Introduction à la programmation
langage SCHEME

V. Berry – Université Montpellier 2

Table des matières

1 Introduction

La programmation (ou l'algorithmique) consiste à exprimer une suite d'actions qui mène à un but précis. L'expression de cette suite d'actions se fait dans un langage très simple et en général très pauvre. L'ordinateur n'est en aucun cas capable de décision et de raisonnement "intelligent" : toutes les situations possibles doivent être prévues par le programmeur.

L'intérêt de l'utilisation d'un ordinateur pour la résolution d'un problème est exclusivement la vitesse. L'humain (le programmeur) lui donne la suite d'instructions à exécuter pour arrivée à la solution de façon correcte en un temps raisonnable.

La résolution d'un problème en utilisant un ordinateur consiste schématiquement en deux étapes :

- l'activité humaine (programmation) qui part d'un problème réel exprimé en langage naturel et qui le modélise sous forme symbolique (ou numérique) en vue de le faire résoudre par un ordinateur,
- l'activité de l'ordinateur (compilation) qui transforme le modèle en un langage de 0 et de 1 compréhensible par la machine.

En cinquante ans d'évolution de l'informatique, la partie compilation n'a fait que croître, libérant le programmeur de la plupart des tâches automatisables tout en permettant de s'exprimer dans un langage de plus en plus proche du langage naturel et de réutiliser des parties déjà programmées (éventuellement sur d'autres ordinateurs : portabilité). Ainsi des langages de haut-niveau permettent d'exprimer de façon concise des actions complexes, l'ordinateur se chargeant lui-même de leur traduction.

Cette évolution n'a fait que réduire la distance séparant le langage naturel du langage compris par l'ordinateur et a entrainé la forminable démocratisation de l'informatique qui rend l'ordinateur accessible à tous.

La **programmation** fait partie de la culture de l'ingénieur et du scientifique. Il faut être capable de programmer pour trouver des solutions adaptées à ses problèmes. Il ne faut pas imaginer que les logiciels vendus dans le commerce puissent résoudre tous les problèmes. Ils apportent en général des réponses assez simples à des problèmes précis et leur coût ne les rend rentables que s'ils sont utilisés de façon répétitive.

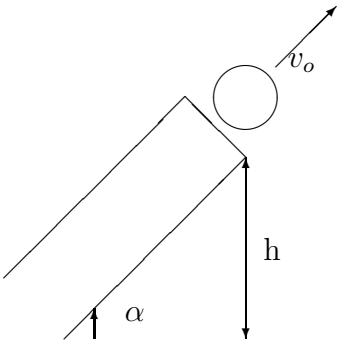
La démarche

Document	**Langage Support**	**Objets manipulés**
Descrition informelle	Langage naturel	Objets manipulés
Description formelle QUOI ?	Langage de spécification rigoureux, sans ambiguité, pauvre	Modélisation Formules Mathématiques
Programmation COMMENT ?	Langage de Programmation	valeurs manipulées par le programme
code machine	Langage machine (ou assembleur)	Adresses mémoire

Exemple :

Description Informelle :

Prenons un canon incliné d'un angle α par rapport à l'horizontale, l'extrémité du canon est à une hauteur h et les obus ont une vitesse v_0 à la sortie du canon. On veut savoir à quelle distance l'obus touchera le sol.



Description Formelle

On précise les hypothèses : on néglige le frottement de l'air, le sol est horizontal et les obus sont soumis à la gravition g qui est supposée constante.

Ainsi l'équition de la trajectoire est donnée par (cf cours de physique) :

$$y \;=\; \frac{-g}{2cos(\alpha)v_0}x^2 \;+\; tg(\alpha)x \;+\; h \;=\; P(x)$$

Il faut donc trouver la racine positive de P(x).

Comment :

Par une méthode spécifique : résolution d'une équation du second degré.

Description :
– calcul du discriminant
– calcul des racines
– donner comme résultat la racine positive
Par une méthode plus générale : la méthode de Newton (avantage : le programme pourra être utilisé même avec des balistiques plus complexes).

Remarque Le programmeur devra prévoir les cas où le programme n'est pas défini (par exemple si h est négatif).

2 Les langages de programmation

Un langage de programmation est en fait un modèle de calcul, c'est-à-dire une façon de décrire la réalisation de calcul.

Il est défini par :

1. une façon de représenter les nombres et les autres objets manipulés,
2. un ensemble d'actions élémentaires (opérations) capables de modifier ces objets,
3. un système d'ordonnancement et d'exécution de ces actions.

De nombreux langages de programmation et modèles de calcul ont été proposés :

– machine de turing (1936)
– fonctions récursives (Kleene)
– langages Pascal, C, Fortran, Lisp, Scheme ...

Proposition : Ces langages de programmation sont tous équivalents. Chaque programme écrit dans l'un quelconque de ces langages peut être traduit dans un autre de ces langages. Ils ont donc tous la même puissance d'expression.

La thèse de CHURCH-TURING conjecture que toute méthode explicable (i.e. non intuitive) inventée par l'homme pour effectuer un certain calcul peut-être traduite dans ces langages.

Comme s'il y avait une limite à l'intelligence (mathématique) humaine et tout ce que sait calculer l'homme peut-être calculé par ordinateur.

Les langages ne se différencie donc pas par leur puissance d'expression mais par leur distance par rapport au langage naturel ou par leur adaptation à la résolution de certaines classes de problèmes.

Classiquement, on les regroupe en 3 grandes familles :

– les langages impératifs : Fortan, Cobol, pascal, Ada, C ...
– les langages fonctionnels : LISP, CaML, **Scheme** ...
– les langages logiques ou déclaratifs : Prolog.

Pour ce cours, on a choisit Scheme. Pourquoi Scheme ?

Parce qu'il est un langage de haut niveau qui possède une syntaxe simple loin des détails techniques et donc on peut se concentrer sur les concepts essentiels de la programmation. Il peut être utilisé rapidement par un débutant pour faire des programmes intéressants.

Il est interactif : en quelques secondes on écrit un programme et on obtient le résultat. Chaque "phrase" est écrite et testée au fur et à mesure, ce qui permet de détecter rapidement les erreurs. En langage impératif, il faut souvent écrire de nombreuses lignes et avoir compris de nombreux concepts pour avoir un ensemble cohérent et testable. Le risque d'erreurs et le temps de correction peuvent alors être importants et décourageants pour les débutants.

Le langage est libre et écrit en dehors de toute logique commerciale (gratuit). Vous pouvez l'installer sur votre ordinateur sans démarche commerciale ou administrative.

Il assure une égalité et une certaine homogénéité entre les étudiants.

Objection ? : la plupart des programmes industriels sont en impératif (Cobol pour la gestion, Fortran pour le calcul scientifique ...)

Rejetée ! : maintenant la plupart des applications sont développées dans des langages spécifiques ou des variantes des langages et on ne peut pas tous les apprendre.

Il n'est pas raisonnable dans la discipline informatique de former les étudiants exclusivement sur les produits du présent. L'important est de maîtriser les concepts fondamentaux pour s'adapter à l'évolution de la discipline informatique. Scheme

permet une mise en pratique très rapide de ces concepts.

Plus tard dans votre cursus (2eme ou 3eme cycle), vous étudierez des lges de programmation plus proches de la pratique du monde professionnel et dépendant de la branche dans laquelle vous vous serez engagés (infomatique,électronique,physique,math,biologie,e et typiques de cette époque future. Quelque soit ces lges, vous re-investirez les connaissances fondamentales que vous aller acquerir cette annee.

Objectif du cours

Maîtriser les concepts importants de la programmation :

- Décomposition d'un problème en sous-problèmes
- Récursivité
- Définition de "boites noires" d'abstraction

Savoir bien écrire les algorithmes et les programmes :

- être exhaustif : envisager tous les cas du problème à résoudre,
- être non ambigü : à tout moment, le choix de la prochaine étape à réaliser doit être parfaitement défini,
- donner le résultat en temps fini (pas de bouclage),
- être exact : le programme résoud bien le problème initial de façon juste,
- essayer d'être efficace (rapidité, utilisation raisonnable de la mémoire ...),
- être clair (réutilisation par un tiers, relecture)

3 Les bases du langage SCHEME

3.1 La boucle intéractive de SCHEME

Toute fonction ne peut être évalueé q'une fois que ses paramètres ont été évalués. L'évaluation d'une fonction scheme se déroule suivant un arbre d'évaluation (cf figure).

3.2 Expressions de base

3.2.1 Booléens - valeurs logiques

Constantes prédéfinies VRAI et FAUX : **#t** et **#f**

c'est ce qu'on utilisera dans les cas ``rés=oui'' ou ``rés=non'', cf algorithmes du cours 1.

Opérateurs : **and, or, not** (donner les tables de vértié).

```
> (and #t #f)          ; attention, notation prefixee
#f
```

3.2.2 Arithméthique

Pour SCHEME , un **nombre** est une suite de chiffres débutant éventuellement par un signe + ou -, pouvant comporter un point décimal (.) et un exposant (e) :

```
> 9
9
> 10e2
1000.0
> -1.2e2
-120.0
```

La plupart des langages distinguent les **nombres entiers** des **nombres réels**. Ces derniers sont représentés en machine par des "nombres flottants" possédant une précision limitée (comment coder π ?). Donc la représentation machine d'un nb réel n'en est parfois qu'une approximation.

EXEMPLE : le prix TTC d'une boite de 10 disquettes notée 69 F HT :

$$Prix\_TTC = Prix\_HT * (1 + \frac{TVA}{100})$$

```
> ( * 69 ( + 1 ( / 18.6 100 )))
81.834
```

et le vendeur affichera souvent 82 F !!!!

Les **opérateurs de comparaison** : <, >, =, <= , >=

Autres opérateurs :

exp log *(en base 10)* **abs cos sin tan asin acos atan**

```
> (exp 1)
2.718281828459045
```

Exemple : comment obtenir le log a base 2 d'un nombre x ?

```
(/ (log x) (log 2))
```

floor, ceiling, round, truncate, quotient, remainder, modulo

(note : pour **round** lorsqu'il y a 2 candidats, c'est l'entier pair qui est choisi : (round 3.5) et (round 4.5) donnent 4).

Exemple : comment obtenir la partie décimale de 5.34 ?

```
>(- 5.34 (truncate 5.34))
0.3399999999999999
```

i.e., 0.34 aux erreurs de représentation près !!!
Est-ce ok pour les nombres négatifs ? Comment procéder ?

```
(abs (- x (truncate x)))
```

Exemple : comment obtenir le nombre de tours d'un tournoi pour 14 joueurs par la méthode 4 vue précédemment ?

```
> (ceiling (/ (log 14) (log 2)))
4.0
```

3.2.3 Notion de type

Chaque information enregistrée en mémoire (connue d'un programme) a un **type** particulier (*booléen* et *nombre* sont deux exemples de type), qui influence la façon dont elle est perçue par le programme, son codage en mémoire et les fonctions qu'on peut lui appliquer (cf ci-dessus).

Insister un peu là-dessus mais pas encore trop.

3.3 Expressions spéciales

Il s'agit d'expressions évaluées de façon spéciale par SCHEME et ayant parfois un "effet de bord" autre que le résultat qu'elles renvoient.

3.3.1 Séquences d'expressions

Normalement, lorsque SCHEME est ds l'état "lecture" de sa boucle intéractive, il attend la saisie d'*une* fonction pour ensuite l'évaluer. Pour demander à SCHEME d'évaluer plusieurs fonctions à la suite :

`(`**sequence** exp_1 exp_2 ... exp_n`)` ou `(`**begin** exp_1 exp_2 ... exp_n`)`.

Evaluation : chq exp_i est évaluée ds l'ordre de la séquence. exp_i peut modifier le contexte courant (variables), exp_{i+1} est évaluée dans ce nouveau contexte. La valeur finale renvoyée est celle de la dernière expression. EXEMPLE :

```
>(sequence 1 (+ 1 3) (* 2 4))
```

L'évaluation procède ainsi :
– évaluer `1` ce qui donne 1
– évaluer `(+ 1 3)` ce qui donne 4
– évaluer `(* 2 4)` ce qui donne 8
finalement la dernière valeur calculée,*i.e.*, 8, est renvoyé à l'utilisateur.

3.3.2 Conditionnelles

Pour écrire les algorithmes du cours précédemment nous svt eu recours aux formules de type si-alors-sinon. En SCHEME la syntaxe est :

(**if** < *condition* > < *clause alors* > < *clause sinon* >)

Evaluation par SCHEME :

1. évaluation de < *condition* > ; soit v sa valeur.
2. Si v est égale à `#t` (VRAI) alors < *clause alors* > est évaluée et sa valeur est renvoyée.
3. Si v est égale à `#f` (FAUX) alors < *clause sinon* > est évaluée et sa valeur est renvoyée.

EXEMPLE : ¿(define entre (a b c) (and (¡ a c) (¿ a b)))

```
>(if (= 0 (- 1 1)) 2 (+ 1 3))
2
>(if (< 0 0) 1 4)
4
```

Le **cond** : il s'agit d'une extention de la forme `if`. Sa syntaxe est :

$$\begin{array}{llllll} (\textbf{cond} & (test_1 & exp_{11} & \dots & exp_{1n}) \\ & (test_2 & exp_{21} & \dots & exp_{2n}) \\ & & \dots & \dots & \\ & (test_m & exp_{m1} & \dots & exp_{mn}) \\ & (\textbf{else} & exp_1 & \dots & exp_n)) \end{array}$$

Evaluation : il s'agit de `if` imbriqués :

1. évaluation de $test_1$; soit v sa valeur.
2. Si $v_1 = \#t$ alors évaluer `(sequence` $exp_{11} \dots exp_{1n}$`)`.
3. Si $v_1 = \#f$ alors évaluer $test_2$, soit v_2 sa valeur.
4. Si $v_2 = \#t$ (et $v_1 = \#f$) alors évaluer `(sequence` $exp_{21} \dots exp_{2n}$`)`.
5. ...
6. si $v_1 = v_2 = \dots = v_m = \#f$ alors évaluer `(sequence` $exp_1 \dots exp_n$`)`.

EXEMPLE :

```
> (cond ((= x 0) 1)
        ((= x 1) 2)
        ((= x 2) 3)
        (else    0))
```

Si x vaut 1, le résultat est 2, si x vaut -3, le résultat est 0.

3.3.3 Définition de nouvelles fonctions - `define`

On peut enrichir le langage en définissant de nouvelles fonctions :

$$(\textbf{define}\ (NomFonction\ param_1 \ldots param_n)\ (exp_1 \ldots exp_m))$$

où les exp_i sont des expressions et $NomFonction, param_1 \ldots param_n$ sont des noms.

EXEMPLE :

$$\begin{aligned} f : x \ &\rightarrow \ y = x \text{ si } x > 0 \\ &\quad\ \ y = 0 \text{ autrement} \end{aligned}$$

```
>(define (f x) (if (< x 0) 0 x))
f
```

L'évaluation de cette expression spéciale **a pour effet de lier** $NomFonction$ au "corps" de la fonction, équivalent à `(sequence` $exp_1 \ldots exp_m$`)`. La **valeur renvoyée** est le nom de la fonction définie.

Les **paramètres** $(param_1 \ldots param_n)$ sont généralement utilisés dans le corps de la fonction et prennent la valeur donnée lors de l'appel de la fonction :

```
> (f 2)
2
> (f 0)
0
> (f -3)
0
```

x paramètre formel 2 paramètre formel	lors de l'appel de la fonction, le paramètre réel remplace le paramètre formel pour l'évaluation du corps de la fonction.

Le nom des paramètres formels n'a pas de conséquence sur l'évaluation de la fonction, donc sur son sens.

EXEMPLE : Comment écrire la fonction donnant le min (ou le max) de deux nombres ?

SOLUTION :

```
>(define (min x y) (if (< x y) x y))
min
> (min 2 3)
2
> (min 3 2)
2
```

EXERCICE : Ecrire une fonction `Prix_TTC` comportant un paramètre appelé `HT`.

EXERCICE : Ecrire la fonction `(entre a b c)` qui perme de savoir si $b < a < c$.

SOLUTION : au choix :

```
>(define entre (a b c) (if (> a b) (< a c) #f))
>(define entre (a b c) (if (and (< a c) (> a b)) #t #f))
>(define entre (a b c) (and (< a c) (> a b)))
```

Composition de fonctions : en math : $f \, o \, g = g(f(x))$, cela se traduit très naturellement en SCHEME : `(g (f x))`.

Commentaires : ds la def de fonctions complexes, il sera de bon ton d'indiquer par qq commentaires l'utilité de différentes parties de la fonction. Un commentaire peut etre place après un symbole "`;`" . Dès qu'il rencontre ce symbole, SCHEME considère la suite de la ligne comme un commentaire et ne l'évalue pas.

> *L'écriture de programmes* SCHEME *consistera principalement à définir de nouvelles fonctions.*

3.3.4 Définition de variables - `define`

La fonction **define** permet aussi de **définir des variables** (cf MR01, MR02,... des calculatrices) mais avec des noms bien plus évocateurs. Un nom de variable est une séquence de cars autres que ( ) " ; [ ] { } |

```
>(define un 1)
un
> (+ un 1)
2
```

La définition d'une variable consiste à attribuer un nom à une certaine zone mémoire contenant la valeur de la variable. Cette valeur peut bien sûr changer (d'où le nom de variable).

L'utilisation de variables permet donc de représenter le résultat de calculs complexes par des noms simples. Ceci nous permet de **décomposer les algorithmes**, les rendant ainsi **plus lisibles** : comparez

```
>(* (* 2 2) 3.14)
12.56
```

qui n'est pas très explicite au 1er regard et

```
>(define pi 3.14)
>(define (surface_cercle rayon) (* pi (* rayon rayon)))
>(surface_cercle 2)
12.56
```

pour calculer la surface d'un cercle de rayon 2.

Si l'on veut maintenant calculer la surface d'un cercle de rayon 3, on peut simplement taper

```
>(surface_cercle 3)
28.26
```

La définition de variables et de fonctions nous **évite aussi d'effectuer plusieurs fois certains calculs** (complexes) : calcul de δ lorsqu'on détermine les solutions de $x^2 - x + 3 = 0$.
L'évaluation d'une variable consiste à renvoyer sa valeur courante (la valeur d'une variable change selon le contexte) :

```
>(define x 1)
x
>x
1
>(+ 1 x)
2
>(define x 4)
x
>(+ 1 x)
5
```

$\Rightarrow$ la même expression évaluée à deux moments distincts n'a pas forcément la même valeur ! En effet,

L'évaluation de toute expression se situe dans un contexte.

Ds le 1er cas, évaluer `(+ 1 x)` revient à évaluer `(+ 1 1)`, et ds le 2ème cas revient à évaluer `(+ 1 4)`.

Le **contexte** est composé de l'ensemble des définitions connues par le système au moment de l'évaluation, *i.e.*, l'ensemble des fonctions définies mais aussi l'ensemble des variables et des valeurs liées à ces variables.

Dans le **cas particulier des fonctions**, l'évaluation s'effectue dans un **contexte modifié**, héritant des définitions (fonctions et variables) du contexte général auquel sont **ajoutées les définitions** de variables dûes à la liaison **des paramètres** formels aux paramètres réels. Ce mini-contexte "disparaît" dès que l'évaluation de la fonction est terminée.

NOTE : il arrive parfois que certaines variables soient **masquées** par des paramètres du même nom :

```
>(define x 1)
```

FIG. 1 – Arbre d'évaluation détaillant les différents contextes et sous-contextes

```
>(define (f x) (+ x 1))
>(f 3)
4
```

Exercice : dessiner les cases mémoires et leur évolution pour le programme ci-dessus.

3.3.5 Forme spéciale `let` - variables et leurs portées

Permet d'associer des valeurs à des noms pendant l'évaluation d'une suite d'expressions :

(let $((n_1\ v_1)\ (n_2\ v_2)$ ... $exp_1\ exp_2$ ...exp_m)

En fait similaire à l'attribution de valeurs aux paramètres d'une fonction lors de son exécution, sauf qu'ici on ne passer pas par une fonction (on n'entend pas réutiliser la suite de caluls effectués).
EXEMPLE :

```
>(let ((x 1) (y 2)) (+ x y)
2
>(let ((x 2) (y 4))
    (+ (let ((x 1)) (+ x y))
       x))
7                          ; masquage de la var x ds le 2eme let!!!!
```

Note : `let*` variante de `let`.

3.4 Les chaines de caractères

Vocabulaire : "chaine" de cars = suite de caractères.

L'information manipulée (et non pas son nom) n'est plus numérique mais de type alphabétique.

3.4.1 Information de typage

Les variables n'ont pas toutes le même type. Selon leur type elles sont représentées différemment en mémoire. Il existe des fonctions permettant de savoir si une variable donnée est de tel ou tel type :

```
>(number? 3)
\#t
>(integer? (round 4.5))
\#t
>(boolean? (> 2 3))
\#t
>(string? "toto")
#t
```

4 La récursivité - notions de compx espace et temps

Méthodes de **décomposition de pbms** :

1. sous-pbm de nature différente que l'on résoud séparemment
2. récursivité : sous-pbm de même nature mais plus faciles (plus petits en taille)

Un algo est dit **récursif** si, dans sa définition, il s'appuye sur lui-même, *i.e.*, si le nom de la fonction qu'on est en train de définir intervient aussi dans la liste des instructions de cette fonction.

Il s'agit d'un type d'algo très classique en informatique, et ayant de nombreuses connexions avec les math :
– suites récurrentes : $u_0 = 12$, $u_n = 3.u_{n-1} + 12$.
– définition par récurrence : depuis le nombre 0 et la fonction $succ : nb \rightarrow nb + 1$, on peut définir ce qu'est un entier : soit 0, soit le successeur d'un entier.
– preuve par récurrence d'une propriété $P(n)$: pour montrer que $P(n)$ est vraie $\forall n \geq n_0$, on montre que P est vraie au rang initial n_0, puis on montre que si P est vraie à un rang $n-1$ quelconque, alors elle est aussi vraie au rang n. (exs : nb de parties d'un ensemble à n elem $= 2^n$).

L'écriture d'une fonction récursive suit la démarche d'une preuve par récurrence (ou par induction) : si on est ds un cas simple (rang initial), on résoud le pbm en qq instructions, sinon pour un rang n qcq, on résoud le pbm au rang $n-1$ puis on fait les ajustements nécessaires pour obtenir une solution du pbm au rang n.

EXEMPLE : la fonction factorielle, donnée généralement comme

$$\begin{aligned} 0! &= 1 \\ n! &= n.(n-1).(n-2)\ldots 1 = \prod_{i=1}^{n} i \end{aligned}$$

On peut reformuler ça de façon récursive :

$$\begin{aligned} 0! &= 1 \\ n! &= n.(n-1)! \end{aligned}$$

On peut interpréter cette définition comme

> "pour résoudre $n!$ il suffit de calculer $(n-1)!$ (*i.e.*, résoudre pbm au rang $n-1$) et de multiplier par n, sauf ds le cas trivial de $n=0$ où il suffit de renvoyer 0."

Cette définition est suffisante pour calculer $n!$ pour n'importe quel n entier, par ex, calculons 4! :

$$\begin{aligned} 4! &= 4.3! \\ &= 4.3.2! \\ &= 4.3.2.1! \\ &= 4.3.2.1.0! \\ &= 4.3.2.1.1 \\ &= 24 \end{aligned}$$

L'écriture de la fonction SCHEME suit en tout point la définition récursive :

```
(define (fact n)
        (if (= n 0) 1                          ; clause d'arret
                    (* n (fact (- n 1)))))     ; cas general
```

Plus généralement, toute fonction récursive écrite correctement **vérifie les axiomes** :
– il doit exister au moins une clause qui n'est pas récursive (clause d'*arrêt*).
– le calcul doit converger (*i.e.*, toute clause générale doit aboutir en un tps fini à une clause d'arrêt).

4.1 Récursivité terminale - notion de complexité en espace

Une autre façon de procéder pour calculer `fact(n)` est de multiplier 1 par 2, puis par 3, par 4, etc jusqu'à n. En pratique pour décrire cet algorithme, nous avons besoin
– d'un compteur variant de 1 à n indiquant quel est le prochain nb par lequel multiplier notre produit courant
– d'une variable retenant la valeur du produit courant.
D'une étape à l'autre nous devons réaliser les opérations suivantes :

$$\begin{aligned} \mathit{produit} &\leftarrow \mathit{produit} \times \mathit{compteur} \\ \mathit{compteur} &\leftarrow \mathit{compteur} + 1 \end{aligned}$$

Question : Quelle est la clause d'arrêt (qd l'algo s'arrête-t-il) **?**

$\Rightarrow$ quand on vient de multiplier par n, *i.e.*, quand $\mathit{compteur} > n$,

Question : Que renvoie-t-on ds ce cas **?**

$\Rightarrow$ la valeur de produit.

Ecrivons ça en SCHEME :

```
(define (fact n)
        (fact2 1 1 n))

(define (fact2 prod cpt n)
        (if (> cpt n) prod
                      (fact2 (* prod cpt) (+ cpt 1) n)))
```

Note : on utilise une 2eme fonction parce que l'utilisation de `prod` et `cpt` sont des détails spécifiques à l'algo ne correspondant pas à des données du pbm, l'utilisateur n'a pas à les connaître. On conserve ainsi une interface commune avec le 1er algo.

Dans les deux cas il s'agit de programmes récursifs, demandant tous les deux un **nb d'opérations, donc un temps d'exécution, proportionnel à** n.

Toutefois l'un est plus intéressant que l'autre, car il **consomme moins de mémoire**. En effet, comparons l'évaluation de ces deux algorithmes sur un même exemple, `fact(4)`.

Pour l'algo 1 (**redonner prog**), la résolution du pbm initial **demande d'abord** la résolution d'un sous-pbm, qui demande lui-même la résolution d'un sous-pbm, etc :

```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 (* 1 (fact 0)))))
```

on arrive maintenant au moment où le test `(= n 0)` est vrai, *i.e.*, on n'a plus besoin de résoudre de sous-pbm. Le résultat de `(fac 0)` est 1, ce qui permet d'achever les calculs :

```
(* 4 (* 3 (* 2 (* 1 1)))))
(* 4 (* 3 (* 2 1)))
(* 4 (* 3 2))
(* 4 6)
24
```

Ces qq lignes montrent l'évolution des opérations pour résoudre le pbm initial : alors qu'**une opération est "active"**, (calcul de `(fact 3)` puis de `(fact 2)`,..., `(* 1 1)`,`(* 2 1)`,....), **d'autres opérations sont "inactives", en attente**,

L'info correspondant à ces operations est **mémorisée** par le système dans une structure particulière, appelée **une pile**. Le nb d'info à mémoriser est proportionnel à n (nb max d'info mémorisées au moment où test `if` est vrai).

Pour l'algo 2 **redonner prog**),

```
(fact 4)
(fact2 1 1 4)
(fact2 1 2 4)
```

```
(fact2 2 3 4)
(fact2 6 4 4)
(fact2 24 5 4)
24
```

Dans ce cas, d'une étape à l'autre du calcul, il n'y a **aucune info à retenir, aucune opération en attente de la résolution d'un sous-pbm**, donc pas besoin de pile en mémoire. En fait, la **taille de la zone mémoire** nécessaire à l'évaluation du programme est **constante** (valeurs de `cpt, prod, n` mises à jours).

L'algo 2 est donc plus intéressant que l'algo 1, car il **consomme moins de mémoire**.

Quelle caractéristique le distingue syntaxiquement de l'autre algo ?

⇒ le fait que **l'appel récursif** de la fonction **est la dernière instruction à évaluer** dans la fonction et que **son résultat n'est impliquée dans aucun calcul**, *i.e.*, la valeur renvoyée par l'évaluation du sous-pbm est la valeur renvoyée par la fonction.

Cette forme de récursivité s'appelle **récursivité terminale**.

Propriété : Tout algorithme/prog écrit au moyen d'instructions de boucles, (`for, while`, etc, cf lges impératifs) possède un algorithme récursif équivalent.

algorithme de (fact n) utilisant une "boucle" :

```
prod = 1
pour i = 1 \‘a n faire :
        prod = prod . cpt
        cpt = cpt + 1
renvoyer prod
```

Propriété : Pour tout algorithme (écrit récursivement), il existe généralement un algorithme récursif équivalent sous forme "terminale".

⇒ Equivalence de puissance d'expression.

L'écriture d'un tel algorithme nécessite souvent l'**utilisation de paramètres supplémentaires**, par ex `prod, cpt` ds `(fact2 n)`.

Note : de même que les fonctions $\prod_{i=0}^{n} f(n)$ comme $fact(n)$, on peut très facilement **calculer des séries** :

$$S_n = \sum_{i=0}^{n} u_i$$

où u_i est le ième terme d'une suite (u_n) donnée. En effet, $S_0 = u_0$ et $S_n = S_{n-1} + u_n$.

EXERCICE : - **paradoxe de la flèche de Zénon** (ou d'Achille et la tortue) - écrivez une fonction SCHEME donnant le nième terme de la série S_n basée sur la suite

$$\begin{aligned} u_0 &= 0 \\ u_n &= \frac{1}{2^n} \end{aligned}$$

EXEMPLE :

$$\begin{aligned} S_0 &= 0 \\ S_1 &= \frac{1}{2} = 0,5 \\ S_2 &= \frac{1}{2} + \frac{1}{4} = 0,75 \\ \dots & \\ S_{20} &= \frac{1}{2} + \ldots + \frac{1}{2^{20}} = 0,999999 \end{aligned}$$

> Vision du temps sous forme d'intervalles, au cours desquels Achille refait une partie de son retard. Supposons que la tortue soit partie avant ds la course et considérons le moment où Achille-au-pied-léger s'élance. La tortue a une vitesse tellement faible par rapport à celle d'Achille, qu'on peut la considérer nulle, et cependant montrer qu'Achille ne la rattrapera jms : dans le prochain intervalle de temps, Achille parcours la moitie de la distance qui le separe de la tortue, mais il reste la moitie de la distance à parcourir, dans l'intervalle de temps suivant, il parcourt encore la moitie de la distance qui le sépare de cette satanée tortue, mais il lui reste toujours une certaine distance à parcourir si petite soit-elle pour rattraper la tortue, et ainsi de suite à l'infini, **bref, selon ce raisonnement, il ne rattrape jms la tortue**. Et encore, on a considéré que la vitesse de la tortue était nulle!!!!

En fait, $\sum \frac{1}{2^i}$ tend vers 1 mais ne l'atteint jamais.

SOLUTION :

```
(define (fleche n)
     (  if (= n 1) 1
                   (+ (/ 1 (puissance 2 n)) (fleche (- n 1)))))

>(fleche 1)
0.5
>(fleche 2)
0.75
>(fleche 20)
0.999999
```

La fonction `puissance` est vue en TD.

4.2 Notion de complexité en temps

Un autre aspect permettant de **comparer deux algorithmes** résolvant un même pbm est celui du temps de calcul nécessaire à leurs évaluations respectives.

Tps d'exécution = fonction de nbx param : vitesse de l'ordinateur, efficacité de l'interpréteur SCHEME .

Cpdt, indépendemment de toute considération de machine ou de lge, il existe un paramètre fondamental de chq programme, qui détermine sa rapidité d'exécution : sa **"complexité" en tps calcul**. Il s'agit de la proportions d'opérations/d'instructions qu'il nécessite pour résoudre un pbm de taille n donnée.

EXEMPLE : soit la suite des nb de Fibonacci, où chq nb est la somme des 2 précédents :

$$Fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{sinon} \end{cases}$$

Note : $fib(n)$ est le nb entier le plus proche de $\Phi^n/5$, où $\Phi = (1+\sqrt{5})/2 \approx 1,6180$ est le nb d'or.

ALGO 2 POUR $fib(n)$:

la définition est récursive et se traduit facilement en un prog SCHEME :

```
(define (fib n)
     (cond ((= n 0) 0)
           ((= n 1) 1)
           (else (+ (fib (- n 1))
                    (fib (- n 2))))))
```

Regardons la structure de l'évaluation engendrée par les appels récursifs sur un exemple, `(fib 5)` :

FIG. 2 – Arbre des appels récursifs. - Sussman et al p.32

La structure de ces appels ressemble à un arbre et l'on parle ainsi de **"récursivité en arbre"**. Syntaxiquement, cela se traduit par le fait que pour résoudre un pbm de taille n, la fonction a besoin de résoudre plusieurs (ici 2) sous-pbms de même nature.

C'est un très bel exemple de récursivité, mais **profondemment inéfficace, pouvez-vous deviner pourquoi ?**

$\Rightarrow$ pour certaines valeurs k, le processus **calcule plusieurs fois** `(fib k)`, ainsi `(fib 0)` est calculé 3 fois, `(fib 1)` 5 fois, ... et tout le sous-arb correspondant au calcul de `(fib 3)` est dupliqué!!!

Il est possible de montrer que le nb de feuilles ds l'arb (`fib 0` ou `fib 1`) est exponentiel en fonction de n. Donc le nb d'opérations nécessaires à l'évaluation de `(fib n)` est exponentiel en n, ce qui réduit bcp l'utilité de ce prog en raison des énormes ressources en tps qu'il nécessite (cf plus tard tableau des tps/compx).

En ce qui concerne l'**espace mémoire** utilisé par l'évaluation de ce programme, il est linéaire en fonction de n : à un moment donné il y a **au pire, de l'ordre de** n opérations en attente : une addition pour `(fib n)`, une pour `(fib n-1)`,..., une pour `(fib 2)`. En fait la liste des opérations en attente à un moment donné est **proportionnelle à la longueur de la branche** courante (du noeud courant jusqu'à la racine.

ALGO 2 POUR $fib(n)$:

Il est aussi possible de calculer $fib(n)$ par un algorithme dont la **récursivité est terminale**. Celui-ci procède en calculant itérativement les termes de la suite en partant des plus simples.

En effet, depuis $fib(k-2)$ et $fib(k-1)$ on obtient simplement $fib(k)$, puis en additionnant $fib(k-1)$ et $fib(k)$ on obtient $fib(k+1)$, etc jusqu'à obtenir $fib(n)$.

Question : De combien de variables avons-nous besoin à chq étape de ce calcul **?**

$\Rightarrow$ de 2 vars a et b pr retenir les deux valeurs précédentes ($a < b$), et d'une, *cpt* pour savoir qd on doit s'arrêter, *i.e.*, qd on a atteint $fib(n)$.

Question : comment modifie-t-on ces variables à chq étape **?**

$\Rightarrow$ simultanément,

$$b \leftarrow a + b \quad \text{et} \quad a \leftarrow b$$

Question : avec quelles valeurs de a et b initier l'algorithme **?**

$\Rightarrow$ avec 0 et 1.

EXERCICE : écrivez l'algorithme correspondant.

SOLUTION :

```
(define (fib n)
        (fib2 1 0 n))
(define (fib2 a b cpt)
        (if (= cpt 0) b
                      (fib2 (+ a b) a (- cpt 1))))
```

EXERCICE : Quel est en gros le nb d'opérations effectuées lors de l'évaluation de $fib(n)$ par cet algo (en fonction de n) ? Qu'en est-il de la taille mémoire nécessaire ?
SOLUTION : $O(n)$ en temps.
$O(1)$ en espace : récursivité terminale !

4.3 Intérêt de la notion de complexité

On a vu précédemment, que les algorithmes pouvaient être comparés en fonction des **ressources qu'ils consomment** : temps calcul et espace mémoire. Pour comparer l'efficicaté des algos on mesure leur **complexité**. Celle-ci est mesurée en fonction d'un **paramètre** n, caractérisant les données du pbm ou leur taille.

En pratique, on se contente de **connaître approximativement** le nb d'opérations ou de cases mémoires, connaître un **ordre de grandeur** nous suffira.

Par exemple, le calcul de $fact(n)$ par l'algo 1 nécessitait de l'ordre de n opérations : on notera $\mathbf{O(n)}$. On n'est pas **à une constante additive ou multiplicative près**.

EXEMPLE : d'ordres de gdeur : $O(1)$, $O(\log n)$, $O(n)$, $O(n^2)$,..., $O(2^n)$,...

> On cherchera à produire des algos dont la consommation en tps et en espace soit d'ordres de gdeurs les plus petits possibles.

Si vous pensez qu'il n'est pas très important de s'intéresser à la compx des algos et que la prochaine génération d'ordinateurs suffira à rendre utile tout algorithme, même ceux de compx exponentielle, le tableau suivant peut vous détromper :

> Cf tableau des tps/compx montrant que les progrès technologiques ne suffisent pas.

4.4 Recursivité croisée - mutuelle

Exemple :

– 0 est pair.
– n est pair ssi $n-1$ est impair.
– n est impair ssi $n-1$ est pair.

La fonction `pair` fait appel à la fonction `impair` et inversement, on pourrait ainsi craindre que l'évaluation ne s'arrête jamais. On parle ici de **récursivité croisée** ou **mutuelle**.

Question : Qu'est-ce qui garantit que toute évaluation se termine vraiment pourt tout $n > 0$ **?**

$\Rightarrow$ le fait que le paramètre avec lequel l'une et l'autre sont exécutées diminue à chaque étape, si bien que la condition d'arrêt de `pair` ou celle d'`impair` est rencontrée au bout d'un temps fini.
Exemple :

```
>pair(3) -> impair(2) -> pair(1) -> impair (0) -> #f
```

Exercice : Ecrivez une fonction `(premier n)` qui renvoie `#t` ssi n est un nb premier, *i.e.*, un nb ayant seulement 2 diviseurs, ex : 1,2,3,5,7,11,13,17,...

Solution :

```
(define (premier n)
        (prem 2 n))
(define (prem div n)
        (cond ((= div n)            \#t)
              ((= 0 (modulo n div)) \#f)
              (else (prem (+ div 1) n))))
```

4.5 Exemples pour exercices

- Exemple : sum(a,b) à coups de +1

$$\begin{array}{ll} sum(a,b) = a & \text{si } b = 0 \\ sum(a,b) = 1 + sum(a,b-1) & \text{si } b > 0 \end{array}$$

- Exemple du calcul de la puissance n de d'un nb m : une implémentation en $O(n)$ et une $O(\log n)$.

- Exemple : calcul des combinaisons en se ramenant a deux combinaisons plus simples C(n-1 p-1) et C(n-1 p) autre facon de faire : par les factorielles

5 Symboles, paires et listes

5.1 Symboles

Les **symboles** sont un type d'information différent des types numériques et booléens. Utiles pour modéliser des concepts de la vie de tous les jours sur lesquels on veut raisonner.

Un symbole est **toute suite de caractères alphanumériques** (sauf les caractères spéciaux) : même contraintes que pour un nom de variables.

Pour dire que la valeur d'un variable est un certain symbole, on utilise l'instruction spéciale `quote` :

```
>(define var (quote Velo))
var
>var
velo
```

On a besoin de `quote` pour dire à SCHEME de **ne pas évaluer** le mot qui suit mais de le prendre tel quel, de le considérer comme une valeur de symbole et non comme un nom de fonction ou de variable.

REMARQUE : si l'on veut éviter la conversion majuscule/minuscule, il faut entourer le symbole de guillemets.

Comme on a fréquemment besoin de “quoter” en SCHEME , il existe une notation abrégée, l'**apostrophe** :

```
>(define var 'velo)
var
>var
velo
>'Professeur
professeur
>'''Professeur''
Professeur}
```

5.2 Paires

Une paire est une structure permettant de regrouper deux informations. On construit une paire à l'aide de la fonction **cons** :

```
>(cons 1 2)
(1 . 2)
```

Une paire est une information structurée mais en tant qu'information elle peut être donnée comme valeur à une variable :

```
>(define doublet (cons 10 20))
doublet
>doublet
(10 . 20)
```

REMARQUE : : Des informations de types différents peuvent être associées :

```
>(cons 'velo 2)
(velo . 2)
```

La première information d'une paire est nommée **car** et la seconde **cdr**. Les fonctions du même nom renvoient respectivement ces éléments :

```
> (car doublet)
10
> (cdr doublet)
20
```

Pour savoir si une variable contient une paire, on dispose de la fonction `pair?` :

```
>(pair? '(10 . 20))
#t
```

Bien sûr, les paires peuvent êtres combinées : on peut définir des paires de paires, etc :

```
>(define a (cons 10 (cons 20 30)))
>(define b (cons (cons 10 20) 30))
>b
((10 . 20) . 30)
>(car b)
(10 . 20)
>(car (car b))
10
>(caar b)
10
>(cdr (car b))
20
>(cdar b)
20
```

Les instructions `car` et `cdr` **combinées** peuvent être abbrégées en une seule commande : **c** < **seq** > **r** où < **seq** > est une liste ordonnée de `a` et de `d` représentant resp `car` et `cdr`.

EXEMPLE : `(cadar x)` signifie `(car (cdr (car x)))`.

5.3 Listes

La structure de données (*i.e.*, représentation d'informations) la plus utilisée en SCHEME

5.3.1 Définition

Une liste est soit
– la liste vide `()`
– une paire `(e . l)` où `l` est une liste

```
(1 . (2 . (3 . ())))        ; est une liste
(1 . (2 . 3))               ; n'est pas une liste
```

Notation : toute liste

$$(e_1 \,.\, (e_2 \,\ldots\, (e_n \,.\, ())\,\ldots\,))$$

sera notée plus commodément

$$(e_1\ e_2\ \ldots\ e_n\ )$$

5.3.2 Fonctions prédéfinies

(*list* $exp_1\ exp_2\ \ldots\ exp_n$) : retourne la liste $(v_1\ v_2\ \ldots\ v_n)$ où chaque v_i est la valeur de exp_i.

`(list ? l)` : renvoie `#t` si `l` est une liste `#f` sinon.

`(null ? l)` : renvoie `#t` si `l` est la liste vide, `#f` sinon.

Quelques exemples :

Expression	**Valeur**
`(list 1 (+ 1 1) (* 3 1))`	(1 2 3)
`(list ? ())`	`#t`
`(list ? (cons 1 ()))`	`#t`
`(list ? 1)`	`#f`
`(null ? ())`	`#t`
`(null ? (cons 1 ()))`	`#f`

Remarque : l'expression spéciale **quote** permet elle aussi de construire une liste, mais celle-ci doit être précisée **explicitement** :

```
>'(a b c)
(a b c)
```

5.3.3 Exemples typiques

La première technique à acquérir pour manipuler une liste est son parcours :

```
(define (affiche l)
        (if (not (null? l))
               (begin (display (car l))
                      (affiche (cdr l)))))
```

On parcourt la liste élément aprés élément, on réalise une opération sur la base de chaque élément rencontré (son affichage) puis on itère **récursivement** sur le reste de la liste. On s'arrête quand la liste est vide.

C'est une structure de programme que l'on retrouvera très souvent.

Autre exemple : compter le nb d'éléments présents dans une liste :

```
(define (longueur l)
        (if (null? l) 0
                      (+ 1 (longueur (cdr l)))))
```

EXERCICE : Transformer la fonction précédente en fonction récursive terminale.
SOLUTION : cf TD.

Autre exemple : écrire une fonction qui calcule la somme des nombres d'une liste :

```
(define (sommer liste)
        (if (list? liste)
            (if (pair? liste)
                (+ (car liste) (sommer (cdr liste)))
                0)
            0))
```

Cette fonction vérifie déjà que son paramètre est une liste, toutefois confrontée à une liste contenant des symboles, une erreur se produira. Pour la rendre plus robuste on peut écrire :

```
(define (liste-de-nb? l)
        (if (null? l)
            #t
            (if (pair? l)
                (if (number? (car l))
                    (liste-de-nb (cdr l))
                    #f)
                #f)))

(define (sommer liste)
        (if (list-de-nb? liste)
            (+ (car liste) (sommer (cdr liste)))
            0))
```

Note : on aurait aussi pu écrire :

```
(define (liste-de-nb? l)
        (or (null? l)
            (and (pair? l)
                 (number? (car l))
                 (liste-de-nb? (cdr l)))))
```

A l'administration de la fac, codage d'un étudiant par une liste d'informations : `(nom prenom numInsee genre age promo)`.

Pour obtenir le prénom d'un étudiant, on peut ensuite définir :

```
(define (prenom etudiant)
        (cadr etudiant))
```

A vous de jouer :

EXERCICE : Ecrire une fonction `BonnePile ?` qui dit renvoie `#t` ssi la liste d'objets passée en paramètre est une pile (décrite de bas en haut) qui tient debout. Les objets peuvent être un cube, un disque, une pyramide et une pile tient debout si elle fait moins de 10 objets de haut et si aucune pyramide n'est située avant un autre objet. Exemple :

Pile d'entrée	Bonne Pile ?
`(cube cube disque pyramide)`	`#t`
`(disque pyramide cube)`	`#f`
$(cube_1 \ \ldots \ cube_{11})$	`#f`

SOLUTION : cf ci dessous :

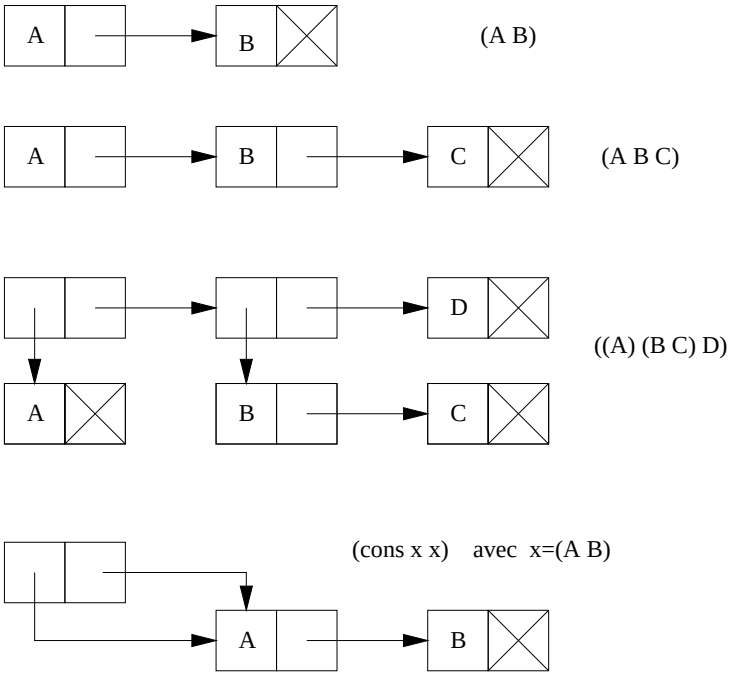
```
(define (BonnePile? l)
  (and (< (length l) 11)
       (BP? l)))
(define (BP? l)
  (if (null? l) #t
    (if (equal? (car l) 'p) (null? (cdr l))
      (BP? (cdr l)))))
```

Pbm : on parcours parfois deux fois la liste. Mieux :

```
(define (BPile? l)
  (BP2? l 1))
(define (BP2? l nb)
  (if (null? l) #t
    (if (> nb 10) #f
     (if (equal? (car l) 'p) (null? (cdr l))
       (BP2? (cdr l) (+ 1 nb))))))
```

5.3.4 Représentation (haut niveau) en mémoire

Rappeler rapidement comment une liste est réalisée en mémoire : chq info est un doublon de cases accolées dont la partie gche contient une valeur (car) et la partie droite un pointeur vers un autre doublon (la suite de la liste).



Leur faire trouver la composition des listes depuis leur représentation en mémoire, cf fig 3.1 p33 de EMSE.

5.3.5 Composition de listes

De même qu'on peut combiner des paires entre elles pour donner entre autres des listes, on peut combiner des listes entre elles.

```
>(list 'a (list 'b 'c) 'd)
(a (b c) d)
```

Dessiner sa représentation en mémoire.

Exemple du codage d'un arbre : sous-arb gche et sous-arb droit.

5.3.6 Pièges à éviter

Exemples de commandes qui marchent ou pas équivalentes ou pas sur cons car cdr.....

```
>(cons 'a '(b c))
(a b c)
>(cons 'a ('b 'c))
ERREUR
```

Dans le dernier cas, la parenthèse ouverte laisse supposer à SCHEME qu'un opérateur va suivre, or il n'en trouve aucun.

```
>(cons '(a b) 'c)
((a b) . c)      ; paire (composee d'une liste et d'un nb) et non liste
```

5.4 Fonctions plus avancées sur les listes

5.4.1 Append

Pour mettre bout à bout deux listes. Exemple

```
> (append '(1 2 3) '(4 5))
(1 2 3 4 5)
```

On écrit ça simplement :

```
(define (append l1 l2)
        (if (null? l1)
            l2
            (cons (car l1) (append (cdr l1) l2))))
```

Noter le cas où on arrive au bout de la liste l1, on renvoie simplement l2 qui est une liste, auquel sont rajoutés en tete les éléments de l1 dans les différents contextes précédents, lorsque on y revient en vidant la pile des appels de fonction.

Cf Figure détaillant l'exemple précédent avec les états mémoire et la pile des contexte - Arbre d'évaluation.

5.4.2 Eléments extrémités d'une liste

```
(define (premier l)
        (car l))


(define (dernier l)
        (if (null? (cdr l))
            (car l)
            (dernier (cdr l))))
```

Pbm : que se passe-t-il si on demande `(dernier '())` ?
Une erreur se produit. Comment corriger ?

```
(define (dernier l)
        if (null? l)
           '()
           (dernier2 l))
```

Attention : integrer ce test a l'interieur de la fonction dernier elle-meme amene a des operations executees inutilement : une fois passee la premiere fois, on sait que le `(cdr l)` n'est pas null donc que l n'est pas vide.

Enlever le dernier élément d'une liste :

```
(define (ote-dernier l)
        (if (null? (cdr l))
            '()
            (cons (car l) (ote-dernier (cdr l)))))
```

Attention même pbm que précédemment si on lui passe une liste vide.

REMARQUE : Les fonctions `length` et `reverse` existent aussi en SCHEME mais peuvent être écrites facilement (cf TD).

EXERCICE : Supposons qu'une date soit codée sous forme d'une liste à trois éléments : `'(4 11 1999)` pour le 4 novembre 1999. Ecrire une fonction `lendemain` qui renvoie la date du lendemain d'une date passée en paramètre. **Question :** votre programme est-il sensible au bug de l'an 2000 **?**

SOLUTION : cf cours Jean-Claude.

REMARQUE : Comment vérifier que deux dates sont identiques ?

```
(define (jour d) (car d))
(define (mois d) (cadr d))
(define (annee d) (caddr d))
(define (egalite d1 d2)
        (and (= (jour d1) (jour d2))
             (= (mois d1) (mois d2))
             (= (annee d1) (annee d2))))
```

Comment compter le nombre de jours entre deux dates ?

```
(define (nb-jours-ecarts d1 d2)
        (if (egalite d1 d2)
            0
            (+ 1 (nb-jours-ecart (lendemain d1) d2))))
```

Attention : ne marche que si $d1 < d2$!!!
Question : Comment faire pour savoir si $d1 < d2$ **?**

$\Rightarrow$: vérifier que les chiffres des dates sont positifs et comparer le nb de jours d'ecart entre `'(0 0 0)` et d1 puis de même avec d2, le plus petit écart désigne la date antérieure à l'autre. La fonction plus robuste se réécrit en :

```
(define (nb-jours-ecart d1 d2)
```

```
(if (and (> 0 (jour d1))
         ...
         (> 0 (annee d2)))
    (let ( (n1 (nb-jours-ecarts2 '(0 0 0) d1))
           (n2 (nb-jours-ecarts2 '(0 0 0) d2)) )
         (abs (- n1 n2)))
    (display ``pbm : nb negatif ds une date !!!'')))
```

où `nb-jours-ecarts2` est la fonction précédemment écrite.

REMARQUE : Traiter autant de cas que possible renforce la qualité d'un programme.

Pour avoir le nb-de-jours que vous avez déjà vécus :
`(nb-jours-ecart date-naissance aujourdhui)` après avoir défini les deux variables.

Petit laïus sur les dates : un type d'info particulier, realise par l'intermediaire des listes. De facon generale, les listes sont un mecanisme puissant qui permet de batir pleins de nouveaux types : dates, arbres, nb complexes, polynomes, ensembles, ...

D'ou prochain chapitre : un cadre et un exemple pour la definition de nouveaux types.

6 Le TDA Ensemble

6.1 Notion de Type Abstrait de Données

Précédemment, nous avons a plusieurs reprises montré comment représenter des notions abstraites non prévues dans SCHEME par défaut : les dates, les piles, les arbres, les polynomes, etc.

En fait, nous avons à chaque fois choisi une représentation pour ces abstractions et défini un certain nombre de fonctions permettant de manipuler des données de ce type.

En fait, ce que nous avons fait à chaque fois est de définir un nouveau de **type de données** SCHEME **pour représenter une abstraction** que nous manipulons

souvent : c'est pourquoi on parle de **Type de données abstrait** (TDA). Il s'agit d'une notion importante en informatique.

Un TDA se caractérise par :
– un nom ;
– un ensemble particulier d'informations spécifiques
– une représentation en machine ;
– des opérateurs sur ce TDA, ie des fonctions permettant de manipuler les différentes valeurs associées à une information de ce type.

EXEMPLE :
– le type DATE ;
– le jour, le mois et l'année ;
– une liste (jour mois année) ;
– les fonctions `jour, mois, annee, lendemain, nbjoursecart`, ...

Quelques mots sur :
– la **représentation en machine** : en SCHEME , la représentation de (presque) tout nouveau TDA passe par l'utilisation des **paires** et souvent plus spécifiquement des **listes**.
– **les opérateurs du TDA** : dans le cas idéal, l'ensemble d'opérateurs définis pour le TDA doivent permettre à un utilisateur quelconque de manipuler une information de ce type et d'accéder à ses différentes parties sans savoir à connaître la façon dont l'information est représentée (ie liste, etc).
  Le TDA peut ainsi être vu comme une extension de SCHEME que le programmeur met à la disposition d'autres utilisateurs, leur permettant de définir des fonctions de plus haut niveau (ie, plus prêt de l'homme par rapport à la machine) par l'intermédiaire des nouvelles abstractions mises à leur disposition.
– **Opérateurs particuliers** : un **constructeur** d'un TDA T est une fonction ayant pour but de retourner un objet de type T depuis les différentes information dont il doit être composé. Exemple : `cons` pour les listes.
  un **selecteur** est une fonction permettant d'accéder à une partie d'information contenue dans une valeur de type T. Exemple : `(jour date)` pour le type DATE, `car, cdr` pour le type liste.

Dans ce chapitre nous allons illustrer ces différentes notions en créant un TDA **Ensemble**, permettant de manipuler des ensembles quelconques d'entiers.

Nous étudierons plusieurs représentations possible pour un ensemble (dans le but d'en choisir une seule une fois connus leurs avantages et inconvénients respectifs) :

1. liste non triée, sans redondance : `(2 3 5) = (5 2 3)`, ...

2. liste non triée, avec redondance : `(2 3 2 5) = (3 2 5 3)`, ...
3. liste triée, sans redondance : `(2 3 5)`.

Dans chaque cas, nous écrirons les fonctions : $\in$, ajout et suppression d'un élem, cardinal d'un ensemble, intersection et union de deux ensembles, inclusion d'un ensemble dans un autre et égalité de deux ensembles.

Dans tous les cas, nous pouvons définir l'ensemble `vide` comme étant la liste vide :`(define vide '())`.

6.2 Le TDA Ensemble - 1ère représentation

Un ensemble = une **liste non triée, sans redondance**.

```
(define (vide? E)
   (null? E))

(define (appartient x E)
   (cond ((vide? E) #f)
         ((= x (car E)) #t)
         (else (appartient x (cdr E)))))

(define (ajoute x E)
   (if (appartient x E)
       E
       (cons (x E))))

(define (supprime x E)
     (cond ((vide? E) '())
           ((= x (car E)) (cdr E))
           (else (cons (car E) (supprime x (cdr E))))))

(define (cardinal E)
   (if (vide? E)
       0
       (+ 1 (cardinal (cdr E)))))
```

```
(define (inter E1 E2)
   (cond ((vide? E1) E2)
         ((appartient (car E1) E2) (cons (car E1) (inter (cdr E1) E2)))
         (else (inter (cdr E1) E2))))

(define (union E1 E2)
   (cond ((vide? E1) E2)
         ((appartient (car E1) E2) (union (cdr E1) E2))
         (else (cons (car E1) (union (cdr E1) E2)))))

(define (inclus E1 E2)
    (cond ((vide? E1) #t)
          ((appartient (car E1) E2) (inclus (cdr E1) E2))
          (else #f)))

(define (egal E1 E2)
      (and (inclus E1 E2)
           (inclus E2 E1)))
```

6.3 Le TDA Ensemble - 2ème représentation

Un ensemble = une **liste non triée, avec redondance**.

```
appartient : idem

(define (ajoute x E) (cons x E))

(define (supprime x E) : idem sauf: continuer qd x est trouve

(define (cardinal E) : attention ne pas compter deux fois le meme elem.

(define (inter E1 E2) : idem

(define (union E1 E2) (append E1 E2))

(define (inclus E1 E2) : idem
```

```
(define (egal E1 E2) : idem
```

6.4 Le TDA Ensemble - 3ème représentation

Un ensemble = une **liste triée, sans redondance**.

```
(define (appartient x E)
   (cond ((vide? E) #f)
         ((= x (car E)) #t)
         ((< x (car E)) #f)
         (else (appartient x (cdr E)))))

(define (ajoute x E)
   (cond ((vide? E) (list x))
         ((= x (car E)) E)
         ((< x (car E)) (cons x E))
         (else (cons (car E) (ajoute x (cdr E))))))

(define (supprime x E)
   (cond ((vide? E) '())
         ((= x (car E)) (cdr E))
         ((< x (car E)) E)
         (else (cons (car E) (supprime x (cdr E))))))

cardinal : idem
```

Pour l'intersection et l'union, l'idee est de progresser a la fois dans E1 et dans E2, en comparant les elements en tete des deux listes.

faire schema et derouler l'algo sur un exemple

`E1 = (1 2 3 4 5 6)` `E2 = (1 3 6 7).`

Puis :

```
(define (inter E1 E2)
```

```
   (cond ((or (vide? E1) (vide? E2)) '())
         ((= (car E1) (car E2)) (cons (car E1) (inter (cdr E1) (cdr E2))))
         ((< (car E1) (car E2)) (inter (cdr E1) E2))
         (else (inter E1 (cdr E2)))))
```

INTERET : on ne parcours chaque liste qu'une seule fois (ie que le minimum oblige) alors que dans les premieres implementations, on parcourait `E2` pour chaque élément de `E1`.

EXERCICE : Ecrivez la fonction `union E1 E2)` suivant la même idée.
SOLUTION :
```
(define (union E1 E2)
(cond ((vide ? E1) E2)
((vide ? E2)) E1)
((= (car E1) (car E2)) (cons (car E1) (union (cdr E1) (cdr E2))))
((< (car E1) (car E2)) (cons (car E1) (union (cdr E1) E2))
(else (cons (car E2) (union E1 (cdr E2))))))
```

EXERCICE : Ecrivez la fonction `(separe E)` qui renvoie une liste composée de deux listes : la liste des éléments impairs de `E` et la liste des éléments pairs de `E`.
Exemple :
```
> (separe '(1 2 4 6 7 8))
( (1 7) (2 4 6 8))
```
Rappel : la fonction `(odd ? x)` renvoie vrai ssi `x` est impair.
SOLUTION :
```
(define (separe E)
(if (vide ? E) (cons '() '())
(let ( (deuxlistes (separe (cdr E))) )
(if (odd ? (car E))
(cons (cons (car E) (car deuxlistes)) (cdr deuxlistes))
(cons (car deuxlistes) (cons (car E) (cdr deuxlistes)))))))
```

```
(define (inclus E1 E2)
   (cond ((vide? E1) #t)
         ((vide? E2) #f)
         ((= (car E1) (car E2)) (inclus (cdr E1) (cdr E2)))
         ((< (car E1) (car E2)) #f)
         (else (inclus E1 (cdr E2)))))
```

```
(define (egal E1 E2)
   (cond ((or (vide? E1) (vide? E2)) (and (vide? E1) (vide? E2)))
         ((= (car E1) (car E2)) (egal (cdr E1) (cdr E2)))
         (else #f)))
```

Rmq : ici aussi un seul parcours (au pire) de E1 et E2 est necessaire.

6.5 Conclusion : choix d'une représentation

Avantages de la 3ème sur la 1ère :
- `appartient, ajoute, suppression` ne nécessitent généralement pas le parcours de toute la liste quand `x` n'est pas dans `E`.
- `inter, union, inclusion, egalite` ne nécessitent qu'un seul parcours (au pire) de `E1, E2`, alors et non pas plusieurs parcours de `E2`.
-

⇒ la troisième représentation est clairement préférable à la première.

Avantages/inconvénients de la 3ème représentation par rapport à la 2ème :
- Av : la taille de la liste gérée est proportionnelle au nombre d'éléments et non au nombre d'ajouts. En conséquence, tous les parcours de l'ensemble sont généralement plus rapides (dès qu'on a plusieurs occurences d'un élément).
- Inc : `ajout` s'effectue en $O(1)$ dans le cas de la 2ème repr. alors qu'on va parcourir en moyenne la moitié de la liste dans le cas de la 3ème repr.
- Inc : `union` nécessite le parcours complet de `E1, E2` dans la 3ème alors que dans la 2ème il ne nécessite que le parcours de E1.
- Av : cardinal en $O(n)$et non $O(n^2)$.
- Av : `inter, inclus, egalite` ne nécessite q'un parcours de `E1,E2` alors que dans la 2ème il demande plusieurs parcours de `E2`.
- Av : *appartient* ne nécessite pas tjs un parcours de toute la liste dans les cas où on recherche un élément qui n'est pas ds l'ensemble.

Conclusion : si l'on souhaite utiliser le TDA pour des ensembles contenant beaucoup d'éléments (non répétés) et faire peu de tests d'appartenance (directement ou indirectement (ie par les fonctions `inter`,etc)) on peut choisir la 2ème repr., sinon la 3ème repr. est plus rapide (cas général).

Question : qu'est-ce qu'il nous faudrait changer si on voulait manipuler des ensembles de symboles et non de nombres **?**

⇒ changer `=` en `equal ?` et définir les fonctions `inferieur ?, superieur ?` pour remplacer `<,>`. Bref définir un ordre en les symboles (ex : celui du dictionnaire).

EXERCICE : Ecrire pour chaque représentation des ensembles la fonction `(est-ensemble ? obj)` qui renvoie `#t` ssi `obj` est un ensemble de nombre en accord avec les conventions imposées par chaque représentation.

SOLUTION : :

```
(define (est-ens1? obj)
  (if (not (list? obj)) #f
      (if (null? obj) #t
          (and (number? (car obj))
               (not (member (car obj) (cdr obj)))
               (est-ens1? (cdr obj))
          ))))

; > (est-ens1? '(33 2 1 4))
; #t
; > (est-ens1? '(1 2 \#C 4))
; #f
; > (est-ens1? '(1 (2 3) 4))
; #f

(define (est-ens2? obj)
  (if (not (list? obj)) #f
      (if (null? obj) #t
          (and (number? (car obj))
               (est-ens2? (cdr obj))
          ))))
; > (est-ens2? '(1 2 1))
; #t
; > (est-ens2? '(1 (2) 1))
; #f

(define (ordre-ok? obj)
  (cond ((null? obj) #t)
        ((null? (cdr obj)) #t)
        (else (and (< (car obj) (cadr obj))
                   (ordre-ok? (cdr obj))))))
```

```
(define (est-ens3? obj)
 (and (est-ens2? obj)
      (ordre-ok? obj)))

; > (est-ens3? '(1 2 3))
; #t
; > (est-ens3? '(1 (2 3)))
; #f
; > (est-ens3? '(titi 2 4))
; #f
```

7 Les tris

Le chapître précédent a montré l'intérêt de disposer de listes triées pour gérer des ensembles. En informatique, il est bien connu qu'il est **plus facile** et **plus rapide** de travailler sur des informations triées plutôt qu'ordonnées arbitrairement.

Il exsite un grand nombre de méthodes pour trier une liste de nombres, nous verrons ici les plus classiques.

7.1 Tri par insertion

Idée : pour trier une liste de $n > 0$ éléments :
– trier les $n - 1$ derniers éléments de la liste $l \Rightarrow l'$.
– insérer le 1er élément à sa place en parcourant la liste triée l'.

Exemple :

```
(trier (6 1 4 2 10)) = insere 6 (tri (1 4 2 10))
                     = insere 6 (insere 1 (tri 4 2 10))
                     = insere 6 (insere 1 (insere 4 (tri 2 10)))
                     = insere 6 (insere 1 (insere 4 (insere 2 (tri 10))))
                     = insere 6 (insere 1 (insere 4 (insere 2 (10))))
                     = insere 6 (insere 1 (insere 4 (2 10)))
                     = insere 6 (insere 1 (2 4 10))
```

```
= insere 6 (1 2 4 10)
= (1 2 4 6 10)
```

Lors du retour des appels récursifs : la liste est reconstituée en plaçant chq élément à la bonne place dans la partie triée de la liste. On peut en qq sortes distinguer deux parties dans la liste à un moment donné : la partie gauche non-triée, et la partie droite triée, qui s'étend progressivement au dépend de la première (cf fig).

FIG. 3 – Découpage virtuel de la liste lors de son tri par insertion

Ce tri peut être écrit de la façon suivante :

```
(define (insere x l)
    (cond ( (null? l) (list x) )
          ( (not (> x (car l))) (cons x l) )
          ( else (cons (car l) (insere x (cdr l))))))

(define (tri_inser l)
   (if (null? l) ()
       (insere (car l) (tri_inser (cdr l)))))
```

A noter la partie `not...(car l)` qui permet d'insérer `x` dès que possible dans la liste, ie de ne pas parcourir les différentes occurences de `x` déjà rencontrées si la liste en contient plusieurs.

L'algorithme est relativement simple mais est-il efficace ?

- Supposons que la liste contienne au départ n éléments.
- La méthode consiste à insérer chacun à sa place dans la fin de la liste déjà triée : on a donc n insertions à faire.
- La première insertion s'effectue dans une liste vide, puis dans une liste de taille 1, de taille 2, etc.
- Dans le pire des cas, l'élément à insérer doit toujours l'être à la fin de la liste déjà triée, ce qui demande un temps proportionnel à sa longueur $|l|$.
- Les n insertions coûtent donc au pire des cas $1 + 2 + ... + n - 1 \approx \sum_1^n i = n \times (n-1)/2 = O(n^2)$

Ainsi, le nombre d'opérations élémentaires que demande cette méthode pour trier une liste de n éléments est proportionnel à n^2 dans le pire des cas.

Ce qui veut dire que si la taille de la liste à trier double, le temps nécessaire pour la trier quadruple.

Si 100 éléments sont triés en 1 seconde, trier 10000 éléments nécessite $10^4 s \approx 3h$.

Note : dans une base de données traditionnelle, le nombre d'éléments à trier sera plus proche de 10000 que de 100, avec pour extrême le fichier national des numéros INSEE : 60 millions de nombres à trier.

Question : Quel est l'agencement initial des n éléments à trier qui est le plus favorable, ie demande le moins d'opérations **?**

$\Rightarrow$ quand la liste l est déjà triée!!!! **Question :** Quel est en gros le nombre d'opérations effectuées dans ce cas **?**

$\Rightarrow$ chaque insertion a lieu en tête de la liste déjà triée, donc recquiert un nombre constant c d'opérations, donc les n insertions s'effectuent en $c \times n$, ie $O(n)$.

7.2 Tri par sélection ou "tri à bulles"

Dans le cas des listes récursives (SCHEME) ces deux tris sont identiques (mais pas dans le cas de listes itératives : tableaux).

Idée : pour trier une liste de $n > 0$ éléments :
- retirer le plus petit élément x de l
- trier les $n - 1$ éléments restant dans l
- insérer x en tête de l

EXEMPLE :

```
(tri (6 1 4 2 10)) = cons 1 (tri (6 4 2 10))
                   = cons 1 (cons 2 (tri 6 4 10))
                   = cons 1 (cons 2 (cons 4 (tri 6 10)))
                   = cons 1 (cons 2 (cons 4 (cons 6 (tri 10))))
                   = cons 1 (cons 2 (cons 4 (cons 6 (10))))
                   = cons 1 (cons 2 (cons 4 (6 10)))
```

```
= cons 1 (cons 2 (4 6 10))
= cons 1 (2 4 6 10)
= (1 2 4 6 10)
```

Traduction en SCHEME : attention : il semblerait que, pour chaque élément, on a besoin d'un premier parcours de la liste pour **connaître** l'élément min de l, puis d'un autre parcours pour le supprimer :

```
(define (trouve_min l)
    (if (null? (cdr l)) (car l)
       (let ( (m (trouve_min (cdr l))) )
            ( min m (car l)))))

(define (supprime x l)
    (if (= x (car l)) (cdr l)
        (cons (car l) (supprime x (cdr l)))))

(define (tri_selection l)
    (if (null? l) '()
        (let ( (m (trouve_min l)) )                  ; 1er parcours de l
             ( cons m (tri_selection
                                  (supprime x l)  ; 2eme parcours de l
                                  ) ))))
```

En fait, on peut faire mieux : on part du fond de la liste et on permute un élément avec celui d'en dessous si ce dernier est plus petit, ce qui a pour résultat final (quand on arrive au début de la liste) d'avoir fait remonter le plus petit élément en tête de la liste.

```
(define (remonte_min l)
   (cond ( (null? l) '() )
         ( (null? (cdr l)) l )
         ( else (let ( (l2 (remonte_min (cdr l))) )
                     (if (< (car l2) (car l))
                           (cons (car l2) (cons (car l) (cdr l2)))
                           (cons (car l) l2))))))
```

Le min étant remonté en tête de liste, il ne reste plus alors qu'à trier le reste de la liste (ie tout sauf le 1er élément) :

```
(define (tri_selection l)
      (cond ( (null? l) '() )
            ( (null? (cdr l)) l)
            ( else (let ( (lmin (remonte_min l)) )
                        ( cons (car l) (tri_selection (cdr l)) )
                  ))))
```

Cette fois le tri de la liste progresse de gauche à droite, à l'étape i les $i-1$ premiers éléments sont triés et les $n-i$ derniers restent à trier (cf fig).

FIG. 4 – Découpage virtuel de la liste lors de son tri par sélection

Question : Quelle est en gros le nb d'opérations nécessaires à cette méthode pour trier une liste de n éléments ? **?**

⇒ Trouver le min, nécessite de parcourir toute la liste, trouver le 2ème min, nécessite de parcourir toute la liste sauf 1 élémént, ... trouver le min des deux derniers éléments nécessite de parcourir les deux derniers éléments. Le nombre d'opération effectuée en tout est donc de $n + (n-1) + ... + 2 \approx \sum_{i=1} n = O(n^2)$.

Tri pas plus efficace que le précédent, même pire d'un certain point de vue :

Question : Y a-t-il une configuration des éléments de l plus ou moins facile à trier **?**

⇒ pas vraiment, quel que soit l'agencement des éléments de l on va aller chercher le plus petit élément jusqu'au bout de la liste et ainsi de suite. Même si la liste est déjà triée, on ne peut pas s'en rendre compte avant d'aller voir jusqu'au bout pour trouver le min, de même pour chercher le 2ème min, car on n'a retenu aucune information de la recherche précédente (ie a-t-on effectué ou pas des permutations).

Alors que dans le cas du tri précédent, plus la liste initiale est proche d'une liste triée, plus on gagne.

7.3 Tri par fusions

On applique ici un principe bien connu en informatique : **diviser pour régner**. Pour résoudre un pbm, cela consiste à le diviser en sous-pbm plus faciles à résoudre (de taille inférieure), puis à combiner les solutions aux différents sous-pbm en une solution pour le pbm initial. Voyons comment cette stratégie peut être appliquée pour trier une liste :

Idée : pour trier une liste de $n > 0$ éléments :
- diviser l en deux listes l_1 et l_2 de même taille (à un près).
- trier l_1 et l_2 séparément $\Rightarrow l_1', l_2'$.
- *fusionner* les listes triées l_1' et l_2', ie les combiner en une liste l' correspondant à l triée.

Exemple :

```
(tri (6 1 4 2 10))
          = fusion (tri (6 1 4)) (tri (2 10))
          = fusion (fusion (tri (6 1)) (4)) (fusion (tri (2)) (tri (10)))
          = fusion (fusion (fusion (tri (6)) (tri (1))) (4)) (fusion (2) (10))
          = fusion (fusion (fusion (6) (1)) (4)) (2 10)
          = fusion (fusion (1 6) (4)) (2 10)
          = fusion (1 4 6) (2 10)
          = (1 2 4 6 10)
```

Dessiner l'arbre des appels de fonction et montrer les répercutions sur la liste de valeurs au fur et à mesure.

Cette fois la progression du tri est plus diffuse : des parties d'abord petites puis de plus en plus grandes de la liste sont triées, jusqu'à ce que la liste soit triée en entier.

La fonction `fusion` a déja été vue en TD et en cours (il s'agit de l'union de deux ensembles représentés sous forme de listes triées !). La cas où `(car l1), (car l2)` sont identiques est géré différemment : on garde ici les deux éléments (un tri est une méthode générale, on part du principe que si l'utilisateur permet des répétitions dans l, alors ce n'est pas au tri de les enlever.

Il nous reste à écrire une fonction `coupe` qui divise une liste en deux sous-listes de même taille (à un près) et la fonction `tri fusion` qui utilise `coupe` et `fusion`.

```
(define (fusion l1 l2)
   (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))))
   ))

(define (coupe l l1 l2)
     (if (null? l) (cons l1 l2)
         (coupe (cdr l) l2 (cons (car l) l1))))

(define (tri_fusion l)
   (if (or (null? l) (null? (cdr l))) l
        (let* ( (pairel1l2 (coupe l '() '()))
                (l1 (car pairel1l2))
                (l2 (cdr pairel1l2)) )
              (fusion (tri_fusion l1) (tri_fusion l2))
        )))
```

Notes sur `couper` : la façon de couper la liste en deux : on ne fait pas de parcours préliminaire de la liste à couper pour connaître la taille que doivent avoir les deux listes construites, il suffit de mettre une fois sur deux un élément de l dans l_1 et l'autre fois sur deux dans l_2. Pour cela on ajoute tjs l'élément en tête de l dans la 1ère sous-liste passée en paramètre, et l'appel d'après se fait en inversant l'ordre de l_1 et l_2 dans les paramètres.

Finalement, quand l ne contient plus aucun élément, on doit renvoyer l_1 et l_2. **Comment faire puisque une fonction** SCHEME **ne peut renvoyer qu'un objet** ? ? ? $\Rightarrow$ on utilise **l'astuce de la paire** : l'élément de gche de cette paire pointe sur l_1 et la partie droite sur l_2.

Noter dans `fusion` le `let*` afin d'évaluer les variables dans l'ordre et non indépendemment les unes des autres.

Cette méthode tri une liste de n éléments en un nombre d'opérations $O(n \log n)$ (croyez-moi sur parole, **pas d'explications : trop compliquées pour eux**).

Comment cette amélioration de la compx se traduit en pratique ? $\Rightarrow$ si 1s pour trier 100 éléments, il ne faut que $3mn20s$ pour en trier 10000 (au lieu de $\approx 3h$

pour les tris précédents).

7.4 Tri rapide

Idée : pour trier une liste l de $n > 0$ éléments :
- prendre le 1er élément p de l ;
- construire la sous-liste l_1 de l contenant les éléments $e < p$ et la sous-liste l_2 de l contenant les éléments $e \geq p$.
- trier récursivement l_1 et l_2.
- renvoyer la liste composée de l_1, p, l_2.

Exemple :

```
(tri (6 1 4 2 10))
          = (tri (1 4 2)) 6 (tri (10))
          = ((tri ()) 1 (tri (2 4))) 6 (10)
          = (() 1 ((tri ()) 2 (tri (4)))) 6 (10)
          = (() 1 (() 2 (4))) 6 (10)
          = (() 1 (2 4)) 6 (10)
          = (1 2 4) 6 (10)
          = (1 2 4 6 10)
```

Ce tri peut être écrit de la façon suivante :

```
(define (divise_pivot p l)
    (if (null? l) (cons '() '())
        (let ((m (divise_pivot (cdr l) p)))
             (if (< (car l) p)
                  (cons (cons (car l) (car m))
                        (cdr m))
                  (cons (car m) (cons (car l) (cdr m)))
             ))))
(define (tri_rapide l)
   (if (null? l) '()
       (let ((m divise_pivot (car l) (cdr l)))
            (append (tri_rapide (car m))
```

```
        (car l)
        (tri_rapide (cdr m))))))
```

La fonction `divise_pivot` renvoie une paire composée des deux listes l_1 et l_2.
REMARQUE : la commande `append` peut prendre plus de 2 listes en paramètres.

Ce tri nécessite en moyenne un temps $O(n \log n)$ et est plus rapide encore que le tri fusion.

Question : Y a-t-il un cas particulièrement défavorable au tri rapide **?**

$\Rightarrow$ Oui : si la liste est triée à l'envers au départ. D'où l'idée de s'assurer qu'on va tomber dans un cas moyen en battant les cartes (ce qu'on peut faire facilement et sans augmenter significativement le coût de l'algo dans le cas d'une *liste itérative* (alors que ici liste récursive), ce qui peut être fait en SCHEME avec un objet de type `vecteur`. La principale différence avec les listes récursives que nous avons manipulé jusqu'à maintenant est que l'on peut accéder directement à chaque élément de la liste au lieu de devoir parcourir tous les éléments le précédent.

EXERCICE : Ecrire une fonction `(kieme k l)` qui renvoit le $k$ième plus petit élément de la liste l, sur une idée similaire au tri rapide.
SOLUTION :

```
(define (kieme k l)
   (let* ((m (divise_pivot (car l) (cdr l)))
          (lgr (longueur (car m))))
         (cond ((= lgr (- k 1)) (car l))
               ((> lgr (- k 1)) (kieme k (car m)))
               (else (kieme (- k lgr) (cdr m)))
         )))
```

On peut aussi définir cette fonction sur le modèle des autres tris vus dans ce chapître, mais cette l'implémentation sur la base du tri rapide reste en moyenne la plus rapide.

8 Les chaînes de caractères (ex : vérification grammaticale d'une phrase)

8.1 Fonctions de base

Les chaînes de caractères sont des objets d'un *type* particulier, tout comme les valeurs numériques, les caractères, les listes, les booléens, etc.

Un objet de type chaîne est enclos de 2 guillemets :
`"bonjour"` est compose de 7 lettres situées en position 0 à 6.

Voici les fonctions les plus utiles :
- `(string? obj)` : renvoie vrai si *obj* est *de type* chaîne de caractère. Attention : un caractère n'est pas de type *chaîne de cars.*
- `(string-length ch)`
- `(substring ch deb fin)` : le car numero *deb* est inclus mais le car numero fin est exclus. EXEMPLE : : `(substring "toto" 2 3) -> "t"`.
- `(string=? ch1 ch2)` : égalite entre deux chaînes
- `(string-ref ch k)` : renvoie le $k$ième caractère de *ch* (valeur renvoyée est de *type* caractère).
- `(string caract1 .. caractn)` : renvoie une chaîne composée des n caractères indiqués. Permet ainsi de transformer qq chose de type caractère en un objet de type chaîne. Exemple :
  ```
  > (string #\A #\B)
  "AB"
  ```
- `(string-append ch1 ch2 ch3)` : attention tous les objets passe doivent etre de type chaîne. Pour ajouter des objets de type caractere, d'abord les transformer en objets de type chaîne de caracteres.
  ```
  > (string-append "je" "march" (string #\e))
  "je marche"
  ```

EXERCICE : Comment ecrire les fonctions `premier, dernier, inverse` ainsi que `(debut-ch ch k), (fin-ch ch k)` ?
SOLUTION :
```
(define (dernier m) (substring m (- string-length m) (string-length m)))
```

Dans la suite de ce chapître nous allons utiliser les fonctions sur les chaînes pour faire manipuler à l'ordinateur des phrases simples exprimées en Français.

EXEMPLE : `("le" "chat" "mange" "un" "lezard")`.

En raison du peu de temps qu'on a à consacrer à ce pbm, on se limite à une structure de phrase très simple, à des verbes du 1er groupe et à quelques pronoms.

8.2 Décomposition de chaînes (ex : vérification des accords)

Plus difficile que la vérification orthographique (éditeurs de texte). Commençons par faire connaître à l'odrinateur un certain nombre d'articles et de pronoms, en indiquant à quelle personne ils correspondent (par convention 1=1ère sing,2=2ème sing, ..., 6 = 3ème plur) :

```
(define Articles '(("le" 3) ("la" 3) ("les" 6) ("un" 3) ("une" 3)
                   ("des" 6) ("du" 3)) ; le numero indique la personne

(define Pronoms   (("je" 1) ("tu" 2) ("il" 3) ("elle" 3) ("nous" 4)
                   ("vous" 5) ("ils" 6) ("elles" 6))
```

On aura souvent besoin de chercher un mot dans ces listes et de connaître le numéro qui lui est associé :

```
(define (cherche mot liste)
     (cond ((null? liste) 0)   ; pas trouve
           ((string=? mot (caar liste)) (cadar liste)) ; renv la 2eme partie
           (else (article (cdr liste)))))
```

On peut facilement faire vérifier à SCHEME que les **accords entre un article et un nom** sont corrects dans une phrase donnée :

```
(define (NomSingPlur? mot)
    (if (or (equal? (dernier mot) #\s)
            (equal? (dernier mot) #\x)) 6
```

```
        3))

(define (accords? phrase)
  (if (null? phrase) #t
      (let ((nb (cherche (car phrase) Articles)))
           (if (= nb 0) (accords? (cdr phrase))) ; pas d'art, on continue
               (and (not (null? (cdr phrase))) ; il reste un mot : un nom
                    (= nb (NomSingPlur? (cadr phrase))) ; ils ont meme nb
                    (accords? (cdr phrase)))))))) ; le reste est correct
```

Comme on se retreint aux phrases sans adjectif, tout ce qui suit un article doit être un nom. On vérifie donc qu'il y a un mot après l'article et que ce mot est accordé avec l'article, ie qu'ils ont même nombre (3 singulier, ou 6 pluriel).

EXEMPLE :

```
> (accords? '("le" "chat" "mange" "un" "lezard"))
#t
> (accords? '("les" "vaches" "regardent" "des" "trains"))
#t
>(accords?  '("le" "chat" "mange" "la" "souris"))
#f ; limite des regles simplistes utilisees pour detecter pluriel
   ; l'ideal serait bien-sur d'avoir un dictionnaire pour les noms
   ; comme on l'a fait pour les articles et pronoms.
```

Dans le même style, on pourrait faire vérifier à SCHEME que l'accord entre le sujet et le verbe est correct.

8.3 Assemblage de chaînes (ex : conjugaison d'un verbe)

Apprenons à SCHEME un nouveau type de mots, les verbes :

```
(define Verbes '("march" "pos" "cherch" "mang" ...))
```

on ne stocke ici que le radical des verbes, à compléter par une terminaison apropriée selon le temps de la phrase (présent, futur, etc). Ces terminaisons (limitées ici au 1er groupe) peuvent être elles-aussi stockées sous forme de listes :

```
(define Temps (("present" ("e" "es" "e" "ons" "ez" "ent"))
               ("futur"   ("erai" "eras" "era" "erons" "erez" "eront"))
               ...
              )
```

Voyons comment on peut écrire une fonction qui **change le temps d'ennonciation d'une phrase** : on repère d'abord la personne p du sujet, puis on détermine la terminaison associée à cette personne pour le temps demandé, enfin on cherche le verbe de la phrase que l'on remplace par son radical complété par cette terminaison :

```
; on suppose que la phrase est de type sujet-verbe-complement
; on verra plus loin comment le verifier prealablement
(define (personne? mot)
  (let ((nbart (cherche (car phrase) Articles)))
       (if (not (= 0 nbart)) nbart             ; soit article
           (cherche (car phrase) Pronoms))))   ; soit pronom

(define (radical? rad mot)
  (cond ((= 0 (string-length rad)) #t)
        ((= 0 (string-length mot)) #f)
        ((equal? (premier rad) (premier mot))
            (radical? (substring 1 (string-length rad))
                      (substring 1 (string-length mot))))
        (else #f)))

(define (cherche-radical v listverb)
  (if (null? listverb) #f              ; radical pas trouve
    (if (radical? (car listverb) v) (car listeverb) ; trouve : le renvoie
        (cherche-radical v (cdr listverb)))))

; cherche le verb et le conjugue quand on le trouve, puis renvoie
; la phrase modifiee
(define  (conjugue2 term fin-ph deb-ph)
  (if (null? phrase) #f ; on n'a pas trouve le verbe de la phrase
      (let ((rad cherche-radical (car phrase) Verbes)) ; est-on sur un verb?
         (if (rad)
             (append deb-ph (string-append rad term) (cdr fin-ph))
```

```
            (conjugue2 term (cdr fin-ph) (append deb-ch (list (car phrase))))
        ))))

(define (conjugue phrase temps)  ; on suppose que la phrase est correcte
      (let (( p  (personne (car phrase))) ; 1er mot est forcement du sujet
            (tps (cherche temps Temps)))
           (if (or (= 0 p) (= 0 tps)) #f  ; pbm qq part
             (let ((terminaison (ieme p tps))) ; simple, vue en TD
                  (conjugue2 terminaison phrase '())
             ))))
```

EXEMPLE :

```
>(conjugue '("les" "vaches" "regardent" "des" "trains") "futur")
("les" "vaches" "regarderont" "des" "trains")
```

8.4 Notion de grammaire d'un langage (ex : structure d'une phrase)

La structure d'une phrase peut être décrite par une *grammaire* simple, par exemple :

```
PHRASE :- SUJET + VERBE + COMPLEMENT

SUJET  :- Article + NOM
       :- Pronom (de type sujet)

NOM    :- toute cha\^{\i}ne commencant par une minuscule

VERBE :- un mot forme de RADICAL + TERMINAISON
RADICAL :- element de la liste Verbes
TERMINAISON : element d'une sous-liste correspondant \`a un Temps

COMPLEMENT :- Article + NOM   ; on ne peut avoir de pronom sujet ici
```

On peut écrire une suites de fonctions SCHEME qui permettent de reconnaître une phrase correcte d'une phrase incorrecte suivant cette grammaire. Ainsi une phrase

est une liste de mots (chaînes) t.q. le 1er sous-groupe de mots joue le role SUJET, le deuxieme soit de nature VERBE, le 3eme joue le role de COMPLEMENT.

La fonction `(sujet? phrase)` verifie que la phrase commence bien par un sujet et renvoie le reste de la liste si elle en trouve un, faux sinon :

```
(define (sujet? phrase)
  (cond ((null? phrase) #f)
        ((not (= 0 (cherche (car phrase) Pronoms))) (cdr phrase))
        ((not (= 0 (cherche (car phrase) Articles)))
            (if (null? cdr phrase) #f ; pas de mot derriere l'article
                (cddr phrase)) ; passe le mot suivant considere un nom
        (else #f)))

(define (verbe? v)
  (cherche-radical v Verbes))
```

La fonction `(complement? l)` s'écrit de façon similaire. La fonction `(verbe? l)` est simple :

Puis on peut écrire la fonction qui vérifie si une phrase est correcte au sens de la grammaire détaillée :

```
(define (phrase? l)
  (let ((l1 (sujet? l)))
       (if (or (not l1) (null? l1)) #f ;
           (if (not verbe? (car l1)) #f
               (null? (complement? (cdr l1)))))))
```

La derniere ligne de cette fonction s'appuye sur le fait que seul `(null? '())` renvoie vrai. La fonction renvoie faux si le sujet, le verbe ou le complement (`(null? #f)` est faux) n'a pas ete correctement identifie, ou s'il reste des mots dans la phrase (ce qui n'est pas en accord avec la grammaire donnee precedemment). Exemple :

```
> (phrase? '("le chat" "mange" "une" "orange"))
#t
> (phrase? '("vous" "mangez" "une" "orange"))
#t
```

```
> (phrase? '("il" "une" "orange"))
#f
> (phrase? '("mange" "une" "orange"))
#f
```

EXERCICE : Ecrivez une fonction `decoupe` qui prend en entrée une phrase stockée dans une seule chaîne de caractères et la renvoie sous forme de liste de chaînes correspondant à ses mots. EXEMPLE :
`>(decoupe "je cherche une fleur")`
`("je" "cherche" "une" "fleur")`
SOLUTION : On découpe la chaîne suivant le caractère "espace". Correction en TD

EXERCICE : Ecrivez une fonction `(filtrage phrase liste-mots)` qui renvoie la liste des mots de `phrase` n'appartenant pas à `liste-mots`. EXEMPLE :
`>(filtrage '("je" "cherche" "une" "fleur") (append Pronoms Articles Relatifs Conjonctions))`
`("cherche" "fleur")`
`>(filtrage (decoupe '("je pense qu' il n' y a pas de graine"))`
`(append Pronoms Articles Relatifs Conjonctions))`
`("pense" "pas" "graine")`
On ne retient ainsi en quelques sortes que les mots significatifs de la phrase, ceci nous servira dans le prochain chapître.
SOLUTION : Vraiment pas dur, voir la correction en TD.

9 Systèmes experts : un pas en Intelligence Artificielle

La dernière fois, nous avons vu comment faire comprendre quelques rudiments de grammaire française à SCHEME . Cette fois-ci nous allons plus loin, en essayant de lui apprendre les rudiments du raisonnement logique.

Nous allons construire un mini **système expert** (expliquer origine sens du mot), capable de faire des déductions depuis un ensemble de faits et suivant certaines règles qu'il aura apprises.

Cf schema montrant interaction entre différentes parties du SE

Voici un exemple de règles que le SE peut connaître initialement (exemple le plus célèbre dans le domaine) :

```
...
```

Note : si une certaine conclusion peut être obtenue dans plusieurs situations différentes, on aura dans la base autant de règles concernant cette conclusion.

Chaque rèlge peut être stockée en SCHEME de la façon suivante :

```
conclusion (premisse_1 premisse_2 ... premisse_n)
```

les *prémisses* sont les faits nécessaires pour que le fait *conclusion* puisse être déduit.
EXEMPLE : :

```
(lilas (monocotyledon - rhizome)
```

Ici le symbole "-" est utilisé pour indiqué qu'il faut que `rhizome` soit faux pour que `lilas` puisse être déduit.

La base de règles dans le cas de notre exemple est ainsi définie :

```
(define BR '(
  phanerogame (fleur graine)
  sapin (phanerograme graine-nue)
  ...
  collibacille (- feuille - fleur - plante)
))
```

Certaines conclusions des règles ci-dessus sont des **faits intermédiaires** que le programme doit utiliser pour déduire des faits qui nous intéresse plus particulièrement, ie qui font parti de **buts** que l'on s'est préalablement fixés. On définit la liste des buts possibles du raisonnement (ie des conclusions qui nous intéressent vivement) :

```
(define BUTS '(
   sapin muguet anemone lilas mousse
   fougere algue champignon collibacille))
```

9.1 Apprendre à raisonner

La méthode de raisonnement la plus simple est de partir d'un ensemble de faits (connus par l'observation) et de demander au programme s'il peut en déduire un des buts qui nous intéressent. Une méthode naturelle est de raisonner en "**marche avant**", ie de passer des faits aux conséquences par l'intermédiaire de la BR. On raisonne par **saturation**, au sens où toutes les règles sont examinées afin de connaître tous les faits qui peuvent être déduits, même si certains ne seront pas utiles dans la déduction du but final.

EXEMPLE : supposons qu'on cherche à identifier une espèce botanique dont l'observation nous apprend les faits suivants : elle a un rhizome, une fleur, des graines à un cotylédon. Cet ensemble de faits se représente :

```
(rhizome fleur graine 1-cotyledon)
```

Les déductions s'enchainent de la façon suivante :

```
(fleur graine) => phanerogame
phanerogame 1-cotyledon => monocotyledon
(monocotyledon rhizome) => muguet
```

Le dernier fait est un des buts possibles, ce qui termine donc la recherche.

Voyons comment écrire la fonction `deduire` correspondant en SCHEME : celle-ci doit prendre en entrée la BR, les BUTS à atteindre, les faits initiaux vrais et les faits initiaux faux. EXEMPLE :

```
> (deduire '(rhizome fleur graine 1-cotyledon) '() BR BUTS)
muguet
> (deduire '(fleur graine 2-cotyledon) '() BR BUTS)
anemone
> (deduire '(chlorophile) '(fleur feuille rhizome BR BUTS)
()
```

Dans le dernier cas, on ne peut atteindre aucun but. Les faits initiaux sont insuffisants.

Le fonctionnement de `deduire` peut être schématisé ainsi :

```
(define (deduire faits-v faits-f regles buts)
  Tant que (aucun but déduit) faire
    {
     Pour chaque regle R de BR
       {
        R est applicable si
          (   pour tout premisse faux PF de R, PF est dans faits-faux
           et pour tout premisse vrai PV de R, PV est dans faits-vrais )
        Si R est applicable alors ajouter sa conclusion a faits-vrais
       }
     Si aucun nouveau fait deduit de BR, alors renvoyer faux
    }
  Renvoyer le but deduit
```

Pour écrire cette fonction on a besoin de diverses petites fonctions : la fonction `applicable ?` teste si une règle de la BR est applicable. Dans le cas positif, elle renvoie sa conclusion et sinon `#f` :

```
(define (applicable? regle faits-v faits-f)
    (if (premisses-valides? (cadr regle) faits-v faits-f)
        (car regle)
        #f))
```

La fonction `premisses-valides ?` vérifie si un ensemble de prémisses est présent dans la base de faits (`faits-v,faits-f`) :

```
(define (premisses-valides? premisses faits-v faits-f)
  (or (null? premisses)                ; on a verifie tous les premisses
      (and (eq? '- (car premisses))    ; premisse negatif
           (membre (cadr premisses) faits-f)
           (premisses-valides? (cddr premisses) faits-v faits-f))
      (and                             ; premisse positif
           (membre (car premisses) faits-v)
           (premisses-valides? (cdr premisses) faits-v faits-f))))
```

La procedure `nouveau-fait ?`, cherche un nouveau fait en examinant l'ensemble des règles de la BR. On teste si chaque règle est applicable, jusqu'à en trouver une, et dans ce cas on renvoie le nouveau fait déduit, sinon on renvoie faux :

```
(define (nouveau-fait? base-regles faits-v faits-f)
    (if (null? regles) #f
        (let ((fait (applicable? base-regles)))
             (if (and fait (not membre fait faits-v))
                fait
             (nouveau-fait? cddr base-regles)))))
```

NOTE : `applicable` est appelée avec l'ensemble des règles non-encore examinée, mais cette fonction ne regarde que la première règle (accés à `car` et `cdr` seulement).

Il ne reste plus qu'à écrire la fonction `deduire` qui, comme dit précédemment, essaye d'appliquer les règles de déductions tant qu'aucun but n'est atteint et que de nouveaux faits peuvent être déduits :

```
(define (deduire faits-v faits-f regles buts)
    (let ((fait (nouveau-fait? regles faits-v faits-f)))
         (cond ((not fait) #f)            ; on ne peut plus rien deduire
               ((membre fait buts) fait)  ; on a atteint un but
               (else (deduire (cons fait faits-v) faits-f regles buts)))))
```

9.2 Apprendre à communiquer

On peut utiliser les **fonctions d'analyse grammaticale vues au chapître précédent** pour permettre à l'utilisateur d'établir un **dialogue** avec le programme de déduction. En effet, l'utilisateur peut vouloir interagir avec le système dans le but de **modifier la base de règles, les faits initiaux connus ou les buts à atteindre**. Par exemple, si le système n'arrive à atteindre aucun but, on peut trouver utile de lui ajouter de nouvelles règles ou de nouveaux faits.

NOTE : les objets manipulés par les fonctions d'analyse grammaticale sont de type chaîne alors que les objets manipulés par les fonctions de déduction sont de type symbole. Pour transformer un objet d'un type en un autre on se sert des fonctions `symbol->string et string->symbol`.

La fonction `session` va gérer une seance de déduction pendant laquelle on cherche à faire raisonner le SE :

```
(define (session faits-v faits-f regles buts)
 (let ((chaine (read)))   ; lit une phrase au clavier
   (if (string=? "fin" chaine) (display "Au revoir")
      (let ((ph (filtrage decoupe chaine)))
        (cond
          ((string=? "pos" (radical? (car ph)))
              (let* ((nouveaux-faits (ajoute-faits (cdr ph) faits-v faits-f)))
                    (set! faits-v (car nouveaux-faits))
                    (set! faits-f (cadr nouveaux-faits))))
              (display "nouveaux faits enregistres"))
          ((string=? "cherch" (radical? (car ph)))
              (set! buts (ajoute-buts (cdr ph) buts))
              (display "nouveaux buts enregistres"))
          ((string=? "trouv" (radical? (car ph)))
              (let ((conclusion (deduire faits-v faits-f)))
                   (ajoute-faits (list conclusion) faits-v faits-f))
              (display (string-append "je deduis " (symbol->string conclusion))))
          ((string=? "affich" (radical? (car ph)))
              (cond ((string=? "faits" (cadr ph)) (display faits-v faits-f))
                    ((string=? "buts" (cadr ph)) (display buts))
                    ((string=? "regles" (cadr ph)) (display regles))
                    (else (display "je ne comprends pas"))))
          ....
        ) ; fin des cas traites
        (session faits-v faits-f regles buts) ; lit la phrase suivante
      ))))
```

La fonction `set!` remplace la valeur de son 1er argument par celle renvoyée par l'évaluation de son 2e argument. Le but de cette fonction est de modifier ds l'environnement courant la valeur d'une variable.

Voici un à quoi peut ressembler une "séance de devination" :

```
>(session '(1-cotyledon) '() BR '())
"je cherche une plante ou une anemone ou un lilas .... ou un colibacille"
```

```
nouveaux buts enregistres
"je pose qu' il y a un rhizome une fleur et une graine"
nouveaux faits enregistres
"affiche les faits"
(graine fleur rhizome 1-cotyledon)
"que trouves tu ?"
je deduis muguet
```

Il reste maintenant à gérer les quatre volontés de l'utilisateur, ie à écrire les diverses fonctions de modification des connaissances du système mentionnées ci-dessus.

Avant d'ajouter un but on vérifie qu'il n'était pas déjà présent :

```
(define (ajoute-buts buts listbuts)
   (if (null? buts) listbuts
       (let ((1er-but (string->symbol (car buts))))
            (if (membre 1er-but listbuts) (ajoute-buts (cdr buts) listbuts)
          (cons 1er-but (ajoute-buts (cdr buts) listbuts))))))
```

La fonction `ajoute-faits` reçoit une liste de faits potentiellement nouveaux (dans laquelle les faits négatifs sont précédés du mot "pas") et les listes de faits vrais et faux déjà connus. La fonction renvoie une paire contenant dans son `car` les faits vrais et dans son `cdr` les faits faux, complétés par les nouveaux faits.

```
(define (ajoute-faits faits faits-v faits-f)
    (cond ((null? faits) (cons faits-v faits-f))
          ((string=? "pas" (car faits))
             (let* ((1er-fait (string->symbol (cadr faits)))
                    (faits-vf (ajoute-faits (cddr faits) faits-v faits-f)))
               (if (membre 1er-fait )
                   faits-vf ; on n'ajoute pas le 1er-fait
                   (cons (car faits-vf) (cons 1er-fait cadr faits-vf)))))
          (else ....
                ; cas symetrique, on a ici un fait positif
)))
```

EXERCICE : Comment modifier la fonction `seance` pour pouvoir enlever des faits ou des buts
SOLUTION : Il suffit de rajouter des cas au `cond` principal, on teste que le 1er mot de la phrase filtree est "enleve" puis selon que le suivant est "fait" ou "but" on oriente vers des fonctions `enleve-but` ou `enleve-fait`, qui sont assez facile à écrire

EXERCICE : Plus difficile : comment permettre l'ajout et la suppression de règles ? Par exemple on aimerait pouvoir dire :
`"je sais que si on a un rhisome et une graine alors on a une plante"`
et que le système comprenne qu'on veut ajouter la règle `plante (rhisome graine)` .
SOLUTION : .