Programmation Fonctionnelle en HOPE

Michel Billaud
billaud@info.iuta.u-bordeaux.fr
Département Informatique
IUT Université de Bordeaux I

Version 27 août 1999

Table des matières

Préface v

1 Introduction 1
1.1 Un peu d'histoire . . . 1
1.2 La programmation impérative . . . 1
1.3 La programmation déclarative . . . 2
1.4 Quelques langages fonctionnels . . . 4
1.5 La crise du logiciel . . . 5
1.6 La programmation fonctionnelle, une solution d'avenir ? . . . 6

2 Programmer avec des Fonctions 7
2.1 Quelques rappels mathématiques . . . 7
2.2 Les fonctions vues comme des boîtes noires . . . 8
2.3 Transparence référentielle . . . 9
2.4 Une session avec HOPE . . . 11
2.5 Un exemple de programme . . . 12
2.5.1 Exemple : Un distributeur de boissons . . . 12
2.5.2 Le prix des boissons . . . 13
2.5.3 Correspondance entre boutons, réservoirs et boissons . . . 13
2.5.4 Le distributeur . . . 14
2.5.5 Un distributeur qui rend la monnaie . . . 14
2.6 Annexe : Quelques éléments du langage HOPE . . . 14

3 Induction et récursion 17
3.1 Un peu d'histoire . . . 17
3.2 Axiomatique de PEANO . . . 21
3.3 Définitions récursives . . . 22
3.4 Raisonnement par récurrence . . . 25
3.4.1 Preuve formelle . . . 25
3.4.2 Preuve par transformation de programme . . . 26
3.5 Fonctions d'ordre supérieur . . . 27
3.6 À propos des fonctions auxiliaires . . . 28
3.7 Exercices et problème . . . 29

4 Les listes 31
4.1 Axiomatique des listes . . . 31
4.2 Les listes en Hope . . . 32
4.2.1 Le type générique `list` . . . 32
4.2.2 Notations simplifiées . . . 32
4.3 Induction naturelle sur les listes . . . 33
4.4 Quelques méthodes de tri . . . 35
4.4.1 Tri par insertion . . . 35

4.4.2 Tri par partition (version naïve) . . . 36
4.4.3 Tri par partition (version améliorée) . . . 37
4.4.4 Le tri-fusion . . . 38
4.5 Fonctionnelles usuelles sur les listes . . . 39
4.6 Exercices . . . 40
4.6.1 Sur les fonctionnelles . . . 40
4.6.2 Sur les transformations de programmes . . . 40

5 Structures arborescentes **41**
5.1 Les arbres binaires . . . 41
5.1.1 Définition inductive des arbres binaires . . . 41
5.1.2 Définition en Hope . . . 42
5.2 Fonctions sur les arbres . . . 43
5.3 Arbres binaires de recherche . . . 43
5.3.1 Insertion dans un arbre binaire de recherche . . . 44
5.3.2 Expressions arithmétiques . . . 44
5.4 Calcul symbolique . . . 46

6 Supplément : Notions de Sémantique **49**
6.1 Quelques généralités . . . 49
6.1.1 Aspects Syntaxiques d'un langage de programmation . . . 49
6.1.2 Sémantique d'un programme . . . 50
6.2 Notion d'Environnement . . . 51
6.3 Sémantique de l'Affectation . . . 52
6.4 Sémantique de la Composition Séquentielle . . . 53
6.5 Sémantique de la Répétition . . . 53
6.6 Conclusion . . . 54

Préface

Ce document a été écrit vers 1991 pour un enseignement de Programmation Fonctionnelle destiné aux étudiants de 2e année du département informatique de l'IUT "A" de Bordeaux, ainsi qu'à ceux du DEST IOE (Informatique des Organisations Européennes) organisé conjointement avec le *Polytechnic* de Sheffield (Grande-Bretagne).

On y retrouve beaucoup des idées pédagogiques développées par Matthew Love pour le cours de programmation fonctionnelle pour le *MSC of Computer Science* du Polytechnic, idées que j'ai empruntées lors d'un séjour à Sheffield en 1990.

Le texte source (en Write pour Windows 2) a été converti en LaTeX(et légèrement édité) au mois d'août 1999, avec quelques modifications très mineures.

Chapitre 1

Introduction

1.1 Un peu d'histoire

Les ordinateurs sont, dans leur principe, des machines assez simples construites autour de quelques types de circuits élémentaires : registres, mémoires, additionneurs, décodeurs, etc.

Ces machines exécutent séquentiellement des programmes, qui sont des suites d'instructions élémentaires enregistrées dans la mémoire centrale. Les données (également en mémoire) que peuvent manipuler ces instructions appartiennent à quelques types bien connus : nombres entiers ou réels, caractères, adresses, etc.

Cette structure simple, appellée architecture Von Neumann (du nom d'un mathématicien américain d'origine hongroise qui contribua au développement du concept de programme enregistré dans les projets ENIAC et EDVAC, vers 1946), convenait tout à fait aux premières applications des calculateurs (années 40) : il s'agissait d'effectuer des suites fastidieuses de calculs répétitifs, pour établir des tables numériques (par exemple calculs balistiques sur le "directeur de tir antiaérien M9" fabriqué par Bell vers 1942), décrypter des messages secrets (machines britanniques Robinson et Colossus), calculs numériques pour la recherche nucléaire (ENIAC), etc.

Références

L'étude détaillée du fonctionnement est faite dans le cours de Première Année de DUT Informatique intitulé "Architecture des Systèmes Informatiques", et on trouvera sans peine des ouvrages sur ce sujet à la bibliothèque.

Pour une perspective historique, consulter *Préhistoire et Histoire des Ordinateurs* de Robert LIGONNIERE (1987), aux éditions Robert Laffont.

1.2 La programmation impérative

Les programmes écrits à l'époque ne pouvaient guère être compliqués, ne serait-ce qu'en raison de la très faible capacité des mémoires centrales (quelques milliers d'octets). Cette capacité augmentant naturellement au cours du temps (et par conséquent la longueur et la complexité des programmes), on s'est avisé qu'il pouvait être intéressant :

- d'écrire des programmes en utilisant des noms mnémotechniques pour chacune des instructions de la machine (naissance du langage d'assemblage), et en donnant des noms symboliques aux emplacements-mémoire destinés à contenir des valeurs intermédiaires du calcul (autrement dit les variables). Ainsi les programmes sont plus faciles à écrire, et surtout à relire ;

- d'utiliser des langages de programmation *évolués* : chaque instruction en langage évolué est traduite par un *compilateur* en une séquence d'instructions dans le langage de la machine (Fortran, etc). Cela facilite la tâche du programmeur ;
- de rendre les programmes indépendants de la machine utilisée : un programme écrit dans un langage normalisé pourra tourner sans trop de modifications sur des machines de marques différentes (Algol, Cobol, etc.);
- d'autoriser le programmeur à se définir ses propres types de données, à partir des types de base et de contructeurs : tableaux, enregistrements, pointeurs (PL/1, Algol/W, etc.);
- d'imposer une certaine discipline au programmeur (programmation structurée), en limitant l'emploi de l'instruction de branchement *goto* qui rend les programmes spécialement illisibles (Pascal) ;
- d'inciter à la *réutilisation de modules* déjà écrits et à la constitution de bibliothèques de modules, en intégrant aux langages de programmation des mécanismes de modularisation (MODULA, ADA, EIFFEL) ;

Cette évolution considérable nous à fait passer, en moins d'un demi-siècle, du code binaire à l'assembleur, Cobol, Fortran, PL/I jusqu'à ADA, Eiffel, etc. Elle préserve cependant deux traits fondamentaux de l'informatique des origines, à savoir la *séquentialité* des programmes et la notion d'*affectation* (modification du contenu d'une variable), qui caractérisent la "*programmation impérative*".

Dans ce style de programmation, il incombe au programmeur de décrire la suite exhaustive des actions que la machine devra effectuer (affectations, comparaisons, additions, etc.) dans un ordre précis pour parvenir au résultat voulu. C'est tout-à-fait fastidieux. Le programmeur qui aborde une tâche d'une certaine importance se trouverait rapidement emporté par un flot de détails de programmation, s'il n'avait une méthode de travail efficace :

- décomposer les problèmes compliqués en problèmes de plus en plus simples ;
- ne pas essayer de réinventer la roue : connaître et utiliser les algorithmes classiques ;
- réfléchir d'abord, programmer ensuite ;
- laisser une trace écrite de son raisonnement (la fameuse documentation).

Ce genre de métier demande une certaine minutie et beaucoup de ténacité[1], mais il n'est pas requis d'avoir un cerveau particulièrement brillant : une bonne formation suffit.

1.3 La programmation déclarative

La plupart des langages de programmation (il en existe plusieurs milliers) relèvent de la catégorie précédente : Fortran, Cobol, PL/1, Basic, Pascal, Ada, C, etc., pour les raisons historiques évoquées ci-dessus (évolution progressive depuis le langage d'assemblage).

Il existe cependant une autre catégorie digne d'intérêt (pour des raisons que nous expliciterons plus loin) : les langages *déclaratifs* .

Dans un langage déclaratif, programmer c'est essentiellement indiquer à une machine la nature des données dont on dispose d'une part, la nature des résultats que l'on veut d'autre part, plutôt que la séquence de traitements qui mène des unes aux autres.

En quelque sorte, les langages déclaratifs décrivent des *spécifications* de traitements plutôt que des algorithmes :

- une *spécification* résume ce que fait une procédure,

[1] parfois à la limite du comportement obsessionnel

- un a*lgorithme* décrit comment la procédure le fait.

Les spécifications et algorithmes font partie de la documentation interne (commentaires) et externe (dossier de programmation) de tout programme sérieux. Voir l'exemple de la figure. 1.1

```
function PGD (n:integer) : integer;

(* Specification : pour tout entier n>1,
   PGD(n) est le plus grand diviseur de n
   qui lui soit strictement inferieur *)

var d : integer;

begin

(* Algorithme :
    boucle descendante de n-1 a 1
    sortie quand d=1 ou d diviseur de n
    le resultat est dans d
*)
    d := n-1;
    while (n mod d)<>0
       do d:=d-1;
    PGD := d;
end;
```

Figure 1.1: Un programme Pascal bien documenté

Traditionnellement ressentie comme une corvée fastidieuse par les programmeurs[2] qui la remettent toujours à plus tard ("quand le programme tournera"), la documentation des programmes contribue à réduire les coûts de programmation :

- Rédiger une spécification, c'est expliquer la vision que l'on a d'un problème. Vision que l'on doit confronter à celle de l'utilisateur, avant de se lancer dans l'écriture d'un programme. Combien de programmes ont été jetés à la poubelle parce qu'ils ne correspondaient pas du tout à ce que le client avait demandé ?
- Lors de l'écriture des programmes, si on n'est pas capable d'écrire, en bon français, ce que l'on veut faire, il est évident que l'on aura de graves difficultés à dire à une machine stupide (par définition), comment elle doit le faire, que ce soit en Pascal, en Cobol, ou quoi que ce soit.[3]. Il faut donc rédiger la documentation *avant* le programme, et non l'inverse.
- Lors de la mise au point : une procédure est valide si les sous-procédures qu'elle utilise sont correctes et employées conformément à leurs spécifications respectives, et si l'algorithme correspond à ce que la procédure est censée faire. Il est plus facile de vérifier une procédure lorsqu'on a sa spécification sous les yeux : la documentation doit faire partie du programme
- Pour la maintenance : il est inutile de relire tout le code pour savoir si l'on peut modifier une procédure sans conséquences graves sur le reste du programme : l'effet d'une procédure est entièrement décrit dans sa spécification. Et on ne passe plus des heures à se demander ce que fait la procédure `toto`.

[2] et les étudiants

[3] *Le vent souffle toujours dans le mauvais sens pour celui qui ne sait pas où il veut aller*

La programmation déclarative existe sous deux formes :

- La Programmation Logique : une formule logique décrit la relation qui existe entre les données et les résultats. Ce qui est à rapprocher des langages d'interrogation de bases de données relationnelles du type SQL. Le langage le plus connu est Prolog.

- La Programmation Fonctionnelle : tout traitement informatique consiste à calculer des résultats à partir de données, c'est donc une fonction $f : \{Donnees\} \rightarrow \{Resultats\}$ au sens mathématique du terme.

1.4 Quelques langages fonctionnels

Il existe un très grand nombre de langages fonctionnels (La figure 1.2 montre le même exemple écrit en Scheme, ML, Miranda, Hope et FP).

Le premier (et le plus connu) a été conçu par John MacCarthy à la fin des années 50. LISP était, au départ, un langage de traitement de listes (LISt Processing language) comportant un "noyau" purement fonctionnel et - pour des raisons d'efficacité - diverses "améliorations" comme les notions de variable et d'affectation.[4]

LISP a de très nombreux descendants. Un des plus prometteurs est Scheme (1974), qui est très utilisé dans l'enseignement [5].

En 1974 apparaissait ML à l'Université d'Edimbourg (Ecosse). ML était au départ le "méta-langage" d'un système de preuve formelle de fonctions récursives*La démonstration automatique est une des branches de l'intelligence artificielle* : le système LCF (Logic of Computable Functions).

En raison de ses nombreuses qualités, le langage ML a été ensuite redéfini par les mêmes auteurs en reprenant des idées de ML et de HOPE (voir plus loin) pour conduire à SML (Standart ML) (1986). Parallèlement, une équipe de l'INRIA (France) a développé CAML, basé sur le concept de Machine Abstraite Catégorique (1986).

Le langage Hope a également été développé à Edimbourg vers 1980. Son nom provient de l'ancienne adresse de l'Institut pour l'Intelligence Artificielle : Hope Park Square. C'est un langage fonctionnel pur, contrairement à Standard ML qui contient des concepts "impératifs" (variables, affectations, pointeurs, traitement des exceptions). Le langage Miranda est également de la même famille.

A près à la même époque (1978) J.W. Backus - un des inventeurs de FORTRAN (1955) et d'ALGOL (1958) [6] - proposait FP.

Quelques références bibliographiques

- Backus, J.W, (1978). *Can programming be liberated from the von Neumann style ? A functional style and its algebra of programs*, Communications of the ACM, 21, 613-41.

- Cousineau, G., Curien, P.L, Mauny, M., (1985). *The Categorical Abstract Machine, in Proc. Conference on Functional Programming Languages and Computer Architecture*, Nancy, 50-64, LNCS 201, Springer Verlag.

- Burstall, R.M., MacQueen, D.B., Sanella, D.T., (1980). *Hope, an Experimental Applicative Language, CSR-62-80, Department of Computer Science*, University of Edimburgh.

- Gordon M.J., Milner, A.J., Wadsworth, C.P, (1979) *Edimburgh LCF*. LNCS 78. Springer Verlag.

[4]Le concepteur d'un langage de programmation doit toujours trouver un compromis entre un objectif de "propreté" (allant parfois jusqu'au minimalisme) du langage qu'il définit, et des concessions souhaitables à l'*efficacité*. Mais comme le souci d'efficacité s'estompe peu à peu avec l'augmentation des performances des machines, il est donc difficile de juger *a posteriori* du caractère raisonnable des compromis qui ont été décidés il y a plusieurs dizaines d'années.

[5]Notamment en Deug et Licence-Maîtrise d'Informatique à Bordeaux

[6]Rappelez-vous, BNF = Backus-Naur Form...

- Scheme :

```
(define (fac n)
  (if   (eqv? n 0)
        1
        (* n (fac (- n 1 )))
))
```

- ML :

```
fun fac(n) =  if n=0
                 then 1
                 else n*fac(n-1)
```

- HOPE :

```
dec fac : num -> num;
--- fac 0 <= 1;
--- fac n <=  n*fac(n-1);
```

- MIRANDA :

```
fac 0 = 1
fac n = n * fac (n - 1)
```

- FP :

```
def fac = eq0 -> 1 ;  * o [ id, fac o ( - [id, 1] ) ]
```

Figure 1.2: Une fonction dans plusieurs langages

- MacCarthy, J., (1960). *Recursive functions of symbolic expressions and their computation by machine*. Communications of the ACM, 3(4), 184-95.
- Rees, J., Clinger, W., eds. (1986). *Revised3 Report on the Algorithmic Language Scheme.* SIGPLAN Notices, 37-79, Vol 21 n 12, Dec. 1986.
- Turner, D.A., (1985). *Miranda, a non-strict functional language with polymorphic types*, in Proc. Conference on Functional Programming Languages and Computer Architecture, Nancy, 1-16, LNCS 201, Springer Verlag.
- Wirsig, M., Sannella, D., (1987). *Une Introduction à la Programmation Fonctionnelle : Hope et ML*, in Technique et Science Informatiques, vol.6 n 6, 517-525, AFCET-Bordas.

1.5 La crise du logiciel

Que l'informatique soit un secteur en pleine expansion, voila bien un lieu commun journalistique d'une trompeuse évidence. Car l'extraordinaire miniaturisation, l'amélioration fantastique des performances et la chute vertigineuse des prix des composants matériels se sont accompagnés, depuis une bonne dizaine d'années, d'une gigantesque crise du logiciel : la production du logiciel est de plus en plus coûteuse. Quelques éléments :

- la part du logiciel dans les coûts informatiques est de plus de 90 % ;
- un service informatique "normal" consacre plus de 80 % de son activité à la maintenance d'applications existantes ;
- un programmeur moyen produit en moyenne 20 à 30 lignes de code par jour (indépendamment du langage de programmation utilisé!) ;
- Les langages Cobol et Fortran ont été conçus dans les 10 premières années de l'informatique d'entreprise. 30 ans plus tard, ils représentent encore 80 % des programmes utilisés et maintenus ;
- Les langages de programmation classiques ne sont pas adaptés (et c'est un euphémisme) à l'utilisation de machines massivement parallèles (par exemple réseaux de 64000 processeurs).

Références : Feuilletez la presse informatique (professionnelle), ainsi que les ouvrages consacrés au "génie logiciel".

1.6 La programmation fonctionnelle, une solution d'avenir ?

Trois aspects de la programmation fonctionnelle permettent de la considérer comme une solution possible à cette crise du logiciel :

- Les programmes fonctionnels sont généralement beaucoup plus courts que leurs homologues impératifs : ils sont écrits plus rapidement, à moindre coût. Ils sont également plus abstraits [7] (on pourra donc réutiliser tels quels des modules d'un programme déjà écrit) et plus faciles à comprendre (moins de "petits détails" de programmation).
- Les programmes fonctionnels se prêtent bien aux techniques de preuve de programmes et de manipulation formelle (transformation de programmes). C'est un style qui se rapproche des techniques modernes de "spécifications formelles" de programmes.
- Ils peuvent facilement, et avec profit, être implantés sur des machines massivement parallèles (par exemple réseaux de 64000 processeurs).

Ces aspects favorables proviennent de la nature mathématique des programmes fonctionnels : les éléments d'un langage fonctionnel sont des fonctions qui décrivent l'obtention de résultats (sorties) à partir de données (entrées) indépendamment de l'environnement où elles sont utilisées (c'est la *transparence référentielle*). On peut très facilement les combiner entre elles, ce qui n'est pas le cas des programmes impératifs.

[7] Il convient de ne pas confondre les différents sens de l'adjectif 'abstrait'. Cf. Dictionnaire de la Langue française Lexis (éditions Larousse). **Abstrait,e** : adj. (lat. abstractus, isolé par la pensée; 1390)

1. Se dit d'une qualité considérée en elle-même, indépendamment de l'objet (concret) dont elle est un des caractères, de sa représentation, ou de tout ce qui dépasse le particulier pour atteindre le général : la grandeur et la couleur sont des qualités abstraites (= concepts). Les noms abstraits, comme "blancheur" et "politesse", désignent en grammaire une qualité ou une manière d'être (contr. CONCRET).
2. Se dit d'une personne (de son esprit ou de son oeuvre) difficile à comprendre à cause de la généralité de son expression ou, péjor. dont la pensée est vague et exprimée de manière confuse : *Je suivais mal son raisonnement abstrait* (syn. non péjor. SUBTIL, contr. CLAIR). *C'est un écrivain abstrait, qui se refuse à illustrer sa pensée par des exemples concrets* (syn. lit. ABSCONS, ABSTRUS). Un exposé abstrait qui ennuyait l'auditoire (syn. pejor. et fam. FUMEUX; contr. PRECIS).
3. Art abstrait, art qui tend à représenter la réalité abstraite et non pas les apparences de la réalité : *L'art abstrait utilise les lignes et les masses pour traduire l'idée ou le sentiment* (contr. FIGURATIF).

Chapitre 2

Programmer avec des Fonctions

2.1 Quelques rappels mathématiques

Soient A et B deux ensembles quelconques. Le *produit cartésien* $A \times B$ est l'ensemble de tous les couples (a, b) dont le premier élément appartient à A et le second à B. Une *relation* entre A et B est un sous-ensemble de $A \times B$.

Une fonction f de A dans B (notée $f : A \rightarrow B$ est une relation telle que, pour tout élément a de A, il existe *au plus* un élément b de B qui soit en correspondance avec lui. Si un tel élément existe, on dit que f est *définie* pour a et on appelle b l'*image* de a par f, ce que l'on note $b = f(a)$.

Les ensembles A et B sont alors appelés respectivement *ensemble de départ* et *ensemble d'arrivée* de la fonction f.

Le *domaine de définition* de f est la partie de A pour laquelle f est définie, c'est-à-dire les éléments qui possèdent une image.

Une fonction est *totale* si elle est définie sur tout l'ensemble de départ (on dit également que c'est une *application* de A dans B, et on le note $f : A \mapsto B$). Dans le cas contraire, c'est une fonction *partielle*.

Prenons un exemple simple, la fonction *signe* qui à tout nombre entier fait correspondre $positif, negatif$ ou *nul* selon le signe du nombre : nous appellerons naturellement cette fonction signe. L'ensemble de départ est Z, l'ensemble d'arrivée est $\{positif, negatif, nul\}$.

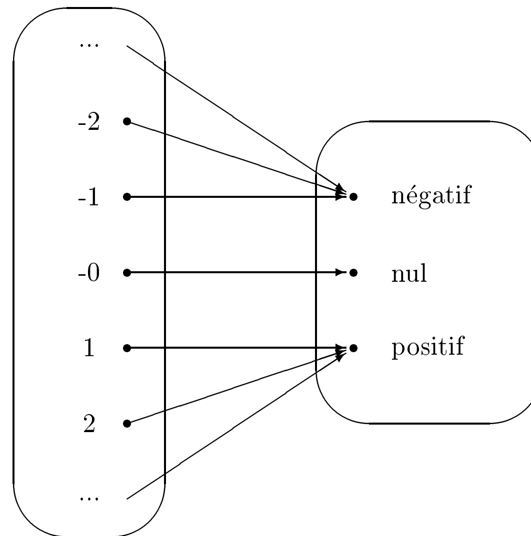
On peut caractériser cette fonction en écrivant la liste des couples qui constituent la relation :

$$signe = \{\ldots(-2, negatif), (-1, negatif), (0, nul), (1, positif), (2, positif), \ldots\}$$

ou bien par une suite d'équations :

$$\begin{array}{rcl} & \vdots & \\ signe(-2) & = & negatif \\ signe(-1) & = & negatif \\ signe(0) & = & nul \\ signe(1) & = & positif \\ signe(2) & = & positif \\ & \vdots & \end{array}$$

ou encore par un diagramme qui montre la correspondance entre les éléments des ensembles de départ et d'arrivée.



Dans tous les cas, il y a un problème de notation évident, dans la mesure où l'ensemble de départ est infini.

On peut aussi décrire *signe* par une simple règle :

$$signe(x) = \begin{cases} negatif & \text{si } x < 0 \\ nul & \text{si } x = 0 \\ positif & \text{si } x > 0 \end{cases}$$

x est appelé le *paramètre formel* de *signe*, et représente n'importe quel élément de l'ensemble de départ de la fonction. Le *corps de la règle* (la partie droite) indique l'élément de l'ensemble d'arrivée qui est en correspondance avec cet x.

Lorsqu'on applique la fonction *signe* à un paramètre *effectif* (par exemple le nombre 6), le corps de la règle est évalué et renvoie la valeur $positif$.

Remarque : Pendant longtemps le concept de fonction ne recouvrait que les fonctions qui possèdent une expression, c'est-à-dire dont on sait déterminer la valeur en tout point. Au XIX ième siècle cette notion s'est généralisée aux correspondances arbitraires, que l'on s'est mis alors à classer selon leurs propriétés (en s'intéressant surtout aux fonctions numériques) : fonctions continues, discontinues, différentiables, intégrables, etc., en faisant apparaître des cas pathologiques défiant l'intuition, comme la fonction de Weierstrass, qui est continue sur un intervalle mais n'est dérivable en aucun de ses points.

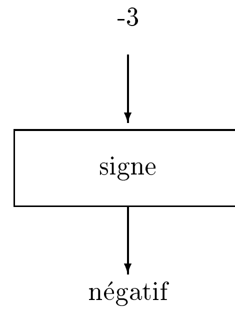
2.2 Les fonctions vues comme des boîtes noires

La plupart des sciences de l'ingénieur (électronique, automatique, mécanique, etc.) utilisent la notion de *boîte noire* [1].

Une boîte noire est un dispositif dont on connaît le comportement (c'est-à-dire les sorties qu'il fournira en réponse à certaines entrées) sans toutefois en connaître (ou en préférant oublier) les détails internes.

On peut voir la fonction *signe* comme une boîte noire avec des entrées (paramètres effectifs) et une sortie (la valeur renvoyée).

[1] qui n'a pas grand chose à voir avec l'aéronautique, où les fameuse boîtes noires sont de couleur orange



La *composition de fonctions* correspond à un agencement de plusieurs boîtes. Voici une fonction qui calcule le maximum de deux nombres :

$$max2(a,b) = \begin{cases} a & \text{si } a > b \\ b & \text{sinon} \end{cases}$$

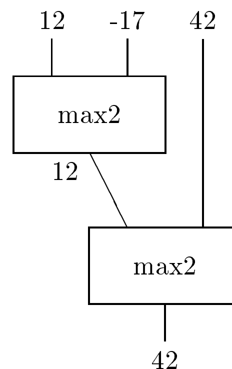
Pour fabriquer une fonction qui calcule le maximum de trois nombres, on peut écrire :

$$max3(a,b,c) = \begin{cases} a & \text{si } a \geq b \text{ et } a \geq c \\ b & \text{si } b \geq a \text{ et } b \geq c \\ c & \text{sinon} \end{cases}$$

mais c'est assez maladroit, car il est clairement préférable d'écrire :

$$max3(a,b,c) = max2(a, max2(b,c))$$

Nous obtenons un assemblage de boites que nous pouvons voir, avec un petit effort d'abstraction, comme une nouvelle boîte noire :



Exercice 2.1 *Représenter la fonction qui renvoie le signe du maximum de 4 nombres.*

A partir de fonctions de base prédéfinies, qui réalisent des opérations simples, nous pouvons donc construire des fonctions de plus en plus élaborées, qui représentent des traitements complexes.

2.3 Transparence référentielle

Si nous pouvons "brancher" à volonté ces boîtes noires entre elles, c'est grâce à une propriété fondamentale des fonctions mathématiques : la *transparence référentielle.* Cette expression [2] signifie

[2] qui peut sembler curieuse à propos de boîtes noires.

simplement que la valeur d'une expression (l'application d'une fonction à des paramètres effectifs) ne dépend pas du tout du référent (le contexte général) qui est donc imperceptible, mais seulement de la valeur des données fournies en entrée.

Pour éclaircir ces propos quelque peu obscurs, considérons l'exemple de la figure 2.1

```
program exemple(output);
var flag : boolean;

    function f (n:integer) : integer;
    begin
      flag := not flag;
      if flag then f := n
              else f := 2*n;
    end;

begin
  flag := true;
  writeln( f(1) + f(2) );
  writeln( f(2) + f(1) );
end.
```

Figure 2.1: L'addition est-elle commutative ?

Si vous avez la curiosité de faire exécuter ce programme, vous ferez une découverte importante : l'addition n'est plus une opération commutative !

Bien entendu, ce n'est pas l'arithmétique qu'il faut remettre en cause, mais plutôt l'assimilation un peu hâtive que l'on peut faire entre les fonctions Pascal et les fonctions mathématiques. Ici la "fonction" `f` dépend d'un "paramètre caché" `flag`, qu'elle modifie par *effet de bord.* .

Cet exemple peut vous sembler artificiel, il est cependant typique de l'esprit des langages impératifs : les procédures consultent et modifient les variables globales. Autrement dit, le résultat d'une évaluation dépend du contexte dans lequel cette évaluation se produit : c'est le contraire de la transparence référentielle.

Un "module logiciel" ne peut être réutilisé a priori que dans un contexte semblable à celui pour lequel il a été conçu, ce qui ne facilite pas la réutilisation de modules déjà écrits : il faudra les adapter, ce qui coûte du temps.

Le programmeur professionnel adoptera donc un style applicatif en évitant les effets de bords ; s'il ne peut pas les éviter, il doit les signaler avec soin dans la documentation. En Génie Logiciel on appelle langages applicatifs les langages de programmation qui interdisent (ou restreignent) l'utilisation d'effets de bords. Les langages fonctionnels purs sont des langages applicatifs : les modules déjà écrits sont immédiatement réutilisables.

Exercice 2.2 *La logique la plus élémentaire nous dit que*

$$A \Rightarrow B$$

et

$$A \Rightarrow C$$

entraînent

$$A \Rightarrow B \text{ et } C$$

Soient maintenant $A =$ *"J'ai 10 Fr",* $B =$*"Je peux me payer un sandwitch à 9 Fr",* $C =$ *"Je peux me payer un demi à 7 Fr". Discutez le paradoxe.*

2.4 Une session avec HOPE

Il existe plusieurs versions du langage Hope, qui ont chacune fait l'objet de plusieurs réalisations. L'interpréteur dont nous disposons sur le HP9000 a été écrit en Pascal (environ 8000 lignes) par un étudiant de Hong-Kong dans une université britannique, et a fait l'objet de portages successifs, ainsi que de diverses corrections. Le produit final est loin d'être parfait, mais il fonctionne suffisamment pour démontrer les concepts de la programmation fonctionnelle.

Dans tout ce qui suit, ce que vous tapez apparaît en caractères soulignés, les réponses de l'ordinateur en maigre, ainsi que nos commentaires qui seront précédés d'un point d'exclamation.

Pour utiliser l'interpréteur Hope sur le HP9000, connectez-vous et tapez la commande :

```
% hope
```

Sur compatibles PC, tapez :

```
A> ichope
```

Il apparaît une bannière, le dialogue peut alors commencer :

```
>: 5+4;                         ! une question comme ca ...
>:  9 : num                     ! le type du resultat est num (nombre)
>: (6-4)*2;
>:  4 : num                     ! sans surprises ...
>: 7 div 3;
>:  2 : num
```

Définissons une fonction, et appliquons-là :

```
>: dec double : num -> num;
>: --- double(n) <= 2*n;
>:
>: double(4);
>:  8 : num
>: double (double (3));
>:  12 : num
>: 1+double(5-17);
>:  -23 : num
```

On peut revoir les définitions :

```
>: display;
dec double  : num  -> num  ;
--- double ( n )
       <=( 2 *  n ) ;
```

Essayons de nouvelles fonctions :

```
>: dec max2 : num X num -> num;      ! X = produit cartésien
>: --- max2 ( a,b ) <= if a>b then a else b;
>: max2 (4,2);
>:  4 : num

>: dec max3 : num X num X num -> num; ! de plus en plus fort
>: --- max3(x,y,z) <= max2(x,max2(y,z));
>: max3(4,12,2);
>:  12 : num
```

Nous pouvons bien sur charger un fichier de fonctions (`ex1.hope` sous Unix, `ex1.hop` sous DOS) :

```
>: load ex1;
data signes == positif ++ negatif ++ nul; ! un ensemble

dec signe : num -> signes ;
--- signe (n) <=     if n<0 then negatif
                else if n>0 then positif
                           else nul;

dec fac : num -> num ;
--- fac (0) <= 1;                                 ! cas particulier n=0
--- fac (n) <= n*fac(n-1);                        ! cas general
```

Essayons ces fonctions :

```
>: signe(12);
>:  positif  : signes
>: signe(6-9);
>:  negatif  : signes
>: fac (2+1) ;
>: 6
>: trace on;                                      ! pour suivre les calculs
>: trace fac;                                     ! de la fonction fac
>: fac(3);
 fac ( 3)                                         ! les appels à fac
 fac ( 2)                                         ! ...
 fac ( 1)
 fac ( 0)
 1                                                ! les réponses
 1                                                ! ...
 2
 6
>:  6 : num
>: trace off;
>: fac(5);
>:  120 : num
>: exit;                                          ! sortie de HOPE
```

2.5 Un exemple de programme

Nous allons écrire maintenant un (petit) programme en HOPE. La programmation fonctionnelle va certainement vous sembler étrange car il n'y aura pas de variables, pas d'affectations, pas de boucles et, à proprement parler, même pas d'instructions !

Au lieu de cela, un programme fonctionnel décrit comment, à certaines données (entrées), on peut associer des résultats (sorties), au moyen de fonctions.

2.5.1 Exemple : Un distributeur de boissons

Pour faire marcher un distributeur de boissons, il faut mettre quelques pièces dans la fente et appuyer sur un bouton. Le breuvage de votre choix s'écoule alors d'un réservoir dans un gobelet.

Bouton	Boisson	Prix	Réservoir
bleu	café	300	1
rouge	chocolat	300	2
vert	pepsi	250	3

Nous allons écrire un programme fonctionnel pour :

- définir le prix des boissons ;
- indiquer la correspondance entre les boutons, les réservoirs et les boissons ;
- indiquer ce qui sort quand on appuie sur un bouton après avoir mis une somme suffisante ;
- indiquer ce qui sort de la machine qui rend la monnaie.

Tout d'abord, précisons les valeurs que peuvent prendre les entrées et sorties :

```
data  bouton == bleu ++ rouge ++ vert ;
data  reservoir == r1 ++ r2 ++ r3 ;
data  boisson == cafe ++ chocolat ++ pepsi ++ rien ;
```

Nous venons de définir trois ensembles, de cardinaux respectifs 3, 3 et 4).

```
type  argent  == num ;
```

Le type `argent` est maintenant synonyme de `num` qui signifie “entier” : cette version de Hope ne connaît pas les nombres en virgule flottante ... Qu'à cela ne tienne, nous compterons en centimes.

2.5.2 Le prix des boissons

Indiquons le nom de la fonction et sa “signature”, c'est-à-dire les ensembles de départ et d'arrivée :

```
dec prix : boisson -> argent;
```

et la valeur des différentes boissons :

```
--- prix(cafe) <= 300;
--- prix(pepsi) <= 250;
--- prix(chocolat) <= 300;
```

Les trois tirets “`---`” ordonnent à l'interpréteur d'ajouter la ligne à la définition de la fonction.

2.5.3 Correspondance entre boutons, réservoirs et boissons

Indiquez maintenant à quel réservoir correspond chaque bouton :

```
dec reserve :   ->  ;

--- reserve (bleu)   <=  r1;

---

---
```

Et le contenu des réservoirs :

```
dec contenu :  reservoir ->  ;

---

---

---
```

2.5.4 Le distributeur

Comment marche le distributeur ? C'est une "boîte noire" que l'on fait fonctionner en mettant de l'argent et en appuyant sur un bouton ; il en ressort alors une boisson. On représente donc le distributeur par une fonction `distrib` :

```
dec distrib :  argent X bouton ->boisson ;
```

Quand on appuie sur le bouton `bou`, cela correspond au réservoir `reserve(bou)` qui délivre la boisson `contenu(reserve(bou))` : le bouton rouge correspond au chocolat par composition des fonctions `reserve` et `contenu`. Mais, attention, nous n'aurons à boire que si nous avons mis assez d'argent, c'est-à-dire une somme au moins égale à `prix(contenu(reserve(bou)))` !

```
--- distrib ( arg , bou ) <=
          if  arg >= prix(contenu(reserve(bou)))
              then contenu(reserve(bou))
              else rien ;
```

Remarques :

- L'ensemble de départ de `distrib` est le produit cartésien de deux ensembles, dont l'un est infini.
- Pour les fonctions `prix`, `reserve` et `contenu` nous avions donné une équation pour chaque élément de l'ensemble de départ. Ici nous écrivons une seule équation, avec deux paramètres formels `arg` et `bou` ; cette équation résume une infinité de cas.

2.5.5 Un distributeur qui rend la monnaie

On met de l'argent et on appuie sur un bouton, il ressort la monnaie et la boisson choisie :

```
dec distrib2 :  argent X bouton -> argent X boisson ;

--- distrib2 (arg , bou ) <=
        if distrib( arg , bou ) = rien
        then ( arg , rien )
        else ( arg - prix(distrib(arg,bou)) , distrib(arg,bou) );
```

Remarque : Ici l'ensemble d'arrivée est également un produit cartésien : les résultats sont des couples notés entre parenthèses.

2.6 Annexe : Quelques éléments du langage HOPE

Domaines prédéfinis

Nom	Signification	Exemples
`num`	nombres entiers	`0,1,2,12654`
`char`	caractères	'a','5','! '
`truval`	valeurs logiques	`true, false`
`alpha,beta`	types variables (génériques)	voir plus loin

Déclaration de domaine

```
data  nom ==  ... ++ ... ++ ... ;
```

Hope ne possède que peu de fonctions prédéfinies : en voici la liste (presque) exhaustive, chacune avec sa signature, et son niveau de priorité quand elle est déclarée infixe.

Opérations arithmétiques

```
infix + , -  : 5 ;
infix * , div , mod : 6 ;
dec + , - , * , div , mod  : (num X num) -> num ;
```

Comparaisons

Ces fonctions permettent de comparer deux objets, à condition qu'ils soient comparables, c'est-à-dire qu'ils appartiennent à un même domaine.

```
infix = , < , > , /= , =< , >= : 6 ;
dec  = , < , > , /= , =< , >= : (alpha X alpha) -> truval ;
```

Opérateurs logiques

```
infix and : 5 ;
infix or  : 4 ;
dec and, or : (truval X truval) -> truval ;
dec not : truval -> truval ;
```

Expressions prédéfinies

Expression conditionnelle :

```
if condition
    then expression
    else expression
```

Expressions qualifiées :

```
let variable == expression
    in expression

expression
  where variable == expression
```

Définition de fonction

```
dec nom : départ -> arrivée  ;
--- nom ( .... ) <= expression ;
...
--- nom ( .... ) <= expression ;
```

Chapitre 3

Induction et récursion

3.1 Un peu d'histoire

Le XIXe est le siècle de la Révolution Industrielle et de la Science Triomphante. C'est l'époque de la foi en un progrès scientifique inéluctable et bénéfique, fondé sur l'étude rigoureuse des faits (positivisme d'Auguste COMTE), et sur l'existence d'un déterminisme qui régirait aussi bien les phénomènes physiques (MAXWELL, BERTHELOT) biologiques (BERNARD, DARWIN), que l'organisation sociale (TAINE, naturalisme de ZOLA) et politique (MARX).

A l'extrême, c'est le scientisme :

> *Une chose évidente d'abord, c'est que chaque découverte pratique de l'esprit humain correspond à un progrès moral, à un progrès de dignité pour l'universalité des hommes. [...] Je suis convaincu que les progrès de la mécanique, de la chimie, seront la rédemption de l'ouvrier ; que le travail matériel de l'humanité ira toujours en diminuant et en devenant moins pénible ; que de la sorte l'humanité deviendra plus libre de vaquer à une vie heureuse, morale, intellectuelle. Aimez la science. Respectez-la, croyez-le, c'est la meilleure amie du peuple, la plus sûre garantie de ses progrès.*
> Ernest RENAN

La mécanisation du calcul

La Révolution Industrielle naît de la mécanisation du travail physique humain. Les travaux de Charles BABBAGE (1792-1871) ont montré à ses contemporains la possibilité de mécaniser également le calcul, c'est-à-dire de faire exécuter des calculs numériques par une machine sous contrôle d'un programme (possibilité théorique surtout, car Charles BABBAGE mourut sans voir sa "Machine Analytique" réalisée, faute de moyens).

C'est le fils ainé de BABBAGE, Henry, qui fabriqua une version très réduite de la Machine Analytique à partir de 1880. Le 21 janvier 1888, la machine imprima une table des 44 premiers multiples de π, avec 29 décimales. Mais un incident technique dû à la technologie employée (roues dentées, cylindres à picots) provoqua une erreur au 32e multiple. Découragé, Henry BABBAGE ne reprit les essais qu'en 1906, il trouva la cause de l'erreur et y remédia. Il put alors faire une démonstration réussie devant l'Académie d'Astronomie. La machine ayant prouvé la justesse des idées de son inventeur, elle fut remise à un musée en 1910.

Lady Ada LOVELACE (1815-1852), l'assistante de Charles BABBAGE, écrivit un jour que la machine "pourrait peut-être ne pas traiter que des nombres".

La formalisation du raisonnement

En 1854, George BOOLE (1815-64) publia son ouvrage *Les Lois de la Pensée* dans lequel il expliquait que le raisonnement logique pouvait être assimilé à du calcul algébrique, relançant alors

la “logique formelle” - étude du raisonnement déductif, des syllogismes, etc. - qui sommeillait depuis quelques siècles. BOOLE rencontra BABBAGE en 1862, mais il mourut trop tôt [1] pour qu'une collaboration ait pu naître.

William Stanley JEVONS (1835-82) fut le seul mathématicien à comprendre immédiatement la portée de l'oeuvre de BOOLE. Il réalisa un “piano logique” qui pouvait résoudre des équations logiques. Très enthousiaste, il aurait sans doute exploré la mécanisation du raisonnement beaucoup plus loin s'il ne s'était noyé accidentellement.

L'axiomatisation des mathématiques

L'état de l'art au XIXe siècle

Les connaissances mathématiques sont bien avancées au XIXe siècle : la plupart des branches ont été créées depuis déjà longtemps :

- Eléments d'EUCLIDE (IIIe avant J.C) : Algèbre, Géométrie, Arithmétique ...
- Traité d'algèbre d'AL-KHWARIZMI (IX-ième siècle)
- Résolution de l'équation du 3e degré par Hieronimo CARDAN (1545), 4e degré par son élève Ludovico FERRARI
- Nombres Négatifs et Complexes : Raphaël BOMBELLI (1526-72)
- Symbolisme Algèbrique : François VIETE (1540-1603)
- Théorie des Nombres : Pierre de FERMAT (1601-65)
- Calcul Infinitésimal : Isaac NEWTON (1642-1727)
- Calcul Différentiel : Gottfried Wilheim LEIBNITZ (1646-1716)

Mais les mathématiciens se heurtent à des paradoxes inextricables, notamment sur l'infiniment petit et l'infiniment grand. Apparaît alors un courant de pensée qui vise, dans le même esprit qu'EUCLIDE, à donner des définitions précises des objets mathématiques que l'on utilisait jusque-là de manière assez intuitive, dans l'intention d'éliminer ces paradoxes.

La démarche d'Euclide

L'oeuvre du grec EUCLIDE (IIIe siècle av. JC) est un modèle de rigueur : ses Eléments récapitulent en une quinzaine de volumes les connaissances mathématiques de l'époque. PROCLUS (Ve siècle après J.C) affirme qu'Euclide, en rassemblant ses Eléments,

> *en a coordonné beaucoup d'Eudoxe, perfectionné beaucoup de Théétète et qu'il a évoqué dans d'irréfutables démonstrations ceux que ses prédécesseurs avaient démontré d'une manière relâchée.*

Le Livre I des Eléments est précédé par une présentation assez intuitive des concepts utilisés dans la suite :

> *Un point est ce qui n'a aucune partie. Une ligne est une longueur sans largeur.*

et par une série de postulats (affirmations admises sans qu'il y ait lieu de les démontrer) qui indiquent les relations entre ces concepts :

> *On demande :*
>
> 1. *qu'on puisse conduire une droite d'un point quelconque à un point quelconque,*

[1] Il contracta une congestion pulmonaire en allant à pied, sous la pluie, donner une conférence au Queen's College de Cork.

2. *qu'on puisse prolonger continuellement, selon sa direction, une droite finie en une droite,*
3. *que d'un point quelconque, et avec un intervalle quelconque, on puisse décrire une circonférence quelconque,*
4. *et que tous les angles droits soient égaux entre eux,*
5. *et que si une droite tombant sur deux droites fait les angles intérieurs du même côté plus petits que deux droits, ces droites, prolongées à l'infini, se rencontreront du côté où les angles sont plus petits que deux droits.*

Après avoir exposé systématiquement les définitions, postulats et axiomes, EUCLIDE en déduit les propriétés élémentaires des triangles, et de leurs bissectrices, milieux des côtés, etc.

Le courant formaliste

Au XIXe les mathématiciens-logiciens du courant "formaliste" (et plus tard Russell, Hilbert, etc.) essaient de formuler les quelques hypothèses (axiomes, postulats) qui sont vraiment indispensables pour définir les objets mathématiques, et dont on pourra déduire toutes les propriétés qui sont "intuitivement vraies".

En 1876 c'est un Allemand, Julius Wilheim Richard DEDEKIND (1831-1916), qui donne une définition formelle de l'ensemble FR des nombres réels à partir des coupures de Q (l'ensemble des rationnels).

> *Une coupure est un partage de Q en deux partitions non vides A et B telles que tout élément de A est plus petit que tout élément de B. A chaque coupure correspond alors un nombre réel unique : par exemple, le nombre irrationnel $\sqrt{2}$ est défini par la coupure*

$$\begin{aligned} A &= \{q | q > 0 \text{ et } q^2 < 2\} \\ B &= \{q | q < 0 \text{ ou } q^2 \geq 2\} \end{aligned}$$

Vers 1880, Georg Ferdinand Ludwig Philipp CANTOR (1845-1918) met au point la théorie des ensembles, qui permet de résoudre de nombreux paradoxes de la théorie des limites liés en fait à l'existence de plusieurs types d'infinis.

Quelques problèmes typiques, que vous pouvez résoudre :

Exercice 3.1 *Montrez qu' il y a autant de couples d'entiers que d'entiers (on peut construire une bijection entre N et N2). Il y a plus de nombres réels dans l'intervalle* $[0, 1[$ *que de nombres entiers : il n'existe pas d'injection de* $[0, 1[$ *dans N).*

Exercice 3.2 *L'ensemble des parties d'un ensemble E est 'plus gros' que cet ensemble : il n'existe pas d'injection de $P(E)$ dans E (résultat dû à CANTOR).*

Malheureusement pour CANTOR, sa théorie fait apparaître à nouveau des paradoxes (comme le paradoxe de RUSSELL : l'ensemble des ensembles qui ne se contiennent pas eux-mêmes se contient-il lui-même ?). La théorie des ensembles sera axiomatisée plus tard par Ernst ZERMELO (1871-1953) en 1908. Génial mais incompris à son époque, CANTOR est mort dans un asile psychiatrique.

En 1889, le mathématicien et logicien Giuseppe PEANO propose (enfin !) une définition formelle de l'ensemble des entiers naturels (positifs).

Mathématiques et Réalité

L'approche "hypothético-déductive" pose le problème du choix du système d'axiomes de base :

- pour LEIBNIZ (et d'autres), "tout ce qui est vrai est démontrable" : pour arriver à tout démontrer toute propriété vraie dans le cadre d'une théorie (c'est la complétude de la théorie), il faut prendre suffisamment d'axiomes ;

- il ne faut pas que les axiomes choisis conduisent à des contradictions (consistance de la théorie) ;
- il faut éviter aussi de choisir des axiomes qui seraient des conséquences des autres axiomes (ils seraient inutiles).

Par exemple, la question s'est longtemps posée de savoir si le fameux 5e postulat d'EUCLIDE

> *par un point extérieur à une droite on peut mener une seule et unique droite parallèle à celle-ci*

était ou non un axiome. De nombreux mathématiciens essayèrent en vain de montrer que ce postulat était une conséquence des autres axiomes de la géométrie d'Euclide, jusqu'en 1826, lorsque LOBATCHEVKI (1793-1856) présente une géométrie non-euclidienne dans laquelle

> *par un point extérieur à une droite passent une infinité de parallèles à cette droite.*

Un peu plus tôt, vers 1813, GAUSS (1777-1844) avait inventé la géométrie hyperbolique,

> *"une étrange géométrie, tout-à-fait différente de la nôtre"*

mais il n'avait pas osé publier ses travaux :

> *"J'appréhende les clameurs des Béotiens si je voulais exprimer complètement mes vues"*

En effet, à l'époque, la géométrie euclidienne était censée *rendre compte de la réalité physique du monde* (KANT); par conséquent ses axiomes étaient donc considérés comme *vrais dans l'absolu* : il était inimaginable de fonder une théorie sur leur négation.

De la même façon, l'hypothèse du continu

> *le cardinal de l'ensemble des nombres réels est le premier qui soit supérieur à celui de l'ensemble des nombres entiers*

formulée par CANTOR qui essaya de la démontrer jusqu'à sa mort, fut montrée *indécidable*[2] par P.J. COHEN en 1963.

L'espoir de trouver un jour quand même un "bon" système d'axiomes qui suffirait à tout démontrer s'effondre en 1931 après la publication du Théorème d'Incomplétude de Kurt GÖDEL (né en 1906) :

> *Toute formulation axiomatique de la théorie des nombres est soit incomplète, soit contradictoire.*

Autrement dit, dans toute théorie T (contenant la théorie des nombres), il existe des propositions indécidables, dont on ne peut démontrer ni la vérité ni la fausseté. Par exemple, la proposition "la théorie T est non-contradictoire" est indécidable dans la théorie T elle-même.

Bonnes lectures :

- Douglas HOFSTADTER (1985), Gödel, Escher et Bach, les Brins d'une Guirlande Eternelle, InterEditions.
- R. APERY, M.CAVEING, et al. (1982) Penser les mathématiques, Points Inédits S29, ed. du Seuil.
- A. DAHAN-DALMEDICO, J. PFEIFFER, (1986). Une histoire des mathématiques, Points Sciences S49, ed. du Seuil.
- Gustave FLAUBERT, Bouvard et Pécuchet, Livre de poche 440-441.

[2] on ne peut pas la démontrer, ni la réfuter : on peut l'ajouter comme nouvel axiome, aussi bien que la proposition contraire

3.2 Axiomatique de PEANO

Cette définition des entiers naturels tient en 5 axiomes, elle repose sur un objet de base (zéro) et un “constructeur” : la fonction qui a tout entier n associe son successeur $n + 1$

1. zéro est un entier
2. tout entier a un successeur, qui est également un entier
3. zéro n'est le successeur d'aucun entier
4. deux entiers différents ont des successeurs différents
5. Principe d'induction : si une propriété P est vraie pour zéro (cas de base) et que P(n) entraîne P(successeur(n)) pour tout entier n, (étape d'induction) alors P est vraie pour tout entier .

C'est le premier exemple d'ensemble défini inductivement : on part d'un objet de base zero (0) auquel on adjoint son successeur (1) puis le successeur du successeur (2), etc. Nous verrons plus loin quantité d'autres ensembles (listes, arbres) définis de la même façon, à partir d'objets de base et de constructeurs.

Le principe d'induction est à la base de la technique de preuve par récurrence : pour prouver qu'une propriété est vraie pour tous les entiers on montre :

- quelle est vraie pour 0 (en général c'est plutôt facile),
- que si elle est vraie pour un entier n (hypothèse de récurrence), alors elle est vraie pour son successeur $n + 1$.

Nous avons donc le droit de définir des fonctions par induction naturelle : une fonction f est définie pour tout entier positif à partir du moment où :

- on connait $f(0)$,
- on peut exprimer $f(n + 1)$ à partir de $f(n)$, pour tout n.

Exercice 3.3 *Construire un ensemble qui satisfasse tous les axiomes de PEANO sauf le 3-ième.*

Exercice 3.4 *Idem, mais en excluant cette fois-ci le 4-ième.*

Exercice 3.5 *Idem, sans le 5-ième.*

Exercice 3.6 *Montrez que l'on peut remplacer le principe d'induction par la variante : << si une propriété Q est vraie pour zero (cas de base) et que Q(zero) et Q(successeur(zero) et .... Q(n) implique Q(successeur(n)) pour tout entier n, alors Q est vraie pour tout entier. >> et comparez les mérites respectifs des deux formulations.*

Le lecteur attentif ne manquera pas de nous soupçonner : peut-on honnêtement prétendre définir les nombres entiers sans parler des opérations arithmétiques comme l'addition, la comparaison, etc. ? Et bien oui ! L'addition, par exemple, n'est pas une notion primitive de la théorie des nombres, mais une fonction que l'on définit par récurrence :

$$\begin{array}{llcl} plus(& 0 & , & p) = & p \\ plus(& succ(n) & , & p) = & succ(plus(n,p)) \end{array}$$

pour tous entiers n et p.

Rassurez-vous, deux plus deux font toujours quatre, en effet

$$deux = succ(succ(zero))$$

et

$$\begin{array}{rcl} plus(deux, deux) & = & plus(succ(succ(zero)), succ(succ(zero))) \\ & = & succ(plus(succ(zero)), succ(succ(zero)))) \\ & = & succ(succ(plus(zero), succ(succ(zero)))) \\ & = & succ(succ(succ(succ(zero)))) \end{array}$$

Exercice 3.7 *Définir la multiplication.*

Exercice 3.8 *Définir la relation inférieur ou égal.*

L'axiomatique de PEANO nous suffit donc pour reconstruire les notions connues de l'arithmétique.

3.3 Définitions récursives

> Il fut saisi par la frénésie des factorielles : 1! = 1 ; 2! = 2 ; 3! = 6 ; 4! = 24 ; 5! = 120 ; 6! = 720 ; 7! = 5 040 ; 8! = 40 320 ; 9! = 362 880 ; 10! = 3 628 800 ; 11! = 39 916 800 ; 12! = 479 001 600 ; [...] 22! = 1 124 000 727 777 607 680 000, soit plus d'un milliard de fois soixante-dix-sept milliards ! Smautf en est aujourd'hui à 76! mais il ne trouve plus de papier au format suffisant et en trouverait-il, il n'y aurait pas de table assez grande pour l'étaler.
>
> La vie mode d'emploi *Georges PEREC*, Livre de Poche 5341 (1978)

Nous allons essayer de spécifier formellement la fonction factorielle, c'est-à-dire de préciser ce qu'elle fait, de manière aussi claire que possible.

Comment spécifier la fonction factorielle?

Premier essai, par une phrase :

> *La fonction factorielle associe, à tout entier positif ou nul n, un autre entier qui est le produit des entiers de 1 à n.*

Cette spécification est correcte : elle identifie clairement et sans ambigüité la fonction dont nous parlons. Mais sa formulation littéraire la rend difficilement exploitable ensuite par le calcul algébrique qui est l'outil de base du raisonnement mathématique : nous préfèrerions une bonne formule.

Seconde tentative, par une expression mathématique :

$$factorielle(n) = 1 \times 2 \times 3 \ldots \times n$$

C'est une définition *elliptique* : il faut un certain effort de la part du lecteur pour comprendre ce qu'il convient de mettre en lieu et place des points de suspension. Elle n'est donc pas idéale.

Et même avant ces points, car avec un peu de mauvaise foi, on pourrait conclure que

$$factorielle(2) = 1 \times 2 \times 3 \times 2 = 12$$

Bref, le "$1 \times 2 \times 3 \ldots$" est à prendre avec des pincettes : lorsqu'il y a des points de suspension ensuite, $1 \times 2 \times 3$ n'est plus égal à 6, mais 1, 2 ou 6 selon les circonstances !

La troisième tentative, et la bonne, sera de définir la fonction factorielle par une spécification récursive .

Construire une spécification récursive

Une spécification est dite *récursive* lorsqu'elle définit un objet mathématique (un ensemble, une relation, une fonction) à l'aide de lui-même.

Ceci doit vous sembler pour le moins obscur, aussi revenons à notre exemple et regardons quelques valeurs de la fonction $factorielle$:

$$\begin{array}{rcl} factorielle(0) & = & 1 \\ factorielle(1) & = & 1 \\ factorielle(2) & = & 1 \times 2 \\ factorielle(3) & = & 1 \times 2 \times 3 \\ factorielle(4) & = & 1 \times 2 \times 3 \times 4 \\ \ldots & & \\ factorielle(n-2) & = & 1 \times 2 \times 3 \times \ldots \times (n-2) \\ factorielle(n-1) & = & 1 \times 2 \times 3 \times \ldots \times (n-2) \times (n-1) \\ factorielle(n) & = & 1 \times 2 \times 3 \times \ldots \times (n-2) \times (n-1) \times n \end{array}$$

Nous remarquons que chaque ligne ne diffère de la précédente que par un seul terme. Par exemple, $factorielle(4)$ est égal à $factorielle(3) \times 4$. Nous sommes donc tentés de définir $factorielle$ par l'équation

$$factorielle(n) = factorielle(n-1) \times n$$

valable pour tout entier positif n, dans laquelle la fonction $factorielle$ est exprimée à partir d'elle-même. Mais il y a un hic : en appliquant cette équation au cas $n = 0$, nous obtenons $factorielle(0) = factorielle(-1) \times 0$ et donc $1 = 0$, ce qui n'est pas raisonnable (sans parler du fait que -1 n'est pas dans le domaine de définition de la fonction).

Il faut donc limiter l'usage de l'équation $factorielle(n) = factorielle(n-1) \times n$ aux cas où n est strictement positif, et préciser ce qui se passe lorsque $n = 0$ (le cas de base). Voici donc une bonne définition de la factorielle :

$$\begin{array}{ll} factorielle(n) & = factorielle(n-1) \times n \quad \text{pour tout entier positif } n \\ factorielle(0) & = 1 \end{array}$$

Traduction en HOPE

Dans le langage HOPE, nous déclarerons cette fonction sous la forme :

```
dec factorielle : num -> num ;
--- factorielle (0) <= 1 ;
--- factorielle (n) <=  factorielle (n-1) * n ;
```

Attention, l'ordre des équations est important en Hope, car l'interprèteur va essayer de les utiliser dans l'ordre où elles ont été déclarées. Le calcul de $factorielle(2)$ se déroule comme suit :

- rejet de la première équation : 2 est différent de la constante 0, donc l'équation ne convient pas.
- essai de la seconde équation : on peut poser $n = 2$. Donc on remplace $factorielle(2)$ par $factorielle(2-1) \times 2$: $factorielle(2) = factorielle(2-1) \times 2 = factorielle(1) \times 2$

reste à résoudre $factorielle(1)$:

- rejet de la première équation car 1 est différent de la constante 0.
- essai de la seconde équation : on peut poser $n = 1$. Donc on remplace $factorielle(1)$ par $factorielle(1-1) \times 2$:

$$\begin{array}{rl} factorielle(2) & = factorielle(1) \times 2 \\ & = factorielle(1-1) \times 1 \times 2 \\ & = factorielle(0) \times 1 \times 2 \end{array}$$

reste à résoudre factorielle(0) :

- la première équation "colle" : on remplace$factorielle(0)$ par1, et on obtient :

$$
\begin{aligned}
factorielle(2) &= factorielle(0) \times 1 \times 2 \\
&= 1 \times 1 \times 2 \\
&= 2
\end{aligned}
$$

Il est clair que si nous avions posé les équations dans l'ordre inverse, nous n'aurions jamais pu détecter le cas de base, et le calcul ne se serait jamais terminé. En règle générale donc, il faut écrire les équations de base en tête, et ensuite les équations de récurrence.

Remarque : on pouvait aussi écrire :

```
dec factorielle : num -> num ;
--- factorielle (n) <= if n=0
                       then 1
                       else factorielle (n-1) * n ;
```

Mais, sous cette forme, on voit moins bien les équations sous-jacentes. Quand c'est possible, il faut éviter l'emploi de la structure conditionnelle.

Exercices :

Exercice 3.9 *Ecrire une fonction* `som` *qui calcule la somme des n premiers entiers strictement positifs, c'est-à-dire :* $som(n) = 1 + 2 + 3 + ... + n$.

Indications :

```
som(0) =
som(1) =
som(2) =
som(3) =
som(4) =
...
```

Exercice 3.10 *Ecrire une fonction* `somcarre` *qui calcule la somme des carrés des n premiers entiers strictement positifs, c'est-à-dire :*

$$somcarre(n) = 1^2 + 2^2 + 3^2 + \ldots. + n^2$$

Exercice 3.11 *Ecrire une fonction* `somcube` *qui calcule la somme des cubes des n premiers entiers strictement positifs, c'est-à-dire :* $somcube(n) = 1^3 + 2^3 + 3^3 + ... + n^3$.

Exercice 3.12 *En utilisant le type prédéfini* `truval` *(*truth value = *booléen) qui possède deux valeurs* `true` *et* `false`*, écrire une fonction* `pair` *qui détermine si un entier (positif) est pair ou non.*

```
pair(0) =
pair(1) =
pair(2) =
pair(3) =
...
```

Exercice 3.13 *Voici la suite de Fibonnacci (Léonard de Pise 1170 ? - 1250) :*

$$1, 1, 2, 3, 5, 8, 13, 34, 55, 89 \ldots$$

Comme vous pouvez le remarquer, chaque élément de cette suite est la somme des deux précédents (sauf les deux premiers, qui sont les cas de base) : il en sera évidemment également ainsi pour les termes suivants.

```
dec fib : num -> num ;

--- fib (1)  <=

--- fib (2)  <=

--- fib (n) <=
```

Exercice 3.14 *Et vous ne pouvez ignorer le triangle de Pascal, les fameux coefficients binomiaux :*

c(0,0)=1
c(1,0)=1 c(1,1)=1
c(2,0)=1 c(2,1)=2 c(2,2)=1
c(3,0)=1 c(3,1)=3 c(3,2)=3 c(3,3)=1
c(4,0)=1 c(4,1)=4 c(4,2)=6 c(4,3)=4 c(4,4)=1

3.4 Raisonnement par récurrence

L'écriture d'une spécification récursive est une activité intellectuelle très proche du raisonnement par récurrence. Soit à démontrer par exemple la proposition :

Proposition Pour tout entier n positif ou nul, on a

$$som(n) = \frac{n \times (n+1)}{2}$$

3.4.1 Preuve formelle

Nous allons d'abord présenter une preuve formelle de cette proposition : la démonstration sera une suite d'étapes dont nous montrerons systématiquement les justifications. Ainsi nous serons tout-à-fait sûrs que notre preuve est inattaquable.

Début de la Preuve.

Nous allons montrer que pour tout entier n possède la propriété $P(n)$ définie par :

$$P(n) \equiv \left(som(n) = \frac{n \times (n+1)}{2} \right)$$

- A. P(0) est vrai en effet :

1.	$P(0) \equiv (som(0) = 0)$	par définition de P
2.	$som(0) = 0$	par définition de $som(0)$
3.	$P(0)$ est vrai	conséquence de 1 et 2

- B. pour tout entier n, $P(n) \Rightarrow P(n+1)$, car

4.	$P(n)$ vrai	Hypothèse
5.	$som(n) = \frac{n \times (n+1)}{2}$	conséquence de 4 et définition de P
6.	$P(n+1) \equiv (som(n+1) = \frac{(n+1) \times (n+2)}{2})$	par définition de P
7.	$som(n+1) = som(n) + (n+1)$	par définition de som
8.	$som(n+1) = \frac{n \times (n+1)}{2} + (n+1)$	conséquence de 7 et 5
9.	$som(n+1) = \frac{n \times (n+1)}{2} + \frac{2(n+1)}{2}$	petit calcul
10.	$som(n+1) = \frac{(n+2) \times (n+1)}{2}$	idem.
11.	$P(n+1)$ vrai	conséquence de 6 et 10
12.	$P(n) \Rightarrow P(n+1)$	car l'hypothèse 4 entraîne 11

- C. D'après A et B, et en vertu du principe de récurrence, nous concluons : la propriété $P(n)$ est vraie pour tout entier n positif ou nul.

Fin de la Preuve.

On peut difficilement mettre en doute la rigueur d'un raisonnement présenté sous cette forme !

3.4.2 Preuve par transformation de programme

Prouver qu'une fonction possède une certaine propriété revient à démontrer qu'une autre fonction est constante. C'est évident : sur notre exemple, la propriété :

Propriété Pour tout entier positif ou nul, on a $som(n) = \frac{n\times(n+1)}{2}$.

est vraie si et seulement si la fonction :

```
dec P : num -> truval;
--- P(n) <= ( som(n) = n*(n+1)/2 );
```

vaut `true` pour tout entier positif ou nul.

Nous allons modifier cette définition de `P`, pas-à-pas, jusqu'à ce que nous puissions conclure que `P` vaut toujours `true`.

- Séparation en deux cas, selon que $n = 0$ ou non. On obtient une définition équivalente :

  ```
  dec P : num -> truval
  --- P(0) <= ( som(0) = 0*(0+1)/2 );
  --- P(n) <= ( som(n) = n*(n+1)/2 );
  ```

- Comme $som(0) = 0$ et, quand n est différent de 0, on a $som(n) = som(n-1) + n$, on simplifie :

  ```
  dec P : num -> truval;
  --- P(0) <= true;
  --- P(n) <= ( som(n-1)+n  = n*(n+1)/2 );
  ```

- Mais nous avons les équivalences :

$$\begin{array}{rll} & som(n-1)+n & = \frac{n\times(n+1)}{2} \\ \Leftrightarrow & som(n-1) & = \frac{n\times(n+1)}{2} - n \\ \Leftrightarrow & som(n-1) & = \frac{n\times(n-1)}{2} \end{array}$$

- Mais, d'après la définition initiale de P, on a :

$$\begin{array}{ll} P(n-1) & = (som(n-1) = \frac{(n-1)\times(n+1-1)}{2}) \\ & = (som(n-1) = \frac{(n-1)\times n}{2}) \end{array}$$

  Ce qui amène à redéfinir une dernière fois P :

  ```
  dec P : num -> truval;
  --- P(0) <= true ;
  --- P(n) <= P(n-1) ;
  ```

L'axiome d'induction nous permet de conclure que pour tout entier positif ou nul n, `P(n)=true`.

Exercice 3.15 *Démontrez que*

$$\begin{array}{ll} somcarre(n) & = 1^2 + 2^2 + 3^2 + \ldots + n^2) \\ & = \frac{n(n+1)(2n+1)}{6} \end{array}$$

Exercice 3.16 $somcube(n) = (1^3 + 2^3 + 3^3 + ... + n^3) = som(n)^2$ *(AL-KARAGI, fin du X-ième siècle).*

Exercice 3.17 $1 \times factorielle(1) + 2 \times factorielle(2) + ... + n \times factorielle(n) = factorielle(n+1) - 1$

3.5 Fonctions d'ordre supérieur

Vous avez sans doute remarqué une certaine ressemblance d'écriture entre les fonctions `som`, `somcarre`, et `somcube`. Si l'on vous demandait d'écrire une fonction `tresor` qui calcule, pour tout entier n, la somme :

$$tresor(n) = azor(1) + azor(2) + ...azor(n)$$

dans laquelle `azor` est une fonction de type "`num -> num`", vous trouveriez que l'enseignant abuse de votre bonne volonté, - ou manque sérieusement d'imagination - en vous posant toujours le même genre d'exercices, que vous savez très bien faire.

Dans la vie quotidienne du programmeur, cette situation se produit fréquemment : il vous faudra écrire un programme P2 qui fait presque la même chose qu'un programme P1 que vous avez déjà écrit auparavant : tâche d'intérêt uniquement alimentaire ! Dans un tel cas bien sûr on ne repart pas de zéro : on récupère le texte de P1, et on le bricole un peu.

Par exemple, on "bidouillerait" la définition de `som` en la rebaptisant `tresor`, et en remplaçant `n+som(n-1)` par `azor(n)+tresor(n-1)`. Et le tour est joué !

La programmation fonctionnelle offre une alternative plus séduisante que ce bricolage indigne : c'est l'utilisation de *fonctions d'ordre supérieur*. Expliquons-nous : dans tous les cas il s'agit d'additionner les valeurs $f(1) + f(2) + ... + f(n)$, pour une certaine fonction f qui est soit l'identité, soit la fonction *carre*, la fonction *cube*, *azor* ou ce qu'on voudra. Alors nous allons écrire une *fonctionnelle* qui fera le calcul voulu pour tout entier n et toute fonction f :

```
dec somfonc : num X ( num -> num ) -> num ;
--- somfonc ( 0 , f )  <= 0 ;
--- somfonc ( n , f )  <= f(n) + somfonc (n-1 , f );
```

Vocabulaire : en mathématiques, on appelle *fonctionnelle* ou *fonction d'ordre supérieur* une fonction dont un (au moins) des paramètres est lui-même une fonction.

Maintenant `tresor` s'écrit plus simplement :

```
dec tresor : num -> num ;
--- tresor ( n) <= somfonc ( n , azor ) ;
```

Pour réécrire `somcarre` à l'aide de `trésor`, on peut :

- soit se donner la peine d'écrire une fonction `carre` :

```
dec carre : num -> num
--- carre(n) <= n*n ;

dec somcarre2 : num -> num ;
--- sommcarre2(n) <= somfonc( n , carre ) ;
```

- soit utiliser une *fonction anonyme* :

```
dec somcarre3 : num -> num ;
--- somcarre3(n) <= somfonc( n , lambda (n) => n*n  ) ;
```

En HOPE la forme :

`lambda ( x, y, z ... ) =>` expression

sert à désigner la fonction anonyme qui à $x, y, z \ldots$ fait correspondre une certaine valeur décrite par l'expression. On peut utiliser les lambda-expressions à volonté, tapez par exemple :

```
(lambda (a,b) => 2*a - 3*b) (10,4) ;
```

et vous verrez apparaître ?

3.6 À propos des fonctions auxiliaires

Pour résoudre un problème compliqué, on le décompose en problèmes plus simples. En programmation fonctionnelle, on aura souvent besoin d'écrire des fonctions auxiliaires, pour résoudre une partie d'un problème. Pensez-y car :

- Souvent on ne peut pas faire autrement ! Par exemple il semble difficile décrire la fonction $f(n,k) = 1^k + 2^k + \ldots + n^k$ sans écrire une fonction auxiliaire *a puissance b*
- Vous aurez peut-être besoin de la même fonction auxiliaire ailleurs.
- L'emploi d'une "bonne" fonction auxiliaire peut conduire à un gain d'efficacité énorme.

Par exemple, avec la définition suivante:

```
dec fib : num -> num ;
--- fib(1) <= 1;
--- fib(2) <= 1 ;
--- fib(n) <= fib(n-1) + fib(n-2);
```

le calcul de $fib(n)$ se fait en $fib(n)$ étapes ! (Ici nous comptons une étape chaque fois que nous franchissons une flèche "`<=`") puisque :

- on trouve $fib(1)$ et $fib(2)$ en 1 étape;
- pour calculer $fib(n)$ il faut
  - franchir une étape
  - calculer $fib(n-1)$
  - calculer $fib(n-2)$

et la fonction fib grandit comme n^2.

Il existe heureusement une bien meilleure méthode, pour calculer cette fonction, qui consiste à calculer deux valeurs de fib à la fois. En effet, définissons la fonction g qui renvoie un couple d'entiers, par l'équation

$$g(n) = (fib(n), fib(n+1))$$

Les valeurs successives de g se calculent facilement de proche en proche, car

$$\begin{aligned} g(n+1) &= (fib(n+1), fib(n+2)) \\ &= (fib(n+1), fib(n+1) + fib(n)) \end{aligned}$$

et donc si $g(n) = (a,b)$, alors $g(n+1) = (b, b+a)$.

Voici donc une définition récursive directe de g :

```
dec g : num -> (num X num) ;
--- g ( 1 ) <= (1 , 1) ;
--- g ( n ) <= let (a,b) == g(n-1)
               in (b,b+a);
```

Maintenant fib s'écrit à partir de g :

```
dec fib : num -> num ;
--- fib ( n ) <= let (a,b) == g(n)
                 in a ;
```

Le calcul de $fib(n)$ peut alors être fait en $n+1$ étapes.

3.7 Exercices et problème

Exercice 3.18 *Exercice ("diviser pour règner")*

```
dec mystere : num X num -> num ;
--- mystere(n,p) <= if n>p
                    then 1
                    else let q == (n+p) div 2
                         in mystere(n,q-1)*q*mystere(q+1,p);
```

Que vaut $mystere(1, n)$ *? Prouvez-le !*

Exercice 3.19 *Exercice 2 ("variable tampon")*

```
dec bizarre : num X num X num -> num ;
--- bizarre ( a, 0, c) <= c;
--- bizarre ( a, b, c) <= bizarre(a-1, b, a*c);
```

Que vaut $bizarre(n, p, 1)$ *?*

Exercice 3.20 *Test de primalité*

Un nombre (entier positif) est premier s'il admet exactement deux diviseurs : l'unité et lui même. Ecrire une fonction qui indique si un nombre est premier.

Indications :

- *contrairement à une opinion très répandue, 1 n'est pas premier !*
- *un nombre* $n \geq 2$ *n'est pas premier s'il a un diviseur dans l'intervalle entier* $\{2 \ldots n-1\}$
- $\{a \ldots b\} = \{a \ldots b-1\} \cup \{b\}$ *si* $a < b$.
- p *est divisible par* q *si et seulement si* p *modulo* $q = 0$.

Chapitre 4

Les listes

4.1 Axiomatique des listes

Les *listes* ou *séquences* sont des groupements d'objets qu'on ne peut accéder que dans un certain ordre préétabli.

Par exemple le troisième enregistrement d'un fichier séquentiel ne peut être consulté qu'après avoir lu les deux premiers successivement.

Pour construire l'ensemble $ListeNombres$ (les listes de nombres) par exemple, il nous suffit de deux choses :

- la liste vide,
- un moyen de rajouter un nombre quelconque à une liste. De proche en proche, nous aurons donc la liste vide, puis les listes à un élément, puis à deux éléments, etc.

Ce qui conduit à l'axiomatique suivante :

1. *vide* est une liste de nombres
2. si n est un nombre et l une liste de nombres, alors $cons(n, l)$ est aussi une liste de nombres.
   Ainsi, un élément de base (la liste vide) et une fonction

   ```
   cons : num X ListeNombres - > ListeNombres
   ```

   nous permettent-ils de construire des listes de plus en plus grandes :

   ```
   vide
   cons(732,vide)
   cons(1789, cons(732 , vide))
   cons(1515, cons(1789 , cons(732 , vide)))
   ```

3. Il n'existe pas de couple (n, l) tel que $cons(n, l) = vide$.
4. cons est injective : si $cons(n, l) = cons(n', l')$, c'est que forcément $n = n'$ et $l = l'$.
5. Si une propriété P est vraie pour *vide* et que $P(l) \Rightarrow P(cons(n, l))$ pour tout entier n et toute liste l alors P est vraie pour toute liste de nombres. (Principe d'induction)

On retrouve là une axiomatique semblable à celle de PEANO pour les entiers. C'est bien normal, il suffit de faire correspondre à toute liste sa longueur :

- la liste vide a pour longueur zéro,
- si l est de longueur n, $cons(a, l)$ est de longueur $n + 1 = succ(n)$.

4.2 Les listes en Hope

En Hope, il suffirait d'écrire (si les listes n'étaient pas déjà prédéfinies) :

```
data ListeNombres == vide ++ cons( num X ListeNombres ) ;
```

Ce qui définirait les listes de nombres (`num`), mais pas les listes de caractères (`char`) ou de booléens (`truval`), ni les autres.

4.2.1 Le type générique `list`

Le langage HOPE nous donne les moyens de construire des listes d'objets d'un type arbitraire, par exemple on peut définir des listes de nombres, des listes de caractères, des listes de listes de caractères:

```
type ListeNombres == list(num);
type Chaine == list(char);
type Texte == list(Chaine);
```

En quelque sorte les listes sont d'un type paramétrable (c'est ce qu'on appelle la *généricité*). Les *listes génériques* HOPE sont prédéfinies sous la forme :

```
typevar truc ;
data list (truc) == nil
                 ++  truc ::  list (truc) ;
```

Remarque sur les notations : par commodité "`::`" a été prédéclarée comme étant une opération infixe, ce qui explique pourquoi on écrit :

```
data list (truc) == nil
                 ++  truc :: list (truc)  ;
```

au lieu de :

```
data list (truc) == nil
                 ++  ::( truc X list (truc)  );
```

mais la signification est la même.

Attention, les listes doivent être homogènes !

- `{(3 :: (true :: nil))` n'est pas une liste convenable.
- `(  ('a',true) :: (('b',false) :: nil ))` est homogène : ses éléments sont de type `(char X truval)`.

4.2.2 Notations simplifiées

La notation des listes par des assemblages de `::` et `nil` est peu commode, à cause de la quantité de parenthèses. Hope permet d'écrire les listes entre crochets, les éléments étant séparés par des virgules. Exemples :

```
[1,2,3,4]  =   (1 :: (2 :: (3 :: (4 :: nil) ) ) )
[ ]        =   nil
```

Dans le cas des listes de caractères, on les écrit entre guillemets :

```
"abcd"  =  [ 'a', 'b', 'c', 'd' ]  =  ('a' :: ('b' :: ('c' :: ('d' :: nil ))))
```

Attention, il ne faudra pas confondre :

- `'a'`, qui est de type `char`,
- `"a" = ['a']` qui est de type `list(char)`,
- `["a"] = [ ['a']]` qui est de type `list(list(char))`.

Exercice 4.1 *Quels est le type des listes suivantes ? Vérifiez vos réponses sur machine.*

```
1-      (1 :: ( 2::nil))
2-      ('a' :: ('b' :: nil))
3-      (1 :: nil)
4-      ('w' :: nil)
5-      nil
6-      (false :: (true :: nil))
7-      "berlingot"
8-      [ "pommes","frites"]
9-      [ [ "pommes","vapeur"],["carottes","sautees"]]
10-     ( 12  :: (true :: nil) )
11-     (1 ::2)
```

4.3 Induction naturelle sur les listes

Les listes sont définies à partir d'un objet de base `nil` et d'un constructeur `::`, selon le *schéma d'induction* :

- *nil* est une liste
- si a est un objet et l une liste, alors $(a :: l)$ est une liste

La majeure partie des fonctions que vous aurez à écrire seront définies selon le même schéma d'induction. Par exemple, la fonction qui retourne la longueur d'une liste :

```
dec long : list(alpha) -> num;   ! le type des éléments est indifférent
--- long ( nil )  <= 0
--- long ( a::l ) <= 1+long(l);
```

Ici nous avons défini la fonction *long* en indiquant :

- sa valeur pour l'élément de base *nil*,
- comment on calcule, à partir de la valeur pour une liste l, la valeur de la fonction pour $(a :: l)$.

Nous pouvons suivre les étapes du calcul de $long([12, 7, 4])$:

$long([12, 7, 4])$	$= 1 + long([7, 4])$	en appliquant la seconde équation
	$= 1 + 1 + long([4])$	idem.
	$= 1 + 1 + 1 + long([])$	idem.
	$= 1 + 1 + 1 + 0$	en appliquant la première.
	$= 3$	

Remarque : C'est une fonction *polymorphe*, dans la mesure où elle peut agir sur plusieurs types de données : listes de nombres, de caractères, etc. Le polymorphisme, ajouté à la généricité, donnent aux langages fonctionnels une grande puissance d'abstraction. Dans les langages impératifs, si l'on veut calculer la longueur de listes d'objets de 12 types différents, il faut écrire 12 sous-programmes.

Exercice 4.2 *Ecrire la fonction "somme des éléments d'une liste de nombres" :*

```
dec somme : list ( num) -> num ;

--- somme ( [ ] )       <=

--- somme ( n :: l )    <=   somme(l)                    ;
```

et montrez les étapes du calcul de somme ([2,5,13,3]) :

```
somme ( [ 2,5,13,3 ] )  =
```

Exercice 4.3 *Ecrire une fonction qui indique si un élément est présent ou non dans une liste. Exemples :*

```
element (  'h' , "la cucarracha" ) = true
element ( 12 , [ 2,3,5,7,11,13] ) = false

dec element :  alpha X list(alpha) ->

--- element ( e , [ ] )         <=

--- element ( e , (a::l) )      <=
```

Exercice 4.4 *Ecrire la fonction "nombre d'éléments qui sont supérieurs à une certaine valeur". Exemple :* `super([1,3,12,4,6,2] , 5) = 2` *puisque 2 éléments sont plus grands que 5.*

```
dec super : list ( num ) X num  -> num ;

--- super ( [ ]     , val )     <=

--- super ( n :: l , val )      <=
```

Exercice 4.5 *Ecrire la fonction "liste des éléments qui sont supérieurs à une certaine valeur".*
Exemple : `suplis([1,3,12,4,6,2] , 5) = [12,6]`

```
dec suplis : list ( num ) X num  ->

--- suplis ( [ ]     , val )    <=

--- suplis ( n :: l , val )     <=
```

Exercice 4.6 *Ecrire une fonction qui, à tout entier positif* n*, fait correspondre la liste [*$n, n-1, n-2, \ldots, 2, 1$*]. Exemple :* `{yop(5) = [5,4,3,2,1]`

```
dec yop :
```

```
---

---
```

Exercice 4.7 *Démontrez que pour tout entier n on a*

$$long(yop(n)) = n$$

Exercice 4.8 *Démontrez que*

$$somme(yop(n)) = som(n)$$

Exercice 4.9 *Ecrire une fonction qui concatène deux listes (c-à-d qui les met bout-à-bout). Exemple :* `conc("abra","cadabra") = "abracadabra"`

```
dec conc : list(alpha) X list(alpha) -> list(alpha) ;

--- conc ( [  ]   ,  l2 )       <=

--- conc ( (a::l) , l2 )        <=
```

Remarque : Cette fonction de concaténation est très utilisée en pratique. Par souci d'efficacité, elle a été intégrée à l'interprèteur sous forme d'une fonction prédéfinie “`<>`” (en notation infixe) :

```
"etoile " <> "des neiges" = "etoile des neiges"
```

Exercice 4.10 *Démontrez les propriétés suivantes de la concaténation :*

- *nil est élément neutre pour l'opération conc :*

$$\begin{aligned} conc(nil, l2) &= l2 \\ conc(l1, nil) &= l1 \end{aligned}$$

- *la concaténation est associative :*

$$conc(l1, conc(l2, l3)) = conc(conc(l1, l2), l3)$$

- $long(conc(l1, l2)) = long(l1) + long(l2)$
- $suplis(conc(l1, l2), val) = conc(suplis(l1, val), suplis(l2, val))$

4.4 Quelques méthodes de tri

A titre d'exemple de programmes fonctionnels sur les listes, nous allons maintenant voir quelques méthodes pour trier une liste de nombres.

4.4.1 Tri par insertion

Le principe : Pour trier la liste à 4 éléments `[8,5,12,3]` :

- on enlève un élément (le premier = 8)
- on trie (récursivement) le reste de la liste : on obtient la liste ordonnée `[3,5,12]`
- on insère le premier élément (8) à sa place

- ce qui donne [3,5,8,12] : le résultat voulu.

Mais comment a-t-on trié la liste à 3 éléments `[5,12,3]` ? Et bien, de la même façon :

- on a enlevé le premier (5)
- on a trié le reste : ce qui donnait `[3,12]`
- on a inséré le premier élément (5) à sa place
- et on a obtenu `[3,5,12]`.

Mais comment a-t-on trié `[12,3]` ?

- ...

Mise en oeuvre : Tout d'abord il nous faut une fonction auxiliaire pour insérer un élément à sa place dans une liste ordonnée.

```
dec insertion : num X list(num) -> num ;

--- insertion (element , [ ] )  <=                       ;

--- insertion (element, premier :: reste)
                           <= if element < premier
                                 then

                                 else
```

Ceci fait, nous pouvons écrire la fonction TriInsertion :

```
dec TriInsertion : list(num) - > list(num);

--- TriInsertion ( [ ] )             <=

--- TriInsertion ( premier :: reste ) <=
```

Cette méthode est facile à programmer, mais elle n'est pas très efficace : en effet dans le pire des cas, par exemple la liste `[15,13,8,5,1]` les insertions se font toujours à la fin. Pour insérer 15 dans la liste triée `[1,5,8,13]` il faut 5 étapes de calcul. Pour insérer 13 dans `[1,5,8]` il a fallu 4 étapes, etc.

Donc, toujours dans le pire des cas (une liste ordonnée en sens inverse de longueur n), il faudra effectuer $n + (n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n+1)}{2}$ étapes. Le temps du calcul est donc de l'ordre de n^2.

4.4.2 Tri par partition (version naïve)

Le principe : Pour trier la liste à 5 éléments `[8,12,5,3,9]`

- on met de côté le premier élement (8)n
- on extrait du reste deux listes :
  - les éléments plus petits que 8,
  - ceux qui sont plus grands ;

- ce qui donne deux listes `[5,3]` et `[12,9]`
- on trie ces deux listes (récursivement) ; on trouve alors `[3,5]` et `[9,12]`
- on regroupe : `[3,5] <> (  [8] <>  [9,12] )`
- ce qui donne la liste triée `[3,5,8,9,12]`.

Mise en oeuvre : Il nous faut d'abord deux fonctions, qui extraient respectivement d'une liste les éléments qui sont plus petits (ou plus grands) qu'un certain nombre.

```
dec PlusPetits : num X list(num) -> list(num);

--- PlusPetits ( n , [ ] ) <=

--- PlusPetits ( n , p :: r ) <=


dec PlusGrands : num X list(num) -> list(num);

--- PlusGrands ( n , [ ] ) <=

--- PlusGrands ( n , p :: r ) <=
```

Et la définition du tri par partition s'en suit facilement :

```
dec TriPartition : list(num) -> list(num) ;
--- TriPartition ( [ ] )    <= [ ]  ;
--- TriPartition ( p :: r ) <=
               let pp == PlusPetits(p, r)
           in let pg == PlusGrands(p, r)
           in TriPartition(pp) <>( [p] <> TriPartition(pg) ) ;
```

4.4.3 Tri par partition (version améliorée)

Le coût de calcul peut être diminué par une technique relativement simple. D'abord où est le problème ? Il vient de ce que le coût d'une concaténation `g<>d` est proportionnel à la longueur de la liste `g`. Et donc le coût de l'évaluation de l'expression :

```
TriPartition(pp) <> ( [p] <> TriPartition(pg) )
```

est la somme du coût du tri de `pp` et `pg`, et d'un facteur proportionnel à la taille de `pp`. (C'eût été encore pire en groupant les parenthèses différemment)

Voici la technique : on définit une nouvelle fonction à partir de `TPConc`, apparemment plus compliquée, en ajoutant un paramètre supplémentaire (appelé parfois *paramètre tampon*):

```
dec TPConc : list(num) X list(num) -> list(num);
--- TPConc( premiere , seconde ) <= TriPartition( premiere ) <> seconde ;
```

Remarquez que :

```
TriPartition( liste ) = TPConc( liste , [ ] )
```

Maintenant nous allons voir que nous pouvons redéfinir `TPConc` de manière à n'utiliser, dans sa définition, ni la concaténation , ni `TriPartition`.

Dédoublons `TPConc`, selon que son premier argument est la liste vide ou pas :

```
dec TPConc : list(num) X list(num) -> list(num);
--- TPConc( [ ]  , seconde ) <= TriPartition( [ ] ) <> seconde ;
--- TPConc(  p::r , seconde ) <= TriPartition( p::r ) <> seconde ;
```

En utilisant la définition de `TriPartition` ceci équivaut à :

```
dec TPConc : list(num) X list(num) -> list(num);
--- TPConc( [ ]  , seconde ) <= [ ] <> seconde ;
--- TPConc ( p :: r , seconde) )
           <=  let pp == PlusPetits (p , r)
            in let pg == PlusGrands(p, r)
               in TriPartition(pp) <> ([p] <> TriPartition(pg)) <> seconde) ;
```

La concaténation étant associative, on va pouvoir introduire TPConc :

```
TriPartition(pp) <> ([p] <> TriPartition(pg) )  <> seconde
        = TriPartition(pp) <> ([p] <> (TriPartition(pg)   <> seconde)) ;
        = TriPartition(pp) <> ([p] <> TPConc(pg, seconde) );
        = TriPartition(pp) <> ( p :: TPConc(pg, seconde) ;
        = TPConc( pp , p::TPConc(pg,seconde) )
```

Ce qui nous mène à une version nettement améliorée du tri par partition :

```
dec TPConc : list(num) X list(num) -> list(num);
--- TPConc( [ ]  , seconde ) <=  seconde ;
--- TPConc ( p :: r , seconde) )
           <=  let  pp == PlusPetits (p , r)
             in let pg == PlusGrands(p, r)
             in TPConc( pp , p::TPConc(pg,seconde) );

dec TriPartition : list(num) -> list(num) ;
--- TriPartition ( liste ) <= TPConc( liste , [ ] );
```

On peut montrer que le côut moyen du tri d'une liste de n éléments est proportionnel à $n \times log(n)$, ce qui est bien meilleur que pour le tri par insertion. Cependant le coût maximal (dans le pire des cas, qui est statistiquement très rare) reste de l'ordre de n^2.

4.4.4 Le tri-fusion

Pour terminer, une méthode à la fois élégante et efficace, puisque ses coûts moyens et maximaux sont proportionnels à $n \times log(n)$.

Principe : Pour trier une liste de 7 éléments `[3,5,2,9,7,1,0]` :

- on partage cette liste en deux sous-listes, en prenant un élément sur deux. On obtient alors deux listes `[3,2,7,0]` et `[5,9,1]`
- on les trie, récursivement. On obtient deux listes ordonnées `[0,2,3,7]` et `[1,5,9]`
- on fusionne ces deux listes, ce qui donne le résultat `[0,1,2,3,5,7,9]`.

L'opération de *fusion* ou *interclassement* (algorithme classique en informatique de gestion) consiste à construire une liste ordonnée à partir de deux listes également ordonnées :

```
dec fusion : list(num) X list(num) -> list(num) ;

--- fusion ( [ ] , l2) <=
```

```
--- fusion ( l1, [ ] ) <=

--- fusion ( p1::r1 , p2::r2 ) <=  if p1<p2     then

                                                else
```

Il faut savoir extraire un élément sur deux :

```
dec RangPair, RangImpair : list(num) -> list(num);

--- RangPair ( [ ] )    <= [ ]
--- RangPair ( p::r )   <= RangImpair(r);

--- RangImpair ( [ ] )  <= [ ] ;
--- RangImpair ( p::r ) <= p :: RangPair(r) ;   !  récursivité croisée
```

et le tri-fusion s'écrit facilement :

```
dec TriFusion : list(num) X list(num) -> list(num) ;
--- TriFusion ( [ ] ) <= [ ] ;
--- TriFusion ( [ seul ] ) <= [ seul ] ;
--- TriFusion ( liste ) <= let (l1,l2) == (RangImpair(liste),RangPair(liste))
                             in Fusion( TriFusion(l1) , TriFusion(l2) );
```

Question : Pourquoi devons nous traiter séparément le cas des listes à un seul élément ?

4.5 Fonctionnelles usuelles sur les listes

Les fonctions simples vues en 4.3 se généralisent facilement en fonctionnelles "d'intérêt général". Par exemple la fonction "nombre des éléments supérieurs à une certaine valeur"

```
dec super : list(num) X num -> num ;
--- super ( [ ] , val ) <= 0 ;
--- super ( n::l , val) <= if n>val     then 1+super(l)
                                        else super(l);
```

est un cas particulier de la fonction "nombre des éléments qui possèdent une certaine propriété P". Il suffit de passer en paramètre le prédicat (fonction à résultat booléen) qui indique si un certain x possède ou non la propriété recherchée. Par exemple on passera le prédicat "`lambda (x) => x>5`" pour compter les éléments supérieurs à 5. Voici la fonctionnelle :

```
dec combien : list(num) X (num -> truval) -> num ;
--- combien ( [ ] , P ) <= 0 ;
--- combien ( n::l , P ) <= if P(n)
                            then 1+combien(l)
                             else combien(l);
```

De plus nous n'avons aucune raison de nous limiter aux listes de nombres : cette fonctionnelle marche pour des listes de tous types, à condition bien sûr que le domaine du prédicat soit du type convenable. Nous obtenons la fonctionnelle :

```
dec combien : list(alpha) X (alpha -> truval) -> num ;
--- combien ( [ ] , P ) <= 0 ;
--- combien ( n::l , P ) <= if P(n)
                            then 1+combien(l,P)
                            else combien(l,P);
```

Exercice 4.11 *1. Généraliser* `suplis` *(liste des éléments supérieurs à une valeur) pour obtenir une fonctionnelle* `selection` *qui extrait d'une liste les éléments qui possèdent une certaine propriété (par exemple ceux qui sont pairs, ou ceux qui sont premiers, ou qui sont entre* `"BERTHE"` *et* `"CHARLOTTE"`*, etc.)*

2. Montrez comment écrire `suplis` *à partir de* `sélection`*.*

4.6 Exercices

4.6.1 Sur les fonctionnelles

Exercice 4.12 *Ecrire une fonction de tri à tout faire (il faudra passer en paramètre un prédicat qui représente la relation d'ordre choisie). Par exemple :*

```
TriGeneral ( [ 1,5,4,3,2 ]  , lambda(a,b) => a < b ) = [1,2,3,4,5]
TriGeneral ( [ 1,5,4,3,2 ]  , lambda(a,b) => a > b ) = [5,4,3,2,1]
```

Exercice 4.13 *1. Ecrire une fonction* `liscar` *qui prend comme paramètre une liste de nombres, et renvoie la liste des carrés de ces nombres. Exemple :*

```
liscar [ 2,8,3 ] = [ 4,64,9 ]
```

2. Genéraliser cette fonction pour obtenir en une fonctionnelle `map` *: liste des images par une certaine fonction.*

3. Exprimer `liscar` *à partir de* `map`*.*

Exercice 4.14 • *Trouvez une fonctionnelle* `reduction` *qui généralise les deux fonctions "somme des éléments d'une liste", et "produit des éléments d'une liste".*

- *Montrez que cette fonctionnelle* `reduction` *permet d'exprimer* `map` *aussi bien que* `combien`*.*

4.6.2 Sur les transformations de programmes

(revoir méthode du paramètre supplémentaire) :

Exercice 4.15 *Ecrire une fonction* `iota` *qui à tout entier n fait correspondre la liste des n premiers entiers positifs dans le sens croissant. Exemple* `iota(5) = [1,2,3,4,5]`.

Soit `iotaconc` *la fonction définie par* `iotaconc( n , liste ) = iota(n) <> liste`

- *donnez une définition récursive directe de* `iotaconc`*.*
- *en déduire une définition plus efficace de* `iota`*.*

Exercice 4.16 *Par la même méthode, donnez une définition efficace (conduisant à un coût linéaire) de l'inverse d'une liste. Exemple :* `inverse("pomme" ) = "emmop"`

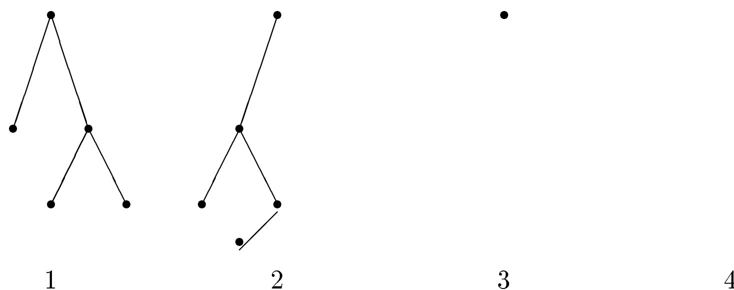
Chapitre 5

Structures arborescentes

5.1 Les arbres binaires

Nul besoin n'est de présenter ici la notion d'arbre : répertoires et fichiers, structure des programmes, tout ou presque est arborescence en Informatique. Dans cette partie nous nous restreindrons à la catégorie la plus simple : les arbres binaires. Nous montrerons un peu plus loin une utilisation (très classique) des arbres binaires pour le tri.

Un arbre binaire est, en quelques mots, un arbre dont les noeuds "portent" deux sous-arbres, à gauche et à droite. Voici quelques exemples (rappelons que, par convention, la racine de l'arbre est en haut) :



Les arbres binaires peuvent être vides : c'est le cas par exemple du sous-arbre droit du second exemple, et aussi du quatrième exemple (invisible). Le troisièmes est réduit à un simple racine. On appelle *feuilles de l'arbre* les sommets dont les deux sous-arbres sont vides.

Exercice 5.1 • *Dessinez tous les arbres à 0, 1, 2, 3 et 4 sommets.*

• *En général, combien y a-t'il d'arbres à n sommets ?*

5.1.1 Définition inductive des arbres binaires

Le plus souvent, on associe une information (*étiquette*) à chaque noeud de l'arbre .

Voici une définition inductive de l'ensemble $ArbreBin(E)$ arbres binaires prenant leurs étiquettes dans un certain ensemble (tout-à-fait arbitraire) E.

Pour commencer, il nous faut un objet représentant l'arbre vide, et une fonction $noeud : ArbreBin(E) \times E \times ArbreBin(E) \rightarrow ArbreBin(E)$ pour fabriquer un arbre à partir de deux arbres plus petits et d'une étiquette.

On pose les axiomes suivants :

1. L'arbre vide appartient à $ArbreBin(E)$
2. si $A1$ et $A2$ appartiennent à $ArbreBin(E)$ et e appartient à E, alors $noeud(A1, e, A2)$ appartient à $ArbreBin(E)$
3. $vide$ n'est l'image par $noeud$ d'aucun triplet $(A1, e, A2)$
4. $noeud$ est injective
5. soit P une propriété sur $ArbreBin(E)$. Si
   - $P(vide)$ est vraie,
   - pour tous $A1, A2$ dans $ArbreBin(E)$ possédant la propriété P, et quelque soit e dans E, on a $P(noeud(A1, e, A2))$;

   alors P est vraie pour tout a dans $ArbreBin(E)$

5.1.2 Définition en Hope

En Hope, c'est une définition d'un type générique :

```
data ArbreBin(alpha) == vide
                     ++ noeud(ArbreBin(alpha) X alpha X ArbreBin(alpha));
```

L'exemple précédent, qui est de type `ArbreBin(char)`, est représenté par l'expression :

```
dec ex1 : ArbreBin(char);
--- ex1 <= noeud( noeud( vide,'b',vide ) ,
                'a' ,
                noeud( noeud(vide,'k',vide) ,
                       'c',
                       noeud(vide,'v',vide) ) ):
```

5.2 Fonctions sur les arbres

La plupart des fonctions sur les arbres sont construites à partir du schéma d'induction naturelle. Par exemple la fonction "Somme des étiquettes d'un arbre de nombres" :

```
dec somarbre : ArbreBin(num) -> num ;

--- somarbre ( vide )   <=

--- somarbre ( noeud(a1,e,a2) )  <=     somarbre(a1)            somarbre(a2)
```

Exercice 5.2 *Ecrire une fonction "liste des étiquettes des feuilles d'un arbre".*

```
dec ListeFeuilles : ArbreBin(alpha) -> list(alpha) ;

--- ListeFeuilles ( vide )  <=

--- ListeFeuilles ( noeud (a1,e,a2) ) <=
```

Exercice 5.3

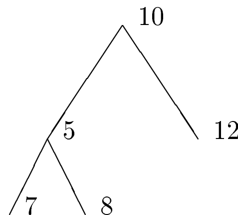
- *Ecrire une fonction $ListePrefixe$ qui renvoie la liste des étiquettes d'un arbre, dans l'ordre préfixe, c'est-à-dire d'abord l'étiquette de la racine, puis celles des sous-arbres gauche et droit. Par exemple :* `ListePrefixe(ex1) = "abckw"`
- *Même question pour l'ordre infixe (d'abord le sous-arbre gauche, puis la racine et enfin le sous-arbre droit). Par exemple :* `ListeInfixe(ex1) <= "bakcw"`
- *Montrez que, pour chacune de ces fonctions, le temps de calcul (évalué en nombre d'étapes) est (dans le pire des cas) proportionnel au carré du nombre de sommets de l'argument.*
- *Transformez ces définitions (méthode du "paramètre tampon") pour obtenir des fonctions dont le temps de calcul soit linéaire.*

5.3 Arbres binaires de recherche

Nous avons vu au chapitre précédent une version du tri par insertion qui n'était guère efficace, parce que nous cherchions à insérer un élément dans une liste. Au lieu de construire une liste, nous allons fabriquer un arbre de recherche qui contiendra tous les objets à trier.

Définition : un *arbre de recherche* est un arbre binaire dans lequel toutes les étiquettes du sous-arbre gauche (resp. droit) sont inférieures (resp. supérieures ou égales) à l'étiquette de la racine, et de même pour tous les sous-arbres de cet arbre.

Exemple :



Exercice 5.4 *Ecrire une fonction qui recherche la plus petite étiquette dans un arbre binaire de recherche (non vide bien sûr).*

```
dec PlusPetite : ArbreBin(num) -> num;

--- PlusPetite
```

Même question pour la plus grande étiquette.

Exercice 5.5 *Ecrire une fonction qui indique si un arbre binaire quelconque est oui ou non un arbre de recherche.*

```
dec estOrdonne : ArbreBin(num) -> truval ;

--- estOrdonne( vide ) <= true ;

--- estOrdonne( noeud(a1,e,a2) ) <=
```

Exercice 5.6 *Montrez que si l'arbre binaire A est un arbre de recherche, alors la liste $ListeInfixe(A)$ est ordonnée.*

5.3.1 Insertion dans un arbre binaire de recherche

Pour insérer 9 dans l'arbre de l'exemple, il faudra d'abord aller à gauche car $9 < 10$, puis à droite puisque $5 < 9$, puis encore à droite car $8 < 9$. La place étant libre, on peut alors y mettre 9.

```
dec Insertion : num  X ArbreBin(num) -> ArbreBin(num);

--- Insertion ( n , vide ) <=

--- Insertion ( n , noeud(a1,e,a2) ) <= if n<e

                        then noeud( Insertion(n,a1) , e , a2 )

                        else
```

Exercice 5.7 *Montrez que si l'arbre A est ordonné, alors Insertion(n,A) est également ordonné.*

Rappelons le principe du tri par insertion. On dispose d'une liste de nombres à trier. On prend les éléments de cette liste, et on les insère un-à-un dans une structure de données qui était vide au départ. Dans la première version la structure était une liste, ici la structure de données c'est un arbre binaire de recherche. Voici comment on fabrique un arbre à partir d'une liste :

```
dec Arbre : list(num) -> ArbreBin(num) ;
--- Arbre ( [ ] ) <= vide ;
--- Arbre ( n::r ) <= Insertion ( n , Arbre( r ) );
```

Et nous obtenons un nouveau tri par insertion :

```
dec TriIns : list( num ) -> list( num ) ;
--- TriIns (liste) <= ListeInfixe (FabriquerArbre ( liste ) ) ;
```

5.3.2 Expressions arithmétiques

Les expressions arithmétiques, comme par exemple $3.a.x + b + 1$ sont également des structures arborescentes familières. Les objets de base en sont les nombres : ici 1 et 3, les variables : a, x, b, et ils sont combinés par les opérateurs arithmétiques : addition, multiplication, etc.

En Hope on peut très facilement créer un ensemble `expr` semblable aux expressions arithmétiques (nous nous limiterons aux 4 opérations de base) en écrivant, dans un premier temps :

```
type chaine == list(char);      ! par commodité

data expr ==                    ! une expression peut être ...
        nombre(num)             !       - un nombre ayant une certaine valeur
++      var(chaine)             !       - une variable avec un nom
++      plus( expr X expr )     !       - la somme ...
++      moins ( expr X expr )   !       - la différence ...
++      mult ( expr X expr )    !       - le produit ...
++      divis ( expr X expr ) ; !       - ou le quotient de deux expressions
```

L'expression "$a.x + b$" sera représentée par :

```
plus( mult ( var("a") , var("x") ) , var("b") );
```

Remarque En Hope on ne peut pas réutiliser les symboles `+`, `-`, etc. comme constructeurs pour de nouveaux types. En effet cette *surcharge* (*overloading*) conduirait à des ambiguités sur les types des objets : la fonction "`+`" serait à la fois de type "`num X num -> num`" et "`expr X expr -> expr`".

Il est préférable de signaler préalablement que `plus`, `moins`, etc. sont des opérateurs infixes avec des priorités semblables à celles de "`+`", "`-`", etc. Ceci conduit à :

```
infix plus, moins : 5 ;
infix mult, divis : 6 ;
type chaine == list(char);

data expr == nombre(num)
          ++ var(chaine)
          ++ expr plus expr
          ++ expr moins expr
          ++ expr mult expr
          ++ expr divis expr ;
```

Déclarons maintenant quelques exemples, sous forme d'une fonction qui renvoie une expression correspondant au numéro d'exemple que l'on veut :

```
dec exemple : num -> expr;
--- exemple(1) <= var "a" mult var "x"  plus var "b" ;
--- exemple(2) <= nombre 1 divis var "x";
--- exemple(3) <= nombre 1 divis exemple(1);
```

Exercice 5.8 • *A quelles expressions correspondent ces trois exemples ?*

```
exemple 1 ->

exemple 2 ->

exemple 3 ->
```

• *Ajouter un exemple 4 représentant* $a.x^2 - b.x + c$.

```
exemple(4) <=
```

5.4 Calcul symbolique

Dans cette partie nous allons développer un petit exemple typique de ce qu'on appelle programmation symbolique ou encore manipulation d'expressions symboliques.

Il s'agit d'écrire un programme capable d'effectuer, comme tout honnête bachelier, la différentiation (ou dérivation) d'une expression par rapport à une variable. Pour un début, nous nous contenterons des expressions simples, limitées aux 4 opérations, que nous avons vues dans la partie précédente.

Vue de près, la différentiation est une opération `diff` qui, à partir d'une expression E et d'une variable V, permet de trouver une autre expression E' qui représente la dérivée de E par rapport à V. Par commodité, nous déclarons `diff` comme opération infixe, ce qui donne :

```
infix diff : 4 ;
dec diff : expr X expr -> expr ;
```

Il ne nous reste plus qu'à étudier les différents cas, qui correspondent aux différentes manières de manières de fabriquer les expressions :

```
--- nombre n            diff  v       <= nombre 0 ;

--- var x               diff  v       <= if var x = v
                                           then

                                           else

--- (f plus g)  diff v  <= let (ff,gg) == (f diff v, g diff v)

                                in ff plus gg ;

--- (f moins g) diff v  <= let (ff,gg) == (f diff v, g diff v)

                                in

--- (f mult g)  diff v  <= let (ff,gg) == (f diff v, g diff v)

                                in

--- (f divis g) diff v  <= let (ff,gg) == (f diff v, g diff v)

                                in
```

Et voilà tout. Ce programme de quelques lignes sait calculer la dérivée d'une expression par rapport à une variable ! Quelques exemples pour s'en convaincre :

```
>: var "x" diff var "x";
>:  nombre ( 1) : expr

>: exemple 1 diff var "x";
>: ((( nombre ( 0) mult  var ("x")) plus ( nombre ( 1) mult  var ("a")))
plus  nombre ( 0)) : expr
```

C'est-à-dire : $0.x + 1.a + 0$

```
>: exemple 2 diff var "x";
>: ((( nombre ( 0) mult  var ("x")) moins ( nombre ( 1) mult  nombre ( 1)))
divis (var ("x") mult  var("x"))) : expr
```

Autrement dit : $\frac{0.x-1.1}{x.x}$

```
>: exemple 3 diff var "x" ;
>: ((( nombre ( 0) mult (( var ("a") mult  var ("x")) plus  var ("b")))
moins ((((nombre ( 0) mult  var("x")) plus ( nombre ( 1) mult  var ("a")))
plus  nombre ( 0)) mult nombre ( 1))) divis ((( var ("a") mult  var ("x"))
plus  var ("b")) mult (( var("a") mult var ("x")) plus  var ("b")))) : expr
```

En clair ?

Bien que surprenants, ces résultats sont tout-à-fait corrects : il suffit de faire quelques simplifications pour retrouver les expressions attendues. Mais pourquoi la fonction n'a-t-elle pas fait ces simplifications ? Tout simplement parce que nous ne l'avons pas demandé !

La manière la plus simple de procéder est de faire les simplifications au moment où l'on construit les expressions. Par exemple nous remplacerons l'équation "dérivée d'une somme d'expressions" par :

```
--- f plus g    diff  v <= let (ff,gg) == (f diff v, g diff v)
                                in ff Plus gg ;
```

où `Plus` est une nouvelle fonction déclarée (préalablement) ainsi :

```
infix Plus : 5 ;
dec Plus : expr X expr -> expr ;
```

La fonction `Plus` construit une somme d'expressions, comme plus, mais elle "sait" effectuer un certain nombre de simplifications :

```
--- nombre n    Plus    nombre p        <= nombre (n+p);
--- nombre 0    Plus    g               <= g;
--- f           Plus    nombre 0        <= f;
--- f           Plus    g               <= f plus g;
```

Exercice 5.9 • *Modifier* `diff` *en introduisant de nouvelles fonctions* `Moins`, `Mult` *et* `Divis`. *(Attention pour la division, Hope ne connait que les nombres entiers.)*

- *Testez sur machine :*

```
exemple 1 diff var "x" =

exemple 2 diff var "x" =

exemple 3 diff var "x" =
```

Chapitre 6

Supplément : Notions de Sémantique

6.1 Quelques généralités

La *sémantique* est un terme de linguistique qui désigne

> *"l'étude du sens (ou contenu) des mots et des énoncés, par opposition à l'étude des formes (morphologie) et à celle des rapports entre les termes dans la phrase" (dictionnaire Lexis).*

A la différence des langues naturelles, les langages de programmation sont des objets linguistiques relativement simples :

- Ayant une syntaxe assez rigide (il faut qu'un programme (le compilateur) puisse analyser une "phrase" sans trop de peine) ;
- Possédant, par rapport aux langues naturelles, une sémantique volontairement très pauvre (une instruction d'un programme doit avoir un sens clairement défini).

6.1.1 Aspects Syntaxiques d'un langage de programmation

Voici par exemple la description de la grammaire d'un petit langage de programmation :

```
<programme>    ::= <entête> <bloc>
<bloc>         ::= début <liste d'instructions> fin
<liste d'instructions>
               ::=   rien
                 |   <instruction> ; <liste d'instructions>
<instruction>  ::= <affectation>
                 |   <bloc>
                 |   <boucle tantque>
                 |   <alternative>
                      ....
<affectation>  ::= <variable> := <expression>
<expression>   ::= <constante>
                 |   <variable>
                 |   <expression> + <expression>
                 |   <expression> * <expression>
                      ....
<tantque>      ::= tantque <condition> faire <instruction>
```

```
<alternative>  ::= si <condition> alors <instruction> sinon <instruction>
<condition>    ::= <expression>  <  <expression>
                 |  ....
                 |  <condition> et <condition>
                 | ...
```

En Hope, nous pouvons définir des objets assez ressemblants :

```
type id         == list(char);

data expr       == nombre(num)
                ++ variable(id)
                ++ expr plus expr
                ++ expr moins expr
                ++ .... ;

data condition  == expr egal expr
                ++ expr infegal expr
                ++ ...
                ++ condition et condition
                ++ non(condition)
                ++ ...

data inst       == id˜:= expr
                ++ bloc(list(inst))
                ++ sialorssinon( condition X inst X inst )
                ++ tantque( condition X inst )
                ++ ...
```

6.1.2 Sémantique d'un programme

En informatique, la sémantique est l'étude de la “signification” des programmes. Comment définir formellement la signification d'un programme ? Prenons un exemple :

```
programme m
donnée en entrée : x,y (entiers positifs)
résultat en sortie : r (entier)
début
     r := 0;
     tantque x>0 faire
          début
               x:=x-1;
               r:=r+y;
          fin
fin
```

La signification de ce programme est une certaine fonction $S(m)$ (S pour Sémantique) qui, à deux entiers x et y, fait correspondre un résultat r en fin de traitement :

$$S(m) : N \times N \to N$$
$$S(m)(n, p) = n \times p$$

Plus généralement, tout programme p a une signification qui est représentée par une certaine fonction $S(p) : D \to R$. Cette fonction est souvent une fonction partielle, dans le sens où le calcul de p sur certaines données peut ne pas conduire à un résultat (le programme boucle, il y a eu

une division par zéro, ...) Le domaine de définition de $S(p)$ est donc naturellement l'ensemble des données pour lesquelles le calcul se termine normalement.

Remarques :

- On note fréquemment $(E \to F)$ l'ensemble des fonctions partielles qui vont d'un ensemble E dans un ensemble F. On pourra donc écrire indifféremment

$$f : E \to F$$

ou

$$f \in (E \to F)$$

- Réfléchissons un peu sur le statut de S dans la notation

$$S(p) : D \to R$$

S est une fonction qui associe à un programme p sa signification, qui est elle-même une fonction qui fabrique un résultat à partir d'une donnée.

Notons P l'ensemble des programmes, on a alors :

$$S : P \to (D \to R)$$

Le problème qui se pose est de savoir comment construire cette fonction S, c'est-à-dire calculer la *signification d'un programme* sans avoir besoin de le faire tourner sur toutes les données possibles.

Nota: Nous limitons ici à une classe très simple de programmes impératifs: programmes structurés, pas d'entrées-sorties, pas de procédures.

6.2 Notion d'Environnement

Le petit programme donné en exemple va nous permettre de préciser quelques notions importantes. Tout d'abord la notion d'*environnement* : un environnement c'est l'association, à un moment donné de l'exécution de ce programme, de certaines valeurs aux variables du programme.

Par exemple si on lance le programme pour $x = 2$ et $y = 3$, on a au départ l'environnement initial

$$e_1 = \{x \equiv 2, y \equiv 3\}$$

puis après l'initialisation

$$e_2 = \{x \equiv 2, y \equiv 3, r \equiv 0\}$$

et ensuite (dans la boucle)

$$e_3 = \{x \equiv 1, y \equiv 3, r \equiv 0\}$$

puis

$$e_4 = \{x \equiv 1, y \equiv, r \equiv 3\}$$

etc. à la fin on aura

$$e_{48} : \{x \to 0, y \to 3, r \to 6\}$$

Deux choses à remarquer :

- Un environnement peut être vu comme une fonction qui part de l'ensemble $Id : chap4 - fin.tex, v1.31999/08/2713 : 34 : 32 billaudExpbillaud$ des identificateurs du programme (ici $Id = x, y, r$) et qui va dans l'ensemble des V des valeurs (ici des entiers).

- Lorsqu'une instruction, (ou une séquence d'instructions) s'exécute, elle modifie l'environnement. On peut donc décrire la signification d'une instruction i (ou d'une séquence par une fonction $S(i)$ qui associe à un environnement (celui où on est juste avant d'exécuter i) un autre environnement (celui d'après l'exécution de i).

$$S(i) : E \to E$$

Par exemple $S(x := x - 1)(e_2) = e_3$

En pratique, on représente souvent un environnement par une liste de doublets (identificateur, valeur).

```
type env == list (chaine X num);
```

Il nous faut alors quelques fonctions auxiliaires

```
dec chercher : chaine X env  -> num ;
--- chercher ( id , [] ) <= 0 ; ! normalement, cas d'erreur
--- chercher ( id,(i,v)::r ) <= if id=i
then v
else chercher(id,r);

dec associer : chaine X num X env  ->  env ;
....
```

La sémantique sera représentée par une fonction :

```
dec S : inst -> (env -> env) ;
```

6.3 Sémantique de l'Affectation

Considérons l'instruction "`r:=ry+`". Nous voulons l'exécuter dans un certain environnement e. Comment fait on ? Tout simplement on additionne les valeurs de r et de y (prises dans l'environnement e) et on modifie la valeur de la variable r dans e ; plus exactement on crée un environnement e' semblable à e, sauf en ce qui concerne la valeur de r.

Nous avons besoin de définir une nouvelle fonction sémantique V (comme valeur) qui renvoie la valeur d'une expression (si cette valeur existe) dans un certain environnement.

par exemple $V(r + y)(e) = 3$

On définit donc :

```
dec V : expr -> (env -> num) ;
```

Et, par induction sur la structure des expressions :

```
--- V(nombre(n))(e)  <= n;
--- V(variable(id))(e)  <= chercher(id,e);
--- V( a plus b ) (e) <= V(a)(e) + V(b)(e)
```

etc ...

En général, si l'on a une affectation `var := expr`, la fonction sémantique S(`var : = expr`) qui lui est associée est définie de la manière suivante :

```
--- S(id := expression)(e)  <= let v == V(expression)(e)
in modifier(id,v,e);
```

6.4 Sémantique de la Composition Séquentielle

Lorsqu'on sait faire une instruction, on peut raisonnablement essayer d'en faire deux à la suite l'une de l'autre ! Interrogeons-nous sur la séquence d'instructions "x:=x-1; r:=r+y".

Supposons que nous l'exécutions en partant de l'environnement e. Aprés la première instruction on se retrouve dans l'environnement $e' = S(\texttt{x: =x-1})(e)$; puis la seconde nous fait passer dans $e" = S(\texttt{r: =r+y})(e')$. Donc

$$\begin{aligned} S(\texttt{x: =x-1; r: =r+y})(e) &= S(\texttt{r: =r+y})(S(\texttt{x: =x-1})(e)) \\ &= (S(\texttt{r: =r+y}) \circ S(\texttt{x: =x-1}))(e) \end{aligned}$$

Pour résumer, si on a deux instructions i et i' :

$$S(i;i') = S(i') \circ S(i)S$$

Ici la composition séquentielle des instructions n'apparaît que dans la structure de bloc, et donc pour des listes d'instructions. Définissons une fonction auxiliaire *Slist* (sémantique d'une liste d'instructions) :

```
dec Slist : list(inst) -> (env -> env);
--- Slist ( [] )(e)  <= e;
--- Slist ( p::r ) ( e ) <= Slist(r) (S (p)(e));
```

On pourra alors poser :

```
--- S ( bloc (l) ) (e)  <= Slist(l);
```

6.5 Sémantique de la Répétition

Examinons maintenant la boucle tant-que. Une boucle tant-que comporte deux éléments : une condition et un corps. Pour évaluer une condition, il nous faut une fonction B (booléenne):

```
dec B : condition -> (env -> truval);
--- B( a egal b)(e) <= V(a)(e) = V(b)(e);
 ...
--- B ( c1 et c2 )(e) <= B(c1)(e) and B(c2)(e);
....
```

Nous savons également définir la signification du corps de la boucle. Comment recoller les morceaux?

Soit *boucle* l'instruction tantque ¡cond¿ faire ¡corps¿/

pour exécuter *boucle* dans l'environnement e, que fait on?

- on évalue la condition (dans e)
- si la condition est fausse, on s'arrête (l'environnement n'a pas changé)
- - si elle est vraie, on exécute le corps (l'environnement est modifié) et on recommence au début (avec ce nouvel environnement e').

Ce qui s'écrit :

```
--- S(tantque(c,i))(e) <= if B(c)(e)
then   S(tantque(c,i))( S(i)(e) )
else   e;
```

Exercice 6.1 1. *Définir la sémantique de l'alternative :* "si <condition> alors <i1> sinon <i2>"

2. *Définir la sémantique de la répétition :* "`répéter <i> jusqu'à <condition>`"
3. *Utiliser cet arsenal mathématique pour démontrer que*
   pour tout $e \in E$ *avec* $e(x) = a$ *et* $e(y) = b$ $(a, b \in N))$, *on a :*

$$(S(m)(e))(r) = a \times b$$

6.6 Conclusion

La programmation s'apprend en général par tâtonnements, ce qui fait parfois conclure un peu hâtivement qu'il s'agit d'un art (ou d'un artisanat, voire un bricolage) plutôt que d'une technique. Il convenait donc de montrer que l'édifice repose sur de robustes fondations mathématiques : composition de fonctions et calcul par récurrence. Qu'en conclure ?

- La notion d'environnement est fondamentale pour la compréhension de la programmation impérative. Cette notion ne fait pas partie du bagage mathématique usuel. D'où les réticences initiales à l'acceptation d'instructions du type : i: =i+1, qui violent l'apparence déclarative.
- toute tentative de compréhension -et d'explication- d'un programme non trivial (comportant par exemple une boucle) utilise nécessairement le raisonnement par récurrence, sous une forme plus ou moins apparente. C'est donc une technique qu'il convient de maîtriser.
- Ceci justifie l'apprentissage de la programmation fonctionnelle, comme langage de programmation familiarisant le programmeur avec le raisonnement explicite par récurrence.