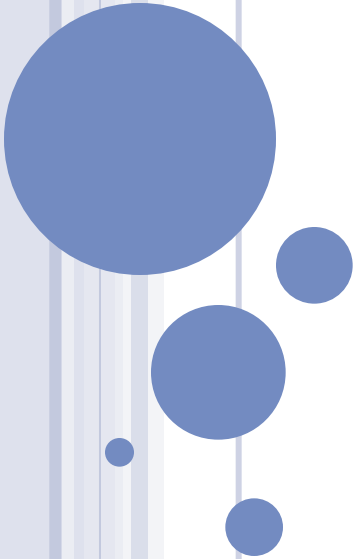


Mémorisation

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

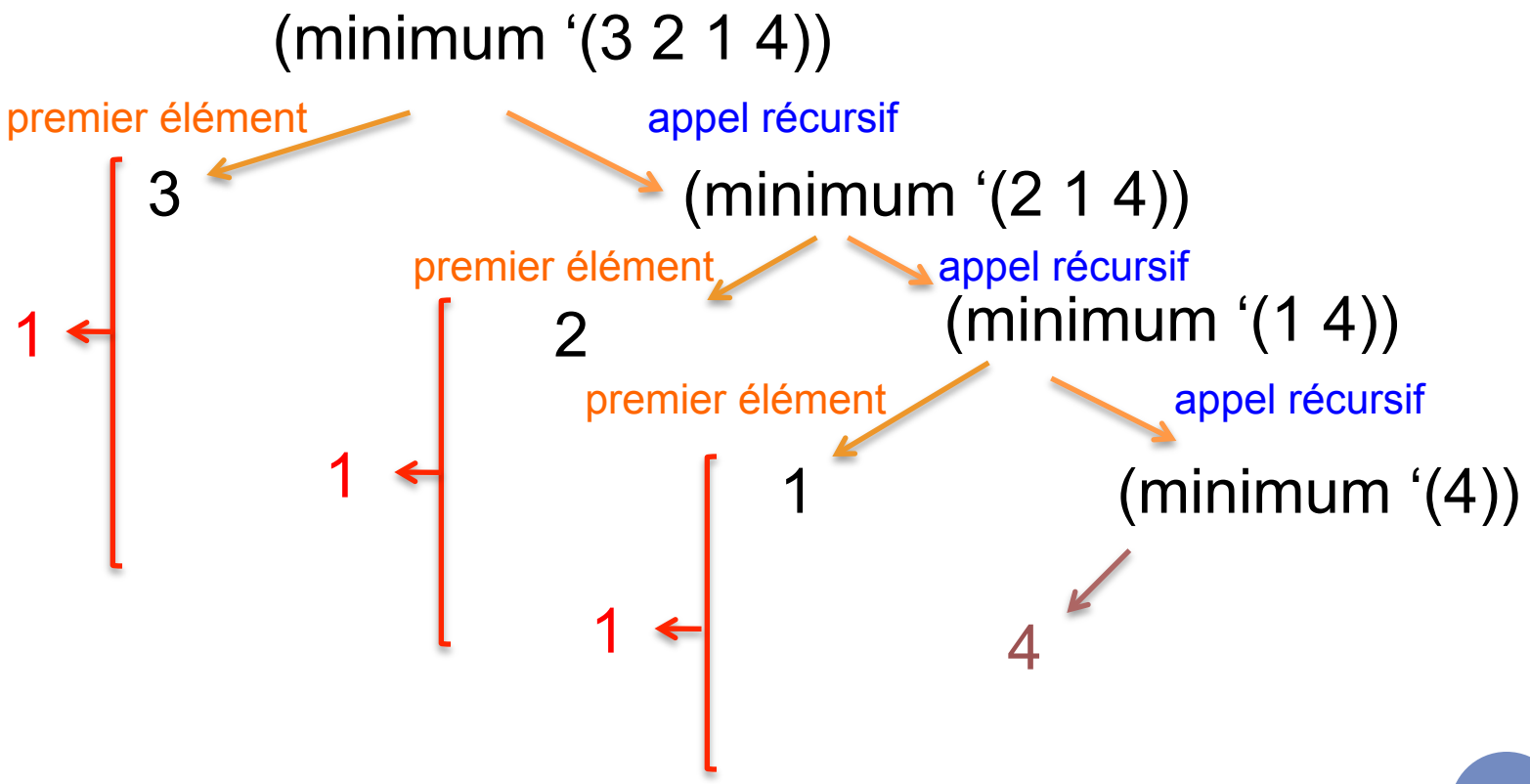


ILLUSTRATION DE L'ALGORITHME VUE AU COURS 1

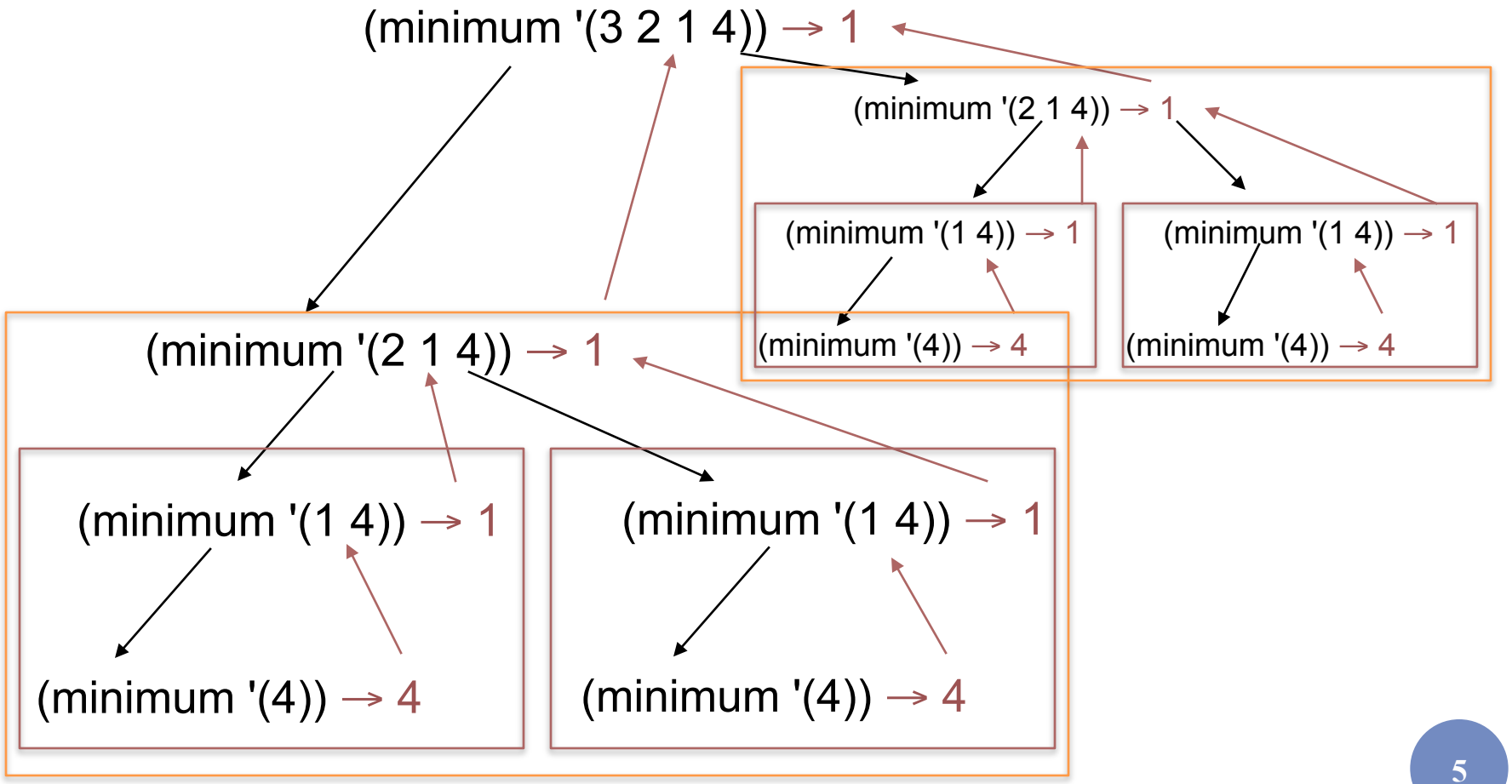
L'illustration correspond à cet algorithme :

```
(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 ((ident$_1$ val$_1$)
(ident$_2$ val$_2$)
…
(ident$_N$ val$_N$))
corps)

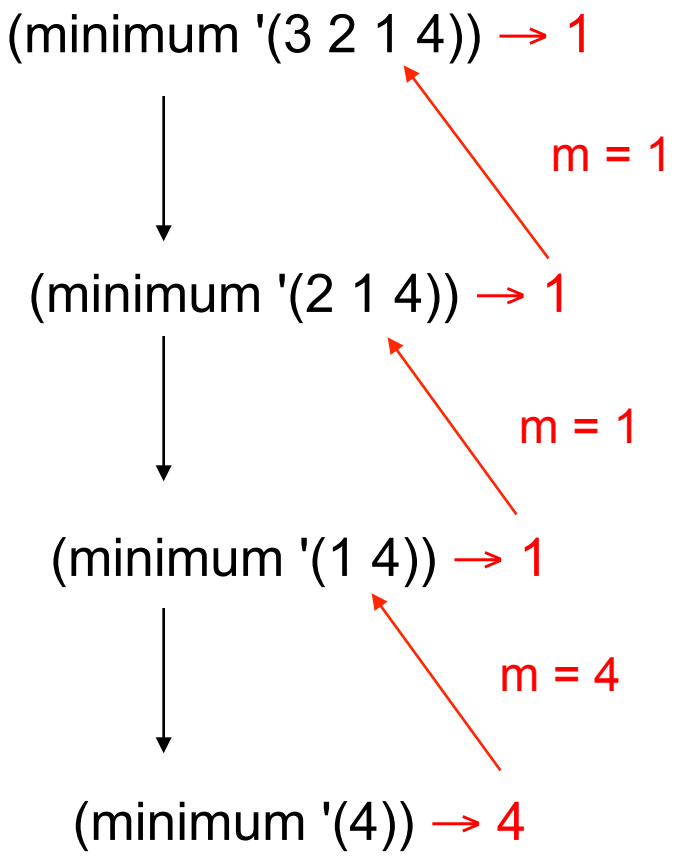
Fonctionnement du let

- 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

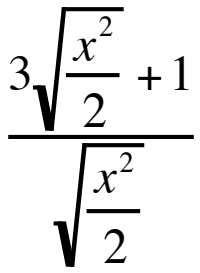
```
(define minimum ; → nombre
   (lambda (l) ; l liste de nombres  non vide
         (if       (null? (cdr l))
                   (car l)
                   (let ((m (minimum (cdr l))))
                         (if       (< (car l) m)
                                   (car l)
                     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

FONCTIONNEMENT DU NOUVEAU PROGRAMME



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*