MÉMORISATION

let

MÉMORISER : POUR QUOI FAIRE ?

Reprenons notre programme minimum :

```
(define minimum ; → nombre
        (lambda (l) ; l liste de nombres non vide
                (if (null? (cdr l))
                    (car l)
                    (if  (< (car l) (minimum (cdr l)))
                         (car l)
                         (minimum (cdr l))))))
```

Illustration de l'algorithme vue au cours 1

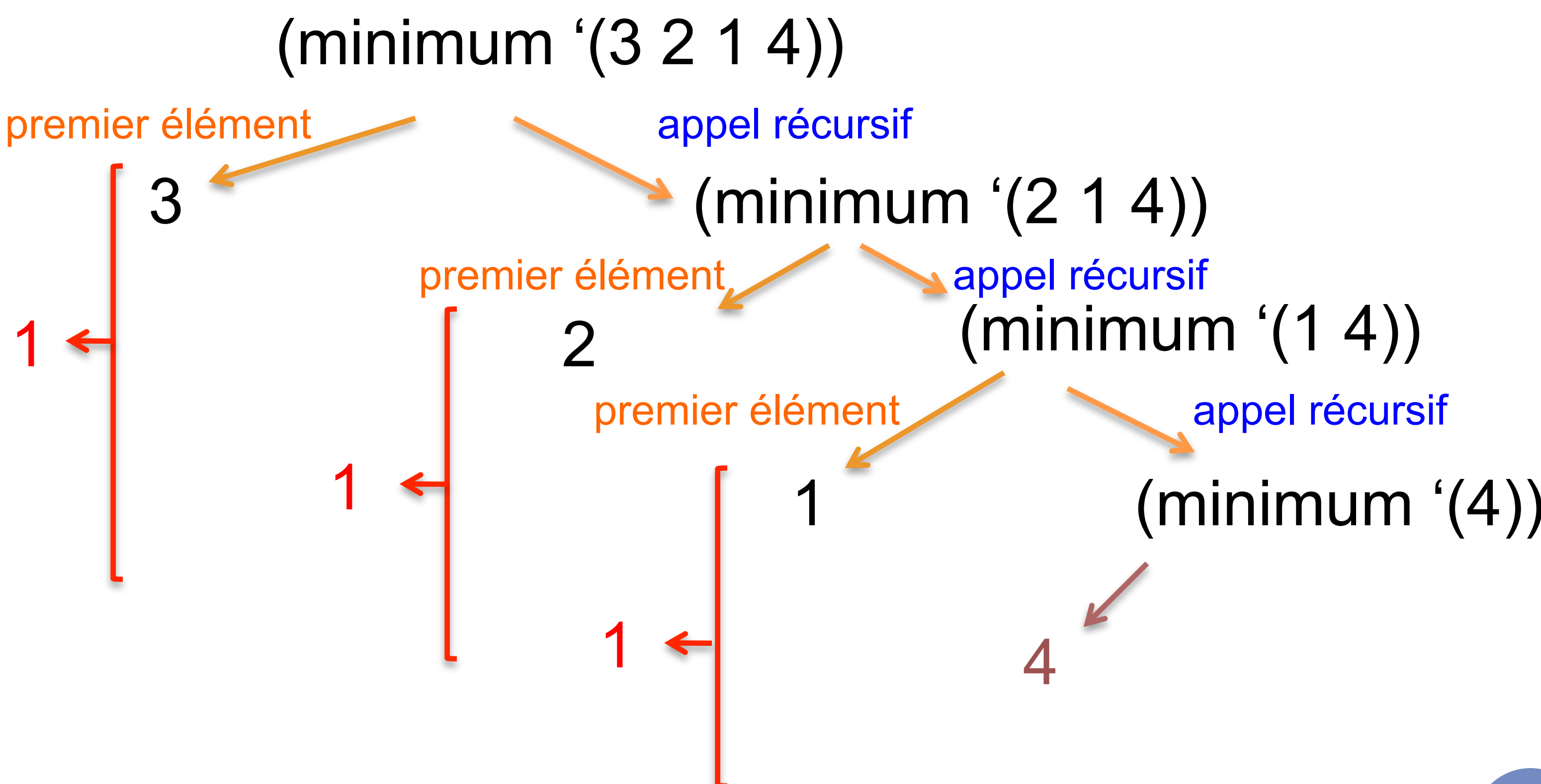


Illustration de l'algorithme vue au cours 1

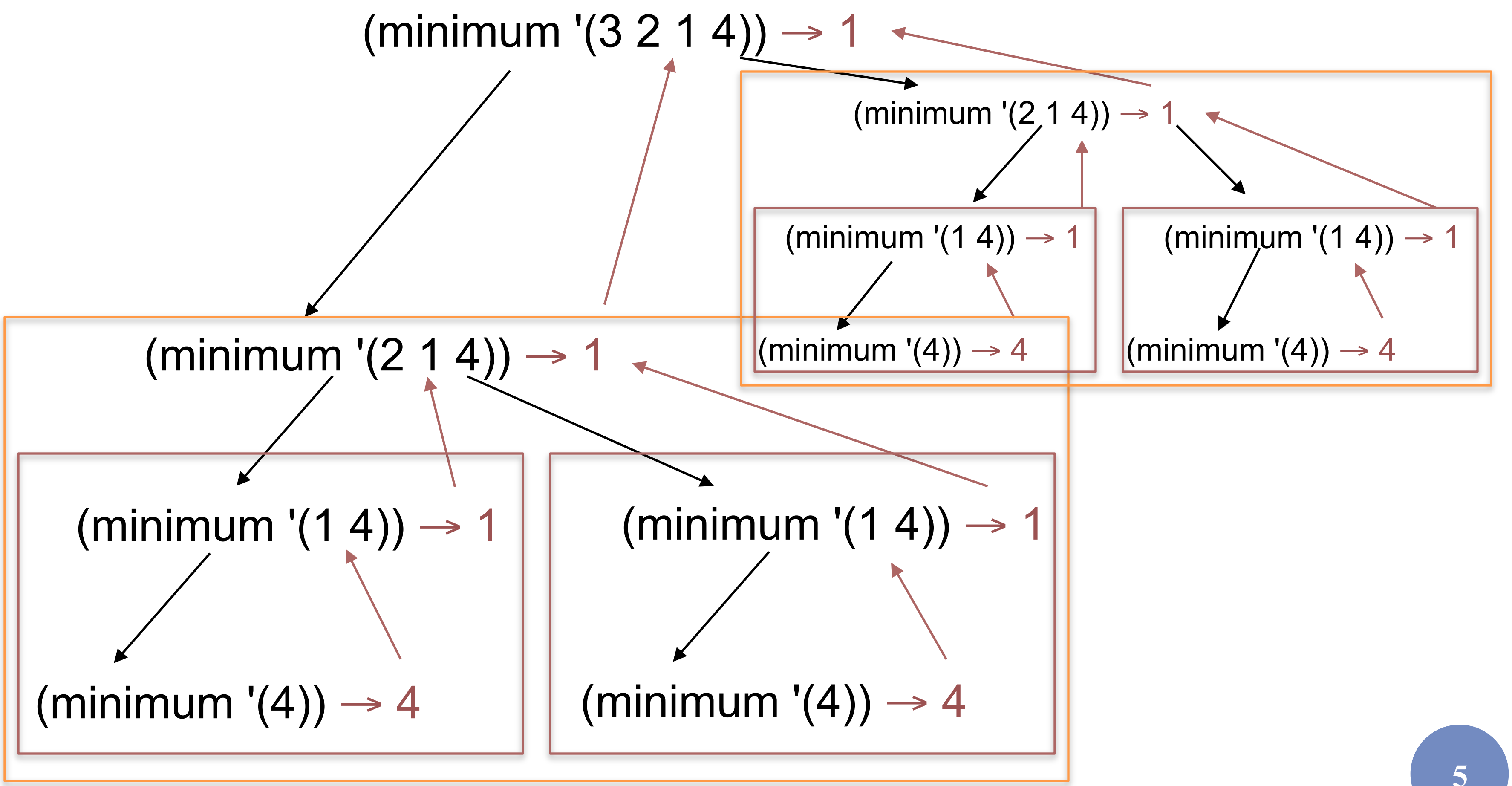
L'illustration correspond à cet algorithme :

```
(define minimum
   (lambda (l)
         (if (null? (cdr l))
            (car l)
         Calcul de  (minimum (cdr l))
         stocké dans min
            (if  (< (car l) min )
               (car l)
               min ))))
```

Notre programme est celui-ci :

```
(define minimum
   (lambda (l)
         (if (null? (cdr l))
                     (car l)
          (if  (<    (car l)
                     (minimum (cdr l)))
             (car l)
            (minimum (cdr l))))))
```

Illustration réelle de notre fonction minimum



Comment mémoriser ?

- On souhaite conserver le résultat du premier appel à *minimum* pour s'en resservir au lieu de provoquer le deuxième appel
- On définit donc un identificateur local (variable locale) grâce à un let

Syntaxe du let

```
(let (
        (ident₁ val₁)
        (ident₂ val₂)
        …
        (identN valN)
     )
  corps
)
```

Fonctionnement du let

- Les *val*$_i$ sont évalués (dans un ordre quelconque) et ces valeurs sont affectées aux *ident*$_i$
- Dans le *corps*, on peut utiliser les *ident*$_i$
- Attention : les *ident*$_i$ ne sont pas définis à l'extérieur du corps

APPLICATION AU PROGRAMME MINIMUM

```
(define minimum ; → nombre
   (lambda (l) ; l liste de nombres   non vide
         (if      (null? (cdr l))
                  (car l)
                  (let ((m (minimum (cdr l))))
                          (if      (< (car l) m)
                                   (car l)
                                   m)))))
```

FONCTIONNEMENT DU NOUVEAU PROGRAMME

(minimum '(3 2 1 4)) → 1

m = 1

(minimum '(2 1 4)) → 1

m = 1

(minimum '(1 4)) → 1

m = 4

(minimum '(4)) → 4

AUTRE EXEMPLE

- Écrire une fonction qui calcule $$\frac{3\sqrt{\frac{x^2}{2}}+1}{\sqrt{\frac{x^2}{2}}}$$

```
(define calcule ; → nombre
        (lambda (x) ; x nombre non nul
                (/ (+ (* 3 (sqrt (/ (sqr x) 2))) 1)
                   (sqrt (/ (sqr x) 2))))))
```

AMÉLIORATION

```
(define calcule ; → nombre
          (lambda (x) ; x nombre non nul
                 (let ((c (sqrt (/ (sqr x) 2))))
                        (/ (+ (* 3 c) 1) c))))
```

$$\frac{3\sqrt{\frac{x^2}{2}}+1}{\sqrt{\frac{x^2}{2}}}$$

- L'utilisation du let permet ici une simplification d'écriture, mais n'améliore pas significativement la complexité de l'algorithme
- Dans le cas d'un appel récursif comme dans le programme *minimum*, l'utilisation du let est primordiale pour la complexité

Quand les identificateurs sont liés

```
(define toto ; → nombre
   (lambda (x) ; x nombre
        (let (    (a (sqr x))
                  (b (+ (* 2 a) 1)))
           (if    (< a 80)
                  (* 3 (+ a 1))
                  (sqrt b)))))
```

→ erreur car les affectations de a et b ont lieu dans un ordre quelconque

LET*

```
(define toto ; → nombre
  (lambda (x) ; x nombre
     (let* ((a (sqr x))
            (b (+ (* 2 a) 1)))
        (if       (< a 80)
                  (* 3 (+ a 1))
                  (sqrt b)))))
```

Les évaluations des identificateurs se font séquentiellement dans un let*

Tris

Tri par fusion

Quel est le problème à résoudre ?

- Soit une liste de nombres : '(5 2 14 1 6)
- On souhaite la trier : '(1 2 5 6 14)

Algorithmes de tri

- Tris par sélection du minimum
  - tri-minimum (TP)
  - tri-bulles (TD)
- Tri par insertion (TD)
- Tri par fusion

- Tri rapide
- Tri par tas
- …

Principes des tris par sélection

- On cherche le minimum de la liste,
  puis on recommence avec le reste de la liste

- Tri du minimum
  - fonction minimum
  - fonction enlève

- Tri bulles
  - fonction bulle, qui sélectionne le minimum
    et l'enlève de la liste en un seul passage

PRINCIPE DU TRI PAR INSERTION

- Principe : on trie récursivement le cdr de la liste, puis on y insère le car
- Exemple :

$$\begin{tikzcd}
\text{'(5 2 14 1 6)} \arrow[r, "\text{cdr}"] \arrow[d, "\text{tri-insertion}"'] & \text{'(2 14 1 6)} \arrow[d, "\text{tri-insertion}"] \\
\text{'(1 2 5 6 14)} & \text{'(1 2 6 14)} \arrow[l, "\text{insérer 5}"']
\end{tikzcd}$$

TRI PAR FUSION : L'APPROCHE « DIVISER POUR RÉGNER »

- Structure récursive :
  pour résoudre un problème donné, l'algorithme s'appelle lui-même récursivement une ou plusieurs fois sur des sous-problèmes très similaires

- Le paradigme « diviser pour régner » donne lieu à trois étapes à chaque niveau de récursivité : diviser, régner, combiner

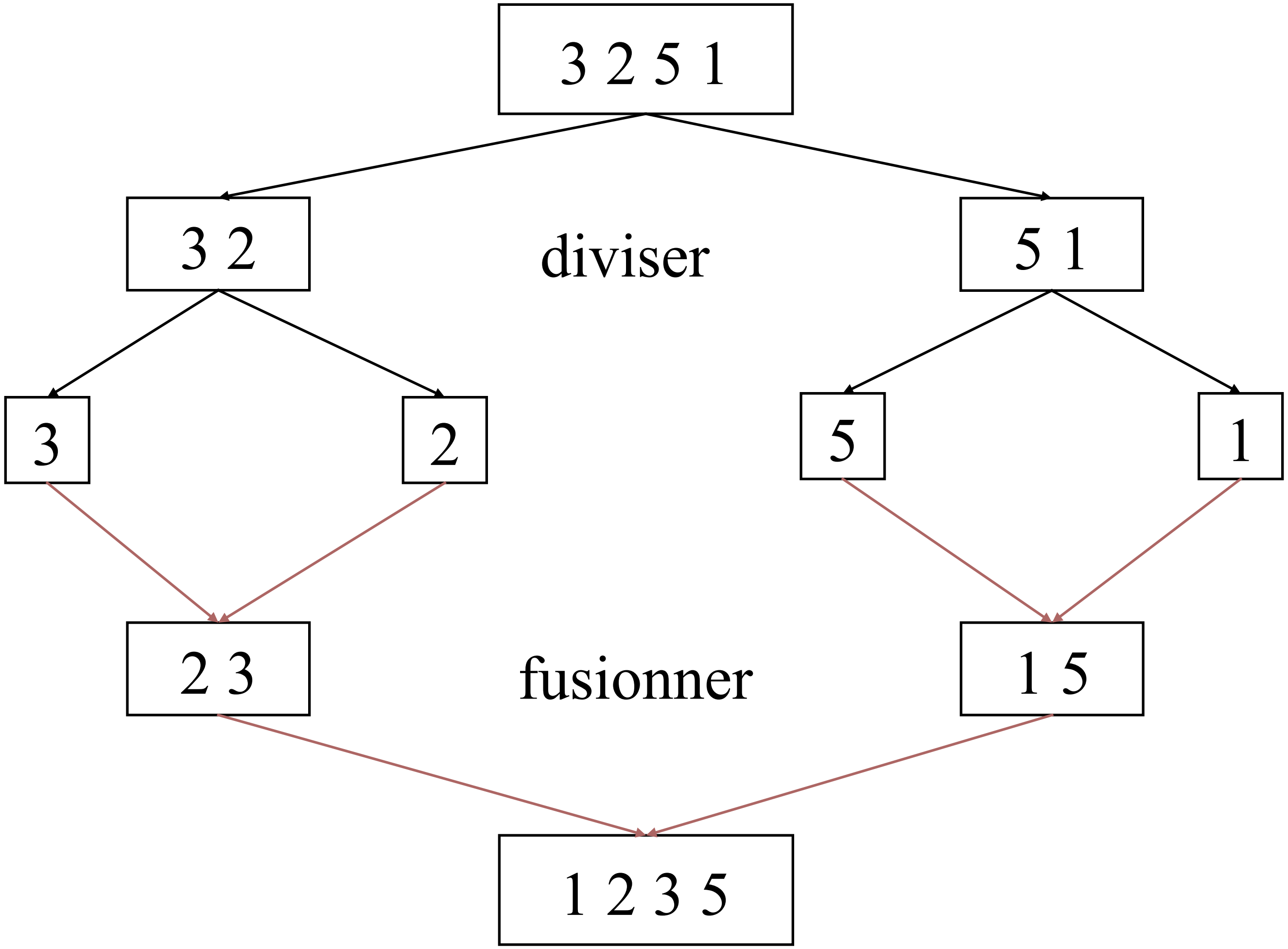
Diviser pour régner : 3 étapes

- Diviser le problème en un certain nombre de sous-problèmes
- Régner sur les sous-problèmes en les résolvant récursivement
  Si la taille d'un sous-problème est assez réduite, on peut le résoudre directement
- Combiner les solutions des sous-problèmes en une solution complète pour le problème initial

TRI PAR FUSION : LE PRINCIPE

- Diviser : diviser la liste de n éléments à trier en deux sous-listes de n/2 éléments
- Régner : trier les deux sous-listes récursivement à l'aide du tri par fusion
- Combiner : fusionner les deux sous-listes triées pour produire la réponse triée

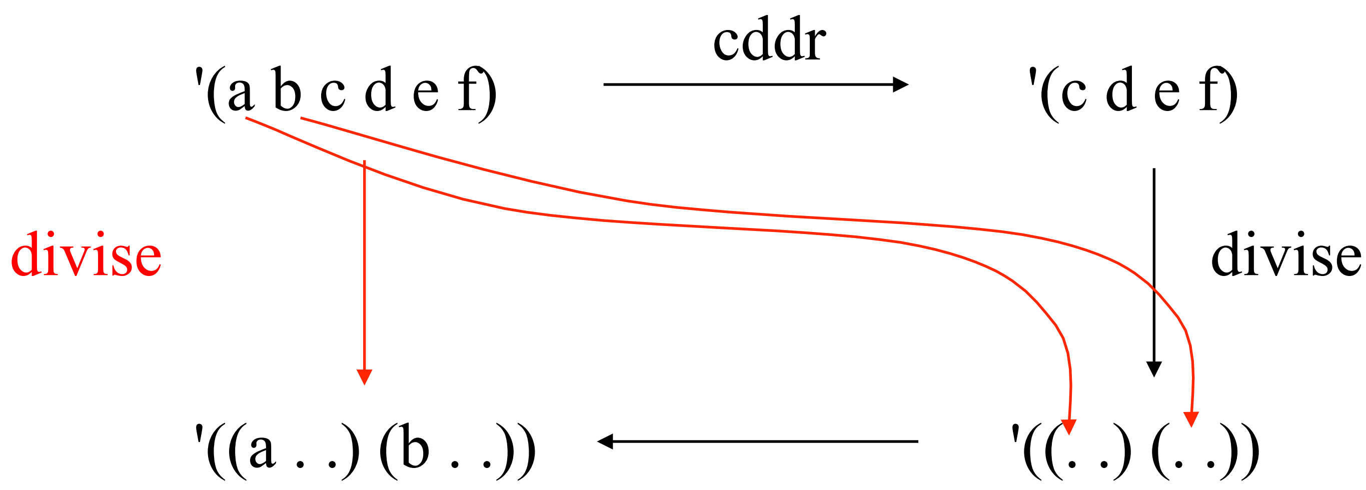
Un exemple



TRI PAR FUSION : LA MÉTHODE

$$
\begin{tikzcd}
\text{'(3 2 5 1)} \arrow[r, "\text{divise}"] \arrow[d, "\text{tri-fusion}"'] & \text{'((3 2) (5 1))} \arrow[d, shift right, "\text{tri-fusion}"'] \arrow[d, shift left, "\text{tri-fusion}"] \\
\text{'(1 2 3 5)} & \text{(2 3) (1 5)} \arrow[l, "\text{fusion}"']
\end{tikzcd}
$$

DIVISER LA LISTE EN DEUX SOUS-LISTES : LA MÉTHODE



DIVISER LA LISTE EN DEUX SOUS-LISTES : LA FONCTION

```
(define divise ; → liste de deux listes
  (lambda (l) ; l liste
    (cond     ((null? l) '(() ()))
              ((null? (cdr l)) (list l '()))
              (else (let ((r (divise (cddr l))))
                 (list (cons (car l) (car r))
                       (cons (cadr l) (cadr r)))))))))
```

Diviser la liste en deux sous-listes : illustration de la fonction

(divise '(3 2 5 1))

r1 → (divise '(5 1)) car → 3 cadr →2
(list (cons 3 (car r1)) (cons 2 (cadr r1))))

r2 → (divise '()) car → 5 cadr →1
(list (cons 5 (car r2)) (cons 1 (cadr r2))))

r2 → '(() ())

r1 → '((5) (1))

'((3 5) (2 1))

FUSIONNER DEUX LISTES TRIÉES : LA MÉTHODE

$$
\begin{tikzcd}
\text{'(2 5 7) '(1 3 4 8)} \arrow[r, "1<2"] \arrow[d, "\text{fusion}"'] & \text{'(2 5 7) '(3 4 8)} \arrow[d, "\text{fusion}"] \\
\text{'(1 2 3 4 5 7 8)} & \text{'(2 3 4 5 7 8)} \arrow[l, "\text{cons 1}"']
\end{tikzcd}
$$

FUSIONNER DEUX LISTES TRIÉES : LA FONCTION

```
(define fusion ; → liste de nb triée
  (lambda (l1 l2) ; listes de nb triées
    (cond ((null? l1) l2)
          ((null? l2) l1)
          ((< (car l1) (car l2))
              (cons (car l1) (fusion (cdr l1) l2)))
          (else
              (cons (car l2) (fusion l1 (cdr l2)))))))
```

FUSIONNER DEUX LISTES TRIÉES : ILLUSTRATION DE LA FONCTION

(fusion '(2 5 7) '(1 3 4 8))

$2 > 1$		
(cons 1 (fusion '(2 5 7) '(3 4 8)))		'(1 2 3 4 5 7 8)
$2 < 3$		
(cons 2 (fusion '(5 7) '(3 4 8)))		'(2 3 4 5 7 8)
$5 > 3$		
(cons 3 (fusion '(5 7) '(4 8)))		'(3 4 5 7 8)
$5 > 4$		
(cons 4 (fusion '(5 7) '(8)))		'(4 5 7 8)
$5 < 8$		
(cons 5 (fusion '(7) '(8)))		'(5 7 8)
$7 < 8$		
(cons 7 (fusion '() '(8)))		'(7 8)
'(8)		

Tri par fusion : la fonction

```
(define tri-fusion ; → liste de nb triée
  (lambda (l) ; liste de nb non vide
    (if (null? (cdr l))
        l
        (let ((r (divise l)))
          (fusion       (tri-fusion (car r))
                        (tri-fusion (cadr r)))))))
```

TRI PAR FUSION : ILLUSTRATION

(tri-fusion '(7 4 9 1))

r1 → (divise '(7 4 9 1)) → ((7 9) (4 1))

(fusion

(tri-fusion '(7 9))

r2 → (divise '(7 9)) → '((7) (9))

(fusion (tri-fusion '(7)) → '(7)

(tri-fusion '(9))) → '(9)

→ '(7 9)

(tri-fusion '(4 1)))

→ '(1 4 7 9)

r3 → (divise '(4 1)) → '((4) (1))

(fusion (tri-fusion '(4)) → '(4)

(tri-fusion '(1))) → '(1)

→ '(1 4)

Calculs en remontant ou en descendant

CALCULS EN REMONTANT OU EN DESCENDANT

- Jusqu'à présent, nous avons toujours effectué les calculs en remontant des appels récursifs
- Exemple : retour sur la fonction factorielle

```
(define factorielle ; → entier positif
        (lambda (n) ; n entier positif
                (if     (= n 0)
                        1
                        (* n (factorielle (- n 1))))))
```

FONCTION FACTORIELLE : ILLUSTRATION

(factorielle 3)

On remonte pour faire les calculs

(* 3 (factorielle 2)) 6

(* 2 (factorielle 1)) 2

(* 1 (factorielle 0)) 1

1

INTRODUIRE UN PARAMÈTRE SUPPLÉMENTAIRE POUR EFFECTUER LES CALCULS EN DESCENDANT

```
(define factorielle-compteur ; → entier positif
  (lambda (n) ; n entier positif
      (fact n 1)))

; effectue le calcul de factorielle(n) en utilisant un paramètre
  supplémentaire res dans lequel on effectue le calcul
(define fact ; → entier positif
 (lambda (n res) ; entiers positifs
  (if (= n 0)
     res
     (fact (- n 1) (* res n)))))
```

Fonction factorielle-compteur : illustration

(factorielle-compteur 3)

(fact 3 1)

(fact 2 (* 1 3))

Les calculs effectués
en descendant
sont stockés dans res

(fact 1 (* 3 2))

(fact 0 (* 6 1))

6

On renvoie directement
le résultat contenu dans res

M. Lefevre - N. Guin - F. Zara

REMARQUES

- La fonction factorielle-compteur est celle qui répond à la spécification.
  Il est indispensable d'écrire une fonction qui répond à la spécification, même si elle ne fait rien d'autre que d'appeler la fonction fact.
  L'utilisateur n'a pas à savoir que nous utilisons un deuxième argument.

- La fonction fact est celle qui fait effectivement tout le travail.

QUEL INTÉRÊT ?

- Dans la fonction fact, on a une récursivité terminale. Avec certains langages de programmation et certains compilateurs, cette récursivité terminale est « dérécursifiée » afin d'améliorer l'efficacité du programme.
- On se rapproche en effet d'une solution itérative :

```
res ← 1
TantQue n>0 Faire
   res ← res*n
   n ← n-1
FinTantQue
Afficher res
```