Conservatoire National des Arts & Métiers
Centre associé de Nice

Le Raisonnement Informatique & La Programmation

Jean Demartini

1994 - 1995 - 1996 - 1997 - 1998 - 1999 - 2000

Informatique - Cycle A

Algorithmique & Programmation

Table des Matières

1. Introduction

Aux temps héroïques, dans les années 50 et 60, l' *informaticien* était celui qui savait utiliser un ordinateur, essentiellement pour faire des calculs. Son but était presque toujours de trouver une méthode de calcul — algorithme mettant en œuvre les organes d'un ordinateur : circuits de calcul, mémoire et organes d'entrée/sortie — décrite sous la forme d'un programme. L'informaticien était un programmeur, ce qui l'amusait beaucoup et l'informatique était une forme d'artisanat.

Aussi longtemps qu'il n'a pas été possible de récupérer commodément les programmes écrits par d'autres, l'informaticien a été programmeur — il passait son temps à réécrire des programmes — et il a fallut qu'il apprenne à écrire, seul ou avec d'autres des programmes de plus en plus complexes et de plus en plus gros.

Pendant longtemps, on a pu croire (ou voulu faire croire) que l'informatique était une science en soi, celle qui consiste à programmer de façon plus ou moins rationnelle des ordinateurs ordinateurs [1] . Bien entendu, cette vision est aussi naïve que celle qui consiste à croire que la mécanique est uniquement ce que font les garagistes, l'électricité uniquement ce que font les électriciens ou la chimie uniquement ce que font les chimistes.

On sait bien que les choses ne sont pas si simples et que chacun de ces domaines techniques nécessite la mise en œuvre de tout un ensemble de connaissances théoriques et pratiques bien différentes. Il en va de même pour l'informatique qui ne peut que prendre le même chemin.

Prenons un exemple très simplifié pour illustrer ce propos, celui de la construction d'un édifice (une maison, une église, une école, un pont...).

[1] Dans son édition 1987, le «Petit Larousse» définit l'informatique par : «Science du traitement automatique et rationnel de l'information considérée comme le support des connaissances et des communications». Nous ne lancerons pas le débat pour essayer de savoir si l'informatique qui nous intéresse est une science ou une technique.

Ne nous posons pas de questions sur la justification (une église ou un pont de plus ?) et la finalité (une école ou une piscine ?) de l'édifice que nous devons construire.

Cet édifice doit rendre un certain service tout en satisfaisant des contraintes (techniques, sociales, économiques...) et sa structure, sa forme seront le résultat d'un compromis entre de nombreux *choix conceptuels* possibles. C'est à l'*architecte* que revient la définition de ce compromis.

La construction de cet édifice va mettre en œuvre différents matériaux et composants (bois, métal, béton, plâtre, vitrages, canalisations...) et différentes formes (poutres, lames, feuilles, voiles, fils, tuyaux...) qui devront être assemblées de telles sorte que les lois physiques qui les régissent soient toutes respectées. L'édifice doit tenir debout et fonctionner (les lampes s'allument quand on actionne les interrupteurs et l'eau coule quand on ouvre les robinets...) même s'il y a du vent ou de la neige. Un *bureau d'étude* et des *ingénieurs* définiront les différents *choix techniques*.

Jusqu'à présent, notre édifice n'existe que dans la tête de ceux qui en parlent, il reste encore à le construire. Cette construction va nécessiter la coordination de nombreux corps de métiers (maçons, charpentiers, carreleurs, plâtriers, électriciens...). C'est une *entreprise de construction* qui assurera cette coordination. Son rôle consiste à planifier les travaux à effectuer et vérifier qu'une tâche est réalisée correctement et au moment où elle doit l'être. Son rôle consiste également à faire en sorte que tout ce qui peut se faire en même temps le soit effectivement.

Finalement, les seuls qui font quelque chose de *concret* sont les corps de métiers. Tous les autres manipulent des *abstractions*. C'est tout cela qui constitue le «Bâtiment».

Parler d'Informatique, aujourd'hui, cache autant de diversité que parler du «Bâtiment», c'est une activité multiformes qui peut être par exemple :

- écrire un programme scolaire en Basic,
- construire et/ou assembler les éléments qui constituent un ordinateur,
- participer à l'écriture d'un gros programme,
- concevoir un système d'information (une base de données),
- concevoir un réseau d'entreprise,
- faire sortir le prince de «Prince of Persia» du piège dans lequel il est tombé,
- choisir des outils bureautiques,
- administrer un réseau et de ses d'utilisateurs,
- dépanner un disque dur,
- démontrer l'inconsistance du λ-calcul non typé,
- changer le fusible de l'imprimante,
- mettre au point un algorithme de cryptage,
- concevoir un langage de programmation,
- ...

et bien d'autres choses.

> Par contre nous nous refusons absolument à appeler «Informatique» le jeu de hasard, immortalisé par le cinéma et les séries télévisées, qui consiste à taper des signes sur un clavier pour essayer de deviner le mot de passe donnant accès aux fichiers d'un personnage malfaisant.

Une classification académique (chez les anglo-américains) des activités de l'informatique peut être celle du Tableau1, page3. Mais cette classification ne concerne que l'activité qui consiste à construire des ordinateurs ou celle qui consiste à écrire des programmes, elle est donc très incomplète.

Tableau 1 : Une classification des activités informatiques.

Génie informatique	Sciences des ordinateurs *Computer science*	Etude des lois physiques (semi-conducteurs, phénomènes de propagation, etc.) et des principes (architecture des machines, méthodes de communication, etc.) utiles pour la conception des ordinateurs
	Ingénierie des ordinateurs *Computer engineering*	Etudes des méthodes utiles pour la fabrication des machines (circuits intégrés, circuits imprimés, connecteurs, gestion de production etc.)
Génie logiciel	Sciences du logiciel *Software science*	Etudes des lois de raisonnement et d'abstraction (logiques diverses, protocoles de communication, mécanisme de synchronisation, etc.) et des structures utiles à la conception des logiciels
	Ingénierie du logiciel *Software engineering*	Etudes des méthodes utiles pour la réalisation des logiciels.

Mais alors, que faut-il enseigner à un informaticien débutant pour lui faciliter l'accès à une profession appelée vaguement **Informaticien ?**

Pendant très longtemps (et encore malheureusement souvent), l'enseignement de l'informatique consistait presqu'uniquement à enseigner un langage de programmation. Si cette approche était tout à fait pardonnable il y a 30 ans, elle relève de l'escroquerie intellectuelle aujourd'hui.

Il est clair que nous n'aborderons pas tous les thèmes suggérés par la liste précédente, nous éliminerons tous ce qui peut s'apprendre «sur le tas» pour peu qu'on soit un peu curieux :

- utiliser naïvement un traitement de texte ou un tableur,
- changer le fusible de l'imprimante,
- ...

ainsi que toute chose qui s'apprendra aisément en lisant une notice, en pratiquant suffisamment ou qui relève en fait d'un autre métier (l'utilisation éclairée des traitements de texte actuels nécessite incontestablement des connaissances en typographie et en composition de document).

Nous éliminerons également tout ce qui (ne) peut (que) s'apprendre par l'expérience :

- gérer une équipe de programmeurs,
- planifier et conduire l'écriture d'un gros programme,
- ...

En fait, si nous ne voulons pas simplement énoncer un catalogue de recettes, nous ne pouvons considérerons que des domaines où il a été **possible** et **utile** d'établir des lois générales. En fait, l'informatique, par son aspect conception, utilise à grande échelle des mécanismes d'abstraction très généraux et ce sont ces mécanismes que nous introduirons et développerons.

La réalisation d'un système peut être schématiquement conduite en 3 étapes :

1. *spécifier* le problème à résoudre,
2. en *concevoir* une solution,

3. *implémenter*[2] cette solution.

L'étape d'implémentation est très analogue à ce que dans un processus de production on appelle la *fabrication* et de même que la fabrication des objets matériels est de plus en plus confiée à des machines, il en ira probablement de même pour l'implémentation des programmes aussi allons-nous concentrer nos efforts sur les mécanismes de spécifications et de conception.

Nous dirons alors que **programmer c'est spécifier et concevoir** (et éventuellement implémenter) et nous qualifierons de programme le résultat d'un travail de spécification ou d'un travail de conception.

Ce cours est **destiné à des débutants**. Il tente de donner une image la plus rationnelle possible de la programmation. Nos connaissances actuelles ne permettent pas de présenter les mécanismes d'abstraction utilisés par l'informatique aussi rigoureusement que peut l'être une théorie mathématique ou un domaine bien mature de la physique, nous serons donc, de temps en temps, obligés d'avouer notre impuissance. Cependant, cette image préfigure probablement les concepts qui sous-tendront les outils de travail qui seront mis à la disposition des programmeurs dans les années futures. Nous sommes bien conscients que tout n'est pas facile dans ce cours aussi avons-nous fait notre possible pour introduire les difficultés de manière progressive et avons-nous éliminé tout ce qui n'était pas strictement nécessaire à la compréhension de l'ensemble. Tous les mécanismes d'abstractions que nous allons rencontrer se rencontrent (quelque fois inconsciemment) dans les activités de la vie courante et n'ont rien, en fait, de «nouveau». Ce qui est nouveau, c'est la nécessité que nous aurons de les analyser attentivement afin d'en découvrir toutes les propriétés.

Si les quelques idées sous-jacente un peu originales que nous introduirons peuvent être attribuées aux logiciens des années 40 — nous considérerons un peu arbitrairement que leur chef de file est Alonzo Church sans toutefois minimiser l'influence des Schönfinkel, Curry, Strachey, Scott, McCarthy, Milner... — ce n'est que dans les années 60 que la pertinence de ces idées dans le domaine de la programmation a été mise en évidence. Les articles de P.J.Landin *The Next 700 programming Languages* [3] et *The mechanical Evaluation of Expressions*[4] marquent un tournant dans la formalisation de l'activité de programmation.

La présentation qui est donnée ici est assez inhabituelle. Elle ne déroute pas des débutants dotés de la culture scientifique d'une classe de terminale de lycée, par contre, ceux qui ont appris l'informatique d'une façon plus traditionnelle ou par quelques années d'exercice de la profession sont souvent perturbés par une approche radicalement différente. Je leur demande d'accepter de faire l'effort d'oublier, provisoirement bien sûr, ce qu'il savent déjà et de «jouer le jeu». Cet effort sera récompensé par l'acquisition d'une compréhension profonde des mécanismes de programmation au sein de laquelle leurs connaissances s'intégreront harmonieusement.

Tout au long de ce cours, vont alterner des résultats établis rigoureusement et des affirmations qui reflètent une opinion de l'auteur. Dans la mesure où aucune faute de raisonnement n'est commise, un résultat «rigoureux» est incontestable, par contre, toute opinion peut être discutée. C'est pourquoi nous avons eu le soucis d'énoncer le plus clairement possible les arguments utilisés pour étayer telle ou telle opinion [5].

---

[2] mot franglais signifiant approximativement «mettre en œuvre, réaliser, construire».

[3] Communication of ACM 9, march 1966, pp 157-166

[4] The Computer Journal 6(4), 1964, pp 157-166

[5] Est-il nécessaire de rappeler qu'une opinion ne vaut que par les arguments sur laquelle elle s'appuie et non pas par la qualité de son auteur ?

Les différents chapitres sont suivis d'exercices de difficultés variées et choisis plus pour l'intérêt de la réflexion que leur résolution nécessite que pour l'intérêt pratique de la solution à laquelle ils conduisent. La résolution de certains n'exige que quelques minutes à un lecteur attentif, d'autres peuvent réclamer plusieurs jours d'un travail difficile. Rien ne distingue un exercice facile d'un exercice (très) difficile afin d'éviter une esquive inconsciente. En fait, ces exercices sont conçus pour que la recherche de la solution apporte plus que la solution elle-même.

La plupart de ces exercices n'ont pas une solution unique et pour éviter que la solution de l'auteur soit considérée comme la meilleure simplement parce que c'est celle de l'auteur, aucune solution n'est donnée. Je ne voudrais pas, non plus, priver le lecteur de l'immense satisfaction de trouver par soi-même. Que celui-ci se rassure, l'expérience lui démontrera que lorsqu'il aura trouvé une solution satisfaisante, il sera convaincu de sa validité.

Le chapitre 2 est un rappel simple de la notion de *fonction* telle que les mathématiciens la définissent aujourd'hui. On mettra en évidence la différence fondamentale entre la vision en extension des mathématiques et la vision en intention de l'informatique. Nous y introduirons une notation bien adaptée à la description de l'intention d'une fonction, la λ-notation.

«Les mathématiciens n'étudient pas des objets, mais des relations entre les objets ; il leur est donc indifférent de remplacer ces objets par d'autres, pourvu que les relations ne changent pas. La matière ne leur importe pas, la forme seule les intéresse.» [6]

A la recherche d'un concept d'abstraction unificateur, deux choix sont possibles, on peut considérer que tout est objet (même les relations entre objets), c'est l'approche en extension des mathématiques actuelles ou que tout est relation (même les objets), c'est l'approche en intention de l'informatique théorique actuelle.

Nous verrons donc les différents mécanismes permettant de définir une fonction en intention. On introduira, en particulier, les deux modes de raisonnement que nous serons amené à utiliser le plus fréquemment, l'induction et la recherche d'un invariant.Ainsi ce chapitre introduit le mécanisme d'abstraction fondamental, l'abstraction par les *fonctions*.

Le chapitre 3 introduit un deuxième mécanisme abstraction, celui qui permet de définir des *choses en soi* qu'on appellera des *données*. On verra, en particulier, que tout ce qui est «données» est aisément construit en utilisant le mécanisme d'abstraction précédent. Ce chapitre est très déroutant pour l'informaticien traditionnel car il prend le contre-pied de l'opinion commune selon laquelle, puisqu'on spécifie nécessairement un problème en définissant les données d'une part et les traitement s de ces données d'autre part, données et traitements sont des notions distinctes et primitives. En fait, ce chapitre n'est pas plus (mais pas moins) déroutant que la découverte des nombres irrationnels au mathématicien débutant.

Le chapitre 4 introduit un troisième mécanisme d'abstraction, celui qui permet de définir *des choses pour organiser des choses* qu'on appellera des *structures*. Ce mécanisme est analogue à l'invention des contenant (les récipients pour ranger des choses) après celle des contenus (les choses à ranger). les abstractions ainsi construites ont des propriétés redoutables qui vont les rendre difficiles à manipuler. En particulier, on touchera du doigt l'impossibilité de définir l'*égalité* de deux choses sur un plan général.

Le chapitre 5 introduit le langage (de programmation ?) **Scheme** permettant de décrire des fonctions, des données ou des structures d'une manière plus formelle. Le fait que ce langage soit défini rigoureusement permet d'en spécifier un interprète. En attendant, nous nous contenterons de considérer que cet interprète existe. Il est clair que, pour nous, **Scheme** n'est

[6] Henri Poincaré dans *La science et l'hypothèse* page 49 - Flammarion - Champs

qu'une forme commode de la λ-notation.

Le chapitre 6 introduit une catégorie de problèmes qui prend en défaut tous ce que nous venons de voir et nous oblige à introduire un nouveau mécanisme, l' *affectation*. Si ce mécanisme permet de résoudre des problèmes utiles, il introduit des difficultés de raisonnement qui nous amènerons à en contrôler strictement l'usage. Les structures de données précédemment introduites ont alors un comportement différent et deviennent des *structure mutables*.

Le chapitre 7, dans le cadre d'un petit problème de gestion presque réaliste, illustre les problèmes qu'on rencontre lorsqu'on transforme l'expression des besoins d'un client en des spécifications générales puis détaillées rigoureuses. En particulier, on verra apparaître la nécessité d'utiliser des stratégies de conception permettant de rendre *robuste* le programme ainsi réalisé.

Le chapitre 8 est très différent. Tous les chapitres précédents s'occupaient essentiellement de conception, ce chapitre va surtout parler d'implémentation et montrer comment on peut passer des spécifications que les chapitres précédents nous ont appris à construire à un programme qui s'exécute. Nous avons pris comme exemple un langage très répandu, le *langage C*. Le langage C n'a pas très bonne presse, certains le considère comme «confus», d'autres lui préfèrent C^{++} «plus moderne» ou ADA «mieux conçu». Nous voulons montrer, dans ce chapitre, qu'un programme confus est à mettre au passif du programmeur et de ses méthodes de raisonnement et non pas du langage qu'il utilise.

Il est peu probable que vous puissiez lire ce qui va suivre d'une seule traite et même dans l'ordre où les choses sont présentées. Certaines notions un peu complexes gagnent à être sautées en première lecture.

> Vous rencontrerez également des paragraphes ayant la forme de celui que vous êtes en train de lire. Ces paragraphes introduisent toujours des notions, des remarques, des extensions qui peuvent être sautées en première lecture.

Les lecteurs de cet ouvrage étant a priori de cultures différentes, nous avons introduit des exemples les plus variés possibles. Ne vous inquiétez donc pas si certains d'entre eux vous paraissent obscurs, sautez les, un autre viendra, emprunté (je l'espère) à votre culture.

2. Les Fonctions

> ...Cependant, dans notre perspective actuelle, pour la précision qu'il nous faut pour commencer, nous n'avons pas besoin d'être très attentifs à définir les choses précisément. Peut-être allez-vous dire «c'est quelque chose de terrible, j'avais appris qu'en science, nous devions tout définir précisément». Nous ne pouvons pas définir n'importe quoi précisément ! Si nous essayons, nous devenons victime de cette paralysie de penser qui affecte les philosophes[1], assis l'un en face de l'autre, l'un disant à l'autre «Vous ne connaissez pas ce dont vous parler !». Le second répond «Qu'entendez-vous par connaissez ? que voulez-vous dire par parler ? Que voulez-vous dire par vous ?» etc. Afin d'être capables de parler constructivement, nous devons simplement nous mettre d'accord sur le fait que nous parlons en gros de la même chose...
>
> R.P.Feynman
> Le Cours de Physique de Feynman

La fonction va représenter notre premier et principal outil de raisonnement, il est donc important d'en avoir une vision claire et de savoir comment le manipuler de façon efficace et cohérente, d'en appréhender la puissance et les limites.

Au sens usuel du terme, on dit qu'une chose[2] est *fonction* d'une autre quand celle-ci dépend de celle-là. Cette définition permet de donner un sens qualitatif ou quantitatif à la notion usuelle de relation de cause à effet mais aussi de décrire une relation entre plusieures choses. A l'issu d'un enseignement de mathématiques, les fonctions sont toujours associées à un calcul et les plus simples d'entre elles sont définies par des *formules*. Mais la fonction définie en tant que formule s'est vite révélée insuffisante car le mot «formule» est lui-même très difficile à définir (pensez, par exemple, aux fonctions trigonométriques, exponentiel-

[1] Feynman parle ici des mauvais philosophes, bien sûr !

[2] Le mot «chose» dénote ici tout ce que nous pouvons imaginer, c'est à dire toutes les *notions abstraites* que nous utilisons pour décrire le monde qui nous entoure.

les...). D'autre part, certaines opérations élémentaires ne sont définies que par une table (souvenons-nous des tables d'addition et de multiplication de notre enfance).

Faute de pouvoir donner un sens précis et général aux formules, les mathématiciens ont été amenés à considérer les fonctions comme des *objets en soi*. Les informaticiens utilisent une formule imagée et disent que les fonctions sont d' *ordre supérieur* ou sont des *citoyens de première classe*.

La construction de la théorie des fonctions a conduit à l'introduction des *ensembles* et le langage des ensembles permet de donner une définition rigoureuse des fonctions dans le cadre du concept plus général d'*application*.

Au début de ce chapitre nous allons rappeler les quelques définitions qui vont nous permettre de manipuler les fonctions (ou applications) d'une manière suffisamment rigoureuse pour nos besoins. En particulier, nous utiliserons le mot fonction au sens usuel (donc à la place du mot application) sans chercher à lui attribuer des nuances supplémentaires subtiles. C'est ce léger abus de langage[3] qui nous permettra de dire, le moment venu, qu'on *applique une fonction*.

2.1 Rappel sur les fonctions

La fonction mathématique permet de décrire la mise en correspondance entre une donnée de départ d'une certaine sorte et une donnée d'arrivée éventuellement d'une autre sorte.

Lorsqu'on sait associer un mécanisme de transformation (de calcul en général) à la définition d'une fonction, on dit qu'elle est définie *en compréhension* ou *en intention*. Par contre lorsque la fonction représente une simple correspondance, on dit qu'elle est définie *en extension* par l'ensemble de toutes les correspondances possibles.

2.2 Les Différentes formes de la définition d'une fonction

L'ensemble des données de départ s'appelle le *domaine* de la fonction et l'ensemble des données d'arrivée s'appelle son *codomaine*. Ainsi la fonction notée «f» fait correspondre à chaque élément, noté x, de son domaine un seul élément, noté $f(x)$, de son codomaine [4].

La définition d'une fonction comporte toujours deux parties :

1. l'indication de son domaine et de son codomaine — dans la plupart des exemples que nous prendrons au début, le domaine et le codomaine de la fonction qu'on définira seront des ensembles de nombres.
2. l'indication de la méthode qui permet de passer de chaque élément du domaine à ceux du codomaine.

Ainsi la définition d'une fonction commence toujours par une phrase de la forme

«Le domaine de la fonction «foo» est l'ensemble des *snarks* tandis que son codomaine est

[3] L'enseignement secondaire français introduit une subtile différence entre la fonction et l'application. Cette nuance n'est pratiquement reprise par personne et nous ne la retiendrons pas.

[4] Il convient donc de bien distinguer la fonction «f» de la valeur f(x) qu'elle peut prendre dans son codomaine.

l'ensemble des *jabberwocks*...»

Ce que les mathématiciens notent

$$\begin{aligned} &\text{foo} : S J \rightarrow \\ &\text{avec} : S = \{\ldots\} \text{snark} \\ &\qquad J = \{\ldots\} \text{jabberwock} \end{aligned}$$

et, quelque fois, dessinent un diagramme sagittal (figure 1, page 10).

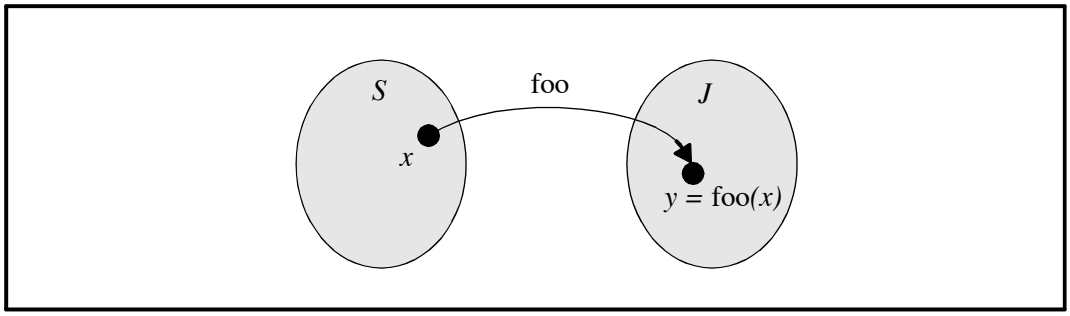


Figure 1 : Graphe de la fonction foo de domaine S et de codomaine J. La fonction «foo» est l'ensemble des flèches qu'on peut ainsi établir.

Et la définition de la fonction se termine par une phrase de la forme

«... et pour associer un *jabberwok* à chaque *snark*, on utilise la méthode suivante...»

Les différentes formes de la définition d'une fonction correspondent aux différentes techniques permettant de décrire la méthode d'association.

2.2.1 Construction d'un domaine ou d'un codomaine

Dans un premier temps, c'est à dire dans ce chapitre, nous allons supposer qu'il existe des *données* et des *fonctions*, que les données sont des entités passives tandis que les fonctions agissent sur les données. Nous verrons plus tard que si cette vision est commode pour apprendre à définir et manipuler des fonctions, elle ne nous convient pas et nous préfèrerons considérer que les données ne seront que des fonctions déguisées [5]. Mais cela est une autre histoire.

Nous suposerons donc qu'il existe quelques domaines primitifs tels l'ensemble des nombres entiers, l'ensemble des signes typographiques, l'ensemble des nombres réels etc. Certains sont finis et d'autres ne le sont pas. Les mathématiciens ont longuement peiné pour définir les ensembles infinis aussi les utiliserons-nous «en l'état» en prenant bien soin de ne pas y introduire d'améliorations de notre cru. Ainsi, moyennant quelques précautions, une approche naïve nous suffira et nous pourrons définir les domaines dont nous avons besoin soit en restreignant un domaine existant à l'aide d'une propriété particulière soit en composant des domaines existant.

[5] Nous venons de voir que les mathématiciens ont pris l'optique exactement inverse et pour eux tout est «donnée». C'est ce qui distingue, entre autre, les informaticiens des mathématiciens. Le paragraphe 2.9, page 39 nous donnera une idée des conséquences de ce choix.

Construction par restriction

Le domaine nécessaire est obtenu en «filtrant» un autre domaine à l'aide d'une propriété exprimée sous la forme d'un *prédicat*. Un prédicat est une affirmation dont on peut dire sûrement si elle est vrai ou fausse. On peut, par exemple, définir l'ensemble des nombres pairs par

$$\text{Nombres-pairs} = \{n \in \mathbf{N} \mid (2 \text{ divise } n)\}$$

ce qui se lit «l'ensemble des nombres pairs est l'ensemble des nombres n tels que n appartient à l'ensemble des nombres entiers et que 2 divise n.» et plus simplement «l'ensemble des nombres pairs est l'ensembles des nombres entiers divisibles par 2.»

On prendra garde de ne pas inclure le domaine que l'on cherche à définir dans la définition elle-même sous peine de risquer de redoutables paradoxes. Le plus célèbre d'entre eux est probablement le «paradoxe du barbier» du au mathématicien-logicien Bertrand Russell «Le barbier est celui qui rase ceux qui ne se rasent pas eux-même. Mais qui rase le barbier ?»

Construction par composition

On est très fréquemment amené à décrire une chose par un ensemble fini de caractéristiques considérées comme pertinentes. Ce regroupement de caractéristiques s'appelle un *n-uplet*. Ainsi, un rectangle peut être caractérisé par le 2-uplet (doublet) de sa longueur et de sa largeur

$$\mathbf{R}^+ = \{x \mid x \in \mathbf{R} \wedge x \geq 0\}$$
$$\mathbf{Rect} = \{(L,l) \mid L \in \mathbf{R}^+ \wedge l \in \mathbf{R}^+\}$$

L'opération qui consiste à associer deux domaines pour construire un domaine de n-uplets s'appelle le *produit cartésien* [6] noté $\cdot$ (dire *croix*) et

$$\mathbf{Rect} = \mathbf{R}^+ \cdot \mathbf{R}^+$$

2.2.2 Définition en intention

On dira qu'une fonction est définie *en intention* lorsque l'association entre chaque élément de son domaine et les éléments correspondants de son codomaine est décrite par une *formule finie calculable*.

Considérons, par exemple, la fonction «produit» dont le domaine est

$$D = \{0,1,2,3,4\} \cdot \{0,1,2,3,4\}$$

et dont le codomaine est

$$C = \{0,1,2,3,4,6,8,9,12,16\}$$

On peut la définir en intension par [7]

$$\text{produit}(x,y) = \{x \cdot y \mid x \in \{0,1,2,3,4\} \wedge y \in \{0,1,2,3,4\}\}$$

Cette définition est *opérationnelle* en ce sens qu'elle nous donne un moyen d'évaluer cette

[6] Cette opération est aussi appelée *produit externe* (cross-product).

[7] Ici, le signe $\cdot$ dénote la multiplication traditionnelle.

fonction.

2.2.3 Définition en extension

On dira qu'une fonction est définie *en extension* lorsque l'association entre chaque élément de son domaine et les éléments correspondants de son codomaine est décrite par la liste exhaustive de tous les doublets (x,y) qui la constitue.

Considérons la fonction «annuaireTélécom» telle que France Télécom peut la définir. Son domaineest l'ensemble des abonnés d'un département et son codomaine celui des numéros de téléphone

annuaireTélécom : $\overrightarrow{NomsNumérosTéléphone}$
avec : $\{$*Kleene*, *Church*, *Schwartz*, *Smith*, *Turing*, *Zenegger*, ⋯$\}$, $\{$93-21-79-00, 93-41-59-60, ⋯$\}$

On peut la définir en extension par

annuaireTélécom = $\{$(*Kleene*, 93-41-59-60), (*Schwartz*, 93-21-79-00), ⋯$\}$

Cette définition de la fonction est dite *non opérationnelle* car aucune méthode de calcul ne lui est associée. En d'autres termes, on a défini *ce qu'est* la fonction mais on ne s'est donné aucun moyen permettant de l'évaluer au fur et à mesure des besoins. Une telle définition n'est possible que lorsque le domaine et le codomaine de la fonction sont des ensembles finis (et pas trop grands).

2.2.4 Techniques de définition

C'est ici que notre route se sépare de celle des mathématiciens. Alors que l'essentiel de leur activité est la définition de concepts nouveaux, l'étude approfondie des propriétés de fonctions très particulières, etc., la nôtre va uniquement consister à définir (et construire puis vendre) les fonctions utiles à nos clients.

Nous avons donc besoin de mettre au point quelques techniques permettant de construire le plus systématiquement possible des définitions de fonctions à partir d'une spécification ou même plus simplement de l'expression d'un besoin.

Il arrive très fréquemment qu'on ne sache pas définir une fonction sur son domaine par une formule unique. Par contre, si on découpe son domaine en un ensemble fini de sous-domaines disjoints, la fonction peut être définie par différentes formules associées aux différents sous-domaines. Pensez, par exemple, au calcul des impôts sur le revenu à partir du quotient familial et des tranches d'imposition.

Une telle forme de définition est un compromis entre la définition en extension (les sous-domaines) et la définition en compréhension (les formules). Cette forme de définition est appelée *définition par morceaux* par les mathématiciens et *définition par cas* par les programmeurs. C'est une forme évoluée de la définition en intention.

L'analyse «par cas» d'un problème peut être effectuée en suivant une des 2 stratégies suivantes :

1. Découpage du domaine en sous-domaines disjoints.
2. Transformation du problème initial en un problème plus simple, la règle de transformation étant recherchée soit *directement*, soit par *induction* soit sous la forme d'un

invariant.

Définition directe «par cas»

Considérons la fonction bien connue qui représente la valeur absolue d'un nombre. Cette fonction, notée $|x|$ que nous appellerons «abs» est telle que, par exemple

$$\begin{aligned} abs(4) &= 4 \\ abs(-2) &= 2 \\ abs(0) &= 0 \end{aligned}$$

Son domaine est, par exemple, l'ensemble des nombres entiers relatifs et son codomaine celui des entiers naturels.On peut partager ce domaine en trois sous-domaines disjoints, celui des nombres entiers positifs, celui de 0 et celui des nombres entiers négatifs, la définition peut alors être mise sous la forme

$$abs(x) = \begin{cases} x & si\ x > 0 \\ 0 & si\ x = 0 \\ -x & si\ x < 0 \end{cases}$$

Les termes de la forme «$si\ x > 0$» qui décrivent le découpage du domaine s'appellent les *gardes* des formules qui leur sont associées. Les termes de la forme «$x > 0$» qui servent à construire les gardes sont des *prédicats*. Une telle définition est dite constituée d'un ensemble de *formules gardées*.

Afin de donner une forme plus facile à manipuler et à écrire, nous allons introduire **une première fonction, que nous considérerons, pour l'instant, comme primitive** (au même titre que les opérations arithmétiques traditionnelles), nous permettant de décrire le rôle joué par l'accolade.

Nous l'appellerons «si» notée [8]

$$si(p, a, b) \equiv p \rightarrow a, b$$

et dont la définition sémantique informelle est

$$Valeur\text{-}de(p \rightarrow a, b) = \begin{cases} Valeur\text{-}de(a) & si\ Valeur\text{-}de(p) = vrai \\ Valeur\text{-}de(b) & si\ Valeur\text{-}de(p) = faux \end{cases}$$

Cette fonction permet de définir le découpage d'un domaine en sous-domaines disjoints par des dichotomies successives. Elle permet de mettre la définition de la fonction «abs» sous les deux formes équivalentes suivantes

$$abs(x) = \begin{array}{l} x > 0 \rightarrow x, \\ x = 0 \rightarrow 0, \\ sinon \rightarrow -x \end{array} \qquad soit\ abs(x) = x > 0 \rightarrow x, -x$$

Nota: *«sinon» dénote un prédicat toujours «vrai».*

L'analyse par cas, telle que la permet la fonction «si» est un outil remarquablement puissant

[8] Cette notation est due à John MacCarthy, concepteur du langage Lisp.

pour définir des fonctions. Nous allons montrer comment l'utiliser pour introduire des définitions curieuses mais très riches.

Recherche d'une récurrence

Essayons de définir la fonction appelée «factorielle», définie sur les nombres entiers positifs, notée *n*!, et que nous appellerons «fact». Cette fonction est telle que

$$fact(3) = 3 \cdot 2 \cdot 1$$
$$fact(4) = 4 \cdot 3 \cdot 2 \cdot 1$$
$$fact(0) = 1$$

On distingue immédiatement le cas trivial qui nous permet d'écrire l'ébauche de définition suivante

$$fact(n) = (n = 0) \rightarrow 1;....?.....$$

On note que le signe = joue une double rôle. Il lie la définition de la fonction «fact» à son nom et il compare n à 1. Cette ambiguïté est en général levée par le contexte. Cette ambiguïté est plus apparente que réelle. En effet, il suffit, pour la lever, de considérer que le signe = qui établit un lien entre un nom et une valeur dénote lui aussi un prédicat et de considérer que la solution d'une équation de la forme $x = 5$ peut être interprétée comme la valeur qu'il faut attribuer à x pour que le prédicat correspondant soit *vrai*.

La fonction «fact» étant une suite de multiplications, il est clair qu'évaluer $fact(n)$ est un problème plus simple qu'évaluer $fact(n-1)$, la règle de transformation étant

$$fact(n) = n \cdot fact(n-1)$$

On peut alors compléter la définition de la fonction «fact»

$$fact(n) = (n = 1) \rightarrow 1;n \cdot fact(n-1)$$

Cette définition est inhabituelle en ce sens qu'elle fait référence à elle-même. On dit qu'elle est *récursive*. La récursivité est le seul mécanisme de construction connu permettant de définir des formes infinies a priori.

C'est ce mécanisme qui permet, par exemple, de définir l'ensemble des nombres entiers en disant simplement «0 est un nombre entier et si n est un nombre entier alors $suivant(n)$ est un nombre entier».

L'approche que nous avons utilisée s'appelle *raisonner par induction*. Elle est possible chaque fois qu'on définit, en fait, une suite indicée (donc infinie dénombrable) de problèmes de la même famille. Elle comporte toujours 2 étapes :

1. Etablir la règle de transformation qui donne la solution du problème d'indice *n* à partir de la solution du problème d'indice *n*-1 supposée connue.
2. Résoudre de façon triviale le problème d'indice initial (1 ou 0 en général).

Analysons l'utilisation de la définition de la fonction «fact» et évaluons, à titre d'exemple, l'expression $fact(4)$. On peut déduire de cette définition l'ensemble des égalités suivantes

$$fact(4) = 4 \cdot fact(3)$$
$$fact(3) = 3 \cdot fact(2)$$
$$fact(2) = 2 \cdot fact(1)$$
$$fact(1) = 1 \cdot fact(0)$$
$$fact(0) = 1$$

qui constituent un système d'équations qui peut être résolu par substitution et dont la solution est évidemment

$$fact(n) = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

Recherche d'un invariant

Définissons, à présent, un prédicat, noté[9] «divise?», qui nous permet de tester si un nombre positif non nul en divise un autre. Ce prédicat est tel que, par exemple

$$\begin{aligned} divise?(2,4) &= vrai \\ divise?(3,22) &= faux \\ divise?(5,80) &= vrai \end{aligned}$$

On distingue immédiatement 2 cas triviaux qui nous permettent d'écrire l'ébauche de définition suivante

$$\begin{aligned} divise?(n,m) = {} & si\ (n = 1) \rightarrow vrai, \\ & si\ (n > m) \rightarrow faux, \\ & sinon \rightarrow ?..... \end{aligned}$$

C'est l'arithmétique qui va nous apporter la partie de définition qui nous manque. En effet, on dit que le nombre n divise le nombre m s'il existe un nombre q supérieur ou égal[10] à 1 tel que $m = q \cdot n$ mais alors

$$\begin{aligned} m &= q \cdot n \\ m &= q \cdot n + n - n \\ m - n &= q \cdot n - n \\ m - n &= (q - 1) \cdot n \end{aligned}$$

En d'autres termes, dans le cas non trivial où m est supérieur à n, la divisibilité de m par n est la même que la divisibilité de m-n par n ce qui se traduit par l'égalité

$$divise?(n,m) = divise?(n, m-n)$$

On dit alors que la divisibilité par n est un *invariant* relativement à la soustraction de n. Plus généralement, si on peut trouver une règle de la forme

$$prédicat?(x) = prédicat?(f_1(x), f_2(x), f_3(x), \cdots)$$

On dira que «prédicat?» est un invariant relativement aux fonctions «» f_1, f_2, f_3 $\cdots$ ces fonctions *préservent l'invariant*.

La forme finale de la définition du prédicat «divise?» est alors

$$\begin{aligned} divise?(n,m) = {} & si\ (n = 1) \rightarrow vrai, \\ & si\ (n > m) \rightarrow faux, \\ & sinon \rightarrow divise?(n, m-n) \end{aligned}$$

---

[9] Comme les prédicats correspondent à une question à laquelle on ne peut répondre que par *vrai* ou *faux*, nous conviendrons de terminer leur nom par?. En règle générale, nous serons très attentif à donner aux choses qui nous intéressent un nom significatif afin de faciliter la lecture de ce que nous écrivons.

[10] Le cas $q = 1$ correspond au cas trivial et le cas $m = n$ au cas trivial . $q = 0$ $n > m$

Cette définition, dans le cas où a vaut 3, engendre les égalités suivantes

$$\begin{aligned} &divise(3,9) = divise(3,6) \\ &divise(3,6) = divise(3,3) \\ &divise(3,3) = vrai \end{aligned}$$

qui constituent un système d'équations dont la solution (par élimination) est évidemment

$$divise(3,9) = vrai$$

et dans le cas où a vaut 3, elle engendre un système d'équations dont la solution est

$$divise(3,8) = faux$$

Chercher un invariant est une technique très puissante [11] pour trouver une définition récursive. Comme elle est assez délicate à utiliser, nous en reparlerons souvent. Nous verrons plus loin qu'une définition récursive obtenue de cette manière a des propriétés très remarquables.

2.2.5 Egalité de deux fonctions

Un *cahier des charges* est (presque) toujours une définition *non opérationnelle* de la fonction «f» à réaliser. Ecrire un programme à partir de ce cahier des charges, c'est définir une fonction «g» de façon *opérationnelle* égale à la fonction «f» spécifiée. Il est donc nécessaire, dans le cadre d'une relation Client-Fournisseur, de définir l'égalité de ces deux fonctions. En général, le contrat définit une procédure de *Recette* du produit livré permettant de vérifier que la fonction «g» livrée peut être considérée comme suffisamment conforme à la fonction «f» commandée.

Notre définition de la fonction ne laisse pas beaucoup de possibilités pour définir cette égalité dans le cas général, on ne peut la définir qu' *en extension*.

Egalité en extension : la plus générale, mais la moins utile.

On dira que les fonctions «f» et «g» sont *égales en extension* si :

1. elles sont définies sur le même domaine et le même codomaine,
2. pour tout élément de leur domaine, «f» et «g» mettent en correspondance le même élément de leur codomaine.

Ainsi, pour tester l'égalité de deux fonctions, cette définition exigerait d'essayer de manière exhaustive tous les éléments de D. Cette procédure est impraticable si D est un ensemble infini (nombres entiers ou réels par exemple), la question «les fonctions «f» et «g» sont-elles égales ?» étant dans ce cas, **indécidable**.

> Nous verrons que cette définition de l'égalité peut conduire à des paradoxes redoutables. En effet, elle suppose qu'on sache comparer entre eux les éléments du codomaine c'est à dire qu'on a su définir leur égalité. Nous n'avons donc pas fini de parler d'égalité.
>
> Ce phénomène n'est ni étrange ni rare. Il correspond au fait qu'il est très souvent (pour ne pas dire pratiquement toujours) impossible de prouver qu'un système fonctionne comme il le devrait en faisant des essais successifs. Tant que les essais sont positifs, on ne peut que

[11] C'est la puissance de cette technique qui a conduit les physiciens à inventer le concept d'énergie en tant qu'invariant universel.

conclure qu'on n'a probablement pas essayé les conditions dans lesquelles ce système ne fonctionne pas. Seuls les essais négatifs sont malheureusement probants !

Même si l'ensemble D n'est pas infini il peut être trop grand pour qu'on puisse tester l'égalité de deux fonctions par exhaustion des éléments de D. Imaginer le temps qui peut être nécessaire à la vérification de l'égalité de deux encyclopédies. De telles tâches, bien que possibles sur un plan théorique mais dont l'évaluation demanderait des temps gigantesques constituent ce que les philosophes logiciens actuels appellent des super-tâches considérées comme pratiquement impossibles à réaliser. Nous verrons dans un prochain chapitre comment vérifier que la définition que nous avons associé à une fonction ne correspond pas à une super-tâche.

Comme ce problème est fréquent, il a été nécessaire d'imaginer une définition plus pratique : l'*égalité en intension*.

Egalité en intention : la définition pratique des mathématiciens.

On dira que les fonctions «f» et «g» sont *égales en intention* si :

1. elles sont définies sur le même domaine et le même codomaine,
2. on peut passer de la définition de «f» à celle de «g» par une suite de transformations qui conservent l'*identité*.

On dira que deux choses sont identiques s'il s'agit, en fait, de la même chose. C'est à dire que toute action effectuée sur l'une peut être perçue sur les deux en même temps. Nous nous contenterons de cette définition informelle. Les mots «identique» et «égal» sont pratiquement synonymes dans le langage courant. Cette ambiguïté rend difficile la description de la situation suivante «En regardant dehors par les deux fenêtres de ma salle à manger, je vois deux voitures que mes yeux ne savent pas distinguer. S'agit-il de deux exemplaires différents du même type de voiture ou s'agit-il d'une voiture et de son reflet dans la vitrine d'en face?». Dans le premier cas on dira que les deux voitures sont **égales** tandis que dans le deuxième qu'elles sont **identiques**.

Considérons, par exemple, les deux fonctions :

$$f(a,b) = (a+b)^2$$
$$g(a,b) = a^2 + b^2$$

Elles sont égales *en intention* car :

$$\begin{aligned} f(a,b) &= (a+b)^2 \\ &= a^2 + baab + b^2 \\ &= a^2 + b^2 \\ &= g(a,b) \end{aligned}$$

Ce critère d'égalité revient à constater que les deux fonctions «f» et «g» ne sont que deux «reflets» différents de la même fonction.

On peut remarquer, sur cet exemple, qu'une grande partie de l'algèbre est consacrée à la définition de transformations (les calculs algébriques) qui préservent l'identité. Cette définition est beaucoup trop sévère pour nous car nous ne saurons pas, en général, transformer les définitions des fonctions que nous serons amenés à considérer tout en préservant leur identité. De telles manipulations sur les fonctions relève d'une branche assez nouvelle de l'informatique : le *calcul formel*.

Equivalence opérationnelle : la définition pragmatique.

Nous constatons que le mot égalité est particulièrement ambigu. Ainsi, plutôt que de parler de l'égalité en soi, on préférera parler d' *équivalence opérationnelle* . Cela revient à consi-

dérer que deux choses sont opérationnellement équivalentes pour nous (égales) si on peut en avoir le *même usage*. On ne pourra donc parler d'égalité (d'équivalence opérationnelle) que dans un certain contexte qu'il nous appartiendra de définir à chaque fois. Pour parler de façon un peu triviale, nous pouvons dire que nous avons réussi à «botter en touche». N'étant pas capables de définir l'égalité en général, nous laisserons ce soin à notre client qui lui, sachant ce que les choses représentent, est le mieux placer pour en définir l'équivalence et nous fournir son *prédicat d'égalité*, en général, sous la forme d'une *procédure de recette*.

> If it paddles like a duck, waddles like a duck and quacks like a duck why not to say: it is a duck ?
>
> Lewis Carroll

2.3 Composition des fonctions

Le seul mécanisme pour élaborer des fonctions complexes est la *composition des fonctions*. Nous l'avons déjà utilisé de manière naturelle lorsque nous avons construit des formules en associant des opérateurs. L'idée en est très simple, considérons les deux fonctions «f» et «g»

$$f : A \rightarrow B$$
$$g : B \rightarrow C$$

Etant donné un élément a de A, on peut trouver un élément b de B tel que $b = f(a)$ puis un élément c de C tel que $c = g(b)$. Il existe une fonction de domaine A et de codomaine C.

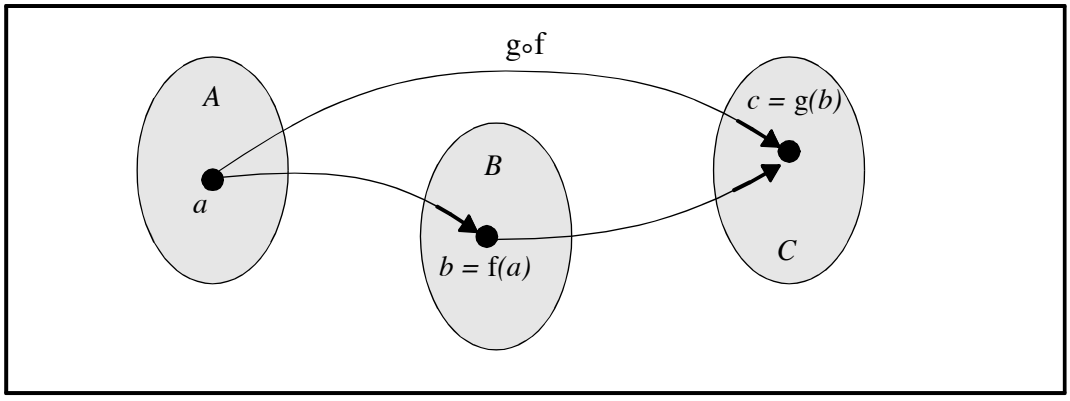


Figure 2 : Composition des deux fonctions f et g.

Cette définition de la fonction composée notée $(g \circ f)$ (cf. figure 2, page 18) peut être également mise sous la forme

$$g \circ f : A \rightarrow C \quad = \{\forall a \in A, \exists b \in B \text{ et } c \in C \mid b = f(a) \text{ et } c = g(b)\}$$

Transparence par référence

La composition des fonctions utilise une propriété fondamentale des fonctions

> Le résultat de l'application d'une fonction à ses arguments dépend uniquement de la valeur de ses arguments. En particulier, si cette opération est effectuée plusieurs fois, elle donne à chaque fois le même résultat.

Cette propriété s'appelle *transparence par référence*. C'est cette propriété qui nous permet d'introduire des variables dans les formules. Tout calcul serait pratiquement impossible s'il n'en était pas ainsi. C'est donc une propriété que nous essaierons de préserver à tout prix.

2.4 Techniques de conception

> Voici en quoi consiste principalement ces actes de l'esprit :
>
> 1. A combiner plusieurs idées simples en une seule; c'est par ce moyen que se font toutes les idées complexes.
> 2. A joindre deux idées ensemble, soit qu'elles soient simples ou complexes, et à les placer l'une près de l'autre, en sorte qu'on les voit toutes à la fois sans les combiner en une seule idée: c'est par là que l'esprit se forme toutes les idées des relations.
> 3. Le troisième de ces actes consiste à séparer des idées d'avec toutes les autres qui existent réellement avec elles: c'est ce qu'on nomme abstraction, et c'est par cette voie que l'esprit se forme toutes les idées générales.
>
> John Locke
> Essai philosophique sur
> l'entendement humain (1690)

Ces 3 mécanismes sont donc connus depuis fort longtemps et nous les utilisons couramment sans y prendre garde. Le premier correspond à l'opération de composition des fonctions, le deuxième nous conduira à la définition et à la structuration des données et fera l'objet d'un chapitre ultérieur et le troisième est à associer à la notion de *procédure*.

> Selon le sens commun, une procédure est la définition d'une méthode de travail présentant un certain caractère de généralité. Nous connaissons tous les procédures nommées recettes de cuisine et la vie professionnelle ou privée est remplie de procédures (souvent administratives). Pendant longtemps, en mathématiques, les procédures ont été définies par des algorithmes décrivant des successions d'actions à effectuer dans le bon ordre, puis la définition algorithmique a été abandonnée (au XIXème siècle) au profit de la notion plus riche de définition fonctionnelle. En ce qui nous concerne, une procédure sera toujours définie par une fonction. Déterminer le bon ordre des choses lors de la définition d'un algorithme est un problème souvent complexe. Par contre, les dépendances fonctionnelles utilisées lors de la définition fonctionnelle d'une procédure induit l'ordre naturel des choses.

Nous allons donc simplement essayer d'analyser assez finement ces mécanismes pour en percevoir tous les aspects qui nous intéressent.

2.4.1 Abstraction par composition & nommage[12]

Lorsqu'on a construit une nouvelle fonction par composition de fonctions plus simples, on lui a donné un *nom symbolique* qui nous a permis de la manipuler comme un tout sans se soucier de ses détails de définition.

> Ainsi, quand nous utilisons les fonctions «+, -,abs,...» nous ne soucions que de l'usage qu'on peut en faire et pas du tout de leurs détails de définition.

En mathématiques, le signe = établit un *lien* entre un *nom* (une variable) et une *valeur* (le résultat d'un calcul, par exemple) de telle sorte que ce nom et cette valeur puissent être uti-

[12] Ce mot dénote l'*action de nommer*.

lisés indifféremment, l'un à la place de l'autre, partout où l'un ou l'autre apparaît.

Considérons, par exemple, l'écriture classique suivante

$$\begin{array}{l}\text{soit :}x = 24\\ \qquad\quad y = 3\\ \text{dans :}xy\end{array}.$$

On y a introduit la variable *x* en donnant un nom à la valeur 24 et la variable *y* en donnant un nom à la valeur 3. Nous avons ensuite utilisé ces variables pour construire une formule. Le nom *x* a été **lié** à la valeur 24 et le nom *y* à la valeur 3. Ces liens sont permanents et définitifs dans tout le contexte défini par «soit:» lequel **donne un sens** à la formule qui lui a été associée par «dans:».

Considérons les deux expressions suivantes

$$()_a \left[\begin{array}{l}\text{soit :}x = 3\\ \qquad\quad y = 10\\ \text{dans :}xy\end{array}\right]. \qquad ()_b \left[\begin{array}{l}\text{soit :}x = 3\\ \qquad\quad y = 10\\ \text{dans :}xyz{+}{+}\end{array}\right]$$

Dans l'expression (a), tous les noms qui interviennent dans la formule apparaissent dans le contexte. Dans ce cas, on dit que la formule est *fermée* et que ses variables sont *liées*. Par contre, dans l'expression (b), la formule contient la variable *z* qui n'a pas été liée. La variable *z* est dite *libre* et la formule est dite *ouverte*.

Une **formule fermée est évaluable** et on dira que la valeur qu'elle prend lors de son évaluation en constitue le *sens*. Par contre, une **formule ouverte n'est pas évaluable** et on dira qu'**elle n'a pas de sens**. Afin de symétriser le comportement des formules ouvertes et fermées, on dira que la valeur d'une formule ouverte est la non-valeur notée $\perp$ (dire *bottom*).

Ainsi

$$\begin{array}{l}34 = 12\\ 34 \cdot \perp x =\end{array}$$

Bâtir une application de cette façon consiste à créer un *environnement* (un contexte) contenant tous les liens (*nom*$\leftrightarrow$*valeur*) qui ont été établis pour donner un sens à une expression. La formule dont les variables sont définies par un environnement constitue ce qu'on appelle la *portée* de ces variables («scope» en anglo-américain), la *durée de vie* («extent» en anglo-américain) des liens qui y sont définis est a priori illimitée.

2.4.2 Abstraction par généralisation

Certaines formules ont un caractère de généralité qu'il serait dommage de ne pas utiliser. Nous savons que le volume d'un parallélépipède rectangle de longueur 3, de largeur 4 et de hauteur 5 est donné par la formule

$$volume34 \cdot \cdot 5$$

On peut être amené, dans une même application, à calculer plusieurs volumes

$$\begin{bmatrix} \text{soit : } volumeClasse = \cdots, \\ \quad volumeCouloir = \cdots, \\ \quad volumeCantine = \cdots, \\ \quad volumeBureau = \cdots, \\ \text{dans : } \cdots\cdots \end{bmatrix}$$

Il est alors avantageux de généraliser la notion de volume en utilisant une expression contenant des *paramètres*

$$\text{volume}(L, l, h) = \cdots h$$

Ces paramètres L, l et h ne sont que des *trous* dans l'expression . Nous avons ensuite donné un nom à cette formule à trous qui est ainsi devenue la fonction «volume».

Lorsqu'on veut utiliser la définition du *volume en général* pour calculer *un volume en particulier* ont lie ses paramètres à des *arguments*. Le volume en général correspond alors à la structure

$$\text{volume} = \begin{bmatrix} \text{soit : } L = \cdots \\ \quad l = \cdots \\ \quad h = \cdots \\ \text{dans : } L \cdots h \end{bmatrix}$$

Les noms introduits pour construire cette généralisation constituent l' *environnement privé* de la fonction. Cet environnement est constitué de noms qui ne sont pas encore liés à des valeurs mais qui lient les variables contenues dans la formule de définition du volume. Une fonction peut donc être considérée comme une formule *fermée évaluable plus tard*.

Nous avons ainsi transformé une formule en fonction. Cette opération s'appelle l' *abstraire fonctionnellement* relativement au triplet des variables L, l et h. Afin de donner une forme plus facile à manipuler et à écrire, nous allons introduire notre **deuxième fonction primitive** pour décrire la construction d'une fonction, nous la noterons λ[13] (dire *lambda*) et nous représenterons son application à ses argument en utilisant la syntaxe spéciale introduite par ses inventeurs.

La définition du volume s'écrira alors

$$\text{volume} \in \mathbf{R}^+ \cdot \mathbf{R}^+ \cdot \mathbf{R}^+ \rightarrow \mathbf{R}^+$$
$$\text{volume} = \lambda(L, l, h) \cdots() \cdots h$$

Ce qui veut dire

> «la fonction *lambda* engendre, à partir de l'expression , une fonction dépendant du 3-uplet (triplet) (L, l, h)

[13] Le nom «lambda» utilisé ici est celui qui a été donné à l'opération d'abstraction dans *le lambda-calcul*, la théorie des fonctions élaborée par Alonzo Church en 1941 puis abondamment développée par Curry et Schönfinkel. On peut regretter son peu de convivialité mais on le conserve en l'honneur de son inventeur. La fonction engendrée par la fonction *lambda* est anonyme, c'est le signe = qui lui a donné le nom de *volume* dans un environnement particulier.

Le «point» sépare les paramètres de la fonction de son expression de définition. Nous verrons que cette expression peut prendre des formes très variées. Il est clair que dans cette expression, les noms L, l et h n'ont pas d'importance et que «volume» peut aussi bien être défini par

$$\text{volume} = \lambda(x,y,z) \cdot x \cdot y \cdot z$$

2.4.3 Abstraction fonctionnelle & Curryfication

Si on reprend l'expression $x \cdot y$, on peut ne l'abstraire que relativement à une seule de ses variables et définir la fonction

$$\lambda z \cdot x \cdot y \cdot z$$

dans ce cas, l'abstraction n'est que partielle et l'expression correspondante est *ouverte*.

Cette fonction peut être abstraite à son tour relativement à sa variable libre y ce qui permet de définir la variable

$$\lambda y \cdot \lambda z \cdot x \cdot y \cdot z$$

Si on abstrait, à présent, cette fonction relativement à sa variable libre x, elle devient la fonction

$$\text{volume} \in \mathbf{R}^+ \to \mathbf{R}^+ \to \mathbf{R}^+ \to \mathbf{R}^+$$
$$\text{volume} = \lambda x \cdot \lambda y \cdot \lambda z \cdot x \cdot y \cdot z$$

qui est à présent fermée. Lorsque les variables libres d'une expression ont été abstraites les unes après les autres, la fonction ainsi obtenue est dite *curryfié* [14]. La curryfication permet de transformer les fonctions dont le paramètre est un n-uplet en une composition de fonctions à un seul paramètre.

Nous verrons un peu plus tard qu'il peut être intéressant de ne pas abstraire toutes les variables d'une expression simultanément. La curryfication est une opération extrêmement puissante et les quelques exemples qui suivent vont le montrer.

Une fonction pour curryfier les fonctions non curryfiées

Imaginons que nous disposions de fonctions dont le paramètre est un doublet (un 2-uplet) et que nous en désirions une version curryfiée. On pourrait, bien sûr, les curryfier «à la main», ce qui deviendrait bien vite très laborieux. Il est plus astucieux de définir la fonction «curry»

$$\text{curry} \in (X \times Y \to Z) \to X \to Y \to Z$$
$$\text{curry} f = \lambda x \cdot \lambda y \cdot f(x,y)$$

[14] L'adjectif curryfiée a été créé en honneur de **Haskell Curry**, le logicien qui a introduit cette notation le premier.

A titre d'exemple, curryfions la fonction d'addition $add(x,y) = x + y$

$$\begin{aligned} addcurry &= curryadd() \\ &= [\lambda f \cdot \lambda x \cdot \lambda y \cdot f(x,y)](add) \\ &= \lambda x \cdot \lambda y \cdot add(x,y) \end{aligned}$$

On peut se demander l'intérêt d'une fonction d'addition curryfiée. C'est son application partielle qui va nous apporter la réponse

$$\begin{aligned} addcurry(1)(x) &= [\lambda y \cdot add(x,y)](1) \\ &= \lambda y \cdot add(1,y) \end{aligned}$$

La fonction $addcurry(1)$ obtenue est une fonction d'incrémentation.

Une fonction pour décurryfier les fonctions curryfiées

On peut, de la même manière, définir une fonction pour décurryfier des fonctions à deux paramètres curryfiées.

$$\begin{aligned} &décurry : (X \to (Y \to Z)) \to (X \times Y \to Z) \\ &décurry(f) = \lambda(x,y) \cdot f(x)(y) \end{aligned}$$

On peut, bien sûr, vérifier que la curryfication et la décurryfication sont bien des opérations inverses.

$$\begin{aligned} décurry(curry(g)) &= [\lambda f \cdot \lambda(x,y) \cdot f(x)(y)]([\lambda f \cdot \lambda x \cdot \lambda y \cdot f(x,y)](g)) \\ &= [\lambda f \cdot \lambda(x,y) \cdot f(x)(y)](\lambda x \cdot \lambda y \cdot g(x,y)) \\ &= [\lambda f \cdot \lambda(x,y) \cdot f(x)(y)](\lambda u \cdot \lambda v \cdot g(u,v)) \\ &= \lambda(x,y) \cdot (\lambda u \cdot \lambda v \cdot g(u,v))(x)(y) \\ &= \lambda(x,y) \cdot g(x,y) \\ &= g \end{aligned}$$

On vérifierait, de même, que $curry(décurry(h)) = h$. On peut remarquer, en passant, qu'il a été nécessaire de renommer les paramètres d'une fonction afin d'éviter une *collision de noms*.

2.4.4 Application d'une fonction à ses arguments & évaluation

Lorsqu'on veut utiliser le *volume en général* pour calculer un *volume en particulier*, on *applique* la fonction «volume» aux arguments qui vont être liés à ses paramètres. L'application de cette fonction aux arguments 5, 7 et 2 s'écrit

$$volume(5,7,2) = [\lambda(L,l,h) \cdot L \cdot l \cdot h](5,7,2) = 5 \cdot 7 \cdot 2$$

Cette expression correspond à la structure suivante

$$volume(5,7,2) = \begin{bmatrix} \text{soit : } L = 5 \\ l = 7 \\ h = 2 \\ \text{dans : } L \cdot l \cdot h \end{bmatrix}$$

Appliquer une fonction à ses arguments, revient à *lier les noms* de son environnement privé aux données fournies comme arguments. Ces liens donnent un sens à l'expression qui a été généralisée. On peut se représenter ce mécanisme en disant que **les arguments ont été substitués aux paramètres** dans l'expression de définition de la fonction. L' *évaluer* consiste à effectuer toutes les opérations alors possibles.

Ainsi

volume-nc5)72*h* ==[2]-(,) , ·()*l*· *h* (5572 · · 2 70

Nous allons revenir bientôt sur le mécanisme de l'application d'une fonction à ses arguments car ce mécanisme présente deux aspects importants.

Considérons, à présent, la version curryfiée de la fonction volume et appliquons-la à l'argument 5

volume-c5)*x* ==··λ[λ*y* λ*z* ·()*xy* *z* (5*y* λ ··λ*z* ·()· *y* *z*

On obtient alors une fonction à deux paramètres. L'application complète aux arguments 5, 7 et 2 s'écrirait donc

volume5)7()2()57= · · 2

2.4.5 Définition d'un environnement

Considérons l'expression précédente

$$\begin{bmatrix} \text{soit : } x = 3 \\ \qquad\quad y = 10 \\ \text{dans : } xy \end{bmatrix}$$

On peut l'écrire sous la forme suivante qui lui servira de définition

·[2]-,()*xy* ·()*xy* (3110 == · 30

Nous constatons, alors, que la fonction «lambda» permet, non seulement de définir une fonction, mais aussi de définir un environnement et qu'il n'est pas nécessaire d'introduire une nouvelle fonction primitive pour décrire le rôle joué par

$$\begin{bmatrix} \text{soit : } \cdots\cdots\cdots \\ \text{dans : } \cdots\cdots\cdots \end{bmatrix}$$

Nous venons surtout de constater qu'un environnement n'est jamais qu'une fonction. Nous utiliserons ces deux écritures équivalentes selon nos besoins.

2.4.6 Evaluation en ordre normal & évaluation en ordre applicatif

Dire que «**les arguments ont été substitués aux paramètres**» laisse un problème subtil en suspens : les arguments sont-ils évalués **avant** d'être substitués ou **après** avoir été substitués ?

Considérons les définitions suivantes

$$\text{carré}\,x = \lambda\ \cdot\ xx$$
$$\text{somme-carré}\,xy = \lambda(,)\ \cdot\ \text{carré}\,x() + \text{carré}\,y()$$
$$\text{foo}\,x = \lambda\ \cdot\ \text{somme-carré}\,() + 1, x + 2$$

puis évaluons .foo

A partir de là deux voies s'offrent à nous. On peut systématiquement évaluer les arguments d'abord, ce qui nous donne

$$\begin{aligned}
\text{foo}\,x &= \cdot[\lambda] \quad \text{somme-carré}\,() + 1, x + 2 \quad ()\\
&= \text{somme-carré}\,() + 2, \quad +\\
&= \text{somme-carré}\,(), \\
&= \cdot[\lambda](,)\ \text{carré}\,x() + \text{carré}\,y() \quad ()\\
&= \text{carré}\,() + \text{carré}\,() \quad ()\\
&= \cdot[\lambda]\ x\ \ xx \quad () + \cdot[\lambda]\ \ xx \quad ()\\
&= \quad + \\
&= \\
&= 41
\end{aligned}$$

Cette technique d'évaluation est dite **en ordre applicatif**. On peut également utiliser **une autre méthode d'évaluation** pour l'expression foo. Cette méthode consiste à substituer les arguments non évalués, elle commence alors par une série d' *expansions*

$$\begin{aligned}
\text{foo}\,x &= \cdot[\lambda] \quad \text{somme-carré}\,() + 1, x + 2 \quad ()\\
&= \text{somme-carré}\,() + 2, \quad +\\
&= \cdot[\lambda](,)\ \text{carré}\,x() + \text{carré}\,y() \quad () + 2 \quad +\\
&= \text{carré}\,() + \text{carré}\,() \quad +()\\
&= \cdot[\lambda]\ x\ \ xx \quad () + \cdot[\lambda]\ \ xx \quad ()\\
&= () \cdot () + () \cdot ()
\end{aligned}$$

suivie de *réductions par la gauche* [15]

$$\begin{aligned}
\text{foo}\,() &= () + \cdot () + () \cdot ()\\
&= () + () \cdot ()\\
&= \quad + () \cdot ()\\
&= () \cdot ()\\
&= \cdot ()\\
&= \\
&= 41
\end{aligned}$$

[15] Le fait que les réductions se font «par la gauche» est lié au fait que dans la représentation de l'application d'une fonction à ses arguments ont note d'abord la fonction (à gauche). La justification de cette règle de réduction est associée au théorème de *Church-Rosser* dont la démonstration serait sans intérêt ici.

Cette méthode qui consiste à «développer complètement puis réduire par la gauche» s'appelle l'évaluation en **ordre normal**. Le théorème de Church-Rosser établit que, dans la plupart des cas, le résultat de l'évaluation est le même — l'exercice **E-20** introduit un cas de figure où le résultat ne serait pas identique. On constate alors que la mécanisation de l'évaluation en ordre applicatif demande moins de travail à la machine qui l'exécute, c'est donc l'ordre applicatif qui sera mis en œuvre en général. Cependant, on rencontrera quelque fois (très rarement heureusement) des fonctions qui ne peuvent être évaluées qu'en ordre normal[16]. Lorsque la distinction sera nécessaire, nous appellerons **forme standard** une expression qui peut être évaluée en ordre applicatif et **forme spéciale** une expression qui ne peut être évaluée qu'en ordre normal.

2.5 Fonctions d'ordre supérieur

Ce paragraphe est assez difficile mais son importance est considérable et il justifie bien l'effort que sa compréhension va exiger.

Considérons l'ensemble des fonctions «foo» telles que

$$foo_i : DC \rightarrow$$

Il peut parfaitement constituer le domaine d'une fonction «baz» telle que :

$$baz(DC \rightarrow) \rightarrow C'$$

Une fonction telle que «baz» est dite d'ordre supérieur car elle manipule des fonctions. Dans un même ordre d'idée, rien n'interdit d'imaginer une fonction qui produit une autre fonction. En fait, on en connaît déjà une, la fonction «lambda».

> Imaginer des fonctions qui manipulent des fonctions n'a pas été simple. C'est Newton qui en eut le premier l'idée lorsqu'il voulut donner des bases rigoureuses à sa théorie mécanique. Mais il fallut attendre Leibnitz pour en établir une formulation rigoureuse lorsqu'il développa le calcul différentiel.

Considérons, par exemple, la dérivée de la fonction «f» pour la valeur x définie par

$$dérivée(f)(x) = \lim_{\varepsilon \rightarrow 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

Le type de la fonction «f» est

$$f : \mathbf{R} \rightarrow \mathbf{R}$$

tandis que le type de la fonction «dérivée» est

$$dérivée : (\mathbf{R} \rightarrow \mathbf{R}) \rightarrow (\mathbf{R} \rightarrow \mathbf{R})$$

Sa définition peut être mise sous la forme

$$dérivée(f)(x) = \lambda(f,x) \cdot \lim_{\varepsilon \rightarrow 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

Nous avons abstrait en une seule fois les paramètres «f» et x. Nous aurions pu les abstraire

[16] Nous en avons déjà rencontré deux sans nous en rendre compte. Seriez-vous capable de les retrouver ?

l'un après l'autre et écrire

$$\text{dérivationf} = \lambda \cdot \lambda_x \cdot \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

La fonction «dérivation» est telle que

$$\text{dérivation} \mathbb{R} \to \mathbb{R}$$

dont on peut, d'ailleurs, donner une version approchée

$$\text{dérivationApprochée}_f = \lambda_{(,)\varepsilon} \cdot \lambda_x \cdot \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

Les fonctions «dérivation» et «dérivationApprochée» sont des fonctions qui prennent une fonction à un argument en argument et qui rendent une fonction à un argument comme résultat, ce sont des **opérateurs de dérivation** qui rendent la fonction dérivée d'une fonction donnée.

Considérons, par exemple une fonction «g» et sa fonction dérivée «g'» définie à l'aide de la fonction «dérivation»

$$g' = \text{dérivation}(g)$$

alors

$$g'f = \left[\lambda \cdot \lambda_x \cdot \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}\right](g)$$

$$g'x = \lambda \cdot \lim_{\varepsilon \to 0} \frac{g(x+\varepsilon) - g(x)}{\varepsilon}$$

qui est bien une fonction dont la définition est égale à celle de la dérivée de «g». La valeur de la fonction «g'» dérivée de la fonction «g» pour la valeur *a* du paramètre est donc égale à

$$g'(a) = \left[\lambda \cdot \lim_{\varepsilon \to 0} \frac{g(x+\varepsilon) - g(x)}{\varepsilon}\right](a) \quad \lim_{\varepsilon \to 0} \frac{g(a+\varepsilon) - g(a)}{\varepsilon}(a) \text{ dérivée de } g.$$

Ainsi introduite, on pourrait penser que les fonctions d'ordre supérieur sont des curiosités mathématiques. Il n'en n'est rien et on peut en trouver l'usage dans de nombreuses situations de la «vie courante».

Supposons que nous ayons besoin de contracter un emprunt à la banque. Cet emprunt est caractérisé par

C capital emprunté en francs

D durée de l'emprunt en années

T : taux des intérêts en %

Afin de prévoir le poids des remboursements annuels, nous désirons définir une fonction qui rend la valeur d'une annuité[17]. C'est une fonction dont l'argument est le numéro de l'année et le résultat est l'annuité correspondante. Cette fonction dépend bien sûr de la nature

[17] Nous ne considérons pas ici un prêt de type immobilier où on rembourse d'abord les intérêts puis ensuite le capital emprunté.

de l'emprunt. Considérons la fonction «emprunt» suivante

$$\text{emprunt} C D T \lambda(,) \quad , \quad \cdot \lambda a \cdot \frac{C}{D} 1 \left(\right) + \frac{T}{100} a$$

et appliquons-là aux caractéristiques d'un emprunt particulier

$$\text{emprunt}(1000512 = \left[\lambda(,) \quad , \quad \cdot \lambda a \cdot \frac{C}{D} \left(\right) 1 + \frac{T}{100} a \right] (1000512$$

soit

$$\text{emprunt}(1000512 == \lambda \cdot \frac{1000}{5} 1 \left(\right) + \frac{12}{100} a \quad \text{mesAnnuités}$$

ce qui est justement la fonction dont nous avions besoin. On constate que la fonction que nous avons appelé «mesAnnuités» a **capturé** l'environnement de la fonction «emprunt» au moment de son évaluation.

Les fonctions d'ordre supérieur vont jouer un rôle fondamental dans la définition d'une application (constituée d'une fonction), nous ne pourrions pas aller beaucoup plus loin sans elles. En particulier on peut commencer à entrevoir qu'on saura définir des fonctions génératrice d'applications. Dans l'exemple précédent, notre application, la fonction «mesAnnuités», est engendrée par utilisation du générateur d'application, la fonction «emprunt».

2.6 Fonctions «à effets»

Il est clair que pour l'instant, nous ne faisons que définir des moyens pour exprimer des idées. Tant que ces moyens sont destinés aux humains, ils peuvent rester un peu informels et surtout, les résultats et les données s'expriment implicitement. Mais dès que nous allons essayer de mécaniser ces moyens, la nécessité pour certaines fonctions va apparaitre. En effet, si une fonction bien choisie permet de définir les propriétés de la machine qu'il nous faut concevoir, il nous faut, en plus, des fonctions qui permettent à cette machine d'interagir avec son environnement. On utilisera ainsi les deux **fonctions primitives spéciales**

$$\text{lire} : \perp \rightarrow X$$
$$\text{écrire} X \rightarrow \perp$$

La première prélève une donnée de l'»extérieur» et la rend, la seconde emet une donnée vers l'»extérieur».

La fonction «lire» est spéciale en ce sens qu'elle viole le principe de transparence par référence. En effet, elle n'a aucun argument et chaque fois qu'on l'appelle c'est pour obtenir une donnée différente en général.

La fonction «écrire» ne produit aucun résultat susceptible d'intervenir dans une évaluation. Elle n'est utilisée que pour «l'effet» qu'elle produit sur l'environnement. Lorsqu'on utilisera la fonction «écrire» on la séparera des autres définitions par le signe «;».

Par exemple, on peut modifier la définition du prédicat «divise?»

```
divise?(n m) = (n = 0) → écrire("divise") ;
               (n > 0) → écrire("ne divise pas") ;
               sinon → divise?(n, m - n)
```

2.7 Fonctions & équations

Pour l'instant, nous avons toujours pu écrire une *définition opérationnelle* («par cas» en général) des fonctions que nous avons introduites. Ce n'est malheureusement pas toujours possible. Il y a des choses qu'on peut définir sans pour autant savoir les construire. Ce problème est apparu, il y a fort longtemps, aux mathématiciens grecs qui découvrirent des nombres *non calculables*[18].

Considérons les nombres x et y définis respectivement par

(a) x tel que $x^2 + 5x - 6 = 0$

(b) y tel que $y^2 - 2 = 0$

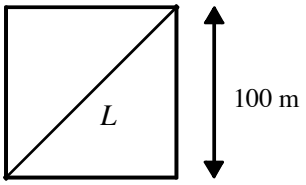
La définition (a) permet de déterminer deux nombres satisfaisant la contrainte de définition

$x_1 = 2$ et $x_2 = -3$

La définition (b) définit parfaitement le nombre *y* mais ne permet pas de le calculer, le nombre *y* est *irrationnel*.

Jusqu'à présent, toutes les fonctions que nous avons rencontrées pouvait être définies par une formule. L'apparition des nombres irrationnels, montre que ce ne sera pas toujours possible. Ce problème n'est pas rare comme le montre la petite histoire suivante.

> «Mérinos a deux fils. Sentant sa fin proche, il décide de rédiger son testament afin de partager son champ carré de 100 mètres de coté entre ses deux fils. Sachant que ses deux fils ne s'entendent pas du tout, il leur tend un piège en imposant que le champ ne puisse être séparé que le long d'une diagonale»



.

> Afin de partager le prix de la clôture en deux parties strictement égales (aucun des deux frère ne veut payer plus que l'autre), ils décident de calculer la longueur de la clôture

$$L = \sqrt{100^2 + 100^2} = 100\sqrt{2}$$

> La longueur de la clôture n'est pas calculable !

[18] uniquement à l'aide des 4 opérations arithmétiques.

Revenons aux nombres irrationnels. C'est Dedekind qui en précisa la nature. Bien sûr, on ne peut pas les calculer, mais on peut les approcher d'aussi près que l'on veut grâce à une *suite convergente*.

Considérons le nombre y défini parl'équation $x^2 = 2$. Ce nombre n'est pas calculable, mais considérons la suite dont le terme courant est

$$x_n = \frac{1}{2}\left(x_{n-1} + \frac{2}{x_{n-1}}\right)$$

On peut montrer qu'elle est convergente, faisons confiance au mathématicien sur ce point et calculons ses quelques premiers termes

termes	définition	valeur
x_0	*arbitraire*	1
x_1	$\frac{1}{2}\left(1 + \frac{2}{1}\right)$	1,5
x_2	$\frac{1}{2}\left(1,5 + \frac{2}{1,5}\right)$	1,4167
x_3	$\frac{1}{2}\left(1,4167 + \frac{2}{1,4167}\right)$	1,41412
....	....	....

Dire que la suite converge revient à dire que

$$\lim_{n \to \infty} x_n = \lim_{n \to \infty} x_{n-1} = x^* \qquad (*)$$

avec

$$x^* = \frac{1}{2}\left(x^* + \frac{2}{x^*}\right)$$

soit

$$(x^*)^2 = 2$$

Ainsi, si on ne peut pas donner une définition calculable d'un nombre irrationnel, on peut toujours lui associer, sous la forme d'une suite convergente, la définition d'un nombre calculable qui s'en rapproche d'aussi près que l'on veut.

En d'autre termes, on vient de s'apercevoir qu'on est capable de définir des nombres sans pour autant être capable de les calculer, ces nombres étant définis comme solution d'une équation. Le calcul d'une approximation de ces nombres revient à résoudre leur équation de définition à l'aide d'une suite convergente. S'il fallait trouver cette suite *à la main* chaque fois qu'un problème non calculable se pose, nous ne serions guère avancés.

Nous allons montrer qu'il est possible de définir une fonction capable de résoudre une équation (presque) quelconque.

Ceci va représenter un progrès majeur qui va nous permettre de construire des abstractions très puissantes et très utiles. Au préalable, il est nécessaire d'étudier une forme particulière d'équations, les «équations de point fixe».

2.7.1 Points fixes d'une fonction

Etant donnée une fonction «f», ses *points fixes x** sont les solutions de l'équation

$$x^* = f(x^*)$$

l'interprétation géométrique du point fixe peut être illustrée par la figure 3, page 31 en con-

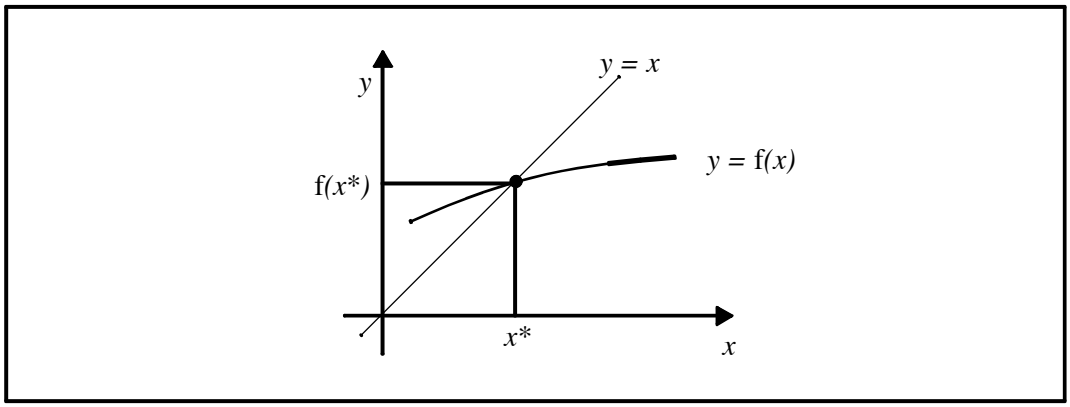


Figure 3 : Point fixe d'une fonction.

sidérant les coordonnées d'un point x^* et $f(x^*)$ point d'intersection entre la courbe $y = f(x)$ et la courbe $y = x$

Comme le nombre x^* n'étant pas forcément calculable, on lui associe la suite

$$x_n = f(x_{n-1})$$

dont les termes se placent facilement en effectuant les constructions de la figure 4, page 31.

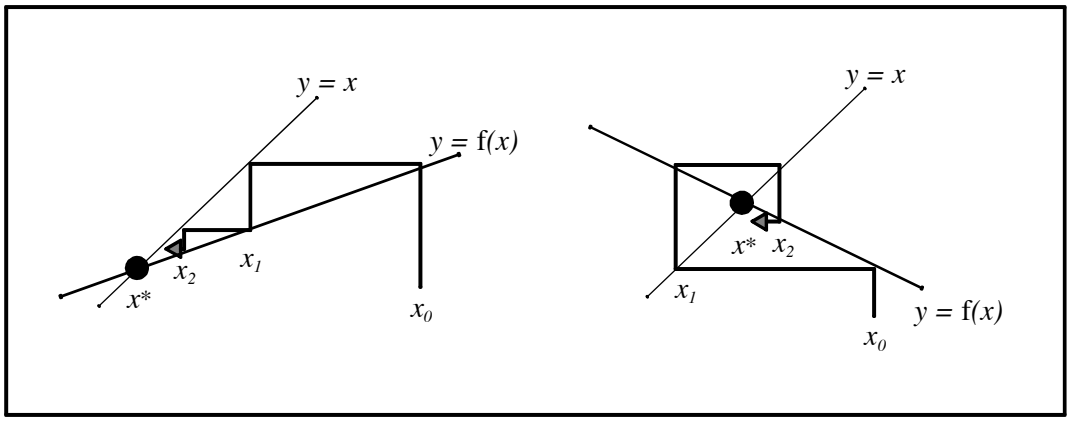


Figure 4 : Convergence d'une suite de point fixe.

Pour que cette suite converge il suffit que pour n suffisamment grand

$$|x_{n+1} - x^*| < |x_n - x^*|$$
$$|f(x_n) - f(x^*)| < |x_n - x^*|$$

Le domaine de convergence peut donc être représentée sur la figure 5, page 32. Il est clair

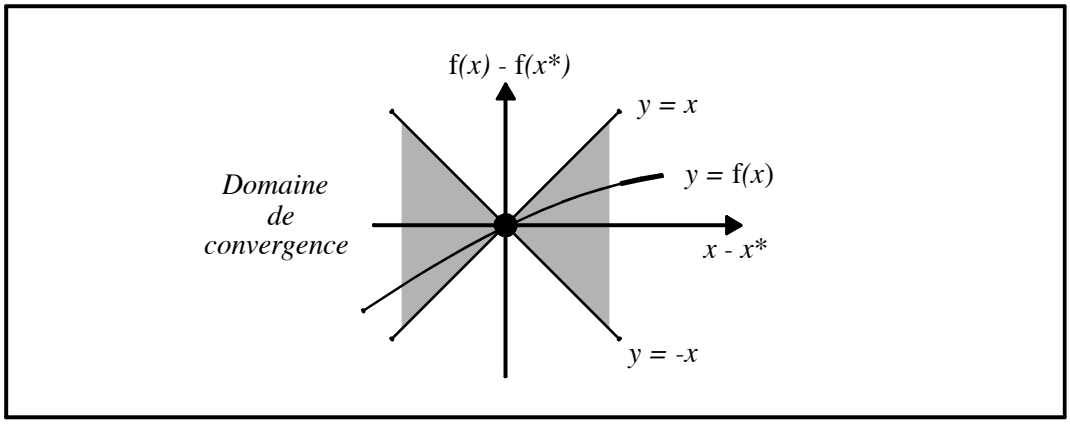


Figure 5 : Domaine de convergence d'une suite de point fixe.

que, lorsque cette suite converge, elle tend vers le point fixe de f(*x*).

Avant d'aller plus loin, voyons comment utiliser une équation de point fixe et considérons l'équation $g(x) = 0$. Trouver une de ses racines (réelles), c'est trouver, par exemple, un des points fixes de la fonction

$$f(x) = a \cdot g(x) + x$$

le coefficient *a* étant déterminé pour que la condition de convergence de la série de point fixe de «f» soit satisfaite. La figure 4, page 31 nous permet de constater que le cas le plus favorable se produit lorsque la fonction $f(x)$ est horizontale, c'est à dire lorsque sa dérivée $f'(x)$ est nulle. Dans ce cas

$$f'(x) = a \cdot g'(x) + 1$$

soit

$$a = -\frac{1}{g'(x)} = -\frac{1}{\text{dérivée de } g(x)}$$

Résoudre l'équation $g(x) = 0$ revient donc chercher le point fixe de

$$f(x) = x - \frac{g(x)}{\text{dérivée de } g(x)}$$

Cette technique de résolution s'appelle la méthode de Newton.

2.7.2 Opérateur de point fixe

La définition de la fonction «pointFixe» qui correspond à la suite de point fixe associée à une fonction «f» est évidemment

$$\text{pointFixe}(f) = x \to f(x) \to x \to \text{pointFixe}(f)(f(x))$$

La définition de cette fonction traduit le fait que la suite de point fixe prend pour valeur:

- la *valeur courante x* de l'estimation de la valeur du point fixe, si la convergence a,

pratiquement, eu lieu,

- la *valeur améliorée* f(x) la convergence n'a pas eu lieu.

2.7.3 Résolution d'une équation quelconque

Pour résoudre l'équation g(x) = 0, il faut d'abord définir la fonction

$$fe(h) = \lambda \cdot \lambda_x \cdot \left(x - \frac{h(x)}{\text{dérivée}(h)(x)} \right)$$

qui engendre la fonction dont le point fixe est la solution de l'équation donnée, puis d'en chercher le point fixe par

pointFixe(x0)(fe(g))

On peut finalement définir la fonction-cosmétique «résoudre»

résoudre(g)(x0) = pointFixe(x0)(fe(g))

2.7.4 Inversion d'une fonction monotone quelconque

Il existe des fonctions monotones — c'est à dire inversibles — telles que la fonction inverse peut être calculée. Malheureusement, il existe des fonctions monotones pour lesquelles l'inverse n'est pas calculable ou est très difficile à définir analytiquement.

Prenons quelques exemples de la vie courante:

fonction $f(x)$	fonction inverse $f^{-1}(x)$
TTCHT = 1186 ,	*HTTTC* = 1186 ,
salaireNet = f(*salaireBrut*)	*difficile à définir*
impôts = f(*revenuImposable*)	*difficile à définir*
....	....

Inverser la fonction «h», c'est trouver pour tout nombre y le nombre x tel que

$$y = h(x)$$

Cette opération peut être représentée par la figure 6, page 33. Inverser «h» revient à résou-

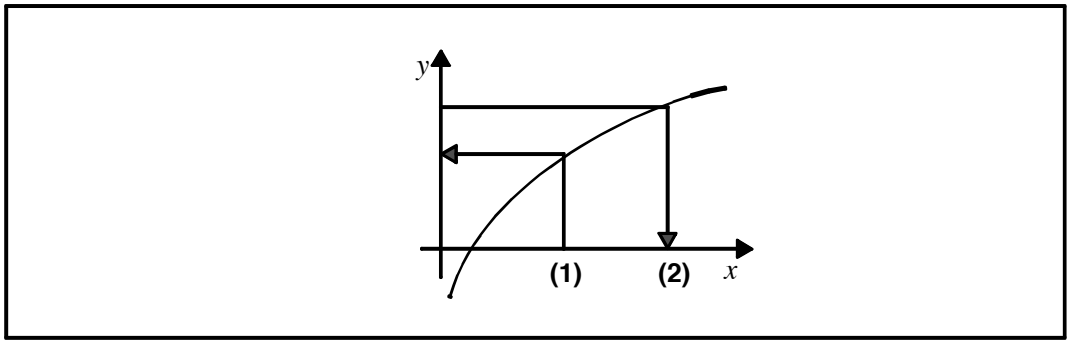


Figure 6 : Utilisation «directe» (1) et «inverse» (2) d'une fonction.

dre l'équation

$$g(x)y == h(x)-0$$

et trouver le point fixe relatif à x, par exemple, de la fonction

$$f(x) = x + \frac{(y - h(x))}{\text{dérivée}(h)(x)}$$

On peut alors définir un générateur «fi» pour la fonction dont on cherche le point fixe

$$fig = \lambda(,) \cdot \lambda x \cdot \left(x + \frac{u - g(x)}{\text{dérivée}(g)(x)}\right)$$

puis de chercher le point fixe par

$$pointFixe(x)(0) fi\ h\ y$$

On peut également définir la fonction-cosmétique «inverser»

$$inverser(y) = \lambda \cdot pointFixe(x)(0) fi\ h\ y$$

qui rend la fonction inverse d'une fonction donnée.

2.7.5 Généralisons un peu (encore ?)

Lorsque nous avons voulu définir la fonction «factorielle» nous avons raisonné par induction et avons obtenu la définition récursive suivante

$$fact(n) = (n = 0) \rightarrow 1 \cdot fact(n-1)$$

soit

$$fact = \lambda n \cdot (n = 0) \rightarrow 1 \cdot fact(n-1)$$

On peut abstraire la variable «fact» du membre de droite puis appliquer la fonction ainsi obtenue à «fact» et obtenir

$$fact = [\lambda f \cdot \lambda n \cdot (n = 0) \rightarrow 1 \cdot n \cdot f(n-1)](fact)$$

Si on pose

$$F = \lambda f \cdot \lambda n \cdot (n = 0) \rightarrow 1 \cdot n \cdot f(n-1)$$

on constate que

$$fact = F(fact)$$

c'est à dire que **la fonction «fact» est le point fixe de la fonction F qu'on appelle une fonctionnelle**.

De la même façon qu'il a été possible de définir un opérateur de point fixe pour les nombres, on peut définir un opérateur de point fixe pour les fonctions (Cf. exercice **E-21** page 45).

2.8 Fonctions & processus

Jusqu'à présent, nous nous sommes consacrés à définir les fonctions dont nous avions besoin et nous ne nous sommes jamais préoccupés du travail qu'allait impliquer leur utilisation. Cette attitude est justifiée dans la plupart des cas de figure. Cependant, de temps à autre, il faudra essayer d'évaluer les conséquences de nos définitions afin, par exemple, d'en

choisir une parmi plusieurs possibles.

Lorsqu'on évalue une expression nous devons effectuer un ensemble d'opérations qu'on appelle un *processus de calcul* qui nous est *totalement indifférent* tant qu'on ne cherche pas à apprécier le travail qu'il représente. Un processus de calcul se déroule en deux phases :

1. Etablir toutes les équations que la définition de la fonction à évaluer permet d'écrire dans le cas qui nous intéresse,
2. résoudre le système d'équations obtenu.

Nous pouvons alors considérer que le travail que représente l'évaluation d'une fonction peut être décrit à partir des trois quantités suivantes:

1. le nombre des **équations** engendrées par l'utilisation de la définition,
2. le nombre des **variables intermédiaires** introduites,
3. le nombre des **opérations élémentaires** à effectuer,

Nous utiliserons le *modèle de substitution* des arguments aux paramètres pour décrire l'évaluation, supposée en ordre applicatif, des expressions que nous allons considérer. Si les définitions non récursives ne posent pas trop de problèmes pour compter les opérations et les variables, les *définitions récursives* engendrent des processus de calcul dont les mécanismes de consommation sont plus difficiles à étudier. C'est donc d'eux dont nous parlerons.

2.8.1 Récursivité linéaire & itération

La première définition récursive que nous avons rencontrée est celle de la fonction factorielle

$$fact(n) = \begin{cases} 1 & \rightarrow 1 \\ n \cdot fact(n-1) & \end{cases}$$

Cette définition est dite *récursive linéaire* car la définition de la fonction ne fait référence qu'une seule fois à la fonction elle-même dans un même sous-domaine.

En utilisant le modèle de substitution, on peut analyser le processus de calcul engendré par l'évaluation de l'expression $fact(6)$

1. Détermination des équations du problème

$$\begin{aligned} fact(6) &= 6 \cdot fact(5) \\ fact(5) &= 5 \cdot fact(4) \\ fact(4) &= 4 \cdot fact(3) \\ fact(3) &= 3 \cdot fact(2) \\ fact(2) &= 2 \cdot fact(1) \\ fact(1) &= 1 \end{aligned}$$

2. Résolution des ces équations

$$\begin{aligned} fact(1) &= 1 \\ fact(2) &= 2 \cdot 1 = 2 \\ fact(3) &= 3 \cdot 2 = 6 \\ fact(4) &= 4 \cdot 6 = 24 \\ fact(5) &= 5 \cdot 24 = 120 \\ fact(6) &= 6 \cdot 120 = 720 \end{aligned}$$

Ce processus entraîne la construction de 6 équations, la définition de **6 variables intermé-**

diaires et la réalisation de **5 multiplications**.

Mais il est possible de définir la fonction «fact» à partir de la recherche d'un invariant. On peut remarquer, en effet, que si on considère la fonction $n!$, son évaluation va nécessiter $n-1$ multiplications et effectuer 1 multiplication diminue de 1 le nombre des multiplications qui restent à effectuer.

L'expression suivante constitue donc un *invariant* de la factorielle

$$\text{invarFact}(f,m,n) \equiv \text{invarFact}(n \cdot f(), m, n-1)$$

ce qui permet de donner une deuxième définition de la fonction factorielle

$$\text{fact}(n) \equiv \lambda() \cdot \begin{bmatrix} \text{soit : invarFact}(f,m) \equiv \lambda() \cdot (m = 1 \rightarrow f(), \text{invarFact}(m \cdot f(), m-1)) \\ \text{dans : invarFact}(1, n, f) \end{bmatrix}$$

La forme que nous venons d'utiliser ne peut pas être celle que nous avions introduite au paragraphe 2.4.5, page 24. En effet, si cela était le cas, nous pourrions l'écrire sous la forme équivalente (?)

$$\text{fact}(n) \equiv \lambda() \cdot [\lambda(f) \cdot f()](\lambda(f,m) \cdot (m = 1 \rightarrow f(), \text{invarFact}(m \cdot f(), m-1)))(1, n, f)$$

qui ne peut pas avoir de sens puisque le nom «invarFact» utilisé ne peut pas être défini.

Nous introduirons alors la forme «soit-rec :» pour dénoter le fait que l'environnement est **étendu de façon récursive** et nous mettrons la définition de la fonction «fact» sous la forme

$$\text{fact}(n) \equiv \lambda() \cdot \begin{bmatrix} \textbf{soit-rec :} \text{invarFact}(f,m) \equiv \lambda() \cdot (m = 1 \rightarrow f(), \text{invarFact}(m \cdot f(), m-1)) \\ \textbf{dans :} \text{invarFact}(1, n, f) \end{bmatrix}$$

Bien qu'il soit possible de donner une forme équivalente à la forme «soit-rec :» en utilisant un opérateur de point fixe (Cf. §2.1 page 45), cette forme est tellement difficile à introduire que nous considérerons cette forme comme **primitive**. Cette forme équivalente exige que les valeurs liées aux noms soient des **fonctions**.

Analysons le processus de calcul engendré par l'évaluation de l'expression $\text{fact}(6)$ mettant en jeu cette autre fonction «fact»

$$\begin{aligned} \text{fact}(6) &\rightarrow \text{invarFact}(1,6,f) \\ &\rightarrow \text{invarFact}(6,5,f) \\ &\rightarrow \text{invarFact}(30,4,f) \\ &\rightarrow \text{invarFact}(120,3,f) \\ &\rightarrow \text{invarFact}(360,2,f) \\ &\rightarrow \text{invarFact}(720,1,f) \\ &\rightarrow 720 \end{aligned}$$

Ce processus entraîne la construction de 7 équations, la réalisation de **5 multiplications** mais il n'a été nécessaire de définir qu' **une seule variable** intermédiaires car les multiplications pouvaient être effectuées au fur et à mesure.

Bien que ces deux processus de calculs produisent le même résultat, ils sont manifestement différents. Le premier processus engendre une expansion suivi d'une contraction. Lors de la phase de développement, il construit *une chaîne d'opérations différées* . Ces opérations seront effectuées successivement lors de la phase de contraction. Le volume de travail qu'il

représente est :

1. le nombre des équations à résoudre: *proportionnel à n*,
2. le nombre des variables intermédiaires à introduire: *proportionnel à n*,
3. le nombre des évaluations effectuées: *proportionnel à n*.

Un tel processus est appelé *processus récursif*.

Le deuxième processus effectue les évaluations nécessaires les unes après les autres immédiatement et sa consommation en ressources de la machine est :

1. le nombre des équations à résoudre: *proportionnel à n*,
2. le nombre des opérations effectuées: *proportionnel à n*.
3. le nombre des variables intermédiaires introduites, **indépendant** *de n*.

Un tel processus est appelé *processus itératif*.

Ainsi, une définition récursive peut engendrer soit un processus récursif, soit un processus itératif.

On peut se demander pourquoi les deux définitions récursives de la factorielle n'engendrent pas le même processus de calcul. A cette question, on peut apporter deux éléments de réponse :

1. La première définition a placé l'opération clé (la multiplication) de telle sorte que celle-ci porte sur la valeur rendue par la fonction appelée. Elle ne peut donc être effectuée qu'**après** que l'appel de la fonction ait rendu cette valeur. Il est alors nécessaire d'introduire une variable intermédiaire puisqu'elle contient la donnée sur laquelle portera l'opération différée.
2. La deuxième a placé l'opération dans les arguments de l'application récursive. Elle est donc effectuée **avant** que l'application de la fonction n'ait lieu. Ainsi, lorsque la fonction appelée rend sa valeur, il n'y a plus aucune opération à effectuer et il n'est pas nécessaire d'introduire une variable intermédiaire. Cette définition est dite *récursive terminale*.

Une définition récursive terminale engendre un processus itératif.

La plupart des langages de programmation utilisés aujourd'hui (Pascal, ADA, C, etc.) ne savent pas identifier une définition récursive terminale et engendrer, dans ce cas, un processus itératif. Ils engendrent donc systématiquement un processus récursif moins efficace. C'est pourquoi, ces langages :

1. sont contraints de fournir au programmeur des formes syntaxiques, nommées *boucles*, pour engendrer un processus itératif,
2. ont contribué à créer le mythe de l'inefficacité de la récursivité.

2.8.2 Récursivité en arbre

La définition d'une fonction est dite *récursive en arbre* lorsque, dans un même sous-domaine, elle fait référence plusieurs fois à la fonction elle-même.

On peut prendre comme exemple la définition du nombre des combinaisons de *m* objets pris

n à *n*

$$C_m^n = \lambda(n,m)\ \cdot <(n,m) \rightarrow 0,$$
$$(n,m) = \rightarrow 1,$$
$$(n = 1) \rightarrow m,$$
$$sinon C(m,n) \rightarrow C(m-1,n-1) + C(m-1,n)$$

Pour retrouver cette définition, il suffit de procéder de la manière suivante :

On retire un objet des m objets parmi lesquels on veut en prélever n. On peut ainsi construire C(m-1,n) combinaisons. Si maintenant on considère les combinaisons de n-1 parmi ces m-1 objets, on construira des combinaisons de n objets en rajoutant systématiquement celui qu'on avait mis de coté.

L'évaluation de l'expression C_5^3 engraîne les évaluations décrites figure 7, page 38.

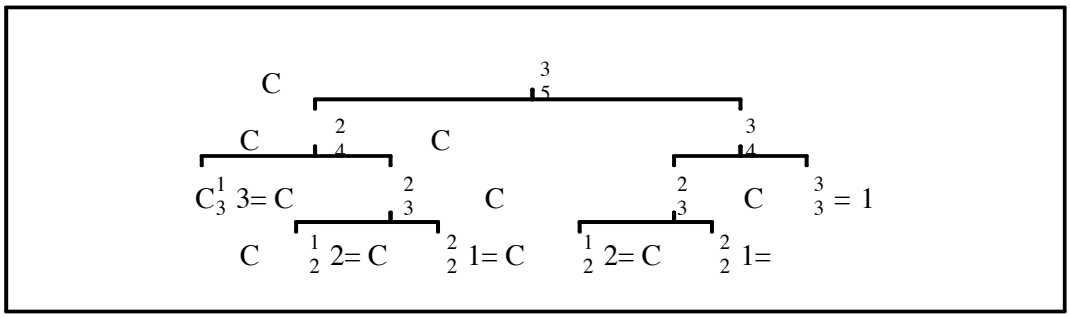


Figure 7 : Récursion en arbre.

En général, le nombre des évaluations effectuées dans un processus récursif en arbre est proportionnel au nombre des noeuds de l'arbre tandis que le nombre des variables intermédiaires à introduire est proportionnel à la profondeur de l'arbre.

Le vrai problème soulevé par la récursivité en arbre réside dans les évaluations multiples d'expressions a priori égales. Ces évaluations superflues peuvent être en nombre considérable et les éviter peut faire gagner énormément de temps. Une technique curieuse peut alors être utilisée. Elle consiste à construire, au fur et à mesure, une table des résultats de la fonction à évaluer afin de ne pas refaire un calcul déjà effectué auparavant. Chaque fois qu'il faut évaluer la fonction, on commence par rechercher dans la table si la valeur n'y figure pas déjà. On verra que cette stratégie nommée *mémoïsation* n'est pas très difficile à mettre en oeuvre, mais elle n'est efficace que s'il est plus rapide de retrouver un résultat dans la table plutôt que de le réévaluer.

2.8.3 Ordre de croissance

Les exemples précédents montrent que les processus de calcul engendrés par l'évaluation d'une fonction peuvent être très différents et le volume de travail requis peut varier considérablement. La notion d'*ordre de croissance* permet de décrire ces différences de comportement en donnant une mesure approximative des grandeurs mesurant l'évolution de la quantité de travail requise par le processus de calcul lorsqu'un paramètre de dimensionnement du problème augmente.

Soit *n* un paramètre mesurant la taille du problème considéré et $R(n)$ la quantité de travail requise par un problème de taille *n*. Dans les exemples précédents, *n* peut être l'argument

de la fonction à évaluer, tandis que R(n) peut être le nombre des équations à définir, le nombres des variables intermédiaires à introduire ou le nombre des opérations à effectuer.

Nous dirons que R(n) est d'ordre de croissance , ce qui s'écrit O(f(n)) ou

$$R(n) = O(f(n)) \quad \text{prononcer : oh de f de n} \qquad (1)$$

s'il existe une constante K, indépendante de n, telle que

$$R(n) \leq K f(n) \qquad (2)$$

pour n suffisamment grand.

Par exemple, le processus récursif engendré par l'évaluation de la première fonction «fact» entraîne la définition d'un nombre d'équations et l'introduction d'un nombre de variables intermédiaires qui croissent comme $O(n)$. En revanche, le processus itératif engendré par l'autre fonction «fact» entraîne la définition d'un nombre d'équations qui croît comme $O(n)$ tandis que le nombre des variables intermédiaires introduites ne croît que comme $O(1)$, c'est à dire est constant.

2.9 Mathématique et informatique - Digression

Ce paragraphe, un peu philosophique, peut être sauté en première lecture. Son seul objectif est d'éclairer les raisons profondes à certaines difficultés que nous allons rencontrer.

Malgré les apparences, mathématique et informatique sont deux activités aussi différentes que peuvent l'être physique et mathématique. Toutes les techniques accomplies utilisent les mathématiques comme outil d'expression et, bien que ce ne soit pas encore tout à fait le cas, l'informatique n'a aucune chance de faire exception.

A l'origine notre connaissance **effective** de la nature ne s'exprimait qu'à travers un ensemble de tours de main et de savoir-faire artisanaux codifiés sous la forme de «méthode pour...[19]». La philosophie traduisait une connaissance «théorique» fondée uniquement sur la réflexion et sans aucune référence expérimentale.

C'est au XVIIème siècle que, sous l'impulsion de Galilée, le lien entre l'étude des phénomènes naturels et les mathématiques est fait. La physique contemporaine était née.

Jusqu'au XIXème les mathématiques ont considéré les fonctions comme des processus de calcul. Une telle vision ne permettant pas d'en donner une définition rigoureuse, il fut nécessaire de considérer les fonctions comme des objets «en soi» qu'on pouvait définir... mais pas forcément associer à une formule. C'est à ce moment là que la différence entre une définition en extension (pas forcément formulable) et la définition en intention (formulable) est apparue.

Depuis lors, pour un mathématicien, tout objet acquiert une sorte d'existence dès lors qu'on a su le définir. Cette façon de voir est incompatible avec l'informatique dont l'objectif est de «vendre des fonctions». Ainsi, on pourrait dire que les mathématiques définissent les fonctions tandis que l'informatique les construit (pour les vendre bien sûr !).

Une conséquence curieuse de ce qui précède est qu'il existe des fonctions qu'on sait définir

[19] L'encyclopédie de Diderot avait pour ambition d'être le recueil de tous les savoir-faire et de tous les tours de main connus.

(spécifier) mais qu'on ne sait pas construire (définir en intention). En fait on va voir qu'on sait définir beaucoup plus de fonctions qu'on ne sait en fabriquer.

Pour simplifier les choses [20], considérons les fonctions qui opèrent sur des nombres et commençons par parler des nombres.

2.9.1 Les nombres de tous les jours

Les nombres «de tous les jours» ne sont que des nombres entiers, même les nombres «à virgule». La raison à cela est qu'il faut les représenter (les écrire, les payer...) et que pour cela nous ne disposons que de **moyens finis** (un nombre fini de symboles représentant un nombre fini d'entités). Il est donc toujours possible d'établir une correspondance entre un nombre de tous les jours et un nombre entier (pensez aux centimes, au millimètres et plus généralement à toutes les sous-unités qui ont été définies justement pour cela). On peut donc dire qu'un nombre de tous les jours est un entier (éventuellement déguisé).

En conséquence, il y a autant de nombres de tous les jours que de nombres entiers, c'est à dire une infinité dite *dénombrables*.

Les nombres réels sont beaucoup plus nombreux que les nombres entiers car on ne sait pas associer un nombre entier à chaque nombre réel (en bref, on ne sait pas numéroter les nombres réels). En fait on a montré que l'ensemble des nombres réels correspondait élément pour élément à l'ensemble des parties de l'ensemble des nombres entiers ; l'ensemble des nombres réel est dit *non-dénombrable*.

Le plus curieux de l'histoire est qu'un ensemble de regroupements finis (doublets, de triplets...) d'entiers est encore dénombrable. En particulier, l'ensemble des nombres rationnels (doublet de deux entiers) est dénombrable et il n'y a pas plus de nombres rationnels que de nombres entiers (nous reviendrons sur ce point de façon très pragmatique un peu plus loin).

2.9.2 Fonctions opérant sur les nombres de tous les jours

Pour simplifier (toujours en apparence) les choses, considérons les fonctions dont le domaine et le codomaine sont l'ensemble des nombres entiers. L'ensemble des fonctions qu'il est possible de définir en extension sur un tel couple domaine, codomaine est l'ensemble des parties d'un ensemble dénombrable. C'est donc un ensemble non dénombrable. En bref, on peut dire qu'il y a autant de fonctions opérant sur un nombre entier pour donner un nombre entier qu'il y a de nombre réels.

Les fonctions que l'informatique va savoir définir s'expriment nécessairement à l'aide de **moyens finis** (un nombre fini de symboles représentant un nombre fini d'entités). Ces fonctions seront dites *calculables* et sont **dénombrables**.

L'informatique ne sait donc construire qu'un ensemble dénombrable de fonctions parmi l'ensemble non dénombrable des fonctions que les mathématiques sauraient définir.

[20] Au moins en apparence car en fait on ne perd rien en généralité en faisant cette hypothèse.

2.10 Conclusions

Ce chapitre avait pour but de montrer toute la richesse expressive de la notion de fonction. Comme nous avons pu le constater, les fonctions permettent de définir des applications sortant du cadre traditionnel des mathématiques. On peut cependant se demander si cette richesse est suffisante pour fonder la programmation uniquement sur le concept de fonction.

Et bien, les théoriciens ont montré que tous les programmes, quels qu'ils soient, écrits dans n'importe quel langage de programmation, pouvaient être définis par une fonction. C'est ce résultat fondamental qui permet d'espérer que dans un avenir pas trop lointain on pourra développer des programmes comme on calcule un pont, des programmes qui «marchent du premier coup».

2.11 Exercices

E-1
1. Définir la fonction «carré» qui élève un nombre x au carré.
2. Définir la fonction «cube» qui élève un nombre x au cube.
3. Définir la fonction «quatrième» qui élève un nombre x à la puissance quatrième.
4. Définir la fonction «moyenne» qui prend la moyenne de deux nombres x et y.

E-2 Définir une fonction «distance» qui calcule la distance euclidienne de deux points d'un plan définis par leurs coordonnées cartésiennes.

E-3 Définir la fonction «quotient» qui rend le quotient de la division du nombre entier positif m par le nombre entier positif n. Evaluer le travail nécessaire à l'évaluation du quotient de deux nombres.

E-4 Définir la fonction «reste» qui rend le reste de la division du nombre entier positif m par le nombre entier positif n. Evaluer le travail nécessaire à l'évaluation du reste de la division d'un nombre par un autre.

E-5 Définir un prédicat «premiers?» qui rend *vrai* si les deux nombres entiers positifs n et m sont premiers entre eux (ne se divisent pas) et *faux* autrement. Evaluer le travail nécessaire à l'évaluation du prédicat «premiers?».

E-6 Supposons que l'addition et la soustraction n'existent pas. Définir les fonctions «additionner» et «soustraire» qui rendent la somme et la différence de deux nombres entiers positifs n et m à partir de deux fonctions élémentaires à déterminer. Evaluer le travail nécessaire à l'évaluation de la somme et de la différence de deux nombres.

E-7 Supposons que la multiplication n'existe pas non plus. Définir la fonction «multiplier» qui rend le produit de deux nombres entiers positifs n et m à partir de l'addition. Evaluer le travail nécessaire à l'évaluation du produit et du quotient de deux nombres.

E-8 Définir une fonction «puissance» qui élève un nombre m à une puissance entière positive n.
1. Donner une *définition récursive* simple de la fonction «puissance». Déterminer le nombre des équations engendrées par une telle définition.
2. Montrer que si n est pair, la définition précédente peut être adaptée de façon à minimiser le nombre des équations construites
3. Montrer que si n est impair, la définition précédente peut être adaptée de façon à mini-

miser le nombre des équations construites.

4. Combiner les deux définitions précédentes en une seule pour obtenir une version optimisée de la fonction «puissance».
5. Evaluer le travail nécessaire à la détermination de la puissance entière d'un nombre selon les différentes versions de la définition de la fonction «puissance».

E-9 On se propose de définir le PGCD de deux nombres entiers positifs. Le PGCD (Plus Grand Commun Diviseur) de deux nombres *a* et *b* est le plus grand nombre qui les divise à la fois tous les deux.

1. Montrer que, étant donné deux nombres *a* et *b*, tout diviseur de *a* et *b* divise aussi soit *a-b*, soit *b-a*.
2. En déduire une définition récursive pour la fonction «pgcd».
3. En déduire une définition du PPCM (Plus Petit Commun Multiple) de deux nombres entiers positifs.
4. Evaluer le travail nécessaire à la détermination du PGCD de deux nombres entiers.

E-10 Définir la fonction «racine» qui calcule la valeur par défaut (valeur entière juste inférieure) de la racine carré d'un nombre entier positif. On utilisera le fait que la racine carré d'un carré parfait est égale au nombre des nombres entiers impairs dont il est la somme.

Par exemple : 25 = 1 + 3 + 5 + 7 + 9

E-11 Définir, à l'aide d'un invariant, une fonction qui rend une valeur approchée de π à partir de l'expression

$$\frac{\pi^2}{6} = 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \frac{1}{25} + \cdots$$

On s'arrêtera lorsque le terme courant n'apporte pas une modification supérieure à un seuil fixé. On s'apercevra que, malheureusement, la précision du résultat n'est pas directement liée à ce seuil. Cette formule a été découverte par le mathématicien suisse Leonhard Euler.

E-12 Définir, à l'aide d'un invariant, une fonction qui rend une valeur approchée de π à partir de l'expression

$$\frac{\pi}{4} = \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \frac{8}{7} \cdot \cdots$$

On s'arrêtera lorsque le terme courant n'apporte pas une modification supérieure à un seuil fixé. Cette formule a été découverte par le grammairien anglais John Wallis en 1655.

E-13 Définir une fonction qui rend une valeur approchée de π à partir de l'expression

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots$$

On s'arrêtera lorsque le terme courant n'apporte pas une modification supérieure à un seuil fixé.

> **Nota:** *cet exercice est plus facile si on remarque qu'il est intéressant d'introduire deux invariants en récurrence croisée. Ces deux invariants peuvent, ensuite, être réunis sous la forme d'un seul.*

Cette formule a été découverte par l'astronome écossais James Grégory en 1671 (certains auteurs attribuent cette formule à Liebnitz)

E-14 Définir la fonction «monnaie» qui détermine la meilleure façon de construire une somme donnée s à partir d'un stock, supposé illimité, de pièces de 20F, de 10F, de 5F, de 2F et de 1F.

Nota: *On supposera qu'il existe une fonction «* afficher(x)*». Pensez à chercher un invariant du problème.*

E-15 Traiter l'exercice précédent en supposant maintenant que le stock de pièces est limité à m_{20} pièces de 20F, m_{10} pièces de 10F, m_5 pièces de 5F, m_2 pièces de 2F et m_1 pièces de 1F.

E-16 Pour résoudre l'équation f(x) = 0 dans l'intervalle [a,b], plutôt que d'utiliser une méthode de point fixe, on opère par approximations successives en procédant de la façon suivante

- Si f(a) et f(b) sont de même signe, il n'y a pas de solution dans cet intervalle.
- Si f(a) et f(b) ne sont pas de même signe, on peut diviser l'intervalle [a,b] en deux parties égales. Si f(a) et f((a+b)/2) sont de même signe, l'intervalle de recherche devient [(a+b)/2,b], sinon, il devient [a,(a+b)/2].

Lorsque la taille de l'intervalle de recherche est inférieure à la précision ε désirée, la recherche est stoppée et le milieu de l'intervalle de recherche constitue l'approximation de la racine de l'équation donnée.

Définir la fonction «résoudre(f,a,b,ε)» qui rend, si elle existe, la valeur approchée, à ε près, de la racine de l'équation f(x) = 0 dans l'intervalle [a,b].

E-17 De nombreux problèmes de recherche opérationnelle se ramènent à la détermination du maximum, dans un intervalle donné [a,b], d'une fonction unimodale (ayant un seul maximum) à un paramètre.

1. Une méthode très simple (et presque la plus efficace) est une recherche par dichotomie. Celle-ci consiste à réduire l'intervalle de recherche de moitié à chaque essai. Définir une fonction maximum(f,a,b,ε) qui rend la valeur du maximum d'une fonction «f» donnée dans un intervalle [a,b] donné à la précision ε. Evaluer le travail nécessaire à la localisation d'un maximum à une précision égale à 1/1000 de l'intervalle de recherche initial.

   Nota: *pensez à localiser le maximum en cherchant une quantité évaluée au milieu de l'intervalle de recherche.*

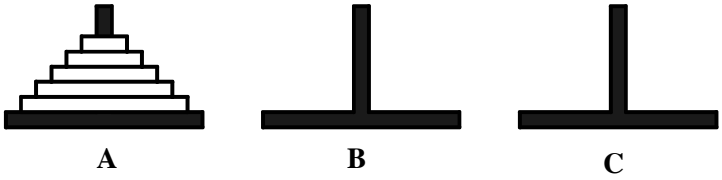
2. On aurait pu aussi utiliser une méthode de point fixe en remarquant que le maximum d'une fonction est caractérisé par le fait que sa dérivée y est nulle. Déterminer la fonction «f» dont il faut déterminer le point fixe pour trouver le maximum de la fonction «g» donnée.

E-18 Une vieille légende Vietnamienne [21] raconte que les moines d'un couvent se relaient depuis fort longtemps pour résoudre le problème suivant :

Une pile de 64 disques placés sur une tige A doit être transférée sur une tige C en s'aidant d'une tige B intermédiaire. Ces disques, étant de plus en plus petits, forment une pile conique. La légende veut que les disques sont déplacés les uns après les autres de telle sorte qu'on ne place jamais un disque sur un disque plus petit que lui.

[21] jolie, mais fausse en fait. Cette charmante histoire a été imaginée en 1833 par le mathématicien français Edouard Lucas.

Au départ, la situation est la suivante



1. Définir une fonction «déplacer(*x*,*y*,*z*,*n*)» qui déplace une pile constituée de *n* disques placée sur la tige *x* vers la tige *z* en s'aidant de la tige *y*. Cette fonction affiche tous les mouvements qu'il est nécessaire d'effectuer sous la forme «A -> B» lorsqu'on déplace le disque situé au sommet de la pile en A pour le placer au sommet de la pile en B.

   Nota: *imaginez que cette fonction existe et décrivez les mouvements à effectuer pour déplacer le disque inférieur de la pile à déplacer. La fonction «déplacer» ne nécessite pas plus de 5 à 6 lignes pour sa définition.*

2. La vieille légende précise, de plus, que la fin du monde aura lieu dès que les moines auront terminé. En supposant que les moines se relaient jours et nuits, qu'ils effectuent un transfert toutes les secondes et qu'ils ne se trompent jamais, déterminer la date de la fin du monde.

E-19 Certains corps de métier utilisent des logiciels de Conception Assistée par ordinateur (CAO) dont les résultats se présentent sous la formes de grands diagrammes, plans ou schémas. La création des documents associés nécessite un périphérique informatique spécial: une *machine à dessiner* («plotter» en anglo-américain).

Cette machine est, en général, constituée d'un grand plan de travail devant lequel se déplace une plume dont les mouvements sont pilotés par deux moteurs lui permettant de se déplacer selon deux axes que nous appellerons arbitrairement axe Nord-Sud et axe Est-Ouest.

Ces deux moteurs sont des moteurs dits pas-à-pas car chaque fois qu'on leur envoie une commande, ils n'effectuent qu'un seul pas. Les déplacements de la plume qui permettent d'effectuer le tracé désiré sont représentés par *la succession des déplacements élémentaires* notés: «n» pour Nord, «s» pour Sud, «e» pour Est et «o» pour Ouest (Cf. figure 8, page 44).

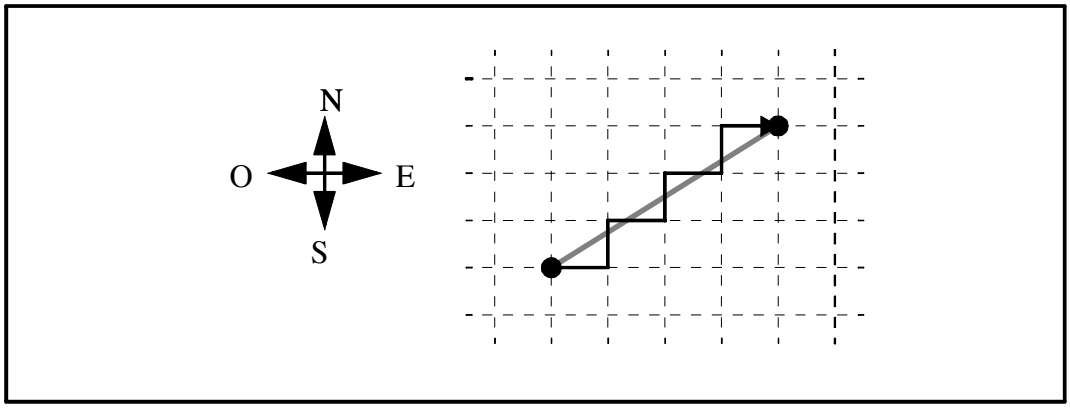


Figure 8 : Tracé en escalier.

Par exemple, le tracé de la figure 6 correspond à la suite des déplacements «e», «n», «e», «n», «e», «n», «e». Bien sûr, la plume dessine en fait de petits escaliers, mais ceux-ci sont

si petits (typiquement 0,1 mm) qu'on ne les voit pratiquement pas à l'oeil nu.

1. Définir la fonction «tracer» qui donne la suite des déplacements qui permettent d'amener la plume du point de coordonnées (x_0, y_0) au point de coordonnées (x_1, y_1) (les coordonnées sont des nombres entiers).

   Nota: *utiliser la fonction «*écrire*» pour envoyer la commande c à la machine à dessiner.*

Il est clair que ce sont les déplacements de la plume qui consomment le plus de temps. Il est donc intéressant de minimiser le nombre de ces déplacements mêmes si, pour cela, on complique la fonction de pilotage.

Comme les deux moteurs sont indépendants, on peut les piloter simultanément ce qui permet de définir 4 déplacements élémentaires supplémentaires: «ne» pour Nord-Est, «se» pour Sud-Est, «so» pour Sud-Ouest et «no» pour Nord-Ouest.

2. Définir la fonction «tracerRapide» qui donne la suite des déplacements qui permettent d'amener la plume du point de coordonnées (x_0, y_0) au point de coordonnées (x_1, y_1)

Dans ce cas, la suite des déplacements élémentaires correspondant à la figure précédente serait : «ne», «ne», «ne», «e».

3. Déterminer le nombre des pas nécessaires pour amener la plume d'un point (x_0, y_0) point (x_1, y_1) en utilisant la fonction «tracer» et en utilisant la fonction «tracerRapide».

E-20 Définir la fonction dont la spécification fut donnée sous la forme du poème suivant

> Trouvez la solution comme le hasard vous conduit,
> Par bonheur à la vérité vous pouvez accéder,
> D'abord procédez à la question,
> Bien qu'aucune vérité n'y soit contenue.
> Une telle fausseté est une si bonne base,
> Que la vérité sera vite trouvée.
> De beaucoup, enlevez beaucoup,
> De trop peu, prenez aussi trop peu.
> A l'excédent, joignez encore le trop peu,
> Et à trop peu, ajoutez trop simplement.
> En croix, multipliez les types contraires,
> Pour que toute vérité, à partir de la fausseté, soit trouvée.
>
> auteur inconnu
> Le Fondement de l'Art (vers 1540)

E-21 **Ce problème est (très ?) difficile**, il consiste à découvrir un opérateur de point fixe pour les fonctions.

1. Considérons la fonction $\lambda x.x$. Quel est le résultat de l'évaluation en ordre normal de l'expression $(\lambda x.x)(\lambda x.x)$?
2. Supposons qu'il existe une fonction Y telle que $Y(\lambda x.x) = (\lambda x.x)(Y(\lambda x.x))$. Quel est le résultat de l'évaluation en ordre normal de l'expression $Y(\lambda x.x)$?
3. Que représente la quantité $Y(\lambda x.x)$ relativement à E.
4. Modifier (légèrement) la définition de la fonction $\lambda x.x$ de telle sorte qu'elle permette la définition d'une fonction pouvant jouer le rôle de Y. Donner votre définition de Y.
5. Montrer que la solution de l'équation (fact 4) (cf. paragraphe 2.7.5, page 34) est égale à $Y(F)$.
6. Evaluer en ordre normal l'expression $(Y(F))(4)$.

3. Les Données

> Nous en arrivons maintenant à l'étape décisive de l'abstraction mathématique: oublions ce que les symboles signifient... [le mathématicien] n'a pas à perdre son temps ; beaucoup d'opérations peuvent être exécutées avec ces symboles sans que nous ayons toujours à l'esprit les objets qu'ils représentent.
>
> Hermann Weyl
> The Mathematical Way of Thinking

Jusqu'à présent, nous avons utilisé des *données primitives* (les nombres) et des *fonctions primitives* (les opérations) pour construire des fonctions. Nous avons ainsi conçu des fonctions qui manipulent des données, des fonctions qui manipulent des fonctions et d'autres qui en produisent. Nous ne nous sommes pas posé la question de savoir d'où ces données et ces fonctions primitives pouvaient bien venir, nous avons simplement fait comme si elles étaient là.

Nous avons acquis la certitude que nous pouvons étendre le nombre des fonctions pratiquement autant que nous le voulons. Par contre, nous n'avons pas encore réussi à étendre le nombre des données primitives.

Vouloir créer un nouveau *type de données* pose trois problèmes:

1. définir l'ensemble des *constantes* ou un *constructeur* pour les éléments de ce type,
2. définir les *opérations* pouvant manipuler les éléments de ce type,
3. en concevoir une *représentation interne* et une *représentation externe*.

Les deux premiers problèmes concerne la définition externe du type de données appelée sa *signature* et le troisième concerne sa réalisation. Essayons de découvrir tout cela à partir de deux exemples.

Nous allons nous placer dans une situation un peu inhabituelle mais qui va nous permettre de comprendre comment on peut partir de presque rien :

1. le mécanisme d'abstraction fonctionnelle, c'est à dire l'opérateur λ,

2. l'application fonctionnelle uniquement, c'est à dire le mécanisme de substitution des arguments d'une fonction à ses paramètres.

Il n'existe donc encore aucune donnée primitive sur laquelle s'appuyer et aucun opérateur primitif à utiliser.

Dans un premier exemple, celui des nombres booléens, nous allons mlontrer comment on peut bâtir, de toutes pièces, un nouveau type de données. Dans un deuxième exemple, celui des nombres rationnels, nous verrons comment combiner des types de données existant pour en construire un nouveau.

Dans un tel contexte, nous ne pourrons utiliser que des fonctions curryfiées puisque nous ne disposons d'aucun mécanisme d'association permettant de construire des n-uplets. Ainsi, lorsque nous serons amené à utiliser des fonctions définie au cours du chapitre précédent, nous en prendrons la version curryfiée.

3.1 Les Nombres booléens

3.1.1 Définition des Constantes booléennes

Pour définir les deux éléments de l'ensemble des nombres booléens que nous appellerons arbitrairement *vrai* et *faux* nous ne disposons que de fonctions curryfiées qui ne peuvent rien faire d'autre que de prendre des arguments et de rendre l'un d'entre eux — ces arguments ne peuvent être que des fonctions puisqu'il n'existe pas encore de données et on ne peut pas les modifier puisqu'il n'existe pas d'opérateurs. Nous devons donc trouver deux fonctions pouvant prendre deux formes différentes.

Il n'y a que deux possibilités

$$\lambda_x \cdot\cdot \lambda_y \quad x$$
$$\lambda_x \cdot\cdot \lambda_y \quad y$$

et nous poserons **arbitrairement** que

$$vraix = \lambda \quad \cdot\cdot \lambda_y \quad x$$
$$fauxx = \lambda \quad \cdot\cdot \lambda_y \quad y$$

Voici donc nos deux constantes [1]. Il nous reste à définir les opérations qui peuvent agir sur elles.

3.1.2 Opérations sur les Nombres booléens

Nous allons nous contenter de définir les opérations de *complémentation*, de *conjonction* et de *disjonction*. Les autres opérations seraient définies sur le même modèle.

Complémentation (opérateur «pas» noté ↼)

L'opération de complémentation pas() *faux* si *x* vaut *vrai* et *vrai* dans le cas contraire.

[1] Nous écririons *vrai* et *faux* en italique car, bien que ce soit des fonctions, nous allons les utiliser comme des données.

Sa définition est donc

$$\text{pas}x = \lambda \cdot x\,faux()\,vrai()$$

En effet

$$\begin{aligned}\text{pas}\,vrai()\,x &= \cdot[\lambda \quad x\,f()\,vrai() \qquad vrai()\\ &= vrai\,faux()\,vrai()\\ &= faux\end{aligned}$$

L'exemple inverse permettrait de vérifier exhaustivement l'égalité de cette définition en compréhension et de la définition en extension qui nous a servi de cahier des charges.

Conjonction (opérateur «et» noté $\wedge$)

La table de définition en extension de la conjonction est la suivante

x	y	$xy\wedge$
faux	*faux*	*faux*
faux	*vrai*	*faux*
vrai	*faux*	*faux*
vrai	*vrai*	*vrai*

la définition de la fonction et est donc

$$\text{et}x = \lambda \cdot\cdot \lambda y \quad xy()\,faux()$$

En effet, prenons un exemple

$$\begin{aligned}vrai\wedge faux \quad &= \cdot\cdot[\lambda x \quad \lambda y \quad xy()\,faux() \quad vrai()\,faux()\\ &= vrai\,faux()\,faux()\\ &= faux\end{aligned}$$

On pourrait ainsi, en traitant les 4 cas de figure, vérifier l'égalité de cette définition et du cahier des charges fourni.

Disjonction (opérateur «ou» noté $\vee$)

La table de définition en extension de la disjonction est la suivante

x	y	$xy\vee$
faux	*faux*	*faux*
faux	*vrai*	*vrai*
vrai	*faux*	*vrai*
vrai	*vrai*	*vrai*

la définition de la fonction ou est donc

$$\text{ou}x = \lambda \cdot\cdot \lambda y \quad x\,vrai()\,y()$$

En effet, prenons un exemple

$$\begin{aligned} vrai\,faux &= \lambda x\ \lambda y\ x\,vrai()y()\ vrai()faux() \\ &= vrai\,vrai()faux() \\ &= vrai \end{aligned}$$

On pourrait ainsi, en traitant les 4 cas de figure, vérifier l'égalité de cette définition et du cahier des charges fourni.

3.1.3 La Fonction «si» (notée $\rightarrow$)

Au chapitre précédent, nous avons introduit une fonction à 3 paramètres appelée «si» que nous noterons ici sous forme curryfiée car nous ne disposons encore d'aucune opération permettant de construire des n-uplets

$$si\,p\,a()b()\ pab \rightarrow ,$$

Lorsque le prédicat *p* prend la valeur *vrai*, cette fonction prend pour valeur la valeur de *a* et prend la valeur de *b* dans le cas contraire.

Avec les booléens que nous venons de construire, il est possible de donner la définition suivante à la fonction «si»

$$si\,p = \lambda a\ \lambda b\ p()b()$$

En effet

$$\begin{aligned} si\,vrai()()b()a &= \\ si\,faux()()b()b &= \end{aligned}$$

Ainsi, comme nous pouvons le constater, la fonction «si» n'est pas primitive comme nous l'avions supposé au premier abord.

3.2 Les Nombres rationnels

De la même manière que nous avons combiné des opérateurs existant pour en construire d'autres, nous pouvons construire de nouveaux *types de données* en *combinant des types de données existant*.

Cette opération correspond à la construction d'un ensemble par *produit cartésien* d'autres ensembles. Considérons l'ensemble des nombres entiers relatifs non nuls , l'ensemble $\mathbb{Z}^*$ des nombres rationnels est l'ensemble produit $\mathbb{Z}\times\mathbb{Z}^*$ muni des opérations adéquates et si $\mathbb{R}$ est l'ensemble des nombres réels, l'ensemble des nombres complexes est construit sur l'ensemble produit $\mathbb{R}\times\mathbb{R}$.

Considérons, par exemple, un *nombre rationnel*. Ce nombre est caractérisé par son *numérateur* et son *dénominateur* qui sont des nombres entiers que nous considérerons comme strictement positifs [2] a priori, il appartient donc à l'ensemble $\mathbb{Q} = \mathbb{N}^+ \cdot \mathbb{N}^+$.

[2] simplement pour éviter les problèmes de signe.

On assimile souvent les nombres rationnels aux fractions et on note

$$q = \frac{numérateur}{dénominateur}$$

Un tel nombre est associé, entre autres, aux opérations suivantes

$$\text{somme : } q_1 + q_2 = \frac{n_1 \cdot d_2 + n_2 \cdot d_1}{d_1 \cdot d_2}$$

$$\text{produit : } q_1 \cdot q_2 = \frac{n_1 \cdot n_2}{d_1 \cdot d_2}$$

$$\text{inverse : } \frac{1}{q} = \frac{d}{n}$$

3.2.1 Définition du Nombre rationnel

Construire un nombre rationnel appartenant à l'ensemble **Q** c'est associer son numérateur et son dénominateur et ce nombre rationnel existe dès qu'on est capable

1. d'associer ses deux composantes pour créer l'entité «nombre rationnel» dans laquelle ces deux composantes sont manipulées en bloc. Nous appellerons «rationnel» la fonction de création d'un nombre rationnel.

$$\text{rationnel} \mathbf{N}^+ \cdot \mathbf{N}^+ \rightarrow \mathbf{Q}$$

2. de sélectionner une des deux composantes pour pouvoir définir les opérations sur les nombres rationnels puisque ces composantes sont définies en termes de numérateur et de dénominateur. Nous appellerons «numérateur» et «dénominateur» les deux fonctions de sélection.

$$\text{numérateur} \mathbf{Q} \rightarrow \mathbf{N}^+$$

$$\text{dénominateur} \mathbf{Q} \rightarrow \mathbf{N}^+$$

Prenons nos désirs pour des réalités [3] et imaginons qu'il existe une fonction de création

$$\text{rationnel}n = \lambda\ ..\lambda_d\ \ldots\ldots()$$

et deux fonctions de sélection

$$\text{numérateur}q = \lambda\ .\ \ldots\ldots()$$
$$\text{dénominateur}q = \lambda\ .\ \ldots\ldots()$$

Ces trois fonctions définissent une *barrière* (Cf. figure 9, page 54) entre le monde des nombres entiers et le monde des nombres rationnels. Nous verrons plus tard comment définir ces fonctions, car pour le moment leur définition n'a pas d'importance.

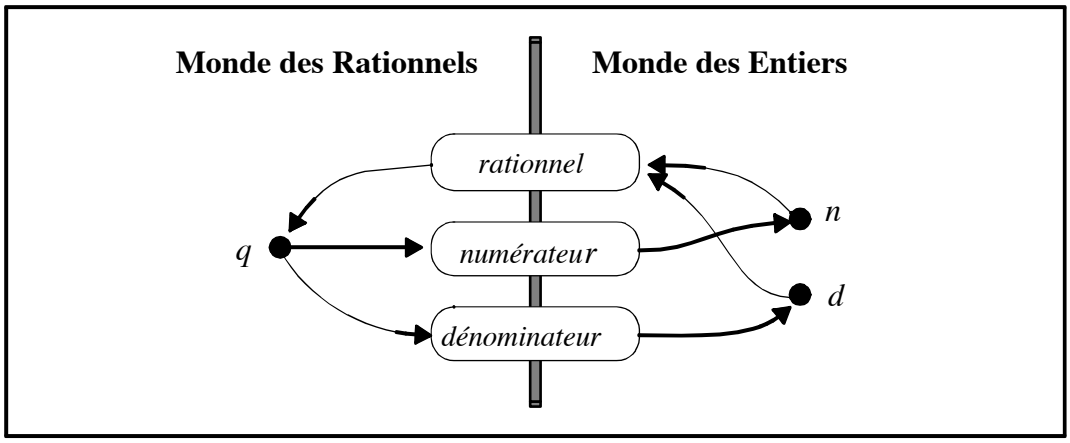


Figure 9 : Le monde des nombres rationnels et le monde des nombres entiers.

3.2.2 Opérations sur les Nombres rationnels

Somme

L'expression de la somme de deux nombres rationnels se déduit immédiatement de sa définition

$$\text{rationnel-somme}q = \lambda\ _1..\lambda_{q_2} \left[\begin{array}{l} \text{soit} : n\ _1 = \text{numérateur}(q)_1 \\ \qquad\quad d_1 = \text{dénominateur}(q)_1 \\ \qquad\quad n_2 = \text{numérateur}(q)_2 \\ \qquad\quad d_2 = \text{dénominateur}(q)_2 \\ \text{dans} : \text{rationnel}\ (n_1 \cdot d_2 + n_2 \cdot d_1)(d_1 \cdot d_2) \end{array}\right]$$

[3] Prendre ses désirs pour des réalités est une méthode de conception extrêmement puissante, nous l'utiliserons souvent.

Produit

L'expression du produit de deux nombres rationnels se déduit également de sa définition

$$\text{rationnel-produit}\, q = \lambda\ {}_1 \cdot\cdot \lambda q_2 \begin{bmatrix} \text{soit} : n_1 = \text{numérateur}(q)_1 \\ d_1 = \text{dénominateur}(q)_1 \\ n_2 = \text{numérateur}(q)_2 \\ d_2 = \text{dénominateur}(q)_2 \\ \text{dans} : \text{rationnel}\ (n_1 \cdot n_2)(d_1 \cdot d_2) \end{bmatrix}$$

Inverse

L'expression de l'inverse d'un nombre rationnel se déduit encore de sa définition

$$\text{rationnel-inverse}\, q = \lambda\ \cdot \begin{bmatrix} \text{soit} : n = \text{numérateur}(q) \\ d = \text{dénominateur}(q) \\ \text{dans} : \text{rationnel}\, d()n() \end{bmatrix}$$

Ainsi, nous constatons que

1. La définition des fonctions «rationnel», «numérateur» et «dénominateur» suffit pour construire l'ensemble des opérations associées aux nombres rationnels. La frontière entre le monde des nombres rationnels et celui des nombres entiers est donc **étanche**.
2. Nous n'avons pas défini de représentation interne pour les nombres rationnels et cela n'a eu manifestement aucune importance.
3. On pourra définir ultérieurement cette représentation interne et même en changer sans avoir à modifier la définition des opérateurs agissant sur les nombres rationnels.

Ces trois remarques sont d'une **importance capitale** car elles montrent qu'il est possible de structurer une application en créant des domaines associés au type des données manipulées. Ces domaines sont caractérisés par la définition de leur frontière constituée de la fonction de création de donnée et des fonctions d'accès aux composantes de la donnée. Un tel domaine est appelé un *type abstrait de données* et il est défini par sa *signature*.

Avant d'aborder la définition d'une représentation pour les nombres rationnels, il est nécessaire d'ouvrir une parenthèse relative à une difficulté de langage courante : les problèmes posée par la *citation*.

3.2.3 Symbole & Citation

Si nous concevons fort bien qu'il est possible de donner un nom aux choses, il semble plus difficile d'utiliser ce nom pour parler des choses. Ce problème n'est pas propre à l'informatique ou aux mathématiques, il est attaché à la notion de langage et les langues naturelles ne sont pas exemptes de problèmes.

En effet, que faut-il répondre à cette invite?
— dis-moi ton nom.

«ton nom» est une réponse aussi concevable que «Alonzo Church» et il faudrait utiliser des guillemets pour lever l'ambiguïté. Ainsi, si le plus souvent nous parlons des choses, il arrive

que nous soyons obligés de parler des choses qui nous permettent de parler des choses [4] et nous utiliserons un mécanisme de *citation* [5].

Jusqu'à présent, lorsque nous avons introduit un nom pour représenter une variable intermédiaire, ce nom était destiné à être évalué. Le résultat de cette évaluation est d'ailleurs la valeur qui a été liée à ce nom.

Nous allons, à présent, être amené à introduire des symboles qui n'ont pas de signification dans le cadre de la définition que nous sommes en train de construire, mais uniquement dans le cadre de leur utilisation. En particulier, ils ne sont pas utilisés pour nommer une variable et ne doivent donc pas être évalués.

Nous conviendrons que les noms **en gras** ne sont que des symboles et que, n'étant liés à aucune valeur, ils ne représentent qu'eux-mêmes. Nous supposerons, de plus, que nous savons comparer deux symboles pour savoir s'ils sont identiques.

3.2.4 Représentations interne du Nombre rationnel

Il est temps maintenant de donner une existence effective aux nombres rationnels en leur définissant une forme interne. Pour bien montrer que la forme de cette représentation n'a pas d'importance, nous allons en imaginer deux.

Représentation par une Fonction

Notre outil de construction étant la fonction, il est naturel, pour nous, d'utiliser une représentation interne sur la base d'une fonction.

Un nombre rationnel sera donc un **objet-fonction** qui associe ses deux composantes, par exemple

$$\text{rationnel}n = \lambda\ \cdots\lambda d\ \ \lambda m\ \ ()m = \textbf{numérateur} \rightarrow n,\ ()m = \textbf{dénominateur} \rightarrow d,\ \textit{sinon} \rightarrow \perp$$

La définition de la fonction constructeur est conçue dans l'optique de la nécessité d'une sélection ultérieure des composantes.

Les fonctions sélecteur se contentent alors d'invoquer la fonction associée au nombre rationnel en l'appliquant à l'argument convenable

$$\text{numérateur}q = \lambda\ \cdot q\textbf{numérateur}()$$
$$\text{dénominateur}q = \lambda\ \cdot q()\textbf{dénominateur}$$

Les deux expressions ci-dessus illustrent bien à quel point la différence entre donnée et fonction est conventionnelle. En effet, q joue dans la même expression tantôt le rôle d'une

---

[4] Pour apprendre à nos enfants à parler français (ou anglais...), nous ne pouvons que parler français (ou anglais...).

[5] L'introduction de la citation dans un langage permet l'apparition de redoutables paradoxes du type de celui-ci:

> La «ligne suivante» est vraie,
> La «ligne précédente» est fausse.

donnée, tantôt le rôle d'une fonction. Cette dualité est encore plus évidente si on écrit

numérateur(q) ≡ q **numérateur**()
dénominateur(q) ≡ q **dénominateur**()

La construction du nombre rationnel a utilisé une technique de programmation très efficace, la *transmission de messages*. Le paramètre *m* de la fonction associée à un nombre rationnel est un *message* à transmettre à ce nombre lorsqu'on lui réclame une de ses composantes. L'utilisation de cette technique n'est qu'ébauchée ici et nous verrons qu'on peut en tirer de multiples avantages.

Représentation par une Donnée primitive

Après avoir utilisé les propriétés des fonctions, nous pouvons utiliser les propriétés des nombres entiers. Nous pouvons, par exemple représenter un nombre rationnel par le nombre entier [6]

$$q = 2^n \cdot 3^d$$

A la place de 2 et 3 on aurait pu prendre n'importe quel couple de nombres entiers premiers entre eux. On peut montrer, bien sûr, que cette représentation est unique et non ambiguë [7].

En effet

$$q_1 = q_2 \Leftrightarrow \begin{cases} n_1 = n_2 \\ d_1 = d_2 \end{cases}$$

La fonction de construction est alors

rationnel n = λ n λ d $2^n \cdot 3^d$

Les fonctions de sélection associées sont un peu plus difficiles à définir que précédemment. Remarquons qu'il est possible de définir les deux invariants suivants

N(q) ≡ n() Nq = ()n / 2 + () 1
D(q) ≡ d() Dq = ()d / 3 + () 1

On peut donc définir le sélecteur de numérateur

numérateur q = λ · [soit-rec : N q = λ n λ d divise ? 2 q () N $\vec{q}$ ()n / 2 + ()n 1 , dans : N q ()0()]

et le sélecteur de dénominateur

dénominateur q = λ · [soit-rec : D = λ q λ d divise ? 3 q () D $\vec{q}$ ()d / 3 + ()d 1 , dans : D q ()0()]

Et voici une autre représentation interne des nombres rationnels qui peut faire

[6] On remarque, en passant, que l'existence d'une telle représentation prouve que l'ensemble des nombres rationnels est dénombrable. Il n'existe donc pas plus de nombres rationnels que de nombres entiers.

[7] Le faire est un exercice intéressant.

parfaitement [8] l'affaire.

3.2.5 Comparaison de deux nombres rationnels

Nous avons vu au chapitre précédent qu'il était très fréquent d'utiliser une forme récursive pour définir de nouvelles fonctions. Cette forme récursive nécessite une définition par cas, un au moins des cas correspondant à un cas trivial non récursif. La définition des différents cas implique l'existence de prédicats de comparaison.

La comparaison de deux nombres rationnels pose un problème délicat dû au fait qu'elle ne doit pas dépendre de la représentation choisie pour les nombre rationnels. La simple comparaison de leurs numérateurs et de leurs dénominateurs ne convient pas. En effet, par définition

$$(q_1 = q_2) \Leftrightarrow \exists m \in \mathbf{N}^+ \mid \begin{cases} (n_1 = mn_2) \wedge (d_1 = md_2) \\ \text{ou} \\ (n_2 = mn_1) \wedge (d_2 = md_1) \end{cases}$$

et on constate qu'un nombre rationnel est le représentant d'une classe d'équivalence. On se trouve alors confronté au choix suivant :

1. soit définir un nombre rationnel en lui conservant la forme qu'on lui a donnée au départ.
2. soit représenter systématiquement un nombre rationnel sous sa forme minimale telle que son numérateur et son dénominateur sont premiers entre eux.

La deuxième approche est très simple à mettre en oeuvre car elle n'implique que la modification du constructeur d'un nombre rationnel. Par contre, elle présente l'inconvénient, relatif, de traduire le nombre rationnel qu'on lui demande de construire, et nous n'en aurons plus qu'une forme *égale mais pas identique*.

Au départ, notre représentation des nombres rationnels a les propriétés suivantes

$$\text{(P1)} \quad n(q) = \text{numérateur}(q) \Leftarrow \text{rationnel}(q) \wedge n(q) = n \quad = \textit{vrai}$$
$$\text{(P2)} \quad d(q) = \text{dénominateur}(q) \Leftarrow \text{rationnel}(q) \wedge n(q) = n \quad = \textit{vrai}$$

etla définition du prédicat *rationnels-égaux?* doit être telle que

$$\text{(P3)} \quad \text{rationnels-égaux?}(q_1, q_2) \Leftarrow \text{rationnel}(q_1) \wedge \text{rationnel}(q_2) \wedge \text{numérateur}(q_1) = \text{numérateur}(q_2) \wedge \text{dénominateur}(q_1) = \text{dénominateur}(q_2) \quad , \quad =$$

Les propriétés P1 et P2 sont des propriétés visibles uniquement du côté des nombres entiers tandis que la propriété P3 n'est visible que du côté des nombres rationnels. Respecter P3, c'est construire un ensemble cohérent de nombres rationnels, vouloir respecter P1, P2 et P3, c'est vouloir, en plus [9], qu'un nombre rationnel soit une boite à deux cases pour y stocker des nombres entiers.

En ce qui nous concerne nous nous contenterons de respecter P3 et allons modifier le constructeur «rationnel». Pour cela nous allons légèrement adapter notre définition d'un nombre

[8] Au moins sur le plan des principes. Il est clair que cette représentation est peu efficace.

[9] Cette confusion des genres conduit souvent à des erreurs (bugs) difficiles à détecter et dont les conséquences sont très difficiles à évaluer.

rationnel en posant

$$q = \frac{n / \text{pgcd}(n)(d)}{d / \text{pgcd}(n)(d)}$$

ce qui conduit à une nouvelle fonction de création

$$\begin{aligned} \text{rationnel}\, n = \lambda\ \cdot\cdot\lambda\, d\ \ \lambda\, m\ \ (m = \textbf{numérateur} &\rightarrow n / \text{pgcd}(n)(d)\ , \\ (m = \textbf{dénominateur} &\rightarrow d / \text{pgcd}(n)(d)\ , \\ \textit{sinon} &\rightarrow \perp \end{aligned}$$

ou bien

$$\text{rationnel}\, n = \lambda\ \cdot\cdot\lambda\, d\ \ 2^{n / \text{pgcd}\, n\, (d)} \cdot 3^{d / \text{pgcd}\, n\, (d)}$$

Ce serait une erreur grave de choisir une représentation minimale puis de concevoir les prédicats de comparaison sur cette hypothèse. L'indépendance des opérations vis à vis de la représentation interne ne serait plus respectée et on ne pourrait plus la choisir librement et éventuellement en changer.

Comme on ne veut pas que la définition des prédicats de comparaison dépende de la représentation interne des nombres rationnels, nous allons faire comme si les nombres rationnels n'étaient pas forcément sous forme minimale et nous allons définir les conditions de comparaison en conséquence

$$\begin{aligned} (q_1 = q_2 &\Leftrightarrow \cdot(n_1\ \ d_2 = \cdot(n_2\ \ d_1 \\ <(q_1\ \ q_2 &\Leftrightarrow \cdot(n_1\ \ d_2 < \cdot(n_2\ \ d_1 \\ >(q_1\ \ q_2 &\Leftrightarrow \cdot(n_1\ \ d_2 > \cdot(n_2\ \ d_1 \end{aligned}$$

Les prédicats de comparaison associés sont alors,

- pour l'égalité

$$\text{rationnels-égaux?}\, q = \lambda\ _1\ \cdot\cdot\lambda\, q_2 \left[\begin{array}{l} \text{soit} : n\ _1 = \text{numérateur}(q)\ _1 \\ \qquad\ d_1 = \text{dénominateur}(q)\ _1 \\ \qquad\ n_2 = \text{numérateur}(q)\ _2 \\ \qquad\ d_2 = \text{dénominateur}(q)\ _2 \\ \text{dans} : n \cdot(\ _1\ \ d_2 = \cdot(n_2\ \ d_1 \end{array}\right]$$

- pour la supériorité

$$\text{rationnels-sup?}\, q = \lambda\ _1\ \cdot\cdot\lambda\, q_2 \left[\begin{array}{l} \text{soit} : n\ _1 = \text{numérateur}(q)\ _1 \\ \qquad\ d_1 = \text{dénominateur}(q)\ _1 \\ \qquad\ n_2 = \text{numérateur}(q)\ _2 \\ \qquad\ d_2 = \text{dénominateur}(q)\ _2 \\ \text{dans} : n \cdot(\ _1\ \ d_2 > \cdot(n_2\ \ d_1 \end{array}\right]$$

- et pour l'infériorité

$$\text{rationnels-inf?}\,q = \lambda_1 \cdot\cdot \lambda_{q_2} \left[\begin{array}{rcl} \text{soit :} n_1 & = & \text{numérateur}(q)_1 \\ d_1 & = & \text{dénominateur}(q)_1 \\ n_2 & = & \text{numérateur}(q)_2 \\ d_2 & = & \text{dénominateur}(q)_2 \\ \text{dans :} n \cdot ()_1 \quad d_2 & < & \cdot ()_2 \quad d_1 \end{array}\right]$$

3.3 Barrières d'abstraction

Quoiqu'il en soit, dans les deux cas, on se retrouve dans la situation illustrée figure 10, page 60.

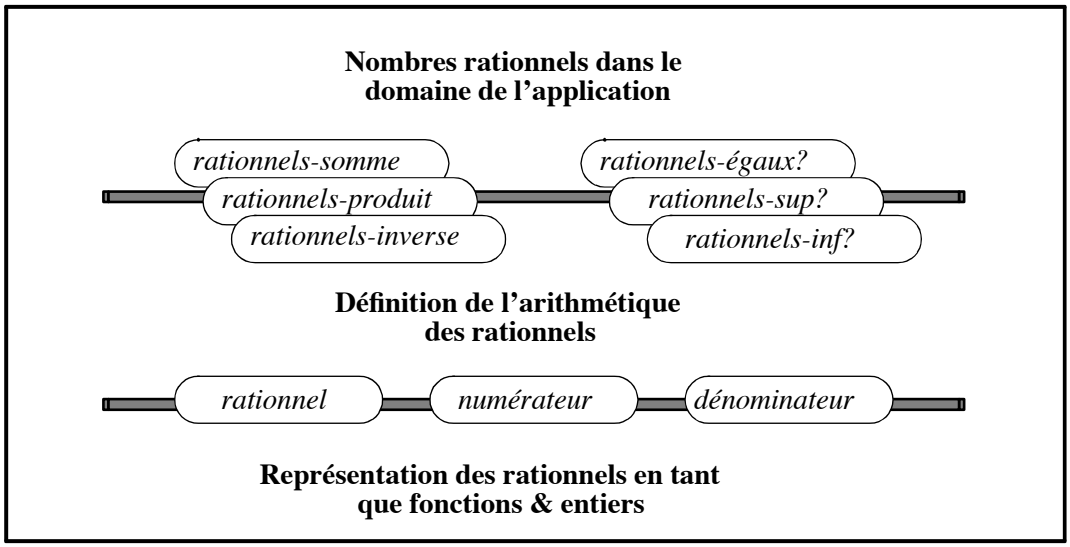


Figure 10 : Barrières d'abstraction correspondant au type abstrait de données : Nombres rationnels.

L'idée sous-jacente à l'abstraction de données est de définir chaque type de données par sa *signature* constituée de :

1. l'ensemble des entités constantes dans ce type ou le constructeur des données du type,
2. l'ensemble des opérations qui permettent d'exprimer toutes les manipulations que peuvent subir les données du type.
3. au moins un prédicat d'égalité afin de permettre une définition récursive des fonctions représentant les opérations licites pour le type.

L'ensemble des fonctions qui permettent de construire ou manipuler les données d'un nouveau type définit son *interface*. Les définitions de ces fonctions, d'accès public, sont rassemblées dans le *paquetage* associé à ce type dit *type abstrait de données*.

La définition de *paquetages* est une méthode de structuration d'application très efficace car:

1. elle permet de scinder l'application en différents domaines indépendants ne commu-

niquant qu'à travers des interfaces restreintes et bien définies.
2. les interfaces peuvent être définies et utilisées avant même que le paquetage lui-même ait été effectivement réalisé.
3. les différents paquetages constituant l'architecture de l'application peuvent être réalisés en parallèle.

L'abstraction par les données correspond à une stratégie de développement très puissante car elle permet de respecter le *principe d'engagement minimal*. Ce principe stipule qu'il est préférable de réaliser en premier les parties d'un travail qui sont les moins susceptibles d'être remises en cause par un fait nouveau [10].

Le savoir-faire d'un chef de projet avisé réside dans son art d'appliquer ce principe de façon clairvoyante.

3.4 Qu'est-ce qu'une donnée?

Avant d'explorer d'autres méthodes de composition et d'abstraction de données, tirons quelques conclusions des deux exemples précédents.

Nous avons tout d'abord créé le type de données *nombres booléen*. Pour cela, nous avons défini :

1. les deux constantes booléennes *vrai* et *faux*, c'est à dire tous les éléments de l'ensemble des données à définir,
2. un ensemble d'opérations comme «et», «ou», «pas» etc.

Puis nous avons créé le type de données *nombre rationnel*. Pour cela, nous avons défini :

1. un constructeur de nombres rationnels «rationnel». En effet, comme l'ensemble des nombres rationnels est infini, il est impossible de définir individuellement tous les nombres rationnels,
2. un ensemble d'opérations «rationnels-somme», «rationnels-produit» etc,
3. les prédicats de comparaison «rationnels-egaux?», «rationnels-sup?» et «rationnels-inf?».

Ces deux types de données semblent très analogues, ils sont cependant conceptuellement très différents.

Les nombres booléens représentent une *interprétation particulière* d'un type de données primitif, ici la fonction, tandis que le nombre rationnel est construit par *assemblage d'éléments d'un type existant* sans que l'interprétation de ceux-ci ne change, ici les entiers. C'est pourquoi, les nombres booléens ne nécessitent pas de constructeur alors que les nombres rationnels en exigent un.

On constate donc qu'il existe deux méthodes pour créer un nouveau type de données :

1. par *changement de l'interprétation* d'un type existant. Les nombres booléens sont une nouvelle interprétation de certaines fonctions,
2. par *composition* (produit cartésien) de types de données existant. Les nombres rationnels sont construits par composition des nombres entiers (qui ne sont qu'une nouvelle interprétation de certaines fonctions, Cf. **E-23** page 63).

[10] En d'autres termes, il n'est jamais bon de prendre des décisions difficilement réversibles trop tôt.

Nous avons montré que, pour représenter des données, il suffisait de définir un *constructeur* et des *sélecteurs*. Mais, réciproquement, est-ce qu'on peut considérer que toute entité ainsi introduite constitue une donnée?

Nous n'avons pas défini le constructeur de nombre rationnel et les sélecteurs de numérateur et de dénominateur n'importe comment. En particulier, nous avons pris soin de vérifier que le prédicat

rationnels-égaux?(*q*, *q*') = (numérateur *q*) × (dénominateur *q*') = (numérateur *q*') × (dénominateur *q*)

était bien toujours *vrai* dans le cadre des différentes représentations que nous avons choisies. Plus généralement, on dira qu'un type abstrait de données est définie par un ensemble de constructeurs et de sélecteurs qui doivent vérifier des conditions spécifiées.

Nous verrons que cette définition du concept de donnée va permettre de «durcir la programmation» en introduisant dans les paquetages de données des *assertions* associées aux conditions de validité spécifiées. Ces assertions constituent l' *invariant du type abstrait de données*.

Nous verrons un peu plus loin qu'il peut être nécessaire également de définir un *prédicat d'identification* associé à un type de données.

3.5 Exercices

E-22 On se propose de définir l'ensemble fini des 10 premiers nombres entiers et de construire une arithmétique traditionnelle sur cet ensemble.

1. En s'inspirant de ce qui a été fait à propos des nombres booléens, donner une définition pour les 10 nombres entiers compris entre **0** et **9**.
2. Définir le prédicat «zéro?» tel que zéro?(**0**) = *vrai* et zéro?(*n*) = *faux* pour toute autre valeur de *n*.
3. Définir les fonctions «suivant» et «précédent» à partir de la table suivante

n	suivant(*n*)	précédent(*n*)
0	1	⊥
1	2	0
...	...	...
8	9	7
9	⊥	8

4. En déduire une définition des fonctions «somme» et «différence» .
5. En déduire une définition des fonction «produit», «quotient» et «reste».
6. Que se passerait-il si les fonctions «suivant» et «précédent» étaient définies à partir de la table suivante ?

n	suivant(*n*)	précédent(*n*)
0	1	9
1	2	0
...	...	...
8	9	7
9	0	8

En particulier que peut-on déduire de la définition de l'opposé d'un nombre ?

l'opposé de x noté ($-x$) est tel que $(-x) + x + 0 =$

E-23 Cet exercice est difficile mais il constitue un excellent entraînement à la manipulation de λ.

Supposons (suivant Alonzo Church) que les nombres entiers positifs soient définis par les fonctions

$$\mathbf{0} = \lambda f x \cdot x$$
$$\mathbf{1} = \lambda f x \cdot f(x)$$
$$\mathbf{2} = \lambda f x \cdot f(f(x))$$
$$\mathbf{3} = \lambda f x \cdot f(f(f(x)))$$
$$\ldots\ldots\ldots$$

Un nombre entier n est donc représenté par n applications successives d'une fonction «f» à un paramètre x.

1. Définir la fonction «suivant».
2. Définir la fonction «somme».
3. Définir la fonction «produit».
4. Définir le prédicat «zéro?» qui rend *vrai* lorsque son argument n est le nombre **0** (*vrai* étant la constante booléenne que nous avons définie au début de ce chapitre).

E-24 Reprenons l'exercice **E-15** du chapitre *Les Fonctions*. Pour rendre la fonction «monnaie» plus générale, on va utiliser l'ensemble des pièces à utiliser ainsi que la quantité des pièces disponibles pour chaque valeur.

1. Quelle type de données peut-on utiliser pour représenter les pièces disponibles?
2. Quelle type de données peut-on utiliser pour représenter le résultat de la conversion?
3. En utilisant les types de données identifiés aux questions précédentes, redéfinir la fonction «monnaie».

E-25 On se propose de construire un paquetage pour l'arithmétique des nombres inexacts [11] comportant 4 chiffres significatifs à partir de celle des nombres entiers. Pour cela, on va définir un type abstrait de données pour représenter ces nombres à virgule sous la forme d'une *mantisse* et d'un *exposant*.

La mantisse représente la valeur comprise entre 0,1 et 0,9999 qui multipliée par une puissance de 10 égale à l'exposant donne le nombre à représenter.

Par exemple

les nombres	sont représentés par	
	mantisse*	exposant
12,456	1245	-2
0,00045	4500	-7
1234578956	1234	6

* la valeur de la mantisse est tronquée, pas arrondie.

La valeur absolue de la forme interne de la mantisse est toujours comprise entre 1000 et 9999 tandis que la valeur de l'exposant s'ajuste en conséquence. On appellera cette représentation : *forme normalisée*.

1. Définir le type abstrait de données **Réels** par son constructeur «réel». Les valeurs de la mantisse et de l'exposant données correspondent pas forcément à une forme normalisée, mais la représentation interne, elle, doit toujours l'être.
2. Définir les sélecteurs «mantisse» et «exposant».
3. Définir les trois prédicats «égaux?», «supérieur?» et «inférieur?».
3. Définir les fonctions «addition», «soustraction», «multiplication» et «division».

   Nota: *toute mantisse devenant, même temporairement, supérieure à 9999 est considérée comme perdue. On supposera que la fonction quotient sur les entiers existe.*

E-26 Les physiciens manipulent des formules qui mettent en présence des constantes sans dimensions (n'ayant aucune unité) et des grandeurs physiques associées à des unités.

On utilise, en général, des systèmes d'unités cohérents c'est à dire dans lesquels toutes les unités dérivent de 3 unités de base:

- unité de masse notée M,
- unité de longueur notée L,
- unité de temps notée T.

Par composition multiplicative, ces trois unités de base permettent de reconstruire toutes les autres unités. Par exemple

- l'unité de vitesse est dénotée (LT^{-1} : mètres par seconde),
- l'unité d'accélération est dénotée (LT^{-2} : mètres par seconde par seconde),
- l'unité de surface est dénotée (L^2 : mètres carré),

[11] Un nombre est dit «inexact» s'il n'est pas calculable. Les nombres entiers et les nombres rationnels sont calculables, tandis que les nombres irrationnels et plus généralement réels ne le sont pas.

- l'unité de volume est dénotée (L^3 mètres cube).

Les unités se déduisent de l'expression de définition de la grandeur physique. Par exemple

- l'unité de force est dénotée MLT^{-2} la force est définie par la formule de Newton : $force = masse \cdot accélération$
- l'unité de pression est dénotée $ML^{-1}T^{-2}$ la pression est définie comme une force par unité de surface.

Sachant que depuis que nous sommes tout petit, on nous a appris à ne pas mélanger les torchons et les serviettes (c'est à dire n'additionner ou ne soustraire que des grandeurs de mêmes unités), il est important, pour un physicien de vérifier l'homogénéité des formules qu'il utilise.

1. Définir le type de données **Dimensions** qui associe les puissances associées à M, L et T dans la construction de l'unité d'une grandeur physique. Par exemple, une masse serait représentée par le triplet <1,0,0>, une vitesse par le triplet <0,1,-1>. Définir alors le constructeur «unités-physiques».

   Nota: *on ne représente que les unités de la grandeur physique, pas sa valeur.*

2. Définir les sélecteurs «masse», «longueur» et «temps» qui rendent les composantes de l'unité associée à la grandeur physique *g*.
3. Définir le prédicat «compatibles?» rendant *vrai* lorsque les deux grandeurs physiques *g1* et *g2* sont représentées par les mêmes unités.
4. Définir les fonctions «multiplier» et «diviser» qui rendent les unités des grandeurs physiques obtenues par multiplication et par division.
5. Définir les fonctions «additionner» et «soustraire» qui soit rendent les unités des grandeurs physiques obtenues par addition ou soustraction, soit rendent le symbole **unités-incompatibles**.
6. A titre d'exemple, écrire les expressions de définition de la *gpsd* (grandeur physique sans dimensions), *masse*, *longueur*, *temps*, puis *surface*, *vitesse* et *force*. Si vos souvenirs de physique le permettent écrire les expressions de définition de *énergie-potentielle* et *énergie-cinétique*.

4. Les Structures.

Le chapitre précédent a montré comment on pouvait introduire des données «en soi», c'est à dire des données pouvant être interprétées concrètement. Les nombres peuvent représenter des quantités, les booléens peuvent représenter des faits (logique des prédicats), des états physiques (niveaux de tension ou de courant pour les électroniciens) les rationnels peuvent être associés aux fractions et représenter des opérations de découpage et de combinaison. Bref, les données peuvent être considérées comme des «choses en soi».

Nous allons, à présent, nous intéresser à des «choses pour organiser les choses», c'est à dire des choses qui n'existent que dans la mesure où on dispose de choses en soi à organiser. Nous parlerons alors de *structures*. Les structures présentent un caractère d'abstraction évident ce qui va rendre leur manipulation un peu plus délicate.

N'allez pas croire que nous abordons, enfin, des problèmes d'informaticien. La notion de structure est présente partout dans ce qui nous entoure. Pour vous en convaincre essayez de définir ce qu'est une pomme ou une bouteille. Décrire la matière qui constitue ces objets est très insuffisant, il faut, de plus, décrire comment cette matière de pomme ou cette matière de bouteille est organisée. Les propriétés des objets les plus courants sont ainsi décrites par leur structure.

4.1 Structures de données.

Lorsque qu'une association de données n'est définies que par un constructeur et des sélecteurs, on dira qu'elle constitue une *structure de données*. Une structure est beaucoup plus simple à définir qu'un type puisqu'il n'est pas nécessaire de lui trouver un invariant, ni de lui définir des opérations selon un schéma théorique bien établi. On en utilisera donc souvent, choisies parmi les 5 modèles suivants :

1. *Paire (2-uplet),*
2. *Enregistrement (n-uplet)*,

3. *Tableau (n-uplet)*,
4. *Liste*,
5. *Arbre*.

De telles structures peuvent être utilisées pour la représentation interne d'un type de données et toute application un peu sérieuse manipule des données composées par association d'un nombre fixe ou variable de données plus élémentaires.

Par exemple :

1. La mécanique localise les points par le *vecteur* de leurs coordonnées qui sont des *nombres réels*.
2. La géométrie utilise des *ensembles* de *points* définis par leurs *coordonnées* pour définir des polygones complexes.
3. Les mathématiques définissent des nombres de plus en plus riches en associant des nombres d'un niveau d'abstraction inférieur. Nous avons vu les *rationnels* construits par l'association de deux *entiers*, on connaît les *complexes* construits par l'association de deux *réels*.
4. La gestion construit des *enregistrements* permettant de représenter un *article* de stock, un *salarié*, un *bulletin de paye*, etc.
5. La gestion construit également des *ensembles* de *salariés*, d'*articles*, etc.
6. Les statistiques représentent des *populations* par des *multi-ensembles* d'*individus*.
7. etc.

4.2 La Paire (2-uplet).

La **Paire**[1] est la simple association de deux éléments quelconques. Nous l'avons déjà rencontrée lorsque nous avons construit les nombres rationnels (1ère version). Nous verrons que son importance est capitale par la variété et la richesse des usages qu'on pourra en avoir.

Son constructeur, dont le nom «cons» est universellement reconnu, peut être défini par

$$\text{cons}x = \lambda \cdots \lambda_y \ \lambda_b \ bx\overrightarrow{y} \ ,$$

Le premier élément de la paire est traditionnellement[2] appelé son *car*, tandis que le deuxième en est le *cdr*. Les fonctions de sélection correspondantes sont

$$\text{car}p = \lambda \cdot pvrai()$$
$$\text{cdr}p = \lambda \cdot pfaux()$$

La **Paire** est principalement utilisée

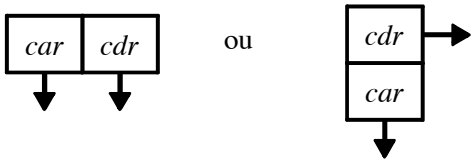
1. pour la construction de doublets,

[1] Pour clarifier le discours, nous désignerons les types et les structures de données que nous définirons par un nom commençant par une majuscule et typographié en **Gras**.

[2] Les noms «cons», «car» et «cdr» (certains prononcent *could-er*) ont une origine historique lointaine. Ce sont les noms des deux instructions machine utilisées pour la première implémentation sur IBM 704 du langage Lisp qui a popularisé les paires et les listes qui en découlent et que nous verrons un peu plus loin.

2. pour la construction d'associations analogues à celle qui nous permettent depuis le début de définir des variables,
3. pour la construction des listes que nous verrons plus loin.

On associe souvent la paire à un petit dessin qui permet de visualiser les constructions qu'on peut faire avec



La flèche utilisée dans ce schéma pointe sur le contenu de la boite car ou de la boite cdr. On appelle donc souvent cette flèche un «pointeur».

Disposant de la paire, on peut redéfinir le constructeur des nombres rationnels

$$\text{rationnel}\,n = \lambda\ \cdot\cdot\lambda\, d\ \ \text{cons}(n)(d)()$$

tandis que les sélecteurs de numérateur et de dénominateur deviennent

$$\text{numérateur}\,q = \lambda\ \cdot\ \text{car}(q)$$
$$\text{dénominateur}\,q = \lambda\ \cdot\ \text{cdr}(q)$$

La paire joue un rôle fondamental car c'est l'association de données la plus simple qu'on puisse concevoir. Ce sera la brique de base de pratiquement toutes les associations de données.

4.3 L'Enregistrement (n-uplet).

Nous appellerons **Enregistrement** la simple association d'un nombre fixe et constant de données de n'importe quel type ou structure, primitif ou non. C'est une simple généralisation de la paire que nous venons de voir.

Par exemple

(**ChurchAlonzo1903WashingtonUSA**)
(**KleeneStephen1909HartfordUSA**)

sont des structures au même titre que

(10205.6318)

Constructeur d'Enregistrements

La fonction de construction d'un enregistrement peut être directement déduite de celle que nous avons utilisée pour construire les paires car elle ne fait aucune hypothèse sur les propriétés des données associées.

Définissons, par exemple, la structure de données **Salarié** utilisée dans le cadre d'une application de gestion du personnel au sein d'une entreprise. Le constructeur d'un salarié peut

être la fonction

$$\text{salarié}n = \lambda\ \cdots\lambda p\ \ \lambda d\ \ \lambda f\ \ \lambda m\ \ \begin{array}{l}()m = \textbf{nom} \rightarrow n\ ,\\ ()m = \textbf{prénom} \rightarrow p\ ,\\ ()m = \textbf{date-naissance} \rightarrow d\ ,\\ ()m = \textbf{fonction} \rightarrow f\ ,\\ \textit{sinon} \rightarrow \bot\end{array}$$

On dit souvent que les symboles de désignation utilisés définissent chacun un *champ* de l'enregistrement. Nous pouvons, ainsi, définir un premier salarié

$$s_1 = \text{salarié}()\textbf{Church}\,\textbf{Alonzo}\,\textbf{1903}\,\textbf{Logicien}$$

soit

$$s_1 = \lambda m \cdot \begin{array}{l}()m = \textbf{nom} \rightarrow \textbf{Church}\ ,\\ ()m = \textbf{prénom} \rightarrow \textbf{Alonzo}\ ,\\ ()m = \textbf{date-naissance} \rightarrow \textbf{1903}\ ,\\ ()m = \textbf{fonction} \rightarrow \textbf{Logicien}\ ,\\ \textit{sinon} \rightarrow \bot\end{array}$$

puis un deuxième

$$s_2 = \text{salarié}()\textbf{Kleene}\,\textbf{Stephen}\,\textbf{1909}\,\textbf{Logicien}$$

ces deux salariés, ainsi que tous ceux que l'on définirait ainsi, sont représentés par des objets-fonction acceptant les messages **nom**, **prénom**, **date-naissance** et **fonction**.

Sélection d'un Champ dans un Enregistrement

Le mécanisme de sélection est, comme précédemment, inclus dans l'objet-fonction représentant la donnée. Dans le cadre de l'exemple précédent, on peut définir les fonctions de sélection suivantes

$$\begin{aligned}\text{nom}s &= \lambda\ \cdot s()\textbf{nom}\\ \text{prénom}s &= \lambda\ \cdot s()\textbf{prénom}\\ \text{date-naissance}s &= \lambda\ \cdot s\textbf{date-naissance}()\\ \text{fonction}s &= \lambda\ \cdot s()\textbf{fonction}\end{aligned}$$

et on peut alors vérifier que

$$\begin{aligned}\text{nom}s_1() &= \textbf{Church}\\ \text{fonction}s_1() &= \textbf{Logicien}\\ \text{nom}s_2() &= \textbf{Kleene}\end{aligned}$$

L'enregistrement est la structure de donnée la plus rustique qu'on puisse concevoir, mais elle est d'une telle généralité qu'on peut l'utiliser pour organiser de très nombreuses applications.

4.4 Le Tableau (n-uplet).

Dans certains cas, on peut ne pas vouloir faire jouer un rôle particulier à certains des champs d'une association de différentes données. Dans ce cas, le plus simple est de leur affecter un simple numéro. Une telle association est appelée **Tableau**.

Le tableau est donc le cas particulier de structure où la dénomination des champs utilise des nombres entiers. On pourra alors utiliser l'arithmétique pour manipuler cette dénomination.

Constructeur de Tableaux

Le constructeur de tableaux peut se déduire immédiatement du constructeur d'enregistrements. Définissons, par exemple, le constructeur [3] pour un tableau de 4 éléments

$$\text{tableau}[4]x = \lambda_{1}\lambda x_2\ \lambda x_3\ \lambda x_4\ \lambda i\ (i = 1 \rightarrow x_1,\ (i = 2 \rightarrow x_2,\ (i = 3 \rightarrow x_3,\ (i = 4 \rightarrow x_4,\ \textit{sinon} \rightarrow \bot$$

puis construisons quelques tableaux de 4 nombres

$$t_1 = \text{tableau}[4](1\ 2\ 3\ 4)$$

$$t_2 = \text{tableau}[4](10\ 20\ 30\ 40)$$

$$t_3 = \text{tableau}[4](100\ 200\ 300\ 400)$$

Ces tableaux sont des objets-fonction acceptant les messages 1, 2, 3 et 4.

Sélection d'un Champ dans un Tableau

Le mécanisme de sélection étant inclus dans le constructeur et les messages acceptés par les tableaux étant des nombres entiers (les numéros de champ), la sélection d'un champ est donc immédiate

$$t_1(1) = 1$$

$$t_2(2) = 20$$

$$t_3(3) = 300$$

$$t_1(6) = \bot$$

Dans le cas particulier des tableaux, le mécanisme de transmission de message se traduit par une formulation tout à fait habituelle et on ne définit pas de fonction de sélection.

Détermination de la Taille d'un Tableau

La forme particulière donnée à la dénomination des champs de tableau permet d'imaginer une fonction universelle rendant la taille d'un tableau quelconque. Pour cela, il suffit de remarquer que tous les éléments d'un tableau ont une valeur et que les numéros de champs sont des entiers consécutifs.

---

[3] Ce constructeur suppose que le numéro de la première composante du tableau est 1. On pourrait considérer, tout aussi bien, que ce numéro est 0.

Dans ces conditions

$$\text{tableau-taille}\,t = \lambda\ \cdot \left[\begin{array}{l}\text{soit-rec :taille}\,n = \lambda\ \cdot ()\,t() = \perp \rightarrow 0 \mid \text{taille}\,n\ +()\ 1 \\ \text{dans :taille}1\ ()\end{array}\right]$$

4.5 Le «Filtrage» et les Fonctions non curryfiées

Si le mécanisme le plus élémentaire ne permet que d'introduire que des fonctions à un seul argument, les premières fonctions que nous avons introduite en avait plusieurs. Pour unifier ces deux points de vue, nous avons introduit la curryfication des fonctions et le paramètre structuré d'une fonction non curryfiée. Cela signifie-t-il qu'il existe un «mécanisme caché» pour structurer les paramètres?

Heureusement, il n'en est rien et ce que nous avons vu va nous permettre de le montrer sur un exemple.

Reprenons la définition que nous avons introduite pour une addition non curryfiée

$$\text{add}_{nc} = \lambda(x,y)\ \cdot\ x+y$$

sa version curryfiée serait

$$\text{add}_{c} = \lambda x \cdot\cdot \lambda y\ \ x+y$$

Nous pouvons définir une fonction «add» à un seul argument en regroupant les deux opérande de l'addition dans une paire et poser

$$\text{add}\,p = \lambda\ \cdot \text{car}\,p + \text{cdr}\,p\ \ ()$$

L'application de cette fonction au deux arguments 3 et 4 s'écrit alors

$$\text{add}(\text{cons}\,3\,4)$$

Nous disposons donc de tous les éléments pour définir des fonctions non curryfiée. Cependant, la lourdeur d'une telle écriture va nous amener à en introduire une forme plus conviviale.

Nous noterons le résultat de la construction d'une paire

$$\text{cons}\,a\,b = (a,b)$$

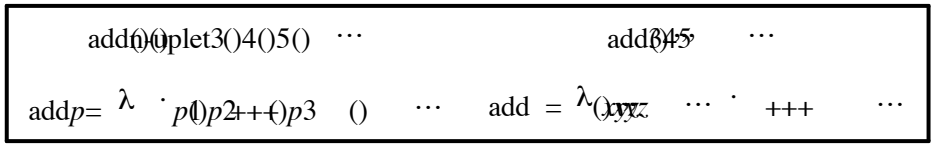
et plus généralement, nous noterons $(a,b,\ldots)$ la construction d'un enregistrement ou d'un tableau.

On utilisera alors les écritures équivalentes suivantes

$\text{add}(\text{cons}\,3\,4)$	$\text{add}\,(3,4)$
$\text{add}\,p = \lambda\ \cdot \text{car}\,p + \text{cdr}\,p\ \ ()$	$\text{add} = \lambda(x,y)\ \cdot\ x+y$

et plus généralement, si on suppose que le constructeur d'un tableau est la fonction n-

uplet.



Ce style d'écriture qui permet de définir des fonctions non curryfiées s'appelle *accès aux paramètres d'une fonction par filtrage* . Ce style d'écriture est tellement commode que nous l'étendrons chaque fois que nous en auront l'occasion.

Ce style est caractérisé par le fait qu'on utilise la même forme syntaxique pour exprimer le constructeur d'une structure (en argument d'une fonction) que pour exprimer l'utilisation des sélecteur de cette même structure (en paramètre d'une fonction).

4.6 La Liste & le Chaînage des Paires

La **Liste** est l'association d'un nombre variable de données de n'importe quel type ou structure, primitif ou non. C'est une utilisation particulière de la **Paire** et la liste ne constitue pas, à proprement parler, une structure, mais plutôt une discipline d'utilisation de la paire.

Par convention: une liste est constituée uniquement de paires dont le *car*[4] est la donnée à placer dans la liste et dont le *cdr* est la suite de la liste. La *cdr* de la dernière paire est $\perp$. Une telle construction constitue une *liste bien formée* .

Les définitions mettant en jeu des listes font un usage intensif des fonctions «car», «cdr» et des compositions de ces deux fonctions. On a donc introduit des abréviations associées à ces compositions.

Dans une composition de «car» et cdr», chaque occurence de «car» est notée «a» et chaque occurence de «cdr» est notée «d». Le résultat de la composition est encadrée par «c» et «r».

Par exemple

$$\begin{aligned} \text{car}(\text{cdr}\,L) &\equiv \\ \text{car}(\text{cdr}(\text{cdr}\,L)) &\equiv \\ \text{car}(\text{cdr}(\text{cdr}(\text{cdr}\,L))) &\equiv \end{aligned}$$

Par exemple, la liste bien formée *l* des nombres 3, 5, 6, et 9 est construite uniquement en utilisant la fonction «cons»

$$L = \text{cons}(3, \text{cons}(5, \text{cons}(6, \text{cons}(9 \quad , \perp$$

[4] Le choix de *car* comme récipient de la donnée correspond à une facilité de réalisation de la première implémentation du langage Lisp.

Cette liste L est telle que

$$\begin{aligned}
car(L) &= 3 \\
cdr(L) &= cons(5, cons(6, cons(9, (), ()), ()), \perp) \\
cadr(L) &= 5 \\
cddr(L) &= cons(6, cons(9, (), ()), \perp) \\
caddr(L) &= 6 \\
cdddr(L) &= cons(9, (), \perp) \\
cadddr(L) &= 9 \\
cddddr(L) &= \perp
\end{aligned}$$

On appelle, en général, **tête** de la liste l la quantité car(l) et **queue** de la liste l la quantité cdr(l). On remarque alors que la queue d'une liste est encore une liste ce qui permet de donner une définition récursive de la liste

$$\begin{aligned}
&liste <> ::= tête <> queue <> \\
&queue <> ::= liste <> \mid \perp \\
&tête <> ::= donnée\ quelconque
\end{aligned}$$

Cette façon de noter la définition des structure récursive est appelée BNF (Backus Naur Form). Dans cette écriture le symbole ::= signifie «est de la forme» et le signe | signifie «ou».

Afin de soutenir nos raisonnements, on utilisera fréquement deux autres représentation d'une liste: une représentation textuelle et une représentation graphique (Cf. figure 11, page 73).

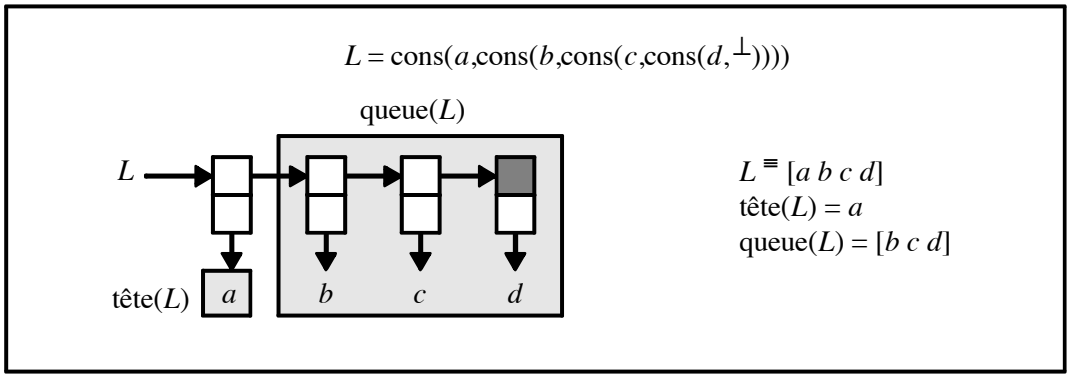


Figure 11 : Différentes représentations d'une liste. Nota: Ne pas confondre une paire en notation pointée et une liste à deux éléments.

Filtrage sur une Liste

Les fonctions définies sur les listes nécessitent très fréquemment d'en séparer la tête de la queue. Nous utiliserons alors une autre représentation de la paire en notant

$$cons(a, b) \equiv a.b \equiv [a|b]$$

On utilisera aussi les simplifications suivantes

$$\begin{aligned}
[a|[b|[c|L]]] &\equiv [a\ b\ c|L] \\
[a|\perp] &\equiv [a] \\
[a|[b|[c|\perp]]] &\equiv [a\ b\ c]
\end{aligned}$$

Détermination de la Taille d'une Liste

La nature récursive de la définition d'une liste permet de définir, par induction, une fonction rendant la taille d'une liste

$$\text{taille-liste} = \lambda[x\cdot L]\cdot\left(\text{vide?}([x\cdot L]) \to 0,\ 1 + \text{taille-liste}(L)\right)$$

Cette définition suppose l'existence du prédicat «vide?» qu'il est facile de définir

$$\text{vide?} = \lambda L\cdot(L = \bot)$$

A titre d'exemple, évaluons l'expression $\text{taille-liste}([abc])$. Cette évaluation entraîne la construction des équations

$$\text{taille-liste}([abc]) = 1 + \text{taille-liste}([bc])$$
$$\text{taille-liste}([bc]) = 1 + \text{taille-liste}([c])$$
$$\text{taille-liste}([c]) = 1 + \text{taille-liste}(\bot)$$
$$\text{taille-liste}(\bot) = 0$$

dont la résolution donne

$$\text{taille-liste}([abc]) = 3$$

Pour définir la fonction taille-liste, nous avons utilisé une technique très utile pour la manipulation des listes : **suivre la queue**. Cette technique va nous permettre de définir de nombreuses opérations sur les listes.

Renversement d'une Liste

Suivre la queue d'une liste entraîne un *ordre implicite* entre les données de la liste. Il peut être utile de renverser cet ordre et pour cela définissons la fonction *renverser*.

Cette définition est (relativement) facile à imaginer si on considère que renverser une liste est une opération pouvant être représentée par le schéma de la figure 12, page 74.

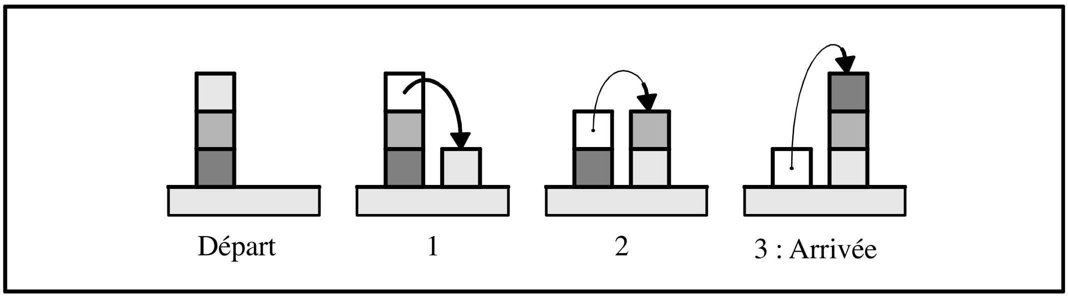


Figure 12 : Renversement d'une liste. On imagine que la liste est une pile de boite qu'on reconstruit à coté.

Tout au long du renversement le nombre des éléments dans la liste de départ ajouté au nombre des éléments de la liste d'arrivée est constant. On peut donc décrire cette opération par un invariant

$$\text{invariant}([x\cdot L_1], L_2) = \text{invariant}(L_1, [x\cdot L_2])$$

En effet, la boite posée au sommet de la pile qui représente la liste $[x\cdot L_1]$ peut être re-

présentée par x et poser une boite sur la pile qui représente la liste peut être décrit par $\cdot[]\cdot x L_2$.

Au départ, on est dans la configuration

$$\text{invariant}(L), \perp$$

et à l'arrivée, dans la configuration

$$\text{invariant}(L), \cdots$$

La définition de la fonction «renverser» est alors

$$\text{renverser} L = \lambda \cdot \left[\begin{array}{l} \text{soit-rec : inv} x L = \text{vide?} x L_1 \cdot \lambda L_2 \cdot \quad ()\, L \quad {}_1 \quad \rightarrow \quad {}_2, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{sinon} \text{inv} \vec{L} \quad ()_1, \cdot[]\cdot x L \quad {}_2 \\ \text{dans : inv} L \quad (), \perp \end{array}\right]$$

Construction d'une Liste ordonnée

Avant de pouvoir renverser l'ordre d'une liste, il faut pouvoir construire une liste ordonnée. Une telle opération rend, à partir d'une liste donnée et d'un élément donné, la liste telle que cet élément se trouve «à sa place». On dira qu'on *insère* l'élément.

Par exemple

$$\text{insère} 7 (2 5 8 \cdot[]\cdot 7 8) = [] 2 5 7 8$$

La place à laquelle doit se retrouver un élément à insérer est déterminée par le prédicat définissant la relation d'ordre désirée.

Définissons, par exemple, une fonction d'insertion pour ordonner des nombres entiers par ordre croissant. Un nombre x est «à sa place» si les deux conditions suivantes sont satisfaites simultanément

$$\begin{array}{rl} \forall x \; () \; \text{suivant-dans-la-liste}(x) & = \textit{vrai} \\ \forall x \; () \; \text{précédent-dans-la-liste}(x) & = \textit{faux} \end{array}$$

La définition de la fonction *insérer*, dans ce cas, est alors

$$\text{insérer} x y L = \lambda (,) \cdot[] \quad \cdot \text{vide?}(y L) \quad \rightarrow [] \;, \\ > (x y) \quad \rightarrow \cdot[]\cdot y \; \text{insérer} x (\vec{L}) \;, \\ \textit{sinon} x \vec{y} L \cdot[]$$

A titre d'exemple, évaluons

$$\begin{array}{rl} \text{insère} 7 (2 5 8 \cdot[]\cdot \text{cons} 2 \text{insérer} 7 5 8) & = () \;, \quad () \;, [] \\ \text{insère} 7 (5 8 \cdot[]\cdot \text{cons} 5 \text{insérer} 7 8) & = () \;, \quad () \;, [] \\ \text{insère} 7 (8 \cdot[]\cdot \text{cons} 7 8) & = () \;, [] \end{array}$$

et

$$\begin{array}{rl} \text{insère} 7 (2 5 8 \cdot[]\cdot \text{cons} 2 \text{cons} 5 \text{cons} 7 8) & = () \;, \quad () \;, \quad () \;, [] \\ & = [] 2 5 7 8 \end{array}$$

Projection d'une Liste

Projeter une liste consiste à construire une autre liste dont tous les éléments sont le résultat de l'application d'une fonction d'arité 1 (à 1 paramètre) à tous les éléments de la liste donnée

$$\text{projeter}(f)[a_1 \cdots a_i] \quad , f = \cdots [f(a_1) \cdots f(a_i)]$$

Par exemple

$$\text{projeter}(2)[3\ 4\ 9\ 16] \quad , \lambda x \cdot x^2 = [9\ 16\ 81\ 256]$$

La définition de la fonction «projeter» est (assez) facile à trouver par induction

$$\text{projeter}\ x\ L = \lambda(f) \cdot [\ ] \quad f \cdot \text{vide?}(x)\ L \quad \to \perp \ , \text{cons}(f(x))(\text{projeter}\ L f)$$

Réduction d'une Liste

Réduire une liste consiste à évaluer l'expression qu'on obtiendrait en appliquant «par la gauche» une fonction d'arité 2 (à 2 paramètres) aux éléments de la liste

$$\text{réduire}(f)[a_0 \cdot a_1 \cdots a_i] = (\cdots (a_0\ f\ a_1)\ f \cdots)\ f\ a_i$$

Par exemple

$$\begin{aligned} \text{réduire}(+)[10\ 20\ 30] &= (10 + 20) + 30 \\ &= 60 \end{aligned}$$

La définition de la fonction «réduire» est facile à trouver par induction

$$\text{réduire}\ f\ x\ L = \lambda(f) \cdot [\ ] \quad \cdot \text{vide?}(x)\ L \quad \to f(x)(\text{réduire}\ f\ L)$$

On peut, à titre d'exemple, évaluer réduire(+)[10 20 30] aux équations

$$\begin{aligned} \text{réduire}(+)[10\ 20\ 30] &= \text{réduire}(+)[20\ 30] \\ \text{réduire}(+)[20\ 30] &= \text{réduire}(+)[30] \\ \text{réduire}(+)[30] &= \text{réduire}(+)[\ ] \\ \text{réduire}(+)[\ ] &= 0 \\ \text{réduire}(+)[30] &= 30 + 0 \\ \text{réduire}(+)[20\ 30] &= 20 + (30 + 0) \\ \text{réduire}(+)[10\ 20\ 30] &= 10 + (20 + (30 + 0)) \\ &= 60 \end{aligned}$$

Lorsque l'opérateur utilisé pour la réduction est associatif, cette opération revient à insérer cet opérateur entre chaque éléments de la liste complétée de l'élément neutre de l'opérateur.

«Produit interne» de deux Listes

Le produit interne [5] n'est pas liée à l'opération de multiplication mais à une façon de combiner deux ensembles pour en construire un troisième. Le produit interne est donc l'opération entre les deux ensembles E, supposés avoir le $F = \{\cdots f_i \cdots\}$

[5] Attention, dans les ouvrages mathématiques français, «produit interne» est l'autre nom du produit scalaire. Les anglo-américains traduisent «produit interne» par «dot-product» ou «inner-product».

même nombre d'éléments, dont la définition est . $EF \cdots = \text{„}\{\} \ , () \cdot e_i \cdot f_i$

Afin de «matérialiser» cette opération, on lui associe une fonction d'arité 2 appliquée à tous les éléments de l'ensemble produit.

Nota: *Cela revient simplement à inclure dans la définition du produit interne la projection qu'on ne manquerait pas d'effectuer sur l'ensemble produit.*

Par exemple [6]

$$\text{prod-int}(\text{)ons } \text{„} \begin{bmatrix} \textbf{jean} \\ \textbf{paul} \\ \textbf{jacques} \end{bmatrix} \begin{bmatrix} 170 \\ 172 \\ 168 \end{bmatrix} = \begin{bmatrix} \cdot[] \ \textbf{jean} \ 170, \\ \cdot[] \ \textbf{paul} \ 172, \\ \cdot[] \ \textbf{jacques} \ 168, \end{bmatrix}$$

Le produit scalaire de deux vecteurs peut être défini comme la réduction par addition du produit interne par multiplication des deux listes qui représentent les deux vecteurs

$$\text{produit-scalaire} v = \lambda(,)_1 \ v_2 \ \cdot \ \text{réduire} + 0 (0) \text{prod-int} \qquad , v_1, v_2$$

Si on suppose que les deux listes sont de même taille, la définition de la fonction de produit interne que nous utiliserons est

$$\text{prod-intf} x = \lambda(,) \cdot []_1 \ L_1 \ \cdot []_2 \ L_2 \ \cdot \ \text{vide?}(\text{)} \cdot []_1 \ L_1 \ \rightarrow \perp \ ,$$
$$\textit{sinon} \ \overrightarrow{x} \cdot []_1 \text{prod}_2\text{-intf} L \qquad (), \ _1, L_2$$

«Produit externe» de deux Listes

La notion de produit externe [7] n'est pas non plus liée à la notion de multiplication mais à la façon de combiner deux ensembles pour en construire un troisième nommée produit cartésien. Le produit externe est donc l'opération entre les deux ensembles $E = \text{„}\{\} \cdot e_i \cdots$ et de $F = \text{„}\{\} \cdot f_j$ et dont la définition est . Dans le cas du produit externe, les deux ensembles n'ont pas nécessairement le même nombre d'éléments. $EF \cdots = \text{„}\{\} \ , () \cdot e_i \cdot f_j \quad | \forall ij$

Afin de «matérialiser» cette opération, on lui associe, comme dans le cas du produit interne, une fonction d'arité 2 appliquée à tous les éléments de l'ensemble produit.

Par exemple

$$\text{prod-ext}(\text{)ons } \text{„} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} \cdot[]_1 \ a & \cdot[]_1 \ b & \cdot[]_1 \ c & \cdot[]_1 \ d \\ \cdot[]_2 \ a & \cdot[]_2 \ b & \cdot[]_2 \ c & \cdot[]_2 \ d \\ \cdot[]_3 \ a & \cdot[]_3 \ b & \cdot[]_3 \ c & \cdot[]_3 \ d \end{bmatrix}$$

Pour trouver la définition du produit externe, il est important de faire les remarques suivantes :

1. le résultat est une liste de listes.
2. chaque sous-liste du résultat est le résultat d'une projection sur la deuxième liste.

---

[6] Pour clarifier l'écriture les listes sont présentées verticalement.

[7] Attention, dans les ouvrages mathématiques français, «produit externe» est l'autre nom du produit vectoriel. Les anglo-américains traduisent «produit externe» par «cross-product» ou «outer-product».

3. si $f = \lambda x.\lambda y.\ldots$ est la fonction utilisée pour définir le produit externe, la fonction utilisée pour la projection est

$$g = \lambda x.\lambda y.\ldots \quad x \cdots y \cdots \quad (l_{1i}) \quad f(x_i, y_i) \quad \lambda \ldots l_{1i} \cdots y \cdots$$

Ces remarques faites, on peut trouver (pas très facilement) la définition de la fonction «prod-ext» sous la forme

$$\text{prod-ext}f L_1 L_2 = \lambda(,)_1, L_2 \cdot \text{projeter}_x(L_1)[\lambda \cdot \text{projeter}_y(L_2)[\lambda \cdot f(x,y), \ldots], L_2], L_1$$

Par exemple

$$\text{prod-ext}(+) \begin{bmatrix}0\\1\\2\\3\end{bmatrix} \begin{bmatrix}0\\1\\2\\3\end{bmatrix} = \begin{bmatrix}0123\\1234\\2345\\3456\end{bmatrix}$$

4.7 Les Nœuds & les Arbres binaires.

Le nœud est une extension peu connue de la paire car son seul intérêt est de permettre la construction des arbres binaires qui eux sont très connus.

4.7.1 Les Nœuds

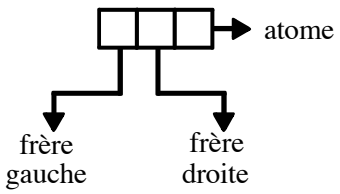
Le constructeur du nœud peut être la fonction

$$\text{nœud}\,g = \lambda \ldots \lambda d\ \lambda a\ \lambda m\ (m = \textbf{gauche} \rightarrow g,\ (m = \textbf{droite} \rightarrow d,\ (m = \textbf{atome} \rightarrow a,\ \textit{sinon} \rightarrow \bot$$

les sélecteurs correspondant sont alors

$$\text{droite}\,n = \lambda \cdot n\textbf{droite}()$$
$$\text{gauche}\,n = \lambda \cdot n\textbf{gauche}()$$
$$\text{atome}\,n = \lambda \cdot n\textbf{atome}()$$

Un nœud est souvent associé à la représentation graphique suivante



En fait, on préfère souvent pour le **Nœud** la définition suivante

$$\text{nœud}\,g\,d\,a = \lambda(,)\ ,\ \cdot[]\ g\ d\ a$$

les sélecteurs correspondant étant

$$\text{droite}_{g \neq d\ a}(\lambda[]) \cdot d$$
$$\text{gauche}_{g \neq d\ a}(\lambda[]) \cdot g$$
$$\text{atome}_{g \neq d\ a}(\lambda[]) \cdot a$$

4.7.2 Les Arbres binaires.

Le nœud est l'élément de base pour la construction des arbres binaires dans lequel il apparaitra sous deux formes. Un tel arbre est la structure hiérarchique représentée par le schéma

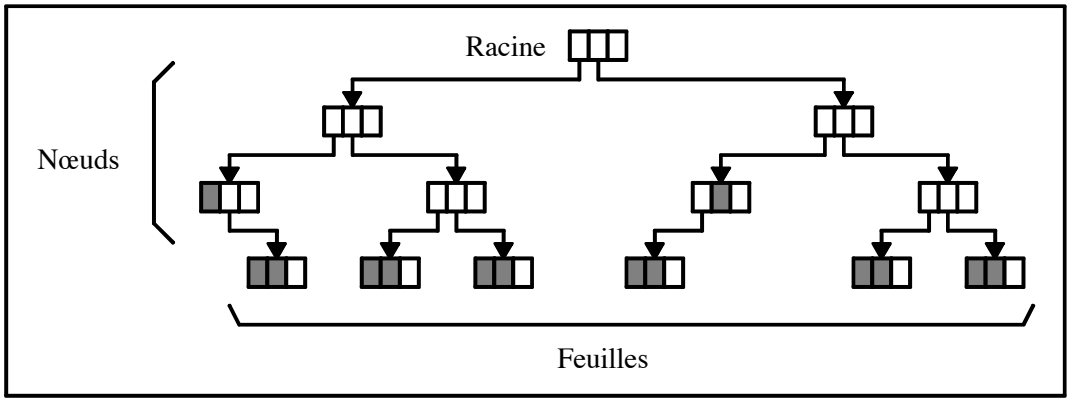


Figure 13 : Structure d'un arbre binaire.

de la figure 13, page 79.

La racine est le nœud auquel s'accroche toute la structure. Les feuilles sont les nœuds qui terminent un «chemin». Les nœuds à proprement parler représentent les embranchements. Cet arbre est dit «binaire» car chaque embranchement ne possède que deux branches. Les atomes qu'il est possible d'accrocher aux différents nœuds vont servir de *décoration*.

Les arbres binaires sont des structures particulièrement efficaces pour représenter des collections d'objets ordonnés. En effet, nous avons déjà rencontré de telles collections lorsque nous avons défini la fonction *insérer* qui permettait de construire une liste ordonnée, mais le temps nécessaire à la recherche d'un élément dans une telle liste est d'ordre $O(n)$. Considérons, par exemple, la collection organisée de la manière décrite figure 14, page 79.

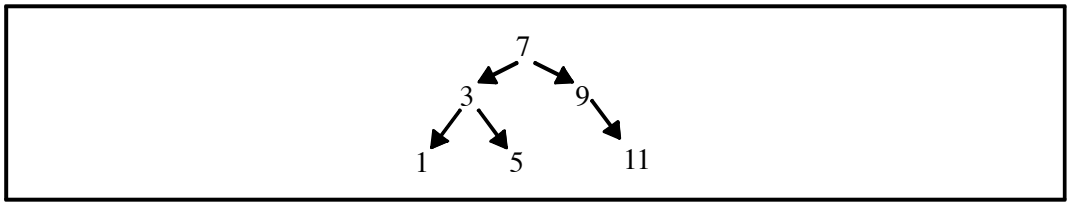


Figure 14 : Collection d'entiers ordonnés.

La recherche d'un élément peut être conduite ainsi :

si l'élément cherché est inférieur à l'atome du nœud, on le recherche dans le sous-

arbre de gauche tandis que s'il est supérieur, on le recherche dans le sous-arbre de droite.

Ainsi, à chaque étape on divise par deux l'ensemble de recherche et le temps de recherche est d'ordre $O(\log n)$ bien sûr, si l'arbre est *équilibré*, c'est à dire harmonieusement réparti entre la droite et la gauche. Cette stratégie de résolution est souvent appelée «diviser pour régner».

Dire qu'un arbre représente des chemins n'est pas fortuit car nous verrons que l'image d'un parcours dans l'arbre est une représentation très commode de l'utilisation qu'on va en avoir.

4.7.3 Recherche dans un Arbre binaire

Définissons le prédicat «appartient?» qui rend vrai si un élément donné se trouve dans un arbre

$$\begin{aligned}\text{appartient?}\,x\,A = \lambda(x,A)\cdot & (\text{arbre-vide?}\,A) \rightarrow \mathit{faux},\\ & (x = \text{atome}\,A) \rightarrow \mathit{vrai},\\ & (x < \text{atome}\,A) \rightarrow \text{appartient?}\,x\,(\text{gauche}\,A),\\ & (x > \text{atome}\,A) \rightarrow \text{appartient?}\,x\,(\text{droite}\,A)\end{aligned}$$

avec

$$\text{arbre-vide?}\,A = \lambda A\cdot A = \perp$$

4.7.4 Construction d'un Arbre binaire

La construction d'un arbre est une opération assez délicate aussi allons-nous en expliquer les détails.

La définition de la fonction «adjoindre» permettant de rajouter un élément x dans un arbre a de telle sorte que ses atomes soient organisés selon le schéma précédent s'obtient par induction.

Si l'arbre est vide, il faut créer un nœud-racine d'atome x sans frère gauche ni frère droite. Si l'arbre n'est pas vide, trois possibilités se présentent :

1. l'élément x est l'atome de la racine de l'arbre a. Il n'est pas nécessaire de construire un arbre différent de a.
2. l'élément x est inférieur à l'atome de la racine de a. Il faut alors construire un nouvel arbre dont la racine a le même atome, dont la branche de droite est identique et dont la branche de gauche est la branche de gauche précédente à laquelle on a adjoint l'élément x.
3. l'élément x est supérieur à l'atome de la racine de a. Il faut alors construire un nouvel arbre dont la racine a le même atome, dont la branche de gauche est identique et dont la branche de droite est la branche de droite précédente à laquelle on a adjoint l'élément x.

La définition de la fonction «adjoindre» est alors

```
                adjoindre=
λ(x,A) . (arbre-vide?(A) → nœud(x,⊥,⊥)
         (x = atome(A) → A,
         (x < atome(A) → nœud(adjoindre(x,gauche(A)),droite(A)),atome(A),
         (x > atome(A) → nœud(gauche(A),adjoindre(x,droite(A))),atome(A))
```

On peut illustrer ce mécanisme en dessinant les arbres obtenus successivement par adjonction de 3, 5, 1, 7 et 9 (Cf. figure 15, page 81).

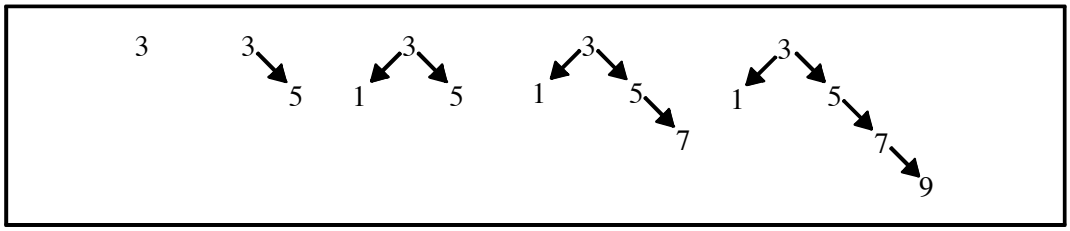


Figure 15 : Construction d'une collection d'entiers ordonnés.

4.7.5 Les Parcours d'un Arbre binaire.

Parcourir un arbre, c'est construire une liste de tous les atomes associés aux différents nœuds. Il existe différentes manière d'effectuer ce parcours:

1. en profondeur par la gauche, ce qui donne une liste des atomes classés par ordre croissant.
2. en profondeur par la droite, ce qui donne une liste des atomes classés par ordre décroissants.

On n'utilise pas forcément une fonction de parcours définie comme nous allons le faire, mais les définitions que nous allons donner peuvent servir à inspirer la définition de fonctions de parcours adaptées à une application particulière.

Parcours «par la gauche»

Le parcours «par la gauche» consiste à explorer la branche de gauche de l'arbre avant de consulter l'atome de la racine puis explorer la branche de droite. Ce parcours est souvent appelé «parcours gauche-racine-droite».

La définition de la fonction de parcours est

```
grda= λ . arbre-vide?(a) → ⊥ ,
          sinon concaténergrdgauche(a)consatome a  grddroite(a)
```

La fonction «concaténer» (Cf. exercice **E-36**) construit la liste en plaçant dans une même liste et dans le même ordre les éléments de deux listes données.

Parcours «par la droite»

Le parcours «par la droite» consiste à explorer la branche de droite de l'arbre avant de consulter l'atome de la racine puis à explorer la branche de gauche. Ce parcours est souvent appelé «parcours droite-racine-gauche».

La définition de la fonction de parcours est

drg$a$= λ · arbre-vide?(t) $\rightarrow \perp$,
sinon$\overrightarrow{\text{con}}$caténerdrgd()oite()()consatomea () ()drggauche$a$

4.8 Codage de Huffman & utilisation des arbres binaires

Lorsqu'on veut transmettre ou enregistrer des données, il est nécessaire de définir un codage rendant compatible la représentation des données et le support physique utilisé (conducteurs électriques, dipôles magnétiques etc.). Actuellement, la technologie que nous maîtrisons le mieux (l'électronique) incite à utiliser un codage binaire qui représente les données par des suites de 0 et de 1 (bits).

On peut définir des codes de taille fixe tel que ceux définis, par exemple, dans le tableau suivant

0	0000	2	0010	4	0100	6	0110	8	1000
1	0001	3	0011	5	0101	7	0111	9	1001

Avec un tel code, le message 1021334 sera représenté par la suite

0001000000100001001100110100

Le décodage est facile à effectuer car il suffit de tronçonner le message en éléments de 4 bits et de consulter la table de codage.

Imaginons que nous utilisions, à présent, un codage dont la taille est variable comme, par exemple, celui défini dans la table suivante

0	0	2	1010	4	1100	6	11100	8	11110
1	100	3	1011	5	1101	7	11101	9	11111

Le message précédent sera, maintenant, représenté par la suite

100010101001011101111100

On remarque que cette suite ne comporte que 23 signes alors que la suite précédente en comportait 28. L'économie réalisée sera maximale si le nombre de bits utilisés pour coder un chiffre est inversement proportionnel à sa fréquence d'usage. Ce problème constitue la clé de la télévision numérique à venir et des réseaux numériques multimedia appelés encore «autoroute de l'information».

Le problème posé par les codes de taille variable est celui du décodage en réception. En effet comment détecter la limite entre les codes des différents chiffres du message? Une solution très élégante à ce problème consiste à définir un codage de telle sorte qu'aucun code complet d'un symbole ne soit le début (préfixe) du code d'un autre symbole. Un tel code [8] est dit *préfixé*, il peut être représenté par l'arbre binaire de la figure 16, page 83. Ce code est tel que les chiffres à coder sont dans les feuilles de l'arbre d'autant plus près de la racine que ce chiffre est fréquemment utilisé. Chaque nœud est décoré par la liste des chiffres situés en dessous.

La technique de codage est très simple. Il suffit de partir de la racine et en descendant jus-

[8] Cette technique de codage a été particulièrement étudiée par David Huffman. On parle donc de code de Huffman.

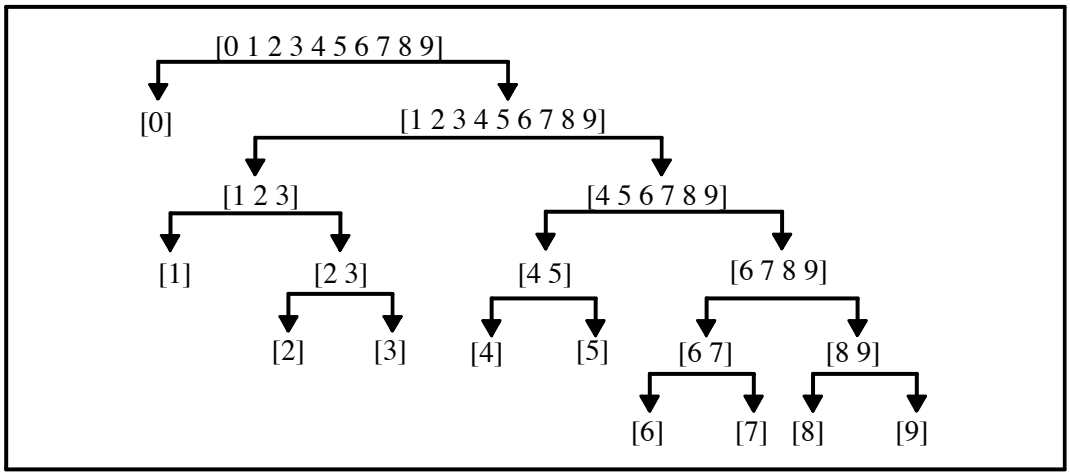


Figure 16 : Arbre de Huffman. Dans cet exemple le chiffre 0 est le plus fréquemment utilisé, suivi par le chiffre 1 puis par les chiffres 2, 3, 4 et 5 et enfin, les chiffres 6, 7, 8 et 9 sont les moins fréquemment utilisés.

qu'à la feuille associée au chiffre à coder, on ajoute un 1 au code chaque fois qu'on emprunte une branche de droite et un 0 chaque fois qu'on emprunte une branche de gauche.

Le décodage consiste à descendre dans l'arbre en prenant la branche de droite chaque fois que le code comporte un 1 et la branche de gauche chaque fois que le code comporte un 0. La feuille à laquelle on aboutit est le chiffre à trouver.

La construction de l'arbre de Huffman correspond aux expressions de définition des feuilles

$$n_0 = \text{nœud}(\bot, \bot, 0) \qquad n_5 = \text{nœud}(\bot, \bot, 5)$$
$$n_1 = \text{nœud}(\bot, \bot, 1) \qquad n_6 = \text{nœud}(\bot, \bot, 6)$$
$$n_2 = \text{nœud}(\bot, \bot, 2) \qquad n_7 = \text{nœud}(\bot, \bot, 7)$$
$$n_3 = \text{nœud}(\bot, \bot, 3) \qquad n_8 = \text{nœud}(\bot, \bot, 8)$$
$$n_4 = \text{nœud}(\bot, \bot, 4) \qquad n_9 = \text{nœud}(\bot, \bot, 9)$$

puis aux expressions de définition des nœuds

$$n_{23} = \text{nœud}(n_2, n_3, 23) \qquad n_{6\cdots9} = \text{nœud}(n_{67}, n_{89}, 6 \cdots 9)$$
$$n_{45} = \text{nœud}(n_4, n_5, 45) \qquad n_{4\cdots9} = \text{nœud}(n_{45}, n_{6\cdots9}, 4 \cdots 9)$$
$$n_{67} = \text{nœud}(n_6, n_7, 67) \qquad n_{1\cdots9} = \text{nœud}(n_{123}, n_{4\cdots9}, 1 \cdots 9)$$
$$n_{89} = \text{nœud}(n_8, n_9, 89) \qquad n_{0\cdots9} = \text{nœud}(n_0, n_{1\cdots9}, 0 \cdots 9)$$
$$n_{123} = \text{nœud}(n_1, n_{23}, 123)$$

et enfin

$$\textit{arbre-codage} = n_{0\cdots9}$$

La fonction de codage d'un chiffre peut alors être définie par

```
                coder-chiffre=
λ(x)a  .        arbre-vide?(a)  → ⊥ ,
                dans-liste?(x)(gauche(a))  → cons(0)(coder-chiffre x gauche(a)) ,
                dans-liste?(x)(droite(a))  → cons(1)(coder-chiffre x droite(a)) ,
```

Le prédicat «dans-liste?», supposé défini [9] au préalable, rend *vrai* si un élément donné se trouve dans une liste donnée.

On peut, à présent, définir la fonction de codage d'un message (supposé présenté dans une liste)

```
                coder-message=
λ(x)a  .        arbre-vide?(a)  → ⊥ ,
                vide?(x)        → ⊥ ,
                sinon → concaténer(coder-chiffre(car x)(a))(coder-message(cdr x)(a)) ,
```

La fonction de décodage d'un chiffre est alors

```
                décoder-chiffre  =
λ(x)a  .        feuille?(a)  → cons(x)(a) ,
 vide?m         ()           → cons(⊥)(⊥)
                sinon →        (car(x) 1 =  → décoder-chiffre(cdr(x))(droite(a))())
 car m                         (car(x) 0 =  → décoder-chiffre(cdr(x))(gauche(a))())
```

et la fonction de décodage d'un message devient

```
                décoder-message  =
λ(x)a  .        vide?(x)  → ⊥ ,
                sinon →  [ soit : c m décoder-chiffre m a () ,
                         [ dans : [ soit : c = car(x)
                                  [        m' = cdr c m()
                                  [ dans : cons(c)(décoder-message m' ()) , a ]]]
```

4.9 Equivalence opérationnelle.

Deux objets quelconques sont dits **opérationnellement équivalents** s'il n'existe aucun moyen de les distinguer. Cette relation est, naturellement, une relation d'équivalence qui garantit que deux objets opérationnellement équivalents peuvent être utilisés indifféremment dans une expression sans en changer la valeur.

Comme l'interprétation d'un type de données est liée à sa définition, il est de la responsabi-

[9] Définissez-le à titre d'exercice.

lité d'un type de données de définir le **prédicat** qui définit l'équivalence opérationnelle de deux objets de ce type. Il ne peut pas y avoir de définition universelle.

Remarque : un type de données est donc finalement défini par ses **constructeurs**, ses **sélecteurs** et son **prédicat d'équivalence opérationnelle**.

Ce problème n'est pas un problème propre à l'informatique, c'est le problème quotidien de la perception du monde qui nous entoure. Seulement si, au quotidien, on peut se contenter d'une perception approximative de la notion d'équivalence, une machine ne peut fonctionner qu'à partir d'une définition claire.

En ce qui concerne les types de données et les structures considérés comme primitifs, l'équivalence opérationnelle de deux objets est supposée définie à l'aide des trois prédicats *identiques?*, *même-valeur?* et *égaux?*. On est ainsi amené à distinguer les types de données qui représentent des choses qu'on appellera **atomes** des **structures** qui sont les choses utilisées pour organiser des choses.

4.9.1 Egalité de deux Atomes.

On dira que deux atomes sont égaux [10] (opérationnellement équivalents) si et seulement si le prédicat «même-valeur?» [11] rend *vrai*.

Si on se souvient (Cf. *Les Fonctions*) que le test d'égalité de deux fonctions est indécidable en général, la comparaison de deux fonctions n'a pas de sens. Considérons les deux fonctions

$$\text{double} x = \lambda \cdot ( ) x x +$$

$$\text{deux-fois} x = \lambda \cdot \cdot ( ) 2 \quad x$$

Elles sont manifestement opérationnellement équivalentes mais il est impossible de le découvrir et

identiques? ( ) double deux-fois =

même-valeur? ( ) double deux-fois =

En d'autres termes les prédicats «identiques?» et «même-valeur?» répondent toujours prudemment, c'est à dire *faux* quand l'égalité n'est pas définie. Par contre s'il est sûr qu'il s'agit de la même fonction

identiques? ( ) double double =

Maintenant comment interpréter l'évaluation suivante?

même-valeur? ( ) cons 1 3 ( ) cons 1 3 =

En tant qu'objet primitif, le résultat rendu par le constructeur *cons* est une paire et deux invocations de «cons» engendrent nécessairement deux paires différentes. La paire est un objet spécial en ce sens que c'est un *objet pour organiser des objets*, ce n'est qu'une structure et la définition de l'équivalence opérationnelle de deux structures n'est pas évidente. Cette difficulté est facile à illustrer à travers le dialogue suivant:

---

[10] Deux atomes sont opérationnellement équivalents s'ils sont de même type et si leurs valeurs peuvent être considérées comme égales dans le cadre de leur type.

[11] C'est ce prédicat que nous avons noté = jusqu'à présent.

— Ces deux bouteilles sont-elles égales?

— Elles se ressemblent bien, mais je ne sais pas ce qu'il y a dedans !

4.9.2 Egalité de deux Structures.

Les structures de base sont les paires et les listes. On dira que deux objets structurés à l'aide de paires et de listes sont égaux (opérationnellement équivalents) si et seulement si le prédicat *égaux?* répond *vrai*.

Deux objets structurés sont égaux si:

1. ils sont structurés de la même manière,
2. leurs atomes correspondants sont égaux au sens de «même-valeur?».

Considérons les exemples suivants

égaux? (cons 3 (cons 13 ())) =

égaux? (cons 15 (cons 16 ())) =

égaux? (cons (cons 124 ()) (cons 35 ())) (cons (cons 124 ()) (cons 35 ())) =

Considérons la situation suivante

l_1 = (cons (cons 2 (cons 3 ())) ⊥)

l_2 = (cons (cons 2 (cons 3 ())) ⊥)

$l_3 = l_1$

Comment interpréter les évaluations suivantes?

même-valeur? l_1 l_2 =

égaux? l_1 l_2 =

égaux? l_1 l_3 =

même-valeur? l_1 l_3 =

Pour «même-valeur?» les deux listes l_1 et l_2 sont différentes puisque «même-valeur?» ne s'intéresse pas à ce qu'elles contiennent, par contre pour «égaux?» elles sont égales puisqu'elles sont structurées de la même manière et que leurs atomes correspondants sont égaux. Quant aux listes l_1 et l_3, comme il s'agit en fait de la même liste sous deux noms différents, «même-valeur?» détecte leur identité.

4.10 Conclusion.

Nous arrivons à un tournant important. Nous pouvons être convaincus, à présent, qu'il est possible de définir toutes les applications possibles en structurant les ensembles de données qu'elle doit manipuler ainsi que les fonctions qui permettent ces manipulations. Par contre il est évident qu'il est temps de définir une technique d'écriture de nos définitions plus commode et adaptée, en particulier, à la construction de définitions de grande taille telles qu'on les rencontrera dans le cadre d'applications réelles.

Le chapitre suivant va, donc, introduire une telle écriture que nous utiliserons alors systématiquement.

4.11 Exercices.

E-27 Chercher par invariant une «définition apparemment équivalente» de la fonction «réduire». A l'aide d'exemples, montrer que si cette définition se comporte comme celle donnée au paragraphe 4.5, page 71 pour certaines opérations, elle en diffère fondamentalement pour d'autres.

E-28 Montrer qu'il est possible de définir la fonction «projeter» (Cf. paragraphe 4.5, page 71) à l'aide de la fonction «réduire». Cela prouve que la fonction «réduire» est plus fondamentale que la fonction «projeter».

E-29 Définir le constructeur et les sélecteurs de la structure **Salarié** définie en utilisant une liste.

E-30 Définir le constructeur et les sélecteurs d'un tableau à 4 champs construit en utilisant une liste. Quel est l'intérêt de cette représentation ?

E-31 Définir la fonction «diviseurs» rendant la liste des diviseurs d'un nombre donné. Par exemple

diviseurs(6) ⇒ [1 2 3]

diviseurs(8) ⇒ [1 2 4]

diviseurs(12) ⇒ [1 2 3 4 6]

E-32 Définir la fonction «parfaits» rendant la liste des «nombres parfaits» compris entre 1 et *n*. Un nombre est «parfait» s'il est égal à la somme de ses diviseurs (Cf. exercice **E-31**). Par exemple

parfaits(10) ⇒ [6]

E-33 Définir la fonction «diviseurs-premiers» rendant la liste des diviseurs premiers d'un nombre donné. Par exemple

diviseurs-premiers(6) ⇒ [2 3]

diviseurs-premiers(8) ⇒ [2 2 2]

E-34 Modifier la fonction «diviseurs-premiers» de l'exercice **E-33** de telle sorte que les diviseurs premiers du nombre donné n'apparaissent qu'une seule fois. Par exemple

diviseurs-premiers(8) ⇒ [2]

Nota: *il est commode de définir une fonction «* quotient-complet» *qui rend le dernier quotient de toutes les divisions qu'on peut effectuer d'un nombre m donné par un nombre n donné. Par exemple*

quotient-complet(28 2) =

E-35 Définir la fonction «premiers» rendant la liste des «nombres premiers» compris entre 1 et *n*. Un nombre est premier s'il n'est divisé que par 1 et par lui-même. Par exemple

premiers(10) ⇒ [7 5 3 2 1]

Nota: *on peut utiliser le résultat de l'exercice* **E-31** *pour élaborer une première solution.*

E-36 Définir la fonction «concaténer» qui place les éléments d'une liste à la suite de ceux d'une

autre. Par exemple

concaténer(1 2 3 4 5 6 , 4 5 6) = 1 2 3 4 5 6 4 5 6

E-37 Nous avons vu qu'une liste peut contenir soit des atomes, soit des paires, soit des listes.

1. Définir le prédicat «paire?» qui rend *vrai* si son argument est une paire. Nous supposerons qu'il existe un prédicat «atome?» rendant *vrai* lorsque son argument n'est pas une paire.
2. Définir la fonction «aplatir» qui rend la liste de tous les atomes contenus dans une liste. Par exemple

aplatir((1 3) 7 (2 (6 9) 0)) ⇒ 1 3 7 2 6 9 0

E-38 Nous savons qu'une liste peut contenir soit des atomes, soit des paires, soit des listes.

1. Définir le prédicat «liste?» rendant *vrai* si son argument est une liste. Nous supposerons qu'il existe un prédicat «paire?» qui rend *vrai* lorsque son argument est une paire.
2. Définir la fonction «renverser-en-profondeur» qui rend la liste qu'on lui donne renversée ainsi que toutes les listes et paires qu'elle contient. Par exemple

renverser-en-profondeur(1 (2 3) 4) ⇒ (4 (3 2) 1)

E-39 Il arrive fréquemment qu'on soit amené à manipuler un ensemble de données. Un ensemble est une collection d'objets telle qu'un objet ne peut s'y trouver qu'une seule fois.

1. Choisir une représentation pour une telle collection et définir le constructeur «ensemble».
2. Définir les primitives suivantes :

«ajouter»	pour ajouter un élément à l'ensemble,
«retirer»	retire un élément de l'ensemble,
«appartient?»	teste l'appartenance d'un objet à l'ensemble,
«union»	effectue l'union de deux ensembles,
«intersection»	effectue l'intersection de deux ensembles,
«différence»	effectue la différence de deux ensembles.

3. En déduire la définition de la fonction «liste → ensemble» qui transforme une liste en l'ensemble correspondant.
4. Evaluer les ordres de croissance en temps des différentes primitives définies ci-dessus.

E-40 Lorsqu'il est nécessaire d'effectuer des statistiques sur une collection d'objets, il est commode de définir une structure adaptée: le **multi-ensemble**. Un multi-ensemble est une collection d'objets telle qu'on peut y trouver le même objet **plusieurs fois**.

1. Choisir une représentation pour une telle collection et définir le constructeur «multi-ensemble».
2. Définir les primitives suivantes :

«ajouter»	pour ajouter l'occurrence d'un élément du multi-ensemble,
«retirer»	retire l'occurrence d'un élément du multi-ensemble,
«occurrence?»	rend le nombre des occurrences d'un élément du multi-ensemble,
«union»	effectue l'union de deux multi-ensembles,
»intersection»	effectue l'intersection de deux multi-ensembles,
différence»	effectue la différence de deux multi-ensembles.

3. En déduire la définition de la fonction «liste → mult-ens» qui transforme une liste en un

multi-ensemble.

4. Evaluer les ordres de croissance en temps des différentes primitives définies ci-dessus.

   Nota: *pensez à associer l'élément et son nombre d'occurrences*.

E-41 En répondant aux questions 4 des exercices **E-39** et **E-40**, vous avez du être désolés du manque d'efficacité flagrant des représentations les plus simples. Il est possible d'améliorer sensiblement les choses en utilisant une représentation plus favorable lorsque les éléments à regrouper sont des **magnitudes** [12].

Nous supposeront que les éléments à regrouper peuvent être classés par les prédicats > et <.

1. En supposant une représentation sous la forme d'un arbre binaire, définir le constructeur «ensemble-trié».
2. Définir les primitives suivantes :

«ajouter»	pour ajouter un élément, à sa place, à l'ensemble trié,
«retirer»	retire un élément de l'ensemble trié,
«appartient?»	teste l'appartenance d'un objet à l'ensemble trié,
«union»	effectue l'union de deux ensembles triés,
«intersection»	effectue l'intersection de deux ensembles triés,
«différence»	effectue la différence de deux ensembles triés.

3. En déduire la définition de la fonction «liste $\rightarrow$ ensemble-trié» qui transforme une liste en ensemble trié.
4. Evaluer les ordres de croissance en temps des différentes primitives définies ci-dessus.

E-42 Lorsqu'on veut étudier comment varie une grandeur en fonction du temps (le cours d'une action en bourse, par exemple) on construit une série chronologique en rangeant toutes les valeurs successives de cette grandeur dans la liste $\{x_N\ x_{N-1}\ x_{N-2}\ \cdots\ x_1\}$. Pour déterminer la «vrai» variation de cette grandeur, on peut éliminer les fluctuations aléatoires en effectuant le *filtrage* dont l'équation récurrente est la suivante

$$y_n = a y_{n-1} + (1-a)\, x_n \quad \text{avec} \quad y_0 \ll 1$$

Définir la fonction «filtrer» rendant la liste filtrée.

E-43 Définir directement la fonction «produit-scalaire» pour calculer le produit-scalaire de deux listes qu'on supposera de taille identique. Le produit-scalaire est défini par l'expression

$$sx = x_0 y_0 + x_1 y_1 \cdots x_N y_N$$

x_i et y_j étant respectivement les termes correspondants de la première et de la deuxième liste.

E-44 De nombreux langages modernes (dont *Scheme*) permettent d'utiliser des nombres entiers dont la taille n'est limitée que par la mémoire disponible sur la machine utilisée et non pas par la taille d'un mot-mémoire (16 ou 32 bits). De tels nombres sont quelque fois appelés «big-num».

De tels nombres entiers permettent d'évaluer des expressions de la forme :

```
[3] (fac 300)
```

[12] Ce termes est issu de l'environnement Smalltalk. Un objet est une magnitude lorsque l'ensemble auquel il appartient est associé à une relation d'ordre total.

```
3060575122164406360353704612972686293885888041735769994167767412594765331767168674655152914224775733499391478887017263688642639077590031542268429279069745598412254769302719546040080122157762521768542559653569035067887252643218962642993652045764488303889097539434896254365322980776521270822437639449120128678675368305712293681943699564604981664502277165001851765464693401122260347297246633325858350687015016979416885035375213755491028912640715715483028228493795263658014523523315693648223436799545940952768206080622328123873838808170496000000000000000000000000000000000000000000000000000000000000000
```

On s'appuiera, bien entendu, sur l'arithmétique des nombres entiers du langage utilisé — ici *Scheme* — et on supposera, pour les besoins de cet examen, que les entiers de *Scheme* ne sont pas des big-nums.

La représentation la plus naturelle pour un big-num est la liste de ses chiffres, **sans zéros non significatifs** [13], en notation décimales. C'est donc cette représentation que nous allons adopter.

1. Il reste un point en suspend. La représentation d'un big-num peut être définie «poids forts en tête» (de liste) ou «poids faibles en tête» (de liste). Quels sont les arguments qui nous ont amené à choisir une représentation poids faibles en tête ?

Dans cette représentation, le nombre 128 sera représenté par la liste `(8 2 1)`. La liste vide — ou la valeur booléenne `#F` — **joue le rôle de la valeur** $\perp$ (bottom).

2. Bien entendu, *Scheme* ne sait pas afficher un big-num. Définir la fonction `(afficher bn)` qui affiche un big-num en notation décimale.

```
(afficher '(8 2 1))        ⇒   128
(afficher '())             ⇒   ()
(afficher #F)              ⇒   ()
```

Nota: *Le double rôle joué par la liste vide — ou la valeur booléenne* `#F` *— nécessitera souvent la définition d'une fonction auxiliaire.*

Nous supposerons que les deux constantes d'utilisation fréquente suivantes ont été définies :

```
(define zero '(0))
(define un '(1))
```

3. Définir les trois prédicats `(egaux? bn1 bn2)`, `(zero? bn)` et `(un? bn)` rendant `#T` si les big-nums `bn1` et `bn2` sont égaux et si, respectivement, le big-num `bn` vaut 0 et 1.
4. Définir la fonction `(naturel-to-bignum n)` qui rend l'entier naturel `n` sous la forme d'un big-num.

```
(naturel-to-bignum 345)    ⇒ (5 4 3)
(naturel-to-bignum -2)     ⇒ ()
```

Nota: *le quotient euclidien correspond à la fonction Scheme* `quotient` *tandis que le reste euclidien correspond à la fonction Scheme* `remainder`.

5. Définir la fonction `(suivant bn)` qui rend le big-num qui suit immédiatement un big-num donné.

```
(suivant '(8 2 1))         ⇒ (9 2 1)
```

[13] Les 0 non significatifs sont à gauche dans la représentation décimale courante. Ainsi, les deux entiers 003 et 3 sont identiques.

```
(suivant '(9 9))          ⇒ (0 0 1)
```

Nota: *s'inspirer de la technique manuelle est une bonne idée.*

6. Définir la fonction `(precedent bn)` qui rend le big-num qui précède immédiatement un big-num donné.

```
(precedent '(8 2 1))      ⇒ (7 2 1)
(precedent '(0 0 1))      ⇒ (9 9)
(precedent zero)          ⇒ ()
```

Nota: *une difficulté dans la définition de cette fonction réside dans le fait qu'il ne faut pas faire apparaître de 0 non significatifs dans la représentation du résultat.*

Les fonctions de base précédente ayant été définies, on peut aborder la définition des opérations arithmétiques principales : somme, différence, produit, quotient euclidien, reste euclidien.

Ces opérations notées respectivement +, -, $\dot{}$, / et % peuvent être définies par les expressions suivantes :

somme :
$n + 0 = n$
$n + m = \text{suivant}(n) + \text{précédent}(m)$

différence :
$n - 0 = n$
$0 - m = \perp$
$n - m = \text{précédent}(n) - \text{précédent}(m)$

produit :
$n \dot{} 0 = 0$
$n \dot{} m = n + n \dot{} \text{précédent}(m)$

quotient :
si $n < m$ $\quad n / m = 0$
si $n \geq m$ $\quad n / m = 1 + (n - m) / m$

reste :
si $n < m$ $\quad n \% m = n$
si $n \geq m$ $\quad n \% m = (n - m) \% m$

avec

inférieur noté < :
$n < 0 = \text{faux}$
$0 < m = \text{vrai}$
$n < m = \text{précédent}(n) < \text{précédent}(m)$

7. Définir les fonctions `(somme bn1 bn2)`, `(difference bn1 bn2)`, `(produit bn1 bn2)`, `(quotient bn1 bn2)` et `(reste bn1 bn2)` conformes à leur spécification ci-dessus.

Nota: *Vous n'avez pas manqué de remarquer de la définition du prédicat < (inférieur) ne fait pas appel à la définition de la différence.*

E-45 On se propose de définir une arithmétique des polynômes. Pour cela on supposera qu'un polynôme est représenté par la liste de ses coefficients classés par puissances croissantes

$$2 + x + 2x^3 + 7x^5 \rightarrow [2\ 1\ 0\ 2\ 0\ 7]$$

Dans un premier temps, on supposera que cette liste a été directement construite à la main en utilisant le constructeur de paires «cons».

1. Définir la fonction «poly-add» donnant la somme de deux polynômes. Cette somme est le polynôme dont le coefficient de degrés *n* est égal à la somme des coefficients corres-

pondants dans les deux polynômes ajoutés.

$$(1 + x + 3x^2) + (2 + x^2) = 3 + x + 4x^2$$

2. Définir la fonction «poly-eval» qui calcule la valeur d'un polynôme pour une valeur donnée de sa variable. On utilisera le schéma récursif de Horner qu'on peut illustrer par l'exemple

$$2 + x + 3x^2 + 7x^5 = 2 + x(1 + x(3 + x(0 + x(0 + x(7)))))$$

3. Définir la fonction «scal-poly-mult» donnant le polynôme produit d'un polynôme et d'un scalaire donné.

$$5 \cdot (2 + x + 3x^2 + 7x^5) \rightarrow (10 + 5x + 15x^2 + 35x^5)$$

4. En déduire la fonction «poly-poly-mult» donnant le produit de deux polynômes.

$$(1 + 2x) \cdot (2 + 3x^2) = 2 + 4x + 3x^2 + 6x^3$$

Nota: *Il peut être utile de se souvenir que la multiplication précédente peut être présentée comme la multiplication manuelle de deux nombres* :

$$\begin{array}{rrrr} & 2 & 0+x & +3x^2 \\ \cdot & 1 & +2x & \\ \hline & 2 & 0+x & +3x^2 \\ + & 0 & +4x & +0x^2 + 6x^3 \\ \hline & 2 & +4x & +3x^2 + 6x^3 \end{array}$$

5. En déduire la fonction «poly-puissance» qui élève un polynôme à une puissance entière donnée.
6. Définir la fonction «poly-dériver» qui donne le polynôme dérivée d'un polynôme donné. On se souviendra que la dérivation a, entre autres, les propriétés suivantes

$$\begin{aligned} \text{dérivation}(k) &= 0 \\ \text{dérivation}(f_1 + f_2) &= \text{dérivation}(f_1) + \text{dérivation}(f_2) \\ \text{dérivation}(k \cdot f) &= k \cdot \text{dérivation}(f) \\ \text{dérivation}(x^n) &= n \cdot x^{n-1} \end{aligned}$$

7. Le constructeur de polynômes que nous avons utilisé est totalement inadapté à la construction de polynôme creux comme, par exemple, $1 + x^{20}$. Définir le constructeur de polynômes «poly-cons» qui traduit une représentation condensée des polynômes en la liste de leurs coefficients. Cette représentation est la liste des coefficients rangés, comme précédemment, en ordre croissant des degrés. Par contre, lorsqu'un terme est une liste à un élément, cet élément est le degrés à associer au coefficient suivant.

   Par exemple

$$[1\ [9]\ 1] \Rightarrow 1 + x^9 \Rightarrow [1\ [9]\ 1]$$
$$[2\ [2]\ 1\ [9]\ 5] \Rightarrow 2 + x^2 + 2x^3 + 5x^9 \Rightarrow [2\ 0\ 1\ 2\ 0\ 0\ 0\ 0\ 5]$$

E-46 Les Egyptiens ne connaissaient que les nombres entiers et les fractions de numérateur 1 (dites fractions égyptiennes). Définir la fonction «décomposer» qui rend la liste des termes de la décomposition d'un nombre rationnel.

Par exemple

décompose(19) = [1/4, 1/8] car 19/8 = 2 + 1/4 + 1/8

E-47 Dans les conditions de l'exercice **E-23** définir la fonction «précédent» qui rend le nombre qui précède celui donné en argument.

> **Nota:** *Cet exercice est difficile. Church lui-même n'y parvint pas. Il venait à peine de se convaincre que cette fonction n'était pas définissable, que Kleene en trouva une définition qui fut publiée en 1981. La difficulté que rencontra Church était que la paire n'avait pas encore été définie.*

E-48 Nous verrons un peu plus loin qu'une structure de données est particulièrement utile, elle est appelée **Dictionnaire**. Un dictionnaire est un ensemble d'associations entre un nom et une valeur.

1. Définir la fonction «dictionnaire» qui rend un dictionnaire vide.
2. Définir la fonction «insérer» qui rend un dictionnaire correspondant à un dictionnaire donné dans lequel on a inséré un nom et la valeur associée.
3. Définir la fonction «modifier» qui rend un dictionnaire tel que la valeur associée à un nom donné dans un dictionnaire donné a été remplacée par une valeur donnée.

   Nota: *commencer par définir l'association entre un nom et une valeur.*

E-49 **Les nombres heureux**[14] - L'adjudant réunit ses hommes pour décider qui serait de corvée de patates.

— Mettez-vous en file indienne et comptez-vous à partir de 2 !

Le premier homme de la file dit 2, le suivant dit 3, le suivant 4 et ainsi de suite.

— Le premier de la file, sortez du rang. Vous êtes dispensé de corvée. Quel votre numéro ?

— 2

répondit le soldat.

— Les hommes de 2 en 2 en commençant à celui qui vient de partir, sortez des rangs, vous êtes de corvée.

Et le processus recommença. Le premier restant dans les rangs avait le numéro 3 et il était heureux : dispensé de corvée. Les hommes de 3 en 3 en commençant à lui sortirent des rangs pour la corvée...

Définir une fonction rendant le nombre heureux numéro *n*. Pour être sûr que vous avez bien compris, voici les premiers nombres heureux

2 3 5 7 11 13 17 23 25 29...

Les nombres heureux ne sont pas nécessairement premiers et les nombres premiers ne sont pas nécessairement heureux.

E-50 **Exercice difficile** - Définir la fonction «placer-reines» qui rend les positions dans lesquelles on peut placer 8 dames sur un échiquier de telle sorte qu'aucune ne soit «en prise».

Nota: *la difficulté de cet exercice est surtout liée à la représentation choisie pour les cases de l'échiquier.*

---

[14] Extrait de *Jeux et casse-tête à programmer* de J.Arsac -Ed. DUNOD

5. Eléments de Programmation.

Data and procedures and the values they amass,
High-order functions to combine and mix and match,
Objects with their local state, the messages they pass,
A property, a package, the control point for a catch —

In the Lambda Order they are all first class.
One Thing to name them all, One Thing to define them,
One Thing to place them in environments and bind them,
In the Lambda Order they are all first-class.

auteur inconnu
Revised[3] Report on the Algorithmic Language Scheme

Notre insistance à donner une définition rigoureuse des fonctions et une description de leurs principales propriétés est justifiée par le fait que, ce que la plupart des langages informatiques appelle «fonction» n'offre pas ces propriétés. Le seul point commun entre les fonctions informatiques usuelles et les fonctions mathématiques est qu'elles rendent toutes les deux un résultat.

Les chapitres précédents ont montré que le concept de fonction, tel que les mathématiques l'ont défini, offre toute la puissance nécessaire à la description des applications informatiques pour en décrire les données, les traitements et l'organisation. L'écriture informelle que nous avons utilisée, si elle nous permet de réfléchir et de traduire par écrit nos raisonnements, n'est pas assez bien définie pour garantir qu'une simple lecture n'engendre pas, de temps en temps, des malentendus.

Ces malentendus ne sont pas très génants tant que la forme écrite est utilisée pour la communication entre des humains car il est toujours possible de discuter et ainsi de faire naître la compréhension. Par contre, ils interdisent la mécanisation de la mise en œuvre des descriptions qui ont été construites.

5.1 Définir & Programmer.

Dans les chapitres précédents, nous avons supposé l'existence de la «notion de fonction» à travers les mécanismes d'application fonctionnelle et d'abstraction fonctionnelle. Nous avons également supposé, bien que ce ne soit pas strictement indispensable, l'existence de données primitives. Pour exprimer nos idées concernant tous ces éléments, nous avons introduit, au fur et à mesure de nos besoins, quelques règles d'écriture soit empruntées directement aux mathématiques soit définies spécifiquement.

5.1.1 Application fonctionnelle.

L'application d'une fonction à ses arguments est toujours écrite sous l'une des deux formes suivantes

<application-fonctionnelle> ::= *<symbole>* (*<argument>* , ⋯) |
<abstraction-fonctionnelle> (*<argument>* , ⋯)
<argument> ::= *<application-fonctionnelle>* |
<abstraction-fonctionnelle> |
<constante>
<constante> ::= *<symbole>* | *<nombre>* | *<paire>* | *<liste>*

5.1.2 Abstraction fonctionnelle.

L'abstraction fonctionnelle est toujours écrite sous la forme

<abstraction-fonctionnelle> ::= λ (*<définition>* , ⋯) . *<symbole>*
<définition> ::= *<application-fonctionnelle>* |
<abstraction-fonctionnelle> |
<constante>

On se souvient cependant, que λ a été introduit comme une fonction qui rend une fonction. L'utilisation de λ est donc typiquement une application fonctionnelle et il n'était pas nécessaire d'introduire une écriture spéciale [1]. Nous l'avons fait d'une part pour sacrifier à la coutume et d'autre part pour particulariser l'abstraction fonctionnelle du fait de son rôle fondamental dans tous nos raisonnements. Ce genre d'écriture est souvent appelée un «sucre syntaxique» pour dénoter le fait que ce n'est pas nécessaire pour survivre mais que c'est tellement bon à manger!

5.1.3 Données primitives.

Afin de pouvoir *nommer* les choses, nous avons supposé l'existence d'un ensemble illimité de *symboles* représentés sous la forme d'une suite ininterrompue de signes typographiques n'introduisant pas d'ambiguité avec d'autres représentations (celle des nombres par exemple). Ces symboles ont été utilisés soit pour donner un nom à des *valeurs* (ce que rend l'éva-

[1] Un théorème fondamental de Schönfinkel et Curry montre que la λ-notation peut entièrement être reconstruite sans l'abstraction fonctionnelle uniquement à partir de deux fonctions primitives convenablement choisies.

luation d'une application fonctionnelle) soit pour être utilisés «en soi» à travers un mécanisme de citation.

Pour pouvoir «faire des calculs», nous avons supposés que nous disposions des nombres ainsi que de toutes les opérations que la théorie des nombres met à notre disposition (Cf. tableau 2, page 98).

Types de données		**Constantes du type**	**Opérateurs de relation**	**Opérateurs internes**
Symboles		**x y foo baz ...**	=[i]	
Booléens		*vrai faux*	néant[ii]	←∨∧...
Nombres	entiers	-2 -1 0 1 2 ...	= > < ≥ ≤ ...	+-/× |...
	réels	1,2 0,12 ...		

Tab. 2 : Types de données supposés informellement prédéfinis.

[i] au sens de identique?

[ii] dans le cas des Booléens, les opérateurs de relations sont des opérateurs internes.

Chaque ligne de ce tableau définit ce qu'on appelle une algèbre et cvonstitue sa signature.

5.1.4 Structures courantes.

Nous avons particulièrement utilisé deux types de structures, les paires dénotées $[\,]_{tetequeue}$ et les listes dénotées $[\,]_{x_1\ x_2\ \cdots\ x_N}$

5.1.5 Formes particulières.

Dans le cas de définitions complexes, nous avons utilisé une première forme qui permet de définir l'environnement (des variables intermédiaires) qui donne un sens à une expression

$$\begin{bmatrix} \text{soit(-rec) :} & \cdots \\ & \cdots \\ & \cdots \\ \text{dans :} & \cdots \end{bmatrix}$$

et une forme qui permet de concevoir des définition «par cas»

$$\langle prédicat \rangle \langle \vec{conséquent} \rangle \langle alternative \rangle$$

présentée verticalement lorsque cela améliore la lisibilité

$\langle prédicat \rangle \langle \vec{conséquent} \rangle$,
$\langle prédicat \rangle \langle \vec{conséquent} \rangle$,
...
$sinon \langle \vec{alternative} \rangle$

5.1.6 Un Langage pour s'exprimer, un Langage pour programmer.

Nous avons vu, au cours des premiers chapitres, que l'abstraction fonctionnelle permet soit d'étendre une algèbre prédéfinie en définissant de nouvelles opérations soit de définir de nouveaux types de données (une nouvelle algèbre).

Les règles d'expression qui permettent d'exprimer des idées constituent un *langage*. Lorsque un langage est conçu pour être interprété de façon mécanique, on l'appelle un *langage de programmation*.

Un langage est caractérisé par :

vocabulaire	l'ensemble des mots qu'il utilise,
syntaxe	les règles de combinaison de ses différents mots pour construire les expressions syntaxiquement «bien formées»,
sémantique	le sens qu'on peut donner aux différentes expressions bien formées.

Contrairement à un «langage naturel», un «langage artificiel mécanisable» nécessite que toute expression bien formée est associée à une et une seule signification.

5.2 Le Langage Scheme.

Nous allons, à présent, introduire le langage de programmation *Scheme* [2] dans lequel les fonctions sont réellement des fonctions et le style de programmation qui lui est associé est donc appelé *programmation fonctionnelle*.

La syntaxe de ce langage (ses règles d'écriture) est extrêmement simple. Nous ne consacrerons donc pas beaucoup de temps aux problèmes de syntaxe pour nous intéresser essentiellement aux *problèmes de sémantique* associés à son utilisation.

Ainsi, comme tout langage efficace, *Scheme* possède trois mécanismes de construction :

1. des *expressions primitives* qui représentent les entités les plus simples du langage considéré,
2. des *moyens de composition* pour construire des expressions composées à partir d'expressions plus simples,
3. des *moyens d'abstraction* pour nommer ou manipuler comme un tout un objet composé.

Le langage *Scheme* est remarquable car non seulement il traduit tous les concepts que nous avons introduits aux chapitres précédents, mais, en plus, nous verrons qu'il permet de définir une **fonction extraordinaire qui permet d'évaluer toutes les autres fonctions** que nous appellerons son **interprète**. Dans un premier temps, nous considèrerons simplement que cette fonction existe, puis petit à petit, nous montrerons comment il est possible de la définir. Un langage qui possède la propriété de pouvoir se définir lui-même est dit *méta-circulaire*.

Nous pourrons donc utiliser le langage *Scheme* à deux fins :

1. pour décrire et formaliser des raisonnements sans ambiguïté. Il peut donc être utilisé à la place de l'écriture informelle que nous avons utilisée auparavant.

[2] L'interprète *Scheme* présenté ici est la version PC-Scheme/Geneva V4.02 disponible librement par `ftp` auprès du serveur `cui.unige.ch` dans le répertoire `/public/pcs` sous la forme de l'exécutable auto-décompressant `pcscheme.exe`.

2. pour demander la résolution des équations engendrées par les définitions qu'il a permis de construire. Le fait que l'interprète accepte ces définitions **prouvent** qu'elles sont correctes [3].

Nous n'introduirons pas toutes les possibilités expressives de *Scheme*, cela pour 3 raisons :

1. certaines décrivent des concepts sémantiques que nous n'avons pas introduit aux chapitres précédents et qui sortent du cadre d'une initiation à l'informatique.
2. certaines concernent des possibilités introduites pour faciliter le développement d'applications informatiques (fonctions de gestion des fichiers, fonctions de manipulation des objets graphiques utilisés dans la conception des interfaces homme-machine par exemple) qui sortent du cadre de la simple analyse d'un problème informatique.
3. certaines peuvent être considérées comme une «érosion sémantique».

Cette maladie (ce terme péjoratif n'engage que l'auteur) a tendance à frapper tous les langages informatiques qui évoluent d'une «forme sémantique aussi pure que possible» vers des formes «abâtardies», souvent sous la pression des compagnies qui commercialisent les environnements de développement associés. Le langage Scheme a pratiquement échappé, à l'heure actuelle, à cette maladie, c'est la principale raison de son choix. Par contre, d'autres langages ont été plus ou moins pervertis (Lisp, Smalltalk donnant C++, etc.).

5.3 Expressions Scheme.

Toute expression Scheme dénote l'application d'une fonction à ses arguments et sa syntaxe est la suivante

```
(fonc a1 a2 a3 ...)
```

Elle représente l'application de la fonction `fonc` aux arguments `a1`, `a2`, `a3` ..., elle est strictement équivalente à l'écriture $fonc(a_1, a_2, a_3, ...)$ que nous avons utilisé jusqu'à présent. *Scheme* n'introduit pas de forme particulière pour l'abstraction fonctionnelle (nous venons de voir que ce n'est pas nécessaire).

Cette écriture dénote également la structure de liste. Ainsi, **la liste est la seule représentation utilisée pour dénoter les entités complexes Scheme**.

Un interprète *Scheme* fonctionne a priori en mode interactif, il affiche une invite à l'écran,« `[1]` » par exemple. L'utilisateur tape alors une expression [4], l'interprète l'évalue, affiche le résultat puis une nouvelle invite.

Nota: *les caractères affichés par l'interprète sont composés en* `courier-italique`

[3] Dire qu'une définition est correcte n'implique pas qu'elle est conforme à son cahier des charges. Cela signifie simplement qu'elle est cohérente et qu'elle engendre un processus de calcul que se comporte conformément à sa définition.

[4] C'est la parenthèse fermante qui dénote la fin de l'expression et non pas des signes typographiques tels que ; ou ↵.

tandis que ceux tapés par l'utilisateur sont composés en `courier-roman`.

```
[1]  (+ 3 4)
7

[2]  (* 5 6 7)
210

[3]  (/ 10 6)
1.66667
```

`+`, `*` et `/` sont des **fonctions primitives** de *Scheme*. La représentation des nombres inexacts[5] comportent toujours un point décimal. Cette manière d'exprimer les calculs s'appelle la *notation polonaise préfixée* . La ligne `[1]` est équivalente à l'expression `3+4` que nous avons utilisé jusqu'à présent.

Cette notation permet de concevoir des fonctions capables de traiter un nombre quelconque d'arguments

```
[4]  (+ 1 2 3 4)
10

[5]  (* 1 2 3 4 5 6)
720

[6]  (- 4 3 2 1)
-2
```

ce qui revient à considérer que ces fonctions sont systématiquement projetées sur la liste de leurs arguments.

Ce type de notation s'applique immédiatement aux expressions emboîtée

```
[7]  (+ (* 3 5) (* 12 3))
30
```

L'expression de la ligne `[7]` correspond à l'écriture `(3*5)+(12*3)`. Des expressions qui nous semblent complexes ne troublent pas l'interprète

```
[8]  (+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
57
```

On utilise fréquemment une présentation indentée (appelée *pretty-printing*) plus lisible par les humains

```
[9]  (+ (* 3
           (+ (* 2 4)
              (+ 3 5)))
        (+ (- 10 7)
           6))
57
```

On constate qu'un tel interprète fonctionne toujours selon le même cycle :

1. *lecture* d'une expression au clavier,
2. *évaluation* de cette expression,
3. *affichage* du résultat.

[5] Les nombres, en tant que données, sont répartis en deux classes, les nombres *exacts* qui peuvent être exactement calculés (les nombres entiers et les nombres rationnels) et les nombres *inexacts* qui ne peuvent pas l'être (nombres réels et les nombres complexes).

Ce cycle est appelé *boucle de lecture-évaluation-impression* (read-eval-print loop), nous y reviendrons. On peut, cependant, donner déjà une ébauche de la définition de la fonction «interprète»

interprète = λ…() . interprète(afficher(**Evaluer**(lire ()))), …

Manifestement l'essentiel de l'interprète se situe au sein de la fonction «Evaluer» et nous y reviendrons.

5.4 Nommage & Environnement.

L'établissement d'un lien entre un nom et une valeur est réalisé en *Scheme* grâce à un opérateur noté `define`.

Par exemple

```
[10](define taille 10)
TAILLE

[11](+ 1 taille)
11
```

Les effets de l'application de l'opérateur `define` à ses arguments sont doubles:

1. création d'un lien (*nom* ↔ *valeur*) introduit dans l'environnement courant,
2. production d'une valeur arbitrairement [6] définie.

Le premier effet, qui ne transparaît pas à travers la valeur rendue, est appelé *effet de bord* (side-effect en anglo-américain).

Voici quelques autres exemples d'utilisation du `define`

```
[12](define pi 3.141592)
PI

[13](define rayon 5)
RAYON

[14](* pi rayon rayon)
78.539816

[15](define circonference [7] (* 2 pi rayon))
CIRCONFERENCE

[16]circonference
31.41592
```

L'opérateur `define` dénote exactement le signe = utilisé pour créer les liens de nommage que nous avons utilisés aux chapitres précédents. La ligne `[10]` est donc équivalente à l'écriture *taille* = 10 tandis que la ligne `[11]` traduit 1 + *taille*

Cette possibilité d'associer des valeurs à des symboles oblige l'interprète à gérer un *dictionnaire* des liens (*nom* ↔ *valeur*). C'est ce dictionnaire qui matérialise l'environnement dont nous avons parlé précédemment (Cf. exercice **E-48**).

---

[6] Dans cette implémentation de *Scheme*, la valeur rendue par l'opérateur `define` est le nom lié lui-même. Pour cette version de *Scheme* les lettres majuscules et les lettres minuscules sont identiques.

[7] Les accents dans les noms définis à titre d'exemple ont été sacrifiés sur l'autel de la compatibilité entre les différentes implémentations de *Scheme*.

On peut, legèrement, améliorer notre ébauche de la définition de la fonction «interprète» en notant que l'évaluation d'une expression revient à lui donner une signification (Cf. paragraphe 2.4.1, page 19) à l'aide d'un **environnement**. Il est alors clair que la fonction «Evaluer» ne peut évaluer une expression que si elle connait l'environnement qui lui donne un sens. La définition de la fonction interprète devient donc

$$\text{interprète} = \lambda\ \cdot\ \text{interprète}(\text{fonction Evaluer lire}\ ())\ e\ ,$$

5.5 Principaux Objets prédéfinis Scheme.

5.5.1 Données Scheme.

Les données pouvant être manipulées en *Scheme* sont:

1. les nombres booléens,
2. les nombres exacts (entiers),
3. les nombres inexacts (réels),
4. les chaînes de caractères,
5. Les fonctions (procédures).

Ces différents types sont associés à des prédicats d'identification et à des règles d'écriture résumés dans le tableau 3, page 103.

Types	Prédicats d'identification			Exemples
booléens	atom?	boolean?		#T #F
nombres entiers		number?	exact? integer?	10 21 -5
nombres réels			inexact? float? real?	45.2 5.1e2 -12.2
symboles		symbol?		'dumpty
chaînes de caractères		string?		"humpty"
fonctions		procedure?		(lambda (x) (* 2 x))

Tab. 3 : Types prédéfinis Scheme que nous utiliserons.

5.5.2 Paire Scheme.

La paire *Scheme* est l'association de deux objets. Cette association aura de nombreux usages et servira d'élément de base pour la construction de presque toutes les données composées.

Constructeur de paires.

Une paire est construite par invocation de la fonction `cons`

```
[17](cons 2 3)
(2 . 3)

[18](cons 34 56)
(34 . 56)
```

la notation dite *paire-pointée* est la représentation donnée par *Scheme* aux paires. Cette représentation peut être utilisée comme la constante de définition d'une paire

```
[19](cons 'a '(b . c))
(A B . C)

[20](cons '(a . b) '(c . d))
((A . B) C . D)
```

Sélecteurs

Le premier élément d'une paire est sa *tête*, le deuxième est sa *queue*. Les sélecteurs de tête et de queue sont respectivement **`car`** et **`cdr`**.

```
[21](cons 'a 34)
(A . 34)

[22](car (cons 'a 34))
A

[23](cdr (cons 'a 34))
34

[24](cdr (cons 'a '(b . c)))
(B . C)

[25](car '(12 . 34))
12

[26](cdr '(12 . 34))
34
```

Prédicat d'identification

Le prédicat `pair?` rend `#T` lorsqu'il est appliqué à une paire.

```
[27](pair? (cons 'a 'b))
#T

[28](pair? 4)
()
```

5.5.3 Liste Scheme.

La liste *Scheme* est une chaîne de paires, comme celle que nous avons définie précédemment. On peut donc construire une liste par des applications successives de la fonction **`cons`**. On dit alors que les éléments de la liste sont successivement *cons-sés*.

```
[29](cons 'a (cons 'b (cons 'c ...)))
(A B C ...)
```

La liste vide est représentée par le symbole **`()`**, une liste *bien formée* est donc de la forme

```
[30](cons 'a (cons 'b (cons 'c (cons 'd '()))))
(A B C D)
```

La forme `(A B C D)` est la représentation donnée par *Scheme* aux listes bien formées et cette représentation peut être utilisée comme la constante de définition d'une liste

```
[31](cons 'a '(b c d))
(A B C D)

[32](cons '(a b) '(b c d))
((A B) B C D)
```

Une liste qui ne se termine pas sur la liste vide est dite *mal formée*.

```
[33](cons 'a (cons 'b (cons 'c 'd)))
(A B C . D)
```

La forme `(a b c . d)` est la représentation donnée par *Scheme* aux listes mal formées et cette représentation peut être utilisée comme la constante de définition d'une liste mal formée. On obtient une liste bien formée lorsqu'on cons-se un objet à une liste bien formée et une liste mal formée lorsqu'on cons-se un objet à une liste mal formée.

On peut aussi construire une liste bien formée directement à partir d'un ensemble d'arguments, le constructeur correspondant est la fonction **`list`**.

```
[34](list 'a 'b '(c d) 'd)
(A B (C D) D)

[35](list)
()
```

L'accès aux différents éléments de la liste se fait par des applications successives des fonctions `car` et `cdr`, on est donc très fréquemment amené à combiner ces deux fonctions et on a défini des formes contractées équivalentes.

```
[36](define caar (lambda (l) (car (car l))))
CAAR

[37](define caaar (lambda (l) (car (car (car l)))))
CAAAR

[38](define cadr (lambda (l) (car (cdr l))))
CADR

[39](define cdadr (lambda (l) (cdr (car (cdr l)))))
CDADR
```

On trouve dans l'environnement standard de *Scheme* des fonctions regroupant jusqu'à quatre invocations des fonctions `car` et/ou `cdr` dans une même combinaison.

Les fonctions suivantes (entre autres) sont disponibles dans l'environnement standard de *Scheme* (Cf. tableau 4, page 106).

`(null? liste)`	Permet de tester si `liste` est vide.
`(length liste)`	Rend la longueur de `liste`.
`(append lst1 lst2)`	Rend une liste égale à la concaténation de `lst1` et `lst2`. Lorsque la deuxième liste est mal formée, le résultat est une liste mal formée.
`(reverse liste)`	Rend une liste égale à `liste` renversée.
`(map fonc liste)`	La fonction `fonc` est appliquée successivement à tous les éléments de `liste`. Les résultats de ces applications sont rendus sous la forme d'une liste. Il s'agit donc de la projection de `fonc` sur `liste`.
`(for-each fonc liste)`	La fonction `fonc` est appliquée successivement à tous les éléments de `liste`. Cette fonction ne produit aucun résultat et ne sert qu'à appliquer une fonction produisant un effet de bord à tous les éléments de la liste.

Tab. 4 : Principales fonctions Scheme disponibles concernant les listes.

5.5.4 Vecteur Scheme.

Le vecteur *Scheme* est analogue au tableau dont les éléments sont repérés par leur position (ces «vecteurs» ne sont en fait que des tableaux hétérogènes).

Le constructeur de vecteur est la fonction **`vector`**.

```
[40](vector 'a 'b 'c 'd)
#(A B C D)

[41](vector 16 '(a . b) '(3 5) "Jo")
#(16 (A . B) (3 5) "Jo")
```

La forme `#(a b c d)` est la représentation donnée par *Scheme* aux vecteurs et cette représentation peut être utilisée comme la constante de définition d'un vecteur.

Le sélecteur de composante est la fonction **`vector-ref`**.

```
[42](vector-ref #(a b c d d e) 5)
E
```

Nota: *la numérotation des composantes de vecteur commence à 0.*

La fonction `vector` permet de construire des vecteurs ayant un nombre quelconque de composantes, une fonction permettant de déterminer la taille d'un vecteur est donc souvent utile.

```
[43](vector-length #(a b c d d e))
6
```

Nota: *comme le vecteur est une structure de taille fixe, il n'est pas possible de lui rajouter des composantes. C'est ce qui le distingue d'une liste.*

5.6 `lambda` : le constructeur de fonctions.

La construction d'une fonction est réalisées par l'opérateur `lambda` (Cf. paragraphe 2.4.2, page 20) dont les paramètres sont :

1. la liste de ses paramètres,
2. son expression de définition, c'est à dire l'expression à abstraire.

La forme générale d'une expression *Scheme* utilisant l'opérateur `lambda` est la suivante

```
(lambda <liste des paramètres> <exp-déf>)
```

La fonction ainsi produite n'a pas de nom, elle peut être liée à un nom par l'utilisation d'un `define` ou directement appliquée à ses arguments.

La forme la plus courante d'une expression *Scheme* utilisant l'opérateur `lambda` est la suivante

```
(lambda (p1 p2 p3 ...) <exp-déf>)
```

Il arrive, plus souvent qu'on peut le penser a priori, qu'il soit nécessaire de définir une fonction dont le nombre des arguments est inconnu a priori, on distingue alors deux cas.

Lorsque la fonction peut ne pas avoir d'arguments du tout, on utilise la forme

```
(lambda pars <exp-déf>)
```

dans laquelle `pars` est la liste (éventuellement vide) des arguments auxquels la fonction est appliquée.

Lorsque la fonction a au moins n arguments, on utilise la forme

```
(lambda (p1 p2 ... pn . pars) <exp-déf>)
```

dans laquelle `p1`, `p2` ... et `pn` représentent les n premiers arguments (toujours présents) et `pars` la liste (éventuellement vide) des arguments supplémentaires.

Les expressions suivantes illustrent l'utilisation de l'opérateur `lambda`. Définissons [8] la fonction `somme-carre`

```
[44](define somme-carre
       (lambda (x y)
              (+ (* x x) (* y y))))
SOMME-CARRE
```

Cette ligne traduit l'expression

$$\text{somme-carré} = \lambda\ (x\ y)\ (+\ (*\ x\ x)\ (*\ y\ y))$$

Dans ce contexte (environnement complété par la définition de `somme-carre`), les deux expressions suivantes sont alors équivalentes

```
[45]((lambda (x y) (+ (* x x) (* y y))) 3 4)
25

[46](somme-carre 3 4)
25
```

[8] Dans ce cours, nous utiliserons la notation dite «de l'Indiana» et non pas celle dite «du MIT». En effet, cette notation est identique à celle que nous avons utilisé jusqu'à présent.

Ces deux lignes traduisent les expressions

$$\cdot[2]_{xy} \quad \cdot()_{xx} \quad + \cdot()_{yy} \qquad (3.4$$

et

somme-carré(3,4)

On peut aussi définir des fonctions qui rendent des fonctions. Reprenons, par exemple, la définition de la fonction `derivation` (Cf. paragraphe 2.5, page 26)

```
[47](define derivation
       (lambda (f e)
         (lambda (x)
           (/ (- (f (+ x e))
                 (f x))
              e))))
DERIVATION
```

Définissons, à présent, la fonction dérivée de la fonction `cos` (cosinus) supposée exister

```
[48](define d-cos (derivation cos 0.00001))
D-COS
```

Cette fonction `d-cos` (équivalente, en fait, à une approximation de la fonction sinus) peut alors être utilisée comme n'importe quelle fonction qu'on aurait défini directement

```
[49](d-cos 0)
0.000005

[50](d-cos (/ pi 2))
0.999999
```

En fait, l'opérateur `define` n'est pas obligatoire et nous pourrions réécrire les lignes `[12]`, `[13]` et `[14]` sous la forme

```
[51]((lambda (pi rayon) (* pi rayon rayon)) 3.141592 5)
78.539816
```

Cette forme, bien que totalement cohérente avec la définition de `lambda`, perd tellement en lisibilité qu'on ne l'utilisera jamais. On utilisera donc soit la forme utilisant `define` soit une autre forme que nous allons voir plus loin.

5.7 Expressions conditionnelles & prédicats.

Dans l'état actuel des choses, nous ne savons définir une fonction qu'à l'aide d'une expression évaluable et nous savons que certaines fonctions ne peuvent être définies que «par cas».

Reprenons la définition de la fonction «abs» qui rend la valeur absolue d'un nombre

$$\mathrm{abs}x()x = \ ^{<()} \ 0 \ \rightarrow -x, x$$

Cette définition suppose l'existence de la fonction «si» (Cf. paragraphe 2.2.4, page 12).

5.7.1 `if` et les expressions conditionnelles.

Le langage *Scheme* introduit l'alternative à travers l'opérateur `if`. La forme générale d'une expression *Scheme* utilisant l'opérateur `if` est la suivante

```
(if <prédicat> <conséquent> <alternative>)
```

ou la forme simplifiée

```
(if <predicat> <conséquent>)
```

La valeur rendue par l'évaluation de cette expression est celle de l'expression `<conséquent>` si la valeur de `<prédicat>` est *vrai* et celle de l'expression `<alternative>` (ou *faux* si elle est absente) dans le cas contraire. Nous verrons un peu plus loin que l'évaluation d'une telle expression pose un subtil problème.

5.7.2 Prédicats.

Un prédicat (Cf. paragraphe 2.2.4, page 12) est une expression qui s'évalue à *vrai* (noté `#T`) ou à *faux* (noté `#F` ou `()` ou `nil`). Une telle expression est bâtie à partir des opérateurs primitifs de relation et des opérateurs booléens `and`, `or` et `not`.

Par exemple

```
[52](< 4 3)
()

[53](= 5 (+ 2 3))
#T
```

5.7.3 Exemples d'utilisation d'expressions conditionnelles.

La définition de la fonction «abs» prend alors la forme

```
[54](define abs
       (lambda (x)
         (if (< x 0) (- x) x)))
ABS
```

L'expression `(if (< x 0) (- x) x)` traduit strictement : Quant à la définition de la fonction «fact», elle peut s'écrire

```
[55](define fact
       (lambda (n)
         (if (= n 0)1(* n (fact (- n 1))))))
FACT

[56](fact 6)
720
```

5.8 `quote` et citations.

Le problème de la citation d'un symbole (Cf. paragraphe 3.2.3, page 55) est résolu en *Scheme* à l'aide de l'opérateur `quote` qui cite son argument sans l'évaluer.

Par exemple

```
[57](define x 4)
X

[58]x
4

[59](quote x)
X
```

On utilise, en général, une écriture allégée **strictement équivalente** qui utilise le signe `'`

```
[60]'x
X
```

Les expressions `(quote x)` et `'x` traduisent la citation du symbole que nous écrivions **x**.

5.9 Equivalence opérationnelle.

La traduction de l'équivalence opérationnelle (Cf. paragraphe 4.9, page 84) nécessite, au moins, les fonctions «identique?», «même-valeur?» et «égaux?». Ces trois fonctions, en ce qui concerne les objets de base *Scheme*, sont prédéfinies sous les noms

identique?	`eq?`
même-valeur?	`eqv?`
égaux?	`equal?`

5.10 Formes dérivées.

Les opérateurs `define`, `lambda` et `if` sont suffisants pour définir les fonctions, les types de données et les structures nécessaires à la conception informatique, nous l'avons montré aux chapitres précédents. Cependant, il est commode de dériver de ces 3 opérateurs les opérateurs supplémentaires

- `cond`
- `let`
- `let*`
- `letrec`

5.10.1 `cond` et les expressions gardées.

L'association de prédicats et d'expressions à l'aide d'un ensemble d'alternatives s'appelle des *expressions gardées* (Cf. paragraphe 2.2.4, page 12). *Scheme* permet de simplifier la définition de ces expressions à l'aide de l'opérateur dérivé `cond` qui peut être simplement défini à partir de l'opérateur `if`.

Considérons la définition suivante de la fonction « `abs`»

```
[61](define abs
       (lambda (x)
         (if (< x 0)
             (- x)
             (if (= x 0)
                 0
                 (if (> x 0)
                     x)))))
ABS
```

Elle peut être écrite sous la forme suivante, équivalente mais plus lisible

```
[62](define abs
       (lambda (x)
         (cond ((< x 0)(- x))
               ((= x 0) 0)
               ((> x 0)x)))))
ABS
```

On utilisera très fréquemment des expressions de la forme

```
(cond (<p1> <e1>) (<p2> <e2>) (<p3> <e3>) ... )
```

strictement équivalente à l'expression suivante qui lui sert de définition [9]

```
(if <p1> <e1>
          (if <p2> <e2>
                   (if <p3> <e3>
                            .....)))
```

Traduisons, par exemple, la définition suivante

$$\text{divise?}(n,m) = \begin{cases} (n = m) & \rightarrow vrai, \\ (n > m) & \rightarrow faux, \\ sinon & \rightarrow \text{divise?}(n, m-n) \end{cases}$$

ce qui donne

```
[63](define divise?
       (lambda (n m)
         (cond ((= n m) #T)
               ((> n m) #F)
              (#T(divise? n (- m n))))))
DIVISE?
```

On a placé, en fin de la suite des prédicats, une *sentinelle* toujours *vrai*, il est agréable de donner à cette sentinelle une forme plus conviviale et de la définir dans l'environnement de *Scheme* sous la forme du nom `else` lié à la valeur *vrai*

```
[64](define else #T)
ELSE
```

[9] Ce n'est pas tout à fait exact, mais nous considèrerons qu'il en est strictement ainsi.

Cette définition de `else` permet de modifier légèrement la définition du prédicat «divise?»

```
[65](define divise?
       (lambda (n m)
         (cond ((= n m) #T)
               ((> n m) #F)
              (else(divise? n (- m n))))))
DIVISE?
```

Traduisons, à présent, la fonction

$$\text{rationnel} \lambda n.\lambda m. \begin{cases} m = \textbf{numérateur} \rightarrow n, \\ m = \textbf{dénominateur} \rightarrow d, \\ \textit{sinon} \rightarrow \bot \end{cases}$$

qui prend la forme

```
[66](define rationnel
       (lambda (n d)
         (lambda (m)
           (cond ((eq? m 'numerateur) n)
                 ((eq? m 'denominateur) d)))))
RATIONNEL
```

Nota: *la constante $\bot$ que nous avons fréquemment utilisée est, en général, représentée par la constante* `#F`.

Cette «petite remarque» met en évidence un point faible du langage Scheme. En effet, nous avons constaté qu'il devait exister autant de $\bot$ différents que de types de données et que chacun avait le sien. Ici, le $\bot$ est unique et il sera souvent difficile de lui faire jouer son rôle normal.

5.10.2 `let` et les extensions de l'environnement.

Nous avons vu que l'utilisation de l'opérateur `define` n'était pas toujours obligatoire et nous pouvions réécrire les lignes `[12]`, `[13]` et `[14]` sous la forme

```
[67]((lambda (pi rayon) (* pi rayon rayon)) 3.141592 5)
78.539816
```

Cette forme peut être réécrite également sous la forme

```
[68](let ((pi 3.141592)
          (rayon 5))
       (* pi rayon rayon))
```

Cette forme utilise l'opérateur prédéfini `let`. On utilisera donc très fréquemment des expressions de la forme

```
(let ((<n1> <v1>) (<n2> <v2>) ...) <exp>)
```

qui est **strictement équivalente** à la forme suivante qui lui sert de définition

```
((lambda (<n1> <n2> ...) <exp>) <v1> <v2> ...)
```

Traduisons la prédicat

$$\text{rationnels-égaux?} q = \lambda,()_1\ q_2\ \cdot \begin{bmatrix} \text{soit :} n_1 = \text{numérateur}(q)_1 \\ \quad d_1 = \text{dénominateur}(q)_1 \\ \quad n_2 = \text{numérateur}(q)_2 \\ \quad d_2 = \text{dénominateur}(q)_2 \\ \text{dans :} n\cdot()_1\ d_2 = \cdot()_2\ d_1 \end{bmatrix}$$

qui prend la forme

```
[69](define rationnels-egaux?
       (lambda (q1 q2)
         (let ((n1 (numerateur q1))
               (d1 (denominateur q1))
               (n2 (numerateur q2))
               (d2 (denominateur q2)))
           (= (* n1 d2) (* n2 d1)))))
RATIONNELS-EGAUX?
```

Plutôt que de mémoriser la lambda-définition de l'opérateur `let`, il est plus simple de considérer que toutes les expressions de définition des variables sont évaluées et que les liens ne sont construits qu'ensuite. Dans ces conditions, il est clair que l'ordre dans lequel les variables intermédiaires sont introduites par `let` n'a pas d'importance.

5.10.3 `let*` et les extensions emboitées.

On peut être amené à introduire des variables intermédiaires qui dépendent les unes des autres. C'est ce qui se produit dans l'expression

$$\begin{bmatrix} \text{soit :} x = 4 \\ \text{dans :} \begin{bmatrix} \text{soit :} y x 2 = + \\ \text{dans :} \begin{bmatrix} \text{soit :} z = 2 y x + \\ \text{dans :} \dfrac{x y}{z} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Cette expression se traduit naturellement en *Scheme* sous la forme

```
(let((x 4))
  (let ((y (+ x 2)))
    (let ((z (+(* 2 y) x)))
      (/ (* x y) z))))
```

Cette écriture traduit le fait que les différentes extensions d'environnement créées par les opérateurs `let` successifs sont emboitées. Comme ce cas de figure est assez fréquent, *Scheme* introduit une forme dérivée utilisant un opérateur noté `let*`

```
(let* ((<n1> <v1>) (<n2> <v2>) (<n3> <v3>) ...) <exp>)
```

qui est strictement équivalente à

```
(let ((<n1> <v1>))
  (let ((<n2> <v2>))
    (let ((<n3> <v3>))
```

```
        ...
        <exp>)...)))
```

L'expression précédente peut alors s'écrire sous la forme

```
(let*((x 4))
       (y (+ x 2))
       (z (+(* 2 y) x)))
  (/ (* x y) z))
```

Plutôt que de mémoriser la lambda-définition de l'opérateur `let*`, il est plus simple de considérer qu'il introduit les variables intermédiaires les unes après les autres, dans l'ordre où elles sont indiquées.

5.10.4 `letrec` et les extensions récursives.

Dans le cas où il est nécessaire de définir un environnement récursif, on utilisera l'opérateur prédéfini `letrec`. Ainsi, la traduction de la fonction

$$\text{fact}n = \lambda\ .\left[\begin{array}{l}\textbf{soit-rec :}\ \text{invarFact}fm = \lambda,() \quad .\ ()m = 0 \rightarrow f \\ \quad \text{sinoninvarFact}mf \qquad\qquad \rightarrow \qquad\qquad ,() \cdot \quad m-1 \\ \textbf{dans :}\ \text{invarFact1}n \quad ,()\end{array}\right]$$

utilise l'opérateur `letrec` et prend la forme

```
[70](define fact
       (lambda (n)
         (letrec ((invar-fact (lambda (f m)
                                (if (= n 0)
                                    f
                                    (invar-fact(* f m)
                                               (- m 1))))))
           (invar-fact 1 n))))
FACT
```

On utilisera très fréquemment des expressions de la forme

```
(letrec ((<n1> <f1>) (<n2> <f2>) ...) <exp>)
```

La définition d'un environnement récursif (Cf. paragraphe 2.4.5, page 24) impose que les valeurs liées aux noms soient uniquement des fonctions.

La forme `letrec` permet d'introduire également des définitions mutuellement récursives telles que, par exemple

```
(letrec ((est-pair?   (lambda (n)
                        (if (= n 0)
                            #T
                            (est-impair? (- n 1)))))
         (est-impair? (lambda (n)
                        (if (= n 0)
                            #F
                            (est-pair? (- n 1))))))
  (est-pair? 9))
```

5.11 Scheme et l'application fonctionnelle.

Nous venons de constater que *Scheme* possède les éléments nécessaires à tout langage de programmation puissant

1. des *éléments primitifs* dont l'évaluation est immédiate, **données** et **opérateurs**,
2. une *méthode de composition* des éléments primitifs, les **expressions** et l'**emboîtement des expressions**,
3. une méthode de **dénomination**.

Scheme étant une mécanique, son mécanisme standard d'application est l' **ordre applicatif**. Cependant nous verrons, comme nous nous y attendons, des formes spéciales qui ne peuvent ête évaluées qu'en **ordre normal** (Cf. paragraphe 2.4.5, page 24).

5.12 Evaluation des expressions composées.

Nous pouvons, à présent, approfondir un peu le mécanisme standard d'évaluation en ordre applicatif d'une expression *Scheme* est, à la base, remarquablement simple. L'évaluation est effectuée en deux temps:

1. Evaluer les sous-expressions de l'expression
2. Appliquer la fonction — résultat le l'évaluation de la sous-expression de gauche — à ses arguments — résultat de l'évaluation des autres sous-expressions.

Ce mécanisme est *récursif* puisqu'une sous-expression est de même nature qu'une expression. C'est dire toute la puissance de la récursion qui permet de décrire un processus sans avoir aucune idée a priori de sa complexité.

Ce mécanisme ne met en jeu que les deux fonctions

$\text{Eval}_\sigma[[\textit{expression}]]$	associe, dans l'environnement σ, une valeur à une expression *Scheme*
$\text{Apply}(v_1, v_2 \cdots v_n)$	applique v_1 [i] à $v_2, \cdots v_n$

[i] Ce mécanisme suppose, bien sûr, que la valeur v_1 représente une fonction.

Cette façon de donner un sens à une expression par la définition d'une fonction qui lui attribue une valeur s'appelle en définir une *sémantique sous forme dénotationnelle* . Illustrons ce mécanisme sur un exemple et considérons l'expression

```
(* (+ 2 (* 4 6)) (+ 3 5 7))
```

et évaluons-la

$$\text{Eval}_\sigma[[\texttt{(* (+ 2 (* 4 6)) (+ 3 5 7))}]]$$
$$= \text{Apply}(\text{Eval}_\sigma[[\texttt{*}]], \text{Eval}_\sigma[[\texttt{(+ 2 (* 4 6))}]] \quad \text{Eval}_\sigma[[\texttt{(+ 3 5 7)}]]$$

avec

$$\text{Eval}_\sigma[[(+\ 2\ (*\ 4\ 6))]] = \text{Apply}(\text{Eval}_\sigma[[+]], \text{Eval}_\sigma[[2]]\ \text{Eval}_\sigma[[(*\ 4\ 6)]])$$
$$\text{Eval}_\sigma[[(*\ 4\ 6)]] = \text{Apply}(\text{Eval}_\sigma[[*]], \text{Eval}_\sigma[[4]]\ \text{Eval}_\sigma[[6]])$$
$$\text{Eval}_\sigma[[(+\ 3\ 5\ 7)]] = \text{Eval}_\sigma[[+]](\text{Eval}_\sigma[[3]]\ \text{Eval}_\sigma[[5]], \text{Eval}_\sigma[[7]])$$

Ce mécanisme d'évaluation peut être décrit par l'arbrede la figure 17, page 116.

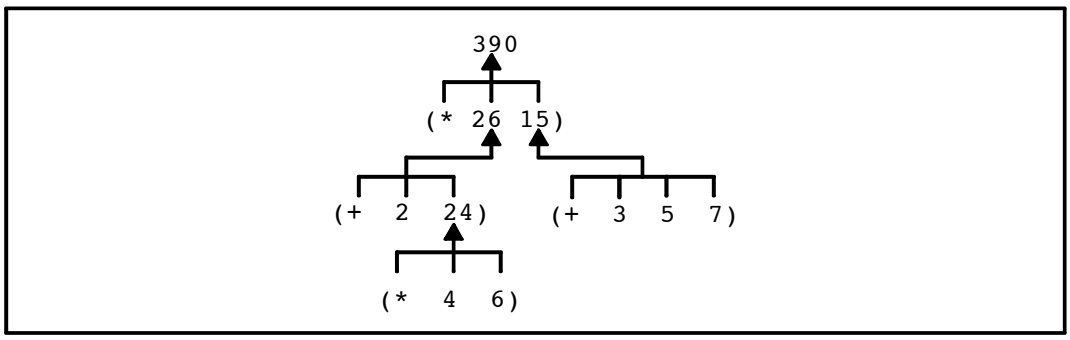


Fig. 17 : Evaluation par accumulation d'arbre.

Les feuilles de l'arbre dénotent soit un opérande soit un opérateur et chaque noeud dénote le résultat de l'évaluation du sous-arbre qui lui est attaché. Cette règle d'évaluation par propagation des valeurs vers le haut s'appelle *processus d'accumulation d'arbre* .

Lorsque le processus d'évaluation (parcours de l'arbre d'évaluation) atteint une feuille de l'arbre, il tombe sur des *opérandes et des opérateurs primitifs* dont les règles d'évaluation sont définies de la manière suivante :

1. la valeur d'une *chaîne de chiffres* est le *nombre* qu'elle représente,
2. la valeur d'un *opérateur prédéfini* est *le mécanisme* qui réalise l'opération correspondante,
3. la valeur d'un *nom* est la *valeur liée* à ce nom dans l'environnement σ courant.

Dans l'exemple qui suit, nous allons pousser le bouchon un peu plus loin. Comme le concept que nous allons introduire maintenant est un peu étrange, n'insistez pas si cela ne vous inspire pas, nous y reviendrons plus tard.

Reprenons l'évaluation de l'expression

```
(* (+ 2 (* 4 6)) (+ 3 5 7))
```

Pour pouvoir pousser l'évaluation plus loin il est nécessaire d'introduire quelques définitions.

Posons, par exemple

$\text{Eval}_\sigma[[+]]$ = **add**	$\text{Eval}_\sigma[[3]]$ = **3**	$\text{Eval}_\sigma[[6]]$ = **6**
$\text{Eval}_\sigma[[*]]$ = **mult**	$\text{Eval}_\sigma[[4]]$ = **4**	$\text{Eval}_\sigma[[7]]$ = **7**
$\text{Eval}_\sigma[[2]]$ = **2**	$\text{Eval}_\sigma[[5]]$ = **5**	...

Les expressions en gras représentent des concepts (formes sémantiques) tandis que les expressions entre double crochets représentent notre façon de les représenter (formes syntaxi-

ques). Dans ce tableau, par exemple, **add** représente le mécanisme d'addition tandis que `+` est notre notation de l'addition, **6** représente le nombre 6 tandis que `6` représente le nom donné à ce nombre [10] (on aurait aussi bien pu le représenter par `20` en utilisant un système de numération en base 3).

Il vient alors

$$\begin{aligned}&\mathrm{Eval}_\sigma[[\texttt{(* (+ 2 (* 4 6)) (+ 3 5 7))}]]\\ &= \mathrm{Apply}(\textbf{mult}, \mathrm{Eval}_\sigma[[\texttt{(+ 2 (* 4 6))}]], \mathrm{Apply}(\textbf{add}, \mathrm{Eval}_\sigma[[\texttt{3}]], \mathrm{Eval}_\sigma[[\texttt{5}]], \mathrm{Eval}_\sigma[[\texttt{7}]]))\\ &= \mathrm{Apply}(\textbf{mult}, \mathrm{Apply}(\textbf{add}, \mathrm{Eval}_\sigma[[\texttt{2}]], \mathrm{Apply}(\textbf{mult}, \mathrm{Eval}_\sigma[[\texttt{4}]], \mathrm{Eval}_\sigma[[\texttt{6}]])), \mathrm{Apply}(\textbf{add}, \mathrm{Eval}_\sigma[[\texttt{3}]], \mathrm{Eval}_\sigma[[\texttt{5}]], \mathrm{Eval}_\sigma[[\texttt{7}]]))\\ &= \mathrm{Apply}(\textbf{mult}, \mathrm{Apply}(\textbf{add}, 2, \mathrm{Apply}(\textbf{mult}, 4, 6)), \mathrm{Apply}(\textbf{add}, 3, 5, 7))\end{aligned}$$

Essayons d'aller plus loin dans la compréhension de l'évaluation d'une expression. La fonction $\mathrm{Apply}(\cdots)$ est le mécanisme qui permet, étant donnée un opérateur et ses arguments, d'effectuer le calcul représenté par la fonction associée à cet opérateur.

Posons alors

$$\begin{aligned}\mathrm{Apply}(\textbf{mult}, x, y, \cdots) &= \mathrm{mult}(x, y, \cdots)\\ \mathrm{Apply}(\textbf{add}, x, y, \cdots) &= \mathrm{add}(x, y, \cdots)\end{aligned}$$

Il vient finalement

$$\begin{aligned}&\mathrm{Eval}_\sigma[[\texttt{(* (+ 2 (* 4 6)) (+ 3 5 7))}]]\\ &= \mathrm{mult}(\mathrm{add}(2, \mathrm{mult}(4, 6)), \mathrm{add}(3, 5, 7))\end{aligned}$$

En conclusion, nous venons de découvrir deux choses remarquables :

1. La fonction $\mathrm{Eval}_\sigma[[\cdots]]$ concerne surtout les opérateurs et les opérandes primitifs et les variables définies dans l'environnement σ auxquels elle attribue une valeur.
2. La fonction $\mathrm{Apply}(\cdots)$ concerne le mécanisme de fonctionnement des opérateurs primitifs et surtout celui des fonctions définies par utilisation de `lambda`. Elle remplace ses paramètres par la valeur de ses arguments puis met en route le mécanisme de l'opérateur primitif ou l'évaluation de l'expression de définition de la fonction.

Nous reviendrons longuement sur la définition des fonctions et sur la fonction $\mathrm{Eval}_\sigma[[\cdots]]$ et $\mathrm{Apply}(\cdots)$ dont nous n'avons fait qu'ébaucher le rôle.

Ces deux fonctions définissent, à elles deux, toute la sémantique du langage.

5.13 Formes spéciales.

Que se passerait-il si les expressions contenant les opérateurs `define`, `lambda`, `if`... étaient évaluées en ordre applicatif?

Pour répondre à cette question, considérons l'expression

```
(define taille 10)
```

Son évaluation en ordre applicatif se ferait en deux étapes :

1. évaluation de `define`, de `taille` et de `10`,
2. application de $\mathrm{Eval}_\sigma[[\texttt{define}]]$ à $\mathrm{Eval}_\sigma[[\texttt{taille}]]$ $\mathrm{Eval}_\sigma[[10]]$

L'évaluation de `define` rend le mécanisme associé et l'évaluation de `10` rend le nombre **10**. L'évaluation de `taille` devrait rendre la valeur qui a été liée au symbole `taille`.

[10] Les anglo-américains distinguent ainsi le *number* (le nombre) du *numeral* (notation du nombre).

Mais ce lien n'aurait pu être établi que lors de la mise en œuvre de l'opérateur `define`. Cette évaluation ne peut donc qu'échouer.

Pour qu'elle réussisse, il faut que le mécanisme associé à l'opérateur `define` soit mis en œuvre avant de tenter l'évaluation de l'argument `taille`. Il faut donc que l'expression qui contient un `define` soit évaluée **en ordre normal**.

Considérons à présent l'expression

```
(cond ((> n 0) n)
      ((= n 0) 0)
      ((< n 0) (- n)))
```

Son évaluation en ordre applicatif se fait en deux étapes:

1. évaluation de `cond`, de `((> n 0) n)`, de `((= n 0) 0)` et de `((<n0)(-n))`,
2. application de $Eval_\sigma[[cond]]$ à $Eval_\sigma[[((> n 0) n)]]$ $Eval_\sigma[[((= n 0) n)]]$ et à $Eval_\sigma[[((< n 0) (- n))]]$

L'évaluation de `cond` rend le mécanisme associé, l'évaluation en ordre applicatif de l'expression

```
((> n 0) n)
```

se fait également en deux étapes:

1. évaluation de `(> n 0)` et de `n`,
2. application de $Eval_\sigma[[(> n 0)]]$ à $Eval_\sigma[[n]]$

Les évaluations réussissent, par contre l'application échoue car $Eval_\sigma[[(> n 0)]]$ vaut *vrai* ou *faux* valeurs qui soit ne sont pas considérées comme des fonctions qui peuvent être appliquées, soit (Cf. paragraphe 3.1.3, page 52) sont considérées comme des fonctions attendant **deux** arguments.

Pour que l'évaluation d'une expression contenant `cond` réussisse, il faut que le mécanisme associé à `cond` soient mis en œuvre avant de tenter l'évaluation des arguments de `cond`. Il faut donc que l'expression qui contient un `cond` soit évaluée également **en ordre normal**.

On appellera *forme standard* une expression *Scheme* qui peut être évaluée en ordre applicatif et *forme spéciale* une expression *Scheme* qui doit être évaluée en ordre normal.

Sujet de réflexion délicat mais passionnant...

La raison pour laquelle `if` est nécessairement une forme spéciale est plus délicate à mettre en évidence. Supposons qu'un «puriste francophone» tente de franciser *Scheme* et définisse la fonction `si`

```
[71](define si(lambda p e1 e2)(if p e1 e2)))
```

Il utilise alors cet opérateur pour définir la fonction `max`

```
[72](define max
       (lambda (x y)
         (si (> x y) x y)))
MAX
```

puis demande l'évaluation desexpressions

```
[73](max 4 5)
5

[74](max 7 8)
8

[75](max 9 9)
9
```

Comme tout semble marcher normalement, il utilise cet opérateur pour redéfinir la fonction `fact`

```
[76](define fact
       (lambda (n)
         (si (= n 0)
             1
             (* n (fact (- n 1))))))
FACT
```

puis demande l'évaluation del'expression

```
[77](fact 6)
```

Mais que se passe-t-il donc? Il est clair que les formes spéciales sont réellement spéciales!

5.14 Où l'on reparle de `define`.

Le mécanisme d'abstraction de base est la définition des variables qui consiste à donner un nom à des valeurs (fonctions ou données) et à les manipuler sous ce nom. Nous avons, ainsi, rencontré les opérateurs `define`, `let`, `let*` et `letrec` pour définir des variables dans des environnements simples ou récursifs.

L'opérateur `define`, que nous avons introduit en premier pour des raisons de commodité, n'est pas nécessaire et il n'est en fait qu'une forme dérivée curieuse de `letrec`. Expliquons-nous sur quelques exemples.

Considérons les trois expressions

```
[78](define foo 4)
FOO

[79](define baz 5)
BAZ

[80](+ foo baz)
9
```

Elles sont strictement équivalentes à l'expression unique

```
[81](let ((foo 4)
          (baz 5))
       (+ foo baz))
9
```

Ainsi, `define` n'est qu'une manière interactive de définir l'environnement qui donne un sens à une expression à évaluer.

Si maintenant on considère les deux expressions

```
[82](define fact
       (lambda (n)
         (if(= n 1)
             1
             (* n (fact (- n 1))))))
FACT

[83](fact 6)
720
```

Elles sont strictement équivalentes à l'expression unique

```
[84](letrec ((fact (lambda (n)
                     (if (= n 1)
                          1
                          (* n (fact (- n 1)))))))
       (fact 6))
720
```

On constate, alors, que l'environnement créé interactivement, par l'intérieur, à l'aide de `define` est un environnement récursif tel que celui créé par `letrec`.

Dans ces conditions, l'utilisation de `define` ne se justifie réellement que dans la phase de développement et de mise au point d'une application. Nous verrons, cependant, qu'on utilise `define` pour étendre l'environnement de l'interpréte *Scheme*, alors qu'une application se traduira par une expression unique.

Nous reviendrons plus loin sur cet aspect fondamental de la programmation fonctionnelle et la construction des **paquetages** (Cf. paragraphe 3.3, page 60).

5.15 Effets de bord.

Le langage *Scheme* a été conçu d'une part comme un support de réflexion utilisable exactement comme l'écriture informelle des chapitres précédents et d'autre part comme un langage mécanisable. C'est cette deuxième utilisation qui justifie l'appellation de «langage de programmation».

Dans ce cas, il peut être nécessaire de disposer de fonctions curieuses utilisée non pas pour la valeur qu'elles rendent, mais pour un effet secondaire. Nous avons rencontré cette nécessité lorsque nous avons introduit l'opérateur `define` et au cours des exercices **E-14**, **E-15**, **E-18** et **E-19** dans le cadre dequels nous avons supposé l'existence des fonction «affiche» et «émettre».

Trois «fonctions» prédéfinies sont souvent utilisées poour provoquer un affichage à l'écran de la machine utilisée pour mécaniser les évaluations des expressions *Scheme*

`display`	provoque l'affichage de la valeur de son argument.
`newline`	provoque «un passage à la ligne».
`writeln`	provoque l'affichage de la valeur de ses arguments puis provoque «un passage à la ligne».

Dans le cas de telles «fonctions[11]», comme l'ordre des choses ne dépend plus d'une dépendance fonctionnelle, mais d'une «perception extérieure» il doit être explicitement indiqué. C'est le rôle de la forme spéciale `begin`. Cette fonction évalue ses arguments de la gauche vers la droite et rend la valeur de celui situé le plus à droite.

Nota: *la fonction* `begin` *est la seule fonction qui garantit l'ordre dans lequel ses arguments sont évalués. C'est en ce sens que la forme* `begin` *est spéciale.*

```
[85](begin (display 4)
           (display 5))
45
[86](begin (writeln "3 + 4 = " (+ 3 4))
           (writeln "3 - 4 = " (- 3 4)))
3 + 4 = 5
3 - 4 = -1
```

les formes spéciales `lambda`, `cond`, `let`, `let*` et `letrec` incluent un `begin` implicite qu'il n'est donc pas nécessaire de rajouter.

5.16 Exercices.

E-51 Donner la transcription en écriture *Scheme* des expressions suivantes

$345+$ $\qquad$ $\lambda_x \cdot 2x$

$345\ +$ $\qquad$ $345() +$

$\frac{3 \cdot x}{y}$ $\qquad$ $\frac{3}{xy}$

fact6() $\qquad$ carré$x=\ \lambda\ \cdot xx$

E-52 Donner la transcription en écriture *Scheme* des formes suivantes

$$\begin{bmatrix} \text{soit :} x\ = 2 \\ \qquad y\ = 3 \\ \text{dans :} xy + \end{bmatrix} \qquad \begin{bmatrix} \text{soit :} x\ = 2 \\ \text{dans :} \begin{bmatrix} \text{soit :} y\ = 3x \\ \text{dans :} \begin{bmatrix} \text{soit :} zxy = + \\ \text{dans :} xyz++ \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

E-53 Donner le définition *Scheme* de la fonction `fact` dont la définition formelle est la suivante

$$\text{fact}n= \lambda\ \cdot \begin{bmatrix} \text{soit-rec :} Ffm= \lambda\ \cdot () m = 1\ \rightarrow fFmf() \cdot\ m-1 \\ \text{dans :} F1n\ , () \end{bmatrix}$$

E-54 Déterminer la valeur des expressions suivantes

```
(cons 'car 'cdr)
(list 'ceci '(est fou))
```

[11] Dans l'environnement *Scheme* présenté ici, la fonction `display` rend une valeur non affichable tandis que les fonctions `newline` et `writeln` rendent `#F`.

```
(cons 'ceci '(est-il fou?))
(quote (+ 2 3))
cons
(quote (quote cons))
(car (quote (quote cons)))
((car (list + - * /)) 2 3)
```

E-55 Etant donnée la liste `'((a b) (c d))`, définir les expressions *Scheme* dont les valeurs sont respectivement `a`, `b`, `c` et `d`.

E-56 Décrire pas à pas l'évaluation des expressions suivantes

```
((car (cdr (list + - * /))) 17 5)
(cons (quote -) (cdr (quote (+ b c))))
(cdr (cdr '(a b c)))
(cons 'd (cddr '(a b c d e f)))
```

E-57 Simplifier les expressions suivantes en éliminant les évaluations redondantes

```
(+ (- (* 3 a) b) (+ (* 3 a) b))
(cons (car (list a b c)) (cdr (list a b c)))
```

E-58 Evaluer l'expression suivante

```
(let ((x 9))
  (* x (let ((x (/ x 3)))
         (+ x x))))
```

E-59 Détecter puis renommer les variables liées qui engendre une collision de nom dans les expressions suivantes, puis les évaluer

```
(let ((x 'a)
      (y 'b))
  (list (let ((x 'c)) (cons x y))
        (let ((y 'd)) (cons x y))))

(let ((x '((a b) c)))
  (cons (let ((x (cdr x)))(car x))
        (let ((x (car x)))
          (cons (let ((x (cdr x)))(car x))
                (cons (let ((x (car x)))x)
                      (cdr x))))))
```

E-60 Evaluer les expressions suivantes

```
(let ((f (lambda (x) x))) (f 'a))
(let ((f (lambda x x)))(f 'a 'b 'c))
(let ((f (lambda (x . y) x)))(f 'a 'b 'c))
(let ((f (lambda (x . y) y)))(f 'a 'b 'c))
```

E-61 Donner une définition du constructeur de liste `list`.

Nota: *Cet exercice est un clin d'oeil, il est très simple.*

E-62 Donner la liste des variables libres contenues dans les expressions suivantes

```
(lambda (f x) (f x))
(lambda (x) (+ x x))
(lambda (x y) (f x y))
(lambda (x) (cons x (f x y)))
(lambda (x) (let ((y (cons x y))) (list x y z)))
```

Nota: `lambda` *ne peut pas être considéré comme une variable libre*.

E-63 Considérons la définition suivante

```
(define ah! (lambda (f x) (f x)))
```

puis évaluer l'expression

```
(ah! ah!)
```

E-64 Donner la définition *Scheme* des fonctions suivantes:

1. fonction `carre` qui élève un nombre `x` au carré.
2. fonction `cube` qui élève un nombre `x` au cube.
3. fonction `quatrieme` qui élève un nombre `x` à la puissance quatrième.
4. fonction `moyenne` qui prend la moyenne de deux nombres `x` et `y`.

E-65 Donner la définition *Scheme* du prédicat `divise?` qui rend `#T` si *n* divise *y* et `#F` autrement.

E-66 Donner la définition *Scheme* de la fonction `reste` qui rend le reste de la division de `m` par `n`.

E-67 Donner la définition *Scheme* de la fonction `quotient` qui rend le quotient de la division du nombre entier positif m par le nombre entier positif `n`.

E-68 Donner la définition *Scheme* du prédicat `premiers?` qui rend `#T` si les deux nombres `n` et `m` sont premiers entre eux (ne se divisent pas) et `#F` autrement.

E-69 Supposons que la multiplication ne soit pas un opérateur primitif de *Scheme*. Définir une fonction `mult` qui multiplie deux nombres entiers positifs `n` et `m` à partir de l'addition supposée primitive.

E-70 Définir la fonction `pgcd` qui rend le PGCD de deux nombres entiers positifs.

E-71 Donner la définition *Scheme* de la fonction `monnaie` de l'exercice **E-14** qui détermine la meilleure façon de construire une somme donnée `s` à partir d'un stock, supposé suffisant, de pièces de 20F, de 10F, de 5F, de 2F et de 1F.

Nota: *On provoquera l'affichage des résultats «intéressants» de l'évaluation de la fonction* `monnaie`.

E-72 Traiter l'exercice précédent en supposant maintenant que le stock de pièces est limité à`m20` pièces de 20F, `m10` pièces de 10F, `m5` pièces de 5F, `m2` pièces de 2F et à `m1` pièces de 1F.

E-73 Vous pouvez imaginer que le processus d'évaluation associé à la définition d'une fonction n'est pas toujours celui qu'on croyait — en d'autres termes il y a un *bug*. Il est alors commode d'afficher la valeur de quelques stades intermédiaires de l'évaluation. La fonction `display` est inadaptée pour jouer ce rôle car elle ne peut pas s'intercaler dans une évaluation.

1. Définir une fonction `afficher` qui pourrait s'intercaler n'importe où dans une évaluation et permettrait, par exemple, d'écrire:

   ```
   (* (+ 3 4) (afficher (+ 7 8)))
   ```

2. En déduire une version tracée, `fact-tracee`, de la fonction factorielle `fact`.
3. Quelle est donc la définition mathématique de la fonction `afficher` ?

Nota: *Cette dernière question un peu délicate. Elle nécessite une réflexion profonde sur la notion d'effet de bord*.

E-74 Utiliser la fonction `map` (Cf. tableau 4, page 106) pour définir la fonction `transpose` qui prend une liste de paires et qui rend une paire de listes.

ex:

```
(transpose '((a . 1) (b . 2) (c . 3))) ⇒ ((a b c) 1 2 3)
```

E-75 **Ce problème est (très ?) difficile**, il constitue une suite à l'exercice **E-21**.

1. Déduire des résultats de l'exercice **E-21** une définition formelle purement fonctionnelle d'une expression de la forme

$$\left[\begin{array}{l}\textbf{soit-rec :} fx = \lambda \ \cdot \cdots f \cdots \\ \textbf{dans :} f \quad ()\cdot\cdot\end{array}\right]$$

2. On suppose que l'évaluation des expression est effectuée, comme dans *Scheme*, en **ordre applicatif**. Donner une définition *Scheme* d'une fonction pouvant jouer le rôle de la fonction *Y* précédente.

   Nota: *cette question est réellement très difficile*.

6. Les Structures mutables

Le problème qui se pose maintenant à nous est un problème de fond qui va nous obliger, dans certains cas, à changer notre point de vue sur la notion de **valeur**. Il se pose dès lors qu'on est amené à distinguer les choses en soi des choses pour organiser les choses ou plus exactement de décrire nos relations avec les choses.

La représentation des objets réels (une pomme, une voiture etc.) utilise toujours un *modèle* construit à partir d'une structure qui regroupe des caractéristiques qui nous semble *pertinentes* (taille, poids, couleur etc.). La construction de ce modèle dépend plus du point de vue que nous avons sur l'objet que de l'objet lui-même. Les philosophes diraient probablement que nous décrivons non pas l'objet, mais la perception que nous en avons.

Prenons un exemple très concret pour illustrer cette nouvelle difficulté et essayons de décrire une voiture. Une voiture prend naissance chez son *constructeur*, déroule son existence chez son *propriétaire* et achève sa vie chez un *casseur*. Il est manifeste que ces trois personnes ne peuvent pas avoir le même point de vue sur les voitures.

Le constructeur

Il assemble la voiture à partir de pièces détachées que lui fournissent ses sous-traitants ou qu'il fabrique lui-même. Lorsque la voiture est assemblée, il la perd de vue. Un constructeur ne voit donc jamais que des voitures neuves, de telles voitures sont «hors du temps». Ce sont donc des objets en soi pouvant être modélisés par une structure du type de celles que nous venons de voir.

L'utilisateur

Il prend en charge la voiture que lui «confie» le constructeur et l'utilise tout au long de sa vie (celle de la voiture). Cette voiture accompagne l'utilisateur pendant une tranche importante de sa vie (celle de l'utilisateur), et celui-ci aura besoin de modéliser son vieillissement à l'aide de caractéristiques nécessairement changeantes (son âge, son kilométrage etc.) et

ce sont finalement ces caractéristiques qui l'intéressent le plus, elles représentent ce qu'on va appeler l'*état* de la voiture.

Le casseur

Il récupère une voiture dans un état variable. Cet état n'est pas de même nature que celui qui intéresse un utilisateur. En effet, pour le casseur, la voiture n'est qu'un source de pièces détachées d'occasion et son état représente les pièces détachées qu'on peut encore en tirer.

Le concept de fonction que nous avons utilisé jusqu'à présent semble, a priori, peu propice à la description des états. Cette impression est, à la fois, vraie et fausse et nous allons, à présent, tenter d'éclairer les choses en analysant en détail un exemple simple qui nous touche de près: notre compte bancaire.

6.1 Etat & Affectation.

On peut énoncer ce problème de la manière suivante: si la valeur à associer à l'expression `(+ 3 4)` est bien définie, quelle valeur peut-on associer à notre compte bancaire? son solde bien sûr !

En fait ce compte bancaire n'est jamais évalué définitivement. Et même plus, il cessera de nous intéresser dès l'instant que nous considérerons sa valeur comme définitivement fixée (à la clôture du compte).

Cela traduit le fait que l'aspect intemporel de la définition d'un objet ne nous convient pas toujours car nous ne nous intéressons pas uniquement à un résultat final mais bien au processus qui permet d'y parvenir. Il peut même se produire que le résultat final ne nous intéresse pas du tout et nous préférons, dans ce cas, définir l' **histoire d'un objet**.

Considérons la gestion de notre compte bancaire et soit $solde()$ la fonction définissant l'histoire du solde de notre compte notée $\langle s_0, s_1, s_2, \cdots, s_n \rangle$. Cette fonction dépend de l'histoire de toutes les opérations qui ont été effectuées sur ce compte et la définition de cette fonction peut être écrite sous la forme

$$\langle s_0, s_1, s_2, \cdots, s_n \rangle = solde(\langle o_0, o_1, o_2, \cdots, o_n \rangle)$$

On aurait pu poser ce problème lorsque nous avons introduit les listes et définir le constructeur de listes de telle sorte que

$$[1,2,3] = cons(1, cons(2, cons(3, liste\text{-}vide))), \quad \{\}() = \langle \rangle, \cdots$$

On pourrait définir, également, la fonction $solde()$ de telle sorte que

$$\langle 0, 20, 10, 5, 20, \cdots \rangle = solde(\langle 0, 20, 10, 5, -15, \cdots \rangle)$$

Cette façon de penser est très riche sur un plan théorique mais peu utilisée sur un plan pratique (au moins dans l'état actuel des langages informatiques courants). Nous préférerons prendre l'approche plus pragmatique de la notion d'état changeant. Cette approche est cependant moins rigoureuse et elle nous fait perdre tout le bénéfice du support mathématique sur lequel nous pouvions nous appuyer jusqu'à présent. Il sera donc très important de bien en apprécier le «prix à payer».

> La difficulté que nous venons de soulever n'est pas nouvelle, elle est de même nature que celle qu'on rencontrée lorsqu'on tente de définir précisément ce qu'est le mouvement. Toutes les tentatives d'élaboration d'une telle définition ont échoué et nous nous contentons, en fait, de la certitude que nous avons que nous parlons à peu près tous de la même chose.

Une des premières mise en évidence de cet échec est bien connue sous la forme du «paradoxe d'Achille et de la tortue» du à Zénon d'Elée. Ce paradoxe consiste à nier l'existence du mouvement en bâtissant un raisonnement «prouvant» qu'il est impossible à Achille de rattraper la tortue — lorsque qu'Achille est à 10 mètres de la tortue, il se met à courir. Il parcourt ces 10 mètres, mais pendant le même temps, la tortue a avancé et Achille ne l'a pas rattrapé. Achille aura donc toujours du chemin à faire chaque fois qu'il aura parcouru le chemin qui aurait dû lui permettre d'atteindre la tortue.

Ce raisonnement, qui heurte manifestement notre bon sens, n'a pas encore été réfuté de manière convaincante, c'est à dire en n'introduisant pas un concept permettant de construire des paradoxes de même nature. Il met clairement en évidence la difficulté qu'il y a à passer d'une vision statique des choses à une vision dynamique.

6.2 Variables d'Etat & Affectation.

Parmi tous les systèmes possédant une histoire, il en existe une catégorie dont l'importance sera fondamentale pour nous, les **systèmes d'état**. Ces systèmes sont tels qu'il est possible de résumer tout leur passé à l'aide d'un nombre fini de variables. Ces variables sont appelées *variables d'état* et leur valeur, qui résume tout le passé du système, définit son *état*.

Dans le cas de tels systèmes, il est très commode de définir une *fonction de transition* décrivant comment on passe d'un état à un autre selon la *valeur actuelle* de ses paramètres (variables d'état et variables d'action).

Ainsi, la construction d'une liste peut être décrite par le modèle d'état

$$l \leftarrow \perp$$
$$l^+ \leftarrow \mathrm{cons}(x, l),$$

et celle du gestionnaire de compte bancaire par

$$s \leftarrow 0$$
$$s^+ \leftarrow s + x$$

Les variables l et s sont les variables qui représentent le présent du système et résument tout le passé, les variables l^+ et s^+ représentent le futur immédiat de ces mêmes variables. L'exposant + dénote le fait que les variables d'état changent de valeur à chaque invocation de la fonction de transition.

On remarque immédiatement que ces variables sont associées à deux types de manipulations

1. on peut les *définir* (opération dénotée =) à partir d'une valeur évaluée indépendamment de leur valeur antérieure — on les *initialise*.
2. on leur *affecte* (opération dénotée $\leftarrow$) une valeur qui dépend de l'état antérieur et de la valeur des paramètres de la fonction de transition.

Ainsi, si on considère un symbole et une valeur, on peut effectuer les deux opérations suivantes :

1. on peut **associer le symbole à la valeur**. Il s'agit du mécanisme d'abstraction par nommage que nous avons introduit au tout début de ce cours. Nous avons appelé ce mécanisme *définition d'une variable*.
2. on peut **associer la valeur au symbole**. Il s'agit du mécanisme d'affectation que nous venons d'introduire. Nous appellerons ce mécanisme *affectation d'une variable*.

Si les opérateurs de définition de variables de *Scheme* sont `define`, `let`, `let*` et `letrec`, l'opérateur d'affectation est `set!`. Une expression impliquant l'opérateur `set!` [1] est de la forme

```
(set! <symbole> <valeur>)
```

Il est clair que l'opérateur `set!` est nécessairement une forme spéciale. En effet, si on considère l'évaluation des expressions suivantes [2]

```
(define foo 6)
foo                                    ⇒ 6
(set! foo 7)                           ⇒ 7
foo                                    ⇒ 7
```

le symbole `foo` ne doit pas être évalué avant l'application de `set!`.

On peut maintenant envisager la conception d'une application de gestion de compte bancaire. Soit `solde` l'état du compte lorsqu'on commence à s'y intéresser. On peut retirer de l'argent

```
(define retirer
  (lambda (x) (set! solde (- solde x))))
```

et en déposer

```
(define deposer
  (lambda (x) (set! solde (+ solde x)))
```

On peut alors évaluer les expression suivantes:

```
(define solde 0)
```

ouvre le compte bancaire, on fait ensuite un versement de 1000 puis on effectue quelques opérations :

```
(deposer 1000)                         ⇒ 1000
(retirer 200)                          ⇒ 800
(deposer 50)                           ⇒ 850
(retirer 200)                          ⇒ 650
(deposer 50)                           ⇒ 700
```

La première remarque qui s'impose est que les entités `retirer` et `deposer` ne sont pas des fonctions puisque les évaluations ci-dessus montrent que leur valeur ne dépend pas que de la valeur de leur argument, elles produisent un *effet de bord*, on les appellera plus vaguement des *procédures*.

Le compte bancaire précédent est peu intéressant car la variable `solde` est unique et on ne peut pas ouvrir plusieurs comptes simultanément. Nous allons donc généraliser les choses en définissant un type de donnée **Compte-bancaire**. Nous inaugurons là une structure de données curieuse car elle rassemble une donnée, `solde`, et deux procédures, `deposer` et `retirer`. Le constructeur peut être celui décrit figure 18, page 130.

Et voici quelques exemples d'opération sur un compte bancaire.

Ouverture d'un compte...

```
(define alonzo-church (compte-bancaire))
```

[1] Le point d'exclamation postfixant le nom`set!` indique qu'il s'agit d'une opération d'affectation. Prononcer "set-bang".

[2] Le signe ⇒ dénote la valeur de l'expression à gauche.

```
(define compte-bancaire
  (lambda ()
    (let* ((solde 0)
           (retirer (lambda (x)(set! solde (- solde x))))
           (deposer (lambda (x)(set! solde (+ solde x)))))
      (lambda (msg)
        (cond ((eq? msg 'consulter)solde)
              ((eq? msg 'retirer) retirer)
              ((eq? msg 'deposer) deposer))))))

(define solde-de(lambda (cb) (cb 'consulter)))

(define deposer-sur(lambda (cb) (cb 'deposer)))

(define retirer-de(lambda (cb) (cb 'retirer)))
```

Fig. 18 : Constructeur et sélecteurs pour un compte bancaire. Notons l'utilisation de `set!`

Quelques versements...

```
((deposer-sur alonzo-chruch) 1000)    ⇒ 1000
((deposer-sur alonzo-church) 200)     ⇒ 1200
```

Consultation du solde...

```
(solde-de alonzo-church)              ⇒ 1200
```

Quelques retraits...

```
((retirer-de alonzo-church) 500)      ⇒ 700
((retirer-de alonzo-church) 300)      ⇒ 400
```

Cette structure est inhabituelle en ce sens que si le sélecteur `solde-de` est identique aux sélecteurs que nous avons définis jusqu'à présent, les deux autres sélecteurs, `retirer-de` et `deposer-sur`, rendent des procédures de modification de l'état d'une donnée de ce type. De tels sélecteurs sont donc appelés des *modificateurs*.

Cette structure est appelée une **structure de données mutables** ou plus brièvement (bien que de façon un peu impropre) un **objet**.

Une structure (type) de donnée mutable est donc définie par :

1. (éventuellement) les **constantes** de ce type,
2. le **constructeur** des données du type,
3. des **sélecteurs**,
4. des **modificateurs**.

6.3 Egalité & Identité.

Considérons les deux comptes bancaires

```
(define alonzo-church(compte-bancaire))
(define stephen-kleene (compte-bancaire))
```

et rapprochons ces deux expressions des deux expressions suivantes

```
(define rat1 (rationnel 3 4))
(define rat2 (rationnel 3 4))
```

Les deux nombres rationnels `rat1` et `rat2` sont égaux (opérationnellement équivalents), par contre, les deux symboles `alonzo-church` et `stephen-kleene` ne peuvent manifestement pas être utilisés de manière interchangeable bien qu'ils représentent tous les deux le résultat de l'évaluation de la même fonction

```
((deposer-sur alonzo-church) 500)     ⇒ 500
((retirer-de alonzo-church) 200)      ⇒ 300
((retirer-de stephen-kleene) 200)     ⇒ -200
```

Il est donc clair que ces deux objets sont différents puisqu'ils ne produisent pas le même effet.

On aurait pu définir un compte joint

```
(define alonzo-church (compte-bancaire))
(define stephen-kleene alonzo-church)
```

c'est à dire donner deux noms différents au même compte.

> La possibilité d'accéder à un même objet par plusieurs noms est appelée pseudonymie. Une source d'erreurs commune est d'oublier que des modifications sur un objet peuvent induire, par effet de bord, des modifications sur un "autre" objet. ces erreurs sont si difficiles à détecter et à analyser que certains ont proposé que les langages de programmation interdisent les effets de bord et la pseudonymie.

Toute opération effectuée sous un nom sera perçue simultanément sous l'autre nom, en fait les comptes `alonzo-church` et `stephen-kleene` sont, dans ce cas, *identiques*.

Cette difficulté à définir une égalité n'est pas due à un défaut d'expression de notre langage mais bien de l'ambiguité intrinsèque de la notion d'objet. Il y a des objets intemporels qui, lorsqu'ils ont été perçus comme égaux sont opérationnellement équivalents et le resteront pour l'éternité et des objets ayant une histoire. Bien sûr, on pourrait dire que deux objets sont égaux s'ils ont été créés par le même constructeur et s'ils sont dans le même état, mais cette égalité ne nous permettrait pas, pour autant, de les considérer comme opérationnellement équivalents et de les utiliser de manière interchangeable.

6.4 «Prix à payer» pour l'Affectation.

Le «prix à payer» pour l'affectation se traduit par deux problèmes qui se posent dès l'instant qu'on utilise l'affectation dans la définition d'une expression. Le premier de ces deux problèmes est plutôt d'ordre théorique tandis que le deuxième est plutôt d'ordre pratique.

Le Modèle de Substitution est pris en défaut ...

Jusqu'à présent, une fonction est caractérisée par son expression de définition et un environnement local décrivant ses variables liées : les paramètres. Ce modèle peut être illustré par l'exemple suivant

$$\lambda_{xy\,\cdot\,xy+} = \begin{bmatrix} \text{soit } :x & = \cdots \\ y & = \cdots \\ \text{dans } :xy+ & \end{bmatrix}$$

Le mécanisme de l'application d'une fonction à ses arguments est, jusqu'à présent, la simple définition des variables liées à partir de la valeur des arguments, ce qui revient à une **simple substitution**.

L'introduction de l'affectation rend notre modéle précédent inopérant et il a fallu en introduire un autre, un peu plus complexe, que nous allons décrire complètement petit à petit. Pour mettre en évidence la difficulté introduite par l'affectation tentons d'analyser la construction d'un compte-bancaire à la lumière du mécanisme précédent et voyons où cette construction échoue.

L'évaluation de l'expression

$$\lambda\cdot\left[\begin{array}{l}\text{soit : } solde\,0 \;=\\ \text{dans : } \left[\begin{array}{l}\text{soit : } \text{retirer}\,x = \lambda\,\cdot \text{affecter}(\ldots)\,solde\,solde\,x\;-\\ \qquad\;\; \text{déposer}\,x = \lambda\,\cdot \text{affecter}(\ldots)\,solde\,solde\,x\;+\\ \text{dans : } \lambda m\cdot(\ldots)\,m = \textbf{solde} \rightarrow solde,\\ \qquad\qquad\quad (\ldots)\,m = \textbf{retirer} \rightarrow \text{retirer},\\ \qquad\qquad\quad (\ldots)\,m = \textbf{déposer} \rightarrow \text{déposer}\end{array}\right]\end{array}\right]$$

devrait engendrer, dans le cadre du modèle de substitution actuel, l'objet suivant

$$\begin{array}{l}\lambda m\cdot(\ldots)\,m = \textbf{solde} \rightarrow \textbf{0},\\ \qquad\qquad (\ldots)\,m = \textbf{retirer} \rightarrow \cdot(\ldots)\,\lambda x\;\text{affecter}(\ldots)\,\textbf{0}\,\textbf{0}\,x\;-\;,\\ \qquad\qquad (\ldots)\,m = \textbf{déposer} \rightarrow \lambda x\cdot\text{affecter}(\ldots)\,\textbf{0}\,\textbf{0}\,x\;+\end{array}$$

Il est clair que la définition de cet objet est absurde.

L'affectation va donc nous obliger à définir un *nouveau modèle* pour décrire l'application d'une fonction à ses arguments. Il s'agit, en fait, de définir une représentation pour les variables, les environnements, les fonctions et de définition la fonction . Apply(·,·)·

Il est nécessaire de «Ramasser les Miettes» ...

Afin de mettre le deuxième problème en évidence, examinons d'un peu plus près l'évaluation de l'expression suivante

```
(define foo (cons 'a 'b))
```

D'une façon imagée, nous dirons que nous venons de «prendre une vue» nommée `foo` sur la paire `(A . B)` et c'est cette vue qui nous permet de la manipuler lorsque nous en avons besoin.

Considérons, à présent, l'évaluation de l'expression

```
(set! foo 8)
```

Nous avons détachée la vue `foo` de la paire `(A . B)` et nous l'avons braquée sur le nombre 8. Mais alors, plus personne n'a de vue sur la paire `(A . B)` qui devient ainsi **inutilisable** ! L'opérateur `set!` a créé une *miette* sous la forme d'un objet inaccessible.

Sur un plan théorique, on peut considérer que le mécanisme de stockage des objets créés dispose d'un emplacement de taille illimitée et l'apparition de ces miettes n'est pas une gène. Sur un plan pratique, par contre, les possibilités de stockage des objets est limitée et ces miettes finissent par occuper tout l'espace disponible et empêcher la création de nouveaux objets. L'évaluation d'une expression échouera faute de place.

Les fonctions de manipulation des environnements doivent donc être complétées d'un mécanisme *ramasse-miettes* (garbage collector) qui récupère systématiquement la place occupée par les objets pour lesquels il peut prouver qu'il n'existe plus aucune vue pointée sur eux.

6.5 Variables & Environnements - Modèle graphique

Le modèle de substitution étant pris en défaut, nous avons besoin d'un nouveau modèle nous permettant d'appréhender comment se fait l'accès aux variables dans les différentes circonstances où elles sont utilisées.

Mais auparavant, il est nécessaire de nous rafraîchir un peu la mémoire. Lorsque nous avons introduit les variables (au paragraphe 2.4.4, page 23), nous n'avons introduit le modèle de substitution que comme une commodité mnémonique tout à fait valable tant qu'on n'utilise pas l'affectation mais qu'il nous faut abandonner à présent.

6.6 Les Structures mutables prédéfinies de Scheme.

L'opérateur `set!` que nous avons introduit va nous permettre de définir des opérateurs de mutation pour les listes et les vecteurs.

Paire mutable.

On peut modifier la définition de `cons` de telle sorte qu'on puisse introduire deux opérateurs de mutation **`set-car!`** et **`set-cdr!`** (Cf. figure 19, page 133).

```
(define (cons x y)
  (let ((set-x!(lambda (v) (set! x v)))
        (set-y! (lambda (v) (set! y v))))
    (lambda (msg)
      (cond  ((eq? msg 'car) x)
             ((eq? msg 'cdr) y)
             ((eq? msg 'set-car!) set-x!)
             ((eq? msg 'set-cdr!) set-y!)))))

(define (car paire) (paire 'car))

(define (cdr paire) (paire 'cdr))

(define (set-car! paire v) ((paire 'set-car!) v))

(define (set-cdr! paire v) ((paire 'set-cdr!) v))
```

Fig. 19 : Définition formelle d'une paire mutable.

Considérons les listes

```
(define x (list 1 2 3))
```

```
(define y (list 'a 'b))
(define z (list 'u 'v))
```

et évaluons

```
(set-car! x y)                          ⇒ ((a b) 2 3)
(set-cdr! x z)                          ⇒ ((a b) u v)
```

Mais que donnerait les évaluations suivantes ?

```
(define x (cons 1 '()))
(define y (cons 2 (cons 3 x)))
```

suivies de

```
(set-cdr! x y)                          ⇒ ???
```

... Il est clair que la structure ainsi obtenue est un brin pathologique !

Vecteur mutable.

On peut doter, de la même manière, les vecteurs d'un opérateur de mutation

```
(vector-set! <vecteur> <indice> <objet>)
```

6.7 Quelques structures mutables très utiles.

6.7.1 La Référence.

Lorsque nous avons commencé à nous intéresser aux comptes bancaires, nous avons défini la fonction

```
(define deposer
  (lambda (x) (set! solde (+ solde x)))
```

Cette fonction était, en fait, trop peu générale pour être intéressante. Une première tentative de généralisation pourrait être

```
(define deposer
  (lambda (x s) (set! s (+ s x)))
```

Tentons de l'utiliser...

```
[87](define solde 0)
SOLDE

[88](deposer 100 solde)
100

[89]solde
0
```

En fait, cette généralisation ne peut pas marcher. L'affectation qui a été effectuée concerne la variable `s` qui appartient à l'environnement créé au moment de l'application de la fonction `deposer` et non pas la variable `solde` elle-même qui est donc restée inchangée. Dans l'état actuel des choses, on ne peut pas définir le modificateur d'une variable passée en argument d'une procédure.

Pour résoudre ce problème, nous allons définir la structure mutable la plus simple qu'on puisse concevoir, la **Référence**, dont le constructeur et le modificateur sont donnés figure 20, page 135.

```
(define reference-a
  (lambda (v)
    (let ((set-v! (lambda (x) (set! v x))))
      (lambda msg
          (cond ((eq?msg nil)v)
                ((eq?(car msg) 'ref!)set-v!))))))

(define ref!
  (lambda (&ref x) ((&ref 'ref!) x) &ref[1]))
```

Fig. 20 : Définition d'une référence initialisée et de son modificateur. Le sélecteur de la valeur référencée est implicite. On remarque que la valeur référencée n'est accessible que par sa référence car elle appartient à l'environnement créé au moment de l'application de la fonction `reference-a`.

[1] Il n'est pas encore possible de justifier l'intérêt qu'on peut avoir à faire rendre la référence et non pas l'objet référencé, nous verrons que c'est un «bon coup stratégique».

Définissons une référence [3] sur une paire. Une référence à un objet est communément appelé «pointeur sur l'objet».

Un pointeur doit être interprété comme la «méthode d'accès» à un objet et non pas, comme on le fait trop souvent, comme sa simple «adresse mémoire» ce qui privilégie beaucoup trop les problèmes de réalisation au détriment des concepts.

```
(define &pp (reference-a (cons 'a 'b)))
```

On obtient le pointeur `&pp` qu'on peut alors *déréférencer* pour récupérer la valeur pointée

```
(&pp)                                    ⇒ (A . B)
```

Modifions, à présent, ce sur quoi pointe `&pp`

```
((ref! &pp 5))                           ⇒ 5
(&pp)                                    ⇒ 5
```

Reconsidérons le probléme posé par la définition d'un modificateur de variable et redéfinissons la fonction `deposer`. La nouvelle version de cette fonction reçoit en arguments une référence sur `solde` et non pas la variable `solde` elle-même

```
(define deposer
  (lambda (x &s) ((ref! &s (+ (&s) x)))))
```

Définissons alors une référence `&solde`

```
[90](define &solde (reference-a 0))
&SOLDE
```

Puis utilisons la fonction `deposer` en lui passant la référence à `solde` en argument

```
[91](deposer 10 &solde)
10

[92](&solde)
10
```

[3] De la même façon que nous avons caractérisés les prédicats par le signe`?` en fin de nom, les modificateurs par le signe `!`, nous caractériserons les références à des variables par le signe `&` en préfixe de leur nom. Cette représentation est empruntée au langage C++.

Lorsqu'on passe une référence en argument d'une fonction on dit qu'on effectue un *appel par référence*. Cette technique d'appel vient compléter celle que nous utilisons depuis le début : l'*appel par valeur*.

6.7.2 La Pile.

Définissons une **pile** de rangement telle que les objets qu'elle contient sont restitués dans l'ordre inverse où ils ont été rangés, dernier entré - premier sorti (LIFO: Last In First Out). Le type **Pile** peut être défini par un constructeur de pile vide `pile-vide`, une procédure de rangement `push` et une procédure de récupération `pop`. Le fonctionnement d'une telle pile peut être illustré par les expressions suivantes

```
(define p (pile-vide))
(push 5 p)                    ⇒ 5
(push 3 p)                    ⇒ 3
(push 'bonjour p)             ⇒ bonjour
(pop p)                       ⇒ bonjour
(pop p)                       ⇒ 3
(pop p)                       ⇒ 5
(pop p)                       ⇒ ()
```

On peut définir une pile comme étant une référence à une liste puisque les fonctions `car` (pour dépiler) et `cons` (pour empiler) opérent toutes les deux sur la tête de la liste. Une définition possible est donnée figure 21, page 136.

```
(define pile-vide
  (lambda () (reference-a nil))

(define push
  (lambda (x &p)
    (ref! &p (cons x (&p))) x))

(define pop
  (lambda (&p)
    (let ((sdp (car (&p))))
      (ref! &p (cdr (&p))) sdp))))
```

Fig. 21 : Définition d'une pile — son constructeur et ses opérateurs d'empilage et de dépilage.

On remarque que l'utilisation des structures mutables entraine «naturellement» la nécessité de l'utilisation du `begin` implicite puisque les modificateurs opérent essentiellement par effet de bord. Le style de programmation qui en résulte est appelé *programmation impérative*.

6.7.3 La File d'attente.

La **File** est une structure de données très utile lorsqu'on veut ranger des objets comme dans une file d'attente: premier entré - premier sorti (FIFO: First In First Out).

La structure **File** peut être définie par le constructeur `file-vide`, une procédure d'entrée `entrer` et une procédure de sortie `sortir`. Le fonctionnement d'une telle file peut être illustré par les expressions suivantes

```
(define f (file-vide))
(entrer 'bonjour f)                    ⇒ bonjour
(entrer 3 f)                           ⇒ 3
(entrer 5 f)                           ⇒ 5
(sortir f)                             ⇒ bonjour
(sortir f)                             ⇒ 3
(sortir f)                             ⇒ 5
```

On peut définir une file comme étant une référence à une liste puisque la fonctions `car` (pour sortir) opére sur sa tête (le début) tandis que la fonction `append` (pour entrer) opérent sur sa fin (Cf. figure 22, page 137).

```
(define file-vide
  (lambda () (reference-a nil)))

(define entrer
  (lambda (x &f)
    (ref! &f (append (&f) (list x))) x))

(define sortir
  (lambda (&f)
    (let ((x (car (&f))))
      (ref! &f (cdr (&f))) x)))
```

Fig. 22 : Définition d'une file — son constructeur et ses opérateurs d'entrée et de sortie.

6.7.4 Le Dictionnaire.

Nous avons déjà eu besoin, et nous aurons souvent besoin, d'une structure pour ranger un ensemble de couples nom$^{\leftrightarrow}$valeur. Une telle structure est appelée un *dictionnaire*, elle permet de ranger une valeur en lui associant un nom unique qui permettra de la retrouver.

On va définir la structure de dictionnaire par son constructeur `dictionnaire-vide` et les deux opérations

`inserer`	réalise soit la définition d'une variable soit son affectation si elle a déjà été définie.
`consulter`	rend la valeur liée à un nom.

Une application évidente d'une variante de dictionnaire est la représentation des environnements. Le fonctionnement d'un dictionnaire peut être illustré par les expressions suivantes

```
(define d (dictionnaire-vide))
(inserer 'baz 10 d)                    ⇒ 10
(inserer 'bar 7 d)                     ⇒ 7
(inserer 'foo 5 d)                     ⇒ 5
(consulter 'foo d)                     ⇒ 5
(consulter 'baz d)                     ⇒ 10
(consulter 'bar d)                     ⇒ 7
(consulter 'syngnathe[4] d)            ⇒ ()
(inserer 'foo 8 d)                     ⇒ 8
```

[4] poisson marin à corps et museau très allongé de l'ordre des catostéomes.

```
(consulter 'foo d)                    ⇒ 8
```

Le dictionnaire sera représenté par une référence à la liste qui contient les paires nom ↔ valeur (Cf. figure 23, page 138). Sa définition utilise la fonction primitive de *Scheme* `assq`

```
(define dictionnaire-vide
  (lambda () (reference-a nil)))

(define consulter
  (lambda (n &d)
    (let ((p (assq n (&d))))
      (if p (cdr p)))))

(define inserer
  (lambda (n v &d)
    (let ((p (assq n (&d)))
          (q (cons n v)))
      (if p (set-cdr! p v) (ref! &d (cons q (&d))))
      v)))
```

Fig. 23 : Définition d'un dictionnaire — son constructeur et ses opérateurs d'insertion et d'extraction.

qui parcourt une liste de paires afin d'en extraire celle dont le `car` est «`eq?`-égal» à une clé donnée et dont la définition formelle est la suivante

```
(define assq
  (lambda (s lp)
    (cond ((null? lp) nil)
          ((eq? s (caar lp)) (car lp))
          (else (assq s (cdr lp))))))
```

Une application curieuse des dictionnaires est la définition d'un **mémoïseur de fonction**. Lorsque certaines fonctions sont très longues à calculer et lorsqu'on se rend compte qu'on refait fréquement le même calcul, il peut être intéressant de stocker les résultats déjà obtenus. Un dictionnaire est tout indiqué pour faire ça.

Un mémoïseur de fonction [5] crée, à partir de la fonction qu'on lui donne, une *fonction mémoïsée*, c'est à dire associée à sa table des résultats déjà calculés. Cette fonction mémoïsée vérifie avant d'effectuer un calcul que le résultat de celui-ci n'est pas déjà dans la table. La définition d'un mémoïseur est donné figure 24, page 138.

```
(define memoiser
  (lambda (f)
    (let ((dr (dictionnaire-vide)))
      (lambda (x)
        (let ((r (consulter x dr)))
          (if r r (inserer x (f x) dr)))))))
```

Fig. 24 : Memoïseur de fonctions.

[5] On ne peut, bien sûr, mémoïser ici que de vraies fonctions à un argument comparable à l'aide du prédicat `eq?`.

Définissons la fonction `fact`

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

et créons une fonction factorielle mémoïsée

```
(define memo-fact (memoiser fact))
```

qui n'effectuera jamais deux fois le même calcul.

6.8 Variables, Environnements & autres Considérations.

Ce paragraphe peut être sauté en première lecture, mais sa lecture peut apporter une compréhension profonde des mécanismes utilisés pour gérer les environnements et les variables.

L'expression

$$\lambda m \cdot (m = \textbf{solde} \rightarrow \textbf{0},\ (m = \textbf{retirer} \rightarrow \cdots,\ (m = \textbf{déposer} \rightarrow \lambda x \cdot \text{affecter}(\cdots, x\ +$$

est absurde car les variable *solde*, *retirer* et *déposer* ont été **évaluées trop tôt**. Le problème ne se poserait pas si les liens nom$\overset{\leftrightarrow}{}$valeur avaient été rangés quelque part pour être évalués seulement **en cas de besoin**.

Une *variable* sera donc représentée par un lien nom$\overset{\leftrightarrow}{}$valeur dans lequel «nom» et «valeur» conservent leur identité et on appellera *environnement* la structure qui permet de ranger les liens en attente d'évaluation.

Afin de fixer les idées, nous allons considérer qu'il existe un ensemble d'environnements E, un ensemble de symboles S et un ensemble de valeurs V. Les symboles de S peuvent être utilisés pour nommer les éléments de V.

Nous supposerons, également, que le type **Paire** est prédéfini et que les fonctions suivantes sont primitives [6].

[6] Nous donnerons un peu plus loin une définition de ces fonctions.

$\delta : ESV \to E$	La fonction $\delta(e,s,v)$ rend l'environnement obtenu en ajoutant le lien créé entre le symbole *s* et la valeur *v* à l'environnement *e* donné.
$\alpha : ESV \to E$	La fonction $\alpha(e,s,v)$ modifie l'environnement *e* donné en remplaçant la valeur associée au symbole *s* par la valeur *v* et rend *v*.
$\eta : ES \to V$	La fonction $\eta(e,s)$ rend la valeur liée au symbole *s* dans l'environnement *e*.
$\rho : ES \to E$	La fonction $\rho(e,s)$ recherche, à partir de l'environnement *e*, l'environnement où le symbole *s* est lié.
$\varepsilon : E \to E$	La fonction $\varepsilon(e)$ rend un environnement vide dont l'environnement *e* est le *père*.

Ces fonctions vont nous permettre de définir la sémantique de l'évaluation d'un symbole, d'une abstraction fonctionnelle et en déduire une définition de la fonction $Apply(\cdot,\cdot)$.

6.8.1 Définitions des Fonctions ρ et ε:

Les fonctions ρ et ε sont facile à définir

$$\rho\sigma = \lambda s \cdot \left[\begin{array}{l}\text{soit } : e = \eta(\sigma, s) \\ \text{dans } : e \neq \bot \to e, \\ \qquad e = \bot \to \left[\begin{array}{l}\text{soit } : e = \rho(\eta\sigma)(s, \textbf{père}), \\ \text{dans } : e \neq \bot \to e, \bot\end{array}\right]\end{array}\right]$$

$$\varepsilon = \lambda e \cdot \delta(\bot, \text{"}\textbf{père}\text{"}, e)$$

6.8.2 Sémantique d'un Symbole.

La valeur d'un symbole est recherchée dans l'environnement dans lequel se fait l'évaluation. Si le symbole n'est pas trouvé dans cet environnement, on poursuit la recherche dans son environnement-père. Le résultat est soit la valeur qui a été liée au symbole, soit $\bot$ si le symbole n'est trouvé nulle part.

$$Eval\sigma[[\mathtt{s}]] = \left[\begin{array}{l}\text{soit } : e = \rho(\sigma, \mathtt{s}) \\ \text{dans } : e \neq \bot \to \eta(e, \mathtt{s}), \bot\end{array}\right]$$

Nota: *ce mécanisme permet de définir plusieurs fois un symbole dans une hiérarchie d'environnements, c'est la valeur liée dans l'environnement courant qui sera obtenue. C'est exactement l'effet que nous voulions obtenir.*

6.8.3 Sémantique de `set!`.

La sémantique de l'opérateur `set!` est facile à définir à présent

$$\text{Eval}_\sigma[\![\texttt{(set! s x)}]\!] = \begin{bmatrix} \text{soit : } e = \rho\sigma(\texttt{s}) \\ \text{dans : } e \neq () \alpha \perp \rightarrow \sigma'' e'' \texttt{sx}, \perp \end{bmatrix}$$

6.8.4 Sémantique d'une Forme `lambda`.

Considérons, dans un premier temps, les abstractions fonctionnelles à un seul paramètre. Ce cas n'est pas un cas particulier puisque nous avons vu (paragraphe 2.4.3, page 22) que toute fonction pouvait être mise sous forme curryfiée. Dans ces conditions, l'évaluation d'une abstraction fonctionnelle peut être définie par

$$\text{Eval}_\sigma[\![\texttt{(lambda (x) <exp>)}]\!] = \text{cons}(\sigma, \text{cons}(\texttt{x}, \text{cons}(\texttt{<exp>}, \perp)))$$

Une procédure est alors la liste de l'environnement qui existait au moment où elle est créée (il est capturé), du symbole qui représente son paramètre et de son expression de définition.

6.8.5 La Fonction Apply(·,·)

La structure des procédures étant définie, on peut définir la fonction Apply(·,·) correspondante.

$$\text{Apply}(p, v) = \begin{bmatrix} \text{soit : } e = \delta\varepsilon(\text{cadr}\,p)(\text{caddr}\,p)(\text{car}\,p)v \\ \text{dans : Eval}_e[\![\text{cadddr}\,p]\!] \end{bmatrix}$$

L'expression de définition de la procédure — cadddr(*p*) — évaluée dans l'environnement obtenu en étendant l'environnement qui avait été capturé au moment de la définition de la procédure — car(*p*) — par l'environnement qui contient le lien créé entre le paramètre de la procédure — cadr(*p*) — et l'argument de l'application — *v*.

6.8.6 Sémantique d'une Forme `letrec`.

La seule difficulté rencontrée dans la définition de la sémantique de `letrec` est la construction de l'environnement récursif associé

$$\text{Eval}_\sigma[\![\texttt{(letrec ((f <exp1>) <exp2>)}]\!] = \begin{bmatrix} \text{soit : } \begin{bmatrix} \text{soit : } e = \delta\varepsilon\sigma(\,)(\,)\,''\texttt{f}\,\perp \\ \text{dans : } \alpha(e'')\texttt{f Eval}_e[\![\texttt{<exp1>}]\!] \end{bmatrix} \\ \text{dans : Eval}_e[\![\texttt{<exp2>}]\!] \end{bmatrix}$$

6.9 Exercices.

E-76 En s'inspirant des structures mutables définies ci-dessus, on se propose de définir une structure mutable **Ensemble**.

1. Définir, au préalable, les fonctions suivantes qui opérent sur une liste

   `(within? x lst)`

   prédicat rendant `#T` si l'objet `x` est un élément de la liste `lst`.

   `(collect pred? lst)`

   rend la liste des éléments de la liste `lst` pour lesquels le prédicat `pred?` rend `#T`.

   `(reject pred? lst)`

   rend la liste des éléments de la liste `lst` pour lesquels le prédicat `pred?` rend `#F`.

2. En utilisant les fonctions précédentes (et d'autres), définir la structure mutable **Ensemble** à travers les procédures suivantes

   `(ensemble-vide)`

   rend un ensemble vide.

   `(appartient-a? x ens)`

   prédicat rendant `#T` si l'objet `x` appartient à l'ensemble `ens`.

   `(ensemble->liste ens)`

   rend la liste des éléments de l'ensemble `ens`.

   `(inserer-dans x ens)`

   ajoute l'objet `x` à l'ensemble `ens`, sauf s'il y est déjà et rend `ens`.

   `(tout-inserer-dans lst ens)`

   insére dans l'ensemble `ens` tous les éléments de la liste `lst` et rend `ens`.

   `(liste->ensemble lst)`

   rend l'ensemble construit à partir de tous les éléments de la liste `lst`.

   `(retirer-de x ens)`

   retire l'objet `x` de l'ensemble `ens` et rend `ens` modifié.

   `(tout-retirer-de ensd ens)`

   retire de l'ensemble `ens` tous les éléments de l'ensemble `ensd` et rend l'ensemble `ens` modifié.

   `(pour-tous-faire proc ens)`

   applique la procédure `proc` à chaque élément de `ens`.

   `(collecter-dans pred? ens)`

   rend l'ensemble de tous les éléments de `ens` pour lesquels le predicat `pred?` rend `#T`.

   `(rejeter-de pred? ens)`

   rend l'ensemble de tous les éléments de `ens` pour lesquels le predicat `pred?` rend `#F`.

Vous avez, probablement sans même vous en rendre compte, fait une hypothèse sur la nature des éléments que cet ensemble peut contenir. Précisez laquelle et trouver une méthode permettant d'éviter de la faire.

E-77 Même exercice que ci-dessus pour la définition d'une structure mutable de **Multi-Ensemble**. Un multi-ensemble (les anglo-américains disent bag) est une collection d'éléments pas nécessairement uniques. Chaque élément est donc caractérisé par son nombre d'occurences.

E-78 Modifier la définition donnée précédemment pour une structure mutable **Dictionnaire** afin d'en faire une structure mutable **Dictionnaire-Trié**. Le paramètre de construction d'un dictionnaire trié vide sera un prédicat rendant `#T` lorsque les deux éléments comparés sont «dans le bon ordre».

E-79 Définir une structure mutable **Agenda** qui permet :

- de fixer un rendez-vous pour une date donnée,
- d'annuler le rendez-vous fixé à une date donnée,
- de vérifier qu'un rendez-vous n'a pas été pris pour une date donnée,
- de faire la liste des rendez-vous à venir,
- d'annuler tous les rendez-vous passés.

Nota: *Afin de ne pas inutilement compliquer le problème, on représentera les dates par un simple nombre entier d'unités de temps, on se limitera à un seul rendez-vous par unité de temps. Un «rendez-vous» sera un objet dont il n'est pas nécessaire de préciser la nature.*

E-80 Considérons la sémantique d'une forme `letrec` telle qu'elle est définie au paragraphe 6.8.6, page 141. Quelle propriété doit nécessairement satisfaire la forme `<exp1>` pour que cette sémantique soit acceptable?

E-81 Reprendre la définition de la file donnée au paragraphe 6.7.3, page 136. Cette définition est peu efficace car pour chaque entrée, une nouvelle liste est créée. Trouver un définition de la file qui n'entraîne pas cet inconvénient.

Nota: *repérer le début et la fin de la file par une paire qu'on peut considérer comme un couple de pointeurs. Attention, cet exercice est difficile.*

7. Représentation des Données abstraites.

Pour illustrer de façon concrète le mécanisme de représentation des données imaginons une petite application très simple, mais très significative tirée du monde de la gestion. Cette étude de cas va, en outre, permettre d'illustrer les différents aspects du développement d'une application informatique :

1. Les *spécifications générales* d'une application sont établies indépendamment de toute idée de réalisation, elles établissent un modèle de l'application à concevoir en définissant son architecture et en mettant en évidence ses *concepts fondamentaux*.
2. Les *spécifications détaillées* précisent les concepts élaborés dans les spécifications générales, elles prennent en compte l'existence d' *outils généraux* pour choisir la représentation utilisée pour les différents concepts de l'application parmi un ensemble de représentations possibles. Elles introduisent les *concepts annexes* nécessaires.
3. Les spécifications détaillées peuvent mettre en évidence des fonctions d'usage général à développer, qui vont accroître la *base d'outils de l'entreprise*. Faire réutilisable est un investissement clé pour l'avenir.

7.1 Le «Mégateuf Gym Center».

Le responsable d'une petite salle de sport a besoin de gérer le fichier de ses clients. Pour cela il établit des fiches-client qui regroupent toutes les informations nécessaires à sa gestion et qui ont la forme suivante

Mégateuf Gym Center
Nom : Prénom Adresse : Date de Naissance :
Activité : Echéance Cotisation :

7.1.1 Cahier des charges

Son système de gestion doit permettre :

1. de tenir à jour la liste des clients,
2. de savoir qui fait quoi,
3. de savoir qui a bien payé sa cotisation.

Ce cahier des charges n'est pas très précis, mais il en va de même pour tous les cahiers des charges. Par contre il va rester sans modification tout au long de cette étude de cas, ce qui n'est pas très réaliste.

7.2 Spécifications générales

Les spécifications générales vont établir un modèle de la gestion de cette salle de sport et décrire les concepts de *client* et d'*ensemble de clients*. Il est de bonne stratégie de spécifier d'abord les entités correspondant aux niveaux d'abstractions les plus élevés. Cela garantit qu'il ne sera pas nécessaire de revenir trop souvent sur une spécification déjà établie lorsqu'on s'aperçoit qu'elle n'est pas adaptée.

Ces spécifications ne définissent que les *interfaces visibles* associées aux paquetages **Fichier-Clients** et **Client**. Les *corps* de ces paquetages ne seront définis que dans le cadre des spécifications détaillées.

7.2.1 Fichier-Clients

Fichier-Clients est manifestement une structure dotée du constructeur et des opérations suivantes :

`(fichier-client-vide)`

Rend un fichier-clients vide.

`(ajouter-client cl fic)`

Ajoute le client `cl` au fichier `fic` s'il n'y est pas déjà et rend le fichier `fic` modifié.

`(sont-client? nm pr fic)`

Rend un fichier contenant les clients dont les nom et prénom sont `nm` et `pr` dans le fichier `fic`.

`(afficher-liste-clients fic)`

Affiche tout le contenu du fichier `fic`.

`(liste-clients-activite act fic)`

Rend un fichier contenant les clients dans `fic` qui pratiquent l'activite `act`.

`(liste-clients-relance de fic)`

Rend un fichier contenant les clients dans `fic` qui n'ont pas renouvelé leur cotisation à la date `de`.

`(supprimer-clients cls fic)`

Retire de `fic` dont tous les éléments contenus dans le fichier `cls`. Cette primitive est typiquement à associer à `sont-client?` ou à `liste-clients-relance` et rend le fichier `fic` modifié.

7.2.2 Client

Un client est l'ensemble des renseignements contenus dans sa fiche-client et on peut le représenter par un type de données dont on définit le constructeur et les sélecteurs.

`(client: nm pr adr ddn act ecot)`

Rend un client défini par son nom, son prénom, son adresse, sa date de naissance, l'activité qu'il pratique et la date d'échéance de sa cotisation.

```
(nom-de cl)
(prenom-de cl)
(adresse-de cl)
(date-naissance-de cl)
(activite-de cl)
(echeance-cotisation-de cl)
```

Rendent respectivement le nom, le prénom, la date de naissance l'activité et la date d'échéance de cotisation du client `cl`.

La création d'un nouveau client fait apparaître la nécessité de définir une **Date** et une **Adresse**.

7.2.3 Date

Les spécifications de **Client** ont mis en évidence la nécessité de créer une structure **Date** dont on définit, a priori, le constructeur et les sélecteurs.

`(date: a m j)`

Rend la nouvelle date correspondant à `a` (année), `m` (mois) et `j`. (jour).

```
(jour-de d)
(mois-de d)
```

`(annee-de d)`

Rendent respectivement le jour, le mois et l'année constituant la date `d`.

7.2.4 Adresse

Les spécifications de **Client** ont mis en évidence la nécessité de créer une structure **Adresse** dont on définit, a priori, le constructeur et les sélecteurs.

`(adresse: n r cp v)`

Rend une adresse associée à `n` (numero), `r` (rue), `cp` (code-postal) et `v` (ville).

```
(numero-de adr)
(rue-de adr)
(code-postal-de adr)
(ville-de adr)
```

Rendent respectivement le numéro, le nom de la rue, le code postal et la ville constituant l'adresse `adr`.

7.2.5 Simulons les Spécifications

Voyons comment vont pouvoir s'exprimer les manipulations de **Fichier-Clients** définies dans le cahier des charges.

Créer quelques clients :

```
(define cl1 (client: "Church"
                     "Alonzo"
                     (adresse: 14
                               "rue de l'implication"
                               "06000"
                               "Cannes")
                     (date: 1 1 1903)
                     'musculation
                     (date: 15 12 1993)))

(define cl2 (client: "Kleene"
                     "Stephen"
                     (adresse: 28
                               "av. de la déduction"
                               "06000"
                               "Menton")
                     (date: 10 1 1909)
                     'aerobique
                     (date: 18 12 1993)))

(define cl3 (client: "Schwartzenegger"
                     "Arnold"
                     (adresse: 42
                               "boulevard du biceps"
                               "06100"
                               "Nice")
                     (date: 21 12 1943)
                     'musculation
                     (date: 16 12 1993)))
```

Définir le fichier-clients :

```
(define fic-MGC (fichier-client-vide))
```

Ajouter quelques clients :

```
(ajouter-client cl1 fic-MGC)
(ajouter-client cl2 fic-MGC)
(ajouter-client cl3 fic-MGC)
```

Afficher la liste des clients:

```
(afficher-liste-clients fic-MGC)

nom: Church
prenom: Alonzo
adresse: 14, rue de l'implication - 06000 Cannes
date de naissance: 1/1/1903
activité: MUSCULATION
échéance de cotisation : 15/12/1993

nom: Kleene
prenom: Stephen
adresse: 28, av. de la déduction - 06000 Menton
date de naissance: 10/1/1909
activité: AEROBIQUE
échéance de cotisation : 18/12/1993

nom: Schwartzenegger
prenom: Arnold
adresse: 42, boulevard du biceps - 06100 Nice
date de naissance: 21/12/1943
activité: MUSCULATION
échéance de cotisation : 16/12/1993
```

Identifier un client:

```
(afficher-liste-clients (sont-client? "Church"
                                      "Alonzo"
                                      fic-MGC))
nom: Church
prenom: Alonzo
adresse: 14, rue de l'implication - 06000 Cannes
date de naissance: 1/1/1903
activité: MUSCULATION
échéance de cotisation : 15/12/1993
```

Liste des clients pratiquant la musculation:

```
(afficher-liste-clients
  (liste-clients-activite 'musculation fic-MGC))

nom : Church
prenom : Alonzo
adresse : 14, rue de l'implication - 06000 Cannes
date de naissance : 1/1/1903
activité : MUSCULATION
échéance de cotisation : 15/12/1993
```

```
nom : Schwartzenegger
prenom : Arnold
adresse : 42, boulevard du biceps - 06100 Nice
date de naissance : 21/12/1943
activité : MUSCULATION
échéance de cotisation : 16/12/1993
```

Liste des clients dont la cotisation n'est pas à jour au 15/12/1993:

```
(afficher-liste-clients
  (liste-clients-relance (date: 15 12 1993) fic-MGC))

nom : Church
prenom : Alonzo
adresse : 14, rue de l'implication - 06000 Cannes
date de naissance : 1/1/1903
activité : MUSCULATION
échéance de cotisation : 15/12/1993
```

Supprimons les clients non à jour au 15/12/1993:

```
(afficher-liste-clients
  (supprimer-clients
    (liste-clients-relance (date: 15 12 1993) fic-MGC)
    fic-MGC))

nom : Kleene
prenom : Stephen
adresse : 28, av. de la déduction - 06000 Menton
date de naissance : 10/1/1909
activité : AEROBIQUE
échéance de cotisation : 18/12/1993

nom : Schwartzenegger
prenom : Arnold
adresse : 42, boulevard du biceps - 06100 Nice
date de naissance : 21/12/1943
activité : MUSCULATION
échéance de cotisation : 16/12/1993
```

Il ne reste plus qu'à vérifier auprès du client que le modèle établi est acceptable. **Il est possible, donc souhaitable, de rédiger dès à présent la notice d'utilisation de l'application** .

7.3 Spécifications détaillées

Les spécifications générales ayant été acceptées, on peut commencer à définir les spécifications détaillées qui décrivent les corps associés aux différents paquetages. Pour cela, il est nécessaire de choisir une représentation de **Fichier-Clients**, de **Client**, de **Date** et de **Adresse**.

7.3.1 Fichier-Clients

La représentation la plus naturelle de **Fichier-Clients** est un **Ensemble** de clients puisqu'on ne connait pas, a priori, le nombre des clients à gérer et qu'il ne doit pas y avoir deux clients

identiques. Cette représentation étant choisie, on peut définir les primitives spécifiées. Pour cela nous allons, bien sûr [1], utiliser les opérations de manipulation des ensembles telles qu'elles ont été définies dans l'exercice **E-76**.

Les spécifications détaillées de **Fichier-Clients** sont donc.

```
(define fichier-client-vide
  (lambda () (ensemble-vide)))

(define ajouter-client
  (lambda (cl fic)
    (inserer-dans cl fic)))

(define afficher-liste-clients
  (lambda (fic)
    (pour-tous-faire afficher-client fic)))

(define sont-clients?
  (lambda (nm pr fic)
    (collecter-dans
      (lambda (cl) (and (eqv? nm (nom-de cl))
                        (eqv? pr (prenom-de cl))))
      fic))))

(define liste-clients-activite
  (lambda (act fic)
    (collecter-dans
      (lambda (cl) (eq? act (activite-de cl)))
      fic))))

(define liste-clients-relance
  (lambda (de fic)
    (collecter-dans
      (lambda (cl)
        (posterieure-a? de (echeance-cotisation-de cl)))
      fic))))

(define supprimer-clients
  (lambda (cls fic)
    (tout-retirer-de cls fic)))
```

On se rend compte que le choix des primitives définies pour la manipulation des ensembles est crucial pour les utilisations futures qui en seront faites. Il est donc fondamental, lorsqu'on définit une nouvelle structure de données de la concevoir dans une optique de réutilisation si elle présente un caractère de généralité prévisible.

7.3.2 Client

La solution la plus simple est de construire le vecteur des différentes informations qui caractérisent le client. Le constructeur et les sélecteurs de **Client** sont:

```
(define client:
  (lambda (nm pr adr ddn act ecot)
    (vector nm pr adr ddn act ecot)))
```

[1] Réutiliser les types, structures et fonctions déjà définies est la clé de la programmation professionnelle.

```
(define nom-de (lambda (cl) (vector-ref cl 0)))
(define prenom-de (lambda (cl) (vector-ref cl 1)))
(define adresse-de (lambda (cl) (vector-ref cl 2)))
(define date-naissance-de (lambda (cl) (vector-ref cl 3)))
(define activite-de (lambda (cl) (vector-ref cl 4)))
(define echeance-cotisation-de (lambda (cl) (vector-ref cl 5)))
```

Les spécifications détaillées de **Fichier-Clients** ont fait apparaître la nécessité de compléter le type **Client** par une primitive d'affichage.

```
(define afficher-client
  (lambda (cl)
    (begin
      (display "nom : ")
      (display (nom-de cl))
      (newline)
      (display "prenom : ")
      (display (prenom-de cl))
      (newline)
      (display "adresse : ")
      (afficher-adresse (adresse-de cl))
      (newline)
      (display "date de naissance : ")
      (afficher-date (date-naissance-de cl))
      (newline)
      (display "activite : ")
      (display (activite-de cl))
      (newline)
      (display "echeance de cotisation : ")
      (afficher-date (echeance-cotisation-de cl))
      (newline)
      (newline))))
```

On peut se rendre compte qu'il manque les modificateurs permettant de tenir compte du fait que si un client change rarement de nom, de prénom et de date de naissance, il peut déménager, changer l'activité qu'il pratique et que la date d'échéance de sa cotisation va changer chaque fois qu'il renouvelle son inscription. Cette limitation n'est pas forcément génante dans une première approche car rien n'empêche de considérer qu'un client modifié est un nouveau client, c'est donc l'hypothèse que nous ferons.

7.3.3 Date

La représentation la plus simple des dates est le vecteur des différentes informations qui la caractérisent. Le constructeur et les sélecteurs de **Date** sont donc

```
(define date: (lambda (j m a) (vector j m a)))

(define jour-de (lambda (d) (vector-ref d 0))
(define mois-de (lambda (d) (vector-ref d 1))
(define annee-de (lambda (d) (vector-ref d 2))
```

Les spécifications détaillées de **Fichier-Clients** ont fait apparaître la nécessité de compléter le type **Date** par des primitives adaptées à la manipulation des dates.

La comparaison de deux dates peut être facilement effectuée en en comparant deux représentations entières. Il suffit que la représentation d'une date soit unique pour que la compa-

raison puisse s'effectuer. On peut prendre, par exemple, un nombre qui *ressemble* au nombre des jours écoulés depuis le 1er janvier 1900

```
(define posterieure-a?
  (lambda (d1 d2)
    (let ((code (lambda (d)
                  (+ (jour-de     d)
                     (* (mois-de  d) 31)
                     (* (annee-de d) 366)))))
      (> (code d1) (code d2)))))
```

L'affichage d'une date peut être défini par

```
(define afficher-date
  (lambda (d)
    (display (jour-de d))
    (display "/")
    (display (mois-de d))
    (display "/")
    (display (annee-de d)))
```

7.3.4 Adresse

La représentation la plus simple d'une adresse est le vecteur des différentes informations qui la caractérisent. Le constructeur et les sélecteurs de **Adresse** sont

```
(define adresse: (lambda (n r cp v) (vector n r cp v)))

(define numero-de (lambda (adr) (vector-ref adr 0)))
(define rue-de (lambda (adr) (vector-ref adr 1)))
(define code-postal-de (lambda (adr) (vector-ref adr 2)))
(define ville-de (lambda (adr) (vector-ref adr 3)))
```

Les spécifications détaillées de **Fichier-Clients** ont fait apparaître la nécessité de compléter le type **Adresse** par une primitive d'affichage.

```
(define afficher-adresse
  (lambda (adr)
    (begin
      (display (numero-de  adr))
      (display ", ")
      (display (rue-de  adr))
      (display " - ")
      (display (code-postal-de adr))
      (display " ")
      (display (ville-de adr)))))
```

7.4 Le «Mégateuf Gym Center» s'aggrandit

Une application informatique n'est pas un objet figé, elle se corrige de ses inévitables dysfonctionnements, s'améliore pour suivre la pression de la concurrence, bref elle vit. Une conception robuste est capable de résister à tous ces changements le plus longtemps possible.

Poursuivons notre étude du «Mégateuf Gym Center» en faisant face à la première maladie des applications informatiques, **l'élargissement de leur champ d'action**.

Après avoir fait de bonnes affaires, le «Mégateuf Gym Center» rachète le «Gold Gym of Venice» qui possède, lui aussi, un système informatisé pour la gestion de ses clients. Les fichiers-clients ayant été conçus indépendamment l'un de l'autre, ils n'ont pas la même structure.

Une astucieuse architecture de réseau permet au responsable du MGC et de GGV d'avoir accès de manière transparente au fichier-clients du GGV et à celui du MGC et bien entendu, il n'est pas question de modifier l'un ou l'autre.

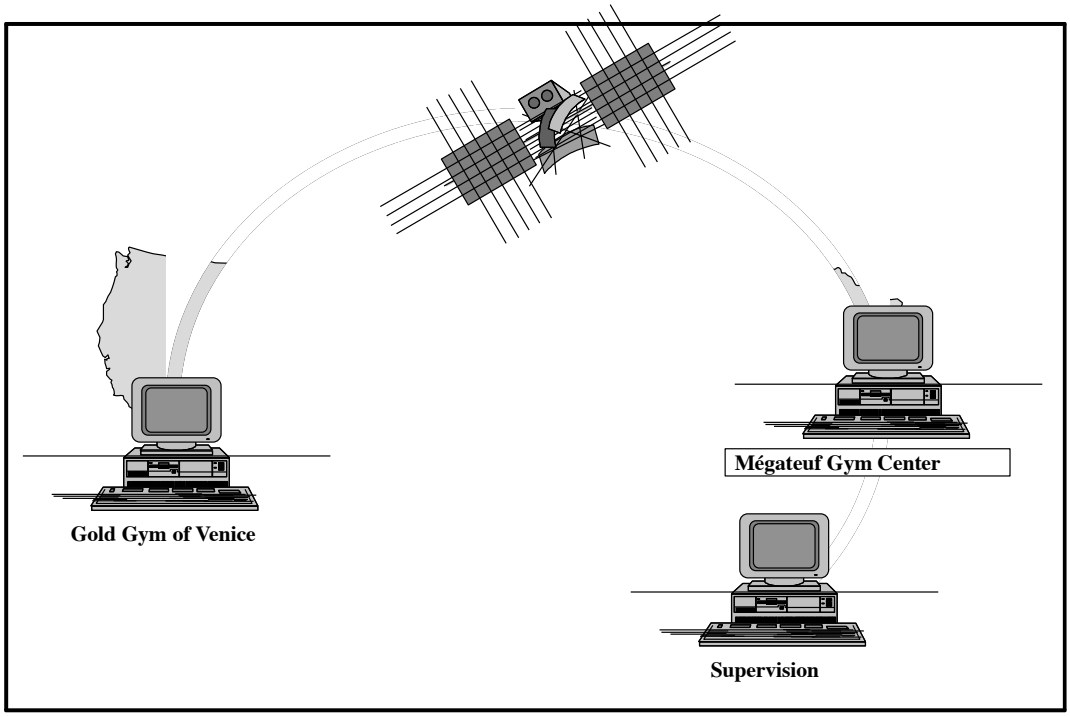


Fig. 25 : Le réseau mondial de gestion. Les deux centres de sport sont gérés indépendamment l'un de l'autre, mais il faudrait pouvoir les superviser.

7.4.1 Specifications générales

Supervision est l'ensemble des fichiers-clients doté du constructeur et des opérations suivantes :

`(connection-fichier centre)`

Rend le fichier-clients du centre `centre`. Cette fonction établit, en fait, la liaison avec le centre désiré et donne un accès au fichier-clients du centre.

`(liste-clients fic)`

Affiche tout le contenu du fichier-clients associé à `fic`.

`(liste-par-activite act fic)`

Affiche le fichier des clients contenus dans le fichier-clients associé à `fic` et qui pratiquent une activité `act` donnée.

`(liste-pour-relance de fic)`

Affiche le fichier des clients associé à `fic` qui n'ont pas renouvelé leur cotisation à la date `de`.

La spécification de la fonction `liste-pour-relance` met en évidence la nécessité d'introduire un type **Date** doté du constructeur et des opérations suivantes:

`(date: j m a)`

Rend la nouvelle date correspondant à `a` (année), `m` (mois) et `j` (jour).

`(date->MGC-date date)`

Convertit `date` dans la représentation d'une date MGC.

`(date->GGV-date date)`

Convertit `date` dans la représentation d'une date GGV.

7.5 Spécifications détaillées de la Supervision.

Les fichiers associés aux systèmes de gestion du MGC et du GGV n'étant pas structurés de la même manière, ils ne peuvent pas être manipulés par les mêmes fonctions.

La structure du fichier-clients MGC est supposée supporter les primitives:

```
(MGC-liste-clients fic)
(MGC-par-activite act fic)
(MGC-pour-relance de fic)
```

tandis que la structure du fichier-clients GGV est supposée supporter les primitives:

```
(GGV-liste-clients fic)
(GGV-par-activite act fic)
(GGV-pour-relance de fic)
```

Il est donc nécessaire d'associer à chaque fichier un *indicateur* permettant de savoir avec quelles primitives on peut le manipuler. La procédure `connection-fichier` va donc associer un **type explicite** au fichier-clients de chaque centre:

```
(define (connection-fichier centre)
  (cons centre (ouvrir-fichier centre)))
```

le type `centre` est un symbole pouvant être soit `MGC` soit `GGV` et la fonction `ouvrir-fichier`, non précisée ici, rend le fichier-clients du centre adressé.

La structure du fichier-clients telle qu'elle est perçue par **Supervision** est donc une paire type$^{\leftrightarrow}$valeur à laquelle on peut associer les deux sélecteurs:

```
(define type-de (lambda (fic) (car fic)))
(define valeur-de (lambda (fic) (cdr fic)))
```

Récapitulons la situation en établissant le tableau faisant le bilan des opérations et des types:

	Types	
Opérateurs	MGC	GGV
`liste-clients`	`MGC-liste-clients`	`GGV-liste-clients`
`par-activite`	`MGC-par-activite`	`GGV-par-activite`
`pour-relance`	`MGC-pour-relance`	`GGV-pour-relance`
`date`	`date->MGC-date`	`date->GGV-date`

L'uniformisation de l'accès aux différents fichier-clients a ainsi été obtenue en définissant une couche de convergence et les opérateurs de **Supervision** créent l'illusion d'une structure unique de fichier. On les appelle alors des *opérateurs génériques*.

Nous avons alors besoin d'une méthode qui permet de sélectionner le bon opérateur au bon moment. Elle nous est fournie à travers les trois approches suivantes :

1. *Aiguillage par le type* - chaque opérateur matérialise une ligne du tableau.
2. *Programmation par transmission de messages* - chaque type matérialise une colonne du tableau.
3. *Programmation dirigée par les données* - le tableau lui-même est associé à un mécanisme d'évaluation général.

7.6 Aiguillage par le type.

Dans ce cas les opérateurs sont intelligents (?), ils savent adapter leur action en fonction du type de la donnée qu'on leur soumet. Les **spécifications détaillées** des primitives de **Supervision** deviennent alors :

```
(define liste-clients
  (lambda (fic)
    (cond ((eq? (type-de fic) 'mgc)
           (MGC-liste-clients (valeur-de fic)))
          ((eq? (type-de fic) 'ggv)
           (GGV-liste-clients (valeur-de fic))))))

(define par-activite
  (lambda (act fic)
    (cond ((eq? (type-de fic) 'mgc)
           (MGC-par-activite act (valeur-de fic)))
          ((eq? (type-de fic) 'ggv)
           (GGV-par-activite act (valeur-de fic))))))

(define pour-relance
  (lambda (de fic)
    (cond ((eq? (type-de fic) 'mgc)
           (MGC-pour-relance (date->MGC-date de)
                             (valeur-de fic)))
          ((eq? (type-de fic) 'ggv)
           (GGV-pour-relance (date->GGV-date de)
                             (valeur-de fic))))))
```

Cette technique permet de rajouter facilement de nouveaux opérateurs, par contre l'introduction d'un nouveau type de données amène à modifier tous les opérateurs génériques existant.

L'aiguillage par le type est une technique robuste vis à vis des changements qui induisent préférentiellement une modification du nombre des opérateurs.

7.7 Programmation par Transmission de Messages

Dans ce cas, les objets sont intelligents (?), ils savent identifier l'opération à effectuer et choisir la procédure adéquate. Définissons, alors, les deux types intelligents **Fichier-MGC** et **Fichier-GGV** à partir de leur constructeur:

```
(define fichier-MGC:
  (lambda (fic)
    (lambda (message)
      (cond ((eq? message 'fichier) fic)
            ((eq? message 'liste-clients) MGC-liste-clients)
            ((eq? message 'par-activite) MGC-par-activite)
            ((eq? message 'pour-relance) MGC-pour-relance)
            ((eq? message 'date) date->MGC-date)))))

(define fichier-GGV:
  (lambda (fic)
    (lambda (message)
      (cond ((eq? message 'fichier) fic)
            ((eq? message 'liste-clients) GGV-liste-clients)
            ((eq? message 'par-activite) GGV-par-activite)
            ((eq? message 'pour-relance) GGV-pour-relance)
            ((eq? message 'date) date->GGV-date)))))
```

Nous venons d'utiliser la technique de **transmission de messages** que nous avions déjà rencontrée. L'opérateur `connection-fichier` devient alors:

```
(define connection-fichier
  (lambda (centre)
    (cond ((eq? centre 'mgc)
           (fichier-MGC: (ouvrir-fichier centre)))
          ((eq? centre 'ggv)
           (fichier-GGV: (ouvrir-fichier centre))))))
```

et enfin les opérateurs génériques sont définis par:

```
(define liste-clients
  (lambda (fic)
    ((fic 'liste-clients) (fic 'fichier))))

(define par-activite
  (lambda (act fic)
    ((fic 'par-activite) act (fic 'fichier))))

(define pour-relance
  (lambda (de fic)
    ((fic 'pour-relance) ((fic 'date) de) (fic 'fichier))))
```

La généricité des opérateurs est obtenue grâce à une propriété puissante de la programmation par transmission de message, le même message est interprété de manière différente par les objets qui le reçoivent, cette propriété est appelée **polymorphisme**. Le polymorphisme est une des méthodes les plus puissantes qu'on connaisse pour créer des opérateurs génériques.

Cette technique permet de rajouter facilement de nouveaux types de données, par contre l'introduction d'un nouvel opérateur amène à modifier tous les types de données concernés existant.

La transmission de messages est une technique robuste vis à vis des changements qui induisent préférentiellement une modification du nombre des types de données.

7.8 Programmation dirigée par les Données.

Afin d'organiser l'aiguillage de manière plus souple, définissons, au préalable, une structure **Aiguillage** représentant un tableau général à deux entrées permettant la réalisation du tableau ci-dessus. Une telle structure est **indépendante** de l'application supervisée, elle est simplement paramétrée pour en décrire l'architecture.

7.8.1 Spécifications générales de l'Aiguillage.

La structure **Table** est un tableau dont les éléments sont repérés par un symbole de ligne et un symbole de colonne. Elle est dotée du constructeur et des opérations suivantes:

`(table-vide)`

Rend une table d'aiguillage vide.

`(table-ajouter-element s-lgn s-col x tab)`

Ajoute l'élément `x` dans la ligne repérée par le symbole `s-lgn` à l'emplacement repéré par le symbole `s-col` dans la table `tab`.

`(table-consulter s-lgn s-col tab)`

Rend l'élément enregistrée à la ligne `s-lgn` et dans la colonne `s-col` dans la table `tab`.

7.8.2 Utilisation de Table dans le Cadre de l'Application.

La table précédente est simplement construite par :

```
(define *mgc-ggv* (table-vide))
```

Cette table d'aiguillage est **nécessairement unique** pour l'application MGC-GGV, il est donc de bonne politique d'en faire une variable globale afin d'être sûr que, par erreur, une autre table n'est pas utilisée.

Il est commode d'introduire une fonction de construction de l'aiguillage

```
(define operateur
  (lambda (tp nm proc)
    (table-ajouter-element nm tp proc *mgc-ggv*)
```

On introduit, alors, toutes les opérations qui ont été définies pour l'application MGC-GGV:

```
(operateur 'mgc 'liste-clients MGC-liste-clients)
(operateur 'mgc 'par-activite MGC-par-activite)
(operateur 'mgc 'pour-relance MGC-pour-relance)
(operateur 'mgc 'date date->MGC-date)

(operateur 'ggv 'liste-clients GGV-liste-clients)
(operateur 'ggv 'par-activite GGV-par-activite)
(operateur 'ggv 'pour-relance GGV-pour-relance)
(operateur 'ggv 'date date->GGV-date)
```

La fonction `connexion-fichier` est identique à celle qui avait été définie pour la réalisation par «aiguillage par le type». Les fichiers sont donc typés explicitement et la méthode d'évaluation associée à l'application MGC-GGV peut alors être définie par:

```
(define faire
  (lambda (fic op . arg)
    (let ((proc ((table-consulter op (type-de fic) *mgc-ggv*))))
      (if (null? arg)
          (proc (valeur-de fic))
          (proc (car arg) (valeur-de fic))))))
```

et les primitives de **Fichier-Clients** peuvent être définies par:

```
(define liste-clients
  (lambda (fic)
    (faire fic 'liste-clients))

(define par-activite
  (lambda (act fic)
    (faire fic 'par-activite act)))

(define pour-relance
  (lambda (de fic)
    (faire fic 'pour-relance (faire fic 'date de))))
```

L'utilisation de ces fonctions est illustrée par les évaluations suivantes :

```
(define fichier-actif (connection-a-fichier 'ggv))
```

Le fichier-clients du centre GGV est ouvert et est accessible sous le nom `fichier-actif`. Pour obtenir la liste des clients, il suffit d'évaluer:

```
(liste-clients fichier-actif)
```

pour obtenir la liste des clients pratiquant la musculation, il suffit d'évaluer:

```
(par-activite 'musculation fichier-actif)
```

et pour obtenir celle des clients dont la cotisation n'est pas à jour au 15/12/1993, il suffit d'évaluer:

```
(pour-relance (date: 15 12 1993) fichier-actif)
```

Rajouter de nouveaux opérateurs et/ou un nouveau type de données devient extrêmement simple, il suffit de définir le symbole nommant ce type ou cet opérateur puis de mettre à jour la table d'aiguillage.

Cette technique est extrèmement robuste et doit être envisagée chaque fois que le cahier des charges de l'application à réaliser ne semble pas très stabilisé.

7.8.3 Spécifications détaillées de Table.

La structure **Table** peut être représentée par un dictionnaire qui regroupe chaque ligne représentée elle-même par un dictionnaire. Ses spécifications détaillées sont alors:

```
(define table-vide
  (lambda () (dictionnaire-vide)))

(define table-ajouter-element
  (lambda (s-lgn s-col x tab)
    (let ((lgn (table-consulter s-lgn tab)))
      (if lgn
```

```
            (inserer s-col x lgn)
            (let ((lgn (inserer s-lgn (dictionnaire-vide) tab)))
              (inserer s-col x lgn))))))

(define table-consulter
  (lambda (s-lgn s-col tab)
    (consulter s-col (consulter s-lgn tab))))
```

7.9 Le «Cycle de Vie d'un Logiciel»

Avant de commencer, rappelons le cadre du **développement professionnel** d'une application informatique en introduisant quelques idées qui seraient à développer considérablement dans le cadre d'un cours consacré au *Génie Logiciel*. On a coutume de représenter le *cycle de vie* d'un logiciel par le schéma «en V» suivant:

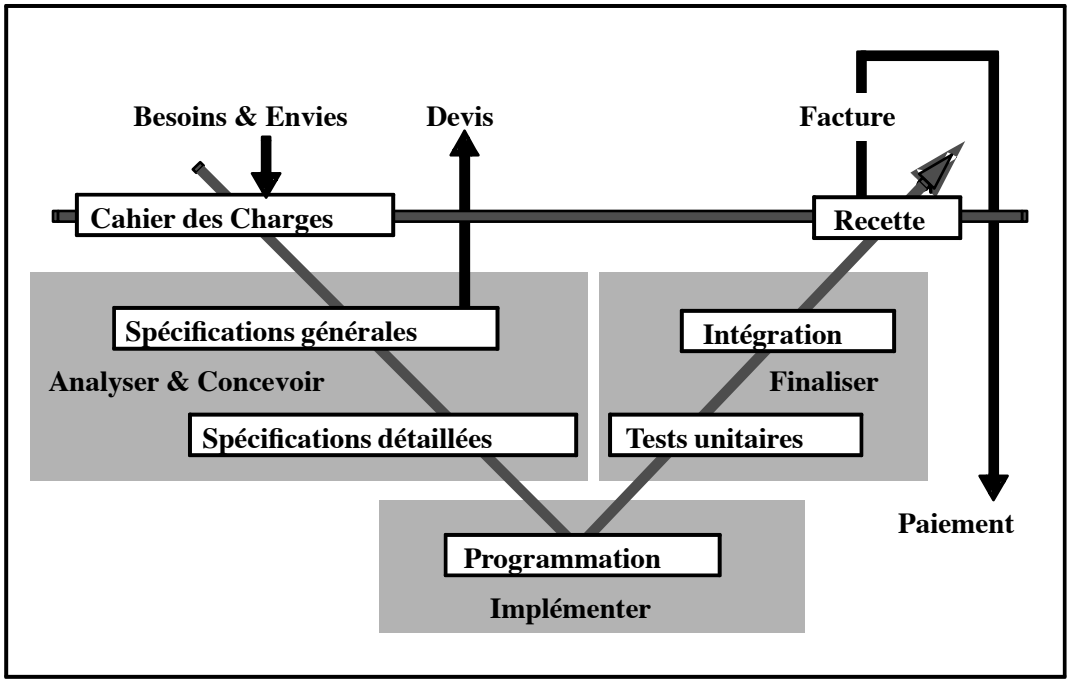


Fig. 26 : Le cycle de vie d'un logiciel.

La programmation dirigée par les données colle remarquablement bien à ce schéma. Les spécifications générales permettent de définir la table d'aiguillage tandis que les spécifications détaillées permettent de définir les opérateurs élémentaires.

Elle est, de plus, particulièrement adaptée au travail en équipe. Chaque membre de l'équipe est chargé de la définition des opérations élémentaires que le chef de projet intègre lorsqu'elles ont été validées.

7.10 Exercices.

E-82 On s'est aperçu au paragraphe 7.3.2, page 151 que les spécifications de **Client** supposent qu'un client est un structure non mutable. Modifiez les spécifications de **Client** pour en faire une structure mutable et modifier, en conséquence, les spécifications des autres structures de données concernées.

E-83 On se propose de réaliser la gestion des stocks d'un petit commerce de détail. Ce commerce revend différents articles qu'il peut se procurer auprès de ses différents fournisseurs.

Nota: *Afin de rendre la résolution de ce problème plus progressive, les questions sont posées en ordre inverse comme si l'analyse avait déjà été faite. Certaines questions ne précisent pas tous les éléments permettant d'élaborer une réponse. Dans ce cas, il vous appartient de faire les choix qui vous semblent judicieux et d'en préciser clairement la nature.*

1. Définir un type **Articles** par son constructeur et ses sélecteurs sachant qu'un article en stock est caractérisé par:
   - *réf* sa référence - un nombre entier positif,
   - *dénom* sa dénomination - une chaîne de caractères,
   - *pvu* son prix de vente unitaire - un nombre entier,
   - *qté* la quantité en stock.
2. Définir un type **Stock** par son constructeur et son sélecteur, sachant qu'un stock est un ensemble d'articles caractérisés par une référence qui doit donc être unique. Définir les fonctions suivantes:

   en-stock?

   réponds *vrai* si un article d'une référence réf donnée est en stock.

   article-en-stock

   rend l'article d'une référence donnée. Cette fonction rend *faux* si l'article n'est pas présent.

   référence-créer

   rend un stock contenant tous les articles d'un stock donné et un nouvel article de référence *réf* de dénomination *dénom* et de prix de vente unitaire *pvu* s'il n'y est pas déjà. Sa quantité en stock *qté* est alors nulle.
3. Utiliser les deux types de données précédents pour définir les fonctions de manipulation suivantes:

   article-ajouter

   ajoute *n* exemplaires de l'article de référence *réf* dans le stock *stk*.

   article-retirer

   retire, si possible, *n* exemplaires de l'article de référence *réf* du stock *stk*.

   stock-valeur

   rend la valeur totale du stock. Cette information permet de valoriser le stock au moment de l'établissement du bilan.

On se propose, à présent, de perfectionner cette gestion de stock en y adjoignant une gestion des réapprovisionnements.

4. Modifier le type **Articles** en lui adjoignant les champs suivants:
   - *pau* prix d'achat unitaire - un nombre entier,
   - *qté-alrt* quantité minimale d'alerte - un nombre entier.

Parmi les fonctions précédemment définies identifier celles qui doivent être modifiées et en donner la nouvelle version.

5. Dans ce nouveau contexte, définir les fonctions suivantes:

   stock-réappro

   donne la liste des références dont la quantité en stock est en-dessous du seuil d'alerte.

   stock-coût-réappro

   détermine le coût d'un réapprovisionnement qui ramènerait toutes les quantités en stock au seuil d'alerte.

6. Modifier le type **Articles** en lui adjoignant le champ
   - *fnrs* fournisseur de l'article - un symbole.

   Parmi les fonctions précédemment définies identifier celles qui doivent être modifiées et en donner la nouvelle version. Dans ce nouveau contexte définir la fonction «stock-fournisseurs» qui rend la liste de tous les fournisseurs chacun étant associé à la liste des produits qu'il fournit.

   Nota: *cet question exige une analyse un peu plus approfondie sur laquelle nous ne donnerons aucune indication*.

7.11 Annexe : structure «Ensemble».

Fonction pour compléter les manipulatiuons de liste :

```
(define within?
  (lambda (x lst)
    (cond ((null? lst) #F)
          ((equal? x (car lst)) #T)
          (else (within? x (cdr lst))))))

(define collect
  (lambda (pred? lst)
    (cond ((null? lst) nil)
          ((pred? (car lst)) (cons (car lst)
                                   (collect pred? (cdr lst))))
          (else (collect pred? (cdr lst))))))

(define reject
  (lambda (pred? lst)
    (let ((not-pred? (lambda (x) (not (pred? x)))))
      (collect not-pred? lst))))
```

Spécifications détaillées de la structure «Ensemble» :

```
(define ensemble-vide
  (lambda () (reference-a nil)))

(define ensemble->liste
  (lambda (ens) (ens)))

(define appartient-a?
  (lambda (x ens) (within? x (ensemble->liste ens))))

(define inserer-dans
  (lambda (x ens)
    (if (appartient-a? x ens)
        ens
        (ref! ens (cons x (ensemble->liste ens))))))

(define tout-inserer-dans
  (lambda (lst ens)
    (if (null? lst)
        ens
        (tout-inserer-dans (cdr lst)
                           (inserer-dans (car lst) ens)))))

(define liste->ensemble
  (lambda (lst)
    (tout-inserer-dans lst (ensemble-vide))))

(define retirer-de
  (lambda (x ens)
    (ref! ens (reject (lambda (y) (equal? y x))
                      (ensemble->liste ens)))))

(define tout-retirer-de
  (lambda (ens1 ens2)
```

```
    (ref! ens2
          (reject (lambda (y) (appartient-a? y ens1))
                  (ensemble->liste ens2)))))

(define pour-tous-faire
  (lambda (proc ens)
    (for-each proc (ensemble->liste ens))))

(define collecter-dans
  (lambda (pred? ens)
    (liste->ensemble (collect pred? (ensemble->liste ens)))))

(define rejeter-de
  (lambda (pred? ens)
    (liste->ensemble (reject pred? (ensemble->liste ens)))))
```

8. Objets & Programmation Orientée Objets

> J'appellerai «société de l'esprit» ce système selon lequel chaque esprit est composé d'un grand nombre de petits processus que nous appellerons agents. Chaque agent ne peut, à lui seul, effectuer que quelques tâches simples ne demandant ni esprit ni réflexion. Pourtant, le regroupement de ces agents en sociétés — selon des modalités bien particulières — peut aboutir à la véritable intelligence.
>
> Marvin Minsky
> La Société de l'Esprit

Nous allons tenter de fixer ici les principales idées sous-jacentes au concept d' *objet*. Ce concept n'est pas apparu sous la forme d'une construction purement théorique ayant trouvé une réalisation, il est le fruit de l'*évolution*. C'est ce qui fait, à la fois, sa force et sa faiblesse. C'est sa force en ce sens qu'il s'est forgé à partir d'une profonde nécessité et c'est sa faiblesse car d'une part il ne peut pas être considéré comme définitif et d'autre part il souffre de la présence de mutants probablement mal formés et n'ayant pas encore subis le filtre de la sélection naturelle.

Dire que le logiciel est «en crise» est un lieu commun. Pourtant, il est incontestable qu'il existe, aujourd'hui de bons logiciels — le traitement de texte que je suis en train d'utiliser en est la preuve. Il est donc possible de créer des logiciels comme on fabrique des appareils électroménagers: économiques, raisonnablement fiables, agréables à utiliser et présentant une esthétique indiscutable.

Mais où se situe donc le problème? Il se situe là où se situait le problème de la construction des grands bâtiments à la fin du moyen-âge, certaines cathédrales s'étant effondrées plusieurs fois avant de trouver leur forme définitive.

On construit, aujourd'hui encore, les logiciels à diffusion restreinte par essais et erreurs, ils ne bénéficient pas de l'évolution lente due à la stabilité du problème à résoudre, de sites de tests préliminaires nombreux ni de la pression de la concurrence qui les contraindrait à

s'améliorer pour survivre. Les professionnels sont bien conscients du problème et font de leur mieux pour y remédier en utilisant des méthodes, des outils etc.

La figure suivante, tirée d'un ouvrage [1] de B.J. Cox, illustre bien comment se situait, il y a une dizaine d'années, le problème posé par ces logiciels-là.

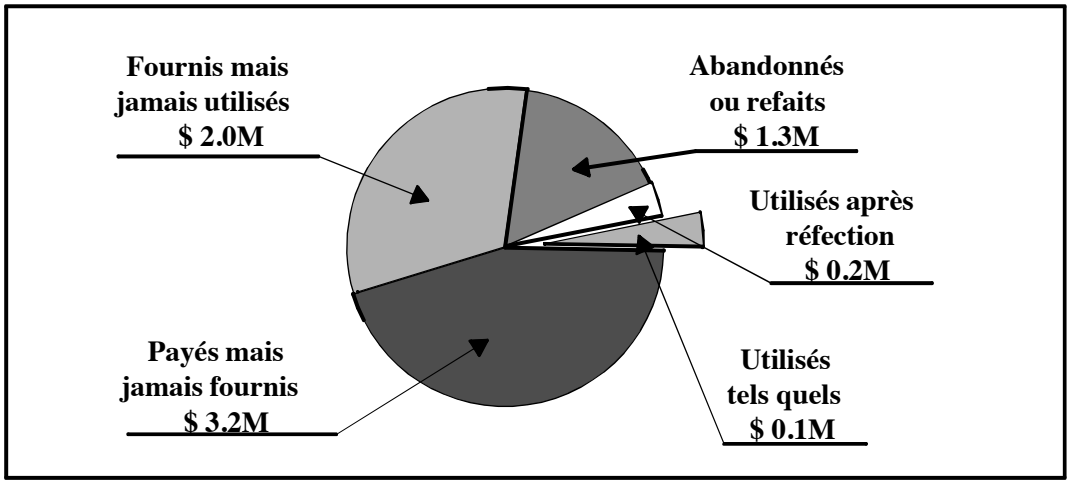


Fig. 27 : Les coûts d'un ensemble de logiciels développés pour l'administration américaine selon le U.S. Government Accounting Office Report - 1979.

Ces logiciels sont fondamentaux car ils gèrent des hôpitaux, des administrations, ils pilotent des avions, des missiles etc. Bien les concevoir est souvent une nécessité vitale.

L'approche objet, en considérant qu'une application est constituée d'un ensemble d' *agents coopérants*, fournit un mécanisme *simple* et *uniforme* pour appréhender les différents aspects qui apparaissent nécessairement lorsque l'échelle d'une application dépasse les possibilités d'un individu unique. Les deux mots *simple* et *uniforme* caractérisent un objet bien formé. Mais qu'il sera difficile de faire simple !

Si certains considérent que l'invention de la machine à vapeur a marqué le début de l'ère industrielle, d'autres pensent (dont J.B.Cox) que son vrai point de départ à plutôt été l'invention de la *pièce détachée interchangeable* .

> En 1798, pour satisfaire une commande gouvernementale de mousquets, Eli Whitney imagine de diviser le travail de telle sorte que chaque pièce de l'arme puisse être fabriquée par un spécialiste selon des standards définis afin que toutes les pièces s'assemblent sans problème.

L'électronique est née pratiquement en même temps que l'informatique, vers les années 1945. La productivité d'un ingénieur électronicien a été multipliée par un facteur d'environ 1000000 tandis que pendant le même temps celle d'un ingénieur informaticien n'était multipliée que par un facteur d'environ 500. Il est clair que l'invention du circuit intégré, la pièce interchangeable parfaite, est probablement la cause d'une telle divergence.

L'*objet* a donc pour objectif de jouer le rôle de la pièce interchangeable dans un système logiciel. En fait le mot *objet* semble, a posteriori, plutôt mal choisi, le mot *agent* introduit par Marvin Minsky serait mieux adapté car une pièce détachée a toujours un rôle fonctionnel incontestable, mais nous respecterons la tradition en parlant d' *objets*.

[1] *Object Oriented Programming - An Evolutionary Approach* - Addison-Wesley.

L'approche objet fait partie des stratégies efficaces pour attaquer la conception d'une application, ce n'est donc pas une méthode à proprement parler, il s'agit plutôt d'un *style de programmation* ou mieux d'un *mode de raisonnement*.

L'idée de pièce interchangeable logicielle n'est pas réellement nouvelle, nous avons tous utilisé, et nous utilisons tous abondamment, des bibliothèques de procédures toutes faites. Mais nous avons pu constater que ces procédures ne constituent pas réellement des pièces interchangeables.

Nous allons donc voir pourquoi les objets sont bien armés pour jouer ce rôle de pièce interchangeable et nous programmerons «objet» pour utiliser son effet démultiplicateur.

8.1 Qu'est-ce qu'un Objet?

Concevoir une application sur une base modulaire est une nécessité pour, au moins, trois raisons :

1. Il est très difficile de traiter efficacement plus de quelques problèmes simultanément,
2. Le cahier des charges d'une application évoluant tout au long de la réalisation, il est plus facile de ne changer que quelques pièces bien identifiées,
3. La maintenance est plus facile lorsque le fait de changer une pièce n'a pas de conséquences sur le comportement des autres pièces de l'application.

Il est bien écrit que l'application est conçue sur une base modulaire, elle n'est pas conçue en morceaux. Le découpage en modules engendre la pagaille sile concept de module et le mécanisme des interactions entre les modules ne sont pas aussi uniformes que possible.

Pour paraphraser Adele Goldberg [2] nous dirons que le secret d'une architecture réussie est l'*utilisation systématique d'une idée très simple* .

L'objet est conçu pour être une entité simple qu'on peut utiliser systématiquement pour bâtir une application modulaire. Pour pouvoir supporter de manière uniforme les différentes approches de modélisation que nous avons rencontrés. L'objet:

1. possède un état interne — il va permettre une utilisation saine de l'affectation,
2. regroupe des fonctions qui lui sont propres — il constitue une sorte de paquettage,
3. peut être manipulé comme un tout — c'est une abstraction.

Doté de toutes ces qualités, l'objet n'est pas une entité passive, il vaut mieux alors le considérer comme un *agent*. L'objet est alors un outil pour construire des *modules uniformes*.

8.1.1 Attributs d'un Objet

Nous allons partir d'une idée un peu curieuse. Considérons les différentes structures de donnée (éventuellement mutables) que nous avons définis jusqu'ici et remarquons qu'elles ont toutes été bâties (ou auraient pu être bâties) sur le même modèle.

```
(define constructeur:
  (lambda (<attributs initialisés>)
    (let ((<attributs non initialisés>)
          (<procédures de modification>))
```

[2] Un des concepteurs de Smalltalk 80 chez Xerox.

```
(lambda <message>
  (cond <interprétation des messages>)))))
```

L'aspect universel de ce «patron» va nous suggérer l'idée qu'il est possible de concevoir une **structure de données pour créer des structures de données** et notre premier problème est la représentation des attributs d'une structure de données.

Un attribut est un couple nom$^{\leftrightarrow}$valeur, et nous ne connaissons pas, a priori, le nombre des attributs à associer à la structure de données à construire. Les attributs sont associés aux opérations suivantes :

1. **Définir** - un nouvel attribut est ajouté à la stucture de données.
2. **Consulter** - pour récupérer la valeur d'un attribut repéré par son nom.
3. **Modifier** - pour changer la valeur d'un attribut repéré par son nom.

L'ensemble des attributs d'une structure de données peut donc être représenté par un dictionnaire que nous supposerons, comme d'habitude, défini par ailleurs et analogue à celui du paragraphe 6.7.4, page 137.

Les spécifications générales du dictionnaire utilisé sont celles de la figure 28, page 169.

`(dictionnaire-vide)`

Rend un dictionnaire vide.

`(dictionnaire-definir nm val dic)`

Introduit un nouveau couple `nm`$^{\leftrightarrow}$`val` dans le dictionnaire `dic`. Rend le symbole `'deja-defini` en cas d'erreur et `'ok` sinon.

`(dictionnaire-affecter nm val dic)`

Remplace la valeur associée à `nm` par `val` dans le dictionnaire `dic`. Rend le symbole `'non-defini` en cas d'erreur et `'ok` sinon.

`(dictionnaire-consulter nm dic)`

Rend la valeur associée à `nm` dans le dictionnaire `dic`. Rend le symbole `'non-defini` en cas d'erreur.

Fig. 28 : Spécification générales du dictionnaire.

Le constructeur de structures de données que nous appellerons **`objet`** peut être défini comme à la figure 29, page 170.

Ce constructeur est très simplifié et nous le perfectionnerons au fur et à mesure de nos besoins, cependant, dans cet état, il illustre parfaitement la faisabilité de notre projet.

On peut ainsi définir un chat

```
(define gros-minet (objet))
```

Ce n'est encore qu'une coquille vide à laquelle nous allons associer deux attributs

```
(gros-minet 'def 'couleur "noir et blanc")
(gros-minet 'def 'age 3)
```

puis les consulter

```
(gros-minet 'get 'couleur)          ⇒ noir et blanc
(gros-minet 'get 'age)              ⇒ 3
```

En tout cas, on peut définir un constructeur de chats

```
(define objet
  (lambda ()
    (let ((attributs (dictionnaire-vide)))
      (lambda (id nom . valeur)
        (cond ((eq? id 'def)
               (dictionnaire-definir nom
                                     (car valeur)
                                     attributs))
              ((eq? id 'set)
               (dictionnaire-affecter nom
                                      (car valeur)
                                      attributs))
              ((eq? id 'get)
               (dictionnaire-consulter nom attributs))))))
```

Fig. 29 : Constructeur d'une structure de données mutable universelle avec un dictionnaire des attributs.

```
(define chat:
  (lambda (c a)
    (let ((nouveau-chat (objet)))
      (nouveau-chat 'def 'couleur c)
      (nouveau-chat 'def 'age a)
      nouveau-chat)))
```

On peut alors créer des chats à l'aide de ce constructeur de chats, on dira qu'on *instancie* des chats.

```
(define tom (chat: "gris et blanc" 2))
```

et accéder à ses attrributs en utilisant le sélecteur universel `get` et le modificateur universel `set`

```
(tom 'get'couleur)                    ⇒ gris et blanc
(tom 'get 'age)                       ⇒ 2
(tom 'set 'age (+ 1 (tom 'get 'age)))
(tom 'get 'age)                       ⇒ 3
```

Manifestement, `objet` est un constructeur universel de structures de données mutables. Il permet également de faire une chose que nos structures de données précédentes ne permettaient pas : rajouter de nouveau attributs.

```
(tom 'def 'griffes "pointues")
```

Il est remarquable de constater toutes les possibilités qu'offre le constructeur `objet` malgré sa grande simplicité. Il permet de créer n'importe quelle structure mutable et à chaque fois qu'on lui ajoute un nouvel attribut, son sélecteur et son modificateur sont fournis avec. Nous n'allons pourtant pas en rester là.

8.1.2 Méthodes & Messages d'un Objet

Perfectionnons un peu notre constructeur universel. En effet, comme nous ne connaissons pas, a priori, le nombre des attributs, il serait très souhaitable de vérifier qu'on ne tente pas d'accéder à un attribut qui n'existe pas ou de redéfinir un attribut qui existe. La prétention de `objet` à être un constructeur universel oblige à prendre beaucoup plus de précautions que d'habitude et à envisager toutes les mauvaises utilisations possibles des structures construites à l'aide d' `objet`.

Méthodes privées & Liens statiques

Ainsi, les expressions

```
(dictionnaire-definir nom (car valeur) attributs)
(dictionnaire-affecter nom (car valeur) attributs)
(dictionnaire-consulter nom attributs)
```

utilisées dans la fonction de gestion des messages gagneront à être remplacée par des fonctions qui vérifient que l'attribut consulté existe.

```
(define objet
  (lambda ()
    (let* (<dictionnaire des attributs>
           (definir (lambda (nom valeur)
                      (let ((cr (dictionnaire-definir
                                  nom valeur attributs)))
                        (if (eq? cr 'deja-defini)
                            (writeln "deja defini : " nom)
                            'ok))))
           (affecter (lambda (nom valeur)
                      (let ((cr (dictionnaire-affecter
                                  nom valeur attributs)))
                        (if (eq? cr 'non-defini)
                            (writeln "non defini : " nom)
                            'ok))))
           (consulter (lambda (nom)
                      (let ((valeur (dictionnaire-consulter
                                      nom attributs)))
                        (if (eq? valeur 'non-defini)
                            (writeln "non defini : " nom)
                            valeur)))))
      <procédure-objet>))
```

La définition du constructeur d'objets doit être modifiée en conséquence et devient celle de la figure 30, page 171 .

```
(define objet
  (lambda ()
    (let* ((attributs (dictionnaire-vide))
           (definir ...)
           (affecter ...)
           (consulter ...))
      (lambda (id nom . valeur)
        (cond ((eq? id 'def) (definir nom valeur))
              ((eq? id 'set) (affecter nom valeur))
              ((eq? id 'get) (consulter nom)))))))
```

Fig. 30 : Définition «sûre» du constructeur d'objets.

Les procédures `definir`, `affecter` et `consulter`, associées aux messages `def`, `set` et `get` constituent les *méthodes* de l'objet. ces procédures étant définies **à l'intérieur** du constructeur d'objet, on les appelle ses **méthodes privées** qui lui sont systématiquement liées au moment de sa construction. On dit alors que le lien entre l'objet et ses méthodes privées est un **lien statique**.

Méthodes publiques & Liens dynamiques

Si dans les exemples précédents, les attributs définis n'étaient, par vocation, que des données, rien n'empêche d'introduire un attribut sous la forme d'une procédure.

```
(define gros-minet (objet))
(gros-minet 'def 'se-défend
            (lambda () (writeln " en griffant")))
```

Cette procédure peut alors être invoquée [3] après avoir été obtenue à l'aide du message `get`.

```
((gros-minet 'get 'se-defend))          ⇒ en griffant
```

On peut ainsi rajouter de nouvelles méthodes. Comme elles auront nécessairement été définies **à l'extérieur** de l'objet et on les appellera ses **méthodes publiques**. Si on prend la précaution de ne pas nommer une méthode publique avec le même nom que celui d'une méthode privée, on peut modifier la définition de objet de telle sorte que les methodse publiques soient directement invoquées.

```
(define objet
  (lambda ()
    (let* (<dictionnaire des attributs>
           <methodes privées>)
      (lambda (id . args)
        (cond ((eq? id 'def) (definir (car args)(cadr args)))
              ((eq? id 'set) (affecter (car args)(cadr args)))
              ((eq? id 'get) (consulter (car args)))
              (else ((consulter id))))))))
```

Tentons une simulation moins triviale de cette possibilité. Définissons un objet `intervalle` doté d'une origine et d'une extrémité.

```
(define intervalle (objet))
(intervalle 'def 'origine 5)
(intervalle 'def 'extremite 10)
```

Définissons une fonction pour le calcul du milieu de l'intervalle

```
(define milieu (lambda (...) ...))
```

et associons cette procédure au message '`milieu` de l'objet `intervalle`

```
(intervalle 'def 'milieu milieu)
```

et envoyons le message `'milieu` à l'objet `intervalle`

```
(intervalle 'milieu)                    ⇒ 7.5
```

Ce fonctionnement étant satisfaisant, essayons de définir réellement la fonction `milieu`

```
(define milieu
  (lambda (nom valeur)
    (/ (+ (<attribut origine>)
          (<attribut extremite>)
       2))
```

Mais que sont donc ces deux termes `<attribut origine>` et `<attribut extremite>` qu'il est nécessaire d'introduire dans la définition de cette fonction?

Et bien, ce sont les attributs de l'objet qui reçoit le message `milieu`. Il est donc nécessaire de rajouter le **destinataire du message** dans les paramètres de la procédure-méthode. Pour respecter la coutume, nous l'appellerons **`self`**.

[3] Notez le double parenthésage.

```
(define moyenne
  (lambda (self)
    (/ (+ (self 'get 'origine)
          (self 'get 'extremite)
       2))
```

La définition de la fonction de gestion des messages devient

```
(define objet
  (lambda ()
    (let* (<dictionnaire des attributs>
           <methodes privées>)
      (lambda (id . args)
        (cond ((eq? id 'def) (definir (car args)(cadr args)))
              ((eq? id 'set) (affecter (car args)(cadr args)))
              ((eq? id 'get) (consulter (car args)))
              (else ((consulter id) self)))))))
```

L'introduction des méthodes publiques introduit un petit problème. Si maintenant, on peut définir des méthodes publiques a priori quelconques, on ne peut plus se contenter du seul argument `self` et il faut prévoir la possibilité d'un nombre quelconque d'arguments supplémentaires. Pour cela il suffit de modifier légèrement la structure des arguments de l'objet-procédure.

```
(define objet
  (lambda ()
    (let* (<dictionnaire des attributs>
           <methodes privées>)
      (lambda (id . args)
        (let ((nom (car args))
              (valeur (cadr args)))
          (cond <traitements des messages de base>
                (else ((consulter id) self args)))))))
```

Le seul problème qui persiste est celui de la définition de `self` dans `objet`. La variable `self` représente l'objet qui reçoit le message, c'est à dire l'objet lui-même. Si on se souvient qu'un objet **est** sa fonction de gestion des messages, on va la définir en lui donnant le nom `self`. La définition de objet devient alors celle de la figure 31, page 174.

Tout objet, dans ce contexte, possède systématiquement trois méthodes privées associées aux messages `'def`, `'set` et `'get`. On peut, en revanche, lui associer autant de méthodes publiques qu'on le désire. Ces méthodes étant attachées à l'objet après sa création, on dit que le lien entre un objet et ses méthodes publiques est un **lien dynamique**.

```
(define objet
  (lambda ()
    (letrec ((attributs (dictionnaire-vide))
             (definir ...)
             (affecter ...)
             (consulter ...)
             (self
               (lambda (id . args)
                 (let ((nom (car args))
                       (valeur (cadr args)))
                   (cond ((eq? id 'def) (definir nom valeur))
                         ((eq? id 'set) (affecter nom valeur))
                         ((eq? id 'get) (consulter nom))
                         (else
                           ((consulter id) self args))))))))
      self)))
```

Fig. 31 : Constructeur d'une structure de données mutable universelle avec un dictionnaire des attributs et des méthodes publiques et un accès aux attributs du recepteur des messages.

8.1.3 Espèce d'un Objet & Héritage.

Nous avons vu que définir l'architecture d'une application c'est définir des objets et des relations entre eux. Nous venons de définir une première relation entre des objets: la relation *appartient-à*. En effet, on peut dire que les attributs d'un objet lui appartiennent, ou que ses attributs le composent. Cette relation permet ainsi de créer une structure hiérarchique des objets de l'applications.

C'est cette relation qui permet de dire: une voiture comporte quatre roues, un moteur, une carrosserie...

Le pouvoir structurant de cette relation est suffisamment puissant pour qu'en général, il ne soit pas nécessaire d'en définir une autre. Cependant, il peut être commode de définir une autre relation entre objets. Cette relation est abondamment utilisée par les naturaliste[4], c'est la relation *est-une-espèce-de* qui leur a permis de classifier les objets de la nature.

C'est cette relation qui permet de dire que si le chien, le chat et la vache sont des mammifères, ces trois espèces d'animaux ont en commun un certains nombre de propriétés. En particulier, ils ont (souvent) des poils et allaitent leurs petits.

Si on considère qu'il existe un *espèce d'animal fictif* le mammifère, le chien est une espèce de mammifère, le chat également ainsi que la vache. On dira que le chien, le chat et la vache *héritent* des propriétés du mammifère.

L'héritage porte sur les attributs(les poils) aussi bien que sur les méthodes (allaiter) que tous les mammifères ont en commun.

Héritage & Surcharge des Attributs et des Méthodes publiques

Hériter d'un attribut ou d'une méthode publique signifie que si on ne trouve pas cet attribut ou cette methode publique au sein d'un objet, on peut aller le chercher au sein de l'objet dont il hérite. Ainsi, si on appelle, conformément à la coutume, **`super`** l'objet dont un objet hé-

[4] L'auteur n'étant pas naturaliste, il prie ses lecteurs de lui pardonner les approximations que ses exemples, empruntés aux sciences naturelles, comportent probablement.

rite, cet héritage se matérialise par la modification des méthodes privées `affecter` et `consulter`. Si l'attribut ou la methode publique cherché ne se trouve pas dans l'objet recevant un message '`set` ou '`get`, ce message est transmis à `super`.

La fonction `affecter` et la fonction `consulter` sont alors redéfinie ainsi

```
(define objet
  (lambda ()
    (letrec (<dictionnaire des attributs>
             <definition des méthodes privées>
             (affecter (lambda (nom valeur)
                         (let ((cr (dictionnaire-affecter
                                     nom valeur attributs)))
                           (if (eq? cr 'non-defini)
                               (super 'set nom valeur)
                               'ok))))
             (consulter (lambda (nom)
                         (let ((valeur (dictionnaire-consulter
                                         nom attributs)))
                           (if (eq? valeur 'non-defini)
                               (super 'get nom)
                               valeur))))
             <definition de self>)
      self)))
```

Pour incorporer l'héritage des attributs et des méthodes publiques, la définition de `objet` doit être modifiée comme indiqué figure 32, page 175.

```
(define objet
  (lambda (super)
    (letrec ((attributs (dictionnaire-vide))
             (definir ...)
             (affecter ...)
             (consulter ...)
             (self
               (lambda (id . args)
                 (let ((nom (car args))
                       (valeur (cadr args)))
                   (cond ((eq? id 'def) (definir nom valeur))
                         ((eq? id 'set) (affecter nom valeur))
                         ((eq? id 'get) (consulter nom))
                         (else
                           ((consulter id) self args))))))))
      self)))
```

Fig. 32 : Constructeur d'une structure de données mutable universelle avec un dictionnaire des attribut et des méthodes, un accès aux attributs du recepteur des messages et un ancêtre dont il hérite des attributs.

Supposons que les mammifères aient, en général, des poils blancs, définissons `mammifere` et dotons-le de poils blancs.

```
(define mammifere (objet nil))
(mammifere 'def 'poils "blanc")
```

Définissons maintenant un chat `felix` en tant que espèce de `mammifere` et en le dotant de griffes.

```
(define felix (objet mammifere))
(felix 'def 'griffes "pointues")
```

Ainsi défini, `felix` possède les poils de `mammifère` et ses griffes en propre et on peut l'interroger connaître ses caractéristiques.

```
(felix 'get 'poils)                    ⇒ blancs
(felix 'get 'griffes)                  ⇒ pointues
```

Je connais un autre chat `gros-minet` qui, comme `felix`, a des griffes, mais qui, lui, a des poils blancs et noirs.

```
(define gros-minet (objet felix))
(gros-minet 'def 'poils "blancs et noirs")
```

Comment est-donc `gros-minet` ?

```
(gros-minet'get 'poils)                ⇒ blancs et noirs
(gros-minet'get 'griffes)              ⇒ pointues
```

On dit que l'attribut `poils` de `gros-minet` a **surchargé** l'attribut `poils` de `mammifere`.

La technique d'héritage permet de décrire un cas général définissant un ensemble de points communs entre différents objets tandis que la technique de surcharge permet d'introduire des exceptions.

Cette technique d'héritage n'est pas tout à fait dans l'esprit de celle utilisée implicitement par les naturalistes. Lorsqu'on dit que le chat hérite des poils du mammifère, cela signifie qu'il hérite de l'attribut poils mais pas de la couleur éventuelle de ces poils. D'autre part, il n'est pas très normal de permettre à `felix` ou à `gros-minet` de modifier la valeur d'un des attributs dont il a hérité. Nous reviendrons donc sur l'héritage afin d'affiner sa définition.

Définissons, à présent, chez `mammifere` une méthode pour nourrir ses petits.

```
(mammifere 'def 'nourrit-ses-petits
           (lambda (self args)
             (writeln "en les allaitant")))
```

Comment `felix` nourrit-il ses petits?

```
(felix 'nourrit-ses-petits)            ⇒ en les allaitant
```

Dotons `gros-minet` d'une méthode associée, elle aussi, au message ' `nourrit-ses-petits`.

```
(gros-minet 'def 'nourrit-ses-petits
            (lambda (self args)
            (writeln "avec soin")))
```

Comment `gros-minet` nourrit-il ses petits?

```
(gros-minet 'nourrit-ses-petits)       ⇒ avec soin
```

La méthode associée à 'nourrit-ses-petits chez `gros-minet` a, comme on s'y attendait, **surchargé** celle de `mammifere`.

Il est assez fréquent qu'on utilise l'héritage des méthodes pour introduire une variation sur une des méthodes de l'objet dont on hérite — cette variation est, par exemple, la correction d'un bug. On définit donc une méthode surchargeant celle qu'on veut modifier. Comme celle-ci n'introduit qu'une variation, il est très utile de pouvoir invoquer la méthode surchargée,

c'est à dire d'envoyer le message correspondant à `super`. Il est donc nécessaire de passer `super` en arguments des procédures-méthode. La définition de `objet` est alors celle de la figure 33, page 177.

```
(define objet
  (lambda (super)
    (letrec ((attributs (dictionnaire-vide))
             (definir ...)
             (affecter ...)
             (consulter ...)
             (self
               (lambda (id . args)
                 (let ((nom (car args))
                       (valeur (cadr args)))
                   (cond ((eq? id 'def) (definir nom valeur))
                         ((eq? id 'set) (affecter nom valeur))
                         ((eq? id 'get) (consulter nom))
                         (else
                           ((consulter id) self super args))))))
      self)))
```

Fig. 33 : Constructeur d'une structure de données mutable universelle avec un dictionnaire des attribut et des méthodes, un accès aux attributs du recepteur des messages, un ancêtre dont il hérite des attributs et des méthodes et un accès direct à son ancêtre.

Modifions, par exemple, la méthode ' `nourrit-ses-petits` de `gros-minet`.

```
(gros-minet 'def 'nourrit-ses-petits
            (lambda (self super args)
              (super 'nourrit-ses-petits)
              (writeln "avec soin")))
```

Comment `gros-minet` nourrit-il ses petits?

```
[93](gros-minet 'nourrit-ses-petits)
en les allaitant
avec soin
```

Notre Père à Tous

Le mécanisme d'héritage des attributs et des méthodes permet de rechercher un attribut ou une méthode manquante. Il est clair que nous n'avons pas encore traité le cas où on ne trouve nulle part l'attribut ou la méthode cherché. Ce problème peut être élégamment résolu en définissant un pseudo-objet dont tout objet hérite a priori.

La définition de «notre père à tous» est donc

```
(lambda (id nom . args)
  (error "symbole inconnu : " nom))
```

Que se passe-t-il, à présent?

```
[94](gros-minet 'courir)
[ERROR encountered!] symbole inconnu :
COURIR
```

```
[95](felix 'get 'moustaches)
[ERROR encountered!] symbole inconnu :
MOUSTACHES
```

Ce pseudo-objet doit systématiquement définir `super` au cas où l'ascendant d'un objet n'a pas été explicitement défini. La définition est facilement modifiée en conséquence. Cette définition peut être considérée comme finale, figure 34, page 178.

```
(define objet
  (lambda ascendant
    (letrec ((attributs (dictionnaire-vide))
             (definir ...)
             (affecter ...)
             (consulter ...)
             (super
               (if (null? ascendant)
                   (lambda (id nom . args)
                     (error "symbole inconnu : " nom))
                   (car ascendant)))
             (self
               (lambda (id . args)
                 (let ((nom (car args))
                       (valeur (cadr args)))
                   (cond ((eq? id 'def) (definir nom valeur))
                         ((eq? id 'set) (affecter nom valeur))
                         ((eq? id 'get) (consulter nom))
                         (else
                           ((consulter id) self super args))))))
      self)))
```

Fig. 34 : Le constructeur d'objets en version finale.

8.2 Objets-Classe & Objets-Instance

Créer des objet est laborieux car, pour chacun d'entre eux, il faut décrire ses attributs et ses méthodes. Il serait très agréable d'automatiser un peu leur fabrication.

Par exemple, il serait agréable de disposer d'un modèle universel de chats permettant de fabriquer autant de chats identiques que nous voulons. Un tel modèle devrait comporter tous les éléments qui permettent de caractériser les chats qui nous intéressent.

Nous dirons, par exemple, que nous nous intéressons aux chats qui ont trois attributs: `poils`, `griffes` et `queue` et dont le comportement est décrit par deux méthodes: `court` et `nourrit-ses-petits`.

Construisons alors un objet **`chat`**.

```
(define chat (objet))
```

Finalement la seule chose que cet objet **`chat`** a à faire, c'est de créer des chats. Il lui faut donc simplement une méthode lui permettant de le faire. Nous l'associerons aux message `'new` pour respecter la coutume.

```
(chat 'set 'new
      (lambda (self super args)
        (let ((un-chat (objet)))
          (un-chat 'def 'poils)
          (un-chat 'def 'griffes)
          (un-chat 'def 'queue)
          (un-chat 'def 'nourrit-ses-petits
                   (lambda (self super args)
                     (writeln "en les allaitant")))
          (un-chat 'def 'court
                   (lambda (self super args)
                     (writeln "en souplesse")))
          un-chat)))
```

Et maintenant fabriquons tout un tas de chats

```
(define felix      (chat 'new))
(define tom        (chat 'new))
(define gros-minet (chat 'new))
```

que nous pouvons individualiser.

```
(felix      'set 'poils 'noirs-et-blancs)
(tom        'set 'poils 'gris-et-blancs)
(gros-minet 'set 'poils 'blancs-et-noirs)
```

Cet objet **chat** qui nous permet de fabriquer tous les chats dont nous avons besoin s'appelle une **classe**. Tous les chats fabriqués par la classe **chat** seront appelés ses **instances**.

Ce qui distingue essentiellement une classe d'une instance, c'est le fait que ses attributs sont destinés à être partagés par toutes ses instances et qu'elle ne reconnaît que le message 'new qui lui permet de fabriquer des instances. Une classe est, si on peut dire, un *moule à instances*.

La classe **chat** que nous venons de définir n'utilise pas la possibilité d'héritage. Que faudrait-il changer pour que cette classe **chat** hérite d'une classe **mammifere** ?

Définissons d'abord la classe **mammifere** en lui attribuant des poils et une méthode pour nourrir ses petits.

```
(define mammifere (objet))

(mammifere 'def 'new
           (lambda (self super args)
             (let ((un-mammifere (objet)))
               (un-mammifere 'def 'poils)
               (un-mammifere 'def 'nourrit-ses-petits
                             (lambda (self super args)
                               (writeln "en les allaitant")))
               un-mammifere)))
```

Définissons maintenant la classe **chat** en la faisant hériter de la classe **mammifere** et en lui attribuant en propre des griffes, une queue et une méthode pour courir.

```
(define chat (objet mammifere))
```

L'antécédent d'une instance est tout simplement obtenu en instanciant l'antécédant de la classe.

```
(chat 'def 'new
      (lambda (self super args)
        (let ((un-chat (objet (super 'new)))))
```

```
(un-chat 'def 'griffes)
(un-chat 'def 'queue)
(un-chat 'def 'court
         (lambda (self super args)
          (writeln "en souplesse")))
un-chat)))
```

La conception d'une application informatique fondée sur un style de programmation orienté objet est la définition de toutes les classes de l'application. L'application, elle-même, est une classe, son lancement est simplement son instanciation.

Pour illustrer ce mécanisme de conception un peu déroutant au début, nous allons prendre un exemple typique: la réalisation d'une serrure à code.

8.3 Une Serrure à Code

Vous connaissez tous les serrures commandées par un petit clavier permettant d'ouvrir la porte donnant accès à un immeuble. Nous allons en donner une description et nous en construirons une espèce de simulateur. La structure du système est celle de la figure 35, page 180.

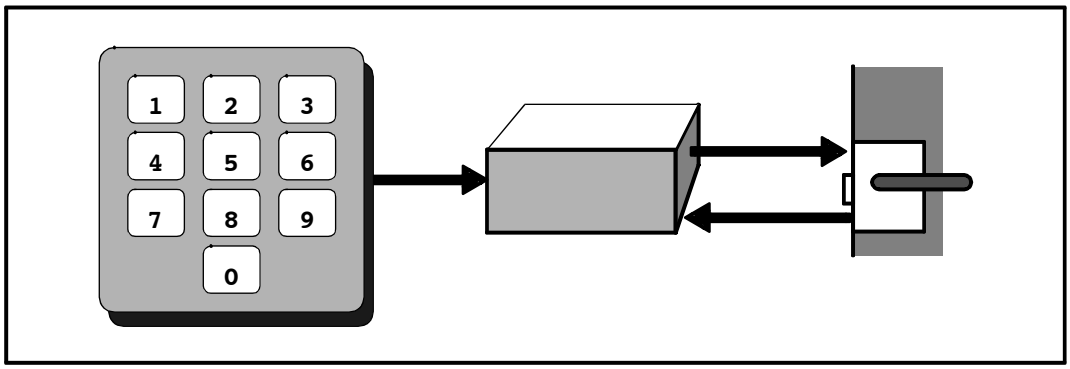


Fig. 35 : Serrure à code : son clavier, son décodeur et sa porte.

Cette serrure est constituée d'un clavier qui comporte des touches, d'un boitier électronique qui détecte les appuis sur les touches, qui ouvre la porte lorsque le code correct (séquence de touches) a été tapé et qui attend que la porte soit refermée pour réarmer la serrure.

8.3.1 Spécifications générale du Système de Serrure

Nous allons structurer notre application à l'aide de trois catégories d'objets :

1. des touches,
2. un codeur,
3. une porte.

Dans un premier temps nous allons concevoir le dialogue qui décrit les interactions entre ces trois objets:

- chaque touche reçoit le message `appuyer` qui lui indique qu'on a appuyé sur elle,

elle le signale alors au codeur en lui envoyant le message `signaler`.

- le codeur reçoit les messages `signaler` de chaque touche, il vérifie la conformité du code puis envoie le message `ouvrir` à la porte lorsque le code est correct.
- la porte reçoit le message `ouvrir` du codeur. Elle déverrouille la serrure. Lorsqu'on a franchit la porte, on lui envoit le message `fermer` pour la refermer. Elle envoie alors le message `rearmer` au codeur qui réarme la serrure. On peut , quand on le désire, lui envoyer le message `entrer?` pour essayer d'entrer.

Le dialogue de base est conçu autour de ces messages et il est décrit figure 36, page 181.

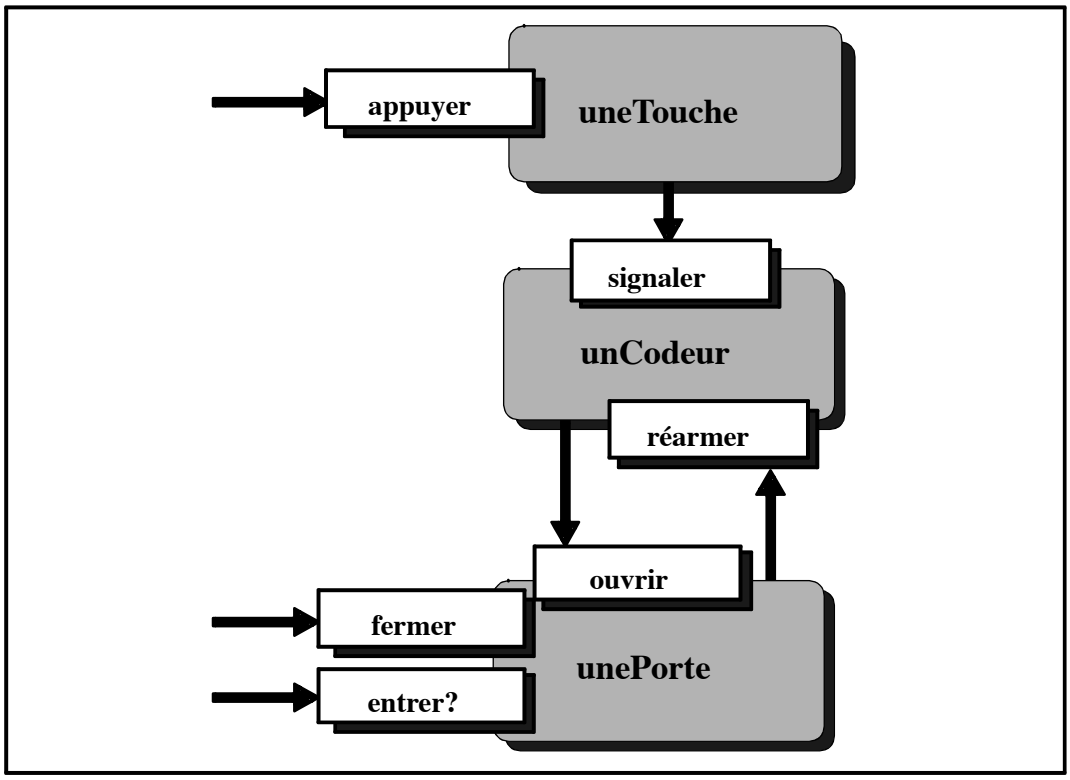


Fig. 36 : Structure du dialogue de la serrure à code.

Cette structure de dialogue étant définie, on peut, à présent, définir les différents objets constituant le système. Nous allons utiliser une représentation graphique relativement classique qui met en évidence les attributs et les méthodes publiques associés à l'objet considéré.

La structure d'une instance de la classe **Porte** est facile à décrire, elle est donnée figure 37, page 182. Elle comporte deux attributs `etat` et `codeur`. Le premier indique si elle est ouverte ou fermée, le deuxième à qui il faudra envoyer le message `rearmer` lorsque la porte aura reçu le message `fermer`. Si on examine les attributs des instances de **Porte**, on constate qu'ils jouent deux rôles totalement différents. Si `etat` peut être considéré comme un composant de porte, `codeur` ne le peut pas. Cet attribut particulier permet simplement à une porte de s'associer à un *partenaire de dialogue*. De tels attributs sont appelés des **accointances**. Nous dirons alors qu'une instance de **Porte** a un attribut et une accointance.

La structure d'une instance de **Touche** peut être décrite sous la forme indiquée figure 38, page 182. Son premier attribut permet de l'identifier car la clé de la serrure sera une suite des codes de touches.

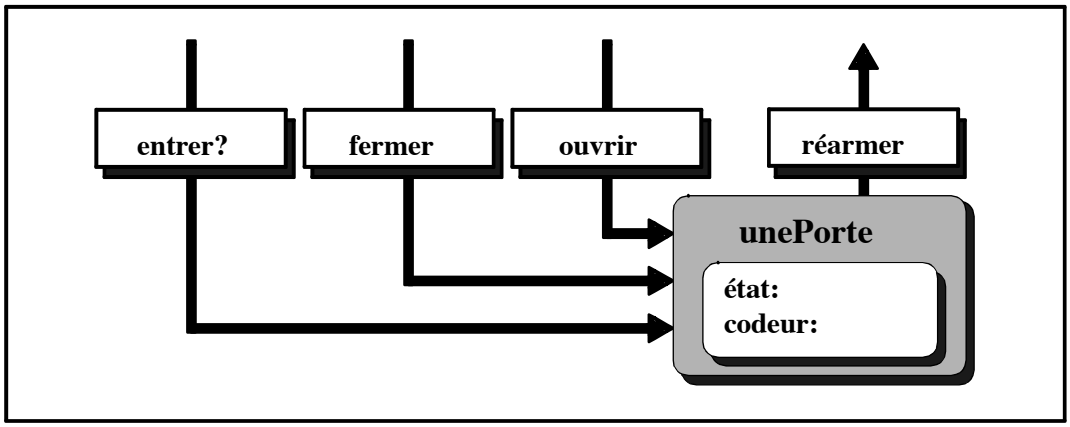


Fig. 37 : Structure d'une instance de Porte.

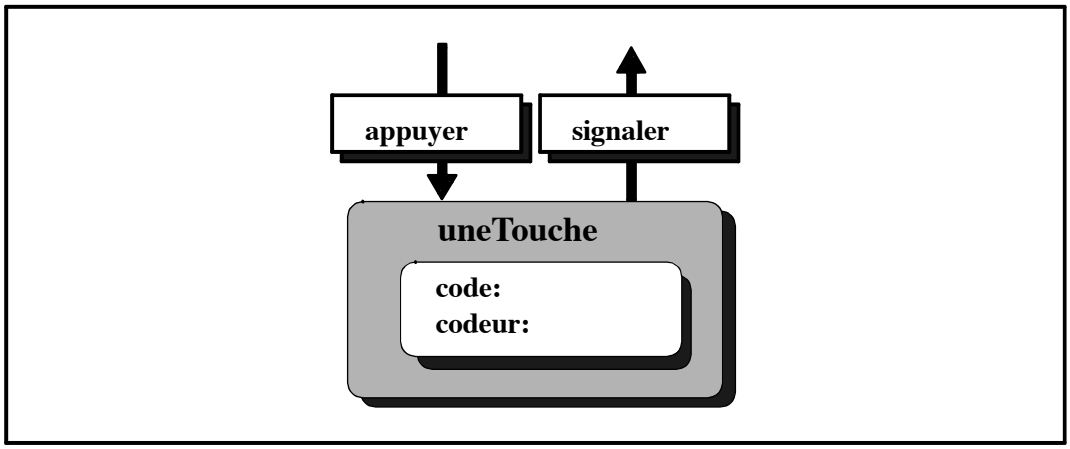


Fig. 38 : Structure d'une instance de Touche.

La structure d'une instance de **Codeur** est celle de la figure 39, page 182. L'attribut `cle`

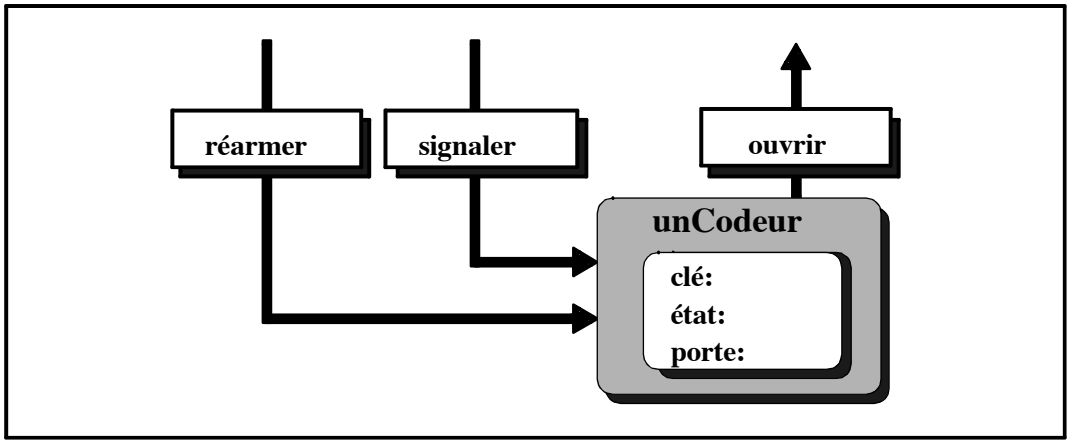


Fig. 39 : Structure d'une instance de Codeur.

représente la séquence des touches sur lesquelles il faut appuyer pour ouvrir la porte et l'attribut `etat` représente ce qu'il reste de la séquence-clé à faire pour ouvrir la porte.

Ces spécifications mettent en évidence une difficulté qui se pose toujours lorsqu'on conçoit une application mettant en œuvre des objets possédant des accointances. L'application n'est dans un état cohérent que lorsque toutes les relations d'accointance sont en place. D'un autre coté, il est fréquent que ces relations d'accointance soient cycliques (ici, porte et codeur), dans ce cas, on ne peut pas créer tous les objets du cycle directement associés à leur accointances.

Il sera nécessaire que certains d'entre eux *s'enregistrent* auprés de leurs accointances au cours de leur processus d'instanciation. Nous dirons, par exemple dans le cas présent, que c'est la porte qui s'enregistre auprès de son codeur. L'ordre dans lequel les objets sont instanciés n'est pas indifférent et il est fondamental, avant d'aller plus loin, de vérifier que tous les cycles de relation d'accointance peuvent être résolus dans l'architecture choisie.

Simulons ces spécifications et à titre d'exemple, définissons une serrure codée à 10 touches.

```
(define le-verrou (codeur 'new '(1 6 3 3)))
(define la-porte (porte 'new le-verrou))
```

Lorsque le codeur est instancié, il n'est pas «terminé» puisqu'il ne peut pas encore connaître «sa porte». Ce n'est que lorsque la porte aura été instanciée et se sera enregistrée auprès de celui-ci qu'il sera dans un état cohérent.

```
(define t0 (touche 'new 0 le-verrou))
(define t1 (touche 'new 1 le-verrou))
(define t2 (touche 'new 2 le-verrou))
(define t3 (touche 'new 3 le-verrou))
(define t4 (touche 'new 4 le-verrou))
(define t5 (touche 'new 5 le-verrou))
(define t6 (touche 'new 6 le-verrou))
(define t7 (touche 'new 7 le-verrou))
(define t8 (touche 'new 8 le-verrou))
(define t9 (touche 'new 9 le-verrou))
```

Une ouverture de la porte donnerait lieu au dialogue suivant

```
(t1 'appuyer)                    ⇒ BEEP
(la-porte 'entrer?)              ⇒ "...desole, fermee !"
(t6 'appuyer)                    ⇒ BEEP
(t3 'appuyer)                    ⇒ BEEP
(t3 'appuyer)                    ⇒ BRRR
(la-porte 'entrer?)              ⇒ "je vous en prie..."
(laporte'fermer)                 ⇒ CLAP
```

Cette forme d'analyse d'une application informatique, fondée sur la définition d'entités qui coopérent en échangeant des messages, est souvent appelée **programmation guidée par les évènements**. Cette approche est très efficace pour le développement d'application interactives; en particulier, tous les logiciels appelés: *Interfaces Graphiques Interactives* [5] (GUI - Graphical User Interface) ont été conçus selon ce principe pour lequel ils ont servi de champ d'expérience.

[5] Les plus connus sont Windows, X-Windows, Motif, Open-look. Mais, à tout seigneur tout honneur, c'est le système d'exploitation des Macintosh lui-même issu de l'environnement Smalltalk développé par le Xerox PARC (Palo Alto Research Center), sur une idée de Alan Kay, qui a fait la démonstration de la valeur extraordinaire de ces interfaces graphiques, en particulier, et de l'approche objet en général.

8.3.2 Spécifications détaillées

Les classes étant considérées comme des «moules à instances», les attributs des classes que nous allons définir vont représenter toute l'information qu'il est nécessaire de connaître pour construire une instance. En particulier, les méthodes publiques associées aux instances seront des attributs des classes.

En effet, comme toutes les instances d'une classe peuvent partager les mêmes méthodes publiques, il n'est pas astucieux de créer ces méthodes au moment où l'instance est crée car, dans ce cas, il s'en créera autant que d'instances.

En faisant des méthodes publiques des attributs de la classe, nous forçons toutes les instances à partager les mêmes procédures qui ne sont ainsi créées qu'une seule fois.

Spécification détaillée de la Classe Porte :

```
(define porte (objet))
```

Les méthodes d'ouverture, de fermeture et d'entrée sont

```
(porte 'def 'ouverture
       (lambda (self super args)
         (self 'set 'etat 'ouverte)
         'brrr)))

(porte 'def 'fermeture
       (lambda (self super args)
         ((self 'get 'codeur) 'rearmer)
         (self 'set 'etat 'fermee)
         'clap))

(porte 'def 'entree
       (lambda (self super args)
         (let ((etat (self 'get 'etat)))
           (cond ((eq? etat 'ouverte)
                  "je vous en prie...")
                 ((eq? etat 'fermee)
                  "...desole, fermee !")))))
```

La méthode de classe pour la création des instances de **Porte** est, quant à elle

```
(porte 'def 'new
       (lambda (self super args)
         (let ((le-codeur (car args))
               (une-porte (objet)))
           (une-porte 'def 'codeur  le-codeur)
           (une-porte 'def 'etat 'fermee)
           (une-porte 'def 'ouvrir (self 'get 'ouverture))
           (une-porte 'def 'fermer (self 'get 'fermeture))
           (une-porte 'def 'entrer? (self 'get 'entree))
           (le-codeur 'set 'porte une-porte)
           une-porte)))
```

Le message de création d'une porte est de la forme

```
(porte 'new un-codeur)
```

Nota: *Comme une porte ne peut être crée qu'après l'instanciation de son codeur, il faut qu'elle s'enregistre auprès de ce codeur.*

Spécification détaillée de la Classe Touche :

```
(define touche (obje))
```

la méthode d'instance pour appuyer sur une touche est

```
(touche 'def 'appui
        (lambda (self super args)
          ((self 'get 'codeur) 'signaler (self 'get 'code))))
```

La méthode de création des instances de **Touche** est, quant à elle

```
(touche 'def 'new
        (lambda (self super args)
          (let ((mon-code (car args))
                (mon-codeur (cadr args))
                (une-touche (objet)))
            (une-touche 'def 'code mon-code)
            (une-touche 'def 'codeur mon-codeur)
            (une-touche 'def 'appuyer (self 'get 'appui))
            une-touche)))
```

Le message de création d'une touche est de la forme

```
(touche 'new 8 le-codeur)
```

Spécification détaillée de la Classe Codeur :

```
(define codeur (objet))
```

L'état du codeur représenté par l'attribut `etat` contient la liste des touches sur lesquelles il faut encore appuyer. Chaque fois que le codeur reçoit la signalisation d'une touche, il vérifie que l'identificateur de la touche est égal à la tête de liste de `etat`.

```
(codeur 'def 'signalisation
        (lambda (self super args)
          (let ((le-code (car args))
                (mon-etat (self 'get 'etat))
                (ma-porte (self 'get 'porte)))
            (if (eq? le-code (car mon-etat))
                (self 'set 'etat (cdr mon-etat))
                (self 'rearmer))
            (if (null? (self 'get 'etat))
                (ma-porte 'ouvrir)
                'beep))))
```

La méthode de réarmement du codeur est

```
(codeur 'def 'rearmement
        (lambda (self super args)
          (self 'set 'etat (self 'get 'cle))
          'ok))
```

La méthode de création des instances de **Codeur** les met dans l'état initial

```
(codeur 'def 'new
        (lambda (self super args)
          (let ((la-cle (car args))
                (un-codeur (objet)))
            (un-codeur 'def 'cle la-clé)
            (un-codeur 'def 'etat la-cle)
            (un-codeur 'def 'porte nil)
            (un-codeur 'def 'signaler
```

```
                    (self 'get 'signalisation))
          (un-codeur 'def 'rearmer
                    (self 'get 'rearmement))
          un-codeur)))
```

Le message de création d'un codeur et de sa clé est de la forme

```
(codeur 'new '(1 6 3 3))
```

8.4 Un petit Coup d'Oeil en Arrière

Mais nous aurions très bien pu réaliser l'implémentation de la serrure à code en n'utilisant que des **type abstraits de données mutables** car les objets impliqués ne nécessitent pas l'héritage.

Ainsi, sur le cahier des charges précédent, et à partir de la **même analyse**, nous allons redéfinir toute l'application précédente en s'appuyant uniquement sur des structures de données. Le dialogue ne pouvant plus être assuré par une transmission de messages, les messages des spécifications-objet sont remplacés par des fonctions. De même, les classes définies dans les spécifications détaillées sont remplacées par la définition du constructeurs des structures de données correspondantes.

Spécifications détaillées de Porte

Une porte est définie par son constructeur.

```
(define porte
  (lambda (codeur)
    (let* ((etat 'fermee)
           (etat! (lambda (e) (set! etat e)))
           (self  (lambda (msg)
                    (cond ((eq? msg 'etat) etat)
                          ((eq? msg 'etat!) etat!)))))
      ((codeur 'porte!) self)
      self)))
```

Les messages «ouvrir», «fermer» et «entrer?» sont implantés à travers des procédures-messages.

```
(define porte-ouvrir
  (lambda (porte)
    ((porte 'etat!) 'ouverte)
    'brrr))

(define porte-fermer
  (lambda (porte)
    ((porte 'etat!) 'fermee)
    'clap))

(define porte-entrer
  (lambda (porte)
    (let ((etat (porte 'etat)))
      (cond ((eq? etat 'ouverte)
             «je vous en prie...»)
            ((eq? etat 'fermee)
             «...desole, fermee !»)))))
```

Spécifications détaillées de Codeur

Un codeur est défini par son constructeur.

```
(define codeur
  (lambda (cle)
    (let* ((etat   cle)
           (etat!  (lambda (e) (set! etat e)))
           (porte  nil)
           (porte! (lambda (p) (set! porte p))))
      (lambda (msg)
        (cond ((eq? msg 'cle)    cle)
              ((eq? msg 'etat) etat)
              ((eq? msg 'etat!) etat!)
              ((eq? msg 'porte) porte)
              ((eq? msg 'porte!) porte!))))))
```

Les messages «réarmer» et «signaler» sont implantés à travers des procédures-messages.

```
(define codeur-rearmer
  (lambda (codeur)
    ((codeur 'etat!) (codeur 'cle))
    'ok))

(define codeur-signaler
  (lambda (code codeur)
    (let ((etat (codeur 'etat)))
      (if (eq? code (car etat))
          ((codeur 'etat!) (cdr etat))
          (codeur-rearmer codeur))
      (if (null? (codeur 'etat))
          (porte-ouvrir (codeur 'porte))
          'beep))))
```

Spécifications détaillées de Touche

Une touche est définie par son codeur.

```
(define touche
  (lambda (code codeur)
    (lambda (msg)
      (cond ((eq? msg 'code) code)
            ((eq? msg 'codeur) codeur)))))
```

Le message «appuyer» est implantés à travers une procédure.

```
(define touche-appuyer
  (lambda (touche)
    (codeur-signaler (touche 'code)
                     (touche 'codeur))))
```

Une touche n'étant crée que pour gérer qu'un seul message, il est possible d'en simplifier considérablement la définition en rassemblant le constructeur et la procédure-message.

```
(define touche-appuyer
  (lambda (code codeur)
    (lambda () (codeur-signaler code codeur))))
```

Nous verrons pourtant, un peu plus loin, un bonne raison de ne pas utiliser cette simplification.

Un petit scénario d'ouverture de la porte

A titre démonstratif, redéfinissons la serrure codée à 10 touches «non simplifiées»

```
(define verrou   (codeur '(1 6 3 3)))
(define la-porte (porte verrou))

(define t0 (touche 0 verrou))
(define t1 (touche 1 verrou))
(define t2 (touche 2 verrou))
(define t3 (touche 3 verrou))
(define t4 (touche 4 verrou))
(define t5 (touche 5 verrou))
(define t6 (touche 6 verrou))
(define t7 (touche 7 verrou))
(define t8 (touche 8 verrou))
(define t9 (touche 9 verrou))
```

Une ouverture de la porte donnerait lieu à un dialogue très analogue au précédent.

```
(touche-appuyer t1)                ⇒ BEEP
(porte-entrer la-porte)            ⇒ "...desole, fermee !"
(t6 touche-appuyer )               ⇒ BEEP
(t3 touche-appuyer )               ⇒ BEEP
(t3 touche-appuyer )               ⇒ BRRR
(porte-entrer la-porte)            ⇒ "je vous en prie..."
(porte-fermer la-porte)            ⇒ CLAP
```

Cet exercice pourrait sembler stérile car pourquoi refaire d'une autre façon ce qui a déjà été fait?

En fait il s'agissait simplement de faire remarquer que le concept d'objet peut exister même en l'absence d'un outil permettant de programmer» objets», il n'est pas nécessaire de disposer d'un langage «objets» officiel [6] pour utiliser le concept d'objet dans l'analyse des problèmes et tous les langages de programmation permettent de bâtir tout ou partie d'un support pour le développement d'objets.

Ce chapitre nous a ainsi permis d'introduire un style de programmation très efficace, la **programmation orientée objets**. Nous l'avons introduite en définissant une extension objet à *Scheme* fondée sur la notion de **classe**, d'**instance**, d'**attribut**, de **méthode** et surtout de l'**héritage** fondé sur le concept de **lien dynamique**.

En résumé :

Un objet est donc un type abstrait de données mutables qui interprète des messages et qui est doté de l'héritage des attributs et des méthodes.

8.5 Exercices

Pour les exercices impliquant une analyse par approche objet, on suppose disponible le langage objet qui a été défini en cours. Il comporte :

`objet` constructeur d'un objet de base sans attributs.

Tout objet reconnaît a priori les messages suivants:

[6] Smalltalk, Flavors, C++, Pascal-Objet, Eiffel...

`def`	définition d'un nouvel attribut ou d'une nouvelle méthode publique.
`set`	modification de la valeur d'un attribut.
`get`	consultation de la valeur d'un attribut.

E-84 Quels sont les inconvénients qui peuvent être rencontrés du fait que les attributs et les méthodes publiques ont été placés dans le même dictionnaire :

1. sur un plan purement fonctionnel,
2. sur le plan de l'éfficacité.

E-85 Comment peut-on remédier simplement aux inconvénients fonctionnels identifiés à l'exercice précédent sans pour autant définir deux dictionnaires séparés.

E-86 Il est clair que la structure de dictionnaire est une structure de donnée fondamentale. En définir une version objet en définissant une classe **Dictionnaire** afin que toutes les applications puissent en profiter. Une instance de **Dictionnaire** disposerait des methodes `'definir`, `'affecter`, `'consulter` et `'pour-tous-faire`.

E-87 En utilisant la classe Dictionnaire définie ci-dessus, définir un objet **Méta-Classe** qui , comme les classes permettent d'instancier des objets, permettrait d'instancier des classes.

E-88 Reprendre la définition de la structure de pile et définir une classe **Pile**.

E-89 Reprendre la définition des nombres rationnels vue au chapitre *Les Données* et en déduire une classe **NombreRationnel**.

E-90 Reprendre les spécifications de l'ensemble défini au chapitre *Représentation des Données abstraites* et définir une classe **Ensemble**.

E-91 Reprendre les spécifications de la file définie au chapitre *Les Structures mutables* et définir une classe **File**.

E-92 La plupart des parkings sont équipés d'un système permettant l'affichage permanent du nombre des places disponibles. Ce système est composé du *distributeur de tickets* situé à l'entrée du parking, d'un *détecteur de passage* situé en sortie du parking, d'un *afficheur* situé à l'extérieur (sur le chemin de l'entrée) et d'un *mécanisme de comptage*.

1. Etablir la liste des objets de l'application.
2. Etablir le dialogue permettant la coopération entre ces objets.
3. Définir les classes associées aux objets de l'application.
4. Définir un scénario démontrant le fonctionnement de cette application.

E-93 Reprendre les exercices *E-66* et *E-67* du chapitre *Eléments de Programmation* et en donner une solution objet.

Nota: *il est commode de définir une classe* **Pièce** *et une classe* **Monnayeur**.

E-94 Comment faut-il définir une classe et ses instances associées pour qu'un attribut de la classe, alors appelé *variable de classe*, soit partagé par toutes les instances de la classe?

Illustrer ce mécanisme en créant la classe **Facture** dont les instances sont définies de la façon suivante:

variable de classe :	taux de tva
variables d'instance :	liste d'écritures, total HT, total TTC.

méthodes :	`'ajouter` - ajoute une écriture à la facture et met le total HT et TTC à jour.
	`'afficher` - affiche la facture.

Il est commode de considérer que les écritures sont des objets et de créer une classe **Ecriture** telle que ses instances sont définies de la façon suivante:

variables d'instances :	dénomination, prix unitaire HT, quantité.
méthode :	`'evaluer` - détermine le prix total HT

E-95 De nombreux langages objet permettent l'héritage multiple des attributs et des méthodes. En d'autres termes, un objet peut hériter de plusieurs ancêtres. Modifier le constructeur d'objets vu en cours de telle sorte qu'il supporte l'héritage multiple.

Lorsqu'il est nécessaire d'aller consulter les ancêtres pour rechercher un attribut ou une méthode non disponible chez le receveur du message, on interroge les ancêtres **dans l'ordre où ils ont été définis**.

Opinion personnelle *: La notion d'héritage n'est déjà pas particulièrement facile à utiliser, alors, à notre avis, l'emploi de l'héritage multiple est presque toujours le signe d'une analyse mal conduite.*

8.6 Annexes

8.6.1 Constructeur d'Objets

```
(define objet
  (lambda ascendant
    (letrec ((attributs (dictionnaire-vide))
             (definir   (lambda (nom valeur)
                          (let ((cr (dictionnaire-definir
                                      nom valeur attributs)))
                            (if (eq? cr 'deja-defini)
                                (error "deja defini : " nom)
                                'ok))))
             (affecter  (lambda (nom valeur)
                          (let (( cr (dictionnaire-affecter
                                       nom valeur attributs)))
                            (if (eq? cr 'non-defini)
                                (super 'set nom valeur)
                                'ok))))
             (consulter (lambda (nom)
                          (let ((valeur (dictionnaire-consulter
                                          nom attributs)))
                            (if (eq? valeur 'non-defini)
                                (super 'get nom)
                                valeur))))
             (super     (if (null? ascendant)
                            (lambda (id nom . args)
                              (error "symbole inconnu : " nom))
                            (car ascendant)))
             (self      (lambda (id . args)
                          (let ((nom     (car args))
                                (valeur  (cadr args)))
                            (cond ((eq? id 'def)
                                   (definir   nom valeur))
                                  ((eq? id 'set)
                                   (affecter  nom valeur))
                                  ((eq? id 'get)
                                   (consulter nom))
                                  (else
                                    ((consulter id)
                                       self super args)))))))
      self)))
```

8.6.2 Dictionnaire des Attributs & des Méthodes

Sa définition ne pose pas de problème particulier car il est très semblable à ceux que nous avons déjà vus.

```
(define dictionnaire-vide
  (lambda () (reference-a nil)))

(define dictionnaire-consulter
  (lambda (s &d)
    (let ((p (assq s (&d))))
      (if p (cdr p) 'non-defini))))
```

```
(define dictionnaire-definir
  (lambda (s v &d)
    (let ((p (assq s (&d)))
          (q (cons s v)))
      (if p
          'deja-defini
          (ref! &d (cons q (&d)))))))

(define dictionnaire-affecter
  (lambda (s v &d)
    (let ((p (assq s (&d)))
          (q (cons s v)))
      (if p
          (set-cdr! p v)
          'non-defini))))
```

Lors de l'implémentation d'un langage objet réel, la réalisation de ce dictionnaire doit être particulièrement soignée car c'est le passage obligé de la gestion des attributs et des méthodes et ses performances sont critiques.

9. Spécifier puis Implémenter.

Jusqu'à présent nous avons beaucoup parlé d'applications mais nous avons (volontairement) complètement oublié la machine (l'ordinateur) et nous avons fait comme si la machine de nos rêves existait. Malheureusement ce n'est pas encore tout à fait vrai et nous n'avons su réaliser (vers 1945) qu'un seul type de machine que nous savons à peu près bien maîtriser: le *processeur de Von Neumann* dont le fonctionnement est uniquement fondé sur l'affectation.

Ce processeur ne permet pas de manipuler directement toutes les abstractions que nous avons imaginées et il va falloir les construire. Le résultat en est un autre style de programmation: la *programmation impérative*.

Ce style de programmation est lié à un type de processeur, c'est son *mode d'emploi* plus ou moins directement exprimé. C'est le style de programmation qu'il est le plus facile d'acquérir sur le tas. Malheureusement, ce style, pratiqué seul, est vite **inefficace** dans un cadre professionnel. C'est la raison de notre insistance sur les techniques de spécification que nous avons développées et du langage *Scheme* utilisé essentiellement comme outil de raisonnement.

Ce dernier chapitre montre donc comment on peut traduire les différents styles de programmation que nous avons introduits en utilisant une programmation impérative illustrée à l'aide d'un langage bien adapté: le **langage C**. Nous allons ainsi retrouver les notions de *type de données*, de *fonction*, d'*environnement*, d'*effets de bord*, d'*état* et de *transmission de messages*.

Ce chapitre n'a pas la prétention d'apprendre à utiliser le *langage C* mais seulement de montrer que les spécifications d'une application élaborées en utilisant le langage *Scheme* peuvent être transposées en *langage C* ou en tout autre langage de la même nature (Pascal, Ada...).

Nous allons uniquement procéder par analyse d'exemples démontrant les principales techniques de transposition utilisables. Nous ne serons pas exhaustifs mais simplement démonstratifs.

Avant de commencer, rappelons le cadre du **développement professionnel** d'une application informatique en introduisant quelques idées qui seraient à développer considérablement dans le cadre d'un cours consacré au *Génie Logiciel*.

Le travail de l'informaticien se situe soit au niveau de l'analyse et de la conception, soit au niveau de l'implémentation, soit au niveau de la finalisation. Il est classique de faire ses classes au niveau de la finalisation puis au niveau du codage avant de prendre la responsabilité d'une analyse et d'une conception.

Une bonne conception a pour objectif de minimiser les coûts directs du développement (coût de l'élaboration des spécifications détaillées et de l'implémentation) et les coûts indirects dus aux aléas de la finalisation. La plupart du temps, les projets informatiques achoppent au niveau des coûts indirects très difficiles à évaluer au moment de la construction du devis qui va fixer le prix du logiciel.

Une analyse propre et complète ainsi qu'une programmation simple et lisible contribuent à minimiser les coûts indirects de développement. De même que l'analyse et la conception d'une application sont grandement facilitées par la connaissance d'un ensemble de **bons coups** une bonne programmation suppose la connaissance des bons coups.

Les bons coups sont de deux natures :

coups tactiques	ils font avancer la solution du problème. Définir un type abstrait de données, définir une fonction sont des coups tactiques.
coups stratégiques	ils empêchent la solution de reculer en mettant en place des barrières. Définir un paquetage pour empêcher la propagation des erreurs, concevoir une architecture simple sont des coups stratégiques.

9.1 Les Eléments de construction d'une Application C.

9.1.1 Expressions.

L'expression est la plus petite entité de structuration d'une application *C*. Elle est construite à partir de l'application de fonctions à leurs arguments. On peut donc la rapprocher d'une expression *Scheme* avec cependant une différence de forme importante.

Sans autre précision, une instruction sera notée

```
<expression>
```

Voici quelques exemples d'expressions *C*

```
5
y+6*z
(a+b)*(c-d)
sin(x)
```

De nombreux opérateurs sont utilisés sous forme **infixée** avec priorité et associativité implicite (en général à gauche) tandis que les fonctions sont utilisée en notation parenthésée habituelle. Une expression *C* est, comme une expression *Scheme*, utilisée soit pour produire une valeur soit pour produire un effet de bord.

Ainsi

```
afficher("Bonjour à tous");
```

ne produit qu'un effet de bord.

Certaines expressions rendent une valeur interprétée comme les valeurs *vrai* et *faux*, ces expressions sont les prédicats notés

```
<prédicat>
```

Les prédicats sont construits à partir des opérations de relation `==` (égal?) `!=` (différent?) `<` (inférieur?) `<=` (inférieur ou égal ?) `>` (supérieur?) `>=` (supérieur ou égal?) et des opérateurs logiques `|` (pas) `&&` (et) `||` (ou).

9.1.2 Instructions.

L'instruction est la plus petite entité de structuration d'une application *C*. On peut la rapprocher d'une expression *Scheme* avec cependant une différence fondamentale: alors que les expressions *Scheme* sont conçues, en général, pour être simplement évaluées et produire une valeur, les instructions *C* impliquent presque systématiquement une évaluation associée à une affectation et sont donc très souvent équivalentes à `set!`.

Sans autre précision, une instruction sera notée

```
<instruction>;
```

La **fin** de chaque instruction *C* est signalée par `;`.

Voici quelques exemples d'instructions *C*

```
x = 5;
x = y+6*z;
x = (a+b)*(c-d);
```

Ce sont des instructions d'affectation construites selon le modèle

```
<expression de gauche> = <expression de droite>;
```

et qui fonctionnent selon le mécanisme suivant:

1. L'expression de droite et l'expression de gauche sont évaluées **dans un ordre quelconque**,
2. l'expression de droite est ensuite **affecté** à ce que représente le résultat de l'évaluation de l'expression de gauche.

L'expression de gauche doit nécessairement représenter **un moyen d'accès à une donnée** [1]. Nous reviendrons, bien sûr, sur les principales techniques permettant de définir l'accès à une variable. Ici nous avons utilisé le fait qu'un *nom* (symbole) permet d'accéder à une donnée.

Une expression *C* est, comme une expression *Scheme*, utilisée pour produire une valeur ou pour produire un effet de bord. Ainsi l'expression de droite de

[1] Le résultat de l'évaluation du terme de gauche doit être ce que les concepteurs du langage C ont appelé une *l-value*.

```
x = 4*sin(6*t);
```

produit une valeur affectée à la variable `x`, tandis que

```
afficher("Bonjour à tous");
```

ne produit qu'un effet de bord. La partie gauche d'une instruction est donc facultative.

9.1.3 Déclarations.

Une déclaration sert à définir une entité manipulée par les instructions, elle est analogue à une expression *Scheme* contenant un `define` ou un `let`.

Sans autre précision, une déclaration sera notée

```
<déclaration>;
```

Voici quelques exemples de déclarations *C*

```
int foo = 5,
    bar = 8;
float baz;
```

tout à fait analogues à l'expression *Scheme*

```
(define foo 5)
```

Nous reviendrons longuement sur le rôle joué par les déclarations et sur la manière de les écrire.

9.1.4 Blocs de code

Le *bloc de code* est la plus petite entité de structuration d'une application *langage C*. C'est une séquence de déclarations et d'instructions.

```
{   <déclaration>;
    ....
    <declaration>;
    <instruction>;
    ....
    <instruction>;}
```

Une instruction pouvant être un bloc de code on peut engendrer des structures emboîtées de la forme:

```
{   <déclaration>;
    {   <déclaration>;
        <instruction>;
        <instruction>;}
    {   <déclaration>;
        <instruction>;
        <instruction>;}
    <instruction>;
    <instruction>;}
```

Il est de bonne discipline d'*indenter* les blocs afin de les mettre clairement en évidence. Nous introduirons chaque fois que l'occasion se présentera des règles de bonne écriture. Le *langage C* étant en format libre (on peut intercaler autant d'espaces, de tabulations ou de retour à la ligne que nous le désirons), ces règles peuvent légèrement varier d'un groupe d'individus à l'autre.

Sans autre précision, nous noterons un bloc de code

```
<bloc de code>
```

Un bloc de code est analogue à la construction *Scheme* suivante

```
(let ((...)
      (...)
      (...))
  ......)
```

Nota: *les blocs de code C ne sont pas des objets de 1ère classe, ils ne peuvent ni être affectés à une variable, ni être invoqués. Ils ne rendent pas de valeur.*

La définition d'un bloc de code peut être considéré soit comme un coup tactique s'il permet, par exemple, d'introduire une nouvelle abstraction soit comme un coup stratégique s'il est simplement utilisé comme une barrière anti-erreurs.

9.1.5 Fonctions & Procédures

Une fonction-procédure *langage C* est essentiellement un *bloc de code nommé* associé à des *paramètres*. Comme en *Scheme*, on parlera de *fonction* lorsque ce bloc de code rend une valeur et n'est le siège d'aucun effet de bord et de *procédure* dans le cas contraire.

Voici, par exemple, un fonction *Scheme*

```
(define (factorielle n)
  (if (= n 1)
    1
    (* n (factorielle (- n 1))))))
```

et son équivalent *C*

```
....
factorielle(n)
    ....
{
    if (n == 1) {
        return(1);
    } else {
        return(n*factorielle(n-1));
    }
}
```

Cet exemple induit deux remarques:

1. l'application de la fonction `factorielle` à son argument est notée `factorielle(n)` comme en mathématique.
2. la fin de l'évaluation de la fonction et la valeur rendue sont indiquées explicitement comme l'argument de la fonction implicite `return(...)`.

On peut également noter, au passage, l'apparition du prédicat `(n == 1)` et de la forme spéciale

```
if <prédicat> <bloc de code> else <bloc de code>
```

dont on note la bonne écriture

```
if <prédicat> {
    <déclaration>;
    ....
    <instruction>;
```

```
} else {
    <déclaration>;
    ....
    <instruction>;
}
```

La construction des fonctions est assurée par la même opération de composition qu'en *Scheme* et le corps d'une fonction peut contenir des invocations à d'autres fonctions. Les définitions récursives sont bien entendu possibles mais **elles engendrent toujours un processus de calcul récursif**. Nous verrons un peu plus loin comment engendrer un processus itératif.

9.1.6 Structures de Données

La construction des types abstraits de données en *Scheme* est très naturelle car *Scheme* assure une intendance que le *langage C* (et les autres) ne prend pas en charge.

Reprenons, par exemple, la définition des **Rationnels** en *Scheme*. Cette définition s'appuyait sur un constructeur et deux sélecteurs

```
(define rationnel
  (lambda (num denom)
    (lambda (msg)
      (cond ((eq? msg 'numerateur) num)
            ((eq? msg 'denominateur) denom)))))
```

Ce constructeur définit, à la fois, l'association du numérateur et du dénominateur d'un nombre rationnel et la méthode d'accès au numérateur et au dénominateur. La définition des fonctions de sélection devient alors triviale

```
(define numerateur (lambda (un-rat) (un-rat 'numerateur))
(define denominateur (lambda (un-rat) (un-rat 'denominateur))
```

La définition *C* des **Rationnels** est un peu plus complexe car elle comporte deux parties.

Il faut d'abord **déclarer** l'association constituée du numérateur et du dénominateur qui constitue un nombre rationnel

```
struct rationnel {
    .... numerateur;
    .... denominateur;};
```

On a ainsi décrit ce qu'on avait appelé une structure dans le chapitre *Les Structures*. Cette description était implicite en *Scheme*, elle doit être explicite en *C*. Une fois cette structure définie, on peut lui associer le constructeur et les sélecteurs suivants

```
....
RationnelCreer(num,denom)
    ....
{....}

....
RationnelNumerateur(unRat)
    ....
{....}

....
RationnelDenominateur(unRat)
    ....
{....}
```

On peut constater qu'il reste quelques points de suspension dont il faudra préciser le contenu et le rôle. Considérons l'ensemble des déclarations suivantes

```
struct rationnel {
    .... numerateur;
    .... denominateur;};

.... RationnelCreer(...);
.... RationnelNumerateur(...);
.... RationnelDenominateur(...);
```

elles décrivent uniquement ce que doit connaître un utilisateur de **Rationnels**, on l'appellera l'*interface* de la structure de données. Cette interface sera utilisée par le réalisateur du type **Rationnels** pour décrire les entitées *exportées* (donc accessibles à tous) et par un utilisateur du type **Rationnels** pour annoncer ce qu'il en *importe*.

Par contre

```
....
RationnelCreer(num,denom)
    ....
{....}

....
RationnelNumerateur(unRat)
    ....
{....}

....
RationnelDenominateur(unRat)
    ....
{....}
```

décrit comment le type **Rationnels** a été réalisé, on l'appellera son *implémentation*. Par nature, l'interface est *publique* tandis que l'implémentation est *cachée* (privée).

Si on voulait caractériser en une seule phrase la différence essentielle qui distingue *Scheme* des langages impératifs comme le *langage C* on pourrait dire:

Scheme associe les données aux fonctions, tandis que *C* associe les fonctions aux données.

Sur un plan pratique, cela se traduit par le fait que pour définir un type abstrait de données en *Scheme* on commence par définir son constructeur (fonction) tandis qu'en *langage C*, on commence par définir l'association des données.

9.2 Architecture d'une Application C.

Sur ces bases, on peut énoncer la **règle de bonne architecture** suivante:

Une application *C orthodoxe* est constituée par :

1. *une fonction* - Cette fonction nommée **`main`** est la fonction invoquée au moment du lancement de l'application.
2. *un ensemble de structures de données* - Ces structures de données représentent les entités manipulées par l'application. elles sont, chacune, constitués d'une interface et d'une implémentation.

Les traducteurs du *langage C* sont toujours inclus dans des environnements de développement comportant essentiellement:

- le traducteur (appelé *compilateur*),
- un éditeur de programme,
- un intégrateur de programmes (appelé *éditeur des liens*),
- des outils d'aide à la mise au point,
- un gestionnaire de versions,
- un documenteur,
- ....

Comme le cas de figure qui nous intéresse est celui où l'application à développer est suffisamment importante pour nécessiter la constitution d'une équipe de développement, l'application a été scindée en de nombreux modules (nous avons vu au cours des chapitres précédents comment y parvenir) qui seront implémentés simultanément et indépendamment par plusieurs personnes.

Par convention, une application *langage C* est constituée de l'ensemble [2] des fichiers définissant les interfaces et de l'ensemble des fichiers définissant les implémentations des types abstraits de données de l'application. Les fichiers décrivant une interface sont nommés **`xxxx.h`** et ceux décrivant une implémentation **`xxxx.c`**.

Par exemple, l'interface et l'implémentation des **Rationnels** seraient nommées [3]

```
rationnel.h
rationnel.c
```

Le couple (`rationnel.h`, `rationnel.c`) constitue le *paquetage des rationnels*.

9.3 Processus de Développement d'une Application.

En quelques mots, on peut décrire l'activité d'une équipe de développeurs lorsqu'elle est arrivée en phase d'implémentation.

Les processeurs utilisés pour construire les ordinateurs ont un mode d'emploi qui est traduit sous la forme d'un langage très rustique. Le *langage C* et a fortiori *Scheme* sont très différents et les programmes doivent être traduits dans le langage natif du processeur. C'est le **compilateur** qui est chargé de ce travail.

Chaque fichier **`xxxx.c`** décrivant l'implémentation d'un des types abstraits de données de l'application est traduit pour produire un fichier **`xxxx.o`** qui est en fait **o**rphelin car il ne peut opérer qu'en coopération avec tous ses frères développées par les membres de l'équipe ou même achetés tout fait.

La réunion de tous les orphelins est la phase d'intégration de l'application. C'est l' **éditeurs des liens** qui en est chargé.

Le processus de développement peut être approximativement représenté par la figure 40, page 202.

[2] Il n'est pas rare de rencontrer des applications comportants plusieurs centaines de fichiers.

[3] Les règles de dénomination des fichiers de l'application peut être légèrement modifiée d'un système d'exploitation à l'autre.

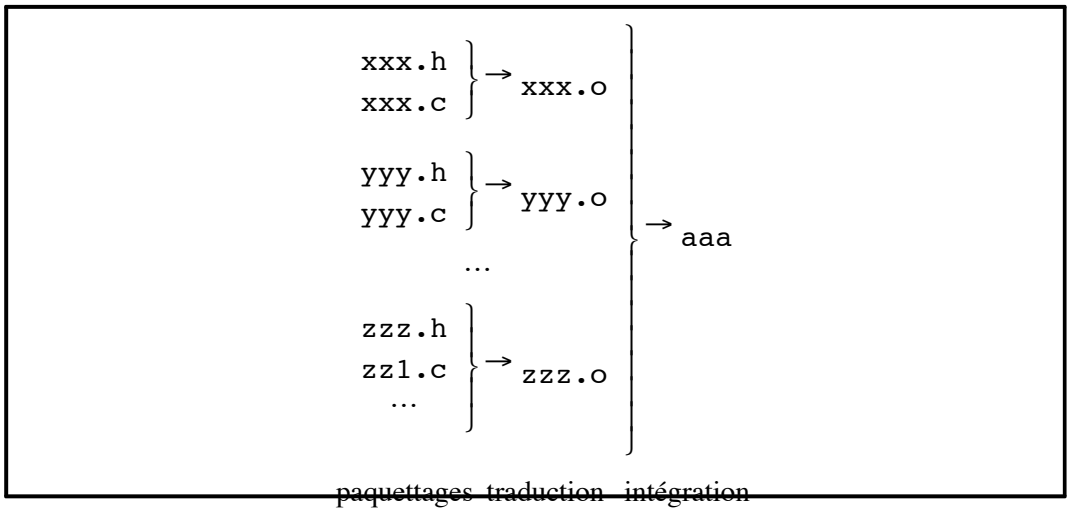


Fig. 40 : Structure d'une application ℂ.

9.4 Constitution des Paquetages.

Imaginons une application utilisant des ensembles. Dans la phase de spécification, nous avons défini les **Ensembles** à partir des **Listes** elles-mêmes définies à partir des **Paires**.

L'interface du type **Paires** est constituée de la déclaration de la structure d'une paire et de la déclaration des fonctions publiques de manipulationd'une paire mises à la disposition des utilisateurs de cette structure. Elle est contenue dans le fichier `paire.h` et ressemble à des déclarations de la forme

```
struct paire {
    .... tete;
    .... queue;};

.... cons(...);
.... car(...);
.... cdr(...);
.... null(...);
```

L'interface du type **Listes** est définies dans le fichier `liste.h`. Cette interface importe les éléments publics du type **Paires** nécessaires à la définition du type **Listes**

```
#include "paire.h"

.... lenght(...);
.... append(...);
.... reverse(...);
.... forEach(...);
```

L'interface du type **Ensembles** est définie dans le fichier `ensemble.h`. Comme ce type est construit à partir des **Listes**, il en importe l'interface

```
#include "liste.h"

struct ensemble {
    .... liste;};

.... EnsembleCreer(...);
.... EnsembleAjouter(...);
```

```
.... EnsembleRetirer(...);
.... EnsembleVide(...);
```

La déclaration d'importation d'une interface est donc

```
#include "xxxxx.h"
```

Le module `monApp.c` utilisant le type **Ensembles** doit importer l'interface associée

```
#include "ensemble.h"

.....
.....
.....
.....
```

Le paquetage **Paires** sera constitué du couple (`paire.h` , `paire.c`), le paquetage **Listes** du couple (`liste.h` , `liste.c`) et **Ensembles** sera constitué du couple (`ensemble.h` , `ensemble.c`).

Si le *langage C* permet de construire des paquetages, c'est au prix d'une discipline volontaire du développeur. L'importance stratégique du paquetage a conduit à l'introduction de ce concept dans certains langages de programmation qui ne le possédaient pas (Pascal) et à concevoir des langages autour de ce concept (Ada).

9.5 Typage explicite des Données.

Toutes les données ont un type, il n'existe donc pas de langage informatique [4] dans lequel les données ne sont pas typées. Par contre il existe *deux conceptions* pour exprimer le fait que toute donnée est typée.

Le concept de données apparait sous deux aspects:

1. la **valeur** de la donnée,
2. le **symbole** qui sert à nommer cette valeur.

Ainsi, nous avons défini deux opérations à propos des données:

définition	qui consiste à donner un nom à une valeur,
affectation	qui consiste à associer une valeur à un nom.

On peut donc définir le type de la données soit en typant son nom, soit en typant sa valeur. La première technique est appelée *typage explicite* tandis que la deuxième est appelée *typage implicite*.

Pour en donner une image frappante on pourrait poser le problème du typage dans les termes suivants:

> Pour organiser une cuisine vaut-il mieux utiliser des boîtes en fer avec des étiquettes dessus (Café, Farine, Sel, Sucre...) ou prendre des boîtes transparentes qui permettent d'en voir le contenu?

Un tel choix est stratégique et il n'y a pas de raison irréfutable permettant de considérer une de ces deux techniques comme meilleure que l'autre. Chacune des deux a ses avantages et

[4] Nous avons exclu de la catégorie des langages informatiques les langages constituant le mode d'emploi des différents processeurs utilisés pour construire des machines informatiques.

ses inconvénients et aucune ne mérite les guerres de religion dont elles sont quelque fois l'objet.

La première permet de se dépanner en toute sécurité en cas de pénurie de boîte, car les boîtes sont, dans l'ensemble, récupérables. Mais comment, simplement par transparence, distinguer sans erreur le sucre du sel?

9.5.1 Types de Base.

Le *langage C* est un langage à **typage explicite**. Ce sont donc les noms des données (les variables) qui sont typés. Ainsi, chaque fois qu'on définira un nouveau nom, il faudra lui associer un type. Ce type est la simple indication de la manière d'interpréter la donnée associée à ce nom. Bien entendu, un traducteur *C* vérifie que les associations de variables sont bien cohérentes relativement à leur type.

9.5.2 Les Données simples.

Le *langage C* permet de définir différents types de données simples à associer à des noms de variables. On dispose essentiellement des types suivants:

`char`	*caractères* - qui représentent le codage des signes typographiques,
`int`	*entiers* - qui représentent les nombres qui tombent juste,
`float`	*réels* - qui représentent les nombres qui ne tombent pas juste.

et un certain nombre de variantes dont nous ne parlerons pas.

On peut donc déclarer, par exemple, deux variables entières `x` et `y` valant respectivement 3 et 4

```
int x = 3;
int y = 4;
```

Cette déclaration aurait également pu s'écrire (notez la virgule)

```
int x = 3,
    y = 4;
```

Les constantes entières sont représentés par des nombres sans virgule (point décimal chez les anglo-saxons)

```
18          56          -34          32500
```

Les constantes réels sont représentées par des nombres avec virgule et avec éventuellement un exposant

```
 12.4       3.14        12.67e6      3.0e-4
```

Les caractères sont représentés par leur dessin typographique entre '

```
'a'         'z'         'A'          '+''4'
```

Il existe des variantes de représentation que nous ne donnerons pas. On peut également définir des noms sans y associer la moindre valeur (demi-définition)

```
int x,y,z;
```

Une valeur pourra leur être associée plus tard par une affectation.

9.5.3 Les Pointeurs.

Alors que *Scheme* gère les environnements à notre place, le *langage C* ne le fait pas et pour permettre cette gestion " à la main ", on dispose d'un type un peu particulier, *la référence à une donnée* appelé communément *pointeur*. Ainsi on pourra définir des références à un caractère, des références à un nombre entier, des références à un nombre réel et , en règle générale, des références à tout type existant.

La définition d'une référence prend la forme suivante

```
<type existant> * <symbole>;
```

Par exemple

```
int * pix;
float * pfy;
char * pcz;
```

`pix`, `pfy` et `pcz` sont respectivement des pointeurs sur (référence à) un entier, un réel et un caractère.

Les pointeurs permettent de mettre en place des mécanismes puissants de gestion des environnements. Le concept de pointeur peut être illustré par l'image suivante:

> Si je veux permettre à un ami d'entrer chez moi, je peux soit lui donner la clé de ma porte (accès direct à mon appartement) soit la clé de ma boite aux lettres dans laquelle j'aurai mis la clé de mon appartement (accès indirect). Ma boite aux lettres est un pointeur sur mon appartement.

En résumé, on peut dire que le pointeur permet des référencement indirects, c'est à dire à *définition différée*. En effet, je peux, au dernier moment, mettre la clé d'un autre appartement dans ma boite aux lettres.

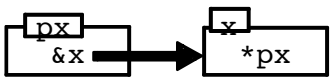
Le concept d'accès est associé aux deux opérateurs `*` et `&`

`&x`	rend un pointeur sur la variable `x`.
`*px`	rend la valeur de la variable dont le pointeur est la valeur de `px`.

On peut, par exemple, écrire

```
int x;
int *px = &x;
```

`px` est alors un pointeur sur `x`. Comme `px` est un pointeur sur `x`, `*px` dénote la valeur de `x`, ce qui peut être illustré par le dessin suivant



Le pointeur étant conçu pour permettre des accès différés, il peut arriver dans quelques cas très intéressants qu'on ne sache pas, au départ, sur quel type d'objets il sera amené à pointer, dans ce cas on utilisera le type fictif `void`[5]

```
void *pp;
```

`pp` est un pointeur sur *on verra bien quoi plus tard* !

Ce type `void` est simplement une indication au traducteur *C* qui lui indique qu'une variable de ce type est réputée compatible avec n'importe quel autre type de variable et que le programmeur prend la responsabilité d'assurer la cohérence de ce qu'il écrit.

[5] Le mot-clé `void` aura une autre signification dans un autre contexte, attention !

9.5.4 Types construits.

Pour pouvoir construire de nouveau types de données, il faut pouvoir *associer* des types déjà définis. Pour cela, on dispose de deux techniques d'association:

tableau dans le cas où l'association est homogène, c'est à dire porte uniquement sur des types identiques. On parlera alors d'un tableau de caractères ou d'un tableau d'entiers.

structure dans le cas où l'association est hétérogène, c'est à dire porte sur des types différents.

On peut comparer les possibilités respectives de Scheme et langage C dans le tableau suivant

Taille	**Scheme**	**langage C**	
fixe	vecteur	homogène	tableau
		hétérogène	structure
variable	liste	à définir	

Tableaux

Un tableau est une association constituée d'un nombre fini d'éléments homogènes (du même type). La déclaration d'un tableau de *quelque chose* a la forme suivante

```
int x[10];
char msg[25];
float taux[5];
```

définissent respectivement un tableau de 10 entiers nommé `x`, un tableau de 25 caractères nommé `msg` et un tableaux de 5 réels nommé `taux`. L'expression `[n]` est le constructeur d'une association de `n` éléments identiques du type de l'expression immédiatement à gauche du nom de l'objet.

Par exemple

```
int x[10];
```

construit une association nommée `x` de 10 éléments de type `int`. On peut décrire des type plus complexes à l'aide de ce mécanisme

```
char roman[5][10][15][80];
```

peut s'interpréter en disant (en partant de la droite) qu'un `roman` est constitué de paquets de 80 caractères (les lignes) regroupés par paquets de 15 (les paragraphes) regroupés par paquets de 10 (les chapitres) eux-mêmes regroupés par paquets de 5.

L'accès à un élément d'un tableau se fait par son indice (celui du premier élément est 0)

```
x[6]        msg[12]     taux[4]
```

sont respectivement l'élément n°6 du tableau `x`, l'élément n°12 du tableau `msg` et le dernier élément du tableau `taux`. Bien entendu, l'indice utilisé peut être une variable entière (attention à ne pas déborder).

Le concept de tableau est identique à celui de vecteur *Scheme*

```
(define x (make-vector 10))
```

produit le même effet que la déclaration *C*

```
int x[10];
```

tandis que les expressions *Scheme*

```
(set! y (vector-ref x 5))
(vector-set! x 5 7)
```

s'évaluent respectivement comme les instructions *C*

```
y = x[5];
x[5]=7;
```

Structure

Une structure est l'association d'un nombre fini d'éléments éventuellement hétérogènes (de types différents). La déclaration d'une nouvelle structure a la forme suivante

```
struct rationnel {
    int numerateur;
    int denominateur;
 };
```

le nom du type qui vient ainsi d'être créé est `struct rationnel` et ses deux composantes sont nommées `numerateur` et `denominateur`. La définition d'un nombre rationnel est identique à celle d'une donnée appartenant à un type de base

```
struct rationnel x = {4,5};
```

définit le nombre rationnel `x` valant `4/5`. On peut, bien entendu, définir un pointeur sur un nombre rationnel

```
struct rationnel *px = &x;
```

La déclaration d'une nouvelle structure associée à la définition d'un nouvel élément équivaut au constructeur correspondant de *Scheme*.

La désignation d'une composante de la structure utilise les notations pointées `x.numerateur` et `x.denominateur` qui représentent respectivement la composante `numerateur` et la composante `denominateur` du nombre rationnel `x`.

Si on suppose qu'il existe un fonction d'affichage `EntierAfficher()`, on obtiendrait

```
EntierAfficher(x.numerateur);          ⇒ 4
EntierAfficher(x.denominateur);        ⇒ 5
```

L'accès aux composantes du nombre rationnel `x` peut, également, être effectué à partir du pointeur `px`

```
EntierAfficher((*px).numerateur);      ⇒ 4
EntierAfficher((*px).denominateur);    ⇒ 5
```

Il existe une notation équivalente

```
px->numerateur
```

La notation pointée joue un rôle de *sélecteur* lorsqu'elle apparait au sein d'une expression à évaluer et de *modificateur* lorsqu'elle apparaît comme membre de gauche d'une affectation

```
EntierAfficher(x.numerateur);          ⇒ 4

x.numerateur = 6;
EntierAfficher(x.numerateur);          ⇒ 6
```

Voici, par exemple, une structure `paire` qui permettrait de construire des listes de caractères [6]

```
struct paire {
    char tete;
    struct paire *queue;
};
```

Si on suppose que `NULL` représente une liste vide, les déclarations

```
struct paire p1 = {'d',NULL},
             p2 = {'c',&p1},
             p3 = {'b',&p2},
             lc = {'a',&p3};
```

permettent de construire la liste `lc` qui contient les 4 caractères `a`, `b`, `c` et `d`.

Enumération.

Nous avons vu au chapitre *Abstraire par les Fonctions* qu'on pouvait définir un ensemble soit à partir d'une propriété commune à tous ses éléments soit en énumérant les éléments qui en font partie. On peut, sur ce principe, définir de nouveau type de données en *langage C* en énumérant simplement les valeurs que les données de ce type peuvent prendre.

Une telle déclaration a la forme suivante

```
enum booleen {vrai,faux};
```

`vrai` et `faux` sont deux nouvelles *constantes* qui vont permettre de donner une valeur aux données de type booléen.

Par exemple, on peut écrire

```
enum booleen a = vrai,
             b = faux,
             c = vrai;
```

Le nom du type qui vient ainsi d'être créé est `enum booleen`. On peut de cette manière définir des types très pratiques

```
enum jour {
    lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche
};
```

ou

```
enum couleur {
    rouge,jaune,vert,bleu,violet
};
```

La définition d'un ensemble énuméré permet de définir des symboles en *langage C* comme le permet la quote-ation *Scheme*. En particulier, on peut définir les identificateurs de message de cette manière

```
enum symbole {
    voir,prendre,extraire,afficher,set,get
};
```

[6] On retrouve à cette occasion le fait que le *langage C* type explicitement les noms de ses données. cela ne nous permet donc pas de définir une paire de n'importe quoi. Nous avons définit une paire pour construire des listes de caractères, nous ne pourrions pas avec cette paire-là construire des listes contenant à la fois des caractères et autre chose.

9.5.5 Nommage des Types.

Les types de données construits sous la forme de structures ont des noms peu commodes et il est intéressant, car cela améliore la lisibilité [7] des programmes, de leur attribuer un nom plus représentatif. On peut définir des synonymes pour désigner un type existant à l'aide de l'opérateur de nommage `typedef` dont la forme générale est la suivante

```
typedef <nom du type> <synonyme>;
```

Ainsi

```
typedef struct rationnel Rationnel;
```

définit le synonyme `Rationnel` au nom de type `struct rationnel`. Ce synonyme peut être utilisé partout à la place du nom standard

```
Rationnel x = {4,5};
```

permet de définir un nombre rationnel.

Il est très utile d'utiliser `typedef` en association avec une discipline personnelle de dénomination. Je vous propose la suivante:

1. le nom d'un type qui représente une entité commence par une majuscule. On nommera, par exemple, `Rationnel`, `Client`.
2. le nom d'un type qui représente un pointeur sur une entité est écrit en majuscule. On nommera ainsi `RATIONNEL` un pointeur sur un `Rationnel`.

On associera donc systématiquement l'opérateur `typedef` à la définition d'une structure

```
struct rationnel {
    int numerateur;
    int denominateur;
};

typedef struct rationnel Rationnel, *RATIONNEL;
```

ce qui permettra d'écrire ultérieurement

```
Rationnel r = {22,7};
RATIONNEL pr = &r;
```

9.5.6 Type d'une Fonction

Le type d'une fonction est défini par son *domaine* associé à son *codomaine*, on dira, par exemple, que la fonction sinus() est de type $R \rightarrow R$ ce qui signifie que son paramètre prend ses valeurs dans R et que son résultat appartient à R. La fonction racine carrée() aura le type $R \rightarrow R$

Le type d'une fonction définie en *langage C* est déclaré au moment de sa définition

```
int
factorielle(n)
```

[7] La lisibilité d'un programme caractérise la facilité avec laquelle on en comprend le fonctionnement en en lisant seulement le texte. Un mythe selon lequel il existe des langage lisibles et des langages non lisibles alimente la guerre de religion des langages, en fait, il existe des programmes clairement structurés et des programmes confus

> ...Ce qui se conçoit bien s'énonce clairement
> Et les mots pour le dire nous viennent aisément...

```
    int n;
{....}
```

définit la fonction `factorielle` comme étant de type $\mathbb{N}\rightarrow\mathbb{N}$. Cette écriture correspond au *langage C* traditionnel, la norme ANSI du *langage C* préconise l'écriture équivalente suivante

```
int
factorielle(int n)
{....}
```

La déclaration du type d'une fonction va nous permettre de préciser la définition des interfaces des types abstraits de données.

Ainsi l'interface du paquetage des **Rationnels** contenue dans le fichier rationnel.h s'écrirait

```
struct rationnel {
    int numerateur;
    int denominateur;
};

typedef struct rationnel Rationnel, *RATIONNEL;

RATIONNEL RationnelCreer (int,int);
int RationnelNumerateur (RATIONNEL);
int RationnelDenominateur (RATIONNEL);
```

tandis que le fichier d'implémentation `rationnel.c` contiendrait

```
#include "rationnel.h"

RATIONNEL
RationnelCreer(num,denom);
    int num;
    int denom;
{...}

int
RationnelNumerateur(unRat);
    RATIONNEL unRat;
{
    return(unRat->numerateur);
}

int
RationnelDenominateur(unRat);
    RATIONNEL unRat;
{
    return(unRat->denominateur);
}
```

Il peut arriver qu'un fonction ne rende pas de valeur (lorsqu'elle ne produit qu'un effet de bord par exemple), il peut arriver également qu'elle n'ait pas de paramètre, dans ce cas le type de son domaine ou de son codomaine est nommé `void` [8].

On déclarerait, par exemple

```
void RationnelAfficher(RATIONNEL);
```

[8] Ce `void` là signifie *jamais rien* !

```
int random(void);
```

9.5.7 Forçage du Type.

Le typage des noms a été introduit pour permettre au traducteur *C* de vérifier la cohérence des expressions *C* que nous avons écrites. La règle de cohérence de base est très simple, elle consiste simplement à considérer qu'une expression n'est cohérente que si toutes les variables impliquées dans l'application d'une fonction à ses argument respectent le type annoncé dans le patron de cette fonction.

Cette règle, comme toute règle générale, supporte deux exceptions:

1. le type fictif `void` ne prèjuge d'aucun type a priori, il faudra donc définir le type effectif utilisé le moment venu.
2. les opérateurs arithmétiques `+`, `-`, `*` et `/` peuvent être définis (avoir un sens) même lorsqu'ils s'applique à des variables de types différents (nombres exacts et nombres inexacts).

Ainsi, un type peut être forcé soit *à notre insu* dans les expressions arithmétiques soit *à notre instigation* lorsqu'il s'agit de définir finalement ce qu'on associe à une variable `void`.

L'expression `4+6.2` est ainsi automatiquement interprétée comme `4.0+6.2` et le nombre exact `4` à été forcé sous la forme inexacte `4.0`. Par contre, certaines fonctions dites génériques, rendent des valeurs qualifiées de `void`.

Il existe, par exemple, une fonction très importante dont le patron est

```
void * malloc(int);
```

Cette fonction rend un pointeur sur *on verra bien quoi plus tard*. Ainsi, lorsqu'on utilise cette fonction, il est nécessaire de préciser comment on va interpréter ce *on verra bien quoi plus tard* en forçant le type de ce que rend cette fonction.

Ecrivons, par exemple

```
RATIONNEL pr;
....
pr = (RATIONNEL)malloc(...);
```

L'opération notée `(RATIONNEL)` dénote le forçage du type *on verra bien quoi plus tard* dans le type `RATIONNEL`.

9.6 Les Environnements d'une Application C.

L'environnement d'une application représente la structure où toutes les données d'une application sont rangées. On distingue les *environnements permanents* dont la structure est fixe pendant toute la durée de l'exécution du programme de l'application et l' *environnement volatil* dont la structure va évoluer en permanence. De plus, selon le degrés d'accessibilité des données, un environnement pourra être qualifié de *local* ou de *global*.

Les environnements sont associés aux différentes abstractions de l'applications

Niveau d'abstraction	Environnement	
Application	global	permanent
Paquetage	local	permanent
Fonction & Bloc de code	privé	permanent
		volatil

Ces différents types d'environnements sont caractérisés par la *durée de vie* (extent) et le *domaine de visibilité* (scope) des variables qui s'y trouvent. Un environnement est qualifié de **permanent** lorsque les données qu'il contient sont *immortelles* et de **volatil** lorsque les données qu'il contient naissent et meurent éventuellement au cours d'une exécution de l'application.

Un environnement est qualifié de **local** lorsque l'accès aux données qui y sont situées est limité aux instructions *C* situées soit dans le paquetage soit dans la fonction propriétaire de cet environnement et de **global** lorsque toutes les instructions *C* de l'application ont accès aux données qu'il contient.

Il existe un *environnement manuel* dont la gestion est *systématiquement effectuée à travers des pointeurs* placés dans les autres environnements. Un des principaux agréments de l'utilisation du *langage C* réside dans la grande facilité qu'il offre pour la gestion d'un tel environnement.

9.6.1 Règles de Visibilité.

Les règles de visibilité associées aux différentes variables d'une application sont régies par la structure emboitée des abstractions qu'il est possible de construire en *langage C* (Cf. figure 41, page 212).

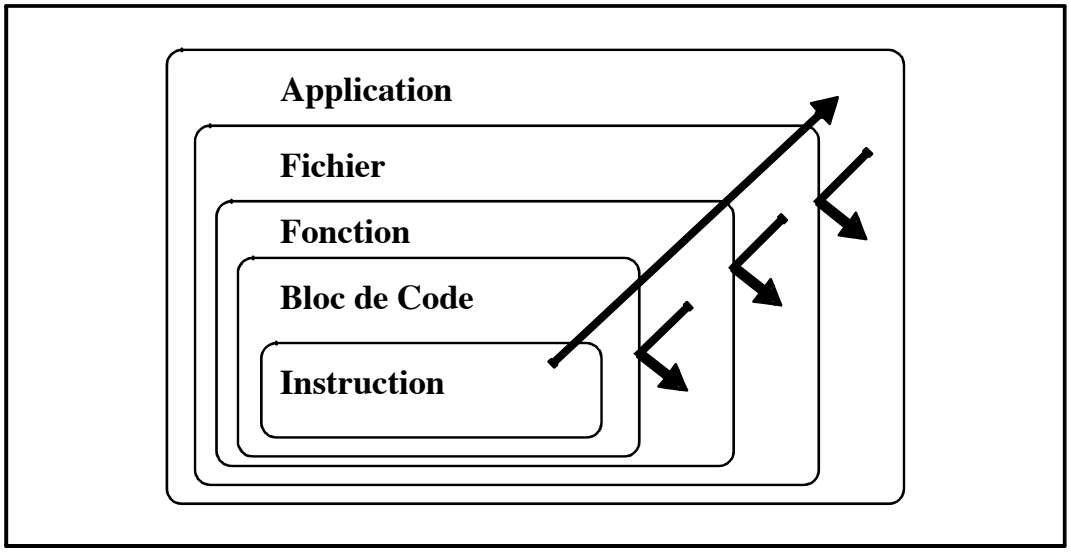


Fig. 41 : Structure emboitée des abstractions langage C.

Ainsi une abstraction a toujours accès aux variables de celles qui l'englobent tandis qu'elle n'a jamais accès aux variables de celles qu'elle contient.

9.6.2 Environnement permanent global d'une Application.

Une application *C* est constituée d'une ensemble de *paquetages* dont les instructions ont librement accès aux données contenues dans l' *environnement permanent global* .

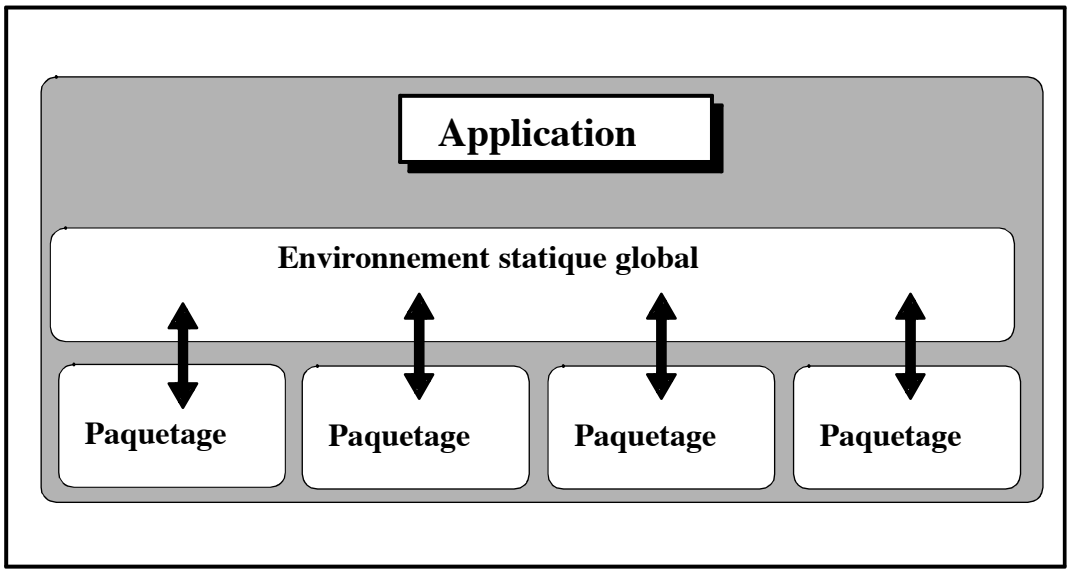


Fig. 42 : Paquetages et environnements statique global.

Les variables appartenant à cet environnement sont déclarées, à l'intérieur des paquetages de l'application qui en ont l'usage, en dehors de toute fonction et sont qualifiées de `extern`. En fait, peu de données justifient d'être placées dans cet environnement.

On y place, en général, un *compte-rendu d'erreur* de façon à le rendre accessible immédiatement à toutes les fonctions de l'application et les données d'un intérêt indiscutablement général. Par exemple, si on utilise un paquetage définissant un type de listes, on y placera la constante *liste vide*.

Considérons une application dont l'architecture comprend les fichiers `foo.c`, `bar.c` et `baz.c` et nous avons défini une variable utilisée comme compte-rendu d'erreur `errorReport`. Cette variable est, par exemple, définie dans le fichier foo.c

```
int errorReport = 0;

main()
{....)
```

Elle est ensuite déclarée dans le fichier `bar.c`.

```
extern int errorReport;

BAR
CreerBar(...)
    ...
{....)
```

puis dans le fichier `baz.c`:

```
extern int errorReport;

BAZ
```

```
CreerBaz(...)
    ...
{....)
```

9.6.3 Environnement statique local d'un Paquetage.

Chaque paquetage est constitué d'un ensemble de *fonctions* dont les instructions ont librement accès aux données contenues dans l'*environnement permanent local* du paquetage. Les instructions appartenant aux autres paquetages n'ont pas accès à ces données.

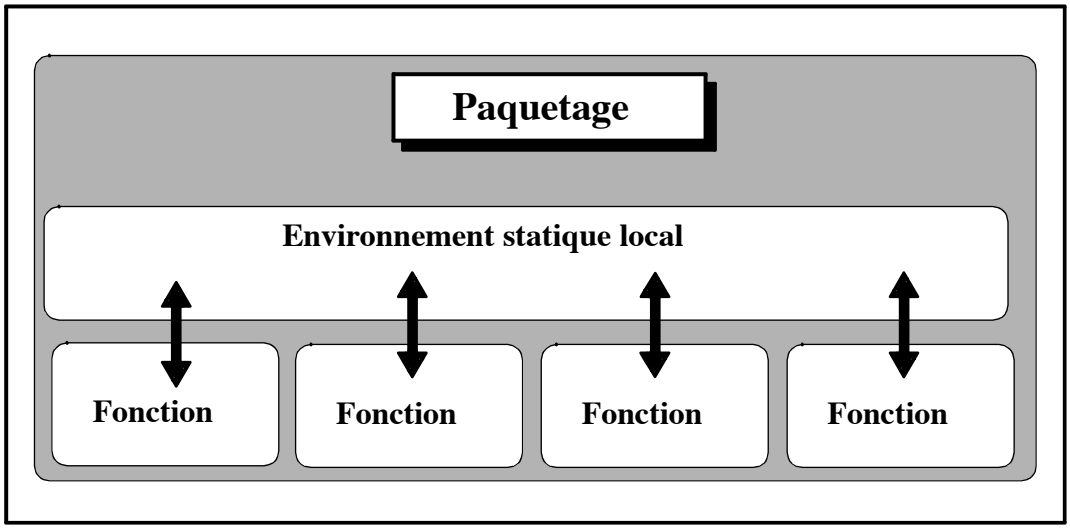


Fig. 43 : Fonctions et environnement statique local.

Les variables appartenant à cet environnement sont déclarées, à l'intérieur des paquetages [9] de l'application qui en ont l'usage, en dehors de toute fonction et sont qualifiées de `static`.

Cet environnement est en général utilisé pour y placer les constantes utiles à l'implémentation du type de données défini par le paquetage.

Le mot `static` dénote ici une restriction de la visibilité du nom ainsi qualifié, on peut également l'utiliser pour empêcher la visibilité de fonctions considérées comme privées.

9.6.4 Environnements privés d'un Bloc de Code.

Le bloc de code, et par extension la fonction, constitue une barrière d'abstraction dans les applications *C* qui limite la visibilité des variables qui y sont définies aux seules instructions qui y sont contenues.

Environnement permanent privé

Chaque fonction, ou plus précisément chaque bloc de code, peut être associé à un *environnement permanent privé*.

Cet environnement contient des données qui ne sont accessibles qu'aux instructions du bloc de code ou de la fonction considéré. Ces données sont qualifiées de `static`. Comme ces

[9] Dans le fichier d'implémentation correspondant `xxx.c`.

données sont immortelles, on retrouve leur valeur chaque fois qu'on entre dans le bloc de code correspondant.

Pour illustrer l'utilisation d'un tel environnement, imaginons une fonction qui compte le nombre de fois où elle a été invoquée

```
int
decompte()
{
    static int comptage = 0;

    comptage = comptage+1;
    return(comptage);
}
```

La variable `comptage` est initialement définie égale à `0`. Comme elle est immortelle, elle est incrémentée à chaque appel de `decompte()` et conserve sa valeur d'un appel à l'autre.

Une telle fonction est utile dans la phase de mise au point d'une application en permettant de repérer les fonctions qui, très utilisées, peuvent justifier une optimisation.

Environnement privé volatil dit «automatique»

Une application possède systématiquement un *environnement volatil privé*. Cet environnement *est étendu* au moment de l'entrée dans tout bloc de code à l'aide des variables qui y sont définies et au moment de tout appel de fonction à l'aide de ses paramètres. Cette *extension est privée* au bloc considéré. La description d'un tel environnement a été vue à la fin du chapitre *Abstraire par les Données*.

Ainsi, les données déclarées dans un bloc de code *naissent* lorsqu'on y entre et *meurent* quand on en sort.

On utilise, en général, l'environnement automatique pour y définir des variables utilisées pour dénoter un résultat intermédiaire dont la durée de vie n'a pas à excéder celle du calcul effectué. Ces variables sont redéfinies à chaque invocation du bloc de code et disparaissent à la fin de l'exécution du bloc. Nous pouvons, par exemple, étendre le paquetage des nombres rationnels en y ajoutant une fonction permettant d'additionner deux nombres rationnels.

```
RATIONNEL
RationnelAjouter(unRat_1,unRat_2)
    RATIONNEL unRat_1;
    RATIONNEL unRat_2;
{
    int num1 = RationnelNumerateur(unRat_1);
    int num2 = RationnelNumerateur(unRat_2);
    int dnm1 = RationnelDenominateur(unRat_1);
    int dnm2 = RationnelDenominateur(unRat_2);
    return(RationnelCreer(num1*dnm2+num2*dnm1,dnm1*dnm2));
}
```

les variables `num1`, `num2`, `dnm1` et `dnm2` sont privées à la fonction `RationnelAjouter()`, elles ne sont définies qu'au moment où on entre dans la fonction et meurent quand on en sort.

Une utilisation intéressante des blocs de code est le confinement d'une correction effectuée lors de la maintenance d'une fonction de l'application. Imaginons, par exemple que la fonction suivante provoque une erreur

```
void
```

```
RectangleEtendre(...)
    ...
{
    int x,y;
    ...
    ...          une erreur est détectée à ce niveau
    ...
}
```

Cette erreur pourrait être corrigée en permutant les deux variables `x` et `y`. Malheureusement si le remède est évident, cette fonction, déjà ancienne et qui a subi de nombreuses corrections, est devenue quasiment incompréhensible et il est pratiquement impossible de remonter l'histoire de `x` et `y`.

La seule solution (à défaut d'une re-écriture de cette fonction) consiste à intercaler un bloc de code pour effectuer la permutation nécessaire

```
void
RectangleEtendre(...)
    ...
{
    int x,y;
    ...
    ...
    { /* correction de l'erreur : debut */
        int z = x;
        x = y;
        y = z;
    } /* correction de l'erreur : fin */
    ...
}
```

La variable `z` ainsi introduite pour effectuer la permutation ne risque pas d'interférer avec une autre variable `z` de la fonction.

En règle générale, il est de bonne politique de ne définir les variables temporaires qu'au niveau du bloc de code où elles sont effectivement utilisées.

9.6.5 Environnement manuel dit «le tas».

L'environnement dynamique manuel correspond à une structure nommée *tas* (heap) dans lequel il est possible d'*allouer* puis de *désallouer* des données. Ce n'est pas une possibilité offerte directement par le *langage C* lui-même, mais par un paquetage spécial dont l'interface résumée est la suivante:

```
void * malloc(int);
void free(void *);
```

`malloc()` est la fonction permettant l'allocation d'une donnée et `free()` celle permettant la destruction de cette donnée et la libération de la place occupée.

D'autre langages impératifs (Pascal, Ada...), incorporent dans le langage lui-même ces deux opérateurs.

Extension de l'Environnement manuel

La fonction `malloc(...)` rend un pointeur sur *on verra bien plus tard* donnant accès à un emplacement alloué dans le tas pouvant contenir l'élément dont la taille est passée en argument. C'est l'opérateur *C* `sizeof(...)` qui nous donne cette taille.

Par exemple

```
sizeof(int)
sizeof(float)
sizeof(Rationnel)
```

rendent respectivement la taille [10] d'un `int`, d'un `float` et d'un `Rationnel`.

Nous pouvons, à présent, définir le constructeur de **Rationnels** et terminer la définition de leur paquetage.

```
RATIONNEL
RationnelCreer(num,denom)
    int num;
    int denom;
{
    RATIONNEL unRat = (RATIONNEL)malloc(sizeof(Rationnel));
    unRat->numerateur = num;
    unRat->denominateur = denom;
    return(unRat);
}
```

Si les pointeurs permettant d'accéder aux données ainsi allouées sont placés dans les environnements permanents ou l'environnement automatique, les données elles-mêmes sont dans le tas.

Libération dans l'Environnement manuel

Une entité qui a été allouée sur le tas peut être détruite à l'aide de la fonction `free(...)` lorsqu'on est sûr qu'elle n'est plus utilisée.

Pour cela, dans le cas du paquetage des **Rationnels**, il est nécessaire de définir un *destructeur*.

```
void
RationnelDetruire(unRat)
    RATIONNEL unRat;
{
    free(unRat);
}
```

Libérer ou ne pas libérer ?

Il n'est pas toujours facile de savoir si on peut détruire un objet ou non, cette question peut même devenir *indécidable*. Pour illustrer le problème, examinons l'application **Serrure à Code** du chapitre *Objets & Programmation Orientée Objets*.

Le mécanisme de transmission de messages utilisé dans le dialogue entre les différents objets de l'application nécessite la présence d'un objet codeur commun en tant qu'accointance à toutes les touches et à la porte.

Imaginons que dans un certain cadre, il soit nécessaire de créer puis de détruire des touches tout au long d'une exécution de l'application. Il est clair que la destruction d'une touche ne doit pas entraîner, a priori, la destruction du codeur. Mais comment savoir si la touche à détruire n'est pas la dernière à accéder au codeur (la porte et toutes les autres touches ont déjà été détruites) auquel cas, il **faut** détruire le codeur?

[10] La taille d'un `int`, d'un `char` ou d'un `float` dépendant de la machine sur laquelle on travaille, la seule façon d'écrire un programme portable d'une machine à l'autre est d'utiliser l'opérateur `sizeof(...)`.

Un remède possible (mais pas systématique) consiste à ne pas partager d'objets entre différents propriétaires. Dans ce cas, un objet est systématiquement dupliqué avant que d'être affecté à un attribut. Il peut donc être nécessaire de définir un *duplicateur* de **Rationnels**.

```
RATIONNEL
RationnelDupliquer(unRat)
    RATIONNEL unRat;
{
    RationnelCreer(unRat->num,unRat->denom);
}
```

9.7 Paquetage de la Paire.

Une des structure de données les plus importantes s'est révélée être la **Liste**. La liste a été définie à partir d'un autre type de données, la **Paire**. Nous allons transposer en *C* d'abord le paquetage de la **Paire** puis celui de la **Liste**. Nous allons ainsi constater que cela fait, l'utilisation de ces paquetages va nous permettre de programmer en *C* dans un style très proche de celui de la programmation *Scheme*.

9.7.1 Spécifications détaillées de la Paire.

La **Paire** peut être définie à partir de son constructeur, de ses sélecteurs et de ses modificateurs.

```
(define cons
  (lambda (car cdr)
    (let ((set-car (lambda (nv) (set! car nv)))
          (set-cdr (lambda (nv)(set! cdr nv))))
      (lambda (msg)
        (cond  ((eq? msg 'car)car)
               ((eq? msg 'cdr)cdr)
               ((eq? msg 'set-car) set-car)
               ((eq? msg 'set-cdr) set-cdr))))))

(define car (lambda (p) (p 'car))
(define cdr (lambda (p) (p 'cdr))
(define set-car! (lambda (p v) ((p 'set-car) v)))
(define set-cdr!(lambda (p v) ((p 'set-cdr) v)))
```

La technique de transposition des messages *Scheme* en fonctions *C* nous étant devenue familière, nous ne la détaillerons pas.

9.7.2 Interface de la Paire.

La structure de la paire (déduite directement des paramètres du constructeur *Scheme*) est contenue dans le fichier de définition d'interface. Il est clair qu'il est particulièrement confortable de définir les fonctions correspondant à celles définies en *Scheme* en leur donnant les mêmes noms [11].

[11] Nous serons obligés de sacrifier les caractères **?** **!** et **:** que nous avions utilisés pour représenter respectivement les prédicats, les modificateurs et les constructeurs sur l'autel de la syntaxe du *langage C*.

La paire étant très souvent utilisée pour construire des listes il est nécessaire de compléter la paire de la *paire vide* et du *prédicat* permettant de savoir si une paire est la paire vide. La technique la plus simple pour définir une constante, ici la paire vide que nous nommerons `nil`, est de se souvenir qu'il n'y a pas de différence entre donnée et procédure et donc de définir une procédure jouant le rôle de cette constante.

L'interface du paquetage de paire est contenu dans le fichier `paire.h`.

```
struct paire {
    void *car;
    void *cdr;
};
typedef struct paire Paire, *PAIRE;

PAIRE cons(void *,void *);
PAIREnil(void);
int null(PAIRE);
void * car(PAIRE);
void * cdr(PAIRE);
void set_car(PAIRE,void *);
void set_cdr(PAIRE,void *);
```

9.7.3 Implémentation de la Paire.

L'implémentation de la paire sera possible dès que nous aurons choisi une représentation pour la paire vide. Le plus simple est d'associer la constante `nil` à une paire pathologique, la plus pathologique de toutes est sans conteste celle dont la tête est elle-même.

> On tombe, ici, sur un problème délicat qu'on rencontre, dans presque tous les langages de programmation, dès lors qu'on veut introduire les constantes associées à un nouveau type de données. En effet, comment être sûr que ces constantes ne seront pas recréées, par hasard, en manipulant d'autres types de données?
>
> On essaye donc d'imaginer des configurations qu'il est pratiquement impossible de créer au hasard. Malheureusement cette méthodes n'est que **probablement sûre** et son utilisation produit (très rarement heureusement) des bugs non reproductibles sur lesquels on pose, en général, un voile pudique.

L'implémentation de ce paquetage est classique et n'attire pas d'autre commentaire. Elle est contenue dans le fichier `paire.c`

```
#include "paire.h"

PAIRE
cons(car,cdr)
    void * car;
    void * cdr;
{
    PAIRE p= (PAIRE)malloc(sizeof(Paire));
    p->car = car;
    p->cdr = cdr;
    return(p);
}

PAIRE
nil()
{
```

```
    PAIRE p = (PAIRE)malloc(sizeof(Paire));

    p->car = p;
    return(p);
}

int
null(p)
    PAIRE p;
{
    return (p == car(p));
}

void *
car(p)
    PAIRE p;
{
    return(p->car);
}

void
cdr(p)
    PAIRE p;
{
    if null(p) {
        return(p);
    } else {
        return(p->cdr);
    }
}

void
set_car(p,x)
    PAIRE p;
    void *x;
{
    p->car = x;
}

void *
set_cdr(p,x)
    PAIRE p;
    void *x;
{
    if !null(p) {p->cdr = x;}
}
```

9.8 Paquetage de la Liste.

Le paquetage associé à la liste est un peu nouveau pour nous en ce sens qu'il ne correspond pas à une structure particulière de données mais simplement à la définition de fonctions de manipulation des paires et des listes.

9.8.1 Spécifications détaillées de la Liste.

Nous nous contenterons de définir les deux fonctions `length` et `append` à titre d'exemple. Il serait très facile d'en rajouter autant que nécessaire. On se souvient que les spécifications de ces deux fonctions sont

```
(define length
  (lambda (ll)
    (if (null? ll)
        0
        (+ 1 (length (cdr ll))))))

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

L'interprète *Scheme* «sait» afficher les listes dans la mesure où elles ne contiennent que des objets qu'il sait afficher. On pourrait cependant spécifier une procédure d'affichage relativement générale

```
(define display-liste
  (lambda (ll)
    (cond ((null? ll) (newline))
          (else (display (car ll))
                        (display " ")
                        (display-liste (cdr ll))))))
```

9.8.2 Interface de la Liste.

Nous verrons un peu plus tard pourquoi il n'est pas possible de définir une fonction d'affichage réellement générique, nous nous contenterons donc, à titre d'exemple, de l'affichage d'une liste de chaînes de caractères.

L'interface du paquetage de liste est contenu dans le fichier `liste.h`.

```
#include "paire.h"

typedef PAIRE LISTE;
void DisplayListeTexte(LISTE);
int length(LISTE);
LISTE append(LISTE,LISTE);
```

9.8.3 Implémentation de la Liste.

L'implémentation de la liste est contenue dans le fichier `liste.c`.

```
#include "liste.h"

int
length(lst)
    LISTE lst;
{
    if (null(lst)) {
        return(0);
    } else {
```

```
        return(1 + length(cdr(lst)));
    }
}

LISTE
append(l1,l2)
    LISTE l1,l2;
{
    if (null(l1)) {
        return(l2);
    } else {
        return(cons(car(l1),append(cdr(l1),l2)));
    }
}
```

Le *langage C* ne sait pas, naturellement, afficher les listes, il faut donc définir une fonction d'affichage pour une liste. Mais, alors que les fonctions précédentes sont génériques, cette fonction-ci ne peut pas l'être. En effet, tant qu'on se contente de manipuler «à l'aveuglette» des éléments contenus dans une liste, cette manipulation peut être générique, par contre, dès qu'il s'agit d'effectuer des opérations sur ces éléments, il faut que l'opération elle-même soit générique.

Par exemple, on ne peut faire la somme des éléments correspondants de deux listes que si on peut être sûrs que ce sont des nombres. Il devient alors clair qu'on ne peut pas faire des listes de n'importe quoi n'importe comment.

Voici, à titre d'exemple, le fonction d'affichage d'une *liste homogène de chaînes de caractères*.

```
void
DisplayListeTexte(lst)
    LISTE lst;
{
    if (null(lst)) {
        printf("\n");
    } else {
        printf(" ");
        printf(car(lst));
        DisplayListe(cdr(lst));
    }
}
```

Je pense que la similitude avec la programmation *Scheme* commence à vous sauter aux yeux.

9.9 Le Paquetage de la Pile.

La paquetage de la **Pile** va se déduire directement des spécifications détaillées que nous avons définies au chapitre consacré aux *Structures mutables*.

9.9.1 Spécifications détaillées de la Pile.

```
(define pile-vide
  (lambda () (reference-a nil))
```

```
(define push
  (lambda (x &p)
    (ref! &p (cons x (&p))) x))

(define pop
  (lambda (&p)
    (let ((sdp (car (&p))))
      (ref! &p (cdr (&p))) sdp))))
```

9.9.2 Interface de la Pile.

L'interface de la **Pile** est contenue dans le fichier `pile.h`.

```
#include "liste.h"

struct pile {
    LISTE lp;
    };
typedef struct pile Pile *PILE;

PILE pile_vide(void);
void *push(void *,PILE);
void * pop(PILE);
```

9.9.3 Implémentation de la Pile

L'implémentation de la **Pile** est contenue dans le fichier `pile.c`.

```
#include "pile.h"

PILE
pile_vide();
{
    PILE p= (PILE)malloc(sizeof(Pile));

    p->lp = nil();
    return(p);
}

void
push(x,p)
    PILE p;
    void * x;
{
    p->lp = cons(x,p->lp);
    return(x);
}

void *
pop(p)
    PILE p;
{
    void * sdp = car(p->lp);

    p->lp = cdr(p->lp);
    return(sdp);
```

```
}
```

9.10 Les Procédures C.

Nous avons déja appris à définir puis à invoquer une procédure. Par exemple, nous avons défini la procédure factorielle.

```
int
factorielle(n)
    int n;
{
    if (n == 1) {
        return(1);
    } else {
        return(n*factorielle(n-1));
    }
}
```

que nous pouvons invoquer dans une instruction de la forme

```
j = factorielle(6);
```

Nous avons également appris à déclarer une procédure pour construire les interfaces des paquetages:

```
int factorielle(int);
```

Approfondissons, à présent, les propriétés des procédures *C* et en particulier demandons-nous en quoi les procédures *C* ne sont pas des citoyens de première classe comme les procédures *Scheme*.

Dans le cadre des langages de programmation, les citoyens de première classe peuvent être:

1. **définis** - ils constituent des valeurs qu'il est possible de nommer afin d'en faire une abstraction.
2. **affectés** - ils peuvent être associés à un nom afin d'être manipulés.
3. **créés** - on peut leur définir un constructeur.
4. **subir une application** - ils peuvent constituer un argument de procédure.
5. **constituer le résultat d'une évaluation**.

Dans tous les langages de programmation, les données sont des citoyens de première classe, par contre, il est assez rare que les procédures le soient [12]. Comparons, sur ce point, les prérogatives des données et des procédures en *Scheme* et en *C* (Cf. tableau 5, page 225).

On remarque que *Scheme* ne sait pas créer de données. Cela ne doit pas nous surprendre puisqu'en fait *Scheme* considère que toutes ses entités sont des fonctions. Nous avons d'ailleurs défini les données à partir de leur constructeur.

Manipuler une procédure comme une donnée nécessite, en fait, deux choses:

1. il faut pouvoir les affecter à une variable. Cela permet de les passer en argument d'une autre procédure ou de les récupérer comme résultat d'une application.
2. il faut disposer d'un opérateur de création de procédure. Cela permet de concevoir des procédures qui construisent des procédures. Cela permet, en plus, de dissocier la

[12] *Scheme* est exceptionnel de ce point de vue là.

	Données langage C	Données Scheme
Définition	`int foo=5;`	`(define foo 5)`
Affectation	`foo=8;`	`(set! foo 8)`
Création	`unRat=(RATIONNEL)malloc(...);`	-
Argument	`factorielle(foo);`	`(factorielle foo)`
Résultat	`return(1);`	`(lambda (...)... 1)`

Tab. 5 : Prérogatives des données.

procédure du nom qu'on lui associe et ainsi de lui donner une existence en tant que telle.

Le *langage C* **ne dispose pas** d'un opérateur pour créer une procédure. On ne peut donc pas définir de procédure qui construirait une procédure. Cela revient à considérer que le nom d'une procédure qui a été définie est une **constante**.

Si on veut pouvoir affecter une procédure à une variable, il est nécessaire de définir un type *procédure*. En fait, le *langage C* permet de définir un *pointeur sur une procédure*. La déclaration d'une tel pointeur est complexe, aussi nous n'en verrons que les formes les plus utiles.

La forme la plus simple est la suivante

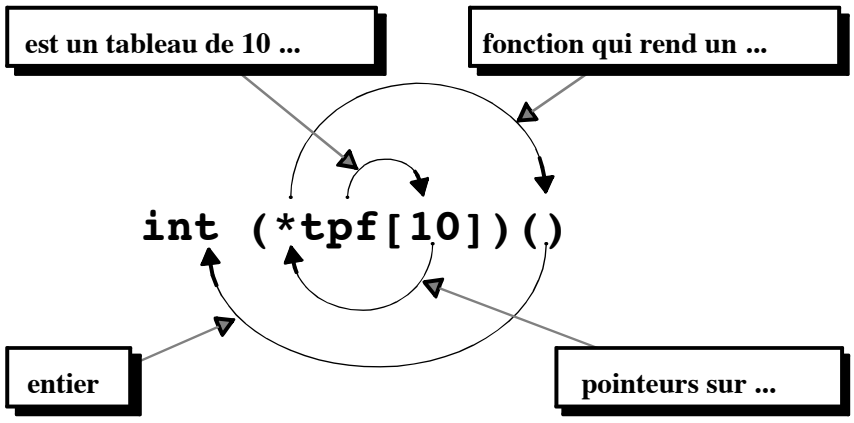
```
int (*pf)();
```

qui définit la variable `pf` comme étant un pointeur sur une fonction qui rend un entier. Une autre forme est également utile

```
int (*tpf[10])();
```

qui définit `tpf` comme étant un tableau de 10 pointeurs sur des fonctions qui rendent des entiers.

La lecture de ces déclarations démarre au nom défini puis avance un coup à droite puis un coup à gauche. Les signes `()` dénotent *fonction*, les signes `[]` dénotent *tableau* et le signe `*` dénote *pointeur*.



La manipulation des procédures en *C* devient facile et est souvent utilisée (Cf. tableau 6, page 226).

	Procédures langage C	**Procédures Scheme**
Définition	`int factorielle(n) int n; {...}`	`(define square (lambda (x) (* x x))`
Affectation	`int (*pf)(); pf = factorielle; ... pf(6);`	`(set! foo square)`
Création	-	`(lambda (x) (* x x))`

Tab. 6 : Prérogatives des fonctions.

Le mécanisme d'évaluation des procédures *C* est identique à celui des procédures *Scheme*. Les arguments de la procédure sont évalués puis celle-ci est appliquées au résultat de cette évaluation. Ce mécanisme que nous avions appelé *évaluation en ordre applicatif* est appelé *passage des arguments par valeur* dans le contexte des langages de programmation impératifs tels *C*, Pascal et Ada.

A titre d'exemple, reprenons le petit exercice de virtuosité qui consistait à redéfinir les booléens uniquement à l'aide de fonctions.

Les constantes `vrai` et `faux` ont été définies à cette occasion à partir des deux fonctions

```
(define vrai (lambda (a b) a)
(define faux (lambda (a b) b)
```

associées à la fonction d'affichage correspondante.

```
(define booleen-afficher (lambda (b) (display (b "V" "F")))
```

Les deux premières fonctions se traduisent immédiatement en *C*.

```
char *
vrai(a,b)
    char *a;
    char *b;
{
    return(a);
}
```

et

```
char *
faux(a,b)
    char *a;
    char *b;
{
    return(b);
}
```

A partir de là il est agréable de définir un synonyme pour le type *pointeur sur une fonction qui rend un pointeur sur des caractères* et qui nous sert de booléen.

```
typedef char * (*Booleen)();
```

Cela nous permet de définir la fonction d'affichage des booléens de manière plus confortable.

```
void
BooleenAfficher(a)
    Booleen a;
{
    printf(a("V","F"));
}
```

Les fonctions `and` et or avaient été définies par

```
(define and (lambda (a b) (a b faux))
(define or (lambda (a b)(a vrai b))
```

ce qui se traduit immédiatement en *C* par

```
Booleen
and(a,b)
    Booleen a;
    Booleen b;
{
    return(a(b,faux));
}

Booleen
or(a,b)
    Booleen a;
    Booleen b;
{
    return(a(vrai,b));
}
```

Un exemple d'utilisation de ces drôles de booléens pourrait être

```
main()
{
    BooleenAfficher(vrai);
    printf("\n");
    BooleenAfficher(faux);
    printf("\n");
    BooleenAfficher(and(faux,faux));
    BooleenAfficher(and(faux,vrai));
    BooleenAfficher(and(vrai,faux));
    BooleenAfficher(and(vrai,vrai));
}
```

9.11 Transmission de Messages en langage C.

La technique de transposition utilisée jusqu'à présent introduit un grand nombre de procédures dans le domaine public de l'application. Cela pose assez rapidement deux problèmes:

1. Il devient vite difficile de trouver des noms de procédure lorsque leur nombre devient élevé (de quelques centaines à quelques milliers).
2. Il est difficile d'utiliser des paquetages d'origine diverses à cause des nombreuses collisions de noms qui ne vont pas manquer de se produire et qui sont insolubles.

Le remède à cette situation est d'utiliser la transmission de messages qui resoud les deux problèmes précédents d'une part du fait que toutes les procédures d'un paquetage deviennent privées et d'autre part du fait que les messages qui leur sont associés permettent le polymorphisme.

Donnons, à titre d'exemple, une implémentation des **Rationnels** utilisant la transmision de messages comme technique d'invocation du constructeur et des sélecteurs associés.

La structure des nombre rationnels n'a pas besoin d'être modifiée, elle est contenue dans le fichier `rationnel.h`

```
struct rationnel {
    int num;
    int denom;
    };
typedef struct rationnel Rationnel, *RATIONNEL;
```

L'interface du paquetage est, elle, complètement différente. On se souvient que dans le cas des langages objets, on avait défini une classe associée à chacun des types d'objet de l'application. Cette classe était une structure de données qui contenait uniquement les méthodes associées à ses futures instances. Nous allons donc définir une structure contenant, sous la forme de pointeurs sur procédure, toutes les primitives associées au type de données à définir.

Cette interface est contenue dans le fichier `rationnel.h`.

```
#include "rationnel.h"

struct rationnel {
    int num;
    int denom;
    };
typedef struct rationnel Rationnel, *RATIONNEL;

struct c_rationnel {
    RATIONNEL (*creer)();
    int (*numerateur)();
    int (*denominateur)();
};

typedef struct c_rationnel Ratio;

extern Ratio RATIO;
```

Nous reviendrons sur le rôle joué sur la déclaration finale du fichier d'interface. Le fichier d'implémentation définit toutes les procédures qui seront à associer aux messages.

Ces procédures seront naturellement privées. Cette implémentation est contenue dans le fichier `rationnel.c`.

```
#include "rationnel.h"

static RATIONNEL
creer(num,denom)
    int num;
    int denom;
{
    RATIONNEL unRat = (RATIONNEL)malloc(sizeof(Rationnel));
    unRat->num = num;
    unRat->denom = denom;
```

```
    return(unRat);
}

static int
numerateur(unRat)
    RATIONNEL unRat;
{
    return(unRat->num);
}

static int
denominateur(unRat)
    RATIONNEL unRat;
{
    return(unRat->denom);
}

Ratio RATIO = {creer,numerateur,denominateur};
```

La dernière ligne du fichier d'implémentation introduit la définition de la donnée globale `RATIO` qui joue le rôle d'un dictionnaire des méthodes. C'est parce que cette donnée globale doit être à la disposition de tous les clients du type **Rationnels** que sa déclaration en tant qu'`extern` à été placée dans le fichier d'interface.

Le fichier `t-rationnel.c` illustre l'utilisation de ces pseudo-messages pour manipuler des instances de **Rationnels**.

```
#include "rationnel.h"

main()
{
    RATIONNEL unRat = RATIO.creer(4,5);

    AfficherEntier(RATIO.numerateur(unRat));
    AfficherEntier(RATIO.denominateur(unRat));
}
```

9.12 Récursion terminale & Processus itératif.

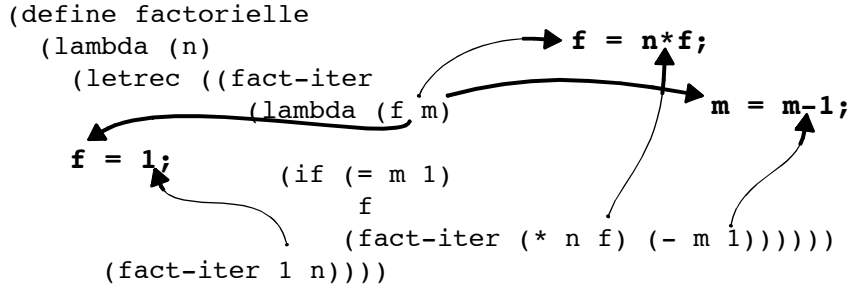
Il nous reste un dernier problème à traiter, c'est celui de la description des processus itératifs que *Scheme* sait associer à une définition récursive terminale mais que le *langageC* ne sait ni reconnaître ni exploiter.

9.12.1 Traduction de l'Invariant.

Nous avons vu qu'une définition récursive terminale pouvait (et devait) toujours être associée à un invariant qui garantit la validité du processus itératif engendré. C'est cet invariant qui va nous servir d'intermédiaire entre la forme *Scheme* et la forme *C* de l'itération.

On se souvient que l'invariant est un prédicat qui est vrai au début de l'itération, qui le reste tout au long de l'itération et qui signale, en devenant faux, que l'itération doit cesser.

Reprenons l'exemple simple de la fonction `factorielle` définie sous forme itérative.



On peut vérifier que l'expression

```
(fact-iter 1 n)
```

a pour but de rendre l'invariant asocié à cette définition *vrai*, on l'appellera l'*initialisation* de l'itération.

L'expression

```
(if (= m 1) ...)
```

interrompt l'itération dès que cet invariant devient *faux*, on l'appellera la *sortie* de l'itération. Quant à la procédure `fact-iter` dont le seul but est de faire tourner l'itération, nous l'appellerons le *corps* de l'itération.

La structure de cette itération est alors décrite figure 44, page 230. La traduction *C* de cette

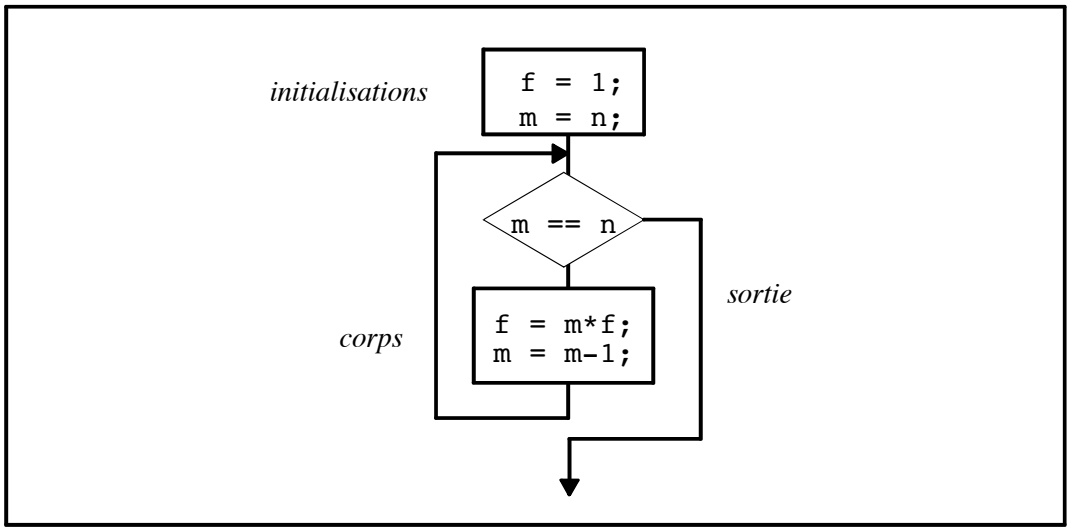


Fig. 44 : Structure d'une itération.

itération peut s'écrire

```
    ....
    f = 1;
    m = n;
loop:
    if (m == 1) goto sortie;
    f = m*f;
    m = m-1;
    goto loop;
```

```
sortie:
    ....
```

`loop:` et `sortie:` s'appellent des labels et l'instruction `goto` permet de rejoindre l'instruction associée au label cible.

Cette instruction `goto` a une très mauvaise réputation car malgré son caractère indispensable, mal utilisée elle transforme les programmes en fouillis indescriptibles. Aussi, dans les années 70, une violente campagne anti `goto` fut déclenchée par les chercheurs en informatique qui définirent des structures de remplacement plus sûres bien que tout aussi efficaces. A l'heure actuelle **il est toujours très mal vu** d'utiliser cette instruction `goto` même lorsqu'elle est irremplaçable [13]. Ainsi donc, l'exemple ci-dessus n'est pas à suivre sans discernement.

9.12.2 Structures de Boucle.

Comme il est fréquent qu'il existe plusieurs conditions rendant l'invariant d'itération *faux*, la structure de boucle la plus générale est la suivante

```
loop {
    ....
    exitif <prédicat 1>;
    ....
    exitif <prédicat 2>;
    ....
    exitif <prédicat 3>;
    ...
    }
```

Malheureusement, cette structure n'existe pas en *C* (nous verrons cependant comment la recréer). Le *langageC* fournit deux structures correspondant aux deux cas particuliers suivants

```
while <predicat> {
    ...
}
```

lorsqu'il n'existe qu'**une seule condition** rendant l'invariant *faux* et que cette condition est à évaluer au début du corps de l'itération, et

```
do {
    ...
} while <predicat>;
```

lorsqu'il n'existe qu'**une seule condition** rendant l'invariant *faux* et que cette condition est à évaluer à la fin du corps de l'itération.

La première forme dénote que l'itération est effectuée tant que le prédicat est *vrai*, tandis que la deuxième dénote que l'itération est effectuée jusqu'à ce que le prédicat devienne *faux*.

Cette restriction rend très malaisée l'utilisation de ces structures dans le cas général. On utilise alors souvent la forme suivante

```
while(1) {
    ....
    if <prédicat 1> break;
```

[13] L'implémentation d'une machine à états finis (CF. cours de *Mathématique pour l'informatique*) est le seul cas où l'utilisation du `goto` s'impose à la fois pour son efficacité et pour sa lisibilité.

```
    ....
    if <prédicat 2> break;
    ....
}
```

Dans ces conditions, la fonction `factorielle` pourrait être définie en *C* par

```
int
factorielle(n)
    int n;
{
    int f = 1;
    while (n != 1) {
        f = n*f;
        n = n-1;
    };
    return(f);
}
```

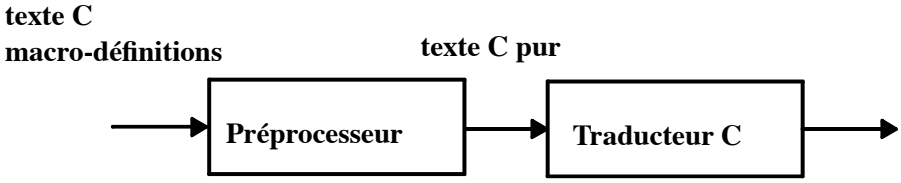
le prédicat `(n != 1)` signifiant `n` différent de `1`, ou par la forme équivalente:

```
int
factorielle(n)
    int n;
{
    int f = 1;
    while(1) {
        if (n <= 1) break;
        f = n*f;
        n = n-1;
    };
    return(f);
}
```

9.13 Le Pré-Processeur C.

Une particularité du langage *C* est d'être systématiquement associé à un *préprocesseur*. Un préprocesseur est une sorte d'éditeur de texte paramétrable permettant de définir des formes textuelles équivalentes entre elles. Ces formes textuelles s'appellent des *macro-définitions*.

Ce préprocesseur est systématiquement invoqué sur le texte des programmes *C* avant qu'ils soient soumis au traducteur *C* lui-même.



Les commandes au préprocesseur sont repérées par le fait que la ligne qui en contient une commence par le signe `#`.

Le préproceseur fonctionnement **uniquement par remplacement textuel**, il n'interprète jamais le texte qu'on lui soumet et les macro-définitions ne sont que des règles de remplacement textuel.

Le préprocesseur peut donc être utilisé sur n'importe quel texte.

9.13.1 Intégration de Fichiers.

Nous avons déjà rencontré la commande d'intégration de fichiers

```
#include «<nom de fichier>»
```

Cette commande insère le fichier indiqué dans le fichier qui la contient. Elle est très utile pour insérer des portions communes de texte *C* pour garantir leur identité totale. C'est pour cela que nous l'utilisons pour être sûrs que l'interface d'un paquetage est bien perçu de façon identique par tous ses clients.

9.13.2 Définition de Constantes.

Les expressions C contiennent parfois des constantes d'intérêt général. Il est alors fondamental de garantir que cette constante est bien vue partout de la même façon. Une commande du préprocesseur permet de définir de telles constantes

```
#define PI 3.1415926
#define TVA 0.186
```

Nous verrons qu'il peut être utile de définir simplement un symbole sans lui associer de valeur

```
#define DEJA_FAIT
```

Lors de la conception d'une application, on définit alors fréquemment un fichier des constantes. Ces constantes peuvent souvent être interprétées comme des symboles un peu analogues aux symboles *Scheme*.

9.13.3 Définition de Formes spéciales.

Une forme spéciale est une définition paramètrée. Bien que sa forme soit très analogue à celle d'une fonction C, elle est interprétée de manière radicalement différente.

Considérons, par exemple, les macros définition suivantes

```
#define loop while(1)
#define exitif(p) if (p) break
```

et la fraction de programme *C* suivante

```
...
loop {
    ...
    ...
    exitif(n<0);
    ...
    exitif(n==0);
    ...
}
...
```

sera remplacé, par le préprocesseur, par le fragment suivant

```
...
while(1){
    ...
```

```
        ...
        if (n<0) break;
        ...
        if (n==0) break;
        ...
    }
    ...
```

Les macro-définitions sont alors très souvent utilisées pour gommer l'aspect ingrat de certaines écritures *C*. Cette souplesse unique dans le modelage de l'aspect extérieur des programmes, est un des arguments majeur du choix très fréquent du couple préprocesseur,traducteur *C* pour le développement de grandes applications.

Les macro-définitions ayant la même forme que les fonctions, sans en avoir le coût d'exécution, certaines optimisations sont réalisées en remplaçant des fonctions frotement sollicitées par des macro-définitions équivalentes.

```
#define RationnelNumerateur(r) r->numerateur
#define RationnelDenominateur(r) r->denominateur
```

Une macro-définition conduisant à un simple remplacement textuel, il peut se produire des effets pervers dont il faut se méfier. Considérons, par exemple, la définition suivante

```
#define somme(x,y) x+y
```

et le fragment de programme

```
...
x = 3*somme(4,5);
...
```

qui prend la forme

```
...
x = 3*4+5);
...
```

ce qui n'est certainement pas le résultat souhaité.

La définition correcte serait

```
#define somme(x,y) (x+y)
```

9.13.4 Traitement conditionnel.

Il arrive fréquemment qu'une application (ou un paquetage) soit cliente, simultanément, d'un paquetage A et d'un paquetage B dont la définition utilise la paquetage A. On peut imaginer un paquetage **Dictionnaire** utilisant à la fois le paquetage **Paire** et le paquetage **Liste**. Il va alors se produire des redéfinitions que le traducteur C ne tolère pas. Il faut donc faire en sorte de ne pas importer une interface qui a déjà été importée.

Il arrive fréquemment, également, qu'on soit amené à développer une application devant être compatible avec différentes plateformes de développement. On veut, par exemple une application pouvant fonctionner à la fois sous Unix, sous MS-DOS et Macintosh. Ces systèmes fonctionnent à l'aide de bibliothèques de fonctions en général incompatibles.

Ce problème est facilement résolu en incluant des commandes conditionnelles construites à partir des commandes

```
#if <prédicat>
#ifdef <symbole>
```

```
#ifndef <symbole>
#else
#endif
```

Il n'est donc pas rare de rencontrer des fichiers d'interface et des fichiers d'implémentation de la forme suivante

```
    ...
    ...<portion de C multi-plateforme>
    ...
#ifdef MSDOS
    ...
    ...<portion de C pour une plateforme MS-DOS>
    ...
#endif
#ifdef UNIX
    ...
    ...<portion de C pour une plateforme UNIX>
    ...
#endif
#ifdef MACINTOSH
    ...
    ...<portion de C pour une plateforme Macintosh>
    ...
#endif
    ...
```

Selon la constante définie (dans un fichier de configuration, bien sûr) les fragments de programmes incompatibles avec la plateforme ciblée ne seront pas soumis au traducteur.

En ce qui concerne les fichiers d'interface, il est très souhaitable de les définir de façon conditionnelle.

L'interface du paquetage de **Paire** devient, par exemple

```
#ifndef PAIRE_DEFINIE

struct paire {
    void *car;
    void *cdr;
};
typedef struct paire Paire, *PAIRE;

PAIRE cons(void *,void *);
PAIREnil(void);
int null(PAIRE);
void * car(PAIRE);
void * cdr(PAIRE);
void set_car(PAIRE,void *);
void set_cdr(PAIRE,void *);

#define PAIRE_DEFINIE
#endif
```

9.14 Conclusion.

Nous n'avons vu qu'une toute petite partie du *langage C*, mais vous avez pu constater qu'elle permet de programmer des applications déjà fort complexes. L'approche que nous avons suivie:

1. Analyse du problème (en utilisant tous les outils de raisonnement à notre disposition comme les mathématiques, la logique, le bon sens, l'expérience...) , et spécifications des besoins,
2. Définition d'une architecture de solution,
3. Spécification des éléments de cette architecture,
4. Formulation de tout ce qui précède à l'aide d'un langage de raisonnement (*Scheme* par exemple - cela permet de vérifier la cohérence de ce qui précède),
5. Programmation dans un environnement de développement (centré autour de langages comme C, Pascal, Ada, Fortran ...),

présente un grand caractère de généralité. C'est une bonne stratégie qui, bien que perfectible (les chercheurs y travaillent), constitue notre meilleur outil de travail.

Nous avons pu constater que nos schémas de raisonnement sont simples et en nombre finalement très faible (même s'ils ne sont pas toujours évidents à mettre en oeuvre). Ils présentent, cependant, une richesse suffisante pour aborder avec succès la plupart des problèmes informatiques qu'on rencontre couramment [14].

9.15 Exercices.

Traduire en *langage C* les exercices posés dans les chapitres précédents sera un excellent entraînement.

E-96 Sur le modèle de la **Pile** (paragraphe 9.9, page 222), donner une implémentation de la **File** telle qu'elle a été spécifiée au paragraphe 6.7.3, page 136 du chapitre *Les Structures mutables*.

E-97 Utiliser les spécifications du dictionnaire définies au paragraphe 6.7.3, page 136 du chapitre *Les Structures mutables* pour en implémenter une version *C*.

Nota: *utilisez la liste implémentée dans ce chapitre*.

E-98 Utiliser les spécifications du dictionnaire définies au paragraphe 6.7.3, page 134 du chapitre *Les Structures mutables* pour en implémenter une version *C*.

Nota: *utilisez la liste implémentée dans ce chapitre*.

E-99 En utilisant les spécifications définies dans le cadres des exercices et **E-14** au chapitres *Les Fonctions*, donner une implémentation *C* du monnayeur.

E-100 Donner une implémentation *C* de la fonction de recherche de point fixe définie au paragraphe 2.7.2, page 32 du chapitre *Les Fonctions*.

[14] Il existe, bien sûr, des problèmes qui résistent à ce style d'attaque. Ils constituent alors des domaines de recherche.

E-101 Donner une implémentation *C* de la fonction de recherche dichotomique de la solution de l'équation $f(x)=0$ définie dans le cadre de l'exercice **E-17**.