Fonctions d'O.S. sur les listes (7)

- `foldr` et `foldl` sont omnipotents:

```
(define somme
  (lambda (lst)
    (foldl + 0 lst)))

(define produit
  (lambda (lst)
    (foldl * 1 lst)))

(define append
  (lambda (lst1 lst2)
    (foldr cons lst2 lst1)))

; (foldr cons '(3 4) '(1 2)) = (cons 1 (cons 2 '(3 4)))

(define reverse
  (lambda (lst)
    (foldl rcons '() lst)))

(define rcons
  (lambda (x y)
    (cons y x)))

; (foldl rcons '() '(1 2)) = (rcons (rcons '() 1) 2)
;                          = (cons 2 (cons 1 '()))

(define length
  (lambda (lst)
    (foldl (lambda (x y) (+ x 1)) 0 lst)))
```


Continuations (1)

- Déf: la **continuation** d'une fonction f c'est une fonction k passée en paramètre à f qui indique quel calcul il faut faire avec "le(s) résultat(s)" de la fonction f
- Exemple: fonction qui additionne 2 nombres

```
(define add
  (lambda (x y cont)
    (cont (+ x y))))

(add 2 3 (lambda (r) (* r r)))  =>  25
```

- Exemple plus utile: "retourner" plus qu'un résultat

```
(define racines   ; racines de ax^2+bx+c=0
  (lambda (a b c cont)
    (let ((d (sqrt (- (* b b) (* 4 a c)))))
      (cont (/ (+ (- b) d) (* 2 a))
            (/ (- (- b) d) (* 2 a))))))

(racines 1 0 -1 (lambda (r1 r2) (* r1 r2)))  =>  -1
```

- Dans cet exemple l'utilisation d'une continuation évite la création d'une liste (pour retourner les 2 racines) et c'est plus élégant que:

```
(let ((r (racines 1 0 -1))) ; autre version de racines
  (let ((r1 (car r)) (r2 (cadr r)))
    (* r1 r2)))
```


Continuations (2)

- Exemple: partitionner une liste en 2 suivant une certaine propriété des éléments (fonction qui retourne un Booléen), il y a 2 résultats:

  1. éléments pour lesquels la fonction donne `#t`
  2. éléments pour lesquels la fonction donne `#f`

```
(define partitionner
  (lambda (f lst cont)
    (if (null? lst)
        (cont '() '())
        (partitionner
          f
          (cdr lst)
          (lambda (a b)
            (if (f (car lst))
                (cont (cons (car lst) a) b)
                (cont a (cons (car lst) b))))))))

(define trier ; utilisation pour implanter ``Quicksort''
  (lambda (f lst)
    (if (null? lst)
        '()
        (let ((pivot (car lst)))
          (partitionner
            (lambda (x) (f x pivot))
            (cdr lst)
            (lambda (a b)
              (append (trier f a)
                      (cons pivot (trier f b)))))))))

(trier < '(5 8 2 3 1 9))  =>  (1 2 3 5 8 9)
```


Continuations (3)

- À l'aide des continuations, il est possible de transformer **n'importe quelle fonction** en une fonction en **forme itérative**
- L'idée c'est d'ajouter une continuation à toute fonction qui est utilisée dans un appel non-terminal, afin de transformer cet appel en un appel terminal (la continuation représente le calcul qui consomme le résultat de l'appel)
- Exemple simple: calcul de la factorielle

```
(define fact
  (lambda (n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))))

; transformation en forme itérative:

(define fact-fi
  (lambda (n cont)
    (if (= n 0)
        (cont 1)
        (fact-fi (- n 1)
                 (lambda (r)
                   (cont (* n r)))))))

(define fact
  (lambda (n)
    (fact-fi n (lambda (r) r))))
```

Continuations (4)

- Trace de l'exécution

```
> (define fact-fi
    (lambda (n cont)
      (if (= n 0)
          (cont 1)
          (fact-fi (- n 1)
                   (lambda (r)
                     (cont (* n r)))))))
> (define fact
    (lambda (n)
      (fact-fi n (lambda (r) r))))
> (trace fact fact-fi)
> (fact 5)
| > (fact 5)
| > (fact-fi 5 '#<procedure #x174>)
| > (fact-fi 4 '#<procedure #x20E>)
| > (fact-fi 3 '#<procedure #x2A8>)
| > (fact-fi 2 '#<procedure #x342>)
| > (fact-fi 1 '#<procedure #x3DC>)
| > (fact-fi 0 '#<procedure #x476>)
| 120
120

; Avec:
;  #<procedure #x174> = (lambda (r) r)
;  #<procedure #x20E> = (lambda (r) ('#<procedure #x174> (* 5 r)))
;  #<procedure #x2A8> = (lambda (r) ('#<procedure #x20E> (* 4 r)))
;  #<procedure #x342> = (lambda (r) ('#<procedure #x2A8> (* 3 r)))
;  #<procedure #x3DC> = (lambda (r) ('#<procedure #x342> (* 2 r)))
;  #<procedure #x476> = (lambda (r) ('#<procedure #x3DC> (* 1 r)))
```

- Remarquable: les fermetures créées pour la continuation de `fact-fi` jouent le rôle des blocs d'activation (mémorisation de `n` et chaînage dynamique avec le prochain bloc par `cont`)

Continuations (5)

- Autre exemple de transformation à forme itérative: `append`

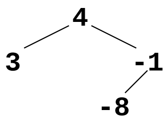
```
; pas en forme itérative:

(define append
  (lambda (lst1 lst2)
    (if (null? lst1)
        lst2
        (cons (car lst1)
              (append (cdr lst1) lst2)))))

; transformation en forme itérative:

(define append-fi
  (lambda (lst1 lst2 cont)
    (if (null? lst1)
        (cont lst2)
        (append-fi (cdr lst1)
                   lst2
                   (lambda (r)
                     (cont (cons (car lst1) r)))))))

(define append
  (lambda (lst1 lst2)
    (append-fi lst1 lst2 (lambda (r) r))))

> (trace append append-fi)
> (append '(1 2 3) '(4 5))
| > (append '(1 2 3) '(4 5))
| > (append-fi '(1 2 3) '(4 5) '#<procedure #x1DB>)
| > (append-fi '(2 3) '(4 5) '#<procedure #x291>)
| > (append-fi '(3) '(4 5) '#<procedure #x345>)
| > (append-fi '() '(4 5) '#<procedure #x3F7>)
| (1 2 3 4 5)
(1 2 3 4 5)

; Avec:
;  #<procedure #x1DB> = (lambda (r) r)
;  #<procedure #x291> = (lambda (r) ('#<procedure #x1DB> (cons 1 r)))
;  #<procedure #x345> = (lambda (r) ('#<procedure #x291> (cons 2 r)))
;  #<procedure #x3F7> = (lambda (r) ('#<procedure #x345> (cons 3 r)))
```


Continuations (6)

- Les continuations sont aussi utiles pour implanter des algorithmes générique de **fouille**

- Un arbre binaire peut être représenté à l'aide de listes: noeud = (*val gauche droite*) et arbre vide = ()

  Donc `(4 (3 () ()) (-1 (-8 () ()) ()))` représente



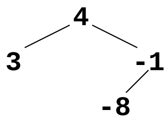
- Voici une fonction de fouille par parcours infixe de l'arbre

```
(define fouille-infixe
  (lambda (arbre visite fin) ; 2 continuations
    (if (null? arbre)
        (fin)
        (fouille-infixe (cadr arbre)
                        visite
                        (lambda ()
                          (visite (car arbre)
                                  (lambda ()
                                    (fouille-infixe
                                      (caddr arbre)
                                      visite
                                      fin))))))))
```


Continuations (7)

- Utilisation 1: trouver le premier nombre négatif

```
(fouille-infixe
  '(4 (3 () ()) (-1 (-8 () ()) ()))
  (lambda (x cont) (if (< x 0) x (cont)))
  (lambda () #f))  =>  -8
```



- Utilisation 2: convertir l'arbre en liste

```
(fouille-infixe
  '(4 (3 () ()) (-1 (-8 () ()) ()))
  (lambda (x cont) (cons x (cont)))
  (lambda () '()))  =>  (3 4 -8 -1)
```

- Utilisation 3: liste des nombres impairs

```
(fouille-infixe
  '(4 (3 () ()) (-1 (-8 () ()) ()))
  (lambda (x cont) (if (odd? x) (cons x (cont)) (cont)))
  (lambda () '()))  =>  (3 -1)
```

Fonctions en résultat (1)

- Les fonctions peuvent en plus **retourner des fonctions** comme résultat

- Par exemple, l'opération de **dérivation** d'une fonction peut être vue comme étant une fonction d'O.S. qui prend une fonction en paramètre et **retourne la fonction qui est sa dérivée**

  Ainsi, la dérivée de la fonction "sin" c'est la fonction "cos"

- Implantation de la fonction de dérivation par approximation de la pente

```
(define d
  (lambda (f)
    (lambda (x)
      (/ (- (f (+ x .0001))
            (f x))
         .0001))))

(define c (d sin))  ; la fonction d retourne une fonction

(c 0)    =>  .999999998   (c .5)   => .877558589
(cos 0)  =>  1            (cos .5) => .877582561
```


Fonctions en résultat (2)

- Déf: soit une fonction f à n paramètres (où $n > 1$), la forme **currifiée** de f est une fonction f' à **un paramètre** (le premier paramètre de f) et qui retourne comme résultat **une fonction currifiée traitant les paramètres restants**

- Une fonction prenant exactement un paramètre est considérée déjà currifiée

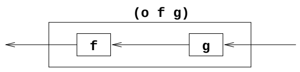
- Exemple: addition

```
; non-currifiées:
(define add (lambda (x y) (+ x y)))
(define mul (lambda (x y) (* x y)))

(add 2 3)  =>  5           (mul 2 3)  =>  6

; currifiées:
(define addc (lambda (x) (lambda (y) (+ x y))))
(define mulc (lambda (x) (lambda (y) (* x y))))

((addc 2) 3)  =>  5        ((mulc 2) 3)  =>  6

(define f (addc 2))
(define g (mulc 2))

(f 3)  =>  5               (g 3)  =>  6
```


Fonctions en résultat (3)

- La fonction de composition de fonctions (l'opérateur "o" en mathématique) retourne également une fonction comme résultat:

```
(define o (lambda (f g) (lambda (x) (f (g x)))))
```




```
((o not odd?) 8)  =>  #t

((o car reverse) '(1 2 3 4))  =>  4
```

- Les fonctions currifiées permettent de facilement **construire** des nouvelles fonctions "au besoin" en les composant:

```
(map (o (addc 1) (mulc 2)) '(10 20 30))  =>  (21 41 61)

(define less-than (lambda (x) (lambda (y) (> x y))))

(garder (o (less-than 5) abs) '(7 2 -6 -3))  =>  (2 -3)

(define comp
  (lambda (fns)
    (foldl o (lambda (x) x) fns)))

((comp (list (addc 1) (mulc 2) sqrt abs)) -9)  =>  7
```


Évaluation paresseuse (1)

- Scheme et la plupart des langages fonctionnels permettent de **retarder** l'évaluation d'une expression

- En Scheme cela se fait avec la forme spéciale `delay` et la fonction `force`:

  – (`delay` ⟨*expr*⟩) retourne une **promesse** P

  – (`force` P) force l'évaluation de ⟨*expr*⟩ (si ce n'est pas déjà fait) et retourne sa valeur

- Exemple simple

```
> (define x (delay (expt 2 100))) ; expt pas appelée
> x
#<promise #x100380CA>
> (force x) ; fait l'évaluation de (expt 2 100)
1267650600228229401496703205376
> (force x) ; retourne la valeur mémorisée
1267650600228229401496703205376
```

- L'évaluation est donc retardée jusqu'à ce qu'un `force` signale le besoin de connaître le résultat de l'évaluation

Évaluation paresseuse (2)

- L'attrait principal de l'évaluation paresseuse c'est de pouvoir manipuler des structures de données **infinies**

- Cela est possible sous la condition que **seulement une partie finie de la structure est examinée**

- Les **listes infinies** (**"streams"** en anglais) sont un exemple classique

- Un **stream** est soit:

  1. une liste vide, i.e. ()

  2. une paire ayant comme champ `cdr` une promesse dont l'évaluation (par `force`) donnera un **stream**

- Exemple: une liste infinie contenant que des 1

```
> (define uns (cons 1 (delay uns)))
> (car uns)
1
> (car (force (cdr uns)))
1
```


Évaluation paresseuse (3)

- Fonctions sur les **streams**

```
(define stream-car (lambda (str) (car str)))
(define stream-cdr (lambda (str) (force (cdr str))))

(define stream-head  ; extraire n premiers éléments
  (lambda (n str)
    (if (or (= n 0) (null? str))
        '()
        (cons (stream-car str)
              (stream-head (- n 1) (stream-cdr str))))))

(stream-head 10 uns)  =>  (1 1 1 1 1 1 1 1 1 1)

(define stream-map2
  (lambda (f str1 str2)
    (if (or (null? str1) (null? str2))
        '()
        (cons (f (stream-car str1) (stream-car str2))
              (delay (stream-map2
                       f
                       (stream-cdr str1)
                       (stream-cdr str2)))))))

(define deuxs (stream-map2 + uns uns))

(stream-head 10 deuxs)  =>  (2 2 2 2 2 2 2 2 2 2)

(define entiers
  (cons 0 (delay (stream-map2 + entiers uns))))

(stream-head 10 entiers)  =>  (0 1 2 3 4 5 6 7 8 9)

(define carres (stream-map2 * entiers entiers))

(stream-head 10 carres)  =>  (0 1 4 9 16 25 36 49 64 81)
```


Évaluation paresseuse (4)

```
(define f    ; quelle est la valeur de cette liste infinie?
  (cons 0 (delay
            (cons 1 (delay
                      (stream-map2 + f (stream-cdr f)))))))

; autre exemple: liste infinie des nombres premiers

(define stream-garder
  (lambda (f str)
    (cond ((null? str)
           '())
          ((f (stream-car str))
           (cons (stream-car str)
                 (delay
                   (stream-garder f (stream-cdr str)))))
          (else
           (stream-garder f (stream-cdr str))))))

(define elim
  (lambda (str)
    (stream-garder
      (lambda (x)
        (not (= (modulo x (stream-car str)) 0)))
      (stream-cdr str))))

(define deux-et+ (stream-cdr (stream-cdr entiers)))

deux-et+                          => (2 3 4 5 6 7 8 9 10 ...
(elim deux-et+)                   => (3 5 7 9 11 13 15 17 19 ...
(elim (elim deux-et+))            => (5 7 11 13 17 19 23 25 ...
(elim (elim (elim deux-et+))) => (7 11 13 17 19 23 29 31 ...

(define crible
  (lambda (str)
    (cons (stream-car str)
          (delay (crible (elim str))))))

(crible deux-et+) => (2 3 5 7 11 13 17 19 23 29 31 ...
```


Affectation et effets de bord (1)

- Scheme n'est pas un langage **purement fonctionnel** car il possède l'**affectation** (et des fonctions d'**entrée/sortie**)

- Le **style impératif** est donc possible

- Afin de traiter l'affectation correctement chaque variable dans l'environnement est en fait une **cellule** dont le contenu peut être modifié par affectation

- L'affectation se fait avec la forme spéciale `set!`:

  – (`set!` ⟨*var*⟩ ⟨*expr*⟩) évalue ⟨*expr*⟩ et stocke le résultat dans la cellule de la variable ⟨*var*⟩

  – `set!` retourne une valeur indéfinie

- Exemple

```
> (define x 1)
> (set! x (+ x 1))
> x
2
> (set! x "allo")
> x
"allo"
```

Affectation et effets de bord (2)

- Scheme possède également la forme spéciale `begin` qui exprime un **séquencement** d'évaluations (utile pour forcer l'ordre d'exécution des effets de bord)
  - `(begin` $\langle expr_1 \rangle$ `...` $\langle expr_N \rangle$`)` évalue chaque sous-expression de gauche à droite
  - la valeur de $\langle expr_N \rangle$ est retournée
- Exemple

```
> (begin (display "hello world!") (newline) (+ 2 3))
hello world!
5
> (define boucle
    (lambda ()
      (begin
        (display "entrer n: ")
        (let ((x (read)))
          (if (number? x)
            (begin
              (write (sqrt x))
              (newline)
              (boucle)))))))
> (boucle)
entrer n: 10
3.1622776601683795
entrer n: 4
2
entrer n: fin
```


Affectation et effets de bord (3)

- Il y a une interaction interessante entre les fermetures et les affectations
- Chaque fermeture mémorise l'environnement de la lambda-expression correspondante **incluant les cellules associées**:

```
> (define accumuler
    (let ((n 0))
      (lambda (x)
        (begin (set! n (+ n x)) n))))
> (accumuler 5)
5
> (accumuler 5)
10
```

- Dans cet exemple, on a encapsulé la variable `n`
- Déf: une variable ou fonction est **encapsulée** si elle est accessible seulement aux parties du programme qui ont une raison légitime d'y accéder
- L'encapsulation permet d'accroître la **modularité** (car dépendances claires) et **robustesse** (on peut garantir des invariants par examination locale du code: par exemple, `n` toujours un nombre)

Affectation et effets de bord (4)

- L'encapsulation est une approche utile en **programmation orientée-objet** (les données sont encapsulées dans la définition de classe)
- Exemple du style O.O. en Scheme: on se sert de fermetures pour représenter les objets

```
> (define creer-compte
    (lambda (solde)
      (lambda (msg)
        (cond ((equal? msg 'depot)
               (lambda (n)
                 (if (< n 0)
                     (error "dépot erroné")
                     (set! solde (+ solde n)))))
              ((equal? msg 'retrait)
               (lambda (n)
                 (if (or (< n 0) (> n solde))
                     (error "retrait erroné")
                     (set! solde (- solde n)))))
              ((equal? msg 'lire-solde)
               (lambda () solde))
              (else
               (error "opération erronée"))))))
> (define c1 (creer-compte 100))
> (define c2 (creer-compte 0))
> ((c1 'retrait) 30)
> ((c2 'depot) 15)
> ((c1 'lire-solde))
70
> ((c2 'lire-solde))
15
> ((c1 'retrait) 80)
*** ERROR -- retrait erroné
```


Affectation et effets de bord (5)

- L'encapsulation sert aussi à créer des **modules** qui **cachent** leur implantation des **clients**
- En Scheme: affectation + **définitions locales**
- Exemple: module de génération de nombres pseudo-aléatoires suivant certaines distributions

```
(define loi-uniforme #f)
(define loi-exponentielle #f)

(let ()   ; frontière d'encapsulation

  (define graine 1673) ; état du générateur
  (define k1 3581) ; constantes du générateur congruentiel
  (define k2 12751)
  (define k3 131072)

  (define entier-aleatoire
    (lambda ()
      (set! graine (modulo (+ (* graine k1) k2) k3))
      graine))

  (define uniforme
    (lambda (a b) ; a et b sont les bornes de l'intervalle
      (+ a (* (- b a) (/ (entier-aleatoire) (- k3 1))))))

  (define exponentielle
    (lambda (m) ; m est la moyenne
      (- (* m (log (uniforme 0 1))))))

  (set! loi-uniforme uniforme)
  (set! loi-exponentielle exponentielle))
```