup, beaucoup plus, lisez « Principes d'implantation de Scheme et Lisp » de l'auteur de ce MOOC aux éditions Paracamplus en vente dans la librairie « Le Monde en Tique ».