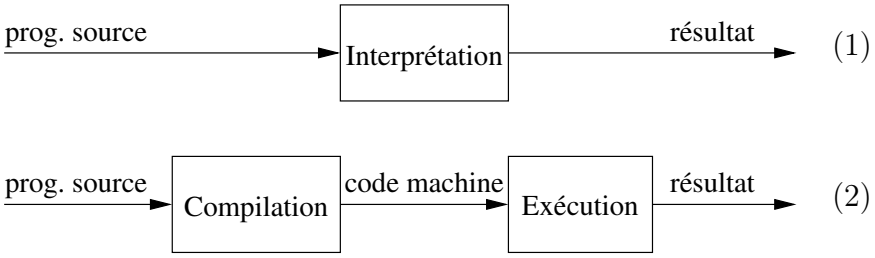
Programmation applicative – L2
TD 1 : Premiers concepts d'évaluation

1 Introduction

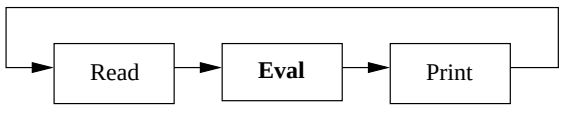
Scheme est un langage de la famille Lisp crée en 1975 par des enseignants (Sussman et Steele entre autres) du célèbre MIT. C'est un langage de programmation *fonctionnel*[1] dont le but est plus universitaire qu'industriel.

Il existe 2 grandes familles de langages de programmation, langage interprété et langage compilé :



— Langage dit *interprété* (1) (ex : Scheme, Lisp). C'est à dire qu'un programme, l'interpréteur, renvoie directement la valeur d'une expression écrite dans le langage.
— Langage dit *compilé* (2) (ex : C++, Java). La compilation sert à transformer le programme source en code machine spécifique et directement exécutable par une machine.

Programmer en Scheme consiste à donner des expressions (respectant la syntaxe de Scheme) à évaluer à l'interpréteur Scheme : la boucle d'interaction REP.



Il existe plusieurs interpréteurs Scheme disponibles en ligne. Nous utiliserons en TP l'interpréteur DrRacket.

1. Se dit d'un style de programmation dans lequel on n'utilise que des fonctions comme en mathématique sans utiliser le mécanisme d'affectation (modification de la valeur d'une variable). En opposition à programmation *impérative*, qui se dit d'un style de programmation dans lequel on utilise des procédures dont les instructions modifient, par l'affectation en mémoire, des variables.

2 Syntaxe de Scheme (notation parenthésée préfixée et S-expressions)

Toute expression syntaxiquement correcte de Scheme est une *S-expression*. Une S-expression est :

— soit un nombre : `1, 2, 3`
— soit un symbole : `x, square, +, #t, #f`
— soit une liste (éventuellement vide) de S-expressions encadrées par 2 parenthèses `(S-expression ... S-expression)`

Scheme utilise la notation parenthésée préfixée : `(opérateur opérande`$_1$ `...` `opérande`$_k$`)`.

— Le parenthésage permet de ne pas avoir à se soucier des priorités entre les opérateurs.
— La notation préfixée à l'avantage de ne pas répéter l'opérateur lorsque il y a plusieurs opérandes.

```
36 + 14                ⇝  (+ 36 14)
3 × (2 + 4)            ⇝  (* 3 (+ 2 4))
1 + 2 + 3 + 4 + 5      ⇝  (+ 1 2 3 4 5)
```

Exercice 1 *Transformer les expressions suivantes en notation préfixée :*

— $5 - 4$
— $2 - 6 + 10$
— $2 + 5 \times 3$
— $3 \times 5 - 7/2$
— $5 + 3 \times (4 - 2) + 5/(2 + 1)$

Exercice 2 *En utilisant les noms de fonctions Scheme :* `+, -, *, /, <, =, sqrt (`$\sqrt{\ldots}$`), and, or, not`*, transformer les expressions algébriques suivantes en expressions Scheme :*

$1 - (2 - 3)$
$3 + 7 \times (2 + 1)$
$(a + 1) \neq b$
$\frac{a \times x + b}{c \times x + d}$
$(1 - 2) - 3$
$a \times x^2 + b \times x + c$
$(a < b)$ *ou* $(a = b^2)$
$\sqrt{b^2 - 4 \times a \times c}$

Exercice 3 *Les expressions Scheme suivantes sont-elles syntaxiquement correctes ? Si ce n'est pas le cas, préciser ce qui ne va pas.*

```
(+ 1 (* 2 3))
+
(+ 2 *)
(+ (/ 2 3) 1))(+ a b)
(= 1)
```

Exercice 4 *Donner la forme préfixée des expressions suivantes :*

a) $2 + 4x^2 - 3x^3$

b) $\frac{sin(x+y) \times cos(x-y)}{1+cos(x+y)}$

3 Les formes spéciales `define` et `lambda`

Une expression Scheme peut être de 3 types :

Expression atomique	Nombre, symbole
Forme spéciale	Définition (`define`)
	Lambda expression (`lambda`)
	Condition (`if, cond`)
	Quotation (`quote`)
	Sequence (`begin`)
	Affectation (`set!`)
Application de fonction	`(opérateur opérande`$_1$ `...` `opérande`$_k$`)`

La forme spéciale `define` permet de faire des abstractions. C'est à dire nommer par des symboles des données ou des fonctions. Ces noms sont stockés dans une mémoire de façon à être réutilisés.

— Définition de données : `(define` $\langle symbole \rangle$ $\langle expression \rangle$`)`

— Définition de fonctions : `(define` $\langle symbole \rangle$ `(lambda (`$\langle param_1 \dots param_k \rangle$`))` $\langle corps \rangle$`))`

La forme spéciale `lambda` est fondamentale en Scheme, c'est elle qui permet de **construire des fonctions**. Remarque – Notez que la forme spéciale `define` ne renvoie pas de valeur.

Exercice 5 *Définir en Scheme les fonctions suivantes à l'aide de lambda-expressions : :*

— $f : x \mapsto x^3$

— $g : y \mapsto 3y + 7$

— $h : z \mapsto 3f(z) + g(z) + 1$

— *La fonction identité* $id : x \mapsto x$.

— *La fonction proj2 :* $(x, y) \mapsto y$.

— *La fonction einstein :* $(u, v) \to \frac{u+v}{1+\frac{uv}{c^2}}$ *avec* $c = 300000$

Exercice 6 *Que calculent les fonctions définies de la façon suivante :*

```
(define f1 (lambda () 0))
(define f2 (lambda (x) (* x x)))
(define f3 (lambda (x y) (+ (f2 x) (f2 y))))
(define distc (lambda (x1 y1 x2 y2) (f3 (- x1 x2) (- y1 y2))))
```

Exercice 7 *Suivre le cheminement de Scheme sur l'exemple suivant. Que se passe-t-il à chacune de ces commandes ?*

```
> (define pi 3.14)
> pi
> (define carre (lambda (x) (* x x)))
> (carre 5)
> (define cercle (lambda (r) (* pi (carre r))))
> (cercle 12)
```

4 La forme spéciale `if`

La forme spéciale `if` est une structure de contrôle fondamentale. Sa syntaxe est : `(if exp-test exp-alors exp-sinon)`. Lors de l'évaluation d'un `if`, il y a d'abord évaluation de `exp-test` et si son résultat est `#t` alors il y a évaluation de `exp-alors` sinon il y a évaluation de `exp-sinon`.

Exercice 8 *Définissez la fonction* `abs` *qui renvoie la valeur absolue d'un nombre. Écrivez ensuite la fonction* `care-div` *qui fait la division de 2 nombres en vérifiant au préalable que le diviseur n'est pas nul.*

5 Expressions et fonctions non récursives

Exercice 9 *Donner la fonction Scheme* `fahrenheit->celsius` *qui permet de convertir une température donnée en degré Fahrenheit en température Celsius en utilisant la formule suivante :*

$$F(C) = \frac{9}{5}(C + 40) - 40$$

Écrire une fonction `liquide-cel?` *pour savoir si une température donnée en degrés celsius correspond à un état de l'eau qui n'est ni glace ni vapeur.*

Donner une fonction `liquide-fah?` *pour savoir si une température donnée en degrés fahrenheit correspond à un état de l'eau qui n'est ni glace ni vapeur.*

Exercice 10 *Donner les fonctions* `puissance-2, puissance-4, puissance-6` *et* `puissance-9`*, qui, étant donné un argument x, calculent respectivement x^2, x^4, x^6 et x^9.*

Programmation applicative – L2
TD 2 : Définitions de fonction, suite

1 Expressions et fonctions non récursives

Exercice 1 Triangles. *Écrire une fonction* `median` *qui, étant donnés trois nombres, donne le nombre médian. Écrire une fonction* `triangle?` *qui permet de savoir si trois nombres peuvent être les longueurs des côtés d'un triangle. De même, écrire les fonctions* `equilateral?`, `isocele?` *et* `rectangle?`.

Exercice 2 Caractères. *Ecrire une fonction qui prend un paramètre en entrée, teste si c'est un caractère minuscule plus petit que* `#\m` *(avec la fonction* `char<?`*) et le cas échéant renvoie le même caractère en majuscule. On aura besoin des fonctions suivantes sur les caractères :* `char-alphabetic?` *qui prend en paramètre un caractère et renvoie vrai si c'est une lettre, faux sinon ;* `char-lower-case?` *qui prend en paramètre un caractère et renvoie vrai si c'est une minuscule, faux sinon, et enfin* `char-upcase` *qui prend en paramètre un caractère et renvoie le même caractère mais en majuscule, s'il existe.*

Exercice 3 Chaîne de caractères. *Ecrire une fonction qui prend deux paramètres en entrée, teste si le premier paramètre est une chaîne de caractères, et si c'en est une, donne la lettre de numéro correspondant au deuxième paramètre. Rappel : on aura besoin des fonctions* `string?` *qui prend un paramètre et renvoie vrai si c'est une chaîne de caractères, faux sinon,* `string-length` *qui renvoie la longueur d'une chaîne de caractères passée en paramètre, et* `string-ref` *qui prend en paramètre une chaîne de caractères et un indice n, et renvoie le caractère d'indice n dans la chaîne.*

Exercice 4 Heures/Minutes/Secondes. *Écrire une fonction* `hms-vers-s` *qui permet de convertir un temps donné en heures, minutes, secondes, en temps exprimé en secondes.*

Écrire une fonction `hms?<` *qui prend 2 temps donnés sous la forme heures-minutes-secondes et qui donne la valeur* `#t` *si et seulement si le premier temps est strictement inférieur au second. On donnera deux variantes de cette fonction : une qui utilise la fonction précédente, et une qui ne l'utilise pas.*

Exercice 5 Tonneau dans une voiture. *Une voiture ne peut contenir que des tonneaux de forme cylindrique tels que la hauteur est inférieur à 1m, le rayon inférieur à 80cm et le volume inférieur à* $1m^3$. *Écrire une fonction* `voiture?` *qui, étant données 2 nombres représentant respectivement la hauteur et le rayon du tonneau en mètre, permet de savoir si le tonneau tient dans une voiture.*

Exercice 6 Viticulture. *Une entreprise viticole expédie une quantité q de vin de prix unitaire prix. Si la commande est de 600 Euros, ou plus, le port est gratuit, sinon il est facturé à 10% de la commande (dans le cas où la commande est inférieure à 600 Euros et que la commande plus le port est supérieur à 600 Euros, le prix est ramené à 600 Euros). Écrire une fonction* `commande` *permettant de calculer la somme à facturer.*

2 Exercice supplémentaire : Calcul de la date de Pâques

Le dimanche de Pâques se situe chaque année entre le 22 mars et le 25 avril, c'est le 1^{er} dimanche après la première pleine lune de l'équinoxe de printemps. La date de Pâques est calculée à partir du nombre d'or d'une année et de l'âge de la Lune appelé l'Epacte. Cet algorithme assez compliqué, caractéristique des religieux, renvoie néanmoins le numéro du jour à partir du 1^{er} mars qui correspond à la date de Pâques. Pour une `annee` exprimée en 4 chiffres posons :

- a = annee modulo 19
- b = annee modulo 4
- c = annee modulo 7
- d = (19a + 24) modulo 30
- e = (2b+4c+6d+5) modulo 7
- Le numéro du jour est alors : 22+d+e

Exercice 7 *Écrivez les fonctions suivantes :*

1. `numjour` *qui prend en argument* `annee` *et renvoie le numéro du jour à partir du* 1^{er} *mars.*
2. `traduit` *qui prend en argument le résultat de la fonction* `numjour` *et imprime la date du dimanche de Pâques :* $\Rightarrow$ `jour mois`.
3. *Enfin,* `paques` *qui imprime la date de Pâques pour une année donnée en argument.*

Remarque — On dispose en Scheme de la fonction `modulo` qui renvoie le reste d'une division. Ainsi pour $n_1 = n_2 n_3 + n_4$ la fonction `(modulo n1 n2)` retourne n_4 et la fonction `(quotient n1 n2)` retourne n_3.

Programmation applicative – L2
TD 3 : Variables, liaisons, portée

1 La forme spéciale `let`

La forme spéciale `let` est très pratique car elle permet de définir des **variables locales**. En effet, il est souvent utile de définir localement à une fonction (ou à une expression) des variables qui sont utilisées seulement dans le corps de cette fonction (ou expression).

```
(let ((⟨var1⟩⟨exp1⟩)
      (⟨var2⟩⟨exp2⟩)
      ...
      (⟨varn⟩⟨expn⟩))
   ⟨corps⟩)
```

Par exemple :

```
> (let ((a 1)(b 2)(c (* 2 3)))
     (+ (* b c) a))
⇒ 13
```

Remarque — Les variables locales définies par un `let` peuvent être des données comme des fonctions. Il est parfois utile de définir des fonctions simplement localement. Par exemple :

```
> (let ((a 1)
        (foo (lambda (x) (+ 2 x))))
     (+ (foo 2) a))
⇒ 5
```

Évaluation de la forme `let` Pour chaque définition (couple *symbole/expression*) le symbole est associé à l'évaluation de son expression. Le corps du `let` est évalué après substitution des variables définies par leur valeur.

Remarque — La substitution variable/valeur est effectuée uniquement dans le corps et pas dans le bloc de définition.

Le corps du `let` est une séquence d'expression. De la même manière qu'avec la forme `begin`, toutes les expressions de la séquence sont évaluées et la valeur de la dernière expression est le résultat renvoyé par le `let`.

Exercice 1 *Quelle est la valeur des expressions suivantes :*

a)
```
(let ((a 2)
      (b (* 2 2))
      (c 10))
   (* c (- a b)))
```

b)
```
(let ((x 5))
   (* x x)
   (let ((x 10))
      (* x x)))
```

c)
```
(let ((a 2)
      (b (let ((a 3) (b 4))
           (- a b)(+ a b))))
   (* a b))
```

2 `let` et `lambda`, même combat !

Lorsque un `let` est évalué il y a substitution des variables locales var_i dans le corps du `let`, par la valeur de l'exp_i qui leur est associé dans la définition du `let`. Ce qui est exactement le comportement de l'évaluation d'une lambda expression. Ainsi :

$$\texttt{(let ((}\langle var_1\rangle\langle exp_1\rangle\texttt{)}\ \texttt{(}\langle var_2\rangle\langle exp_2\rangle\texttt{)}\ \ldots\ \texttt{(}\langle var_n\rangle\langle exp_n\rangle\texttt{)))}\ \langle corps\rangle\texttt{)} \quad\Leftrightarrow\quad \texttt{((lambda (}\langle var_1\rangle\langle var_2\rangle\ldots\langle var_n\rangle\texttt{)}\ \langle corps\rangle\texttt{)}\ \langle exp_1\rangle\langle exp_n\rangle\ldots\langle exp_n\rangle\texttt{)}$$

Exercice 2 *Ré-écrire une des expressions de l'exercice 1 en utilisant* `lambda`*.*

3 Liaison et portée d'une variable

Les notions de liaison et de portée d'une **variable** sont fondamentales dans tous les langages informatiques. On appelle **liaison** le lien qui est effectué entre un symbole var_i et une valeur correspondant au résultat de l'évaluation de l'expression exp_i. Ce qui signifie que le symbole var_i désigne la valeur de exp_i. On dit que ce lien a un espace de validité que l'on appelle **portée**.

Par exemple, l'expression `(define a (+ 2 3))` lie le symbole `a` à la valeur de `(+ 2 3)` c'est à dire `5`. La portée d'un `define` au *top-level* (c'est à dire au niveau du prompteur "`>`") est toute la session Scheme, on parle de **variable globale**. Par contre, dans l'expression `(let ((a (+ 2 2))) a)` le symbole `a` est lié à la valeur `4` seulement dans le corps du `let`, on parle alors de **variable locale**. Ainsi :

```
> (define a (+ 2 3))                  ⇒ rien
> a                                   ⇒ 5
> (let ((a (+ 1 1)) (b 2)) (+ a b))   ⇒ 4
> a                                   ⇒ 5
> b                                   ⇒ erreur - b non défini
```

Remarque — On note que dans l'expression `let` le `a` global n'est pas visible car il est caché par le `a` local. De façon générale, les variables définies par les liaisons d'un `let` cachent les autres définitions seulement le temps de l'évaluation du corps du `let`.

Lorsque Scheme évalue une variable, il regarde à quoi celle-ci est liée avant tout de façon locale, si une liaison est trouvée alors il l'utilise, sinon il regarde au niveau local supérieur. Ainsi :

```
> (let ((b 2)) (+ a b))  ⇒ 7
```

Exercice 3 *Quelle est la valeur des expressions suivantes tapées au top-level :*

```
(define a 10)

(let ((x a)) (* 2 x))
(let ((a a)) (* 2 a))

(let ((a (* 2 a))) (* 2 a))

(let ((a (* 2 a)) (b 5) (c (+ 1 a)))
   (+ a b c))
```

Exercice 4 *Quel est l'affichage engendré par l'expression suivante ?*

```
(let ((a 'b) (b 3))
   (display 'a)(display " : ")(display a)(newline)
   (display 'b)(display " : ")(display b)(newline)
   (let ((a 4) (b a))
      (display 'a)(display " : ")(display a)(newline)
      (display 'b)(display " : ")(display b)))
```

3.1 Les autres `let` : `let*`, `letrec`

Pour faire des liaisons en séquence dans un `let` on doit utiliser la forme spéciale `let*`. Elle permet d'utiliser dans le calcul de exp_j la valeur de var_i si $i < j$. Elle permet de voir des variables liées préalablement dans le `let*`. On a donc l'équivalence :

```
(let* ((⟨var1⟩⟨exp1⟩)
       (⟨var2⟩⟨exp2⟩)
       ...
       (⟨varn⟩⟨expn⟩))
   ⟨corps⟩)
```

$\Leftrightarrow$

```
(let ((⟨var1⟩⟨exp1⟩))
   (let ((⟨var2⟩⟨exp2⟩))
      ...
      (let ((⟨varn⟩⟨expn⟩))
         ⟨corps⟩)...)
```

Par exemple pour la fonction `numjour` de l'exercice 5 de la feuille TD2, qui renvoyait le nombre de jour pour la date de Pâques, il est pratique d'utiliser un `let*` :

```
(define (numjour-let annee)
   (let* ((a (modulo annee 19))
          (b (modulo annee 4))
          (c (modulo annee 7))
          (d (modulo (+ (* 19 a) 24) 30))
          (e (modulo (+ (* 2 b) (* 4 c) (* 6 d) 5) 7)))
      (+ 22 d e)))
```

Il existe également un `let` qui permet de gérer les appels récursifs. C'est la forme spéciale `letrec` que nous détaillerons plus tard. Par exemple :

```
(letrec ((fact (lambda (x) (if (= 0 x) 1 (* x (fact (- x 1)))))))
         (a 5))
     (fact a))
```

Exercice 5 *Quelle est la valeur des expressions suivantes tapées au top-level :*

```
(define a 10)

(let ((a 2)(b a))
   (* a b))

(let* ((a 2)(b a))
   (* a b))
```

```
(let* ((a (* 2 a)) (b 5) (c (+ 1 a)))
   (+ a b c))

(letrec ((a (* 2 a)) (b 5) (c (+ 1 a)))
     (+ a b c))

(let ((x 5))
  (let* ((y (+ x 10))
         (z (* x y)))
    (+ x y z)))
```

Remarque — Notez que dans les liaisons en séquences, `letrec` se comporte comme `let*`.

Exercice 6 *Quelle est la valeur des expressions suivantes :*

```
(define a 10)

(define (foo x) (* a x))

(let ((a 0)) (foo 5))

(define a 100)

(let ((a 0)) (foo 5))

((lambda (a)
   (let ((a 1)
         (foo (lambda (x) (+ a x))))
     (foo a))) 5)

((lambda (a)
   (let* ((a 1)
          (foo (lambda (x) (+ a x))))
     (foo a))) 5)
```

Programmation applicative – L2
TD 4 : Fonctions récursives

1 Combinatoire

Exercice 1 *Donner la définition de la fonction* `somme_carres` *par récursion telle que :*

$$\texttt{(somme\_carres n)} = \sum_{i=1}^{n} i^2.$$

Exercice 2 *Définir la fonction puissance qui calcule la valeur de* n^k *:*

```
(define (puissance n k) ...)
```

Exercice 3 *Petit problème. Faut-il être d'accord avec l'échange suivant : chaque jour pendant 30 jours, on vous donne 300 000 euro. En échange vous donnez le premier jour 1 centime, puis 2 centimes le deuxième jour, 4 le troisième, etc. En doublant la valeur chaque jour. Afin de vous aider dans ce choix, écrivez les fonctions qui vous permettrons de calculer :*

1. *La somme des valeurs que vous devez donner pour un nombre de jours n.*
2. *Le gain éventuel obtenu entre la somme qu'on vous donne et la somme que vous devez donner pour un nombre de jours n.*

2 Nombres vus comme des suites de chiffres

Exercice 4 *Définir la fonction* `sommechiffres` *qui calcule la somme des chiffres qui compose le nombre entier positif. Exemples :* `(sommechiffres 1432)` *renvoie 10.*

Exercice 5 *Définir la fonction* `avec?` *qui prend en argument un chiffre c et un nombre x et qui rend en valeur* **vrai** *si le chiffre c apparaît dans le nombre x.*

Exercice 6 *Définir la fonction* `puissancemax` *qui prend en argument un entier p et un entier m, et qui rend en valeur le plus grand nombre* p^n *tel que* $p^n \leq m$ *. On suppose que* $p \geq 2$ *et que* $m \geq 1$*. Exemples :*

```
> ( puissancemax 2 18)
16
```

```
> ( puissancemax 10 1432)
1000
```

Exercice 7 *Définir la fonction* `chiffrescroissants` *qui, pour un argument n, construit le nombre* `1234...n`*. Exemples :*

```
> ( chiffrescroissants 8)
12345678
```

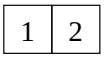
```
> ( chiffrescroissants 13)
12345678910111213
```

3 Zéro d'une fonction

Exercice 8 *Définir la fonction* `zero` *qui prend en argument une fonction f, et deux bornes réelles a et b. Cette fonction doit rendre en valeur un zéro de la fonction f (c'est-à-dire une valeur du paramètre x, telle que $f(x) = 0$), qui soit entre a et b. Il est supposé que $f(a) * f(b) \leq 0$. La méthode consiste à faire une recherche dichotomique en découpant l'intervalle $[a, b]$ en 2 sous-intervalles et en relançant récursivement sur le premier sous-intervalle ou le second. La précision du résultat est donnée par une variable globale* `prec` *telle que par exemple :* `(define prec 0.001)` *Cela signifie que la valeur calculée doit être à moins de* `prec` *du zéro.*

Programmation applicative – L2
TD 5 : Introduction aux paires et aux listes

1 La structure de données `pair`

Une paire, ou doublet, est constituée de deux valeurs : `(cons 1 2)` $\rightsquigarrow$ 

Le constructeur de paire est l'opérateur `cons`, l'opérateur `car` permet d'accéder au premier élément de la paire et `cdr` permet d'accéder au second élément de la paire.

2 Structure de données liste

Rappel : `(car l)` donne la tête de la liste et `(cdr l)` donne la queue de la liste (c'est-à-dire la liste privée de la tête), `(cons t r)` réunit une tête et une queue de liste pour construire une liste. Écrire `'e` retourne l'expression `e` elle-même, et non le résultat de son évaluation. Par exemple, `'(+ 5 6)` retourne la liste `(+ 5 6)` alors que l'expression `(+ 5 6)` renvoie `11`.

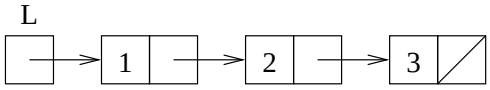
Une liste se définit à partir de la structure de paire et de la valeur **liste vide**. La liste vide est considérée comme une expression atomique et se note `null` ou `()`.

Une expression `L` est une liste soit :

— Si `L` est la liste vide

— Si `(cdr L)` est une liste

Une liste est donc une chaîne de doublets qui se termine par la valeur liste vide.

`(define L (cons 1 (cons 2 (cons 3 ()))))` 

La structure de liste émerge de la *propriété de fermeture* du `cons`, c'est-à-dire le fait de créer des paires dont les éléments sont des paires.

Exercice 1 *Répondez aux questions suivantes :*

1. *L'expression suivante est-elle une liste ?* `(cons (cons 1 2) (cons 3 4))`
2. *Est-ce que la liste* `(1 2 3)` *est une paire ?*
3. *Est-ce que toutes les listes sont des paires ?*

Exercice 2 *Indiquer si les listes suivantes sont bien parenthésées.*

```
(5 (((6 7 9 'r))) 8 12)
(9 (((8 4 5 (4) 25) 3) 7)
(() ())
(56 'p ('K 6) 8 (() (7 6 2)))
```

Exercice 3 *Donner le* `car` *et le* `cdr` *des listes suivantes :*

```
(2)
()
((1 2) 3 4)
(() ())
(1 (2 (3 4 5) 6) 7)
(((8 4) 6 (5 4))(7 8 9))
```

Exercice 4 *Indiquer ce que retournent les expressions suivantes :*

```
1.   (+ ( 2 6) 10)
2.   '(+ ( 2 6) 10)
3.   (car (car  '((5 9) 7 5)))
4.   (car (cdr '((5 9) 7 5)))
5.   (cdr (car '((5 9) 7 5)))
6.   (cdr (cdr '((5 9) 7 5)))
7.   (cons '(a b) '(c d))
8.   (cons (+ 2 5) '(e f))
9.   (cons 'a '(b c d))
10.  (cons 'a '())
11.  (cons (car '(a b c)) (cdr '(a b c)))
12.  (let ((l '(a (b c e))))
          (if (equal? (cdr l) ()) 'yes 'nope))
```

Exercice 5 *Soit l'expression suivante :*

```
(define a (cons '+ (cons (cons ' * (cons (+ 2 6) (cons 3 ()))) (cons (+ 1 3) ()))))
```

1. *Donner la valeur de* `a`
2. *Donner la valeur de* `(cdr a)`
3. *Donner la valeur de* `(cdadr a)`

3 Fonctions de manipulations de listes

Exercice 6 *Écrire une fonction* `premdeux?` *qui indique si un élément se trouve soit en première place soit en seconde place d'une liste. Par exemple :*

```
> (premdeux? 'a '(a b c d))
>  #t
> (premdeux? 'b '(a b c d))
>  #t
> (premdeux? 'c '(a b c d))
>  #f
```

La fonction `list` prend un nombre quelconque d'éléments en paramètre, et construit la liste formée de ces éléments. Par exemple :

```
(list 1 2 3 4 5)        ⇒  (1 2 3 4 5)
(list '1 '(+) '(2 3))   ⇒  (1 (+) (2 3))
```

La fonction `append` prend un nombre quelconque de listes en paramètre, et construit une nouvelle liste qui est la concaténation du contenu de ces listes. Par exemple :

```
(append '(a b) '(c d) '() '((e)))  ⇒  (a b c d (e))
(append '(+) '(2 3))               ⇒  (+ 2 3)
```

Exercice 7 *Pour être sur de ne pas confondre, quelle est la valeur des expressions suivantes ?*

```
(cons '(a b) '(c d))
(list '(a b) '(c d))
(append '(a b) '(c d))
```

Exercice 8 *Donnez la valeur des expressions suivantes (utilisant* `append` *et* `list`*) :*

```
1.  (append (cons 'a (cons 'b ()))  (list (+ 2 3) 'a))
2.  (+ 2 (cadar (list (cons 3 (list 7 (+ 2 4))) (+ 4 5))))
3.  (append (list (cons 'a 'b)) (cons 'c (list 'd)))
4.  (list (cons 'a ()) 'b)
5.  (list (cons 'a (cons 'b ()))  (list (+ 2 3) 'a))
6.  (* 2 (cadar (list (append (cons 3 (list 7 (+ 2 4))) (list (+ 4 5))))))
```

4 Appliquer un traitement à tous les éléments d'une liste

Pour appliquer un opérateur sur tous les éléments d'une liste :

1. On applique l'opérateur sur le `car` de la liste,
2. Tant que la liste n'est pas vide, on appelle récursivement l'opérateur sur le `cdr` de la liste.

Exercice 9 *Écrire une fonction* `length` *qui calcule la longueur d'une liste, c'est-à-dire qui compte son nombre d'éléments.*

Exercice 10 *La fonction* `map` *est la fonction générique pour appliquer une fonction à chaque élément d'une liste. Elle prend en paramètre la fonction à appliquer, et la liste sur laquelle l'appliquer. Exemple :*

```
> (define (plusun n)
    (+ n 1))
> (map plusun '(1 2 3 4 5 6))
> (2 3 4 5 6 7)
```

Écrire une fonction `double` *qui double chaque élément d'une liste de nombres :*

1. *Sans utiliser* `map`
2. *En utilisant* `map`

Programmation applicative – L2
TD 6 : Récursivité terminale

1 Notion de récursivité enveloppée et récursivité terminale

1.1 Récursivité enveloppée

La fonction factorielle $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$ et $0! = 1$

```
(define fact
    (lambda (n)
        (if (= n 0) 1                  ⟵ condition d'arrêt
            (* n (fact (- n 1))))))    ⟵ appel récursif
```

Exercice 1 *Écrire la séquence d'expansion de* `(fact 5)`*. Commenter.*

L'appel récursif `(fact (- n 1))` est dit "**enveloppé**". C'est à dire qu'il fait partie d'un autre calcul, ici la multiplication par `n`.

1.2 La notion d'accumulateur

Il existe un type d'appel récursif qui n'est pas enveloppé. Grâce à un accumulateur on peut sortir l'appel récursif du calcul qui l'enveloppe. L'accumulateur sert à mémoriser le résultat des calculs intermédiaires au cours des différents appels. La fonction `fact` peut être redéfinie à l'aide d'un accumulateur de cette manière :

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

(define fact-iter
   (lambda (n) (fact-acc n 1)))
```

Ce type d'appel récursif est appelé "**terminal**", car l'appel récursif n'est plus imbriqué dans aucun calcul. On parle également de **forme itérative**.

Exercice 2 *Écrire la séquence d'expansion de* `(fact-acc 5 1)`*. Comparer à l'exercice précédent et commenter.*

Exercice 3 *Soit la fonction* `sum-squares-1-n` *qui associe à un entier n la somme des carrés de* 1 *à n :*

$$\texttt{sum-squares-1-n}(n) = 1 + 2^2 + 3^2 + \ldots + n^2$$

1. *Écrire la fonction* `sum-squares-1-n` *en récursivité enveloppée.*
2. *Écrire la fonction* `sum-squares-1-n-iter` *en récursivité terminale.*

Exercice 4 *Inversion d'une liste*

1. *Définir la fonction récursive naïve qui inverse les éléments d'une liste.*
2. *Définir la fonction récursive terminale qui inverse les éléments d'une liste.*

2 Fonctions basiques sur les listes

Exercice 5 *Écrire une fonction* `sommeliste` *qui prend en paramètre une liste d'entiers et qui rend comme résultat la somme des entiers de la liste. En donner une version en récursivité terminale.*

Exercice 6 *Écrire une fonction* `sommerangimpair` *qui prend en paramètre une liste d'entiers et qui fait la somme des entiers à la première, la troisième, la cinquième, ... place dans la liste. Exemple :*

```
> (sommerangimpair  ' ( 3  5  7  4  6  5  4  2 ) )
> 20     ; car  20 =  3 + 7 + 6 + 4
```

En donner une version en récursivité terminale.

3 Fonctions de parcours de listes

Exercice 7 *Donner la fonction* `appartient?` *qui prend un élément et une liste comme arguments, qui retourne true si l'élément appartient à la liste, et false autrement. Exemples :*

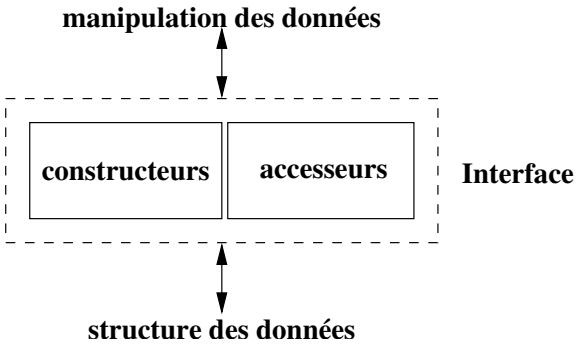
```
> (appartient?  'b  '( a  b  c  d ))
> #t
> (appartient?  'e  '( a  b  c  d ))
> #f
```

Exercice 8 *Écrire une fonction* `debut` *qui prend une liste et un entier* `k` *en paramètre, et qui rend comme résultat la liste composée des* `k` *premiers éléments de la liste. Dans le cas où la liste initiale possède moins de* `k` *éléments, le résultat sera la liste ellemême. Exemples :*

```
> (debut   ' ( 3  5  7  6  4  2 )  4 )
> ( 3  5  7  6 )
> (debut   ' ( 3  5  7  6  4  2 )  8 )
> ( 3  5  7  6  4  2 )
```

Programmation applicative 2 – L2
TD 7 : Abstraire avec des données

1 Utiliser une structure abstraite de données



Cela consiste à isoler [1] la partie qui définit comment des données sont structurées de la partie qui indique comment les utiliser. Il s'agit alors de fournir une **interface** (à l'aide de **constructeurs** et d'**accesseurs**). Pour manipuler des structures de données, il suffit de connaître l'interface associée, sans nécessairement connaître la structure.

Exemple : La structure de données `pair` :

constructeur	accesseurs
`cons` : $exp \times exp \rightarrow pair$ $e_1, e_2 \mapsto$ `(`$\overline{\texttt{e}}_1$ `.` $\overline{\texttt{e}}_2$`)`	`car` : $pair \rightarrow exp$ `(`$\texttt{e}_1$ `.` $\texttt{e}_2$`)` $\mapsto \texttt{e}_1$ `cdr` : $pair \rightarrow exp$ `(`$\texttt{e}_1$ `.` $\texttt{e}_2$`)` $\mapsto \texttt{e}_2$

2 Implantation des nombres complexes

Le nombre complexe $a + i.b$ sera ici implanté par une liste `( a b )`.

Exercice 1 *Écrire les fonctions d'accès, de construction et de test liées aux nombres complexes :*
- *Construire le complexe $a + ib$ par* `(faire_complexe a b )`
- *Accéder aux parties réelles et imaginaires par les fonctions* `partie_reel` *et* `partie_imag`.
- *Tester si un nombre complexe c est un réel pur par l'appel* `(reel? C)`*, s'il est imaginaire pur par* `(imag? C )`.

1. On parle de barrière d'abstraction.

Exercice 2 *Écrire les quatre fonctions* `somme_complexe`, `produit_complexe`, `inverse_complexe` *et* `divise_complexe`.

Exercice 3 *Écrire la fonction qui élève un nombre complexe à une puissance entière (éventuellement négative).*

3 Implantation des polynômes

Un polynôme à une variable est un objet mathématique que l'on définit normalement comme une somme de termes $p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + ... + a_kx^k$ chaque terme étant de la forme a_kx^k où x est la variable du polynôme, k le degré du terme et a_k son coefficient. On appelle ordre d'un polynôme le plus grand degré de ses termes. Une manière classique de représenter ces types de polynômes en Scheme est de les représenter sous la forme d'une liste formée de la suite $a_0, .., a_n$ de tous les coefficients. Un polynôme de degré n est donc représenté ainsi : `(a0 a1 ... an)`

Si un terme est nul, le coefficient vaut 0. Par la suite, toutes les opérations sur des polynômes sont supposées s'appliquer sur des polynômes de même variable.

Exercice 4 Fonctions élémentaires. *Définir*

1. *la fonction* `monome` *telle que* `(monome deg coef)` *retourne le monôme de degré* `deg` *et de coefficient* `coef`.
2. *la fonction* `addpoly` *telle que* `(addpoly p1 p2)` *retourne la somme des deux polynômes* `p1` *et* `p2`. *Exemple :*

```
> (addpoly '( 4  5  0  3  8 )   '( 1  2 ))
> ( 5  3  0  3  8 )
```

3. *la fonction* `multconst` *telle que* `(multconst p c)` *retourne le produit du polynôme* `p` *par la constante* `c`. *Exemple :*

```
> (multconst '( 4  5  0  3  8 )    5  )
> ( 20  25  0  15  40 )
```

4. *la fonction* `multvar` *qui multiplie le polynôme* `p` *en argument par le monôme* `x`. *Ainsi le polynôme* $x^2 + 3x + 4$ *donne* $x^3 + 3x^2 + 4x$.
5. *la fonction* `evalpoly` *telle que* `(evalpoly p v)` *calcule la valeur du polynôme* `p` *en* `v`. *Exemples :*

```
> (evalpoly  '( 4  5 )  2)
> 14
> (evalpoly  '( 3  2  1 )  1)
> 6
```

Exercice 5 Affichage d'un polynôme. *Définir*

1. *la fonction* `monome->string` *telle que* `(monome->string ak k)` *rende en valeur la chaîne de caractères correspondant au monôme* a_kx^k . *Exemples :*

```
> (monome->string  3  7 )
> "3x^7"
> (monome->string  3  1 )
> "3x"
```

2. *la fonction* `polynome->string` *telle que* `(polynome->string p )` *rende en valeur la chaîne de caractères correspondant au polynôme* `p`.

Exercice 6 Multiplication de polynômes. *Définir la fonction* `multpoly` *qui fait le produit de ses deux polynômes passés en argument.*

Notation creuse Dans cet exercice, il est utilisé une nouvelle implantation des polynômes, dite en « notation creuse » : le polynôme $p(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + ... + a_kx^k$ est représenté par la liste des monômes qui composent le polynôme, où un monôme a_kx^k est représenté par un couple `( k . ak )`. L'ordre des monômes est quelconque dans la liste, mais il ne doit pas y avoir deux monômes de même degré. Par exemple, les deux listes `(( 0 . a0 ) ( 1 . a1 )... ( n . an ))` et `( ( n . an ) ... ( 1 . a1 ) ( 0 . a0 ) )` représentent le même polynôme $p(x)$. L'intérêt de cette représentation tient à l'inutilité de mettre les monômes de coefficient nul. Par exemple, le polynôme $x^{27} + 1$ sera représenté par exemple par la liste : `( (0 . 1) (27 . 1))`.

Exercice 7 *Définir*

1. *la fonction* `addmonome` *qui prend en argument un monôme et un polynôme et qui en fait la somme.*
2. *la multiplication de deux polynômes pour cette notation.*
3. *les deux fonctions de conversion* `pleine->creuse` *et* `creuse->pleine` *qui transforment un polynôme en argument d'une notation vers l'autre.*

Programmation applicative – L2
TD 8 : Récursivité sur les arbres

1 Récursivité arborescente : rendre la monnaie

On rappelle l'exemple vu en cours : le nombre de façons de rendre une somme S (par exemple donnée en centimes d'euros) en utilisant n types de pièces (8 types de pièces, de 1, 2, 5, 10, 20, 50, 100 et 200 cts) est :

- Si $S = 0$, il y a 1 solution qui consiste à ne rien faire.
- Sinon si $S < 0$, aucune façon de rendre une somme négative ou si $n = 0$, aucune solution pour rendre une somme non nulle avec aucune pièce
- Sinon : le nombre de façons de changer la même somme S avec $n-1$ types de pièces, plus le nombre de façons de changer la somme $(S - d)$ avec n pièces, d étant la valeur du premier des types de pièces

et dont voici l'implémentation :

```
(define (rendreMonnaie somme nbSortesPieces valeurPiece)
   (define (rendre somme n)
     (cond ((= somme 0) 1)
           ((or (< somme 0) (= n 0)) 0)
           (#t (+ (rendre somme (- n 1)) (rendre (- somme (valeurPiece n)) n)))))
(rendre somme nbSortesPieces))

(define (valeurPiecesEuro piece)
   (cond ((= piece 1) 1)
         ((= piece 2) 2)
         ((= piece 3) 5)
         ((= piece 4) 10)
         ((= piece 5) 20)
         ((= piece 6 ) 50)
         ((= piece 7 ) 100)
         ((= piece 8 ) 200)
         ))

(define (rendreEuro somme)
   (rendreMonnaie somme 8 valeurPiecesEuro))
```

Exemple : il y a 4 façons de rendre 5 centimes, 11 façons de rendre 10 centimes et 4112 façons de rendre 100 centimes, 73682 façons de rendre 200 centimes.

Exercice 1 *Modifier le programme précédent pour qu'il rende la liste des solutions plutôt que leur nombre.*

Exemple : Pour 10 centimes, voici les 11 solutions

```
((1 1 1 1 1 1 1 1 1 1) (1 1 1 1 1 1 1 1 2) (1 1 1 1 1 1 2 2) (1 1 1 1 2 2 2)
 (1 1 2 2 2 2) (2 2 2 2 2) (1 1 1 1 1 5) (1 1 1 2 5) (1 2 2 5) (5 5) (10))
```

2 Récursivité sur les arbres

Il sera question d'arbres colorés dans cette partie. Un arbre rouge-noir est un arbre binaire dont les nœuds sont étiquetés par un couple d'informations :

— un entier

— une couleur, soit rouge, soit noir

La figure 2 représente un arbre rouge-noir (les nœuds grisés représentent les nœuds rouges).

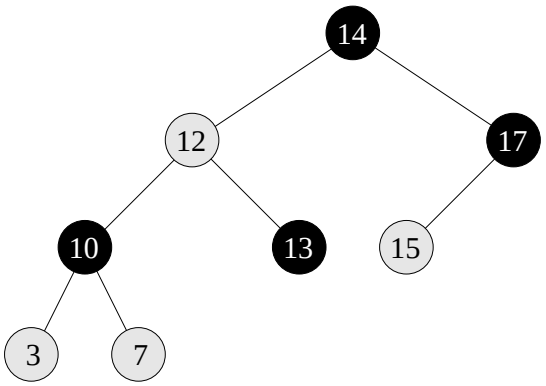


FIGURE 1 – Un arbre rouge-noir

Dans les arbres classiques, les nœuds sont étiquetés par des éléments atomiques comme des entiers par exemple. Il n'y a donc pas besoin de manipuler le contenu de ces nœuds d'une façon spéciale. Ici les nœuds comportent deux informations : l'entier et la couleur. On va donc commencer par s'intéresser à cette "sous-structure de données" de nœuds, avant de traiter la structure de données d'arbre rouge-noir.

2.1 Structure de données Nœud

Voici l'interface souhaitée pour la structure de données Nœud :

Constructeur

`make-noeud int col`	Construit le nœud ayant pour entier `int` et pour couleur `col`

Accesseurs

`get-noeud-int node`	Renvoie la valeur de l'entier du nœud `node`
`get-noeud-col node`	Renvoie la valeur de la couleur du nœud `node`

Prédicats

`noir? node`	Renvoie vrai si la couleur du nœud est noir, faux sinon
`pair? node`	Renvoie vrai si l'entier du nœud est pair, faux sinon

Exercice 2 *Par quelle structure de données choisissez-vous de représenter un nœud ? Expliciter notamment le choix de la représentation de la couleur.*

Exercice 3 *Implémenter l'interface pour la structure de données Nœud.*

2.2 Structure de données ArbreRN

Voici l'interface pour manipuler les arbres rouge-noir :

Constante

`empty-arn`	L'arbre vide

Prédicat

`empty-arn?`	Renvoie Vrai si l'arbre est vide, Faux sinon
`leaf-arn?`	Renvoie Vrai si l'arbre est une feuille, Faux sinon.

Constructeur

`make-arn root left right` Construit un ArbreRN ayant pour racine le nœud `root`, et comme sous-arbres gauche et droite respectivement `left` et `right`

Accesseurs

`get-root-arn arn`	Donne la racine d'un ArbreRN
`get-left-arn arn`	Donne le sous-arbre gauche d'un ArbreRN
`get-right-arn arn`	Donne le sous-arbre droit d'un ArbreRN

NB : Dans la suite on ne demande PAS d'implémenter cette interface, mais seulement de l'utiliser. . .

Exercice 4 *Donnez une procédure* `pairARN` *qui à partir d'un arbre RN, donne un arbre RN de même structure et avec les mêmes valeurs de nœuds mais pour lequel :*

- *un nœud de valeur paire est rouge*
- *un nœud de valeur impaire est noir*

Exercice 5 *Une branche monochrome est une branche d'un arbre RN dont tous les nœuds sont de la même couleur, de la racine jusqu'à la feuille. Donnez une procédure* `monochrome?` *qui renvoie vrai s'il existe une branche monochrome dans l'arbre, et faux sinon.*