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

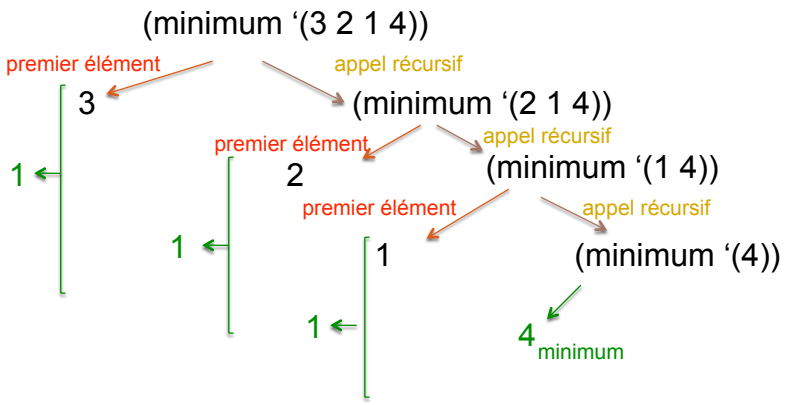


Illustration de l'algorithme vue au cours 1

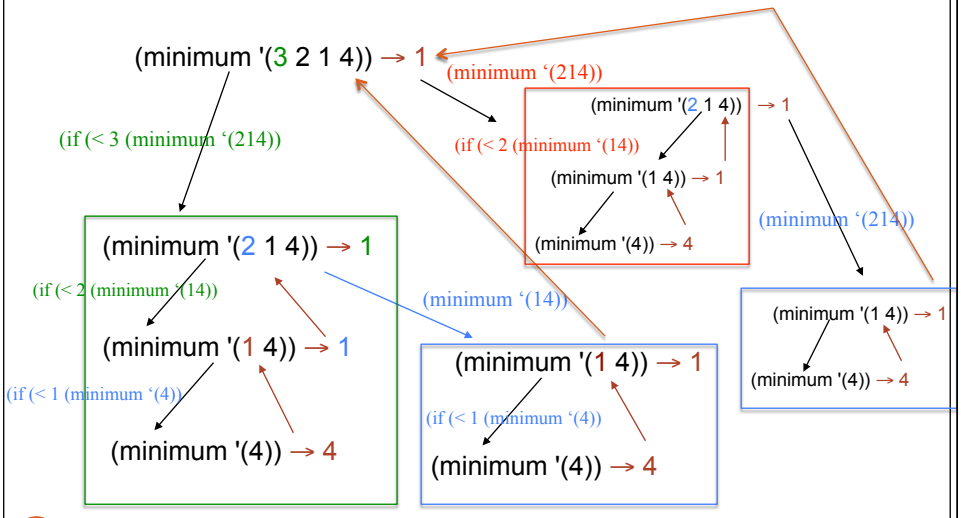
Illustration correspond à cet algorithme :

```
(define minimum ; → nombre
   (lambda (l) ; l liste de nombres non vide
         (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 ; → 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 réelle de la 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)

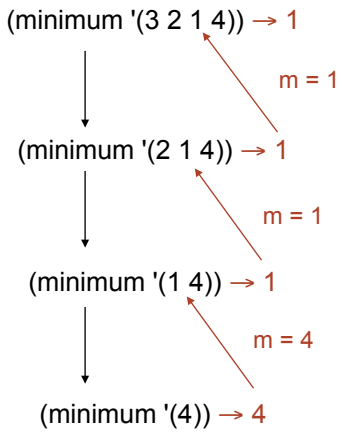
Fonctionnement du let

- 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

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

```
(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 strict[t] positif
      (let ((c (sqrt (/ (sqr x) 2))))
            (/ (+ (* 3 c) 1) c))))
```

- 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*