Introduction à la programmation en Common Lisp

Francis Leboutte

Le 26 novembre 2002
Mise à jour mineure : le 1er février 2004

www.algo.be

IDDN.BE.010.0093308.000.R.C.2001.035.42000

Table des matières

1. INTRODUCTION 3

2. ORDINATEUR ET LANGAGE DE PROGRAMMATION 3

3. PREMIER CONTACT, PREMIER PROGRAMME 4

3.1. ABSTRACTION PROCEDURALE 5
3.2. ACTIVITES ET EXERCICES 6

4. CONCEPTS DE BASE 7

4.1. EXPRESSION LISP 7
4.2. LA BOUCLE D'INTERACTION LECTURE-EVALUATION-ECRITURE 8
4.3. L'EVALUATION D'UN APPEL DE FONCTION LISP 9
4.3.1. L'évaluation d'un atome 9
4.3.2. L'évaluation d'un appel de fonction 9
4.3.3. Règle de l'évaluation d'un appel de fonction 10
4.4. L'OPERATEUR QUOTE BLOQUE L'EVALUATION 10

5. DEFINITION D'UNE FONCTION 11

5.1. EXEMPLES 12
5.2. VARIABLES LEXICALES ET DYNAMIQUES 13
5.3. SYNTAXES COMPAREES 14

6. L'OPERATEUR D'AFFECTATION 15

7. TYPES D'OPERATEURS 16

8. VARIABLES LOCALES ET GLOBALES 16

9. DONNEES, LISTE 18

9.1. NOTES 21
9.2. EXERCICES 21

10. EXPRESSIONS CONDITIONNELLES ET LOGIQUES 22

10.1. VALEURS DE VERITE, PREDICATS 22
10.2. OPERATEURS CONDITIONNELS 24
10.3. EXPRESSIONS LOGIQUES 26
10.4. EXERCICES 27

11. OPERATEURS D'ITERATION 28

11.1. EXERCICES 31

12. FONCTIONS D'ENTREE-SORTIE 32

12.1. LECTURE 32
12.2. ECRITURE 32

13. RECURSION 34

13.1. EXERCICES 35

14. PARAMETRAGE DES FONCTIONS 35

1. Introduction

Ce document s'adresse autant à des débutants en Common Lisp qu'à des débutants en programmation; sa lecture ne demande aucun pré-requis. Etant une introduction à la programmation, seule une petite partie du Common Lisp est présentée, celle utile à l'apprentissage de quelques principes programmation.

Pourquoi avoir choisi le Common Lisp (CL ou Lisp en bref) pour une introduction à la programmation? D'un point de vue pédagogique, le Lisp offre un certain nombre d'avantages. La syntaxe du Lisp est simple et s'apprend très rapidement. Le Lisp est un langage interactif; ceci veut dire qu'en pratique, lorsqu'on écrit un programme dans un environnement de programmation Lisp, l'interaction avec la machine est immédiate et permanente : à tout moment le programmeur peut tester ce qu'il est en train de faire, par petits morceaux, un peu à la manière de ce qui se passe lors de l'utilisation d'une calculette[1]. Le Common Lisp est richement doté en terme de structures de donnée et d'opérateurs prédéfinis. Cette simplicité de la syntaxe et de l'interaction, ainsi que la richesse du langage, font que l'étudiant peut **se concentrer sur ce qui est vraiment important, c'est-à-dire la représentation et la résolution des problèmes posés.** Un autre intérêt du Common Lisp est que c'est un langage multi-paradigmes : il permet la programmation impérative, la programmation fonctionnelle et la programmation orientée-objets.

Voir la page WEB www.algo.be/clr.html pour plus d'information sur le Common Lisp : caractéristiques, utilisation professionnelle, environnements de programmation (dont certains gratuits), ressources et bibliographie (en particulier, le Traité de Programmation en Common Lisp de Robert Strandh et Irène Durand, également disponible en ligne).

2. Ordinateur et langage de programmation

Un langage de programmation est un langage que comprend un ordinateur. Il permet de communiquer des instructions et de l'information (des données) à un ordinateur afin de lui faire exécuter une tâche particulière. Instructions et données constituent le programme.

En plus d'être un moyen de communication avec un ordinateur, un langage de programmation possède deux autres fonctions essentielles. Premièrement, c'est un moyen de communication entre programmeurs (les programmeurs passent une grande partie de leur temps à lire des programmes). Deuxièmement, un langage de programmation nous permet de représenter et d'organiser nos idées au sujet des tâches à accomplir par un ordinateur. Vu sous cet angle, on devine que les langages de programmation diffèrent par leur pouvoir d'expression[2], c'est-à-dire par la facilité avec laquelle ils permettent de programmer une solution à un problème donné. Le Lisp est un langage dont le pouvoir d'expression est très grand, ce qui procure un avantage technologique à ses utilisateurs [3].

---

[1] Le Common Lisp est un langage *interactif* permettant une compilation *incrémentale* des composants d'un programme

[2] On parle aussi du niveau d'un langage

[3] Une affirmation difficile à démontrer rapidement : même si vous allez trouver quelques arguments dans ce sens tout au long de cette introduction au Common Lisp, il vous faudra

3. Premier contact, premier programme

Supposons que vous soyez devant un ordinateur et qu'un environnement de programmation Lisp ait été lancé. Vous avez devant les yeux la fenêtre de **l'interacteur Lisp** qui attend que vous introduisiez quelque chose et est prêt à vous répondre. Le fait que l'interacteur Lisp attend est matérialisé par une **invite** (*prompt* en anglais), par exemple le signe > qui sera utilisé dans la suite de ce cours. Introduisez à l'aide du clavier l'**expression Lisp** suivante : *(+ 2 3)*. La réponse s'inscrit immédiatement à l'écran : *5*. C'est le résultat de l'**évaluation** (calcul) de cette expression. Dans la suite, ce type d'interaction sera représenté comme ceci :

```
> (+ 2 3)
5
```

On peut utiliser le Lisp[4] comme une calculette et calculer interactivement toute sorte d'expressions numériques, y compris des expressions imbriquées :

```
> (+ 1 3 5)
9

> 9
9

> (* 3 3)
9

> (+ (* 3 3) 3)
12

> (- 10 5 4)
1

> (/ 4 5)
4/5

> (/ 4 5.0)
0.8

> (/ (* 2/3 15) 2)
5

> (expt 2 4)
16

> (round 33.45)
33
0.45
```

---

lire quelques uns des documents et livres cités dans la page http://www.algo.be/clr.html pour vous convaincre de sa véracité.

[4] Dans le suite, le mot *Lisp* sera aussi utilisé comme abréviation de l'interacteur Lisp

```
> (round 33.6)
34
-0.4
```

Ces exemples simples illustrent déjà quelques concepts de programmation en CL :

- dans ces interactions avec l'interacteur Lisp, on a utilisé des **fonctions** Lisp existantes[5], comme la fonction +, dans des expressions Lisp comme *(+ 1 3 5)*. On remarque :
  - une expression Lisp comme *(+ 1 3 5)* commence et fini par une parenthèse, c'est-à-dire est une **liste** (***list***);
  - l'opérateur (la fonction +) précède les arguments que sont 1, 3 et 5 : le Lisp utilise la notation **préfixe**.
- Il n'y a pas d'ambiguïté dans les expressions numériques en CL. Comparez (* 2 (+ 3 4)) avec l'expression infixe[6] 2 * 3 + 4
  Autre intérêt de la notation préfixe : le nombre d'arguments d'un opérateur qui peut être 0, 1 ou plus.
- Quelques uns des **types de donnée**[7] numériques prédéfinis en Lisp : nombre entier (***integer*** – en Common Lisp, ce type est sans limite de grandeur[8]), rationnel (***ratio***)[9] et réel avec point décimal[10] (***float***)
- Le type de donnée symbole (***symbol***), utilisé ici pour nommer les opérateurs; par exemple les symboles +, * et *round.*
- Un nombre est un objet auto-évaluant.
- L'évaluation d'une expression Lisp peut donner lieu à plusieurs valeurs, comme celle de *(round 33.45)* qui donne le nombre entier le plus proche de 33.45 et le reste.

3.1. Abstraction procédurale

Pour convertir un montant en FB (franc belge) en €, il suffit de le diviser par 40.3399; par exemple, pour convertir 50 FB :

---

[5] ou primitives du langage
[6] l'opérateur est placé entre les arguments
[7] Tout langage de programmation comporte un certain nombre de types de donnée prédéfinis. Ces types permettent de représenter et manipuler les données dans un programme.
[8] Dans la plupart des langages, l'entier le plus grand est limité par le nombre de bits dans un mot (une place) de la mémoire de la machine; par exemple le mot de 32 bits sur un Pentium, soit un entier positif le plus grand égal à (- (expt 2 32) 1)), ce qui fait 4294967295 (à supposer qu'aucun bit ne soit utilisé pour le signe ou autre chose). Dans un langage comme le C ou Java, l'addition ou la multiplication de 2 nombres entiers qui sont dans l'intervalle autorisé peut donner une erreur si le résultat est hors de l'intervalle autorisé. Ce ne sera pas le cas en Common Lisp où il n'y a pas de limite sur la taille des entiers.
[9] rapport entre deux nombres entiers, un type de donnée rarement disponible dans les langages de programmation. Comparez du point de vue de la précision de la représentation en mémoire : 1/3 et 0.33333334
[10] ou encore nombre décimal, flottant et nombre à virgule

```
> (/ 50 40.3399)
1.2394676
```

En Common Lisp comme dans n'importe quel autre langage, il n'y a pas de fonction primitive pour ce faire; par compte il est aisé de **définir** une nouvelle fonction *FB-en-euros* qui fera la conversion :

```
> (defun FB-en-euros (x)
    (/ x 40.3399))
FB-en-euros
```

On peut maintenant **appliquer** cette fonction à 50 ou à n'importe quel autre montant :

```
> (FB-en-euros 50)
1.2394676

> (FB-en-euros 1000.5)
24.801746
```

En définissant une tâche et en la nommant de la sorte, on a créé une abstraction procédurale. C'est un mécanisme essentiel dans l'écriture de programmes et la production de logiciels.

3.2. Activités et exercices

- Visiter la page de présentation du Common Lisp http://www.algo.be/clr.html, en particulier les sections consacrées aux livres et manuels disponibles en ligne.
- Exploration de l'environnement de programmation Lisp choisi, en particulier des principales commandes de l'éditeur.
- Utiliser l'interacteur comme une calculette. Même chose dans l'éditeur en utilisant la commande d'évaluation. Comparer ces deux façons de faire.
- Essayer les fonctions numériques simples (+ - * /) avec 0, 1 et plusieurs arguments.
- Dans l'éditeur créer un fichier contenant la fonction *FB-en-euros.* Evaluer[11] et appliquer cette fonction. Comparer l'état de l'environnement Lisp après l'exercice précédent et après celui-ci.
- Sur le modèle de la fonction *FB-en-euros*, définir une fonction périmètre d'un rectangle; compléter :
  (defun périmètre (l h)
    ...)
- Compléter cette définition :
  (defun jours-dans-année ()
    ...)
- Attention au **style de programmation** :
  - le retour à la ligne et l'indentation des lignes aident à la lecture du code (l'indentation des lignes est automatique dans un éditeur Lisp)
  - ne mettre un espace que quand c'est nécessaire (un seul espace suffit).

[11] ou compiler incrémentalement

Les exemples qui suivent ne provoqueront pas d'erreur d'exécution mais réduisent la **lisibilité** :

```
(defun FB-en-euros (x) (/ x 40.3399))

( defun FB-en-euros ( x )
   ( / x 40.3399))

(defun FB-en-euros     (x)
   (/ x 40.3399))
```

Par compte, oublier un espace comme ci-dessous, entre un opérateur et son premier argument, constitue une **erreur de syntaxe** :

```
(defun FB-en-euros (x)
   (/x 40.3399))
```

4. Concepts de base

4.1. Expression Lisp

Voici quelques expressions Lisp valides :

- `12`
- `3.14`
- `(+ 1 2)`
- `(* 1 2 3)`
- `(/ 1 (* 1 2 3))`
- ```
  (defun FB-en-euros (montant)
      (/ montant 40.3399))
  ```
- `(FB-en-euros 50)`

Une expression est soit un atome, comme *12* ou *montant*, soit une liste d'expressions. Une liste d'expressions est faite de zéro, une ou plusieurs *expressions* entre parenthèses.

Tous les programmes Lisp sont constitués d'expressions de la sorte.[12] Vous venez donc d'apprendre l'essentiel de la **syntaxe**[13] du Lisp.

[12] Cette simplicité syntaxique est un des points forts du Lisp. La plupart des autres langages ont une syntaxe plus compliquée; comme le C par exemple, qui par ailleurs, nécessite l'utilisation d'un plus grand nombre de délimiteurs syntaxiques différents :

, . ( ) ; [ ] { } ->

contre les deux parenthèses seulement en Lisp.

[13] Syntaxe d'un langage : ensemble des règles à appliquer pour écrire des instructions valides
  ```

Le premier élément d'une expression de type liste destinée à être calculée (exécutée) est un **opérateur**. Le plus souvent un opérateur est une **fonction** comme dans les exemples suivants :

- `(+ 1 2)`
- `(* 1 2 3)`
- `(/ 1 (* 1 2 3))`
- `(FB-en-euros 50)`

Ces exemples sont des **appels de fonction** (ou **applications de fonction**). Un appel de fonction s'écrit toujours selon la syntaxe suivante :

(fonction argument1 argument2 ...)

un argument étant une expression Lisp.

On verra dans le chapitre suivant comment tout appel de fonction est évalué[14] selon une règle unique.

Remarques

- La définition d'une expression est une définition **récursive** (la définition d'une expression est formulée en terme d'elle-même).
- Au lieu d'expression Lisp, on utilise souvent le mot **forme,** quand on veut insister sur le fait que l'expression est destinée à être calculée.

4.2. La boucle d'interaction lecture-évaluation-écriture

Une interaction avec le Lisp comme

```
> (+ 2 3)
5
```

correspond à un cycle de ce qui est appelé la **boucle d'interaction** lecture-évaluation-écriture (*Read-Eval-Print*) de l'environnement Lisp et composé de 3 étapes bien distinctes :

1. *Read :* lecture d'une expression, étape réalisée par le *lecteur* (*reader)*
2. *Eval :* évaluation de l'expression, étape réalisée par l'*évaluateur* (*evaluator)*
3. *Print :* impression du résultat du calcul dans la fenêtre d'interaction, étape réalisée par l'*imprimeur - printer*.

Dès qu'un cycle est terminé, l'invite (>) réapparaît, ce qui signifie que l'environnement Lisp est prêt pour une autre interaction, par exemple :

```
> 5
5
```


[14] autrement dit, le résultat de l'application d'une fonction est calculé

4.3. L'évaluation d'un appel de fonction Lisp

4.3.1. L'évaluation d'un atome

Certains atomes sont évalués à eux-mêmes; c'est le cas des nombres (atomes numériques) :

```
> 23.5
23.5
```

Les chaînes de caractères[15] et les 2 symboles spéciaux t et *nil*, dont on verra l'utilité plus loin, sont d'autres exemples d'objets auto-évaluants :

```
> "Suzy"
"Suzy"

> nil
nil

> t
t
```

Une valeur (une donnée) peut être *associée* à un symbole (voir plus loin, le chapitre *L'opérateur d'affectation*). L'évaluation d'un symbole produit la valeur associée (s'il n'y a pas de valeur associée, l'évaluation d'un symbole produit une erreur) :

```
> vitesse
60
```

4.3.2. L'évaluation d'un appel de fonction

Tout appel de fonction répond à une **règle unique d'évaluation**.
Par exemple, dans le cas de cet appel qui est constitué de la fonction + suivi de 3 arguments (3 expressions atomiques) :

```
> (+ 1 2 3)
6
```

Que fait le Lisp lors de l'évaluation de cet appel :

1. il retrouve la fonction dont le nom est + (1er élément)
2. il évalue chacun des éléments restants de la liste (les arguments de la fonction), de gauche à droite. Dans cet exemple, les arguments sont auto-évaluants
3. il applique la fonction trouvée en 1 aux valeurs trouvées en 2.

Ce processus est récursif; ainsi dans l'évaluation de

```
(+ 12.5 (* 2.5 2))
```

les étapes sont :

[15] un autre type de donnée (***string***)

1. retrouver la procédure associée au symbole +
2. évaluer l'argument 12.5, soit : 12.5 => **12.5**
3. évaluer l'argument (* 2.5 2), soit :
   3.1. retrouver la procédure associée au symbole *
   3.2. évaluation du 1er argument : 2.5 => 2.5
   3.3. évaluation du 2ème argument : 2 => 2
   3.4. l'application de la procédure * aux deux arguments donne **5**
4. appliquer la procédure + aux valeurs 12.5 et 5, soit 17.5

4.3.3. Règle de l'évaluation d'un appel de fonction

Tous les appels de fonction ont cette syntaxe :

(fonction argument1 argument2 ...) 0, 1 ou plusieurs arguments
Un argument étant une expression Lisp

Tous les appels de fonction sont évalués selon cette règle :

1. retrouver la fonction associée au premier élément de la liste
2. évaluer les arguments de gauche à droite
3. appliquer la fonction trouvée aux valeurs des arguments.

4.4. L'opérateur quote *bloque l'évaluation*

La liste est un autre type de données disponible dans le Common Lisp. Une liste est représentée comme zéro, un ou plusieurs éléments entre parenthèses. Par exemple,

```
(mes 2 "chiens")
```

est une liste.

L'évaluation de cette liste donne une erreur :

```
> (mes 2 "chiens")
erreur: mes est une fonction non définie
```

En effet, l'évaluateur cherche la procédure associée à *mes*, or *mes* n'a pas de procédure associée (*mes* n'est pas un opérateur), d'où l'affichage d'un message d'erreur. Par contre :

```
> (quote (mes 2 "chiens"))
(mes 2 "chiens")
```

L'opérateur *quote* prend un seul argument. Contrairement à la règle d'évaluation d'un appel de fonction, l'argument de *quote* n'est pas évalué (*quote* fait exception à la règle et est dit **opérateur spécial**). L'application de *quote* à un argument renvoie simplement l'argument non évalué, autrement dit *quote* bloque l'évaluation de l'argument.

Autre exemple. La fonction *first* renvoie le premier élément d'une liste :

```
> (first (a b c))
erreur: a est une fonction non définie
```

Par contre :

```
> (first (quote (a b c))
a
```

De même :

```
> (quote (a b c))
(a b c)

> (quote a)
a
```

L'opérateur *quote* étant souvent utilisé, le Lisp comporte une abréviation de *quote*, celle du caractère apostrophe (un caractère dit ***caractère macro*** en *Lisp*) :

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

> (first '(a b c))
a

> 'a
a
```

La fonction *list* est l'opérateur de construction d'une liste :

```
> (list 'a 'b 'c (list 1 2 3))
(a b c (1 2 3))
```

Remarque

C'est le ***reader*** (1ère étape de la boucle *Read-Eval-Print)* qui réalise l'*expansion* des caractères macros de telle sorte que, par exemple, '(a b c) devienne (*quote* (a b c)). De façon générale, le reader convertit '<expression> en (*quote* <expression>), en préparation du travail réalisé par l'évaluateur (***eval***, 2ème étape de la boucle d'interaction).[16]

5. Définition d'une fonction

defun[17] est un opérateur du langage Common Lisp qui permet au programmeur de définir une nouvelle fonction (une nouvelle abstraction c'est-à-dire quelque chose qui a un nom et qui correspond à une tâche à accomplir) :

[16] Un programmeur Lisp peut définir ses propres *caractères macros*. Cette possibilité d'attacher un comportement fonctionnel à un caractère permet de définir aisément de nouveaux caractères syntaxiques : c'est une des propriétés du Lisp très appréciée pour implanter un *langage enchâssé* (ou *langage d'extension*).
[17] defun : DEfine FUNction

```
(defun nom (paramètre-1 paramètre-2 ...)
  forme-1
  forme-2
  ...
  forme-n)
```

où:

- *nom* est le nom de la fonction (un symbole),
- *(paramètre-1 paramètre-2 ...)* est la liste des paramètres (une liste de 0, 1 ou plusieurs symboles)
- *forme-1 forme-2 ... forme-n* est une suite d'expressions Lisp (0, 1 ou plusieurs) et constitue le **corps** de la fonction. La valeur retournée par une application de fonction définie avec *defun* est celle de la dernière expression du corps de la fonction.

5.1. Exemples

```
> (defun FB-en-euros (montant)
    (/ montant 40.3399))
FB-en-euros

> (FB-en-euros 50)
1.2394676
```

Même fonction, mais le résultat étant arrondi aux centimes[18] :

```
> (defun FB-en-euros (x)
    (/ (round (* 100
                 (/ x 40.3399)))
       100.0)))
FB-en-euros[19]

> (FB-en-euros 50)
1.24
```

Une fonction sans argument :

```
> (defun jours-dans-année ()
    (+ (* 30 4)
       (* 31 7)
       28))
jours-dans-année
```

[18] Piste :

```
> (round (* 100 1.2394676))
124
```

[19] Notez :

- l'indentation en relation avec la structure logique de la fonction
- l'utilisation de 100.0 pour obtenir un résultat de type *float* (propagation du type)

```
> (jours-dans-année)
365
```

5.2. Variables lexicales et dynamiques

Lors de l'évaluation d'un appel de fonction, le corps de la fonction est évalué avec les noms *paramètre-1, paramètre-2, ...* accessibles en tant que variables de programme (variables locales[20]**)**. Au début de l'évaluation du corps de la fonction, chacune de ces variables est associée à une valeur initiale qui est le résultat de l'évaluation de l'argument correspondant dans l'appel de fonction. On dit que la variable est **liée** (*bound*) à une valeur.

Soit la fonction FB-en-euros redéfinie de la sorte :

```
> (defun FB-en-euros (x)
    (print x)
    (/ x 40.3399))
FB-en-euros
```

Par exemple, lors de l'évaluation de l'appel (FB-en-euros (* 2 50)), la variable (paramètre) *x* est *initialement* liée à *100*, le résultat de l'évaluation de *(* 2 50)* . L'évaluation de la 1ère forme du corps *(print x)* a pour effet d'écrire *100* dans la fenêtre de l'interacteur (*print* est une fonction d'écriture). L'évaluation de la 2ème forme du corps *(/ x 40.3399)*, où *x* est remplacé par 100, donne 2.4789352 . Comme c'est la dernière forme, c'est aussi la valeur retournée comme résultat de l'évaluation de l'appel *(FB-en-euros (* 2 50))* :

```
> (FB-en-euros (* 2 50))
100
2.4789352
```

Ce **lien** (*binding*) entre x et sa valeur n'existe que durant l'évaluation de l'appel. Lorsque celle-ci est terminée, le lien disparaît. De plus la variable x et, à fortiori, son lien, ne sont pas visibles dans les corps des fonctions qui seraient appelées dans forme-1, forme-2, etc. (**sous-fonctions**); de fait :

```
> (defun FB-en-euros (x)
    (test-accès-x)
    (/ x 40.3399))
FB-en-euros

> (defun test-accès-x ()
     (print x))
test-accès-x

> (FB-en-euros 100)
erreur: dans test-accès-x, la variable x n'a pas de valeur
```

C'est pourquoi on parle de **variable lexicale**, c'est-à-dire une variable dont la portée (*scope*) est limitée au texte (portion de code) où elle est définie : elle n'est accessible que

[20] Variable locale : variable dont l'accès est limité au code où elle est définie; par exemple au corps de la fonction où la variable est définie.

dans le texte où elle est définie (dans cet exemple, dans le corps de la définition de la fonction où elle définie en tant que paramètre de la fonction).

En Lisp, **par défaut, toute variable locale est une variable lexicale** (comme la variable x ci-dessus). Moyennant une déclaration appropriée, on peut faire d'une variable locale une **variable dynamique**[21], c'est-à-dire une variable dont la portée n'est pas limitée à la portion de code où elle est définie, mais s'étend dans les sous-fonctions[22] :

```
> (defun FB-en-euros (x)
    (declare (special x))
    (test-accès-x)
    (/ x 40.3399))
FB-en-euros

> (FB-en-euros 100)
100
2.4789352
```

L'appel à test-accès-x ayant pour effet d'écrire 100 avant d'écrire le résultat final 2.4789352

Note
On voit que les symboles sont utilisés comme noms[23] de variables. C'est pourquoi un symbole qui est utilisé comme donnée doit être *quoté*, sinon il sera traité comme une variable devant avoir une valeur :

```
> 'pomme
pomme

> pomme
erreur: la variable pomme n'a pas de lien

> (list 'pomme 'cerise)
(pomme cerise)
```

5.3. Syntaxes comparées

Soit à définir une fonction qui ajoute 1 à un nombre.

En Pascal :

```
function plus1 (x : integer) : integer;
begin
    plus1 := x + 1;
end;
```

En Lisp :

[21] aussi appelée **variable spéciale**
[22] et les sous-fonctions des sous-fonctions et ainsi de suite
[23] Dans d'autres langages, on parlerait d'identifiant

```
(defun plus1 (x)
  (+ x 1))
```

Notez les 6 délimiteurs syntaxiques Pascal utilisés (;, :, begin, end et les 2 parenthèses) contre 2 seulement en Lisp.

En Pascal, comme dans de nombreux langages :

- pour utiliser ou tester la fonction, il faut encore l'intégrer dans un programme et compiler ce programme
- les déclarations de type de donnée (*integer*) sont obligatoires
- le type est associé aux variables et aux fonctions.

En Lisp :

- pour utiliser ou tester la fonction, il suffit de la compiler (compilation incrémentale)
- les déclarations de type de donnée sont optionnelles
- le type est associé aux valeurs (le type est *manifeste*, c'est comme si chaque valeur était accolée d'une étiquette où son type serait inscrit).

6. L'opérateur d'affectation

Il est possible d'associer une valeur à un symbole; une des façons de faire est d'utiliser l'opérateur d'affectation *setf*. Supposons qu'aucune valeur ne soit encore affectée au symbole *vitesse* :

```
> vitesse
erreur: la variable vitesse n'a pas de valeur

> (setf vitesse 60)
60

> vitesse
60

> (setf vitesse 100)
100

> vitesse
100
```

L'opérateur *setf* a 2 particularités :

1. Il n'évalue pas le premier argument, donc il ne respecte pas la règle générale d'évaluation d'une expression vue plus haut. *setf*, comme *quote* et *defun*, fait partie des **opérateurs** dits ***spéciaux*** : ceux-ci ont ***leurs propres règles d'évaluation***. Parmi les primitives du *Lisp* (c'est-à-dire les opérateurs définis par la norme du langage), ces opérateurs spéciaux sont minoritaires. Bien entendu, les fonctions définies à

l'aide de *defun* respectent la règle générale d'évaluation.

2. Après avoir renvoyé une valeur, l'opérateur *setf* a un effet persistant en ce sens qu'il a associé de façon permanente une valeur à un symbole : c'est le principe de l'affectation; on dit aussi que *setf* a un *effet de bord* (ou un effet secondaire) [24].

Ce faisant, on a définit une variable globale dont le nom est vitesse, une notion qui sera expliquée plus en détail dans un chapitre ultérieur.

7. Types d'opérateurs

Il y a trois types d'opérateurs en Lisp, les **fonctions**, les **opérateurs spéciaux** (*defun, quote, setf, ...*) et les **macros**. Parmi les opérateurs prédéfinis du langage CL, les fonctions sont largement majoritaires. Il n'est pas nécessaire de faire la différence en une macro et opérateur spécial, il suffit de savoir que ce ne sont pas des fonctions et comment les utiliser : macros et opérateurs spéciaux ne suivent pas la règle d'évaluation d'un appel de fonction mais ont chacun leur règle particulière. Un programmeur peut définir ses propres macros.

8. Variables locales et globales

L'opérateur *let* permet de créer des variables locales (lexicales par défaut, comme vu précédemment). Par exemple, au lieu d'écrire *plus-1-et-carré,* comme ceci :

```
(defun plus-1-et-carré (x)
  (* (+ x 1)
     (+ x 1)))
```

il vaut mieux utiliser une variable locale, ce qui évite à l'évaluateur de calculer deux fois la forme *(+ x 1)* :

```
(defun plus-1-et-carré (x)
  (let ((x-plus-1 (+ x 1)))
    (* x-plus-1 x-plus-1)))
```

La syntaxe du *let* est :

```
(let ((var-1 forme-1)
      (var-2 forme-2)
      ...)
  expression-1
  ...
  expression-n)
```

let est un opérateur spécial (n'est pas une fonction). Donc, une expression *(let (...) ...)* a sa propre règle d'évaluation : chaque variable *var-n* est d'abord initialisée selon la valeur de la

[24] Dans le modèle de programmation purement fonctionnel, les fonctions n'ont pas d'effet de bord. La majorité des fonctions *Common Lisp* n'ont pas d'effet de bord.

forme correspondante (*forme-1*, etc. – forme Lisp ou expression Lisp sont synonymes). Autrement dit, évaluer toutes les *forme-n* et ensuite lier les valeurs trouvées aux *var-n* correspondantes. Ensuite les expressions du corps du *let* sont évaluées les unes après les autres, dans l'ordre. La valeur de la dernière expression est retournée comme valeur du *let* (et à ce moment les variables cessent d'exister) :

```
> (let ((x 1)
        (y 2))
    (print x)
    (setf x y)
    (print x)
    "fin")
1
2
"fin"
```

Notez l'utilisation de l'opérateur d'affectation (*setf*) pour modifier le lien de la variable locale x.

*let** est similaire à *let*, sauf que les liens des variables sont construits successivement, ce qui, par exemple, veut dire que dans l'expression *forme-2*, la variable *var-1* peut être utilisée :

```
> (let* ((x 1)
         (y (* 4 x)))
    (print x)
    (setf x y)
    (print x)
    "fin")
1
4
"fin"
```

Une autre sorte de variable est la **variable globale**, une variable qui est visible partout (c'est donc une variable dynamique). En principe, on utilise l'opérateur spécial *defvar* pour définir une variable globale. *defvar* prend 2 arguments; le 1er est le nom de la variable, un symbole qui n'est pas évalué (il ne faut pas le quoté). Le 2ème est une expression dont l'évaluation donnera la valeur initiale de la variable :

```
> (defvar *globale* 2002)
*globale*

> *globale*
2002

> (setf *globale* 2003)
2003

> (defun test-*globale* ()
      *globale*)
```

```
> (test-*globale*)
2003
```

Par convention, le nom d'une variable globale commence et finit par une astérisque, afin de faciliter la lecture et la mise au point des programmes. En fait, il n'est pas nécessaire d'utiliser *defvar* : dès le moment où le 1er argument de setf est un symbole qui n'est pas une variable locale, ce symbole sera considéré comme une variable globale (cette façon de définir une variable globale *implicitement* n'est généralement pas une bonne pratique).

9. Données, liste

Nous avons déjà rencontré quelques uns des types de donnée prédéfinis en Lisp :

- le nombre entier (*integer*), suite de chiffres avec un signe optionnel; par exemple : 2002
- le nombre rationnel (*ratio*) : 4/5
- le nombre décimal (*float*) : 23.45
- le symbole (*symbol*), utilisé par exemple pour nommer les opérateurs et les variables
- la chaîne de caractères (*string*), suite de caractères entre guillemets à l'anglaise : "ANSI Common Lisp".

En Lisp, les entiers et les rationnels sont aussi grands que l'on veut. Les nombres et les chaînes de caractères sont auto-évaluants.

Le Lisp dispose des types de donnée rencontrés dans la plupart des autres langages, comme le nombre entier, le nombre décimal, la chaîne de caractères et d'autres. Par contre, il en possède quelques uns qui sont plus rares, comme le nombre rationnel (*ratio*), le symbole et la liste. La liste (*list*) s'écrit comme une suite d'éléments (données) entre parenthèses. Chaque élément peut être de n'importe quel type, y compris du type liste :

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

> '(a b (1 2))
(a b (1 2))
```

La fonction *list* est un des opérateurs de construction d'une liste :

```
> (list 1 2 3)
(1 2 3)

> (list 'a 'b 'c (list 1 2 3))
(a b c (1 2 3))
```

Une liste peut être vide c'est-à-dire ne contenir aucun élément; la **liste vide** est représentée par () ou par le symbole *nil* :

```
> (list)
nil

> ()
nil

> nil
nil
```

La liste vide () et le symbole *nil* sont identiques. De plus *nil* (symbole ou liste vide) est auto-évaluant.

Il existe de nombreux opérateurs pour construire et manipuler les listes. En voici quelques exemples :

Constructeur :

```
> (list 1 2 3)
(1 2 3)
```

Accesseurs :

```
> (first (list 1 2 3))
1

> (second (list 1 2 3))
2

> (third (list 1 2 3))
3
```

Accesseur général : la fonction *nth* permet d'accéder au n*ème* élément d'une liste (les éléments étant numérotés à partir de 0).

```
> (setf *fruits* (list 'pomme 'poire 'orange))
(pomme poire orange)

> *fruits*
(pomme poire orange)

> (nth 0 *fruits*)
pomme

> (nth 9 (list 0 1 2 3 4 5 6 7 8 9 10))
9
```

rest retourne le reste d'une liste, c'est-à-dire la liste moins son premier élément :

```
> (rest *fruits*)
(poire orange)
```

```
> (rest (rest *fruits*))
(orange)

> (rest (rest (rest *fruits*)))
nil
```

Divers :

```
> (length *fruits*)
3

> (remove 'poire *fruits*)
(pomme orange)

> *fruits*
(pomme poire orange)

> (remove 'poire *fruits*)
(pomme orange)

> *fruits*
(pomme poire orange)

> (find 'poire *fruits*)
poire

> (find 'prune *fruits*)
nil

> (cons 'prune *fruits*)
(prune pomme orange)

> *fruits*
(pomme orange)

> (push 'prune *fruits*)
(prune pomme orange)

> *fruits*
(prune pomme orange)

> (rest *fruits*)
(pomme orange)

> *fruits*
(prune pomme orange)

> (rest (cons 'noix *fruits*))
(prune pomme orange)
```

Remove, comme la plupart des fonctions Lisp, est une fonction au sens propre : elle calcule un résultat uniquement selon ses arguments et sans les modifier (elle n'a pas d'effet de bord). Même chose pour *cons* qui permet de construire une nouvelle liste à partir d'une liste

existante et d'un élément à ajouter en tête de la liste. *Push* est une macro qui combine l'action de *cons* et de *setf*.

9.1. Notes

La liste est utilisée aussi bien pour les données simples que pour les expressions Lisp destinées à être évaluées (ou *formes*) qui constituent les programmes. De fait, en Lisp, il n'y a **pas de distinction entre les programmes et les données**[25] : la liste *(+ 1 2 3)* peut être vue comme une donnée ou comme une forme, selon le contexte. Par exemple :

```
> (+ 1 2 3)
6

> '(+ 1 2 3)
(+ 1 2 3)
```

En Lisp, toute donnée (ou valeur) est représentée par un objet ayant un **type manifeste** ; le type d'un objet est dit manifeste s'il peut être testé et reconnu à l'exécution (le langage est typé dynamiquement). Contrairement aux langages où le type est statique (C, Fortran,...), il n'est pas nécessaire en Lisp de déclarer le type d'une variable. On peut déclarer le type d'une variable, limitant ainsi les valeurs de cette variable à une certaine gamme d'objets (pour des raisons d'optimisation par exemple, le compilateur mettant à profit cette information pour générer un code machine plus efficace).

9.2. Exercices

```
;;; Définir une fonction d'un argument et qui retourne une liste
;;; contenant 2 fois l'argument; compléter :
(defun doublon (x)
  )

;;; fonction de 2 arguments qui en retourne la liste, le 2éme argument
;;; en tête
(defun liste-des-args-inversés (n1 n2)
  )

;;; fonction prenant 1 argument qui est une liste de 2 éléments et qui
;;; retourne une nouvelle liste contenant les 2 éléments dans l'ordre
;;; inverse
(defun inverser-paire (paire)
  )

;;; Fonction de 3 arguments numérique qui en retourne la moyenne
(defun moyenne-de-3 (n1 n2 n3)
  )

;;; Fonction de 1 argument qui est une liste de 3 nombres qui en retourne la moyenne
(defun moyenne-de-triplet (triplet)
  )
```

[25] Avec pour conséquence qu'un programme Lisp peut générer un programme Lisp.

10. Expressions conditionnelles et logiques

10.1. Valeurs de vérité, prédicats

Les 2 valeurs de vérité, *vrai* et *faux*, sont représentées par les 2 symboles spéciaux *t* et *nil* (au lieu de dire valeurs de vérité, on dit aussi valeurs booléennes ou valeurs logiques). Comme *nil*, *t* est un symbole auto-évaluant :

```
> t
t

> nil
nil
```

La fonction *evenp* vérifie si un nombre est pair :

```
> (evenp 4)
t

> (evenp 3)
nil
```

Toute fonction qui retourne une valeur qui doit être interprétée comme valeur de vérité (vrai ou faux) est appelée un **prédicat**.

Un autre prédicat est la fonction = qui vérifie l'égalité de nombres :

```
> (= 5 4)
nil

> (= 5 5)
t

> (= 5 5 5)
t

> (= 5 5 2)
nil

> (= 2 2.0)
t
```

Il existe divers prédicats d'égalité à utiliser selon l'objectif poursuivi. A ce stade, le plus simple est d'utiliser = pour les nombres et *equal* dans les autres cas. *Equal* peut être défini comme un prédicat d'égalité qui retourne *t* si ses arguments sont écrits de la même façon par l'imprimeur (*printer)* :

```
> (equal (list 'pomme 'poire) (list 'pomme 'poire))
t
```

```
> (equal 'pomme (first (list 'pomme 'poire)))
t

> (equal (list 'pomme) 'pomme)
nil
```

Remarquez qu'il n'y a qu'un symbole pomme, peu importe la façon dont le programmeur l'écrit, l'imprimeur l'écrira toujours de la même façon :

```
> 'POMme
pomme

> 'pomme
pomme

> (equal 'pomme 'POMme)
t
```

Par contre, dans une chaîne de caractères, écrire les caractères en majuscule ou en minuscule importe :

```
> "POmme"
"POmme"

> "pomme"
"pomme"

> (equal "POmme" "pomme")
nil

> (equal "pomme" "pomme")
t
```

Une définition de prédicat :

```
> (defun cinq? (nombre)
    (= 5 nombre))
cinq?

> (cinq? 2)
nil

> (cinq? 5)
t
```

Notes

- par convention, on met un ? à la fin du nom d'un prédicat (autre convention, terminer le nom par *–p*).
- Selon le contexte, le symbole nil joue différents rôles : celui de liste vide ou celui du faux logique (valeur de vérité).

Quelques autres prédicats: > , < , >= , <= , null, integerp

```
> (> 2 1)
t

> (> 2 2)
nil

> (>= 2 2)
t

> (> 5.5 2 1)
t

> (> 5.5 2 2)
nil
```

null est un prédicat qui teste si son argument est une liste vide :

```
> (null ())
t

> (null nil)
t

> (null (list 1 2))
nil
```

integerp est un prédicat qui teste si son argument est nombre entier :

```
> (integerp 11)
t

> (integerp 12.5)
nil

> (integerp 'pomme)
nil

> (integerp (list 'pomme 'poire))
nil
```

10.2. Opérateurs conditionnels

Un opérateur conditionnel permet d'exécuter du code sous condition. L'opérateur conditionnel Lisp le plus simple est *if* : il prend 3 arguments (3 expressions successives) et correspond à l'expression conditionnelle classique *si* ... *alors* ... *sinon* ... Sa syntaxe est :

```
(if forme-test
    forme-1
   forme-2)
```

if est un opérateur spécial; une expression *(if ...)* a sa propre règle d'évaluation : la 1ere expression (*forme-test*) est évaluée; si la valeur de *test* est vrai, alors *forme-1* est évaluée et sa valeur est retournée. Sinon c'est *forme-2* qui est évaluée et sa valeur retournée.

Le prédicat *evenp* prend un argument qui doit être un nombre entier et teste si c'est un nombre pair :

```
> (if (evenp 2)
      "pair"
    "impair")
"pair"

> (if (evenp 3)
      "pair"
    "impair")
"impair"
```

Attention, vu que l'argument de *evenp* doit être un entier :

```
> (evenp 10.5)
erreur: 10.5 n'est pas un entier
```

Note
Dans un contexte nécessitant une valeur logique, **n'importe quoi, sauf le symbole *nil*, est considéré comme *vrai*** (seul nil est considéré comme *faux*). Dans les 2 exemples ci-dessous la forme *'jamais* n'est jamais évaluée :

```
> (if 2002
      'toujours
    'jamais)
toujours

> (if nil
      'jamais
    'toujours)
toujours
```

Il existe d'autres opérateurs conditionnels, comme l'opérateur conditionnel *cond*, un *if* généralisé :

```
(defun taux-intérêt (montant)
   (cond ((< montant 0) –10)
         ((< montant 1000) 2)
         ((< montant 5000) 3)
         ((< montant 10000) 4)
         (t 5)))
```

Cette fonction taux-intérêt calcule le taux d'intérêt applicable à un montant donné en fonction de ce montant. Une forme *cond* est constituée d'une suite de clauses évaluées tour à tour. La 1ère partie d'une clause est une forme (forme-test) dont la valeur est considérée comme une valeur de vérité. L'évaluation d'une forme *cond* s'arrête à la première clause dont la forme-test est vrai : l'évaluation du reste de la clause donne la valeur de la forme *cond*[26]. Par exemple, la première clause étant ((< montant 0) -10), si le montant est inférieur à 0, la valeur de la forme *cond* sera -10 (le taux d'intérêt est de -10%). Si le montant est plus petit que 1000 (et plus grand ou égal à 0), le taux d'intérêt est de 2%, et ainsi de suite, pour en arriver à l'alternative finale dont la forme-test est *t* : si le montant est plus grand ou égal à 10000 le taux est de 5%. Notez l'importance de l'ordre des clauses dans cet exemple.

10.3. Expressions logiques

Les opérateurs logiques permettant de construire des expressions logiques sont *not* (non), *and* (et) et *or* (ou).

and et *or* sont des opérateurs de type *macro* qui acceptent 0, 1 ou plusieurs arguments. *and* et *or* n'étant pas des fonctions, ils ont chacun leur propre règle d'évaluation :

- *and* retourne nil dès qu'un seul de ses arguments est faux (les arguments suivants ne sont pas évalués). Sinon (si tous ses arguments sont vrai), il retourne la valeur du dernier argument.
- *or* retourne nil si tous ses arguments sont faux. Sinon il retourne la valeur du 1er argument qui est différent de nil (les arguments suivants ne sont pas évalués).

not est une fonction qui reçoit un et une seul argument et qui réalise la négation logique de son argument. Si l'argument est vrai, *not* retourne nil. Si l'argument est faux, *not* retourne t.

```
> (and 3 4 5)
5

> (and 3 4 nil 5)
nil

> (and 3 (evenp 5))
nil

> (and 3 (evenp 5) 8)
nil

> (or nil 3 4 nil 5))
3
```

[26] Syntaxe d'une clause :

```
(forme-test forme-1 forme-2 ... forme-n)
```

C'est-à-dire une liste d'une forme-test suivie de 0, 1 ou plusieurs formes. Pour la valeur retournée, seule compte la dernière forme, forme-n (ou la forme-test, si celle-ci n'est suivie d'aucune forme).

```
> (or nil t 4 nil 5))
t

> (or nil (evenp 4))
t

> (or nil (evenp 4) 8)
t

> (or nil nil)
nil

> (not t)
nil

> (not nil)
t

> (not "C++")
nil

> (not (and 3 4 5))
nil
```

On pourrait définir le prédicat *evenp* (primitive du Lisp qui teste si son argument est un nombre pair) comme ceci :

```
(defun pair? (n)
  (if (integerp (/ n 2))
      t
    nil))
```

Ou plus simplement :

```
(defun pair? (n)
   (integerp (/ n 2)))
```

En utilisant *pair?*, la fonction suivante teste si son argument est un nombre qui soit impair et plus grand que 100 :

```
(defun impair>100? (n)
  (and (numberp n)
       (not (pair? n))
       (> n 100)))
```

Notez l'utilisation de *numberp* qui teste si l'argument est un nombre avant qu'il ne soit éventuellement passé à la fonction *pair?* (car / n'accepte qu'un nombre en argument).

10.4. Exercices

;;; Fonction d'un argument
;;; - qui retourne t si l'argument est différent de nil (vrai)

```
;;; - qui retourne nil sinon (si l'argument est nil)
(defun vrai? (x)
  )

;;; Définir un prédicat qui fait le contraire du prédicat vrai?
(defun faux? (x)
  )

;;; Définir une fonction non, équivalente au not l'opérateur de
;;; négation logique
(defun non (x)
  )

;;; Ecrire une fonction qui prend 2 arguments de type nombre et qui
;;; retourne le plus grand.
(defun >-de-deux (n1 n2)
  )

;;; Ecrire une fonction qui prend 3 arguments de type nombre et qui
;;; retourne le plus grand.
(defun >-de-trois (n1 n2 n3)
  )
```

11. Opérateurs d'itération

Le plus simple des opérateurs d'itération Lisp est celui qui permet de parcourir tous les éléments d'une liste, la macro *dolist* dont la syntaxe est :

```
(dolist (var forme &optional forme-résultat)
   forme-1
   ...
   forme-n)
```

Tout d'abord, l'argument *forme* est évalué et doit retourner une liste. La variable *var* est liée au 1er élément de la liste; ensuite chacune des formes (forme-1, ... , forme-n) est évaluée tour à tour. Ensuite, la variable *var* est liée au 2ème élément de la liste; chacune des formes (forme-1, ... , forme-n) est de nouveau évaluée tour à tour. Et ainsi de suite jusqu'à épuisement de la liste. A ce moment, l'argument optionnel *forme-résultat* est évalué (s'il est présent) et retourné comme valeur de la forme *(dolist (...) ...)* . Si *forme-résultat* n'est pas présent, c'est nil qui est retourné, comme dans cet exemple :

```
> (defun imprime-chaque-item (liste)
    (dolist (i liste)
      (print i)))
imprime-chaque-item
```

```
> (imprime-chaque-item '(2 3 4))
2
3
4
nil
```

Avec *dolist*, on peut définir l'équivalent de *length*, la fonction qui retourne la longueur d'une liste (c'est-à-dire son nombre d'éléments) :

```
> (defun longueur (liste)
    (let ((long 0))
      (dolist (i liste long)
        (setf long (+ long 1)))))
 longueur

> (longueur '(2 3 4))
3
```

Pour interrompre le déroulement d'une boucle *dolist*, on utilise la macro *return* qui prend un argument optionnel (à évaluer), qui sera la valeur retournée par la forme *(dolist (...) ...)* . Soit à définir, en utilisant *dolist*, l'équivalent de *third* (c'est-à-dire une fonction qui retourne le 3ème élément d'une liste ou nil si la longueur de la liste est < que 3) :

```
(defun 3ème (liste)
  (let ((i 1))
    (dolist (e liste)
      (when (= 3 i)
        (return e))
      (setf i (+ 1 i)))))
```

Il existe d'autres opérateurs d'itération prédéfinis, certains très généraux comme la macro *loop*, dont voici une des possibilités; c'est une itération pour *n* variant de 1 à 10 inclus, avec collecte dans une liste des valeurs de *n* qui sont paires (par défaut le pas est de 1, c'est-à-dire que *n* est incrémenté de 1 à chaque cycle) :

```
> (loop for n from 1 to 10
        when (evenp n)
        collect n)
(2 4 6 8 10)
```

Pour un programmeur, il est facile de définir un opérateur d'itération qui lui manquerait. Voici une définition de l'opérateur *tant-que* (*while* en anglais, un classique dans certains langages). Sans entrer dans les détails[27] :

[27] *Tant-que* est un opérateur de type macro, défini à l'aide de l'opérateur *defmacro*. L'utilisation de *defmacro* est un sujet qui sort du cadre d'une introduction au Lisp. C'est un outil très puissant, sans équivalent dans la plupart des autres langages de programmation.

```
> (defmacro tant-que (test &body body)
    `(do ()
         ((not ,test))
       ,@body))
tant-que
```

Ce qui correspond à la syntaxe :

```
(tant-que test
    forme-1
    forme-2
    ...
    forme-n)
```

L'itération a lieu tant que l'évaluation de la forme *test* donne vrai. Par exemple, on peut utiliser *tant-que* pour définir la fonction suivante qui parcourt une liste de nombres un par un et les imprime, tant qu'ils sont plus petits qu'une valeur limite[28] :

```
> (defun imprimer-tant-que-< (liste limite)
    (let* ((position 0)
           (n (nth position liste)))
      (tant-que (and n (< n limite))
         (print n)
         (setf position (+ 1 position))
         (setf n (nth position liste)))))
imprimer-tant-que-<

> (imprimer-tant-que-< (list 1 2 3 4 5 6 7 8 9) 4)
1
2
3
nil
```

En pratique, cette fonction s'écrira simplement et efficacement :

```
(defun imprimer-tant-que-< (liste limite)
  (dolist (n liste)
    (if (< n limite)
        (print n)
      (return))))
```

La conclusion est la même dans le cas d'un vecteur de nombres au lieu d'une liste de nombres, utiliser l'opérateur *tant-que* rend la définition de la fonction inutilement complexe :

[28] Accéder les éléments *successifs* d'une liste de cette façon (*nth*) n'est pas efficace.

```
(defun imprimer-tant-que-< (vecteur limite)
  (let ((position 0)
        (longueur (length vecteur))
        (plus-petit? t))
    (tant-que (and plus-petit?
                   (< position longueur))
       (let ((n (svref vecteur position)))
         (if (< n limite)
             (print n)
           (setf plus-petit? nil))
         (setf position (+ 1 position))))))
```

svref est une fonction d'accès à un vecteur (1er argument, le vecteur; 2ème argument l'indice à compter à partir de 0).

```
> (defun imprimer-tant-que-< (vecteur limite)
    (dotimes (i (length vecteur))
      (let ((n (svref vecteur i)))
        (if (< n limite)
            (print n)
          (return)))))
imprimer-tant-que-<
```

dotimes est similaire à *dolist*; la variable *i* est liée tour à tour de 0 au nombre d'éléments dans le vecteur moins 1.

```
> (imprimer-tant-que-< (vector 1 2 3 4 5 6 7 8 9) 4)
1
2
3
nil
```

vector est une des fonctions de création de vecteur :

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

11.1. Exercices

;;; Fonction qui reçoit en argument une liste de nombres et en retourne
;;; la somme (en utilisant dolist)
(defun somme (liste)
)

;;; Fonction d'un argument qui est une liste de nombres et qui en
;;; retourne la moyenne. Utiliser dolist ou la fonction somme ci-dessus.
(defun moyenne (liste)
)

12. Fonctions d'entrée-sortie

12.1. Lecture

La fonction *read* est une des fonctions qui permet de lire des données (saisir ou entrer des données), par exemple à partir de la fenêtre de l'interacteur ou d'un fichier. Elle lit une expression lisp et la retourne. Quand elle utilisée dans la fenêtre de l'interacteur, la fonction *read* attend indéfiniment qu'une expression lisp *complète* soit tapée pour ensuite la retourner :

```
> (read)
pomme
pomme
```

Dans l'interaction ci-dessus, le symbole **pomme** (en gras) est tapé par l'utilisateur (suivi d'un retour à la ligne pour valider - touche *Entrer*). Ensuite la fonction *read* lit ce symbole *pomme* et le retourne. Idem avec une liste :

```
> (read)
(pomme poire)
(pomme poire)
```

Si l'utilisateur tape un retour à la ligne (touche *Entrer*) à la fin du symbole *poire*, *read* continue d'attendre l'introduction d'une parenthèse marquant la fin de l'expression :

```
> (read)
(pomme poire
)
(pomme poire)
```

12.2. Ecriture

Il y a plusieurs fonctions d'écritures, dont la fonction *print* déjà vue. La fonction *format* est la fonction d'écriture générale qui permet de formater du texte :

```
(format destination contrôle param-1 param-2 ...)
```

Dans un appel, *format* est suivi de 2 arguments obligatoires (*destination* et *contrôle*) et de 0, 1 ou plusieurs arguments optionnels (*param-1* , *param-2* , ...). Le 1er argument obligatoire est ***destination*** qui spécifie où écrire (fenêtre, fichier, ...). Le 2ème est une chaîne de caractères de contrôle pour spécifier quoi écrire et comment écrire les arguments optionnels éventuels (***chaîne de contrôle*** en bref).

Voici quelques exemples dans le cas où *destination* est t, ce qui veut dire écrire dans la fenêtre de l'interacteur :

```
> (format t " Introduire un nombre: ")
 Introduire un nombre:
nil
```

Voici quelques unes des directives à insérer dans la chaîne de contrôle. Chaque fois que la ***directive ~%*** apparaît dans la chaîne de contrôle, elle impose un saut à la ligne :

```
> (format t "~% Bonjour,~%  introduisez un nombre : ")

 Bonjour,
  introduisez un nombre :
nil
```

La ***directive ~&*** impose un saut à la ligne sauf si le curseur est déjà en début de ligne :

```
> (format t "~& Bonjour,~&  introduisez un nombre : ")
 Bonjour,
  introduisez un nombre :
nil
```

La ***directive ~a*** indique une position à remplir par un des arguments optionnels qui sont utilisés dans leur ordre d'apparition :

```
> (format t " ~A plus ~A égale ~A" 2 3 5)
 2 plus 3 égale 5
nil
```

Si l'argument *destination* est **nil**, l'écriture se fait dans une chaîne de caractères (*string*) qui est retournée comme valeur de l'appel à *format* :

```
> (format nil " ~A plus ~A égale ~A" 2 3 5)
" 2 plus 3 égale 5"
```

La connaissance de *format* et *read* nous permet de définir une fonction de conversion de FB en euros qui demande à l'utilisateur le montant à convertir :

```
> (defun FB-en-euros-u ()
    (format t " Entrez un montant en FB à convertir en euros : ")
    (let ((montant (read)))
      (format t "  ~A FB font ~A euros" montant
                                       (FB-en-euros montant))))
FB-en-euros-u

> (FB-en-euros-u)
 Entrez un montant en FB à convertir en euros : 100
  100 FB font 2.48 euros
nil
```

La version suivante de FB-en-euros-u est plus fiable car elle vérifie que le montant saisi est bien un nombre (si ce n'est pas un nombre, la fonction se rappelle récursivement) :

```
(defun FB-en-euros-u ()
  (format t " SVP, entrez un montant en FB : ")
  (let ((montant (read)))
    (if (numberp montant)
        (format t " ~A FB font ~A euros" montant
                                        (FB-en-euros montant))
      (FB-en-euros-u))))
```

13. Récursion

Une fonction récursive est une fonction qui s'appelle elle-même. Voici une version récursive de la fonction *longueur* définie précédemment :

```
(defun longueur (liste)
  (if (null liste)
      0
    (+ 1
       (longueur (rest liste)))))
```

Cette définition exprime que la longueur d'une liste est :

1. soit 0 si la liste est vide
2. soit 1 plus la longueur du restant de la liste si la liste n'est pas vide.

Une définition de fonction récursive comprend toujours ces deux étapes : un cas de base[29] qui correspond à une situation simple (limite) et un cas général où une valeur de la fonction est calculée selon une valeur de cette fonction déjà calculée[30].

Souvent, la solution à certains types de problème ne peut s'exprimer simplement que sous forme récursive, en général suite à la nature récursive des données du problème. De fait, les nombres entiers et les listes peuvent être définis de façon récursive, d'où la simplicité des solutions récursives aux problèmes mettant en œuvre des listes et des nombres entiers. Ainsi la définition de la factorielle :

```
> (defun factorielle (n)
    (if (= 0 n)
        1
      (* n
         (factorielle (- n 1)))))
factorielle

> (factorielle 4)
24
```

Une critique non fondée des fonctions récursives est qu'elles sont inefficaces : de fait, un bon compilateur (Lisp ou autre) peut convertir certaines fonctions récursives en fonctions itératives, sans perte d'efficacité.

---

[29] ou cas de terminaison

[30] autrement dit : le cas en *n* est calculé en fonction de la solution du cas en *n-1*

13.1. Exercices

Écrire une fonction retournant la somme des *n* premiers nombres entiers positifs, chacun pris au carré. Par exemple :

```
> (somme-carrés 3)
14
```

14. Paramétrage des fonctions

Le paramétrage des fonctions Lisp est très souple : en plus des arguments requis, il est possible de définir des arguments optionnels et par mots-clés. Par exemple, on peut définir une fonction générale de conversion de l'euro qui par défaut convertirait en FB :

```
(defun en-euros (x &optional (pays :be))
  (* x
     (case pays
       (:be 40.3399)
       (:fr 6.55957)
       (:de 1.95583)
       (:maroc 10.34)
       (otherwise (error "Ne connaît pas le pays ~S" pays)))))
```

L'évaluation de (en-euros 1) ou (en-euros 1 :be) donne le même résultat 40.3399
Pour la conversion en FF :

```
> (en-euros 1 :fr)
6.55957
```

La fonction *en-euros* a deux arguments : le 1er est le montant en euros à convertir, un argument obligatoire; le 2ème est optionnel, c'est le pays de la monnaie dans laquelle il faut convertir, et qui, par défaut est :be (Belgique).

:be , :fr , :maroc sont des symboles particuliers, de type *keyword* (mot clef), un sous-type de *symbol*. Leur particularité première est d'être auto-évaluant :

```
> :maroc
:maroc
```

case est un opérateur conditionnel de type *macro*. Il permet de construire une expression conditionnelle dont la valeur dépend d'une valeur clef (*pays* dans l'exemple).

error est une fonction CL qui permet de déclencher une erreur.

On aurait pu définir la fonction *en-euros* en utilisant un paramètre par mot clef :

```
(defun en-euros (x &key (pays :be))
  (* x
     (case pays
       (:be 40.3399)
       (:fr 6.55957)
       (:de 1.95583)
       (:maroc 10.34)
       (otherwise (error "Ne connaît pas le pays ~S" pays)))))
```

Les appels à cette 2ème version de *en-euros* s'écrivent :

```
(en-euros 1)
(en-euros 1 :pays :be)
(en-euros 1 :pays :maroc)
```

L'avantage des paramètres par mot clef, c'est qu'ils aident à la lisibilité du code (à condition de bien choisir le nom des paramètres - imaginez une fonction ayant une quinzaine de paramètres...). D'autre part, lors de l'application, l'ordre des arguments n'a aucune importance. Soit une fonction *fun* ayant 20 paramètres par mot clef, *arg1*, *arg2*, ..., *arg20* (et aucun paramètre requis). Un appel pourrait s'écrire :

```
(fun :arg15 22
     :arg11 33)
```

A supposer la même fonction, mais avec 20 paramètres optionnels (&*optional*) au lieu des 20 paramètres par mot clef; pour écrire un appel équivalent à celui ci-dessus, il faudrait écrire les 15 premiers arguments, dans l'ordre :

```
(fun ... 33 ... 22)
```