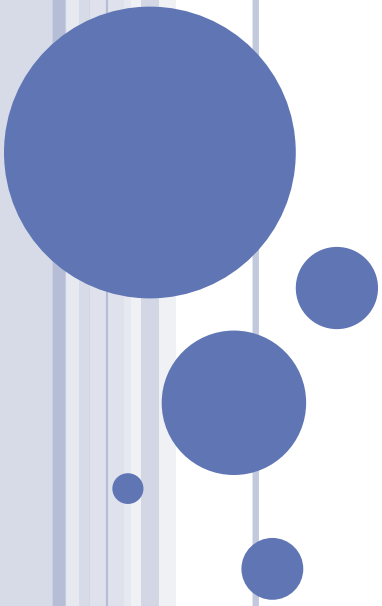


Mémorisation

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

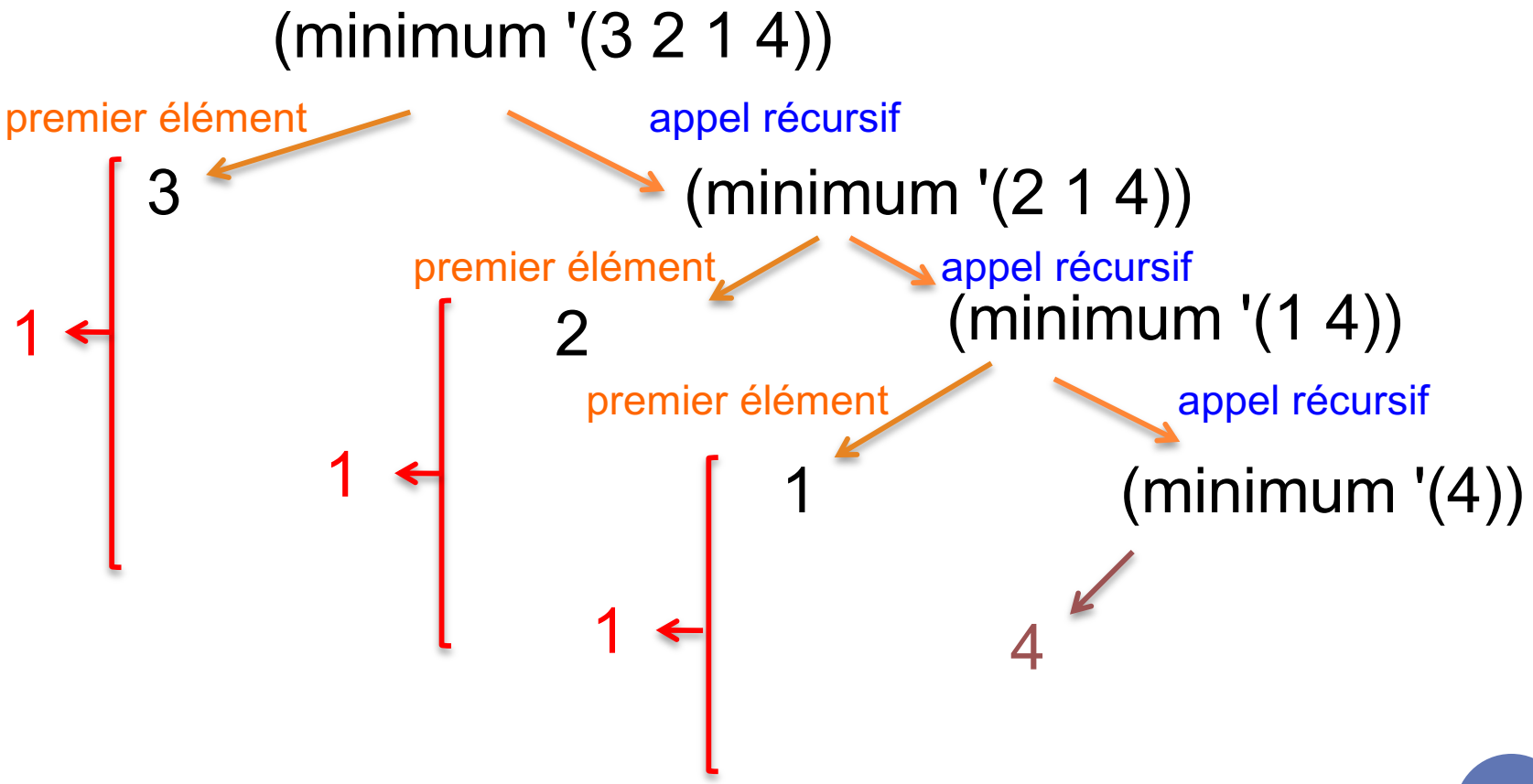


ILLUSTRATION DE L'ALGORITHME VUE AU COURS 1

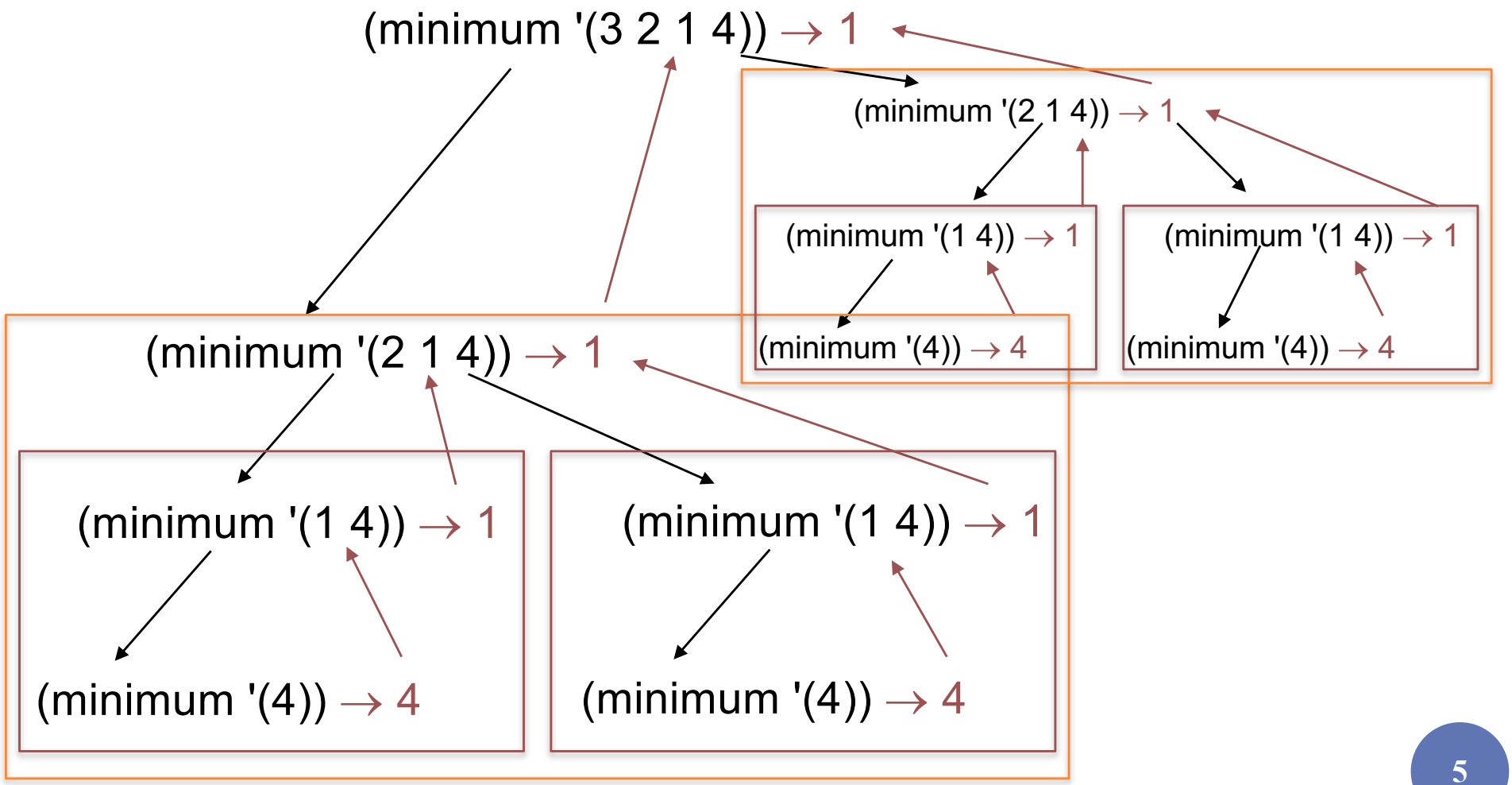
L'illustration correspond à cet algorithme :

```
(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
)
```

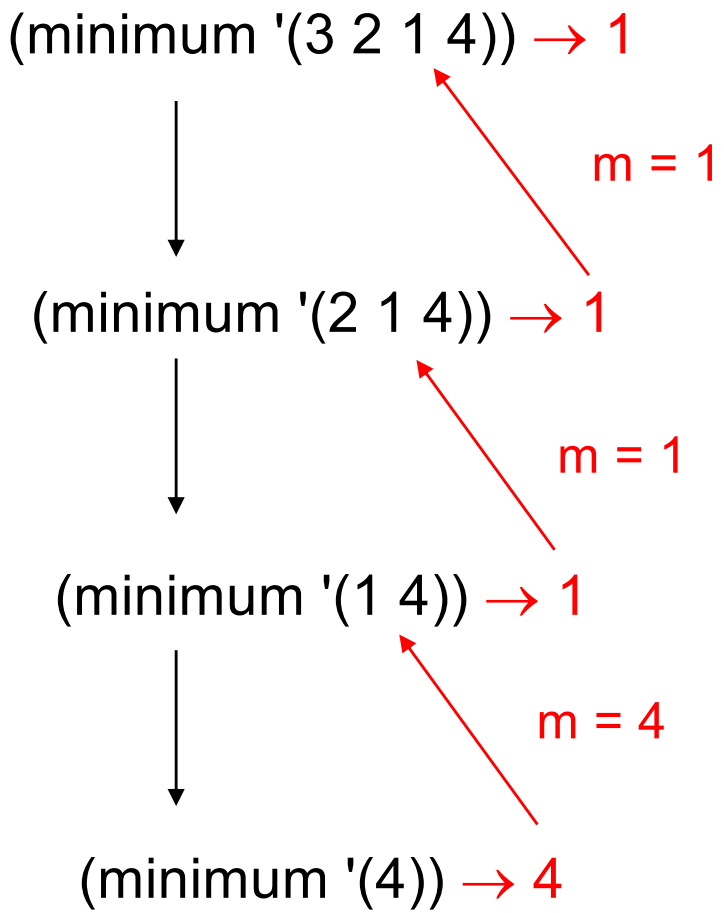
FONCTIONNEMENT DU LET

- Les val_i sont évaluées (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

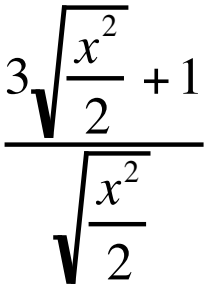
```
(define minimum ; → nombre
   (lambda (l) ; l liste de nombres  non vide
         (if       (null? (cdr l))
                   (car l)
                   (let ((m (minimum (cdr l))))
                         (if       (< (car l) m)
                                   (car l)
                     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

FONCTIONNEMENT DU NOUVEAU PROGRAMME



AUTRE EXEMPLE

- Écrire une fonction qui calcule



```
(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

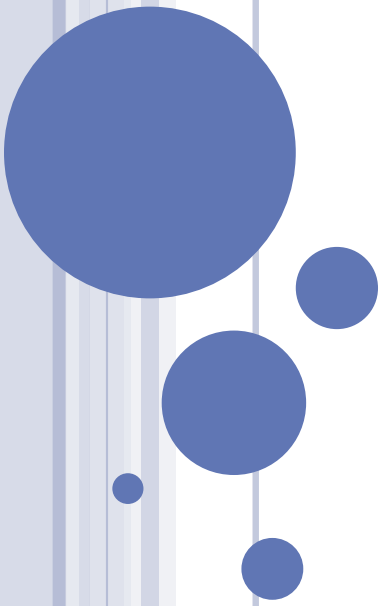
```
(define toto ; → nombre
   (lambda (x) ; x nombre
         (let (    (a (sqr x))
                   (b (+ (* 2 a) 1)))
              (if    (< a 80)
                   (* 3 (+ a 1))
                   (sqrt b)))))
```

→ 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 (TDTP)
- Tri par insertion (TDTP)
- Tri par fusion (CM)

- Tri rapide
- Tri par tas
- …

PRINCIPES DES TRIS PAR SÉLECTION (EN TDTP)

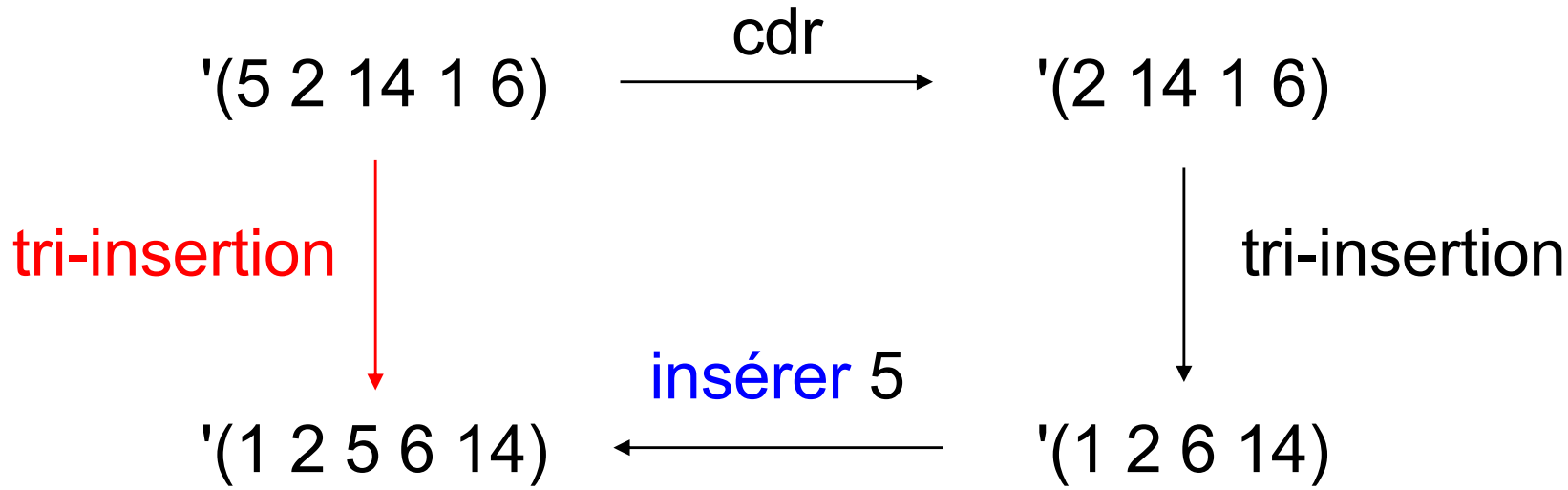
- On cherche le minimum de la liste,
  puis on recommence avec le reste de la liste

- 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 (en TDTP)

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



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

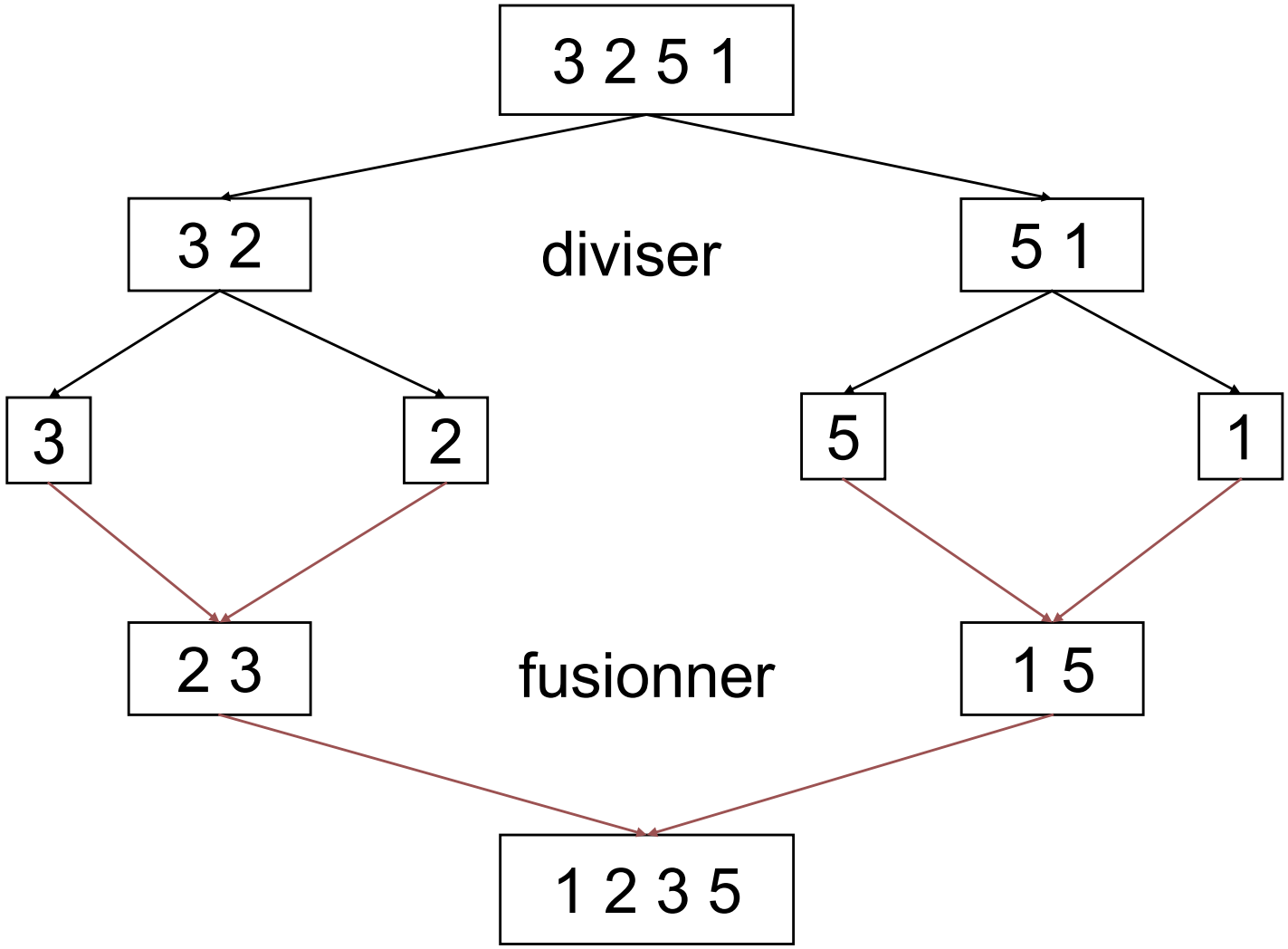
- Régner sur les sous-problèmes en les résolvant récursivement
  Si la taille d'un sous-problème est assez réduite, on peut le résoudre directement

- Combiner les solutions des sous-problèmes en une solution complète pour le problème initial

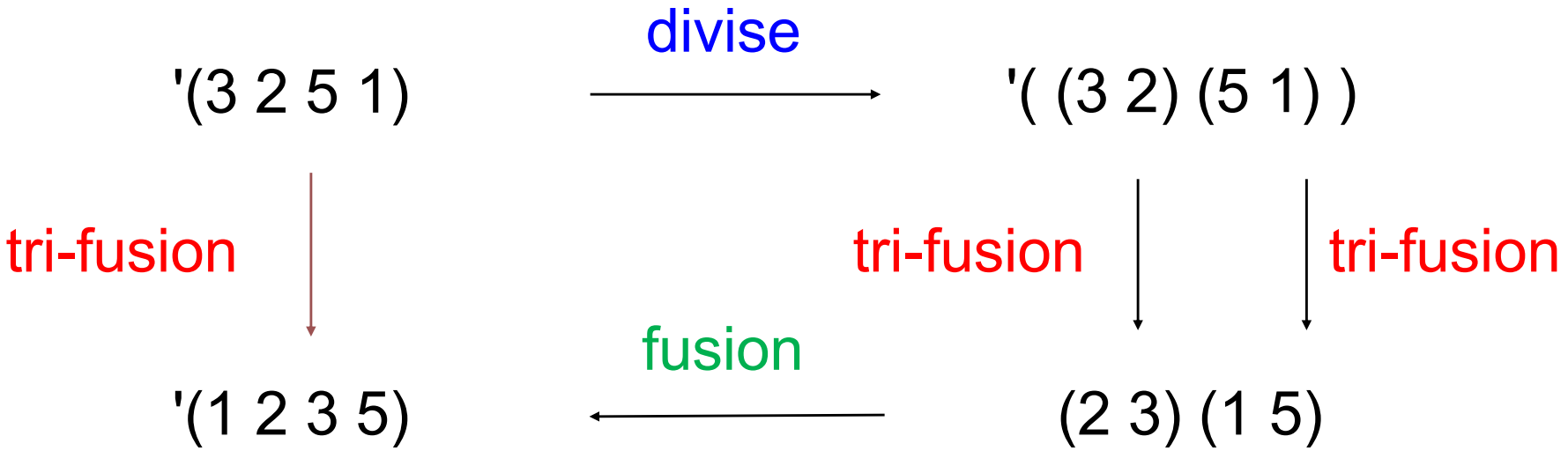
TRI PAR FUSION : LE PRINCIPE

- 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

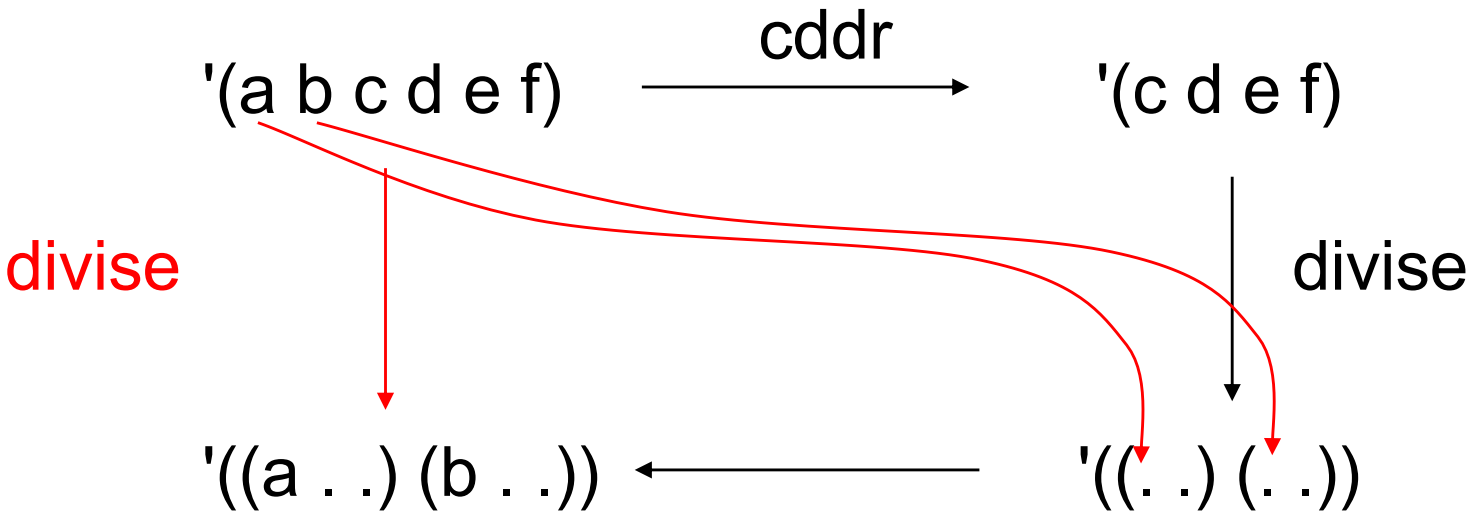


TRI PAR FUSION : LA MÉTHODE



→ trois fonctions à définir : divise, tri-fusion, fusion

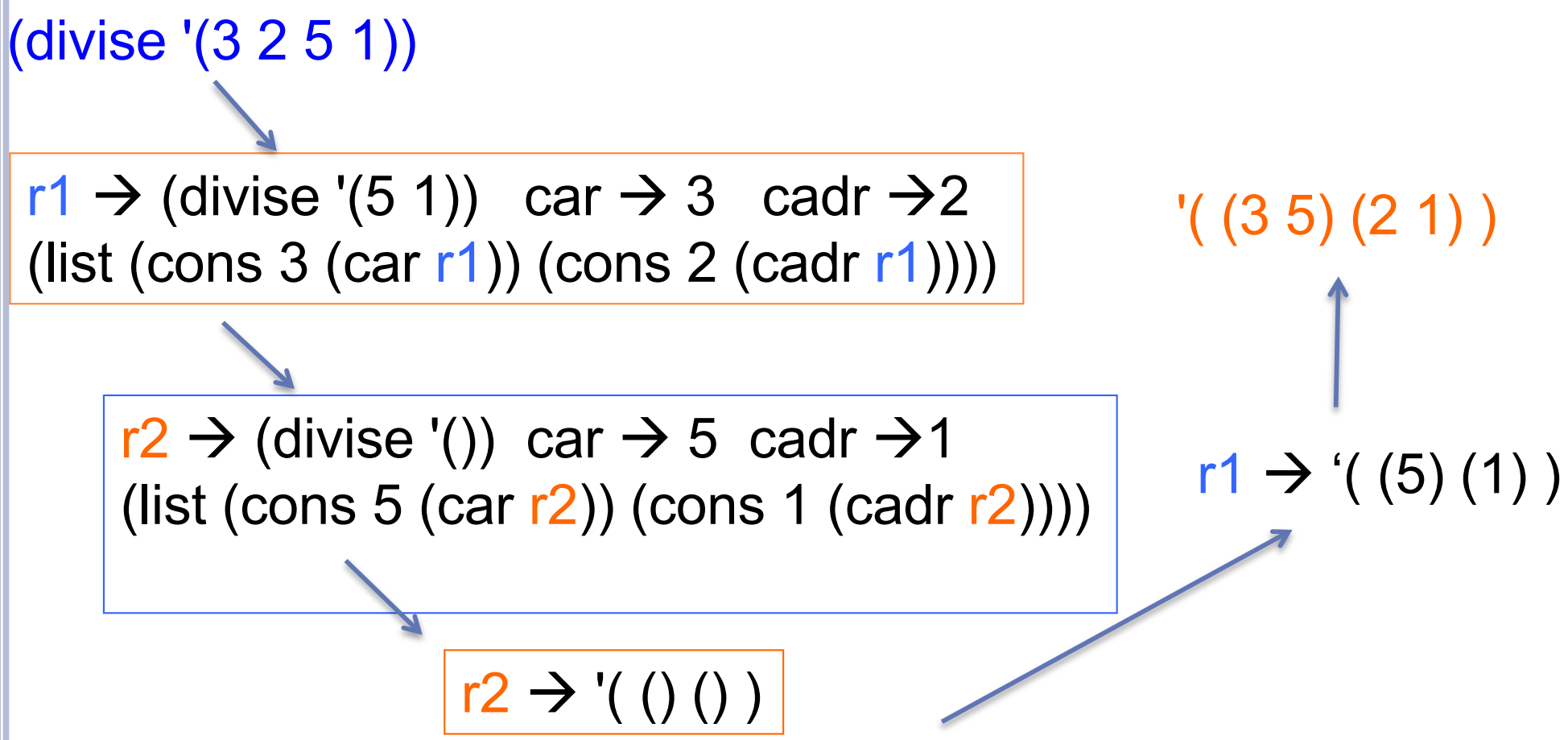
DIVISER LA LISTE EN DEUX SOUS-LISTES : LA MÉTHODE



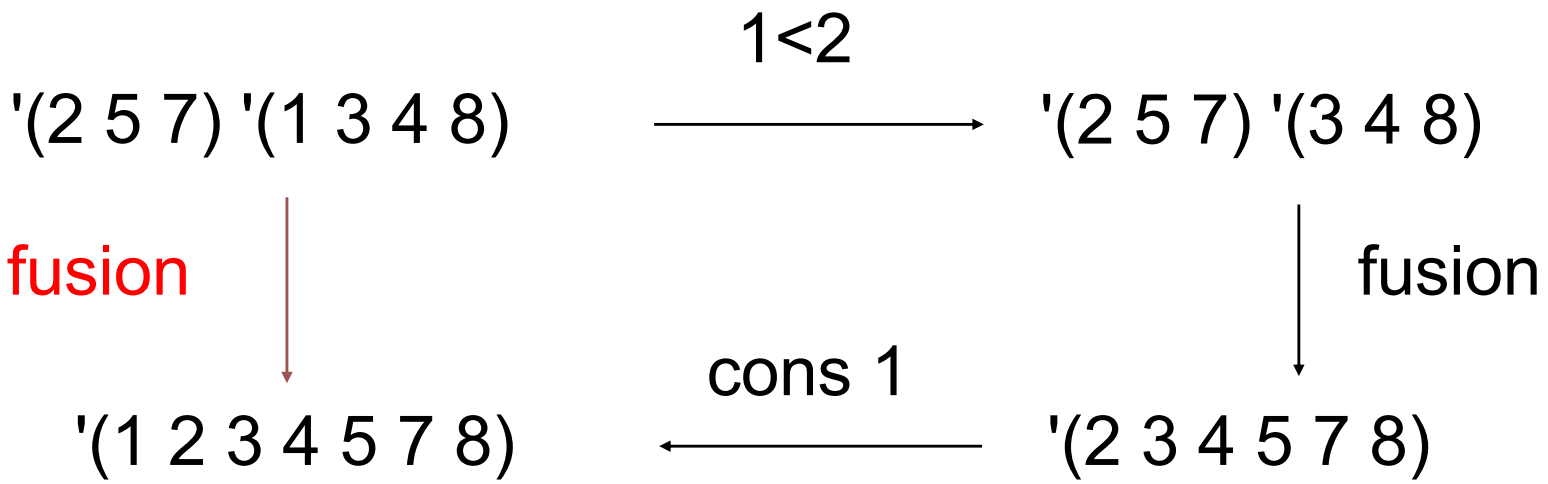
DIVISER LA LISTE EN DEUX SOUS-LISTES : LA FONCTION

```
(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



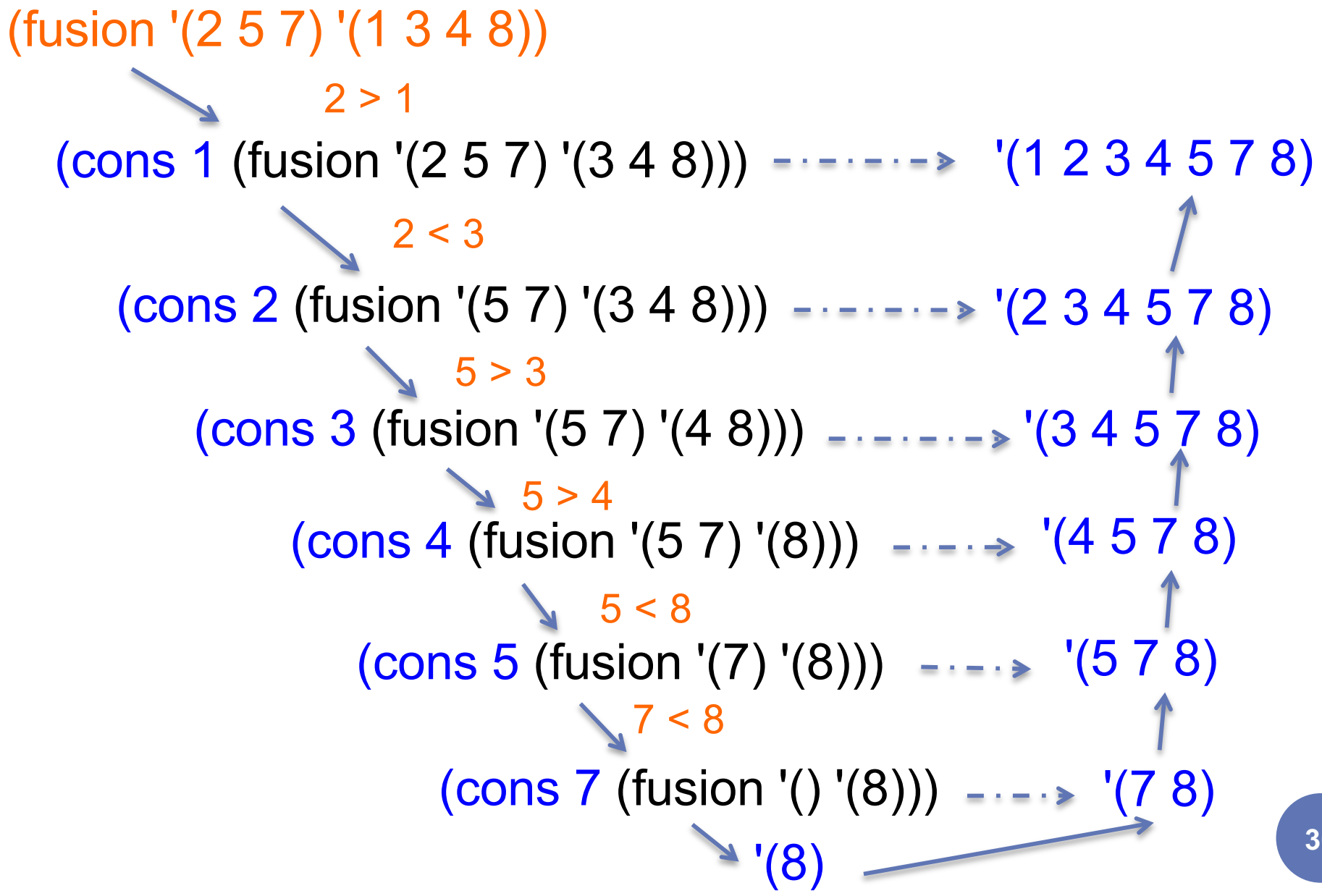
FUSIONNER DEUX LISTES TRIÉES : LA MÉTHODE



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



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))) ; r = ( (…) (…) )
          (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)

  (tri-fusion '(4 1)))

'(7 9)

'(1 4 7 9)

    r3 → (divise '(4 1)) ⟶ '((4) (1))

    (fusion (tri-fusion '(4)) ⟶ '(4)

            (tri-fusion '(1))) ⟶ '(1)

'(1 4)

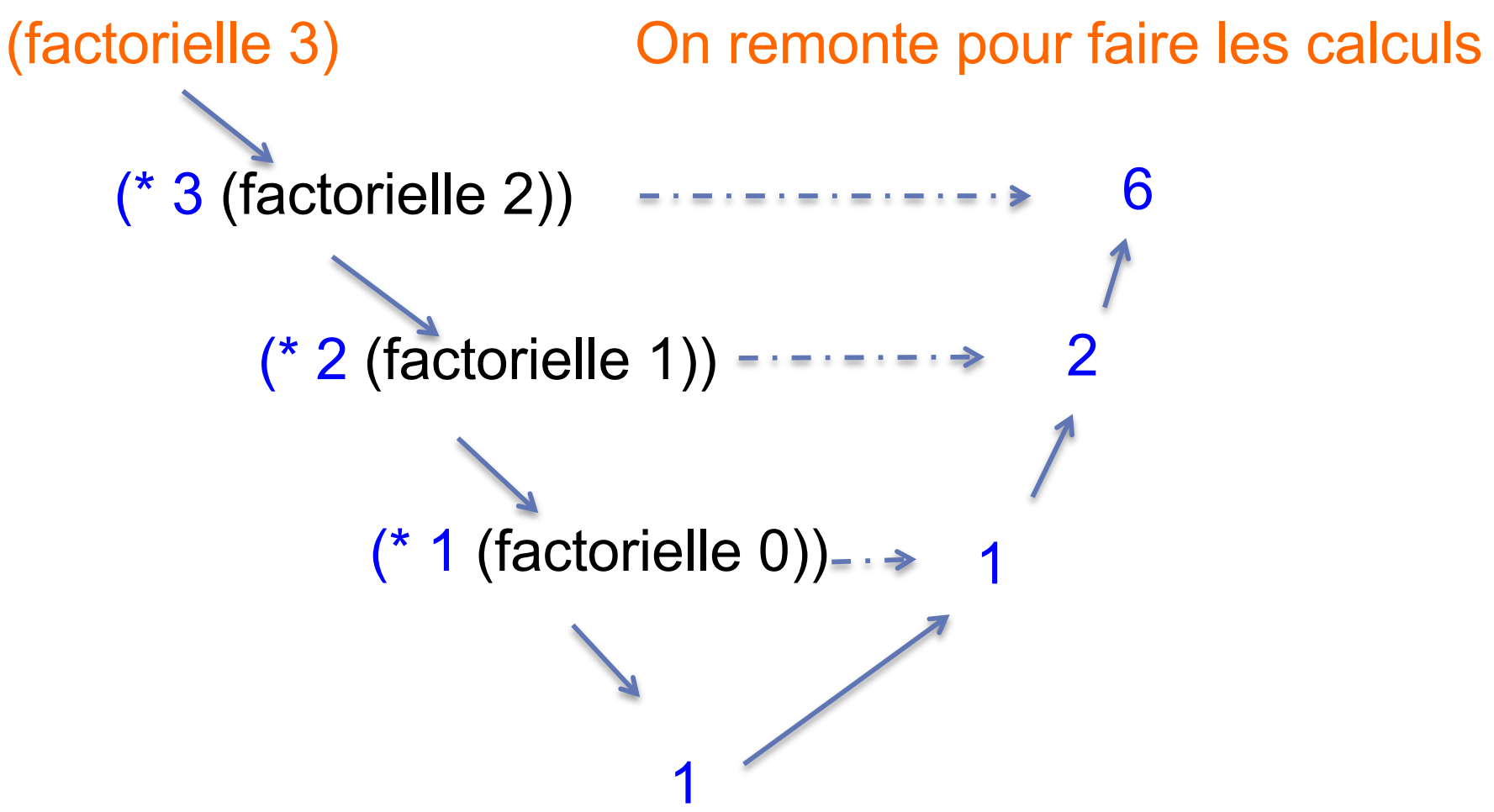
Calculs en remontant ou en descendant

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



EFFECTUER LES CALCULS EN DESCENDANT

(fact 3 1) ; n = 3, res = 1 au départ

(fact 2 (* 1 3)) ; n = 2, res = 3

Les calculs effectués
en descendant
sont stockés dans res

(fact 1 (* 3 2)) ; n = 1, res = 6

(fact 0 (* 6 1)) ; n = 0, res = 6

6

On renvoie directement
le résultat contenu dans res

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)))))
```

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 paramètre.
- La fonction fact est celle qui fait effectivement tout le travail.
- On se rapproche d'une solution itérative :

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

QUEL INTÉRÊT ?

- Simplifier l'écriture des fonctions qui renvoient une liste de 2 résultats ou plus, en introduisant autant de paramètres que de résultats.
- Exemple de la fonction divise :

```
(define divise ; -> liste de 2 sous-listes de taille similaires
  (lambda (l) ; liste d'éléments
    (div l '() '())))  ; ici on initialise 2 listes vides

(define div ;     -> liste de 2 sous-listes
  (lambda (l l1 l2) ; 3 listes : l est la donnée,
                        ; l1 et l2 les résultats qui se construisent
    (cond
      ((null? l) (list l1 l2)) ; cas d'arrêt : on renvoie la liste de 2 résultats
      ((null? (cdr l)) (list (cons (car l) l1) l2))
      (else (div (cddr l) ; sinon on fait un appel résursif en construisant l1 et l2
                 (cons (car l) l1)
                 (cons (cadr l) l2))))))
```

- Remarque : les listes se construisent à l'envers.
  Ici ce n'est pas gênant. Sinon on peut utiliser la fonction reverse dans le cas d'arrêt.