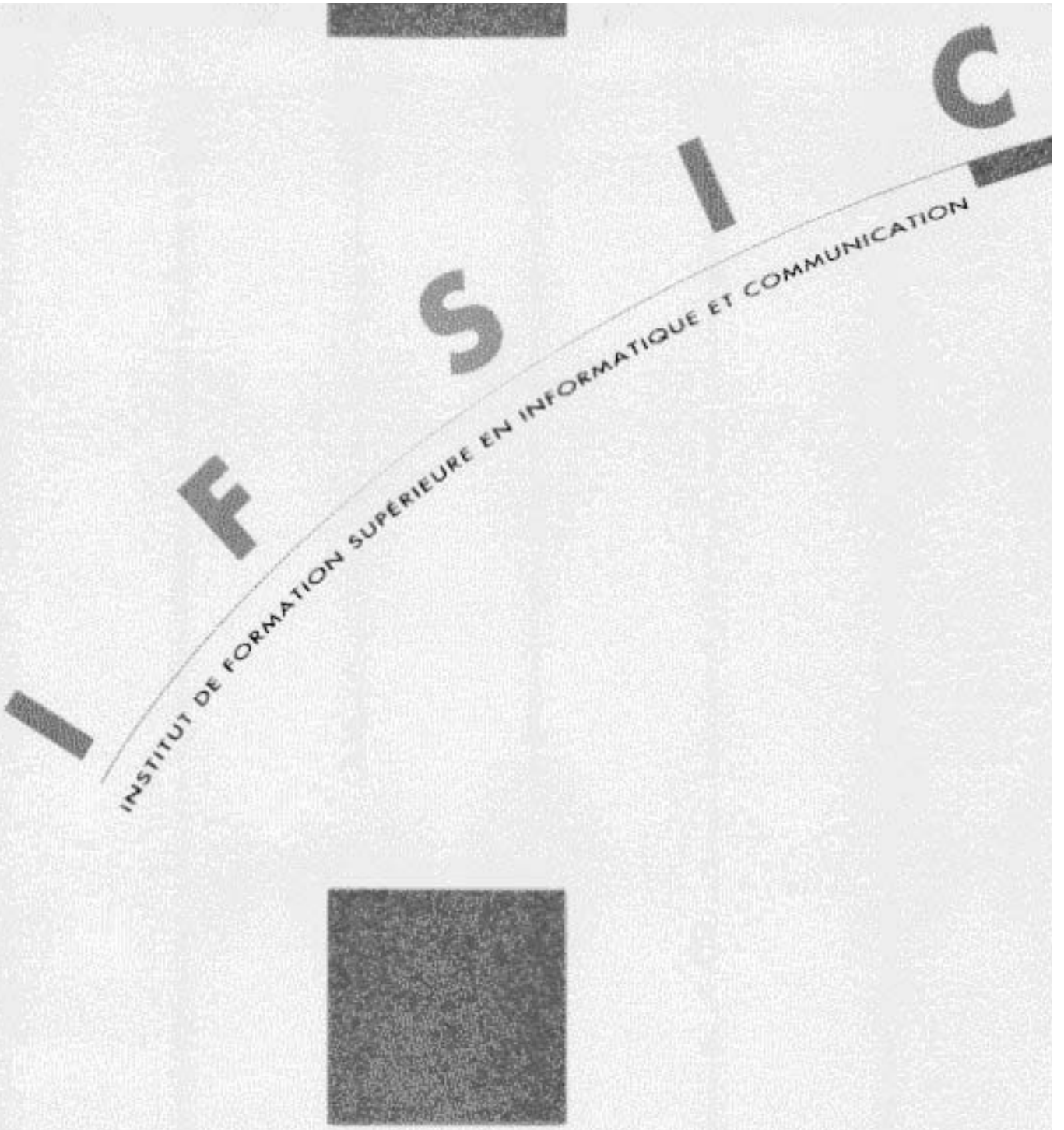


MÉTHODES ET OUTILS DE L'INFORMATIQUE : APPROCHE FONCTIONNELLE

VERSION 2.3

JUIN 2003

A. FORET, **D. H**ERMAN

89

Méthodes et outils de l'informatique: approche fonctionnelle

Cours conçu par
Annie FORET et Daniel HERMAN

Université de Rennes 1 - Ifsic

Avant-propos

Ce cours a été dispensé dans le cadre des Deug scientifiques de l'Université de Rennes 1 de 1990 à 1998. La version 2, destinée aux étudiants du premier niveau du Deug Sciences, mentions Mass et Mias, est une refonte assez profonde d'un cours (version 1.0 en 1991) créé dans le cadre du Deug SSM, section MPI et du Deug Mass. Depuis la création du Deug Sciences et Technologies (1999) cet enseignement n'est plus dispensé en tant que module homogène d'initiation à l'algorithmique, même si la matière qui le compose est reprise dans diverses UE.

Sous sa forme initiale, cet enseignement est conçu pour un volume hebdomadaire de 2 h de cours, 2 h de TD et 2 h de TP sur un semestre de la première année.

Il s'agit du premier cours d'informatique d'un cursus universitaire et, pour la plupart des étudiants, de leur premier contact avec la discipline. L'idée que nous avons retenue est de limiter le champ de l'étude à des notions d'algorithmique et de programmation en faisant l'impasse sur les notions de variable, de fichier et de système. En revanche, nous avons voulu faire porter l'effort de réflexion sur les méthodes de raisonnement (couches d'abstractions, récurrence et structures de données récursives). Compte tenu de ce cahier des charges, le langage Lisp, par le biais d'un de ses dialectes (Scheme), a été retenu comme support d'expérimentation.

Au cours de la seconde année du Deug, les étudiants concernés doivent suivre, en tronc commun, un cours de programmation impérative. À Rennes 1 ils avaient la possibilité de s'inscrire à un cours consacré à la fois à la réalisation d'un projet (génie logiciel) et à des bases plus théoriques d'informatique (logique et théorie des langages).

L'idée d'utiliser une approche fonctionnelle pour l'initiation à l'informatique n'est pas propre à Rennes et de nombreuses expériences similaires ont été développées, tant en France qu'à l'étranger. Au plan hexagonal, on peut, entre autres, consulter:

- Les langages applicatifs dans l'enseignement de l'informatique. Actes de la journée MRT, Paris, 20 mars 1991. Bigre éd.
- Les langages applicatifs dans l'enseignement de l'informatique. Actes des 2es journées, Rennes, 25 et 26 mars 1993. Spécif, bulletin spécial hors série (novembre 1993).

Avertissements

Il y a toujours une différence entre le contenu d'un polycopié et l'enseignement réellement dispensé. Nous avons souhaité que l'étudiant curieux puisse trouver dans un support écrit des compléments ou des approfondissements des sujets effectivement traités. Il y a donc plus d'informations dans cet ouvrage que ce qui est nécessaire pour l'examen. En revanche, tout le travail expérimental (TP et libre service), composante indispensable de tout enseignement d'informatique, n'y est pas abordé. L'étudiant doit donc être conscient de la nécessité d'un minimum de pratique sur machine.

Ce document comporte encore de nombreuses erreurs ou imperfections. Le lecteur est donc prié d'être indulgent et de communiquer ses réactions et commentaires (sur la forme comme sur le fond) à Daniel.Herman@irisa.fr.

Bibliographie

Fallait-il, compte tenu du côté naturellement imparfait de ce type d'exercice, rédiger un polycopié? Puisque le lecteur a en main un tel document, il est clair que nous avons choisi de donner une réponse positive à cette question. Les raisons en sont doubles:

- les étudiants français n'achètent que très peu de livres,
- un enseignement de masse nécessite un référentiel commun à tous les intervenants.

Il est cependant important de donner une bibliographie, et nous avons retenu trois ouvrages permettant d'approfondir les connaissances acquises en première initiation.

Le premier d'entre eux est un cours d'initiation à l'algorithmique et à la programmation fonctionnelle: il poursuit donc des buts semblables aux nôtres.

- Harold ABELSON, Gerald Jay SUSSMAN avec Julie SUSSMAN, *Structure et interprétation des programmes informatiques.* InterEditions (traduction 1989, édition originale 1985).

Le titre qui suit trace un panorama remarquablement complet de l'algorithmique et des outils mathématiques indispensables à l'art de la programmation.

- Alfred V. AHO, Jeffrey D. ULLMAN, *Concepts fondamentaux de l'informatique*, Dunod (traduction 1993).

Enfin, l'ouvrage de vulgarisation scientifique qui termine cette liste évoque une question fondamentale, celle des rapports entre intelligence et machines.

- Douglas HOFSTADTER, *Gödel, Escher, Bach. Les Brins d'une Guirlande Eternelle*. InterEditions (traduction 1985, édition originale 1979).

Remerciements

Parmi les intervenants, il y a, chaque année, de nombreux doctorants. Enthousiastes et sérieux, à l'écoute des étudiants et compétents, ils sont une des plus grandes richesses de notre système éducatif.

L'enseignement de masse suppose une organisation lourde: merci à vous, Anne GRAZON et Sophie ROBIN, tant pour vos talents que pour votre gentillesse.

L'apport de Martine VERGNE est inestimable: ses propositions sont à l'origine de la deuxième version du polycopié.

Serge LE HUITOUZE a pris le temps de nous communiquer de nombreuses remarques ou suggestions: c'est toujours un immense plaisir que d'être confronté à son esprit acéré.

L'enseignement, et surtout l'enseignement en Deug, est un travail d'équipe. Depuis 1990, nombreux sont ceux qui ont fait vivre, donc évoluer, cet enseignement. Que tous ceux qui ont contribué à cette expérience pédagogique soient ici remerciés.

Annie FORET, Daniel HERMAN
Rennes
Janvier 1997

Daniel HERMAN
Rennes
Janvier 1999
Juin 2003

Architecture générale du cours

Le cours comporte 8 chapitres et une annexe; la table des matières et l'index sont à la fin de l'ouvrage.

CHAPITRE 1

Informatique, machines et algorithmes

Ce chapitre, introductif, vise à:

- Introduire les notions de machines, algorithmes, langages et syntaxe.
- Décrire le champ couvert par l'informatique et en montrer la partie centrale.

Le premier de ces deux points a un usage immédiat alors que le second a plutôt une vocation culturelle. En conséquence, seule une petite partie de ce chapitre sera traitée en cours; le reste (la partie culturelle) est à lire et, surtout, à relire à la fin de l'année.

1.1 Informatique et ordinateurs

1.1.1 Essai de définition de l'informatique

Le dictionnaire *Robert* donne du mot informatique la définition suivante:

INFORMATIQUE. Science de l'information; ensemble des techniques de la collecte, du tri, de la mise en mémoire, de la transmission et de l'utilisation des informations traitées automatiquement à l'aide de programmes (logiciels) mis en œuvre sur ordinateur.

Nous retiendrons de cette définition que l'informatique est une discipline qui traite d'informations et d'ordinateurs et nous noterons que les anglo-saxons utilisent, pour couvrir le même domaine, deux termes: *data processing* (traitement de l'information) et *computer science* (science des ordinateurs). Nous nous préoccuperons plus particulièrement du deuxième aspect et définir ce qu'est un ordinateur est un des buts de ce chapitre.

Informatique: science, technique ou outil?
Consultons à nouveau le *Robert*:

SCIENCE. Tout corps de connaissances ayant un objet déterminé et reconnu et une méthode propre; domaine du savoir en ce sens.

TECHNIQUE. Ensemble de procédés méthodiques, fondés sur des connaissances scientifiques, employés à la production.

OUTIL. Objet fabriqué qui sert à agir sur la matière, à faire un travail.

S'il est clair que l'informatique est un outil (on est, tous les jours, confronté à son utilisation: dans les banques, à la Sncf, quand on s'inscrit à l'Université) et une technique (des constructeurs et des sociétés de service -SSII- produisent ce qu'ils vendent), on peut se demander si c'est une science. Notre réponse est positive: d'une part une partie de l'informatique (la science du calcul), antérieure à l'apparition d'ordinateurs est une branche des mathématiques et, d'autre part, les années récentes ont vu se développer une méthodologie spécifique au domaine.

Il est important de comprendre que la discipline informatique présente les trois volets, outil, technique et science. À ce sujet, on fait souvent une comparaison avec le secteur de l'automobile. L'automobile comporte des aspects qui relèvent de sciences diverses: mécanique, thermique, énergétique… C'est une technique, tant au niveau de la production qu'au niveau de la réparation, et c'est enfin un outil lorsqu'on se contente de conduire son véhicule.

Pour ce qui concerne la formation des hommes, on constate, dans l'exemple de l'automobile, que la formation de l'utilisateur de l'outil (dans les auto-écoles) ne fait plus appel à aucun des aspects scientifiques ou techniques. C'est, quoique le phénomène soit plus récent, le cas pour l'informatique.

Dans le cadre de ce cours, nous ne viserons donc pas de formation à l'informatique en tant qu'outil mais nous porterons notre attention sur les aspects scientifiques et techniques.

1.1.2 Ordinateurs: éléments historiques

1.1.2.1 Machines non programmables

Une machine (un calculateur) non programmable effectue un travail (un calcul) f, toujours le même, sur un objet donné en entrée, et délivre un résultat. Ce type d'automatisme, très fruste, nécessite une conduite humaine, que la machine soit sur une chaîne de montage ou qu'elle soit une simple calculette. Une telle machine correspond au schéma de la Figure 1.

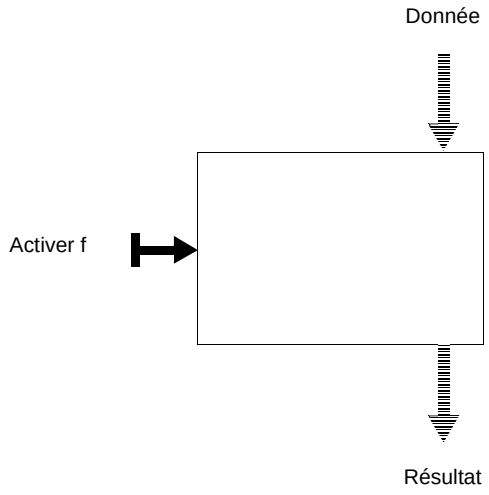
Supposons, par exemple, qu'on veuille réaliser un calcul C qui produise à partir d'une donnée a, une valeur x_{10} ainsi définie:

$$C \begin{cases} x_0 = a \\ x_i = f(x_{i-1}) \end{cases}$$

Une manière de faire consiste à utiliser un homme qui:

- entre a ; récupère x_1 en sortie,
- entre x_1 (la valeur qu'il vient d'obtenir); récupère x_2 en sortie…

Figure 1 Calculateur non programmable



On a évidemment envie de « mécaniser » (automatiser) le procédé ; pour ce faire, il est souhaitable, que le calculateur puisse conserver, entre deux activations successives de sa fonction de base f, la valeur qu'il a produite. Une solution consiste à introduire la notion de *mémoire*.

1.1.2.2 La mémoire

Notre machine (cf. Figure 2) est alors munie de trois commandes : introduction d'une donnée dans la mémoire, sortie d'un résultat (à partir de la mémoire), activation de f, qui travaille sur la mémoire.

Le calcul C précédent peut alors être réalisé de la manière suivante (l'humain est toujours nécessaire mais son travail est simplifié) :

- Entrer a dans la mémoire
- Activer f
- …

  10 fois
- …
- Sortir le résultat.

Une amélioration (Figure 3) consiste à augmenter le nombre d'opérations de la machine: au lieu d'une seule fonction f, on dispose de p fonctions f_1,…, f_p.

Figure 1 — Calculateur non programmable, avec mémoire

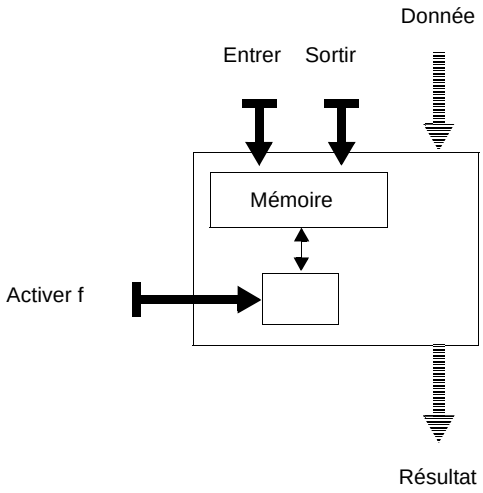
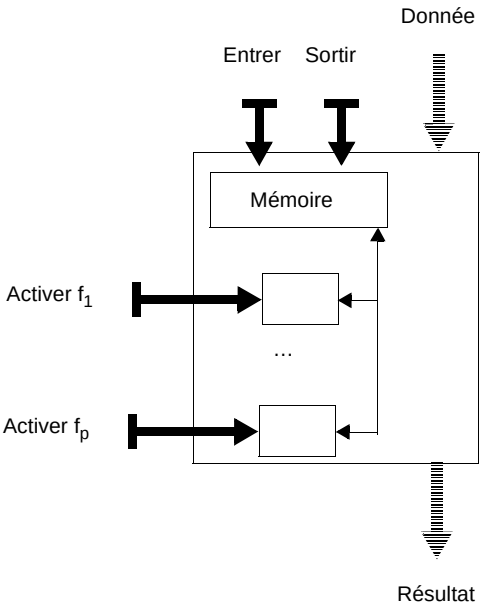


Figure 2 — Calculateur multifonctions, non programmable, avec mémoire



Si $f = f_3$, le calcul C est donc réalisé ainsi :

- Entrer a dans la mémoire,
- Activer f_3, 10 fois,

• Sortir le résultat.

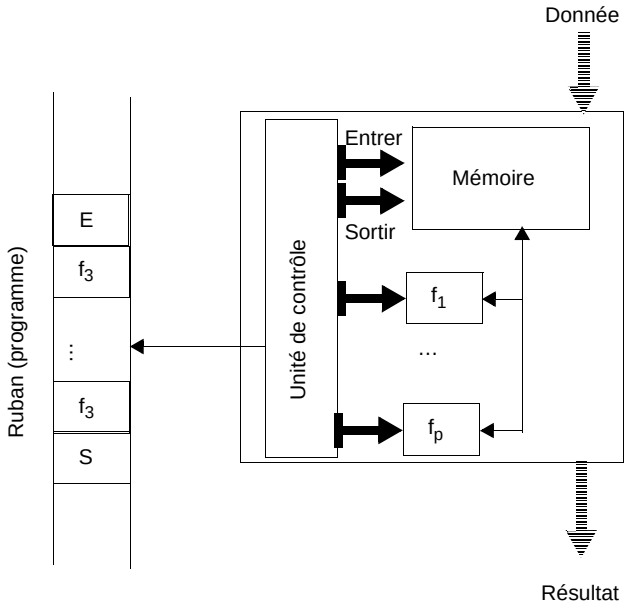
1.1.2.3 Calculateurs à programmes extérieurs et machines de VON NEUMANN

On peut imaginer que la machine est munie d'une tête de lecture devant laquelle peut défiler un ruban, et d'une unité, dite *unité de contrôle* capable de:

• commander le défilement du ruban devant la tête de lecture,
• lire le contenu du ruban et le décoder,
• commander l'activation des fonctions.

On dit que la machine est programmable et que son programme est extérieur.

Figure 3 Calculateur à programme extérieur



Cette idée a vu le jour au début du XIX^{è} siècle. La première machine fonctionnant selon ces principes est due à JACQUARD : c'est un métier à tisser. Peu après, BABBAGE (1839) appliqua les mêmes principes pour réaliser une machine effectuant des calculs numériques : on peut considérer que la machine conçue par BABBAGE est le premier calculateur automatique au monde. Le premier programmeur de tous les temps fut son épouse, Lady Ada LOVELACE, fille de Lord BYRON.

Les deux progrès suivants consistèrent en :

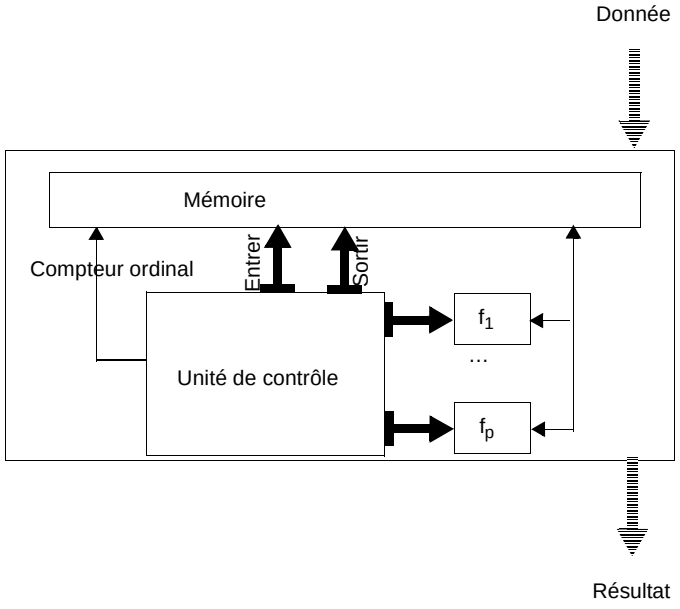
• l'apparition de l'électronique,
• la notion de programme enregistré.

En 1946, est apparu le premier calculateur électronique, Eniac. Comme la machine de BABBAGE, son programme était extérieur. Pour mesurer les progrès de la technologie depuis cette date, on peut noter que Eniac:

- pesait 30 tonnes
- occupait une surface au sol de 160 m^2
- consommait 150 KW
- effectuait une addition en 0,2 ms et une multiplication en 2,8 ms.

Vers les années 30, le mathématicien VON NEUMANN conçut l'idée d'enregistrer les programmes dans la mémoire. L'équivalent de la tête de lecture est ce qu'on appelle le compteur ordinal ; la grande majorité des machines actuelles sont conçues selon ces principes: on les appelle machines de VON NEUMANN ou, plus simplement, *ordinateurs*.

Figure 4 — Ordinateur



Remarque : L'idée que les programmes sont des données comme les autres, inscrites dans la mémoire, est fondamentale : elle permet d'écrire des programmes qui produisent (calculent) d'autres programmes : assembleurs, compilateurs…

Les premiers ordinateurs (Edvac, Edsac) ont vu le jour entre 1946 et 1949.

Depuis, l'évolution est avant tout (mais de manière ô combien spectaculaire) quantitative, tant au niveau des performances que de la taille du parc.

1.1.2.4 Chronologie sommaire

Les années 50
Au plan technologique, la première génération d'ordinateurs utilise des tubes. L'ordre de grandeur du temps nécessaire pour faire une addition est encore de l'ordre de 0,1ms. Trois grandes compagnies, Univac, CDC et IBM, commencent à se faire un nom.

- **1951** Univac 1, premier calculateur à usage civil: 5000 tubes, addition en 0,120ms, 12000 caractères en mémoire.
- **1953** IBM 701 (militaire) et 702 (civil), puis le 650, fleuron d'IBM pour les années 50.

La fin de la décennie voit le passage à la seconde génération: transistors, mémoires à tores de ferrite.

- **1958** CDC 1604 (concepteur Seymour CRAY, qu'on retrouvera): 25000 transistors, 32 K mots de 48 bits (tores de ferrite)

Les années 60
En 1960 on assiste au lancement de la série 7000 d'IBM. Cette série marque le début d'un long règne quasi sans partage de la compagnie qui atteindra rapidement plus de 50% du marché mondial. La décennie est marquée par la série 360 d'IBM, relayée par la série 370 à partir de 1971. On considère que la conception et la mise en fabrication de la série 360 est l'un des grands projets (en taille) menés par l'humanité. C'est en tout cas le premier système moderne (on n'a pas qualitativement fait beaucoup mieux depuis) commercialisé.

Le passage à la troisième génération, celle de l'utilisation massive des circuits intégrés, est effective à la fin des années 60. L'ordre de grandeur du temps nécessaire pour effectuer une addition est 1µs.

Les années 70
Les années 70 sont marquées par l'apparition des micro-processeurs (ordinateurs entièrement intégrés sur une seule puce).

- **1971** La société Intel sort le 4004, premier micro-processeur.

Le « phénomène micro » entraîne une diffusion de l'ordinateur dans tous les secteurs des sociétés industrialisées. Les coûts baissent et les performances augmentent sans cesse. C'est la fin du quasi monopole d'IBM (qui conserve sa place sur son créneau). Le Japon émerge comme pays constructeur.

Les années 80
Outre la poursuite de la montée en charge du phénomène micro, (l'apparition d'IBM sur ce créneau crée un standard de fait, le fameux PC) trois innovations majeures apparaissent au cours de la décennie qui voit également disparaître la notion même de génération:

- l'apparition des super-calculateurs: en 1980 Seymour CRAY lance le CRAY 1, premier super calculateur. On commence, sur ce type de technologie, à

compter en nano secondes. Pour se faire une idée, on calculera quelle distance parcourt la lumière en 1ns;

- la généralisation des réseaux de transmissions de données et des systèmes informatiques dits répartis;
- l'apparition d'interfaces homme-machine plus conviviales. Conçues par la société Xerox, ces interfaces ont été popularisées par la société Apple.

Les années 90

Les progrès technologiques se poursuivent et les micro-ordinateurs atteignent des performances qui leur permettent d'aborder des applications comme le traitement d'images. Les capacités de stockage (CD) ainsi que les débits augmentent également et le poste de travail micro devient un poste multi-média: textes, sons et images couleurs, fixes ou animées.

Les réseaux informatiques continuent à s'étendre (l'Internet a une couverture mondiale) et les réseaux haut-débits préfigurent les « autoroutes » de l'information.

1.2 Algorithmes

1.2.1 Définition

La notion d'*algorithme* du nom du mathématicien arabe AL-KHWARIZMI, VIII^{e} siècle, n'est pas spécifique à l'informatique.

Si on a un problème à résoudre (ou une tâche à accomplir) on peut procéder en deux temps:

1. Décrire (prévoir) une manière d'enchaîner plusieurs opérations élémentaires (élémentaires au sens où soit on sait les faire facilement, soit on dispose d'une machine capable de les faire automatiquement). Il est évident que la description qui est donnée doit être finie.
2. Effectuer les opérations élémentaires prévues en suivant fidèlement la description qui a été donnée de la manière de les enchaîner et obtenir ainsi le résultat.

Le résultat de l'étape 1 est ce qu'on appelle un *algorithme*, l'étape 2 est ce qu'on appelle un *processus de calcul* (ou plus simplement calcul). Un processus de calcul est l'*exécution* d'un algorithme.

1.2.2 Exemples

Exemple 1

Les premiers pas en mathématiques d'un élève de l'école primaire consistent à apprendre des algorithmes: la manière de poser et d'effectuer des opérations (additions et multiplication) sont bien des descriptions (finies) de la manière d'enchaîner des opérations élémentaires (additions et multiplications d'un seul chiffre, d'où les fameuses tables qu'on doit apprendre

par cœur) qui, fidèlement exécutées permettent de réaliser des opérations compliquées.

Il est intéressant de constater que la manière de multiplier deux nombres qui nous a été enseignée n'est pas la seule. Par le passé, en Russie, on utilisait, pour multiplier deux nombres a et b, l'algorithme ci-dessous:

Figure 5 — Multiplication russe[1]

Etape 1	Initialiser un tableau à 2 colonnes avec une première ligne contenant a et b.
Etape 2	Ajouter une ligne au tableau en utilisant la ligne précédente: on divise par 2 le nombre trouvé dans la première colonne et on multiplie par 2 le nombre trouvé dans la seconde. On répète cette étape jusqu'à ce qu'un 1 apparaisse dans la première colonne.
Etape 3	Rayer toutes les lignes du tableau qui correspondent à un nombre pair dans la première colonne.
Etape 4	Faire la somme des nombres restant dans la seconde colonne: on obtient alors le résultat cherché.

Exemple d'exécution

Soit à multiplier 216 par 324

Etape 1	216	324
Etape 2	216	324 108648 541296 272592 135184 610368 320736 141472
Etape 3	~~216~~	~~324~~ ~~108648~~ ~~541296~~ 272592 135184 ~~610368~~ 320736 141472
Etape 4		**69984**

1. Jean-François BOULICAUT m'a demandé des références sur cet algorithme de multiplication. À ma grande honte, je n'ai pas retrouvé mes sources… Je n'affirmerais pas que le procédé ici décrit ait été utilisé en russie, mais pourquoi pas? Il s'agit d'une version de l'algorithme habituel, exécuté sur des nombres en base 2, même si leur représentation en base 10 est utilisée pour noter les résultats intermédiaires.

Il est important de comprendre qu'un algorithme n'est pas une recette magique: il doit reposer sur des principes qui permettent de prouver qu'il calcule bien son résultat. Dans le cas qui nous intéresse, le principe est simple:

$(2\ n)\ a = n\ (2a)$ (1)
$(2\ n+1)\ a = n\ (2a) + a$ (2)

L'égalité (1) permet de comprendre que lorsqu'on passe d'une ligne paire à la suivante on ne change pas le résultat. L'égalité (2) justifie, elle, la nécessité de sommer ce qui correspond aux lignes impaires.

On constate que les étapes élémentaires de cet algorithme de multiplication sont, outre les techniques d'addition, la multiplication et la division par 2.

Exemple 2
Une recette de cuisine répond à notre définition de ce qu'est un algorithme.

Exemple 3
Une partition musicale est aussi un exemple d'algorithme: c'est une description finie, réalisée dans un langage codé qui, exécutée par un orchestre, donne un résultat parfois agréable à écouter.

Exemple 4
Nous donnons une version caricaturale d'un algorithme visant à éviter le blocage des roues d'une automobile lors du freinage.

Figure 6 — Surveillance d'une roue pour un « freinage ABS »

```
Pour chaque roue,
    tant que la pédale de frein est sollicitée, répéter suffisamment rapidement
        si la roue est bloquée
            alors relâcher le piston de frein
            sinon enfoncer le piston de frein
```

1.2.3 Commentaires

La différence entre la description de l'enchaînement (algorithme) et l'enchaînement lui-même (processus de calcul) peut paraître subtile au néophyte. Elle est toutefois fondamentale. Pour mieux faire sentir la différence, nous ferons deux remarques.

1) La description « *avance d'un pas et recommence indéfiniment* » est une description finie qui produirait, sur une éventuelle machine l'enchaînement infini qui consiste à répéter la même opération élémentaire, « *avancer d'un pas* ». Dans le même ordre d'idées, la description finie « *nage tout droit jusqu'au bout de l'océan* » peut engendrer des enchaînements finis ou infinis.

2) Une machine, lorsqu'elle enchaîne des ordres élémentaires, utilise des données qu'elle acquiert, quand elle en a besoin, dans le monde extérieur. La rédaction de la description se fait dans l'ignorance de ces données. On

conçoit donc des algorithmes comme « *consulter le prix hors taxe, lui ajouter 20,6% et donner le prix TTC* » sans connaître les données sur lesquels ils s'appliqueront.

Du point de vue de la terminologie, on retiendra les termes suivants:

Définition 1.1 Algorithme

Un *algorithme* est une description finie d'un enchaînement d'étapes élémentaires capable de résoudre un problème donné.

Définition 1.2 Processus de calcul

Un *processus de calcul*, ou encore plus simplement un *calcul*, est une exécution d'un algorithme: c'est donc une réalisation particulière d'un enchaînement d'étapes élémentaires.

Un même algorithme peut donc donner lieu à plusieurs processus de calcul : c'est ce qui se passe à chaque fois qu'on utilise l'algorithme.

Définition 1.3 Statique

Le terme *statique* est utilisé pour qualifier les propriétés d'un algorithme indépendamment de toute exécution.

Définition 1.4 Dynamique

Le terme *dynamique* est utilisé pour qualifier les propriétés d'un calcul (c'est à dire les propriétés du comportement de l'algorithme pendant l'exécution).

1.2.4 Programmes et machines

Du cheminement qui, au 1.1, nous a conduit à la machine de VON NEUMANN, on peut retenir qu'une *machine* (ordinateur ou autre) offre un certain nombre de fonctions (commandes, ordres) élémentaires que nous appelons *opérations primitives*. Lorsqu'une machine réalise un calcul (une tâche) particulier, elle exécute un certain nombre d'opérations primitives. L'enchaînement de ces opérations est régi par un *programme*. Le programme est une description finie d'un enchaînement d'opérations primitives, exprimée dans un *langage* décodable par l'unité de contrôle de la machine.

On constate donc qu'un programme est un algorithme: description d'un enchaînement d'étapes élémentaires qui sont des opérations primitives de la machine.

Un programme est donc un cas particulier d'algorithme: ce qui le rend particulier, c'est d'une part le fait qu'il est exprimé dans un langage souvent

peu lisible et d'autre part l'existence d'une machine capable de décoder et d'interpréter ce langage.

Il est cependant clair qu'on peut exprimer des algorithmes sans faire référence à une machine particulière: on utilise alors un langage assez libre et on fait l'hypothèse implicite qu'on est capable de construire une machine capable d'exécuter les étapes élémentaires qu'on a utilisées.

A titre d'illustration, considérons le problème qui consiste à déterminer le pgcd de deux nombres a et b.

Algorithme A (ridicule)

Appliquer sur a et b l'opération pgcd

Algorithme B

Figure 7

Algorithme d'Euclide

1) Si a = b, le résultat est a.
2) Dans le cas contraire, remplaçons le plus grand des deux nombres a et b par
leur différence et recommençons.

L'algorithme A est ridicule dans la mesure où il fait l'hypothèse d'une machine capable de résoudre directement le problème posé: on n'a en aucune manière avancé vers une quelconque solution. En revanche, pour l'algorithme B (proposé par Euclide à une époque où on ne parlait pas d'ordinateurs) on fait l'hypothèse qu'on dispose d'une machine capable de faire des comparaisons, des soustractions… problèmes qui, eux, sont *plus simples* que le problème de départ.

Nous retrouverons, tout au long de ce cours, ce principe fondamental de la programmation: *résoudre un problème donné en composant des solutions à des sous-problèmes plus simples.*

Définition 1.5 Démarche descendante

Pour résoudre un problème, la *démarche descendante* consiste à:

- Décomposer le problème en sous-problèmes plus simples.
- Résoudre chacun des sous-problèmes.
- Combiner leurs solutions.

1.3 Langages de programmation

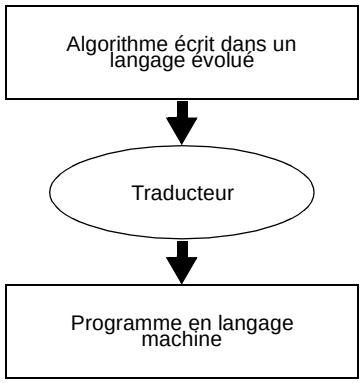
1.3.1 Généralités

Un algorithme, étant une *description*, est exprimé à l'aide d'un langage. Un programme est un algorithme écrit dans un langage particulier, décodable par l'unité de contrôle de la machine.

Nous ne décrivons pas plus avant, pour l'instant, cette notion; les calculateurs à programmes extérieurs sont ceux qui permettent mieux de l'appréhender: un piano mécanique, par exemple, interprète le contenu d'un support matériel (carton) qui comporte des perforations: le langage des « trous » est décodable par l'unité de contrôle de la machine.

Depuis la fin des années 50, on ne programme plus les ordinateurs en les alimentant, comme un piano mécanique, à l'aide d'un programme directement décodable par la machine. C'est un des apports les plus fertiles de la notion de programme enregistré: le programme étant dans la mémoire, on a pu concevoir des programmes appelés *compilateurs*, *traducteurs* (ou *interpréteurs*) qui

- lisent un algorithme écrit dans un langage plus compréhensible par un être humain; on appelle un tel langage un *langage évolué*,
- traduisent cet algorithme en un programme décodable par l'unité de contrôle de la machine et donc exécutable par cette même machine.



Les langages que nous avons qualifiés d'évolués ne le sont pas tant que ça... Les règles de la grammaire qui les définissent (ce qu'on appelle la *syntaxe* du langage) sont strictes. La signification de ce qu'on peut écrire avec de tels langages (ce qu'on appelle la *sémantique* du langage) est pauvre. En effet, conçus pour exprimer finalement des programmes, ils permettent d'exprimer deux types de choses:

- des opérations primitives à peine idéalisées susceptibles d'être fournies par tout ordinateur,
- des opérateurs d'enchaînement très stéréotypés.

Résumé de la terminologie

Définition 1.6 Machine

Une *machine* fournit:

- Un certain nombre d'opérations primitives
- Des possibilités d'enchaîner ces opérations primitives.

Définition 1.7 Programme

Un *programme* est un algorithme exprimé dans un langage susceptible d'être interprété par une machine.

Définition 1.8 Syntaxe d'un langage

La *syntaxe* d'un langage est un ensemble de règles caractérisant l'ensemble des programmes qu'on a le droit d'écrire.

Un programme syntaxiquement correct n'a pas forcément un sens et ne calcule pas obligatoirement ce que le programmeur souhaite (les programmeurs font des erreurs…). Un programme syntaxiquement incorrect n'est, tout simplement, pas un programme!

Définition 1.9 Sémantique

La *sémantique* d'un langage est un ensemble de propriétés décrivant la signification des programmes d'un langage donné. Par extension, la sémantique d'un programme décrit ce qu'il calcule.

Pour fixer les idées, nous donnons deux exemples:

1.3.2 Exemples

Nous donnons ci-après plusieurs versions de l'algorithme d'EUCLIDE exprimées dans divers langages de programmation.

Programme Pascal

Le langage Pascal, conçu par l'informaticien suisse Niklaus WIRTH, a été conçu dans les années 70.

Figure 8 Un programme Pascal

```
function Pgcd (a, b: integer) : integer;
begin
    if a < b
        then Pgcd:= Pgcd(a, b-a)
        else begin
            if a > b
                then
```

```
                    Pgcd:= Pgcd(a-b,b)
                else
                    Pgcd:= a
            end;
end;
```

Programme Java
Le langage Java fut conçu dans les années 1990 par James GOSLING, au sein de la société *Sun Microsystems* .

Figure 9 Un programme java

```
public static int pgcd (int a, int b) {
    if (a < b) return pgcd (a, b-a) ;
    else {
        if (a > b) return pgcd (a-b, b) ;
        else return a ;
    }
} // pgcd
```

Programme Scheme
Scheme est un dialecte de Lisp, langage conçu par John MCCARTHY dans les années 50.

Figure 10 Un programme Scheme

```
(define pgcd (lambda (a b)
    (cond
        ((< a b) (pgcd a (-b a)))
        ((> a b) (pgcd (-a b) b))
        (#t          a))))
```

1.4 Algorithmes et langages : éléments historiques

1.4.1 Algorithmes

L'historique des algorithmes remonte à la plus haute antiquité, et déborde largement du cadre que nous nous sommes fixé.

Nous nous contenterons de citer le fait que les plus anciens textes mathématiques connus sont des algorithmes (Mésopotamie): l'humanité s'est d'abord posé des problèmes concrets et a donc développé des solutions particulières pour les résoudre. Le développement (la percée) des mathématiques réalisée par les Grecs a fait que les chercheurs se sont plus consacrés à des généralisations et quoique l'histoire des mathématiques soit émaillée d'algorithmes (algorithme d'EUCLIDE – pgcd – algorithme d'ÉRATHOSTÈNE – primalité – techniques de résolution d'équations...) il a fallu attendre la fin

du XIX^{e} siècle pour que la notion même d'algorithme devienne un objet d'études.

A cette époque, en effet, les progrès des sciences ont engendré un certain optimisme. Les mathématiciens, en particulier, ont conçu l'idée que, sous réserve d'un formalisme ad hoc, on pouvait réussir à automatiser les processus de démonstration. En d'autres termes, on pensait que la réponse à la question fondamentale:

> Etant donné un problème, est-il toujours possible de trouver un algorithme pour le résoudre?

était positive. Au début du XXè siècle, HILBERT a proposé à la communauté un défi: établir formellement cette propriété.

Dans les années 30, deux mathématiciens, Alan TURING et Alonzo CHURCH, ont mis un point final, malheureusement *négatif*, à cette discussion: *il existe des problèmes pour lesquels il n'y a pas d'algorithme.*

Sans rentrer dans les détails, on peut justifier très informellement ce résultat.

Champ de l'étude

- Problèmes F = {fonctions de **N** dans **N**}
- Algorithmes A = {algorithmes qui, à partir d'un entier, fournissent un autre entier}

Bagage mathématique nécessaire

- ensembles dénombrables (qui peuvent être mis en bijection avec **N**)
- existence d'ensembles « plus grands » que les ensembles dénombrables. Exemple: il y a « plus » d'éléments dans **R** que dans **N**.

Principe de la démonstration

- Etant entendu que deux fonctions différentes ne peuvent être calculées par le même algorithme, on va montrer que l'ensemble F est plus grand que l'ensemble A.

Démonstration

Premier point

- A *est dénombrable.*

Justification informelle: un algorithme est un texte fini. On peut numéroter les textes (tous les textes d'une lettre en ordre alphabétique, puis tous les textes de 2 lettres...).

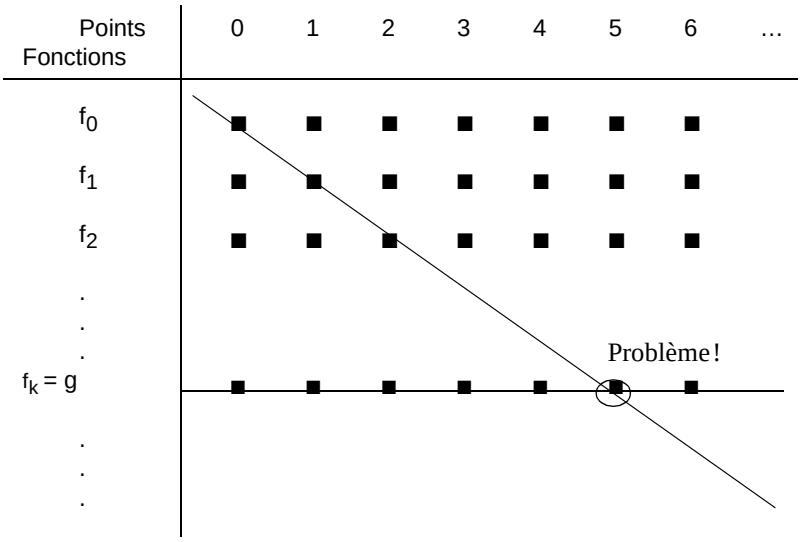
Deuxième point

- F *n'est pas dénombrable.*

La démonstration de cette propriété utilise un procédé élégant, celui de la diagonale de CANTOR:

Si F était dénombrable, on pourrait numéroter toutes les fonctions: f_0, f_1...

On peut représenter chaque fonction par la liste des valeurs qu'elle prend en 0, 1... On peut donc imaginer le tableau suivant :



Considérons maintenant la fonction g définie par $g(i) = f_i(i) + 1$. Comme g est une fonction de **N** dans **N**, elle doit être dans la liste et avoir un numéro, k. On a quelques problèmes pour connaître la valeur de g au point k ! Cette contradiction permet de conclure que F n'est pas dénombrable.

En conséquence : *Il y a des problèmes pour lesquels on ne peut pas fournir d'algorithmes.*

Depuis cette date, les mathématiciens et les informaticiens sont plus humbles. Au delà de cette anecdote, pour établir cette propriété, tant TURING que CHURCH ont proposé un modèle abstrait de machine universelle. Le modèle proposé par CHURCH, appelé λ-calcul, est à la base du langage que nous étudions dans le cadre de ce cours.

1.4.2 Langages de programmation

Les premiers ordinateurs ont été programmés directement en langage machine. À la fin des années 50 sont apparus les trois premiers langages évolués : Fortran, Cobol et Lisp.

Dès cette époque, on peut distinguer deux lignées évolutives :

- La première, que nous qualifions d'*impérative*, comprend des langages très calqués sur les structures des machines. De par cette propriété, ils sont souvent efficaces (en temps d'exécution des programmes qu'ils produisent) mais d'une technicité de programmation délicate, ce qui augmente le coût de production des programmes.
- La seconde, que nous qualifions de *déclarative*, comprend des langages qui reposent eux sur des concepts plus proches du raisonnement humain.

Les avantages sont duaux: parfois moins efficaces, ils réduisent considérablement les coûts de production.

Ces deux lignées se fertilisent mutuellement: une meilleure connaissance des schémas d'exécution a permis de rendre l'exécution des langages déclaratifs plus efficace et, réciproquement, les progrès des langages déclaratifs ont eu un impact sur les fonctionnalités offertes par les langages impératifs. Compte tenu de ces progrès, les différences se sont estompées. Toutefois, la dichotomie impératif / industrie *vs* déclaratif / recherche est encore d'actualité.

Nous donnons dans le tableau ci-après quelques langages célèbres[1]:

Figure 11 Quelques langages représentatifs

	Famille impérative	**Famille déclarative**
Années 50 (fin)	Fortran - Cobol	Lisp
60	Algol 60 - Basic	
70	Pascal	Prolog
80	C - Ada - Langages à objets (C++, Eiffel…)	Dérivés de Lisp et Prolog, ML
90	Ada 95, Java…	

Pour la famille déclarative, les deux langages cités, Lisp et Prolog, reposent respectivement sur les concepts de fonction et de relation. Dans le cadre de ce cours, nous étudierons un dérivé de Lisp: Scheme.

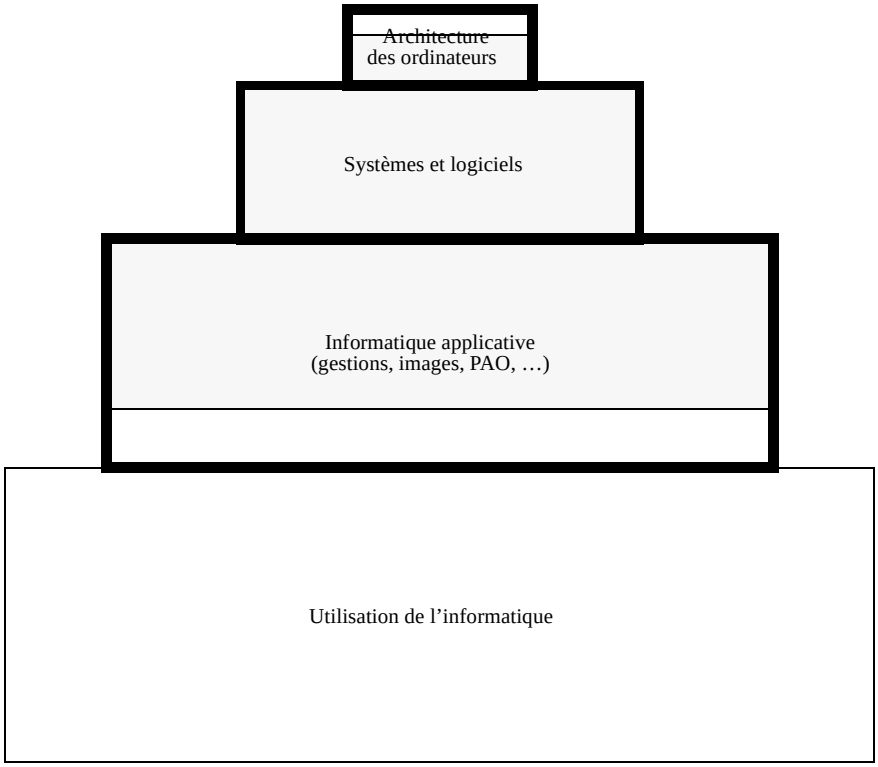
1.5 Qu'est-ce que l'informatique? Où sont les vrais problèmes?

L'informatique, au début des années 90, est un domaine vaste en explosion. Toutefois, la notion d'algorithme et son corollaire, la programmation est centrale[2].

1. D'autres langages sont conçus pour des utilisations spécifiques: langages des tableurs (excel), langages d'interfaçage (visual basic)…

2. Malgré les progrès réalisés en 10 ans, cette constatation reste d'actualité pour les années 2000.

Pour illustrer cette remarque, on peut faire le dessin sectoriel qui suit, dans lequel apparaissent en gris les domaines où algorithmique et programmation sont des notions centrales :



En conséquence, ce cours, qui est une initiation, sera consacré à l'*algorithmique* et la *programmation*.

Ceci étant, qu'est-ce que l'apprentissage de l'algorithmique et de la programmation ?

Dans les années 60, c'était presque uniquement l'apprentissage d'*un* langage de programmation (en général le plus répandu).

Depuis, l'informatique a explosé. Le secteur de la production de logiciel est devenu énorme (la France est le deuxième producteur mondial) et le coût de cette activité est devenu prohibitif. Les raisons de cette explosion des coûts sont multiples :

- La complexité est abominable (on écrit des programmes d'un million de lignes).
- La production en série ne s'applique pas vraiment, les procédés de production sont mal systématisés.
- La production de programme est une activité intellectuelle.

Ces trois considérations ont des conséquences pratiques dommageables:

- Les logiciels livrés comportent des erreurs! On imagine mal qu'un produit puisse être vendu avec des vices cachés; c'est pourtant le cas des logiciels...
- Les délais, et donc les coûts de production, sont particulièrement difficiles à évaluer.

Pour faire évoluer cette situation, le seul remède actuellement disponible consiste en l'emploi de *méthodes* (souvent contraignantes) de production.

Dans le cadre de ce cours, il n'est pas question d'appliquer les méthodes employées dans un contexte industriel car celles-ci sont adaptées et à la taille des produits et au contexte de production. Toutefois, nous développerons des éléments de méthode adaptés au contexte particulier de l'initiation.

Les buts du cours sont donc triples:

- Apprendre à concevoir des algorithmes.
- Apprendre à les exprimer à l'aide d'un langage de programmation.
- Apprendre des éléments méthodologiques visant à rationaliser le processus de production de programme.

1.6 Exercices

Ce chapitre n'est que très partiellement traité en cours. Sa principale raison d'être est de donner un certain nombre d'éléments de réflexion et de culture scientifique sur des objets très présents dans la vie courante — les algorithmes — mais paradoxalement très absents des programmes académiques.

Dans le même ordre d'idées, les exercices qui suivent ne sont pas traités en séance: ils ont pour but de favoriser une réflexion personnelle sur ce qu'est un algorithme et sur la nécessité d'élaborer un formalisme et des outils propres à en faire un objet d'étude.

1.6.1 De la difficulté de définir des opérations élémentaires

Exercice 1.1 Retrouver son chemin

1. Rédiger, en français, une note permettant à quelqu'un qui ne connaît pas la ville de se rendre en un lieu donné de Rennes.
2. Valider cette note avec un candide qui s'efforcera d'être le moins imaginatif possible, et répertorier toutes les erreurs, imprécisions et ambiguïtés.
3. Après corrections éventuelles, déterminer les connaissances indispensables pour utiliser cette note: latéralisation, lecture, code de la route, etc.

1.6.2 De la difficulté d'exprimer des algorithmes

Exercice 1.2 Nœud de chaise

Expliquer, par écrit et sans faire de dessin, comment faire un nœud de chaise.

Exercice 1.3 Addition et multiplication

Rédiger, en français, les algorithmes d'addition et de multiplication vus à l'école primaire.

Exercice 1.4 Test de primalité

1. Proposer un algorithme permettant de déterminer si un nombre est premier.
2. Proposer un algorithme permettant d'établir la liste des n premiers nombres premiers.
3. Chercher dans une bibliothèque ce qu'on appelle le crible d'ERATOSTHÈNE. Dire pourquoi on arrête la recherche à $\sqrt{n}$.

Exercice 1.5 Algorithme babylonien[1]

Un texte mathématique babylonien propose l'algorithme suivant:

« J'ai additionné la surface et mon côté de carré: 45. Tu poseras 1, la wāsitum . Tu fractionneras la moitié de 1 (: 30). Tu multiplieras 30 et 30 (: 15). Tu ajouteras 15 à 45: 1. 1 (en) est la racine carrée. Tu soustrairas le 30, que tu as multiplié, de 1 (: 30). 30 est le côté de carré ».

1. En tenant compte du fait que les babyloniens utilisaient souvent la base 60, essayer de comprendre cet algorithme.

Outre le problème posé par le système de numération[2], la formulation précédente recèle une difficulté paradoxale: l'algorithme est exprimé sur un exemple et non dans sa généralité. Si on fait cette abstraction, on obtient une formulation plus lisible:

« La donnée représente la somme de la surface et du côté d'un carré. Pour trouver la valeur du côté, il suffit d'effectuer les étapes suivantes:

1) On prend 1.
2) On divise par 2.
3) On élève au carré.
4) On ajoute la donnée.
5) On prend la racine carrée.
6) On soustrait le résultat de 2).

Le résultat cherché est la valeur obtenue à l'issue de l'étape 6 ».

1. Le texte de l'algorithme est emprunté aux *Éléments d'histoire des sciences*, Michel SERRES, Bordas (1989).

2. Nous conseillons vivement au lecteur de consulter l'*Histoire universelle des chiffres*, Georges IFRAH, Laffont, Collection Bouquins (1994).

2. Justifier cet algorithme. Le formuler en termes plus modernes.

Exercice 1.6 Tri d'un paquet de fiches

On dispose d'un paquet de fiches nominatives. Donner un algorithme pour qu'un individu peu imaginatif mais très méthodique puisse les classer dans l'ordre alphabétique.

1.6.3 De la complexité des algorithmes

Exercice 1.7 Coût de différents algorithmes de tri

On cherche à trier un paquet de fiches nominatives et on envisage deux algorithmes.

Le premier, dit *tri par extraction*, consiste à:

1) Parcourir le paquet pour déterminer la première fiche;
2) Mettre cette fiche à part;
3) Recommencer la même opération sur le paquet restant, jusqu'à ce que toutes les fiches aient été classées.

Le second, dit *tri fusion*, consiste à:

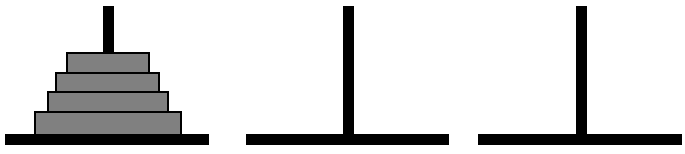
1) Séparer le paquet en deux;
2) Trier séparément, et en appliquant le même algorithme, chaque demi-paquet;
3) Fusionner les deux paquets résultant de l'étape précédente.

On considère que, pour comparer deux fiches, l'opérateur met une seconde et que les autres opérations prennent un temps négligeable.

- Evaluer sommairement, pour chacun des deux algorithmes, le temps mis pour trier 30 fiches, 1 000 fiches, 1 000 000 de fiches.

Exercice 1.8 Tours de Hanoï

Etant donné trois plots et une pile de n disques, le problème dit des tours de Hanoï consiste à déplacer la pile du plot 1 au plot 3 en transférant un seul disque à la fois d'un plot à un autre. On ne peut bouger un disque que s'il est situé sur le sommet d'une pile et on ne peut poser un disque que sur un plot vide ou sur un disque de taille supérieure.



1. Formuler un algorithme pour résoudre le problème général: *déplacer une pile de n disques du plot i au plot j*. Pour ce faire on utilisera une technique déjà utilisée dans l'exercice précédent, qui consiste à supposer qu'on est capable, en utilisant le même algorithme, de résoudre un problème plus simple: *déplacer une pile de n-1 disques d'un plot k au plot p*.

2. En supposant qu'il faut 1 s pour déplacer un disque, évaluer le temps nécessaire pour déplacer une pile de 3 disques, 10 disques, 100 disques.

1.6.4 Des problèmes sans solution

Exercice 1.9 Non dénombrabilité de R

On se propose de montrer que le sous-ensemble [0, 1[de **R** n'est pas dénombrable (ne peut pas être mis en bijection avec **N**). Pour ce faire, on peut représenter les éléments de [0, 1[par leur développement décimal. Si on suppose que [0, 1[est dénombrable, on peut en numéroter les éléments et donc faire un tableau comme celui qui a été défini pour les fonctions de **N** dans **N**.

- Utiliser le procédé de CANTOR pour démontrer que **R** n'est pas dénombrable.

Exercice 1.10 Un problème sans solution

On a montré en cours qu'il existait des problèmes sans algorithme. Il est toutefois plus difficile d'exhiber de tels problèmes. Le problème dit de l'arrêt en est un exemple.

1) Les algorithmes que nous considérons sont ceux qui admettent en entrée un ou plusieurs entiers et qui produisent un entier en résultat. Nous ne formaliserons pas le langage d'expression de ces algorithmes. Les trois algorithmes qui suivent pourraient être écrits dans n'importe quel langage de programmation.

```
PLUS (n):
        Entrer une valeur pour n;
        Lui ajouter 1;
        Sortir le résultat
BOUCLE(n):
        Entrer une valeur pour n;
        Lui ajouter 1 et recommencer (cette étape);
        Sortir le résultat
GRIMPER(n):
        Entrer une valeur pour n;
        Partir de 5, ajouter 1 et recommencer jusqu'à ce qu'on obtienne
        une valeur égale à n;
        Sortir le résultat
```

2) Un algorithme est un texte rédigé à l'aide d'un alphabet. Comme un texte est une notion difficile à manipuler il est commode de disposer d'une représentation de tout texte par un nombre[1] (objet plus facile à traiter). Pour sim-

1. Le procédé de numérotation des algorithmes que nous décrivons ici (considérer le texte comme un nombre en base n) est fruste et demanderait à être précisé pour traiter le problème du chiffre 0. D'ordinaire on adopte un autre procédé, dit numérotation de GÖDEL; on représente un texte T (en utilisant la suite infinie des nombres premiers) par $2^{n1} \times 3^{n2} \times 5^{n3} \times \ldots$ où n_i est le numéro du $i^{ème}$ caractère du texte. L'unicité de la décomposition en facteurs premiers permet d'assurer la correspondance inverse.

plifier, on considère que l'alphabet est fini et on numérote ses caractères de 1 à n-1. À tout algorithme A on peut associer un entier *N(*A*)*, en prenant par exemple le nombre en base n obtenu en remplaçant chaque caractère du texte par le chiffre de la base n correspondant à son numéro. Réciproquement, à un entier i, on peut associer le texte (l'algorithme) *A(i)*.

3) Il existe des algorithmes, comme PLUS qui se terminent toujours. D'autres, par contre, ne se terminent jamais; c'est le cas de BOUCLE. Enfin, certains algorithmes, peuvent ou non s'arrêter, selon la donnée qui leur est fournie. Si on fournit la donnée 6 à GRIMPER, il se termine en produisant la valeur 6. Si on lui fournit 2, il ne se terminera jamais.

4) On s'intéresse à la possibilité de rédiger un algorithme TA (Test Arrêt) qui, prenant en entrée un entier a et un entier n produit 1 si l'algorithme *A(*a*)* s'arrête lorsqu'on lui fournit la donnée n et qui produit 0 dans le cas contraire.

S'il était possible de concevoir un algorithme TA, l'exécution TA(*N*(GRIMPER), 2) délivrerait 0 alors que TA(*N*(GRIMPER), 6) délivrerait 1.

On va montrer qu'*il est impossible d'écrire un algorithme comme* TA.

5) Si TA existe, l'algorithme TB suivant aussi:

```
TB(a):
    Entrer une valeur pour a;
    Exécuter TA en lui fournissant a et a;
    Si le résultat est 1, boucler;
    sortir le résultat
```

1. Que se passe-t-il si on exécute TB en lui donnant en donnée *N(*TB*)*? Que peut-on en conclure pour TA?

2. Rapprocher ce procédé de démonstration de l'argument diagonal de CANTOR.

CHAPITRE 2

Programmation fonctionnelle

2.1 Langages de programmation et fonctions

On rappelle que:

- Une machine fournit un certain nombre d'opérations primitives.
- Un algorithme, pour résoudre un problème, est une description d'une manière d'enchaîner des opérations primitives de la machine qui, exécutée par cette machine, délivre une solution.

Pour exprimer un algorithme, on utilise un langage et par conséquent, tout langage fournit:

- des *expressions primitives* qui correspondent aux opérations de la machine,
- des moyens d'exprimer des enchaînements d'opérations: ce sont des *outils de composition* des expressions du langage, qu'elles soient primitives ou déjà composées.

Par ailleurs, nous avons souligné l'intérêt de la démarche descendante; cette démarche est facilitée si on dispose du moyen de nommer (on dit souvent abstraire) des expressions du langage afin de les manipuler comme si elles étaient des opérations élémentaires.

Définition 2.1 Fonctionnalités offertes par un langage de programmation

Un *langage de programmation* fournit les trois fonctionnalités suivantes:

- Des expressions primitives.
- Des outils de composition d'expressions.
- Des outils de désignation qui permettent de nommer des expressions du langage.

Les expressions du langage peuvent être plus ou moins barbares mais, dans la première moitié du XX^{è} siècle, un mathématicien, Alonzo CHURCH, a suggéré que la notion de fonction (comme expression de base d'un langage) avec l'application de fonction comme moyen de composition permettait d'exprimer n'importe quel algorithme.

Exemple
On dispose d'une machine munie d'une opération primitive qui à partir de deux entiers fournit la valeur du plus petit. On peut décrire cette opération par une fonction que nous nommons Min:

Min:	entier x entier	→	entier
	x, y	→	le plus petit des deux nombres x et y.

Problème
On dispose de trois nombres et on veut connaître le plus petit des trois. On cherche donc à écrire une fonction:

MinDe3:	entier x entier x entier	→	entier
	x, y, z	→	le plus petit des 3 nombres x, y, z

Algorithme
On peut définir la fonction MinDe3 en posant:

MinDe3(x, y, z) = Min(Min(x, y), z)

Les langages de programmation qui reposent sur l'usage des fonctions comme concept central sont appelés *langages fonctionnels*. Le grand ancêtre est le langage Lisp (conçu par MCCARTHY à la fin des années 50). Le langage utilisé dans ce cours, Scheme, est un dérivé de Lisp. En Scheme, la fonction MinDe3 s'exprime de la manière suivante:

```
; MinDe3 :  entier x entier x entier      →   entier
;           x, y, z                       →   le plus petit des 3 nombres x, y, z
(define MinDe3 (lambda (x y z)
      (Min (Min x y) z)))
```

Dans ce chapitre, nous présentons le langage Scheme dans sa quasi-totalité.

2.2 Objets primitifs et types

Un programme Scheme est appelé *expression* Scheme. Nous découvrons, au fur et à mesure, les diverses catégories d'expressions Scheme. Pour chacune d'entre elles, il convient d'en donner la syntaxe (la forme) et la sémantique (ce qu'elle est censée faire). Pour la sémantique, on retient une caractéristique importante du langage: l'exécution – dans la terminologie fonctionnelle on dit *évaluation* – de toute expression Scheme délivre une valeur (a un résultat). Une manière de donner la sémantique d'une expression Scheme consiste justement à décrire quel est le résultat de son évaluation.

Les expressions Scheme manipulent des objets, qu'ils soient primitifs (directement fournis par le langage) ou construits (par le programmeur). Nous nous intéressons, dans un premier temps, aux objets primitifs.

2.2.1 Types primitifs et constantes

En informatique, un *type*[1] caractérise un ensemble d'objets et les opérations primitives qui leur sont applicables. Scheme utilise un certain nombre de types primitifs: ainsi, le type naturel caractérise l'ensemble des entiers positifs ou nuls[2] sur lesquels on dispose des opérations usuelles (+, x…). On dispose d'autres types pour les nombres:

- entier[3] (*integer*) pour l'ensemble des entiers relatifs.
- réel (*real*) pour l'ensemble des réels positifs et négatifs manipulés par la machine.
- nombre (*number*) pour l'ensemble entier ∪ réel.

Un type joue un rôle fondamental en informatique: le type booléen qui caractérise l'ensemble à deux valeurs {vrai, faux}: les opérations les plus fréquentes sur les booléens sont décrites au § 2.4.1.1.

Pour ce qui concerne les objets primitifs, tout langage offre une manière d'utiliser directement certaines valeurs: on parle de *notations de constantes*. En Scheme:

- l'expression 123 est la notation d'une constante dont la valeur est l'entier naturel qui correspond à la notation décimale 123;
- l'expression 2.7 est la notation d'une constante dont la valeur est le réel qui correspond à la notation décimale 2,7[4];
- Les expressions #t et #f sont deux notations de constantes dont les valeurs sont respectivement les booléens vrai et faux.

On constate qu'un même objet peut relever de plusieurs types: la constante 12 est un naturel, un entier et un nombre. En revanche, 12 n'est ni un booléen, ni un réel.

1. La notion de type est fondamentale en programmation. Toutefois, c'est une notion complexe à appréhender et le langage qui sert de support à ce cours, Scheme, se prête mal à la présentation rigoureuse d'un système cohérent. Confrontés à ce problème – une notion importante à présenter à l'aide d'un outil insuffisant – nous avons adopté un « profil bas »: l'étudiant doit impérativement retenir l'obligation d'annoncer clairement et avec un maximum de précision le type des objets qu'il manipule. L'apprentissage ultérieur de langages fortement typés permettra de combler cette lacune importante.

2. Dans la version du langage Scheme utilisée, on manipule des entiers dont la représentation occupe une taille variable. On est donc capable de représenter de très grands entiers, la limite étant évidemment fonction de la mémoire disponible, ce qui nous autorise à parler de l'ensemble des entiers.

3. On remarquera qu'en informatique, lorsqu'on parle d'entiers (en anglais *integer*) il s'agit des entiers relatifs.

4. Les anglo-saxons utilisent des conventions qui ne sont pas les nôtres: le point (.) sépare la partie entière de la partie décimale alors que la virgule (,) sert souvent à séparer les grands nombres par paquets de 3 chiffres. Leur position dominante en informatique fait que dans les langages de programmation c'est le point décimal qui est utilisé.

Conventions

- Souvent, l'ensemble caractérisé par un type a un élément qui joue de manière évidente un rôle particulier (souvent l'élément neutre d'une opération fondamentale pour les objets du type). On utilisera, quand le contexte s'y prête sans ambiguïté, le suffixe + pour qualifier le type qui caractérise l'ensemble privé de l'élément singulier. Ainsi, naturel+ = naturel - {0}
- Le type indifférent caractérise l'ensemble de tous les objets Scheme.

Définition 2.2 Expression Scheme: notation de constante

Syntaxe

Une *notation de constante* Scheme est une expression Scheme. Les principales notations de constantes sont:

- Constantes entières: suite de chiffres décimaux
- Constantes réelles: deux suites de chiffres décimaux séparées par un point
- Constantes booléennes: #t ou #f

Sémantique

- Le résultat de l'évaluation d'une notation de constante est la valeur de la constante qu'elle représente.

2.2.2 Objets et désignation

Scheme étant un langage dit fonctionnel il doit permettre de manipuler des fonctions. Nous voyons au § 2.3.1 comment le programmeur peut créer des fonctions, en combinant des fonctions existantes. Il est toutefois nécessaire de disposer de points de départ et Scheme fournit un certain nombre de fonctions prédéfinies.

Ces dernières ont des noms (ou *identificateurs*) qui permettent aux programmeurs de les utiliser. Un identificateur, dans Scheme, est une suite quelconque de caractères sans espaces (quelconque signifie ici « à l'exception des notations de constantes »).

On retient, et ceci est vrai dans tout langage de programmation, que le rôle d'un identificateur est de *désigner* un objet. L'établissement des relations de

désignation entre identificateurs et objets est une des fonctionnalités fondamentales des langages de programmation.

Définition 2.3 Identificateur et désignation

En informatique, un *identificateur* est un nom dont le rôle est de désigner un objet.

- Dans la plupart des langages un identificateur commence par une lettre et peut être suivi de lettres, de chiffres et d'un nombre restreint de caractères spéciaux comme « _ » ou « - ».
- En Scheme un identificateur est une suite « presque » quelconque[a] de caractères.

a. Un identificateur Scheme doit pouvoir être distingué des autres constructions du langage; cela implique quelques restrictions évidentes à l'usage

Ainsi, dans Scheme, l'identificateur + désigne une fonction dont le type est, en réalité, un peu compliqué. Nous admettrons pour l'instant que son type est nombre x nombre $\rightarrow$ nombre. On dispose bien sûr des fonctions arithmétiques habituelles.

Définition 2.4 Expression Scheme: identificateur

Syntaxe

- *Une notation de constante est une expression Scheme.*
- Un *identificateur* est une expression Scheme.

Sémantique

- Le résultat de l'évaluation d'un identificateur est la valeur de l'objet qu'il désigne.

2.2.3 Appel de fonction

Donnons tout d'abord un minimum de terminologie sur les fonctions. Une caractéristique essentielle d'une fonction est la donnée de son *ensemble de départ* et de son *ensemble d'arrivée*. Cela définit ce qu'on appelle le *type* de la fonction. Si l'ensemble de départ d'une fonction Min est entier x entier et son ensemble d'arrivée entier, nous dirons que son type est entier x entier $\rightarrow$ entier. Dans le cas général, l'ensemble de départ est un produit cartésien et on appelle *arité* la dimension de ce produit cartésien. La fonction Min est d'arité 2; on dira qu'elle prend deux *paramètres* (ou arguments) et qu'elle délivre un *résultat*.

Définition 2.5 Type d'une fonction

Le type d'une fonction est la donnée du type de ses paramètres (son ensemble de départ, en général un produit cartésien) et du type de son résultat (son ensemble d'arrivée). Il se note par:

$$D_1 \times \ldots \times D_n \rightarrow A$$

Un *appel* d'une fonction d'arité n désignée par l'identificateur nom est une expression Scheme qui a la forme générale suivante:

(nom <expression$_1$>... <expression$_n$>)

où les <expression$_i$> sont appelées *paramètres effectifs* (en anglais *actual parameters*) de la fonction.

Le résultat d'un appel est le résultat de l'application de la fonction désignée aux valeurs délivrées par l'évaluation des paramètres qui sont fournis. Ainsi, (+ 1 2) délivre la valeur 3.

Définition 2.6 Expression Scheme: appel

Syntaxe

- *Une notation de constante est une expression Scheme.*
- *Un identificateur est une expression Scheme.*
- Un *appel* est une expression Scheme de la forme (nom <expression$_1$>... <expression$_n$>).

Sémantique

- Le résultat de l'évaluation d'un appel est le résultat de l'application de la fonction désignée par nom aux valeurs délivrées par l'évaluation des <expression$_i$>.

2.2.4 Contrôle statique de type

Lorsqu'on rédige un appel de fonction, il se peut que le type des paramètres effectifs ne correspondent pas au type de la fonction: on dit que l'expression est *mal typée.* Ainsi, si on sait que:

- le type de + est nombre x nombre $\rightarrow$ nombre
- a désigne un naturel
- b un booléen
- non désigne une fonction dont le type est booléen $\rightarrow$ booléen

on peut effectuer ce qu'on appelle des contrôles statiques de types (ces contrôles sont dits statiques parcequ'ils sont réalisés avant l'exécution).

Exemples

Expression	Résultat du contrôle de type	Explications
(+ 1 (+ a 12))	bien typé	• L'appel interne est bien typé: le nombre de paramètres est correct; a étant un naturel est également un nombre; 12 aussi • L'appel externe également: deux paramètres qui sont des nombres.
(+ b 1)	mal typé	• Un paramètre booléen alors qu'un nombre est requis
(1 2 3)	mal typé	• 1 n'est pas une fonction
(non #f)	bien typé	• Bon nombre de paramètres • Type du paramètre correct
(non b a)	mal typé	• Deux paramètres alors que non n'en suppose qu'un

Ce type de contrôle pourrait être pris en charge par le langage. En Scheme ce n'est pas le cas. Par conséquent le programmeur doit systématiquement faire lui-même, et avant toute exécution, le contrôle statique de type. Il est ridicule, et répréhensible, d'exécuter une expression mal typée.

Il faut cependant être conscient que si on confie à la machine Scheme une expression mal typée, elle essaye quand même de l'exécuter. Au cours de l'exécution, certains contrôles de type sont effectués (nombre de paramètres des fonctions par exemple) mais pas tous. Le résultat, qui n'a pas (et ne peut pas avoir) de sens est sans intérêt.

2.2.5 Premières exécutions

2.2.5.1 Dialogue avec Scheme

Scheme est ce qu'on appelle un interpréteur: c'est un programme qui dialogue avec le programmeur. Ce dialogue prend la forme suivante:

- L'interpréteur affiche une séquence de caractères appelée *invite* (en anglais *prompt*).
- Le programmeur frappe alors une expression du langage.
- Cette expression est évaluée par l'interpréteur qui affiche le résultat de l'évaluation ou un message d'erreur.
- Une nouvelle invite permet d'itérer le processus.

2.2.5.2 Evaluation des expressions

1. Le résultat de l'évaluation d'un identificateur est la valeur de l'objet qu'il désigne.

[1] 12
12
[2] +
#<procedure +>
[3] a
[VM ERROR encountered!] Variable not defined in current environment
a

Commentaires
La valeur associée à l'identificateur + est une fonction (nous retiendrons le fait que les mots procédure et fonction ont, dans le cadre de ce cours, le même sens). C'est une des fonctions primitives fournies par l'interpréteur Scheme. En [3], l'interpréteur signale qu'il n'y a pas de valeur associée à l'identificateur a; après avoir signalé l'erreur, il démarre un dialogue (invite [Inspect]) qui, dans le cas général, doit permettre au programmeur d'analyser les causes de l'erreur. On sort de ce dialogue en répondant CTRL q (touches Control et q).

2. Le résultat d'un appel est le résultat de l'application de la fonction désignée aux paramètres qui sont fournis.

[4] (+ 12 1)
13
[5] (+ (+ 12 1) (+ 12 1))
26

Commentaires
En [5], on remarque que les paramètres fournis peuvent être le résultat d'appels de fonctions. On dit que la règle d'évaluation est *récursive*: pour évaluer un appel, on évalue d'abord les paramètres, qui sont eux mêmes des appels auxquels on applique la même stratégie. Toute (bonne) récursivité doit pouvoir se terminer; la fin du processus intervient lorsqu'on cherche à évaluer des expressions qui désignent des valeurs.

2.2.6 Forme normale de BACKUS et NAUR

On constate que la définition des expressions Scheme, en augmentant, devient quelque peu verbeuse. Les informaticiens utilisent souvent pour décrire la syntaxe d'un langage un formalisme appelé BNF (BAKKUS-NAUR Form). Avec ce formalisme on peut résumer notre connaissance des expressions Scheme de la manière suivante:

<expression>	::=	<notation> | <identificateur> | <appel>
<notation>	::=	**constantes Scheme**
<identificateur>	::=	**identificateurs Scheme**
<appel>	::=	**(**<identificateur> <suite_de_paramètres>**)**
<suite_de_paramètres>	::=	**rien** | <expression> <suite_de_paramètres>

Ce formalisme utilise des règles (qui sont de même nature que les règles de grammaire du français) composées d'une partie gauche, de « ::= » et d'une partie droite. Le symbole « ::= » se lit « peut être », le symbole « | » se lit « ou ». Une partie droite comme **(**<identificateur> <suite_de_paramètres>**)** se lit comme: « **(** » suivi d'un <identificateur>, suivi d'une <suite_de_paramètres>, suivi de « **)** ».

2.2.7 Etablissement de la relation de désignation

Dans Scheme on utilise, pour établir une relation de désignation entre un identificateur et un objet, un opérateur primitif: l'opérateur define.

Définition 2.7 Expression Scheme: définition

Syntaxe

<expression> ::= <*notation*> | <*identificateur*> | <*appel*> | <définition>

…

<définition> ::= **(define** <identificateur> <expression> **)**

Sémantique

- La valeur d'une définition est l'identificateur <identificateur>.
- L'identificateur <identificateur> désigne le résultat de l'évaluation de <expression>

Remarques

- Il est déconseillé d'utiliser les définitions comme sous-expressions d'une expression Scheme
- Le type d'un identificateur est le type atome.

Du point de vue externe, la syntaxe de define est donc homogène avec un appel de fonction.

Exemples

[6] (define un 1)
un
[7] un
1
[8] (+ un 1)
2
[9] (define deux (+ un un))
deux
[10] deux
2
[11] (un)
[VM ERROR encountered!] Attempt to call a non-procedural object
(1)
…
[13] (define un 6)
un

[14] un
6
[15] deux
2

On remarque en [11] un exemple de contrôle de type fait, à l'exécution, par la machine Scheme. Un programmeur consciencieux n'aurait jamais du essayer d'exécuter une telle expression.

2.3 Création de fonctions

2.3.1 Expression d'une fonction

Considérons la fonction Carré ainsi définie:

Carré:	nombre	→	nombre
	x	→	x multiplié par x

Avec ce que nous avons vu du langage, nous sommes capables, en utilisant la fonction primitive de multiplication (qui est fournie par Scheme et qui est désignée par l'identificateur *), de calculer le carré de n'importe quel nombre. Pour définir la fonction Carré, il faut un peu plus: la possibilité de dire « j'appelle x un nombre quelconque et je veux évaluer (* x x) ». Pour ce faire, on dispose d'un opérateur primitif, lambda.

L'expression (lambda (x) (* x x)) délivre en résultat une fonction qui, à un nombre, est capable d'associer son carré; le type de cette fonction est donc nombre → nombre. On peut lire une telle expression ainsi:

(lambda	**(x)**	**(* x x)**	**)**
On construit une fonction	Cette fonction a un paramètre. On l'appelle x	Le résultat de cette fonction est obtenu en multipliant x par lui-même	

Du point de vue de la terminologie, on dit que x est un *paramètre formel*, et que l'expression (* x x) est le *corps* de la fonction.

Pour évaluer une fonction, il faut lui fournir une(des) valeur(s) pour son(ses) paramètre(s) (on rappelle que les valeurs ainsi fournies sont appelées *paramètres effectifs*). Nous étendons la définition de l'appel de fonction:

l'expression ((lambda (x) (* x x)) 2) est un appel de fonction et délivre la valeur 4.

Définition 2.8 Expression Scheme: fonction

Syntaxe

<expression> ::= <*notation*> | <*identificateur*> | <appel> | <*définition*> | <fonction>

…

<appel> ::= **(**<expression> <suite_de_pe>**)**

<suite_de_pe> ::= **rien** | <expression> <suite_de_pe>

<fonction> ::= **(lambda (** <suite_de_pf> **)** <expression> **)**

<suite_de_pf> ::= **rien** | <identificateur> <suite_de_pf>

- La première expression d'un appel doit délivrer une fonction

Sémantique

- Le résultat d'une <fonction> est un objet qui est une fonction.

Définition 2.9 Paramètres formels et paramètres effectifs

- Les identificateurs de la partie <suite_de_pf> d'une fonction sont appelés paramètres formels
- Les valeurs fournies à l'appel sont appelées paramètres effectifs.

Une fonction comme (lambda (x) (* x x)) est ce qu'on appelle une fonction *anonyme* (on ne lui a pas encore donné de nom). Il est toutefois possible de combiner désignation et expressions de fonction. Par exemple:

```
[16] ((lambda (x) (* x x)) 2)
4
[17] (define Carré (lambda (x) (* x x)))
Carré
[18] (Carré 2)
4
[19] (define
        SommeDeCarrés
        (lambda (x y)
            (+ (carré x) (carré y))))
SommeDeCarrés
[20] (SommeDeCarrés 3 4)
25
```

Il faut toutefois être soigneux: il est clair que tout programmeur fait des erreurs et il importe d'être capable de les analyser. On réfléchira aux causes des erreurs (qui auraient pu être repérées par un contrôle de type) de la séquence suivante:

```
[21] (Carré cote)
[VM ERROR encountered!] Variable not defined in current environment
COTE
[22] (Carré (un))
```

[VM ERROR encountered!] Attempt to call a non-procedural object (1)
[23] Carré
#<PROCEDURE CARRE>
[24] (Carré 3 4)
[VM ERROR encountered!] Invalid argument count: Function expected 1 argument(s) but was called with 2 as follows:
(#<PROCEDURE CARRE> 3 4)
[25] (SommeDeCarrés 3)
[VM ERROR encountered!] Invalid argument count: Function expected 2 argument(s) but was called with1 as follows:
(#<PROCEDURE SOMMEDECARRES> 3)

2.3.2 Modèle de substitution pour appliquer une fonction

Pour évaluer un appel de fonction, Scheme fait des choses assez compliquées dont on peut donner des modèles plus simples, qui, toutefois, ne décrivent pas tous les cas de figure. Nous donnons ici un premier modèle, le *modèle d'évaluation en ordre applicatif*, qui décrit assez bien ce que nous avons vu jusqu'à présent.

Modèle d'évaluation en ordre applicatif
Pour évaluer un appel de fonction, on évalue d'abord ses paramètres, puis on évalue ensuite le corps de la fonction en remplaçant toute occurrence de paramètre formel par la valeur calculée pour le paramètre effectif correspondant.

Ce modèle est récursif et la récursivité s'arrête lorsqu'on obtient une valeur ou lorsqu'on arrive sur une des procédures primitives. Par exemple:

(SommeDeCarrés un 2)
(SommeDeCarrés 1 2)
(+ (Carré 1) (Carré 2))
(+ (* 1 1) (Carré 2))
(+ 1 (Carré 2))
(+ 1 (* 2 2))
(+ 1 4)
5

2.4 Conditionnelles et prédicats

2.4.1 Étude par cas

Nous savons définir des fonctions pour lesquelles on dispose d'une expression analytique homogène. Ce n'est pas toujours le cas comme en témoigne la fonction suivante:

Abs:	entier	→	naturel
	x	→	| x |

| x | est défini par:

$$|x| = \begin{cases} x \textbf{ si } x \geq 0 \\ 0 - x \textbf{ si } x < 0 \end{cases}$$

Le domaine de définition de la fonction Abs est divisé en deux et pour chacun des cas, on dispose d'une expression simple. Ce type d'analyse d'un problème (partition du domaine de définition et traitement de chaque sous-domaine comme un sous-problème distinct) est fréquent en informatique. On dit qu'on a réalisé une *étude par cas*.

Pour traduire le résultat d'une étude par cas on a besoin d'un nouveau type d'expression Scheme, la *conditionnelle*.

La forme syntaxique générale de cette expression est la suivante:

(cond
    (<p_1> <e_1>)
    …
    (<p_n> <e_n>))

La fonction Abs peut être programmée comme suit:

```
; Abs:      entier          →     naturel
;           x               →     | x |
(define Abs (lambda (x)
      (cond
            ((>= x 0)       x)
            ((<  x 0)       (- 0 x))))
```

Remarques

- Les deux premières lignes commencent par le symbole ; (point virgule). Ces deux lignes sont des *commentaires* qui sont ignorés par la machine Scheme mais dont le rôle est crucial pour la lisibilité du programme. Dans le cadre de ce cours, nous nous imposerons d'associer à chaque fonction un commentaire précisant son type et la propriété définissant son résultat. Notons que ce commentaire est très utile pour effectuer un contrôle statique de type.
- >= et < sont des fonctions primitives de Scheme. On peut voir leur type ainsi:

  ```
  >=:        nombre x nombre    →    booléen
  ```

  Rappelons que les valeurs vrai et faux introduites au § 2.2.1 sont appelées valeurs booléennes ou tout simplement booléens.

Sémantique

Informellement, lors de l'évaluation de l'expression cond dans Abs, on évalue d'abord (>= x 0); si le résultat est vrai alors on évalue ce qui est associé à (>= x 0), c'est-à-dire x. Si, par contre, le résultat est faux, on s'intéresse à la suite et on évalue alors (< x 0)…

Avant de décrire la sémantique générale d'une expression cond, il nous faut préciser la notion de valeur booléenne.

2.4.1.1 Booléens

Les trois fonctions de base du calcul booléen sont le *et* ($\wedge$), le *ou* ($\vee$) et le *non* ($\neg$). Ces fonctions sont définies ainsi:

p	**¬ p**
vrai	faux
faux	vrai

p	**q**	**p ∨ q**
vrai	vrai	vrai
vrai	faux	vrai
faux	vrai	vrai
faux	faux	faux

p	**q**	**p ∧ q**
vrai	vrai	vrai
vrai	faux	faux
faux	vrai	faux
faux	faux	faux

On remarque que le ou et le et sont commutatifs et associatifs et on peut établir certaines identités remarquables, comme:

- $\neg\neg a = a$
- $\neg (a \vee b) = \neg a \wedge \neg b$
- $\neg (a \wedge b) = \neg a \vee \neg b$
- $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
- $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

En Scheme les deux constantes booléennes se notent #t et #f et le langage offre un certain nombre de fonctions dont le résultat est une valeur booléenne. Par exemple:

>=:	nombre x nombre	→	booléen
integer?[1]:	indifférent[2]	→	booléen

Les trois fonctions de base sur les booléens portent leurs noms anglo-saxons: and, or et not.

and:	booléen x booléen	→	booléen
	a, b	→	$a \wedge b$
or:	booléen x booléen	→	booléen
	a, b	→	$a \vee b$
not:	booléen	→	booléen
	a	→	$\neg a$

2.4.1.2 Conditionnelle Scheme

Dans la forme générale de l'opérateur cond, les couples ($<p_i>$ $<e_i>$) sont appelés clauses, les $<p_i>$ prédicats (expressions dont le résultat est un booléen) et les $<e_i>$ conséquences. Pour évaluer un cond, on évalue d'abord $<p_1>$, puis éventuellement $<p_2>$, $<p_3>$... jusqu'au premier $<p_i>$ délivrant la valeur vrai. S'il en existe un, le résultat du cond est le résultat de l'évaluation de la conséquence $<e_i>$ associée. Si aucun des $<p_i>$ n'est vrai, cond a un résultat indéfini[1].

1. Les identificateurs de fonctions dont le résultat est booléen sont souvent terminés par un « ? ». Il s'agit d'une convention (et uniquement d'une convention) pratique que nous conseillons d'adopter.

2. Le type indifférent (cf. § 2.2.1) est un type qui regroupe tous les objets Scheme: c'est donc l'union de tous les types. Son usage signifie que la fonction peut être appliquée à un objet quelconque.

1. Il importe de ne pas confondre résultat indéfini et type indifférent: un résultat indéfini signifie que le résultat n'a pas de sens (il s'agit en fait d'une erreur, d'une exécution non prévue). Selon les mises en œuvre du langage, l'exécution peut être arrêtée ou le résultat peut être une valeur arbitraire. Dans le cas qui nous intéresse, il est peu souhaitable que les conditions prévues par le programmeur soient toutes fausses: cela témoigne souvent d'une lacune dans la conception. Pour éviter de tels cas de figure, le dernier prédicat d'un cond est souvent la valeur vrai, ce qui fait que, dans le pire cas, la dernière branche délivre le résultat.

Définition 2.10 Expression Scheme: conditionnelle

Syntaxe

<expression> ::= <*notation*> | <*identificateur*> | <*appel*> | <*définition*> | <*fonction*> | <conditionnelle>

<conditionnelle> ::= **(** **cond** <pred-exp> **)**
<pred-exp> ::= **(** <prédicat> <expression> **)** |
(<prédicat> <expression> **)** <pred-exp>
<prédicat> ::= <expression>

- <prédicat> est une expression à résultat booléen.

Sémantique

- Le résultat d'une <conditionnelle> est le résultat de l'<expression> associée au premier <prédicat> délivrant la valeur vrai. S'il n'y en a pas, le résultat est indéfini.

On constate que l'opérateur cond permet donc de traduire une étude par cas. Toutefois, il faut remarquer que si, dans une bonne étude par cas, on fait une partition du domaine de définition, l'opérateur cond lui, ne vérifie ni que les $<p_i>$ sont disjoints ni que tous les cas sont traités.

Définition 2.11 Étude par cas

Lorsqu'on cherche un algorithme pour une fonction, une *étude par cas* est une méthode qui consiste à réaliser une partition du domaine de définition et à rechercher un algorithme pour chacun des éléments de la partition.

- Il est important que le « découpage » du domaine soit bien une partition: les éléments doivent être disjoints deux à deux et leur union doit recouvrir le domaine en entier.
- Le résultat d'une étude par cas se traduit directement en Scheme par une <conditionnelle>.

2.5 Désignation et contextes

Le rôle d'un identificateur est de désigner un objet. Le programmeur va donc utiliser un nombre parfois grand d'identificateurs et il est nécessaire de disposer de règles précises pour pouvoir déterminer quel objet est associé à un identificateur donné.

2.5.1 Identificateurs libres et liés

Considérons la fonction:

```
(lambda (r) (* Pi (Carré r)))
```

Le corps de cette fonction utilise quatre identificateurs: *, Pi, Carré et r. Au vu de ce seul texte, on comprend ce que désigne r lorsque le corps de la fonction est exécuté; il désigne à ce moment le paramètre effectif qui a été fourni à l'appel. On comprend également que, à l'extérieur de ce corps, l'identificateur r soit n'a pas de sens, soit désigne autre chose. En revanche, le texte de cette fonction ne permet pas, à lui seul, de dire ce que désignent *, Pi et *Carré*.

Ces constatations triviales entraînent une question et une remarque de bon sens.

Question: Quelles sont les règles qui permettent de savoir ce que désigne un identificateur?

Remarque: Quelles que soient les règles en question, elles supposent la donnée du programme complet pour les appliquer. Cependant, d'un point de vue méthodologique, il est commode de considérer chaque fonction de manière isolée. On en tire deux conséquences: (1) il est souhaitable qu'une fonction dépende le moins possible d'identificateurs autres que ses paramètres et (2) le choix d'identificateurs parlants est impératif. À titre d'illustration, considérons quatre versions du calcul de l'aire d'un cercle:

Version 1: (lambda (r) (* Pi (Carré r)))
Version 2: (lambda () (* Pi (Carré r)))
Version 3: (lambda (r) (op1 val (op2 r)))
Version 4: (lambda (r op1 val op2) (op1 val (op2 r)))

- La version 1 est lisible; on constate qu'il lui est nécessaire de disposer de deux fonctions (* et Carré) et d'une valeur (Pi) dont on imagine sans peine le sens.
- La version 2 peut éventuellement calculer l'aire d'un cercle, si par hasard r désigne une valeur qui est son rayon; cette hypothèse est déraisonnable.
- La version 3, illisible, ne permet pas d'imaginer qu'il s'agit d'un calcul d'aire.
- La version 4 permet de calculer beaucoup de choses; en particulier, un appel fournissant rayon, multiplication, Pi et Carré permet de calculer l'aire d'un cercle; c'est quand même lourd.

Considérons la fonction (lambda (r) (* Pi (Carré r))). Du point de vue terminologique, quand on examine le corps de cette fonction, on dit que l'identificateur r est *lié* alors que les identificateurs *, Pi et Carré sont dits *libres*.

Sur un plan général, on doit se soucier d'avoir le moins possible d'identificateurs libres (ie. avoir de bonnes raisons pour laisser libres des identificateurs) et, dans tous les cas, veiller à ce que leurs noms soient judicieusement choisis.

2.5.2 Choix des identificateurs

Il est donc particulièrement important d'apprendre à choisir avec discernement les identificateurs qu'on utilise. Nous donnons quelques conseils:

1. Les identificateurs réduits à une seule lettre (ou à une lettre suivie d'un chiffre) doivent être impérativement limités aux paramètres formels d'une fonction dont le corps est petit (c'est le cas de x dans la fonction Carré). Dans ce cas, on respecte un certain nombre de conventions issues de notre culture mathématique (i et n sont des entiers…).
2. Il ne faut pas hésiter à utiliser des identificateurs longs[1] (et parlants). Pour ce faire, on est souvent amené à utiliser plusieurs mots. Deux conventions se partagent les faveurs de la profession:

- On n'utilise que des lettres et des chiffres et les divers mots sont mis en valeur par l'utilisation de capitales[2]: Carré, MiseEnFacteur, PremierElément…
- On n'utilise que des lettres, des chiffres et un séparateur (en général « - » ou « _ »): carré, mise-en-facteur, premier-élément…

Il est important de comprendre que la pertinence du choix des identificateurs est un des facteurs de qualité d'un programme. Pour cette raison, des normes très contraignantes sont imposées dans les entreprises qui produisent des logiciels.

Dans le cadre de cet ouvrage nous utilisons en général les capitales. Toutefois, pour distinguer les fonctions prédéfinies de Scheme, nous les notons en minuscules et nous conservons leurs noms anglo-saxons.

2.5.3 Blocs et portée statique des identificateurs

Si on n'a pas de problèmes pour déterminer quel objet désigne un identificateur lié, il faut des règles pour donner un sens aux identificateurs libres. Un langage comme Scheme est un langage à *structure de bloc.*

Considérons l'appel de fonction suivant:

```
(
    (lambda (x y)
        (+ x
            (
                (lambda (x z)
                    (+ x (+ y z))
                )
                3 4
            )
        )
    )
```

1. Les anglo-saxons sont, à l'évidence, avantagés: par le nombre important de mots courts dans le lexique de l'anglais, par la structure des génitifs, par le mode de formation des noms composés…

2. Attention: Scheme (comme d'autres langages de programmation) ne distingue pas capitales et minuscules: carré et Carré sont deux variantes du même identificateurs.

```
      1 2
)
```

Dans le corps de la fonction interne (lambda (x z) (+ x (+ y z))) les identificateurs x et z sont liés, mais y est libre. En revanche, dans le corps de la fonction externe, x et y sont liés. Il semble assez naturel que dans l'expression (+ x (+ y z)), x désigne le x de la fonction interne et y le y de la fonction externe. On dit qu'il s'agit d'une *structure de blocs* imbriqués.

Une expression (lambda <pf> <corps>) définit un *bloc* (le corps de la fonction); les identificateurs cités dans <pf> ont une *portée* dite statique (ou lexicale): ils ont un sens dans le bloc défini par la fonction, sous réserve de ne pas être redéfinis par un bloc interne.

On peut schématiser l'exemple précédent ainsi:

Bloc 1 : x_1, y_1
…
x_1
…
Bloc 2 : x_2, z_2

…
x_2
…
y_1
…
z_2
…
Fin du bloc 2

…
x_1
…
y_1
…
Fin du bloc 1

En résumé, on dispose d'une règle permettant d'associer statiquement une *occurrence d'utilisation* d'un identificateur à son *occurrence de définition*.

Définition 2.12 Identificateurs: occurrences de définition et occurrences d'utilisation

- Une *occurrence de définition*, pour un identificateur est son apparition dans un define ou dans une liste de paramètres formels.
- Toutes les autres occurrences sont des *occurrences d'utilisation.*
- En Scheme, le lien entre occurrence d'utilisation et occurrence de définition est fait statiquement en utilisant la plus « proche » occurrence de définition au sens de l'imbrication des blocs.

2.5.4 Le bloc 0

Le bloc dit 0 est un cas particulier. On peut le voir comme un environnement où il y a, potentiellement, tous les identificateurs, certains d'entre eux désignant les objets prédéfinis en Scheme. Le rôle de define consiste à modifier cet environnement. Nous illustrons ce mécanisme sur la séquence suivante:

```
[0]  (define Carré (lambda (x) (* x x)))
Carré
[1]  (define Pi 3)
pi
[2]  (define AireC (lambda (r) (* Pi (Carré r))))
AireC
[3]  (define DeuxPi (* 2 pi))
DeuxPi
[4]  (AireC 1)
3
[5]  DeuxPi
6
[6]  (define Pi 3.1)
pi
[7]  (AireC 1)
;                    Pi, utilisé dans AireC a changé en [6]. Le résultat en [7] diffère
;                    donc du résultat en [4]
3.1
[8]  DeuxPi
;                    DeuxPi désigne une valeur, 6, définie en [3].
;                    Cette valeur est indépendantes des valeurs ultérieures de Pi.
6
[9]  (define Gris (lambda (Carré)
              (- (* Carré Carré) (AireC (/ Carré 2)))))
;                    L'identificateur Carré du bloc 1 masque, dans le bloc 1,
;                    l'identificateur Carré du bloc 0
Gris
[10] (Gris 2)
0.9
```

2.6 Point sur le langage Scheme

2.6.1 Fonctionnalités introduites

La grammaire qui suit fait le point de ce que nous avons vu du langage Scheme.

Il importe de comprendre ce que fait l'interpréteur lorsqu'il rencontre une expression de la forme (<truc> ...). Compte tenu de la syntaxe donnée, il regarde si <truc> est define, lambda ou cond. Si ce n'est pas le cas, <truc> doit absolument lui permettre d'obtenir une fonction! Cette simple constatation permet d'expliquer certains messages d'erreur comme *Attempt to call a non-procedural object* provoqué par l'expression (1 2 3).

Grammaire du langage Scheme

Définition 2.13 Expressions Scheme: syntaxe générale		
<expression>	::=	<notation> | <identificateur> | <appel> | <définition> | <fonction> | <conditionnelle>
<notation>	::=	**constante Scheme**
<identificateur>	::=	**identificateur Scheme**
<appel>	::=	**(** <expression> <suite_de_pe> **)**
<suite_de_pe>	::=	**rien** | <expression> <suite_de_pe>
<définition>	::=	**(define** <identificateur> <expression> **)**
<fonction>	::=	**(lambda (** <suite_de_pf> **)** <expression> **)**
<suite_de_pf>	::=	**rien** | <identificateur> <suite_de_pf>
<conditionnelle>	::=	**(cond** <pred-exp> **)**
<pred-exp>	::=	**(** <prédicat> <expression> **)** | **(** <prédicat> <expression> **)** <pred-exp>
<prédicat>	::=	<expression>

2.6.2 Types de base et fonctions prédéfinies

Nous utiliserons par la suite un certain nombre de types de bases et de fonctions prédéfinies[1] utilisables en Scheme.

Types de base

- naturel = ensemble des entiers positifs ou nuls.
- entier (*integer*) = ensemble des entiers relatifs.
- réel (*real*) = ensemble des réels positifs et négatifs manipulés par la machine.
- nombre (*number*) = entier $\cup$ réel.
- booléen (*boolean*) = ensemble des deux booléens.
- indifférent = ensemble de tous les objets Scheme.

On dispose des fonctions prédéfinies suivantes:

2.6.2.1 Fonctions arithmétiques

De nombreuses fonctions Scheme ont, en réalité, un type qui sort du cadre que nous avons défini: elles admettent un nombre quelconque de paramètres. Ainsi, le type entier... $\rightarrow$ entier désigne le type d'une fonction ayant un ou plusieurs paramètres entiers.

1. Les fonctions ayant un réel statut dans le langage, nous utilisons les noms anglo-saxons fournis par le distributeur du logiciel. Les types, eux, relevant d'une convention, nous conserverons les noms français en indiquant leur correspondance en anglais.

+:	nombre…	→	nombre
-:	nombre…	→	nombre
*:	nombre…	→	nombre
/:	nombre…	→	nombre

remainder:	entier x entier - {0}	→	entier
quotient:	entier x entier - {0}	→	entier

2.6.2.2 Fonctions de comparaison

=:	nombre x nombre	→	booléen
>:	nombre x nombre	→	booléen
<:	nombre x nombre	→	booléen
<=:	nombre x nombre	→	booléen
>=:	nombre *x* nombre	→	booléen

equal?:indifférent x indifférent→ booléen

2.6.2.3 Fonctions booléennes

and:	booléen…	→	booléen
or:	booléen…	→	booléen
not:	booléen	→	booléen

2.6.2.4 Tests de type

integer?:	indifférent	→	booléen
real?:	indifférent	→	booléen
number?:	indifférent	→	booléen

2.7 Conclusions

À l'issue de ce chapitre nous disposons de la connaissance presque complète des fonctionnalités du langage Scheme. Il est cependant clair que cette connaissance ne permet pas à un débutant de programmer!

Ce qui lui manque ne fait pas partie du langage: il s'agit d'un savoir-faire qui ne s'acquiert qu'en pratiquant mais il s'agit surtout d'*éléments méthodologiques.* Ces derniers forment le cœur de l'apprentissage de l'art de programmer et l'essentiel du cours leur est consacré.

2.8 Exercices

2.8.1 Notions de base

Exercice 2.1 Scheme: expressions simples

Donner le résultat de l'évaluation des expressions Scheme qui suivent.

```
(+ 2 (* 4 5))
(* (+ 2 3) (+ 4 5))
(+ (* (+ 2 3) 4) (* (+ 2 3) 5))
(* (* (* (* 1 2) 3) 4) 5)
(* 1 (* 2 (* 3 (* 4 5))))
```

```
((lambda (x y) (* (+ x y) 3)) 2 5)
(* 3 ((lambda (x) (* x x)) 3))
((lambda (x y) (+ (* x y) 2)) (+ 7 11) (+ 5 2))
((lambda (x y z) (+ (* x y) (* x z))) 2 3 4)
((lambda (x y z u) z) 1023 917 79 402)
```

Exercice 2.2 Composition d'appels de fonctions

On considère les identificateurs:

a:	nombre		
b:	nombre		
c:	nombre		
-:	nombre x nombre	→	nombre
+:	nombre x nombre	→	nombre
*:	nombre x nombre	→	nombre
/:	nombre x nombre	→	nombre
sqrt:	nombre	→	nombre

On suppose que +, -, *, / désignent les fonctions habituelles, que sqrt désigne une fonction calculant la racine carrée de son paramètre et que a, b et c désignent trois nombres quelconques.

- Proposer une expression Scheme pour calculer $\frac{-b + \sqrt{b^2-4ac}}{2a}$.

Exercice 2.3 Calcul d'aires (1)

Concevoir et coder en Scheme des fonctions qui calculent les aires des objets suivants: carré, rectangle, cercle, triangle quelconque, surface d'un cylindre.

Exercice 2.4 Calcul d'aires (2)

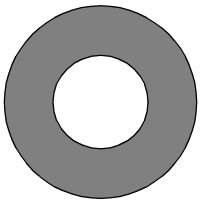
L'aire d'un triangle équilatéral de côté a peut être calculée par la fonction:

ATREQ:	nombre	→	nombre
	a	→	$\frac{a^2\sqrt{3}}{4}$

- Coder cette fonction en Scheme en prenant 1.732 pour valeur de $\sqrt{3}$.
- Concevoir et coder une fonction calculant l'aire d'un hexagone régulier de côté a.

Exercice 2.5 Calcul d'aires (3)

Concevoir et coder une fonction calculant l'aire d'une couronne définie par deux cercles concentriques de rayon R1 et R2.



Exercice 2.6 Calcul de volumes

Concevoir et coder des fonctions qui calculent le volume d'un cube, d'un parallélépipède rectangle, d'une sphère, d'un prisme à base hexagonale...

2.8.2 Conditionnelles

Exercice 2.7 Propriétés des fonctions booléennes

Exprimer

- $\neg (a \vee b)$ à l'aide de $\wedge$ et de $\neg$
- $\neg (a \wedge b)$ à l'aide de $\vee$ et de $\neg$
- a => b à l'aide de $\vee$ et de $\neg$
- a => b à l'aide de $\wedge$ et de $\neg$
- a <=> b à l'aide de $\wedge$ et de $\neg$

Exercice 2.8 Fonctions booléennes de base

Ecrire les trois fonctions booléennes Et, Ou, Non (sans utiliser, bien sûr, les fonctions prédéfinies and, or et not).

Exercice 2.9 Fonctions booléennes évoluées

Ecrire les fonctions booléennes OuExclusif, Implique et Équivalent.

Exercice 2.10 Maximum de 2 entiers

Ecrire une fonction Max qui délivre en résultat le plus grand des deux entiers fournis en paramètre.

Exercice 2.11 Facturation (1)

Un marchand de vin expédie une quantité q de vin de prix unitaire p. Si le total de la commande est d'au moins 500 F, le port est gratuit. Sinon, il est facturé 10% de la commande. Écrire une fonction qui, étant donné q et p délivre la somme à payer.

Exercice 2.12 Facturation (2)

Même exercice que l'exercice précédent mais dans le cas où le port n'est pas gratuit, il est de 10% de la commande avec un minimum de 10 F.

2.8.3 Typage

Exercice 2.13 Etude d'une fonction

Soit le programme Scheme:

```
; À compléter
(define Test (lambda (x y)
      (cond ((not (or (<= x y) (= x (* y y))))      (Test x (+ y 1)))
            (t                                      (= x (* y y))))))
```

- Quel est le type de cette fonction?
- Se termine-t-elle toujours? Pourquoi?

Exercice 2.14 Booléens

On convient que les quatre fonctions qui suivent ont un paramètre de type entier.

```
; À compléter
(define Entre3et5 (lambda (x)
      (and (> x 3) (< x 5))))
; À compléter
(define Test1 (lambda (x)
      (cond
            ((= (Entre3et5 x) #f)        1)
            (#t                          2))))
; À compléter
(define Test2 (lambda (x)
      (cond
            ((equal? (Entre3et5 x) #f)   1)
            (#t                          2))))
; À compléter
(define Test3 (lambda (x)
      (cond
            ((not (Entre3et5 x))         1)
            (#t                          2))))
```

- Effectuer le contrôle de type de ces quatre fonctions.

CHAPITRE 3

Analyse descendante

3.1 Conception et réalisation de programmes

Dans le cadre de ce cours, nous n'écrirons que des programmes très courts: il s'agit et d'un cours et d'une initiation. Dans la réalité les logiciels à concevoir et à écrire sont gros (certains atteignent le million de lignes). Pour en venir à bout, il faut utiliser des méthodes rigoureuses de production dont le débutant a parfois du mal à apprécier l'intérêt. Nous introduirons, au fur et à mesure des besoins, des éléments succincts de méthode, et il est impératif de les utiliser.

La démarche de conception et de réalisation d'un logiciel comporte quatre phases.

Établissement et négociation du cahier des charges avec le client
Même si c'est une évidence, il est utile de répéter avec force que lorsqu'on décide de réaliser un programme, il faut savoir le plus précisément possible ce qu'il doit faire. Il y a trois raisons à cela:

- De nombreuses insatisfactions des clients sur les logiciels qu'ils ont payés très chers proviennent d'ambiguïtés, d'imprécisions ou d'oublis sur ce qu'ils attendaient.
- Comme tout produit manufacturé, un programme doit pouvoir subir des contrôles de conformité: encore faut-il savoir à quoi le programme doit être conforme.
- La précision du cahier des charges facilite les étapes ultérieures: c'est un facteur crucial de la productivité de la conception.

Il faut donc dialoguer avec le client, l'aider à exprimer ses besoins et produire un document précis. Ce document est ce qu'on appelle la *spécification* du problème.

Recherche d'un algorithme
Muni d'une spécification, on peut commencer à concevoir le programme. Les débutants ont du mal à admettre que cette phase, cruciale, n'utilise pas le langage de programmation qu'ils apprennent. La difficulté est toute autre: il s'agit de maîtriser la complexité du problème (souvent apocalyptique) et de proposer un algorithme précis, complet et dont on doit pouvoir analyser la correction.

Cette phase est l'objet principal de notre attention dans ce cours, même si nous ne donnons que des éléments de méthode adaptés à de petits programmes[1].

À l'issue de cette phase, on fournit une *description de l'algorithme*.

Codage de l'algorithme à l'aide d'un langage de programmation
Muni de la description de l'algorithme, il est possible de l'exprimer à l'aide du langage de programmation retenu. Cette phase, dite de *codage*, certes très technique, ne pose en réalité que des problèmes d'organisation et de contrôle de qualité.

Mise au point
Malgré toutes les précautions prises, l'expérience montre que la plupart des programmes comportent des erreurs. Il est clair qu'une politique de test (la *mise au point*) permet d'en repérer et d'en corriger un certain nombre.

Recette
Lorsqu'on livre le logiciel au client, ce dernier lui fait subir un certain nombre de vérifications de conformité.

3.1.1 Spécification

La première étape fournit ce qu'on appelle une *spécification* du problème. La spécification doit être précise et complète. Pour ce faire, on utilise souvent des langages de spécification. Dans le cadre où nous travaillons (approche dite fonctionnelle) un problème à résoudre c'est une fonction à écrire. En conséquence la spécification comprendra obligatoirement le type de la fonction et une description en français du résultat qu'elle doit produire à partir de ses paramètres. Ainsi, la spécification du problème « recherche de la valeur absolue d'un entier » est:

Abs:	entier	→	naturel
	x	→	| x |

Dans le cadre du cours, pour décrire la relation entre le résultat et les paramètres on utilise un langage informel (français, expressions mathématiques) mais on cherche une description:

- précise et sans ambiguïté,

1. Un autre module de cours (génie logiciel), proposé en Deug, vise à approfondir le sujet.

• indépendante d'un algorithme particulier.

Définition 3.1 Spécification

La *spécification* d'une fonction est:

• la donnée de son type,

• l'expression de la propriété qui relie son résultat et ses paramètres.

3.1.2 Recherche d'un algorithme

La phase de recherche d'un algorithme est la plus difficile dans la mesure où elle suppose un effort de créativité. Un certain nombre de grandes techniques peuvent cependant être dégagées. L'une d'entre elles, formalisée par DESCARTES, consiste à diviser le problème initial en sous-problèmes plus faciles à résoudre. Ce type d'approche est appelée *démarche descendante* (cf. définition 1.5 page 12).

Dans notre contexte, cela revient à spécifier et traiter un certain nombre de sous-fonctions; on parle de découpage fonctionnel. À l'issue de la phase de recherche d'un algorithme on doit produire:

• Un document d'analyse présentant le *découpage fonctionnel*.

• Pour chaque fonction, sa spécification et un algorithme. On utilise à ce stade, un langage assez libre, le *langage de description*.

Les principales caractéristiques du langage de description que nous utilisons dans ce cours sont les suivantes:

Définition 3.2 Principales caractéristiques du langage de description

• La structure de base du *langage de description* est une expression de la forme:

NomDeFonction(paramètres) = algorithme

• On utilise de préférence les notations mathématiques habituelles. En particulier, on utilise la notation usuelle pour les fonctions: f(x, y) et non (f x y).

• Le résultat d'une étude par cas est exprimé par une accolade et les divers cas se lisent d'ordinaire du haut vers le bas; l'usage de sinon signifie « dans tous les cas non prévus plus haut ».

Il faut insister, lors de cette étape, sur le fait qu'on doit être capable de prouver qu'on a un algorithme qui répond à ses spécifications: comme dans tout processus de production, les erreurs détectées tardivement coûtent très cher. La détection précoce est donc un impératif économique.

3.1.3 Codage

La phase de codage, même si elle est longue et fastidieuse, même si on y introduit des erreurs difficiles à corriger, ne demande que peu de créativité,

pourvu que l'on possède bien le langage de programmation utilisé. On attire donc l'attention du lecteur sur le fait que les difficultés qu'il va rencontrer à ce stade sont dues à son inexpérience: ce qui est réellement difficile, c'est l'étape précédente.

3.1.4 Mise au point

La phase de mise au point se déroule devant des machines et fait appel à des outils et méthodes que nous n'aborderons pas dans le cadre de ce cours. On notera toutefois que l'apparition de machine dans le processus est tardive. Dans la réalité, on utilise des machines plus tôt dans la mesure où on dispose d'outils informatiques d'aide à l'élaboration des phases précédentes. On retiendra donc, pour ce qui nous concerne, qu'il est inutile de se présenter devant une machine avant le stade de la mise au point.

3.1.5 Recette

La recette est la remise au client du produit; elle s'accompagne donc d'une procédure de contrôle de conformité vis-à-vis du cahier des charges.

3.2 Application

Le problème posé par le client est le suivant: étant donné un réel positif p, trouver la valeur de $\sqrt{p}$.

On constate que les connaissances utiles pour faire face à un tel défi relèvent de deux champs distincts: il faut des connaissances sur le domaine d'application et des connaissances purement informatiques, qui sont essentiellement des éléments méthodologiques. Pour ce qui concerne le premier point, de telles connaissances ne s'inventent pas et il faut souvent recourir à des spécialistes. Cependant, une culture scientifique générale permet de résoudre bon nombre de cas usuels.

3.2.1 Cahier des charges

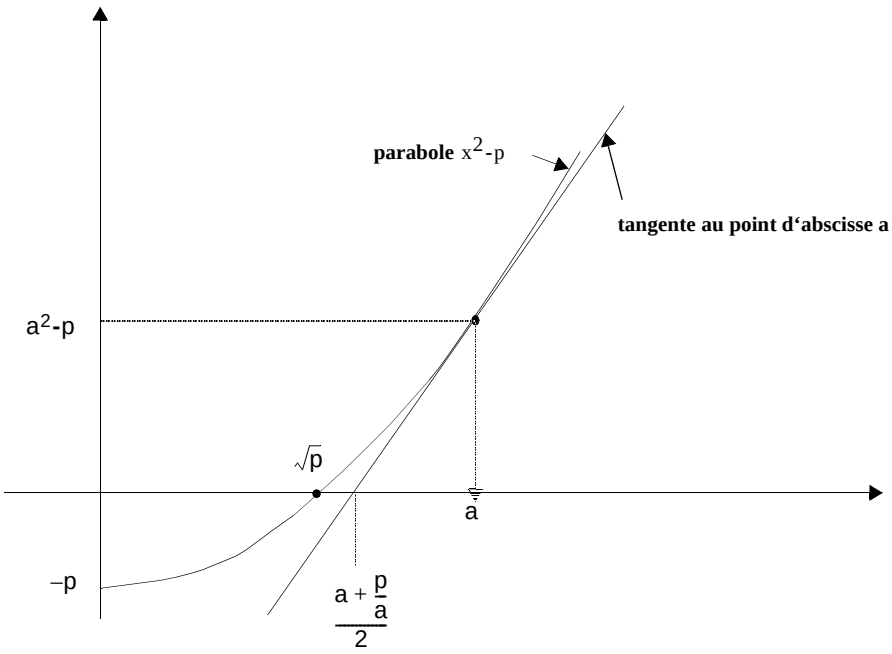
Pour le problème qui nous préoccupe il faut savoir que l'algorithmique sur les réels vise d'ordinaire des approximations et non des résultats exacts. En d'autres termes le problème doit être reformulé ainsi: étant donnés deux réels positifs p et e, on veut trouver un nombre q tel que $| q^2 - p | < e$. D'où la spécification:

Racine:	réel$^+$ x réel$^+$	$\rightarrow$	réel$^+$
	p, e	$\rightarrow$	un nombre q tel que $\| q^2 - p \| < e$

3.2.2 Recherche d'un algorithme

Pour résoudre ce problème on peut utiliser un algorithme, l'algorithme de NEWTON[1], dont nous esquissons une brève justification.

Si on considère la parabole d'équation $y = x^2 - p$, on voit qu'elle coupe l'axe



des x au point d'abscisse $\sqrt{p}$. Soit un point d'abscisse $a > \sqrt{p}$ et traçons une droite, tangente à la parabole au point d'abscisse a. La pente de cette tangente étant 2a (puisque y'=2x), elle coupe l'axe des x au point d'abscisse $b = \frac{a + \frac{p}{a}}{2}$. Il est clair que $\sqrt{p} < b < a$. En d'autres termes, si on considère que a est une approximation de $\sqrt{p}$ alors b est une meilleure approximation de $\sqrt{p}$. On peut évidemment itérer le procédé en partant de b et, ce faisant, calculer une suite de valeurs strictement décroissante et minorée par $\sqrt{p}$.

Remarque

Si on part d'une valeur $0 < a < \sqrt{p}$, la valeur b calculée est supérieure à $\sqrt{p}$, ce qui nous ramène au cas de figure étudié.

Nous disposons ainsi d'un moyen de trouver une approximation aussi bonne qu'on veut de q en partant d'une approximation initiale quelconque.

1. Cet algorithme a d'autres applications que le calcul d'une racine carrée: il permet de trouver une approximation d'un zéro de certaines fonctions.

A ce stade des opérations on dispose d'une idée de l'algorithme et pour trouver une bonne découpe fonctionnelle, on adopte une démarche purement informatique, dite *démarche descendante*: on décompose le problème initial en sous-problèmes plus simples à résoudre.

- Appelons RacIter une fonction à trois paramètres, p, e (avec la signification précédente) et a qui, à partir d'une approximation a de q, délivre le résultat attendu.

RacIter:	$réel^+ \times réel^+ \times réel^+$	$\rightarrow$	$réel^+$
	p, e, a	$\rightarrow$	à partir d'une approximation a de la racine de p, délivre un nombre q tel que $\lvert q^2 - p \rvert < e$

On peut écrire:

$$Racine(p, e) = RacIter(p, e, 1)$$

- Supposons maintenant qu'on dispose de deux fonctions, AssezBon? et Améliore ainsi spécifiées:

AssezBon?:	$réel^+ \times réel^+ \times réel^+$	$\rightarrow$	booléen
	p, a, e	$\rightarrow$	$\lvert a^2 - p \rvert < e$
Améliore:	$réel^+ \times réel^+$	$\rightarrow$	$réel^+$
	p, a	$\rightarrow$	$(a + (p/a))/2$

On peut écrire:

$$AssezBon?(p, a, e) = Abs(a^2 - p) < e$$

$$Améliore(p, a) = \frac{a + \frac{p}{a}}{2}$$

$$RacIter(p, e, a) = \begin{cases} a \textbf{ si } AssezBon?(p, a, e) \\ RacIter(p, e, Améliore(p, a)) \textbf{ sinon} \end{cases}$$

Si on dispose des fonctions Abs et Carré décrites au chapitre précédent, on peut considérer qu'on est arrivé à des problèmes qu'on sait résoudre et la phase de recherche d'un algorithme est terminée. L'expression de l'algorithme consiste en les quatre équations que nous avons posées pour définir Racine, RacIter, AssezBon? et Améliore.

L'outil de base du langage que nous utilisons étant la fonction, il n'y a pas lieu de s'étonner du fait que pour décrire des algorithmes, nous définissons des fonctions. Le lecteur remarquera que, pour conserver au langage de description son caractère peu contraint et naturel, nous utilisons les notations mathématiques usuelles.

A l'issue de la phase de recherche d'un algorithme, on est en mesure de fournir un dossier facilitant et le codage et l'évolution ultérieure du produit.

Définition 3.3 Contenu d'un dossier d'analyse

Un dossier d'analyse contient les éléments suivants:

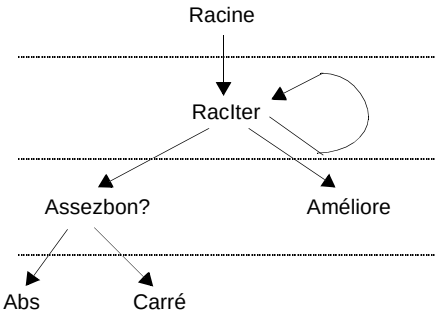
- Spécification du problème posé
- Découpage fonctionnel
- Spécifications des fonctions introduites
- Algorithmes de toutes les fonctions

Recherche d'une approximation de la racine d'un réel p

1. Problème

Racine: $\text{réel}^+ \times \text{réel}^+ \rightarrow \text{réel}^+$
p, e → un nombre q tel que $| q^2 - p | < e$

2. Découpage fonctionnel



3. Fonctions introduites

RacIter: $\text{réel}^+ \times \text{réel}^+ \times \text{réel}^+ \rightarrow \text{réel}^+$
p, e, a → à partir d'une approximation a de la racine de p, délivre un nombre q tel que $| q^2 - p | < e$

AssezBon?: $\text{réel}^+ \times \text{réel}^+ \times \text{réel}^+ \rightarrow$ booléen
p, a, e → $| a^2 - p | < e$

Améliore: $\text{réel}^+ \times \text{réel}^+ \rightarrow \text{réel}^+$
p, a → $(a + (p/a))/2$

4. Algorithmes

Racine(p, e) = RacIter(p, e, 1)

$$\text{RacIter}(p, e, a) = \begin{cases} a \textbf{ si } \text{AssezBon?}(p, a, e) \\ \text{RacIter}(p, e, \text{Améliore}(p, a)) \textbf{ sinon} \end{cases}$$

$\text{AssezBon?}(p, a, e) = \text{Abs}(a^2 - p) < e$

$$\text{Améliore}(p, a) = \frac{a + \frac{p}{a}}{2}$$

Muni de ce dossier, la phase suivante devient assez mécanique.

Phase de codage
On obtient le programme Scheme:

```
; Racine:    réel+ x réel+  ->     réel+
;            p, e           ->     un nombre q tel que | q^2 - p | < e
(define Racine (lambda (p e)
      (RacIter p e 1)))

; RacIter:   réel+ x réel+ x réel+  ->     réel+
;            p, e, a                ->     à partir d'une approximation a de la
;                                          racine de p, délivre un nombre q tel que
;                                          | q^2 - p | < e
(define RacIter (lambda (p e a)
      (cond
            ((AssezBon? p a e)   a)
            (#t                  (RacIter p e (Améliore p a))))))

; AssezBon?:réel+ x réel+ x réel+ ->     booléen
;            p, a, e                ->     | a^2 - p | < e
(define AssezBon? (lambda (p a e)
      (< (Abs (- (Carré a) p)) e))))

; Améliore:  réel+ x réel+          ->     réel+
;            p, a                   ->     (a + (p/a))/2
(define Améliore (lambda (p a)
      (/ (+ a (/ p a)) 2)))
```

Les phases de mise au point et de recette ne sont pas traitées dans ce polycopié. Il faut être conscient qu'elles sont indispensables et souvent difficiles.

3.3 Conclusions

A l'issue de ce chapitre nous sommes en possession d'un certain nombre d'outils techniques ou méthodologiques: connaissances de base sur le lan-

gage de codage, Scheme, et méthode générale d'approche et de résolution d'un problème sur laquelle il convient d'insister.

Définition 3.4 Méthodologie générale de résolution de problèmes

La *résolution d'un problème* passe par les phases suivantes:

- Etablissement et négociation du cahier des charges avec le client, donnant lieu à une *spécification* (type et description de la fonction à écrire).
- Recherche d'un algorithme en utilisant une *approche descendante* (décomposition du problème en problèmes plus simples à résoudre), donnant lieu à un *découpage fonctionnel* et à un algorithme exprimé à l'aide d'un *langage de description* (équations fonctionnelles utilisant les notations mathématiques habituelles).
- Codage de l'algorithme à l'aide d'un langage de programmation.
- Mise au point.
- Recette.

On retiendra le terme d'*abstraction fonctionnelle* pour éclairer ces éléments méthodologiques. Quand on a décomposé un problème en sous-problèmes plus simples, on a d'une certaine manière défini une machine abstraite munie des bonnes fonctions (celles qui résolvent les sous-problèmes) pour résoudre le problème posé.

3.4 Exercices

Exercice 3.1 Calcul d'un zéro d'une fonction par la méthode dichotomique

Soit f une fonction continue sur [a,b]. Si $f(a)f(b) \leq 0$, alors f possède un zéro dans [a,b] dont on cherche une approximation.

Pour ce faire, on construit une suite décroissante d'intervalle $I_n = [a_n,b_n]$ telle que $I_o = [a,b]$, chaque intervalle vérifiant la propriété $f(a_n)f(b_n) \leq 0$.

Pour passer de I_n à I_{n+1}, on peut utiliser le milieu c_n de l'intervalle $[a_n,b_n]$, ce qui mène à:

si $f(a_n).f(c_n) < 0$
alors $I_{n+1} = [a_n, c_n]$ sinon $I_{n+1} = [c_n, b_n]$

Les deux suites a_n et b_n convergent alors vers un zéro c de f.

Construire un dossier d'analyse et de codage pour résoudre le problème du calcul d'un zéro d'une fonction f.

Exercice 3.2 Calcul d'un point fixe

Etant donné une fonction f et un nombre a, il se peut que la suite $u_0 = a$, $u_1 = f(a)$, $u_2 = f(f(a))$ ... converge; on dit alors que la fonction admet un point fixe.

Concevoir une fonction PointFixe qui, à partir de deux nombres a et e, calcule une valeur u_i vérifiant $|u_{i-1} - u_i| < e$ s'il en existe une. On ajoutera un paramètre n bornant la recherche aux u_i, $0 \leq i \leq n$.

CHAPITRE 4

Conception de fonctions récursives

4.1 Récursivité

La fonction RacIter que nous avons vue au chapitre précédent a une particularité a priori étonnante: elle s'appelle elle-même. On dit que RacIter est une fonction *récursive* et la récursivité est le concept central de ce cours.

4.1.1 Définitions récursives

Considérons la fonction suivante:

Fact:	naturel	→	naturel
	n	→	n!

Pour programmer une telle fonction, on doit, de même que pour l'exemple de la racine carrée, faire un minimum appel à des propriétés du domaine d'application. On peut considérer que n! est défini par les propriétés:

$$n! = \begin{cases} 1 \textbf{ si } n=0 \\ n \times (n-1)! \textbf{ si } n > 0 \end{cases}$$

On remarque d'abord que cette définition se prête bien à une étude par cas. On effectue en effet une partition du domaine naturel en {0} et naturel - {0}. Si on se contente de traduire littéralement cette définition on obtient:

```
(define Fact (lambda (n)
    (cond
        ((= n 0)        1)
        ((> n 0)        (* n (Fact (- n 1)))))))
```

Une définition de fonction comme celle que nous avons donnée pour n! est appelée définition *récursive*: la définition de la fonction utilise la fonction elle-même.

Définition 4.1 Récursivité

En informatique, une fonction f est dite récursive lorsque l'algorithme employé pour son calcul utilise f.

Remarque
La récursivité peut être directe – f appelle f – ou indirecte (on dit croisée) lorsque f appelle g qui appelle… qui appelle f.

Des définitions récursives permettent d'obtenir, en Scheme, des programmes récursifs et l'étude de la récursivité est un domaine important de la programmation. Constatons en premier lieu que, si on applique le modèle d'évaluation en ordre applicatif, on obtient bien le résultat souhaité.

```
(Fact 6)
(* 6 (Fact 5))
(* 6 (* 5 (Fact 4)))
(* 6 (* 5 (* 4 (Fact 3))))
(* 6 (* 5 (* 4 (* 3 (Fact 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (Fact 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 (* 1 (Fact 0)))))))
(* 6 (* 5 (* 4 (* 3 (* 2 (* 1 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

4.1.2 Bonnes propriétés d'une définition récursive

Il est clair que toutes les programmations récursives ne donnent pas des exécutions finies. On conçoit aisément que l'exemple suivant est ridicule, même s'il repose sur une propriété qui n'est pas fausse ($n! = \frac{(n+1)!}{n+1}$).

$$\text{MauvaisFact}(n) = \begin{cases} 1 \textbf{ si } n = 0 \\ \frac{\text{MauvaisFact}(n+1)}{n+1} \textbf{ sinon} \end{cases}$$

En fait, à toute bonne définition récursive on peut associer un *ordre strict*[1] . Les divers cas récursifs de la définition doivent être conçus de manière à assurer, d'appels en appels, une diminution stricte du (des) paramètre(s) concerné(s).

D'appels récursifs en appels récursifs, la suite des paramètres concernés doit impérativement atteindre, en un nombre fini d'étapes, un des éléments

particuliers du domaine pour lequel le problème peut être résolu simplement.

Il s'agit en fait d'un cas particulier d'application de la démarche descendante: un problème est décomposé en sous-problèmes plus simples à résoudre. L'énoncé du problème ne changeant pas (f appelle f), le gain en simplicité est réalisé en changeant le (les) paramètre(s). L'ordre associé à une récursivité doit donc assurer une décroissance stricte de la difficulté du problème à résoudre.

Il est important de disposer d'une méthode assurant au concepteur un maximum de sécurité pour concevoir des fonctions récursives. La méthode que nous utilisons dans ce cours[1] comporte quatre étapes.

1. **Équation récursive générale.** En étudiant les propriétés du domaine de la fonction, on essaye de se placer dans une situation qu'on estime être « la plus générale possible ». On essaye alors d'exprimer une *équation récursive*. Il est clair que cette étape suppose, de la part du programmeur, un effort de créativité.

2. **Définition de l'ordre associé.** On définit un ordre sur le domaine de la fonction. Cet ordre doit posséder de bonnes propriétés.
   L'équation récursive étant de la forme f(x) = …f(s(x))… – on exprime en fait la solution du problème f(x) en utilisant un sous-problème[2] f(s(x)) –, il est clair que pour assurer la « décroissance de la difficulté », s(x) doit être plus petit (plus « simple ») que x pour cet ordre.

   De plus, les suites de terme général $s^i(x)$ ne doivent pas contenir de sous-suites infinies strictement décroissantes. L'ordre associé doit être ce que les mathématiciens appellent un *ordre bien fondé*[3]. Ce n'est pas le cas de celui qui est utilisé par MauvaisFact.

3. **Étude critique de l'équation récursive.** On étudie ensuite l'équation récursive de manière à repérer tous les cas où elle ne peut pas être appliquée[4].

---

1. Il existe des différences entre les terminologies anglo-saxonnes et françaises. Pour simplifier, (1) un ordre peut être strict ($x < y$) ou large ($x \leq y$) et, conformément au français courant, les ordres seront stricts par défaut. En français, (2) une relation d'ordre (antiréflexive et transitive) n'a pas à être totale: entre deux éléments on n'a pas forcément $x < y$ ou $y < x$ ou $x=y$. La relation est donc partielle par défaut.

1. La méthodologie que nous utilisons est due à Raymond DURAND de l'Université de Paris 6. Elle est présentée dans « *Conception des programmes applicatifs: méthodologie et transformations* », thèse d'État, Paris (1985). On peut également consulter, du même auteur, un polycopié de l' Institut de programmation de Paris 6: « *Programmation applicative* ».

2. Dans le cas général il peut y avoir plusieurs sous-problèmes f(s1(x)), f(s2(x))… Le lecteur adaptera sans difficulté la démarche.

3. Un *ordre bien fondé* est un ordre partiel tel qu'il n'existe aucune suite infinie strictement décroissante pour cet ordre.

4. L'équation récursive peut ne pas être applicable parce que la situation générale, comme l'indique son nom, ne couvre pas tous les cas. Il y a une autre raison de procéder à une étude critique systématique: les être humains font des erreurs…

Ces cas entrent dans l'une des trois catégories (non exclusives) suivantes:

- L'équation est mal typée (elle n'a donc pas de sens).
- La décroissance stricte des paramètres n'est pas assurée (on n'a aucune chance de résoudre un problème en le transformant en un problème exactement aussi difficile).
- L'équation donne un résultat faux sur une partie du domaine.

À l'issue de cette étape, on a déterminé un certain nombre de cas particuliers qui ne sont pas réglés par l'équation récursive. On doit alors porter un jugement de valeur sur la qualité du résultat. Ce jugement peut remettre en cause l'équation récursive et dans ce cas on repart à l'étape 1. Si, en revanche, le résultat est jugé sain, on passe à l'étape 4.

4. **Traitement des cas particuliers.** Pour chacun des cas particuliers recensés à l'étape précédente, on conçoit un algorithme approprié. Les difficultés rencontrées et la qualité des algorithmes obtenus pendant cette phase peuvent également conduire à une remise en cause du choix de départ (retour à l'étape 1).

Au terme de ces quatre étapes on a tous les éléments pour écrire un *algorithme récursif*, qu'il ne reste plus qu'à coder. Il est prudent de vérifier qu'on est dans les conditions d'une bonne étude par cas: les cas où l'équation s'applique et les cas particuliers retenus doivent donner une partition du domaine de définition, les valeurs associées à chaque élément de la partition doivent être calculables.

Nous résumons cette démarche.

Définition 4.2 Méthodologie de conception de fonctions récursives

Pour concevoir une fonction récursive on procède en quatre étapes:

1. Équation récursive générale
2. Définition de l'ordre associé
3. Étude critique de l'équation récursive: contrôle de type, décroissance stricte, validité sur l'intégralité du domaine
4. Traitement des cas particuliers

On peut alors exprimer l'algorithme.

4.1.3 Commentaires et précisions

4.1.3.1 Suite d'appels et ordre sur le domaine

Lors de l'étape 2, pour assurer de bonnes propriétés aux séquences d'appels de la fonction et s'abstraire des séquences particulières, on choisit de définir un ordre strict valide sur tout le domaine: il y a beaucoup moins de risques de se tromper. Pratiquement:

- On définit un ordre $\prec$ sur le domaine. Cet ordre peut être partiel; il ne doit pas exister de suites infinies strictement décroissantes pour cet ordre.
- On vérifie, sur l'équation récursive $f(x) = \ldots f(s(x)) \ldots$ que x et $s(x)$ sont comparables et que $s(x) \prec x$. Dans le cas où l'équation récursive fait apparaître plusieurs sous-problèmes $f(x) = \ldots f(s_1(x)) \ldots f(s_2(x)) \ldots$, on effectue cette vérification pour tous les $s_i(x)$.

Si ces conditions sont satisfaites, on a la certitude que l'équation récursive ne pas être utilisée une infinité de fois: toute suite d'appels est obligatoirement de longueur finie puisqu'elle est ordonnée selon un ordre strict bien fondé.

Exemples d'ordres bien fondés

1. Sur les entiers naturels, l'ordre habituel ($a < b$) est bien fondé; on remarquera que l'ordre $a > b$ ne l'est pas. Sur les entiers relatifs, aucun des deux ordres habituels n'est bien fondé.
2. Il est fréquent de s'intéresser à l'ensemble des couples de naturels. De nombreux ordres bien fondés peuvent être définis, ainsi les ordres suivants sont-ils bien fondés:

- $(a, b) \prec (c, d)$ si et seulement si $a < c$
- $(a, b) \prec (c, d)$ si et seulement si $b < d$
- $(a, b) \prec (c, d)$ si et seulement si $a + b < c + d$

4.1.3.2 Précision du typage

L'étape 3 joue un double rôle. C'est un garde-fou (on repère des oublis qui peuvent remettre en cause l'étape 1), mais c'est aussi un moyen de déterminer les cas particuliers. Nous insistons sur le contrôle du type de l'équation: si le problème a été spécifié avec précision, cette seule opération permet, en général, de repérer les cas d'arrêt.

Les débutants ont souvent le défaut de commencer par exprimer les cas d'arrêt de la récursivité (ils apparaissent comme simples). Cette approche n'est pas bonne: on se trompe souvent (oublis), on aboutit parfois à des expressions inutilement compliquées (cas d'arrêt pouvant être traités par l'équation récursive) et on risque de passer à côté de la logique de l'algorithme.

4.1.4 Exemple

Détaillons cette méthode d'analyse sur l'exemple de la fonction Fact.

Problème

Fact:	naturel	$\rightarrow$	naturel
	n	$\rightarrow$	n!

1. **Equation récursive**

   $\mathrm{Fact}(n) = n \times \mathrm{Fact}(n-1)$

2. **Ordre.** On choisit l'ordre habituel sur $\mathbf{N}$), qui possède les bonnes propriétés souhaitées. On remarque que si on avait pris le type entier comme domaine pour Fact, l'ordre aurait été rejeté.

3. **Étude critique[1].**

   Typage. Compte tenu du domaine de définition, l'équation est mal typée pour n=0 car, pour cette valeur, le paramètre de l'appel Fact(n - 1) n'est pas un naturel.

   Ordre. Il n'y a pas de cas de non-décroissance stricte.

   Validité. En dehors du cas n=0 (cas déjà repéré), l'équation est valide sur tout le domaine.

4. **Traitement du cas particulier.** Pour n=0, on sait que 0!=1.

Algorithme récursif

$$Fact(n) = \begin{cases} 1 \textbf{ si } n=0 \\ n \times Fact(n-1) \textbf{ si } n > 0 \end{cases}$$

Codage

```
;Fact:        naturel        ->      naturel
;             n              ->      n!
(define Fact (lambda (n)
      (cond
            ((= n 0)        1)
            ((> n 0)        (* n (Fact (- n 1)))))))
```

Contrôle de la validité des données

La spécification de la fonction peut être vue comme un contrat qui lie la personne qui implante l'algorithme de la fonction et la personne qui utilise cette fonction pour exprimer d'autres algorithmes. Une conséquence du respect de ce contrat est que, pour tout appel de la fonction, les paramètres effectifs sont forcément du type attendu.

Toutefois, il faut être conscient que des erreurs peuvent être commises et donc qu'il y a un risque d'appel incorrect (du point de vue du type). Comme Scheme n'assure pas de contrôle statique, deux cas peuvent se présenter: soit on se considère comme certain de la validité des appels qui peuvent être faits (on a effectué le contôle statique de type), soit on a des doutes et on adopte une stratégie (pompeusement appelée programmation défensive) qui consiste à vérifier au préalable l'appartenance au domaine de définition[2]. Dans le cas qui nous intéresse, l'appel de Fact est alors encapsulé par une autre fonction, ContrôleFact.

```
; Naturel? :  indifferent     ->      booleen
;             x               ->      x est un naturel
(define Naturel? (lambda (n)
      (cond
            ((integer? n)   (>= n 0)
            (#t             #f))))
```

1. Le lecteur doit être conscient de l'*impérieuse nécessité* de se poser les trois questions du typage, de la décroissance stricte des paramètres et de la validité sur l'ensemble du domaine. Par la suite, et pour alléger la rédaction, nous ne mentionnons que le résultat de l'étape 3, c'est à dire les cas particuliers repérés.

2. Dans d'autres langages un mécanisme (*traitement des exceptions*) fournit un moyen élégant et sûr pour contrôler ce type d'erreur.

```
; ControleFact :    indifferent    ->    naturel U { nil }
;                   n              ->    n! si n est un naturel, nil sinon
(define ContrôleFact (lambda (n)
      (cond
            ((Naturel? n)          (Fact n))
            (#t                    nil))))
```

Dans le cadre de ce cours, nous nous plaçons systématiquement dans le cas où le contrôle de la validité des appels est statiquement fait (le contrat de la spécification est respecté par les deux parties). Le style défensif n'a donc pas à être utilisé[1].

4.2 Caractères et mots

Parmi les objets Scheme que nous avons rencontrés seuls les entiers naturels se prêtent naturellement à la récursivité dans la mesure où le type naturel est facilement muni d'un ordre bien fondé. La fonction RacIter, qui travaille sur les réels est bien récursive et il est possible, quoique difficile, d'étudier un ordre associé sur les valeurs prises par les séquences d'appel. Il se trouve que, d'un point de vue général, l'ordre habituel sur R n'est pas un ordre bien fondé. De plus il n'y a pas de bon ordre[2] pratiquement utilisable sur les réels: c'est une des raisons qui font que la programmation sur les réels est une discipline difficile.

L'arithmétique est un domaine où le débutant manque parfois d'intuition, aussi, pour étudier plus avant la récursivité, nous introduisons de nouveaux objets, les caractères (le type ascii), et les mots (le type mot)[3].

4.2.1 Le type ascii

La machine Scheme permet de manipuler des objets, les caractères d'un alphabet international (l'alphabet Ascii[4]) qui correspondent aux touches du clavier et à un certain nombre de caractères spéciaux.

Notation de caractères

- Le caractère x est désigné par #\x
- Certains caractères, dits spéciaux, comme l'espace, le saut de ligne ou la marque de tabulation sont désignés respectivement par les notations #\space, #\newline et #\tab.

1. Lorsqu'on fait des entrées-sorties, le problème n'est plus contrôlable statiquement (il est même dangereux de supposer que les utilisateurs qui effectuent les entrées ne font pas d'erreur); notons que dans le cadre de ce cours nous ne nous intéressons pas aux entrées-sorties.

2. Un bon ordre est un ordre bien fondé total.

3. Ces deux types sont des versions rennaises des types Scheme *char* et *string*.

4. L'alphabet ascii est muni d'un ordre respectant l'ordre alphabétique usuel.

Fonctions de manipulation de caractères

Ascii?:	indifférent	→	booléen
	détermine si un objet est un ascii		
>Ascii?:	ascii x ascii	→	booléen
	c1, c2	→	*c1 est situé après c2 dans l'ordre standard des caractères ascii*

; on dispose également d'autres fonctions de comparaison (préfixes >=, <, <=)

MinAscii :	ascii		
	le plus petit des caractères ascii		
MaxAscii :	ascii		
	le plus grand des caractères ascii		
PredAscii :	ascii – {MinAscii}	→	ascii – {MaxAscii}
	c	→	*le prédécesseur immédiat de c*
SuccAscii :	ascii – {MaxAscii}	→	ascii – {MinAscii}
	c	→	*le successeur immédiat de c*

Remarque
Il est important de comprendre que le caractère #\1 et l'entier 1 sont deux objets différents.

4.2.2 Le type mot

Un mot[1] est une suite finie de caractères, considérée comme un tout (un seul objet). On appelle longueur (ou taille) d'un mot le nombre de caractères qui le composent. On note souvent la longueur de m par | m |.

Notation pour les mots

- La suite ne comportant aucun caractère est appelée mot vide: on note le mot vide par "".
- La suite des trois caractères #\a, #\b et #\c est désignée par "abc".
- Le type mot - {""} sera noté mot+.

Fonctions de manipulation de mots[2]

Mot? :	indifférent	→	booléen
	détermine si un objet est un mot		
MotVide? :	mot	→	booléen
	détermine si un mot est le mot vide		
AjoutDroit :	mot x ascii	→	mot+
	m, c	→	*un mot de longueur	m	+ 1, obtenu en ajoutant à la fin de m le caractère c*
CarDroit :	mot+	→	ascii
	m	→	*le dernier caractère de m*
SaufDroit :	mot+	→	mot
	m	→	*un mot de longueur	m	- 1, obtenu en amputant m de son dernier caractère*

1. En théorie des langages, un mot est une suite finie d'éléments d'un ensemble V appelé vocabulaire. L'ensemble des mots sur V est appelé monoïde libre engendré par V et on le note d'ordinaire par V*. L'ensemble des mots que nous étudions dans ce chapitre est donc ascii*.

2. Les opérations que nous avons retenues relèvent d'un choix pédagogique (on cherche à étudier la récursivité) et non de la volonté de définir judicieusement les opérations fondamentales sur les mots.

AjoutGauche :	ascii x mot	→	mot+
	c, m	→	*un mot de longueur	m	+ 1, obtenu en ajoutant au début de m le caractère c*
CarGauche :	mot+	→	ascii
	m	→	*le premier caractère de m*
SaufGauche :	mot+	→	mot
	m	→	*un mot de longueur	m	- 1, obtenu en amputant m de son premier caractère*

Remarque
On remarquera que les objets #\a, "a" et a (identificateur) sont différents.

Définition 4.3 Caractères et mots

Le type ascii recouvre un ensemble *fini* d'éléments: les *caractères* de l'alphabet international Ascii.

Le type mot recouvre l'ensemble des suites finies de caractères ascii.

Notations
- #\a est une notation de caractère ascii.
- "mot" et "" sont des notations de mots.
- une notation de caractère ou de mot est une expression Scheme.

4.3 Programmer sur des mots

4.3.1 Récursivité gauche ou droite

Le type mot se prête, d'une manière naturelle, à deux modes principaux d'utilisation de la récursivité:

- récursivité à droite: on passe de f(x) à f(SaufGauche(x));
- récursivité à gauche: on passe de f(x) à f(SaufDroit(x)).

Dans les deux cas, l'ordre associé est fondé sur la taille des mots.

Nous illustrons ces deux modes sur une des opérations du type mot, la fonction Coller.

4.3.1.1 Concaténation

Définition 4.4 Concaténation

Concaténer deux mots consiste à construire le mot qui contient les caractères du premier suivis des caractères du second.

Cette opération est associative et admet un élément neutre, le mot vide[a].

a. Un ensemble muni d'une loi de composition interne associative qui possède un élément neutre est appelé monoïde. L'ensemble V* des mots sur un vocabulaire V est appelé monoïde libre. Le rôle fondamental de la concaténation sur le type mot mériterait probablement d'en faire une opération primitive du type.

Nous cherchons à programmer la fonction:

Problème

Coller :	mot x mot	→	mot
	m, n	→	la concaténation de m et de n

Nous choisissons d'essayer de faire porter la récursivité sur le premier paramètre et nous étudions deux algorithmes.

4.3.1.2 Coller, version 1: récursivité à droite

Analyse

1. Il s'agit donc d'exprimer Coller(m, ...) en fonction de Coller(SaufGauche(m), ...):

Coller(m, n) = AjoutGauche(CarGauche(m), Coller(SaufGauche(m), n))

2. Il s'agit de munir le domaine d'un ordre bien fondé $\prec$ tel que:

- (Saufgauche(m), n) $\prec$ (m, n)

Pour ce faire, on peut opter pour:

- (m', n') $\prec$ (m, n) si et seulement si | m' | < | m |

On constate que, dans ce cas particulier, le paramètre n n'intervient pas. On peut imaginer qu'il est fréquent qu'un certain nombre des paramètres de la fonction n'interviennent pas sur la récursivité. On se contente alors d'exprimer l'ordre sur les paramètres utiles. D'où, pour notre équation récursive:

- m' $\prec$ m si et seulement si | m' | < | m |

Ce dernier ordre basé sur la taille des mots est d'un usage fréquent; nous l'appelons *ordre habituel sur les mots.*

3. L'équation récursive est mal typée pour m = "" car CarGauche n'est pas défini sur le mot vide.

4. Coller("", n) = n

Algorithme

$$\text{Coller}(m, n) = \begin{cases} n \textbf{ si } \text{MotVide?}(m) \\ \text{AjoutGauche}(\text{CarGauche}(m), \text{Coller}(\text{SaufGauche}(m), n)) \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{sinon} \end{cases}$$

4.3.1.3 Coller, version 2: récursivité à gauche

Analyse

1. On cherche donc une équation récursive utilisant Coller(SaufDroit(m), ...):

$$Coller(m, n) = Coller(SaufDroit(m), AjoutGauche(CarDroit(m), n))$$

2. On adopte le même ordre que précédemment:

- $(m', n') \lhd (m, n)$ si et seulement si $|m'| < |m|$

et on constate que cet ordre convient:

- $(SaufDroit(m), AjoutGauche(CarDroit(m), n)) \lhd (m, n)$

3. L'équation récursive est mal typée pour m = ""

4. Coller("", n) = n

Algorithme

$$Coller(m, n) = \begin{cases} n & \textbf{si } MotVide?(m) \\ Coller(SaufDroit(m), AjoutGauche(CarDroit(m), n)) & \textbf{sinon} \end{cases}$$

4.3.1.4 Récursivité à gauche et à droite

On peut combiner les deux types de récursivité.

Un *palindrome* est un mot qui se lit de la même manière de gauche à droite ou de droite à gauche. Ainsi, été et laval sont des palindromes.

Problème

Palindrome?:	mot	$\rightarrow$	booléen
	m	$\rightarrow$	m est un palindrome

Analyse

1. Il est clair que si on ampute un palindrome de ses deux extrémités on obtient un palindrome. D'où l'équation récursive:

$$Palindrome?(m) = \begin{cases} faux & \textbf{si } CarGauche(m) \neq CarDroit(m) \\ Palindrome?(SaufDroit(SaufGauche(m))) & \textbf{sinon} \end{cases}$$

On remarque qu'une équation récursive peut résulter d'une étude par cas[1].

2. On utilise l'ordre habituel sur les mots.

3. L'équation récursive est mal typée:

- sur le mot vide (utilisation de CarDroit(m), de SaufGauche(m)...)
- sur tous les mots u de longueur 1: SaufGauche(u) étant vide, SaufDroit(SaufGauche(u)) n'est pas défini.

1. Dans ce cas particulier, une équation récursive plus élégante ne résulte pas d'une étude pas cas mais procède directement de la définition de ce qu'est un palindrome:

$$Palindrome?(m) = ((CarGauche(m) = CarDroit(m)) \wedge Palindrome?(SaufDroit(SaufGauche(m))))$$

4. Un mot de longueur 0 ou 1 est un palindrome. Pour déterminer si un mot est de longueur 1, on peut utiliser le test MotVide?(SaufGauche(m)). Ce test n'étant pas défini pour m = "", il faut s'assurer qu'il est correctement appliqué.

Algorithme

$$\text{Palindrome?}(m) = \begin{cases} \text{vrai } \textbf{si } \text{MotVide?}(m) \\ \text{vrai } \textbf{si } \text{MotVide?}(\text{SaufGauche}(m)) \\ \textbf{sinon} \begin{cases} \text{faux } \textbf{si } \text{CarGauche}(m) \neq \text{CarDroit}(m) \\ \text{Palindrome?}(\text{SaufDroit}(\text{SaufGauche}(m))) \\ \qquad\qquad \textbf{sinon} \end{cases} \end{cases}$$

4.3.2 Choix des paramètres de récursivité

Pour la fonction Coller, nous avons choisi de développer la récursivité sur le premier paramètre. Le lecteur pourra se convaincre qu'on peut la faire porter sur le second (on obtient deux algorithmes symétriques).

La seconde version de Coller a cependant une caractéristique qu'il convient d'examiner: les deux paramètres varient d'appels en appels et le second augmente en taille! Le lecteur pourra se convaincre qu'on peut faire abstraction du second paramètre (l'ordre $(m', n') \lhd (m, n)$ si et seulement si $|m'| < |m|$ convient). Cette constatation justifie l'appellation « récursivité fondée sur le premier paramètre ».

Toutefois, il existe des cas où plusieurs paramètres doivent varier de manière conjointe.

On dit qu'un mot m est préfixe d'un mot n si et seulement s'il existe un mot a tel que Coller(m, a) = n.

Problème

Préfixe?:	mot x mot	→	booléen
	m, n	→	m est un préfixe de n

Analyse

1. Il est clair que si on ampute deux mots dont le premier est un préfixe du second de leur premier caractère on obtient deux mots dont le premier est toujours un préfixe du second. D'où l'équation récursive:

$$\text{Préfixe?}(m, n) = \begin{cases} \text{faux } \textbf{si } \text{CarGauche}(m) \neq \text{CarGauche}(n) \\ \text{Préfixe?}(\text{SaufGauche}(m), \text{SaufGauche}(n)) \\ \qquad\qquad \textbf{sinon} \end{cases}$$

2. On peut utiliser l'ordre partiel $(m', n') \lhd (m, n)$ si et seulement si $|m'| < |m|$ et $|n| < |n'|$.

3. L'équation récursive est mal typée pour les mots de longueur 0.

4. Préfixe?("", n) = vrai et Préfixe?(m, "") = MotVide?(m)

Algorithme

$$\text{Préfixe?}(m, n) = \begin{cases} \text{vrai } \textbf{si } \text{MotVide?}(m) \\ \text{MotVide?}(m) \textbf{ si } \text{MotVide?}(n) \\ \textbf{sinon} \begin{cases} \text{faux } \textbf{si } \text{CarGauche}(m) \neq \text{CarGauche}(n) \\ \text{Préfixe?}(\text{SaufGauche}(m), \text{SaufGauche}(n)) \\ \qquad\qquad \textbf{sinon} \end{cases} \end{cases}$$

4.3.3 Calculs

Les algorithmes que nous venons d'étudier n'utilisent que la structure des mots. Il est clair qu'on peut réaliser des calculs sur des objets qu'on élabore. Le lecteur pourra s'exercer avec profit à concevoir des fonction simples comme le calcul de la longueur d'un mot, du nombre d'occurrences d'un caractère donné dans un mot…

L'exemple que nous traitons est un schéma d'algorithme assez général dit schéma de HORNER.

4.3.3.1 Schéma de HORNER

Un mot qui ne contient que les caractères #\0 et #\1 peut être considéré comme l'écriture d'un entier naturel en base 2. Ainsi:

- "1011" représente l'entier naturel noté 11 en décimal
- "100" représente l'entier naturel noté 4 en décimal
- "0" représente l'entier naturel noté 0 en décimal

Nous donnons des noms aux types qui nous intéressent:

- chiffre est le type {#\0, #\1}
- b2 est le type des mots qui ne contiennent que des caractères du type chiffre; on convient que b2 ne contient pas le mot vide.

Nous utilisons une fonction CarNat dont la spécification et l'algorithme sont:

CarNat:	chiffre	→	naturel
	c	→	le naturel dont la représentation en base 2 ne comporte qu'un chiffre, le chiffre c

$$\text{CarNat}(c) = \begin{cases} 0 \textbf{ si } c = \#\backslash 0 \\ 1 \textbf{ si } c = \#\backslash 1 \end{cases}$$

Problème

B2Nat:	b2	→	naturel
	m	→	le naturel dont une représentation en base 2 est m

Analyse

1. L'équation récursive qui suit est appelée Schéma de HORNER.

$$\text{B2Nat}(m) = \text{B2Nat}(\text{SaufDroit}(m)) \times 2 + \text{CarNat}(\text{CarDroit}(m))$$

2. L'ordre est fondé sur la taille des mots

3. L'équation récursive est mal typée pour les mots de longueur 1 puisque B2Nat ne peut admettre le mot vide en paramètre.

4. B2Nat(m) = CarNat(CarDroit(m)) si | m | = 1

Algorithme

$$B2Nat(m) = \begin{cases} CarNat(CarDroit(m)) \textbf{ si } MotVide?(SaufDroit(m)) \\ B2Nat(SaufDroit(m)) \times 2 + CarNat(CarDroit(m)) \textbf{ sinon} \end{cases}$$

4.3.3.2 Calculs de mots

On peut se poser le problème dans l'autre sens.

Problème

NatB2:	naturel	→	b2
	n	→	la représentation en base 2, sans zéros inutiles, de n

Analyse

1. Appelons Mul2, Div2 et Mod2 des fonctions qui, respectivement, multiplie un entier par 2, le divise par 2 et calcule le reste de sa division par 2.

Trivialement:

n = Mul2(Div2(n)) + Mod2(n)

Cette relation est bien entendu vraie même si on travaille en binaire. En utilisant une fonction d'addition AddBin, qui a pour type b2 x chiffre → b2, et en considérant que Mul2 et Div2 sont du type b2 → b2 et Mod2 du type b2 → chiffre, on peut alors écrire:

NatB2(n) = AddBin(Mul2(Div2(NatB2(n))), Mod2(NatB2(n)))

En posant:

EntCar:	{0, 1}	→	chiffre
	n	→	la représentation en base 2 de n

On constate que:

Mod2(NatB2(n)) = EntCar(n **modulo** 2)

Div2(NatB2(n)) = NatB2(n **quotient** 2)

AddBin(Mul2(m), c) = AjoutDroit(m, c)

On peut en déduire une équation récursive pour NatB2.

NatB2(n) = AjoutDroit(NatB2(n **quotient** 2), EntCar(n **modulo** 2))

2. L'ordre est l'ordre habituel sur **N**.

3. L'équation récursive est toujours bien typée. En revanche, la décroissance stricte n'est pas assurée pour n = 0 et l'équation donne un résultat incorrect (avec un zéro non significatif) pour n = 1.

4. NatB2(0) = "0" et NatB2(1) = "1"

Algorithme

$$\text{NatB2}(n) = \begin{cases} \text{"0" } \textbf{si } n = 0 \\ \text{"1" } \textbf{si } n = 1 \\ \text{AjoutDroit}(\text{NatB2}(n \textbf{ quotient } 2), \text{EntCar}(n \textbf{ modulo } 2)) \\ \quad \textbf{sinon} \end{cases}$$

4.3.4 Mots et ordres

L'ordre qu'on utilise le plus fréquemment sur les mots est l'ordre fondé sur leurs tailles. Remarquons toutefois que cet ordre, que nous qualifions d'*ordinaire,* est partiel ("a" et "b", par exemple, sont incomparables).

Il est possible de munir l'ensemble des mots d'autres ordres. Nous présentons dans ce paragraphe deux ordres totaux, l'ordre *alphabétique* (ou *lexicographique*) et un ordre que nous appelons *standard*. Ces ordres sont fréquemment utilisés (en général à d'autres fins que la conception de fonctions récursives) en informatique.

Ces deux ordres utilisent le fait que le type ascii est muni d'un ordre (grâce à la fonction <=Ascii?), qui, sur les caractères de l'alphabet romain respecte l'ordre alphabétique habituel.

L'ordre lexicographique est défini de la manière suivante:

- "" ≤ m, quel que soit m
- u ≤ v si CarGauche(u) ≤ CarGauche(v)
- u ≤ v si CarGauche(u) = CarGauche(v) et SaufGauche(u) ≤ SaufGauche(v)

Il se trouve que l'ordre lexicographique, n'est pas bien fondé.

L'ordre standard est une extension de l'ordre ordinaire (qui, rappelons-le, n'est que partiel): pour comparer deux mots de la même taille, on utilise l'ordre lexicographique.

4.3.5 Pour terminer

La méthodologie de conception de fonction récursive ne doit pas faire oublier la méthodologie générale vue au chapitre précédent. Nous traitons donc un exemple plus complet.

Problème

Un mot v est une anagramme de u si le mot v contient exactement les mêmes caractères que u (en même nombre): "barre" est une anagramme de "arbre".

Anagramme?:	mot x mot	→	booléen
	u, v	→	*v est une anagramme de u*

Il est facile de se rendre compte que v est une anagramme de u si le premier caractère c de u est présent dans v et si le mot obtenu en enlevant à v une occurrence du caractère c est une anagramme du mot droit de u.

Cette constatation conduit à introduire deux sous-problèmes:

Présent?:	ascii x mot	→	booléen
	c, u	→	*c est un des caractères de u*
MoinsUneOcc :	ascii x mot	→	mot
	c, u	→	*u amputé d'une occurrence de c,* ***sachant que u contient le caractère c***

Le lecteur analysera la solution que nous proposons.

Algorithmes

$$\text{Anagramme?}(u, v) = \begin{cases} \text{MotVide?}(v) \textbf{ si } \text{MotVide?}(u) \\ \textbf{sinon} \begin{cases} \text{Anagramme?}(\text{SaufGauche}(u), \text{MoinsUneOcc}(\text{CarGauche}(u), v)) \\ \quad \textbf{si } \text{Présent?}(\text{CarGauche}(u), v) \\ \text{faux } \textbf{sinon} \end{cases} \end{cases}$$

$$\text{MoinsUneOcc}(c, u) = \begin{cases} \text{SaufGauche}(u) \textbf{ si } \text{CarGauche}(u) = c \\ \text{AjoutGauche}(\text{CarGauche}(u), \text{MoinsUneOcc}(c, \text{SaufGauche}(u))) \textbf{ sinon} \end{cases}$$

$$\text{Présent?}(c, u) = \begin{cases} \text{faux } \textbf{si } \text{MotVide?}(u) \\ \textbf{sinon} \begin{cases} \text{vrai } \textbf{si } \text{CarGauche}(u) = c \\ \text{Présent?}(c, \text{SaufGauche}(u)) \textbf{ sinon} \end{cases} \end{cases}$$

Compte tenu d'une certaine similitude entre MoinsUneOcc et Présent?, le lecteur est invité à chercher un autre découpage fonctionnel dans lequel une seule fonction fait l'ensemble du travail.

4.4 Récursivité arithmétique

L'ensemble des entiers naturels étant muni, de manière standard, d'un bon ordre, on peut appliquer la méthodologie que nous préconisons. Toutefois, l'arithmétique étant une discipline difficile, cela demande un peu d'expérience dès qu'on cherche à utiliser d'autres ordres.

4.4.1 Un problème simple

Nous cherchons à déterminer si un naturel strictement positif a est un diviseur d'un autre naturel strictement positif b.

Problème

Div?:	naturel+ x naturel+	→	booléen
	a,b	→	a est un diviseur de b

1. **Equation récursive.** On peut exploiter une propriété simple de la divisibilité:

   $\text{Div?}(a, b) = \text{Div?}(a, b - a)$

2. **Ordre.** La récursivité ne porte que sur le second paramètre et l'ordre associé est l'ordre habituel sur les naturels.
3. **Étude critique.** L'équation récursive est mal typée si b - a n'est pas un naturel strictement positif, c'est-à-dire si $a \geq b$.

   Compte-tenu du fait que a est strictement positif, la décroissance est bien stricte. On remarquera que, si on avait opté pour une spécification erronée (ou incomplète) en prenant un naturel pour a, on aurait repéré cette lacune en découvrant le caractère non strict de la diminution des paramètres. On insiste sur le fait que l'étape d'étude critique permet, éventuellement, de « mettre le doigt » sur une lacune de la spécification et donc d'éviter une erreur.
4. **Cas particulier.** Quand a est supérieur ou égal à b, a divise b si et seulement si a = b.

D'où :

Algorithme récursif

$$Div?(a, b) = \begin{cases} a = b \textbf{ si } a \geq b \\ \textbf{sinon } Div?(a, b - a) \end{cases}$$

4.4.2 Un ordre plus compliqué

Problème

Pgcd : naturel$^+$ x naturel$^+$ → naturel$^+$
a,b → le pgcd de a et b

1. **Equation récursive.** On sait que :

$$Pgcd(a, b) = \begin{cases} Pgcd(a\text{-}b, b) \textbf{ si } (b < a) \\ Pgcd(a, b\text{-}a) \textbf{ si } (a < b) \end{cases}$$

2. **Ordre.** On peut utiliser l'ordre :
   (a', b') ≺ (a, b) si et seulement si a'+b' < a+b
3. **Étude critique.** L'équation récursive n'est pas valide sur l'intégralité du domaine, le cas a = b n'étant pas pris en compte.
4. **Cas particulier.** On sait que Pgcd(a, a)=a.

D'où :

Algorithme récursif

$$Pgcd(a, b) = \begin{cases} a \textbf{ si } a=b \\ \textbf{sinon} \begin{cases} Pgcd(a\text{-}b, b) \textbf{ si } (b < a) \\ Pgcd(a, b\text{-}a) \textbf{ si } (a < b) \end{cases} \end{cases}$$

4.4.3 Une équation récursive curieuse

Les tests de primalité font encore aujourd'hui l'objet de recherches. L'algorithme que nous proposons est le plus naïf qui soit (son coût est loin d'être

raisonnable). Il nous permet toutefois d'introduire une fonction, PlgDivDans, dont l'analyse est curieuse.

Problème

PlgDivDans : naturel+ x naturel+ → naturel+
a, n → le plus grand diviseur de a dans {1,..., n}

1. **Equation récursive.** On exploite une propriété évidente: si le plus grand diviseur de a dans {1,..., n} n'est pas n, il est dans {1,..., n - 1}.

$$PlgDivDans(a, n) = \begin{cases} n \textbf{ si } Div?(n, a) \\ PlgDivDans(a, n-1) \textbf{ sinon} \end{cases}$$

2. **Ordre.** La récursivité ne porte que sur le second paramètre et l'ordre associé est l'ordre habituel sur les naturels.
3. **Étude critique.** Cette équation récursive est particulière: pour qu'elle soit mal typée il faut que n - 1 ne soit pas un naturel strictement positif, donc que n soit égal à 1. On peut facilement montrer que dans ce cas, l'appel récursif n'a pas lieu car Div?(1, a) est toujours vrai.
4. **Cas particulier.** Il n'y a donc pas de cas particulier, D'où:

Algorithme récursif

$$PlgDivdans(a, n) = \begin{cases} n \textbf{ si } Div?(n, a) \\ PlgDivDans(a, n-1) \textbf{ sinon} \end{cases}$$

On peut donc donner un algorithme simple, et brutal, de test de primalité:

Test de primalité

Premier? : naturel → booléen
n → *n est premier*

$$Premier?(n) = (PlgDivDans(n, n-1) = 1)$$

Il est facile d'améliorer cet algorithme en tirant parti du fait que si n n'est pas premier il admet un diviseur (autre que 1) inférieur ou égal à $\sqrt{n}$.

4.4.4 Une équation récursive non valide sur l'intégralité du domaine

Nous nous intéressons à la représentation en base 10 d'un entier naturel en nous interdisant, cependant, de passer par un calcul explicite de cette représentation.

Problème

NbChB10 : naturel → naturel+
n → *le nombre de chiffres de la représentation (sans zéro superflu) en base 10 de n*

1. **Equation récursive.** En divisant n par 10, on peut espérer obtenir un nombre dont la représentation en base 10 compte un chiffre de moins.
   $NbChB10(n) = 1 + NbChB10(quotient(n, 10))$
2. **Ordre.** Ordre habituel sur les naturels.
3. **Étude critique.** Cette équation récursive est toujours bien typée.

Cependant, la décroissance n'est plus stricte pour $n = 0$.

À ce stade, on serait tenté d'écrire l'algorithme:

$$\text{NbChB10}(n) = \begin{cases} 1 \textbf{ si } n = 0 \\ 1 + \text{NbChB10}(\text{quotient}(n, 10)) \textbf{ sinon} \end{cases}$$

Cet algorithme est faux (il suffit de l'essayer pour $n = 1$): le troisième point de l'étape de validation consiste justement à se poser la question de la validité, sur l'ensemble du domaine, de l'équation récursive. Celle-ci est fausse pour $n < 10$.

4. **Cas particulier.** Les entiers inférieurs à 10 s'écrivent, en décimal, avec un seul chiffre. D'où:

Algorithme récursif

$$\text{NbChB10}(n) = \begin{cases} 1 \textbf{ si } n < 10 \\ 1 + \text{NbChB10}(\text{quotient}(n, 10)) \textbf{ sinon} \end{cases}$$

4.4.5 Cas pathologiques

La méthode que nous préconisons suppose de munir le domaine d'un ordre bien fondé. Il faut cependant être conscient que cette méthode est une « condition suffisante » pour concevoir des fonctions qui « marchent ». Il existe des problèmes pour lesquels elle n'est pas pratiquement applicable: l'algorithmique sur les réels, par exemple, suppose d'autres méthodes, plus complexes.

Même si on se limite à l'arithmétique, on peut exhiber des fonctions:

- pour lesquelles il est difficile d'exhiber un ordre bien fondé,
- pour lesquelles la recherche d'un ordre bien fondé est encore un problème ouvert.

Nous citons deux exemples d'école:

Fonction dite de McCarthy

$$\text{F91}(x) = \begin{cases} x - 10 \textbf{ si } x > 100 \\ \text{F91}(\text{F91}(x + 11)) \textbf{ sinon} \end{cases}$$

Cette fonction délivre 91 pour $x < 102$. Il n'est pas évident de s'en convaincre…

Fonction dite de Collatz

$$\text{F1}(x) = \begin{cases} 1 \textbf{ si } x = 0 \\ 1 \textbf{ si } x = 1 \\ \textbf{sinon} \begin{cases} \text{F1}(x \textbf{ div } 2) \textbf{ si } x \textbf{ est pair} \\ \text{F1}(3x + 1) \textbf{ sinon} \end{cases} \end{cases}$$

Cette fonction, si elle calcule quelque chose, ne peut délivrer que 1. Il semble qu'elle se termine toujours (donc qu'elle vaut 1 partout) mais ce résultat n'est toujours pas, à notre connaissance, établi.

4.5 Conclusions

En fait, concevoir un algorithme récursif est un cas particulier d'un autre dogme méthodologique:

- On exprime la solution d'un problème P en combinant les solutions de sous-problèmes plus simples P_1, ..., P_n
- On traite chacun des P_i de la même manière jusqu'à l'obtention de sous-problèmes que l'on sait résoudre facilement.

Dans le cas récursif, P = f(x) et l'un (au moins) des P_i est f(s(x)). Pour que la méthode marche, il suffit que s(x) soit strictement plus simple que x (dans un sens que la méthode que nous utilisons explicite), ce qui peut être noté par $s(x) \prec x$.

On retient que la méthode que nous préconisons comporte quatre étapes:

Méthodologie générale de conception de fonctions récursives

1. Étude des propriétés du domaine d'application permettant d'écrire une *équation récursive*.
2. Étude de l'ordre associé.
3. Détermination des cas particuliers (éléments sur lesquels l'équation ne peut plus s'appliquer).
4. Étude des traitements à appliquer pour les cas particuliers.

4.6 Exercices

4.6.1 Calculs sur les mots

Exercice 4.1 Comptages

1. Concevoir une fonction qui calcule la longueur d'un mot.
2. Concevoir une fonction qui détermine si un caractère est présent dans un mot.
3. Concevoir une fonction qui compte le nombre d'occurrences d'un caractère dans un mot.
4. Sachant que la fonction Nba (resp. Nbb) donne le nombre d'occurrences du caractère \#a (resp. \#b) dans un mot donné, concevoir une fonction qui calcule, pour un mot m, Nba(m) - Nbb(m). On cherchera un algorithme « direct » (qui n'utilise pas explicitement Nba et Nbb et qui ne parcourt qu'une fois le mot m).

Exercice 4.2 Facteurs

Étant donné un mot m, on dit que le mot w est un facteur de m si il existe deux mots u et v tels que m = Coller(u, Coller(w, v)).

1. Concevoir une fonction qui, à partir de deux mots, détermine si le premier est un facteur du second.
2. Concevoir une fonction qui, à partir de deux mots m et w, calcule le nombre d'occurrences du facteur w dans m.

Exercice 4.3 Réarrangements de caractères

Le mot miroir d'un mot m est le mot obtenu en lisant m de droite à gauche.

1. Concevoir une fonction qui calcule le mot miroir d'un mot donné.

Étant donné un mot m non vide de longueur k inférieure à 10, un mot p de longueur k et contenant tous les caractères \#1 à \#k est appelé *permutation des indices* de m. Par exemple, "25341" est une permutation des indices de "abcde". À partir d'un mot et d'une permutation de ses indices on peut définir un mot appelé permutation de m: "becda" est la permutation de "abcde" obtenue à partir de "25341".

2. Concevoir une fonction qui, à partir d'un mot et d'une permutation de ses indices, calcule la permutation associée.

Exercice 4.4 Textes

On appelle texte un mot n'utilisant que des lettres, le caractère \#, (virgule) le caractère \#. (point) et le caractère #\space (blanc). Toute suite de blancs est appelée espace.

1. Concevoir une fonction qui, à partir d'un texte calcule un texte où tous les espaces sont réduits à un seul caractère.

Un texte sera dit normal s'il ne commence pas par un espace, s'il ne se termine pas par un espace et si tous ses espaces sont réduits à un seul caractère.

2. Concevoir une fonction qui normalise un texte.

Un texte sera dit typographiquement correct s'il est normal et s'il respecte les conventions typographiques françaises d'usage du point et de la virgule.

3. Concevoir une fonction qui rend un texte typographiquement correct.

4.6.2 Récursivités arithmétiques

Exercice 4.5 Combinaisons

On cherche à définir une fonction calculant C_n^k, n et k étant fournis, à partir des relations

$$C_n^0 = 1$$

$$C_n^n = 1$$

$$C_n^k = \frac{n}{k} \times C_{n-1}^{k-1} \quad \textbf{si} \quad (k \neq 0) \textbf{ et } (n \neq k)$$

Exercice 4.6 Puissance

Écrire une fonction Puissance qui calcule la puissance $n^{ième}$ de l'entier x, *x* et n étant fournis en paramètres.

Exercice 4.7 Division entière et modulo

Écrire, à l'aide de l'opérateur de soustraction, deux fonctions qui calculent respectivement le quotient et le reste de la division entière de deux entiers positifs.

Exercice 4.8 Calcul de la somme des entiers de 1 à n

Écrire une fonction qui, étant donné un nombre n, calcule la somme des n premiers entiers naturels.

Exercice 4.9 Suite de FIBONACCI

La suite définie par:

$u_0 = 0$
$u_1 = 1$
$u_i = u_{i-1} + u_{i-2}$ pour $i \geq 2$

est appelée suite de FIBONACCI. Écrire une fonction qui, à partir de n, calcule u_n.

Exercice 4.10 Somme de carrés

- Écrire une fonction DifCarrés? qui, à partir de deux entiers a et c, a < c, délivre vrai si et seulement si il existe un entier b entre a et c tel que $c^2 = a^2 + b^2$.
- Écrire, en utilisant éventuellement la fonction précédente, une fonction SomCarrés? qui, à partir d'un entier c cherche s'il existe deux entiers strictement positifs a et b tels que $c^2 = a^2 + b^2$.

4.6.3 Divers

Exercice 4.11 *It's a wild world*

Une prairie est peuplée de m moutons, l loups et s serpents dont la morsure est mortelle. La population évolue de la manière suivante:

- Le matin, chaque loup mange un mouton.
- Le midi, chaque serpent mord un loup.
- Le soir, chaque mouton écrase un serpent.

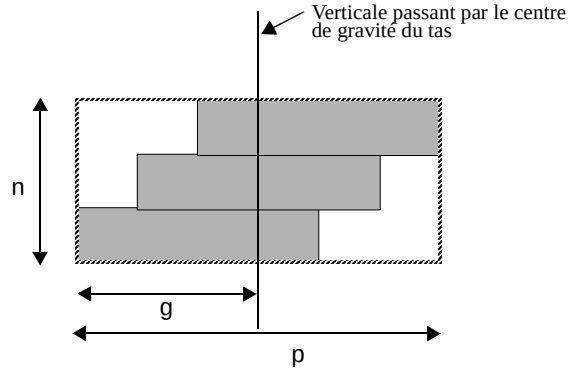
Les fonctions demandées prennent pour paramètres la population initiale: les nombres m, l et s.

1. Concevoir une fonction qui détermine l'espèce qui s'éteint en premier.

2. Concevoir une fonction qui détermine la date de l'extinction de la première espèce.

Exercice 4.12 *Golden bridge*

On considère un tas de n briques identiques de longueur 1, posées en équilibre les unes sur les autres. Un tas est caractérisé par trois grandeurs comme indiqué sur le dessin:



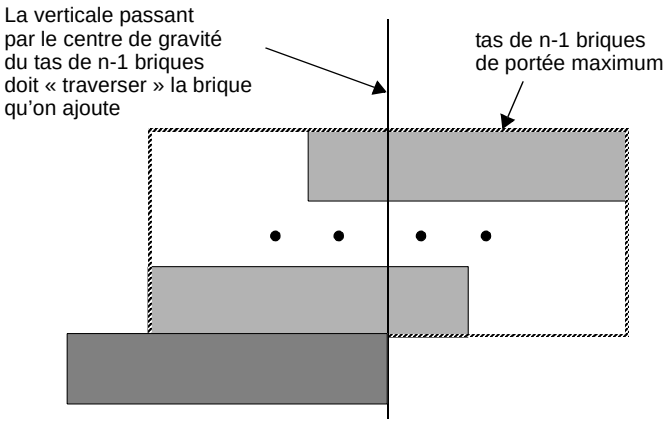
On appelle:

- n, le nombre de briques,
- p, la portée du tas,
- g, la projection du tas.

On mesure g et p à partir de l'extrémité gauche de la brique posée sur le sol.

Pour construire un tas de n briques en équilibre ayant une portée maximum, il suffit de procéder de la manière suivante: on prend un tas de n-1

briques ayant une portée maximum et on le pose sur 1 brique, en le décalant le plus possible vers la droite.



- Concevoir et rédiger une fonction qui, à partir d'un nombre de briques n calcule la portée maximum d'un tas.

CHAPITRE 5

Données, types et abstraction

5.1 Données et types

5.1.1 Ensembles et types

On rappelle que la notion de *type* permet de regrouper les objets en classes ayant des propriétés communes et cette notion permet d'analyser un grand nombre de causes d'erreurs.

Il est clair que notre définition sommaire « type = ensemble » est peu précise: d'une part elle laisse dans l'ombre les raisons qui conduisent à adopter un nouveau terme (type et non ensemble) et d'autre part elle suppose de posséder une définition claire de ce qu'est un ensemble.

Pour ce qui est de l'adoption d'une terminologie particulière (type) il s'agit en fait de mettre l'accent sur ce qu'on veut faire de cette notion: contrôle de qualité des programmes, méthodologie de production.

En mathématiques, on peut définir des ensembles en extension (ce qui n'est praticable que pour des ensembles finis) ou en intention. L'informaticien, lui, est plus directement intéressé par ce qu'il est capable de faire avec les objets d'un type particulier. Ce point de vue conduit à associer à un type la liste des opérations applicables sur les objets du type: ce qu'on appelle ses *fonctions d'accès.* Nous illustrons ce point de vue sur deux types d'objets manipulés au chapitre précédent, le type ascii et le type mot.

5.1.2 Types, volet spécification

Nous utilisons une notation syntaxique (qui n'appartient pas au langage Scheme) pour donner la spécification d'un type.

5.1.2.1 Fonctions d'accès et abstraction

À l'aide de notre notation, dont nous précisons au fur et à mesure la signification, le type ascii peut être spécifié ainsi :

type ascii **volet spécification**
fonctions d'accès
constantes
#\x: ascii
toutes les notations de la forme #\x où x correspond à la frappe d'un caractère au clavier désignent un caractère ascii
#\space, #\newline, #\tab… : ascii
ces notations correspondent à des caractères dits « spéciaux » comme le blanc, le retour à la ligne, la tabulation.
MinAscii : ascii
le plus petit des caractères ascii
MaxAscii : ascii
le plus grand des caractères ascii
test de type
Ascii? : indifférent → booléen
détermine si un objet est un caractère ascii
opérations
>Ascii? : ascii x ascii → booléen
c1, c2 → *c1 est situé après c2 dans l'ordre standard des caractères ascii*
; on dispose également d'autres fonctions de comparaison (préfixes >=, <, <=)
PredAscii : ascii – {MinAscii} → ascii – {MaxAscii}
c → *le prédécesseur immédiat de c*
SuccAscii : ascii – {MaxAscii} → ascii – {MinAscii}
c → *le successeur immédiat de c*
fin spécification

Il s'agit de la *spécification* d'un type (usage des mots clés **volet spécification**). Une spécification comporte une partie obligatoire et des parties optionnelles ici absentes. La partie **fonctions d'accès** donne la liste exhaustive des opérations considérées comme primitives, applicables sur les objets du type: tout se passe comme si la machine avec laquelle on travaille est munie de ces opérations.

En réalité, la machine Scheme ne connaît qu'un certain nombre de types primitifs (ascii en est un) et le programmeur doit concevoir et mettre en œuvre les types des objets dont il estime avoir l'utilité.

- Concevoir un type consiste à donner son **volet spécification**. On peut donc, une fois le type conçu, l'utiliser comme s'il était fourni par une machine abstraite. Cette manière de procéder est résumée par le terme *abstraction de données.*
- Mettre en œuvre un type consiste à donner des algorithmes pour les fonctions d'accès, en utilisant des types primitifs ou des types dont on a donné la spécification et qui devront, à leur tour, être mis en œuvre. Il s'agit donc de concevoir, selon une approche descendante, une hiérarchie de machines abstraites reposant, *in fine*, sur la machine Scheme.

La liste des fonctions d'accès comporte au minimum leurs noms et leurs spécifications (type et relation entre paramètres et résultat). Pour aider le lecteur, un certain nombre de pragmats (commentaires) en italique peuvent être ajoutés. Ainsi, il est agréable de regrouper les fonctions par rubriques.

Parmi les rubriques classiques, la rubrique *constantes* précise les notations (s'il y en a) ou les identificateurs qui permettent de désigner directement des objets du type. La rubrique *test de type* est, la plupart du temps, utile.

Définition 5.1 Type abstrait de donnée: spécification

Spécifier un *type abstrait de donnée* consiste à:

- Lui attribuer un nom
- Donner la spécification des fonctions d'accès aux objets du type dont on suppose l'existence.

Remarques

- La liste des fonctions d'accès peut être structurée en rubriques; les plus fréquentes sont: *constantes*, *test de type*, *opérations d'extraction*, *opérations de construction*.
- Scheme n'offre aucune fonctionnalité permettant de définir des types abstraits. Le langage de spécification de type que nous utilisons est donc une convention imposée lors de la conception de l'algorithme. Il y a une identité de rôle entre spécification de fonction et spécification de type.

5.1.2.2 Exemple

Nous donnons ci-après la spécification du type mot introduit dans le chapitre précédent[1].

type mot **volet spécification**
fonctions d'accès
constantes
"" : mot
représente le mot vide
"x" : mot
toutes les notations de la forme "x" *où* x *est une séquence de caractères sont des mots*
test de type
Mot? : indifférent → booléen
détermine si un objet est un mot

1. On rappelle que la spécification de ce type a été conçue pour permettre quelques exercices de programmation récursive. Une définition réaliste du type mot devrait inclure la fonction de base du type: la concaténation.

opération d'observation

MotVide? :	mot	→	booléen
	détermine si un mot est le mot vide		

opérations d'extraction

CarDroit :	mot+	→	ascii
	m	→	*le dernier caractère de m*
SaufDroit :	mot+	→	mot
	m	→	*un mot de longueur | m | - 1, obtenu en amputant m de son dernier caractère*
CarGauche :	mot+	→	ascii
	m	→	*le premier caractère de m*
SaufGauche :	mot+	→	mot
	m	→	*un mot de longueur | m | - 1, obtenu en amputant m de son premier caractère*

opérations de construction

AjoutDroit :	mot x ascii	→	mot+
	m, c	→	*un mot de longueur | m | + 1, obtenu en ajoutant à la fin de m le caractère c*
AjoutGauche :	ascii x mot	→	mot+
	c, m	→	*un mot de longueur | m | + 1, obtenu en ajoutant au début de m le caractère c*

fin spécification

L'ensemble des spécifications vise à fournir au programmeur:

- des explications sur ce que font les fonctions d'accès;
- la possibilité de raisonner sur les programmes qu'il écrit.

Les spécifications constituent en outre le cahier des charges que toute mise en œuvre du type devra respecter.

5.1.3 Mise en œuvre

Le second volet d'une définition de type consiste à rédiger des programmes qui mettent en œuvre les fonctions d'accès. Avec les éléments techniques dont nous disposons à ce stade de l'exposé, la mise en œuvre du type mot n'est guère intéressante: elle consiste à habiller un type prédéfini, le type string. Pour aller plus loin, et construire des types intéressants, il est néces-

saire de disposer d'un type primitif important, le doublet, que nous introduisons en commençant à traiter un exemple non trivial.

Définition 5.2 Type abstrait de donnée: mise en œuvre

Mettre en œuvre un *type abstrait de donnée* consiste à:

- Choisir une manière de représenter les objets du type en utilisant d'autres types de données.
- Donner l'algorithme de ses fonctions d'accès.

Remarques

- Pour un même type abstrait, on peut imaginer plusieurs mises en œuvre différentes.
- L'intérêt principal de la démarche réside dans la possibilité de changer de mise en œuvre sans avoir à modifier les programmes qui utilisent des objets du type.
- On doit évidemment se convaincre de la conformité entre les algorithmes fournis et leurs spécifications.

5.2 Couches d'abstractions, hiérarchie de machines abstraites et approche ascendante

5.2.1 Un problème de programmation

Un problème, que nous appelons « contrôle de durées », nous sert de fil directeur au cours de ce chapitre et du suivant.

Problème « contrôle de durées »
On s'intéresse à un processus dont la réalisation comporte des étapes successives numérotées de 1 à n. À titre d'exemple on peut considérer que le processus en question est la fabrication d'un objet sur une chaîne de montage. On appelle phase de réalisation l'ensemble des opérations effectuées entre deux étapes i et j (avec évidemment i < j). Un certain nombre de chronométreurs sont en charge de relever les durées (qui sont mesurées en heures, minutes et secondes) de certaines phases.

À partir d'un certain nombre de relevés effectués par les chronométreurs, on veut pouvoir effectuer certains calculs ou comparaisons qui supposent des comparaisons, des additions ou des soustractions de durées. Ainsi,

- à partir des durées des phases [1,5] et [5, 8] on peut calculer la durée de la phase [1, 8],
- à partir des durées des phases [1,5] et [3, 5] on peut calculer la durée de la phase [1, 3].

Nous ne réalisons, compte tenu de l'énoncé du problème, qu'une étude très simplifiée: un problème concret est plus compliqué, d'autres solutions sont

envisageables. Nous nous focalisons sur les types de données qu'il serait utile de pouvoir manipuler.

5.2.2 Structuration du problème

Nous cherchons à définir et implanter un certain nombre de fonctions utiles aux personnes chargées du dépouillement des mesures. Ces fonctions doivent pouvoir manipuler des objets qui sont:

- des relevés
- des phases
- des durées

Si on admet qu'on dispose de types que nous appelons relevé, phase et durée, les fonctions d'accès aux objets de ces type doivent permettre de programmer les fonctions du dépouillement. Nous revenons sur le dépouillement proprement dit au chapitre suivant (nous manquons, à ce stade de l'exposé, de certains concepts indispensables pour concevoir de telles fonctions) et nous nous contentons de définir et mettre en œuvre ces trois types.

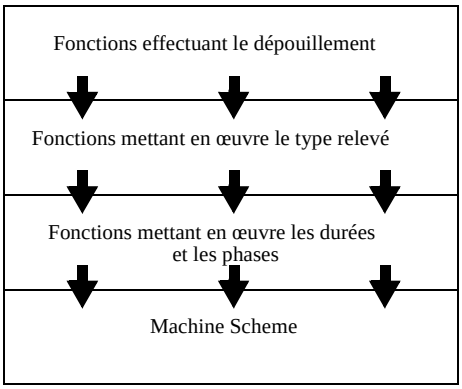
Intuitivement on peut imaginer que les type durée et phase peuvent être mis en œuvre sur la machine Scheme (ils utilisent manifestement des entiers), alors que le type relevé suppose probablement, pour sa mise en œuvre, qu'on dispose des types phase et durée.

On dit qu'on cherche à définir une hiérarchie de types à trois niveaux: les niveaux 2 (relevé), 1 (phase et durée) et 0 (machine Scheme). Cette hiérarchie peut se concevoir comme l'enrichissement progressif de la machine Scheme.

5.2.3 Hiérarchie de machines abstraites

Au niveau de l'application, on peut considérer qu'on dispose d'une machine fournissant les types relevé, phase et durée. Cette machine est immatérielle (d'où le terme machine abstraite): sa « construction » est réalisée par des programmes – les fonctions qui mettent en œuvre le type relevé – qui utilisent une autre machine abstraite (types phase et durée). Le programme complet est donc structuré en couches successives d'abstractions jusqu'à ce que les fonctions dont on a l'utilité soient fournies par la machine réelle dont on dispose[1].

1. En fait, la hiérarchie de machines abstraites ne s'arrête pas à la machine Scheme: cette dernière est mise en œuvre par l'interpréteur Scheme qui tourne sur une autre machine, toujours abstraite, la machine MS-DOS qui, enfin, est mise en œuvre sur une machine réelle.



5.3 Spécification des types relevé, phase et durée

5.3.1 Spécification du type relevé

Pour spécifier proprement le type relevé on a besoin de savoir comment il va être utilisé. En l'absence des spécifications complètes de l'application on est réduit à imaginer, à partir de l'énoncé du problème, quelques fonctionnalités utiles, voire indispensables.

Fonctions de création et d'extraction

Les composantes logiques d'un relevé sont la phase et la durée. On a besoin de trois fonctions: extraction de la phase ou de la durée d'un relevé, construction d'un relevé à partir d'une phase et d'une durée.

Calculs sur les relevés

On souhaite pouvoir, à partir de deux relevés existants, calculer des relevés par addition ou soustraction, lorsque les phases le permettent: si les deux phases sont immédiatement consécutives on peut effectuer une « addition », si la phase du second commence ou termine la phase du premier, on peut « soustraire ».

On obtient la spécification:

type relevé **volet spécification**
fonctions d'accès
Fonctions de création et d'extraction

Phase :	relevé	→	phase
Durée :	relevé	→	durée
	ces deux fonctions délivrent la composante d'un relevé qui correspond à leurs noms		
ConsR:	phase x durée	→	relevé
	p, d	→	*le relevé d'une durée d pour la phase p*

Calculs sur les relevés

+R:	relevé x relevé	→	relevé ∪ {"erreur"}
			effectue « l'addition » de deux relevés, lorsque leurs phases sont consécutives

-R:	relevé x relevé	→	relevé ∪ {"erreur"}
			effectue « soustraction » de deux relevés, lorsque la phase du second commence ou termine la phase du premier

fin spécification

5.3.2 Spécification du type phase

Une phase étant repérée par deux étapes (qui sont des naturels), on doit pouvoir créer une phase à partir de deux naturels et extraire, à partir d'une phase, ses étapes de début et de fin.

Par ailleurs, il est clair qu'au vu de la spécification du type relevé, on a besoin de savoir, étant donné deux phases, si elles se suivent immédiatement, si l'une commence ou termine l'autre.

Enfin, il est pratique de disposer de fonctions qui « additionnent » ou « soustraient » des phases lorsque c'est possible.

type phase **volet spécification**
fonctions d'accès

Fonctions de création et d'extraction

DébP :	phase	→	naturel
FinP :	phase	→	naturel
	ces deux fonctions délivrent respectivement le début et la fin d'une phase		
ConsP:	naturel x naturel	→	phase ∪ {"erreur"}
	d, f	→	*la phase débutant à l'étape d et finissant à l'étape f si d < f, "erreur" sinon*

Calculs sur les phases

SeSuivent?:	phase x phase	→	booléen
	p1, p2	→	*la fin de p1 et le début de p2 coïncident*
+P:	phase x phase	→	phase ∪ {"erreur"}
	p1, p2	→	*une phase dont le début est celui de p1 et la fin celle de p2 si p1 et p2 se suivent, "erreur" sinon*
Commence?:	phase x phase	→	booléen
	p1, p2	→	*le début de p1 et le début de p2 coïncident et la fin de p1 est strictement plus petite que la fin de p2*
Termine?:	phase x phase	→	booléen
	p1, p2	→	*la fin de p1 et la fin de p2 coïncident et le début de p1 est strictement plus grand que le début de p2*
-P:	phase x phase	→	phase ∪ {"erreur"}
	p1, p2	→	*une phase qui correspond à p1 « amputée » de p2 si p2 commence ou termine p1, "erreur" sinon*

fin spécification

5.3.3 Spécification du type durée

Un des problèmes posés par le type durée réside dans les multiples possibilités d'exprimer une durée donnée: les durées 61s et 1mn1s ont la même valeur. Quelque soit la manière dont une durée est enregistrée, on peut définir ce que nous appelons ses composantes normalisées: le nombre d'heures (pas de restriction), de minutes (plus petit que 60), de secondes (plus petit que 60).

Pour créer une durée, on peut imaginer qu'on vient de lire le résultat d'un chronométrage: on donne un nombre d'heures, de minutes et de secondes. Il est toutefois possible que le chronométreur n'exprime pas la lecture qu'il vient de faire en composantes normalisées: on admet qu'il peut faire des lectures comme 0 h 0 mn 125 s.

Les autres fonctionnalités que nous introduisons sont assez naturelles: elles visent à faire des calculs et des comparaisons de durées.

type durée **volet spécification**
fonctions d'accès
opérations de normalisation

Heures :	durée	→	naturel
Minutes :	durée	→	{0..59}
Secondes :	durée	→	{0..59}

ces trois fonctions délivrent la composante normalisée qui correspond à leurs noms

comparaisons et calculs

=D?:	durée x durée	→	booléen
>D?:	durée x durée	→	booléen
+D:	durée x durée	→	durée
-D:	durée x durée	→	durée ∪ {"erreur"}

chacune de ces quatre fonctions effectue le calcul correspondant à son nom

initialisations

Chrono:	naturel x naturel x naturel	→	durée

*création d'une durée avec un nombre **quelconque** d'heures, minutes, secondes.*

fin spécification

5.4 Le type doublet

On imagine sans trop de mal que, pour mettre en œuvre le type phase, il serait pratique de disposer d'un moyen d'agglomérer deux naturels représentants le début et la fin de la phase.

5.4.1 Spécification du type doublet

Un type prédéfini de Scheme, le doublet, permet d'agglomérer deux objets quelconques.

type doublet, **volet spécification**
fonctions d'accès

test de type

pair?	:	indifférent	→	booléen
		détermine si un objet est un doublet		

opération de création

cons	:	indifférent x indifférent	→	doublet
		a, b	→	*un doublet dont la première composante est a, la seconde b*

opérations d'extraction

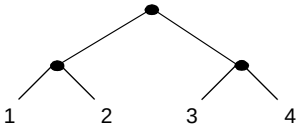
car	:	doublet	→	indifférent
		délivre la première composante du doublet		
cdr	:	doublet	→	indifférent
		délivre la seconde composante du doublet		

fin spécification

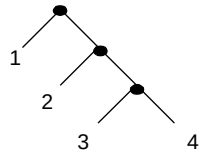
Ce type n'a pas de volet mise en œuvre explicite (il est justement prédéfini). Il est très pratique de visualiser les doublets par des dessins:

cons(1, 2)

cons(cons(1, 2), cons(3, 4))



cons(1, cons(2, cons(3, 4)))



car(cons(cons(1, 2), cons(3, 4)))

Pour mettre en œuvre des triplets, des quadruplets… on utilise plusieurs doublets (cf. figure précédente). Pour accéder à des objets dans une composition de doublets il y a lieu de combiner opérations car et cdr. Scheme utilise une convention pratique:

cadr(x) = car(cdr(x))
cddr(x) = cdr(cdr(x))
cdar(x) = cdr(car(x))
caar(x) = car(car(x))
caddr(x) = car(cdr(cdr(x)))
…

Pour être précis, le type que nous venons de définir devrait être appelé doublet de indifférent et indifférent, à partir duquel on peut sans peine imaginer un type doublet de entier et booléen…

5.4.2 Mise en œuvre du type phase

On rappelle que mettre en œuvre un type consiste à:

1. Adopter une représentation pour les objets de ce type, laquelle peut, évidemment, utiliser d'autres types.
2. Donner un algorithme pour toutes les fonctions d'accès du type. Pour donner cet algorithme on utilise naturellement la connaissance qu'on a de la représentation adoptée.

On peut donner, pour le type phase, une mise en œuvre triviale

type phase **volet mise en œuvre**
principe
• Une phase est mise en œuvre par un doublet de naturels, cons(D, F), où D et F représentent respectivement les étapes de début et de fin de la phase.
algorithmes des accès
Fonctions de création et d'extraction

$DébP = car$

$FinP = cdr$

$$ConsP(d, f) = \begin{cases} \text{"erreur"} \textbf{ si } d \geq f \\ cons(d, f) \textbf{ sinon} \end{cases}$$

Calculs sur les phases

$SeSuivent?(p1, p2) = (cdr(p1) = car(p2))$

$$+P(p1, p2) = \begin{cases} cons(car(p1), cdr(p2)) \\ \qquad \textbf{si } SeSuivent?(p1, p2) \\ \text{"erreur"} \textbf{ sinon} \end{cases}$$

$Commence?(p1, p2) = (car(p1) = car(p2)) \wedge (cdr(p1) < cdr(p2))$

$Termine?(p1, p2) = (cdr(p1) = cdr(p2)) \wedge (car(p1) > car(p2))$

$$-P(p1, p2) = \begin{cases} cons(cdr(p2), cdr(p1)) \\ \qquad \textbf{si } Commence?(p2, p1) \\ cons(car(p1), car(p2)) \\ \qquad \textbf{si } Termine?(p2, p1) \\ \text{"erreur"} \textbf{ sinon} \end{cases}$$

fin mise en œuvre

On remarque un phénomène que nous qualifions par « franchissement d'une barrière d'abstraction »: lors de l'appel, les paramètres effectifs fournis à Termine?, par exemple, sont des phases (ces objets ne sont manipulables, dans la fonction appelante, que par l'intermédiaire des fonctions d'accès au type phase) alors que, pendant l'exécution de l'algorithme de Termine?, il s'agit de doublets, ce qui justifie le fait qu'on puisse leur appliquer la fonction car.

Scheme ne prenant pas en charge de telles barrières, il est important de se discipliner et de ne pas considérer, à l'extérieur des fonctions d'accès, une phase comme un doublet. Le respect de cette convention conditionne tout changement ultérieur de mise en œuvre.

5.4.3 Mise en œuvre du type relevé

Il est facile d'opter pour un doublet <durée, phase> comme représentation d'un relevé.

type relevé **volet mise en œuvre**
principe
• Un relevé est mis en œuvre par un doublet, cons(D, P), où D et P représentent respectivement la durée et la phase du relevé
algorithmes des accès
Fonctions de création et d'extraction
Phase = cdr
Durée = car
ConsR = cons
Calculs sur les relevés

$$+R(r1, r2) = \begin{cases} cons(+D(car(r1), car(r2)), +P(cdr(r1), cdr(r2))) \\ \quad \textbf{si } SeSuivent?(cdr(r1), cdr(r2)) \\ \text{"erreur" } \textbf{sinon} \end{cases}$$

$$-R(r1, r2) = \begin{cases} cons(-D(car(r2), car(r1)), -P(cdr(r2), cdr(r1))) \\ \quad \textbf{si } Commence?(cdr(r2), cdr(r1)) \vee Termine?(cdr(r2), cdr(r1)) \\ \text{"erreur" } \textbf{sinon} \end{cases}$$

fin mise en œuvre

5.5 Mises en œuvre du type durée

5.5.1 Modifications d'un logiciel et changement de mise en œuvre

Un des intérêts majeurs de la démarche que nous présentons dans ce chapitre réside dans la possibilité de maîtriser les évolutions des programmes obtenus.

Nous demandons au lecteur d'admettre que, dans un cadre réaliste, on entretient les programmes qui ont été écrits. La nécessité de la maintenance d'un logiciel a deux causes principales:

- On y découvre des dysfonctionnements qui doivent donc être corrigés.
- On lui fait subir des évolutions: ajout ou modifications de fonctionnalités, transport sur d'autres plate-formes…

Pour se convaincre qu'un logiciel évolue, le lecteur peut se renseigner sur le numéro de la version courante d'un produit de grande diffusion.

Pour qu'un logiciel soit « maintenable », il importe de pouvoir localiser avec précision les parties du logiciel susceptibles d'être modifiées. La pertinence de cette localisation (et aussi son étendue) a un impact sur le coût (et la qualité) de la modification.

Nous illustrons cette propriété en étudiant deux mises en œuvre plausibles pour le type durée. On pourra se convaincre que ces mises en œuvre sont interchangeables et qu'on localise avec précision, au niveau du codage, les changements à effectuer.

5.5.2 Type durée : une première mise en œuvre

Il est clair qu'on peut représenter une durée par le nombre total de secondes :

type durée **volet mise en œuvre**
principe
- Une durée est mise en œuvre par un naturel qui représente sa valeur en secondes.

algorithmes des accès
opérations d'extraction

Heures(t) = t **div** 3600

Minutes(t) = (t **mod** 3600) **div** 60

Secondes(t) = t **mod** 60

comparaisons et calculs

=D? = **=**

>D? = **>**

+D = +

$$\text{-D}(t1, t2) = \begin{cases} \text{"erreur" } \textbf{si } t1 < t2 \\ t1 - t2 \textbf{ sinon} \end{cases}$$

initialisations

Chrono(h, m, s) = $h \times 3600 + m \times 60 + s$

fin mise en œuvre

5.5.3 Une seconde mise en œuvre du type durée

Une durée étant déterminée par un nombre d'heures, de minutes et de secondes une mise en œuvre évidente consiste à opter pour une composition de doublets qui agglomère ses composantes normalisées.

type durée **volet mise en œuvre**
principe
- Une durée est mise en œuvre par un doublet
  cons(cons(H, M), S)
  où H, M, et S sont ses composantes normalisées

algorithmes des accès
fonctions auxiliaires

Total : durée → naturel
calcule la durée totale en secondes

Total(t) = cdr(t) + 60 × (cdar(t) + 60 × caar(t))

Normer : naturel → durée
crée une durée à partir d'un nombre de secondes

Normer(s) = cons(cons(s **div** 3600, (s **mod** 3600) **div** 60), s **mod** 60)

fonctions d'accès
opérations d'extraction

Heure = caar
Minutes = cdar
Secondes = cdr
comparaisons et calculs
=D? = equal?
>D?(t1, t2) = Total(t1)**>**Total(t2)
+D(t1, t2) = Normer(Total(t1) + Total(t2))

$$\text{-D}(t1, t2) = \begin{cases} \text{"erreur" } \textbf{si } TotalC(t1) < TotalC(t2) \\ Normer(TotalC(t1) - TotalC(t2)) \textbf{ sinon} \end{cases}$$

initialisations
Chrono(h, m, s) = Normer($h \times 3600 + m \times 60 + s$)
fin mise en œuvre

On peut se poser la question générale du choix d'une mise en œuvre parmi les diverses possibilités envisageables. Deux facteurs sont à prendre en compte:

- la difficulté de conception: la conception n'étant faite qu'une seule fois, ce qui compte vraiment c'est le temps consacré à cette tâche et la qualité du résultat. Ce critère n'est déterminant que si les différences sont extrêmement marquées.
- le coût de l'utilisation (en temps d'exécution par exemple). Ce coût dépend du profil d'utilisation: dans l'exemple que nous avons traité, tout dépend des fréquences relatives des opérations d'extraction et des autres opérations[1].

5.6 Codage en Scheme

Le codage en Scheme est particulièrement fastidieux: il faut coder tous les algorithmes de *toutes* les mises en œuvre!

5.7 Conclusions

La notion d'abstraction qui permet de décomposer un problème en sous-problèmes plus simples à résoudre est centrale en informatique. En effet, si n'importe qui est capable d'écrire, sans utiliser de méthodes particulières, un programme de 50 lignes on constate que, dans un contexte réaliste, l'écriture de logiciels (dont certains dépassent le million de lignes) suppose d'appliquer des méthodes rigoureuses de décomposition et d'organisation du travail.

1. En dépit des apparences, les opérations d'extraction de la première mise en œuvre peuvent être relativement coûteuses: les fonctions *quotient* et *remainder* de la machine Scheme, qui ne sont pas des opérations primitives, peuvent prendre un temps non négligeable.

Une des difficultés d'un cours d'initiation réside dans le fait que la taille des exemples traités fait ressentir ces méthodes comme des contraintes. Dans la réalité, à partir de 500 lignes elles deviennent indispensables.

Pour décomposer il faut abstraire, et les langages fonctionnels (Scheme en est un) possèdent, comme nous l'avons vu à l'occasion des chapitres précédents, l'outil de base d'abstraction: la notion de fonction. L'abstraction de données qui a fait l'objet de ce chapitre est plus subtile à appréhender mais tout aussi fondamentale. Dans l'état actuel d'avancement du cours, il est difficile de l'utiliser en vraie grandeur; aussi, au préalable, nous donnerons au chapitre suivant les bases techniques nécessaires pour rédiger des fonctions travaillant sur des listes. Une fois ces bases acquises, nous reprendrons l'étude des abstractions de données (chapitre consacré aux arbres).

5.8 Exercices

Exercice 5.1 Type relatif

1. Spécifier un type relatif (entier) en se limitant à l'addition et la soustraction.
2. Mettre en œuvre ce type à l'aide d'un doublet de naturels.

Exercice 5.2 Type fraction et type rationnel

Spécifier et mettre en œuvre un type fraction et un type rationnel.

Exercice 5.3 Type durée

Proposer une mise en œuvre du type durée dans laquelle on implante des composantes quelconques (non normalisées) d'une durée.

- On réfléchira aux simplifications que peut induire ce choix pour la mise en œuvre de la fonction +D.
- On réfléchira aux raisons qui militent pour l'existence de la fonction d'accès =D?.

Exercice 5.4 Type date (1)

On s'intéresse à des dates (jour / mois / année) postérieures au 1er janvier 1900.

1. Spécifier un type abstrait permettant, entre autres, de comparer deux dates, d'obtenir, à partir d'une date, la date de la veille ou du lendemain.
2. Utiliser ce type pour programmer les fonctions:

- Intervalle?(d1, d2), qui vérifie que d1 et d2 peuvent être les dates de début et de fin d'un événement.
- NbJours(d1, d2) qui compte le nombre de jours entre d1 et d2.
- JourDeLaSemaine(d) qui donne le jour de la semaine d'une date d (on sait que le 1er janvier 1900 était un lundi).

3. Proposer une première mise en œuvre utilisant un triplet.
4. Proposer une autre mise en œuvre

Exercice 5.5 Type date (2)

1. Modifier la spécification de l'exercice précédent de manière à pouvoir obtenir, à partir d'une date, des mots qui :

- correspondent au codage français habituel: 23/12/1995
- correspondent au codage anglo-saxon: 12/23/1995

2. Modifier la spécification de l'exercice précédent de manière à pouvoir obtenir, à partir d'une date:

- le numéro du jour dans l'année
- le numéro de la semaine dans l'année

CHAPITRE 6

Une structure de données récursive à droite: la liste

6.1 Définition et mise en œuvre

6.1.1 Définition

Les doublets sont commodes pour agglomérer des informations dont on connaît le nombre. Si on voulait représenter tous les individus d'une organisation (entreprise, étudiants…) à l'aide d'une composition de doublets, il faudrait connaître statiquement leur nombre et celui-ci ne pourrait plus varier. Dans le même ordre d'idées, des structures complexes de doublets sont pénibles à manipuler lorsqu'elles sont trop grosses.

Il existe donc des cas de figure où l'on souhaite agglomérer des choses dont on ne connaît pas le nombre ou dont le nombre est trop important pour qu'on ait envie de nommer chacune des composantes. La *liste* est une structure de ce type.

Définition 6.1 Liste, éléments et longueur

Une *liste* est soit:

- une liste particulière, appelée liste vide
- un doublet dont la seconde composante est une *liste*

Dans le cas d'une liste non vide la première composante de chacun des doublets de la liste est appelée *élément* de la liste. La liste vide n'a pas d'éléments.

La *longueur* d'une liste L est son nombre d'éléments; on la note par | L |. La liste vide a pour longueur 0.

Remarque
En Scheme, la liste vide est désignée par l'identificateur nil ou notée par une paire de parenthèses.

Fonctionnellement, on distingue très bien listes et doublets : le doublet a un caractère primitif (il agglomère 2 choses quelconques et, pour l'instant c'est le seul mode d'agglomération dont nous disposons) alors que la liste est une structure essentiellement récursive, qu'on peut construire à l'aide de doublets et de la liste vide. On peut très bien traduire cette différence par une hiérarchie d'abstractions.

type liste **volet spécification**
accès
Constante
nil : liste
Test de type
liste? : indifférent → booléen
détermine si un objet est une liste
Observation
null? : liste → booléen
détermine si une liste est la liste vide
Construction
cons : indifférent x liste → liste
A, L → *une liste dont le premier élément est A et dont les suivants sont ceux de L*
Extraction
car : liste - {nil} → indifférent
délivre le premier élément d'une liste non vide
cdr : liste - {nil} → liste
délivre la liste amputée de son premier élément (la liste de départ est non vide)
fin spécification

6.1.2 Mise en œuvre

Compte tenu de la définition, il est particulièrement simple de mettre en œuvre les listes non vides. Pour la liste vide, il suffit de choisir un objet quelconque qui puisse facilement être distingué.

type liste **volet mise en œuvre**
principe
- la liste vide est représentée par la valeur associée à l'identificateur nil.
- une liste non vide est représentée par un doublet dont la seconde composante est une liste.

algorithmes des accès
nil désigne la liste vide (objet primitif de Scheme)
null? désigne une fonction Scheme prédéfinie
cons = cons
car = car
cdr = cdr

$$\text{liste?}(x) = \begin{cases} x = \text{nil} & \textbf{si } \neg\text{pair?}(x) \\ \text{liste?}(\text{cdr}(x)) & \textbf{sinon} \end{cases}$$

fin mise en œuvre

On remarque que le seul accès non prédéfini est le test de type liste?.

On rappelle que, lorsqu'on donne l'algorithme des accès, on franchit une « barrière d'abstraction ». Dans le corps de ces algorithmes, les paramètres de type liste non vides sont des doublets: on peut donc leur appliquer des fonctions d'accès aux doublets (comme cons, car et cdr). Le fait qu'on utilise les même noms est anecdotique (et pratique).

6.1.3 Listes particulières

Dans la définition que nous avons donnée, les éléments d'une liste ont un type quelconque. Si on fixe le type des éléments, on obtient des particularisations du type général. Nous donnons des noms à quelques types dont nous aurons l'occasion de nous servir.

- type atome = indifférent - {fonctions} - doublet
- type atome+ = atome - {nil}
- type liste = liste de indifférent
- type plate = liste d'atomes+
- type liste-ent = liste d'entiers
- type liste-ascii = liste d'ascii
- type liste-mot = liste de mots

On s'exercera avec profit à définir les fonctions de test de type associées. Pour ce faire, on peut utiliser les fonctions Scheme prédéfinies:

atom?: indifférent → booléen
détermine si un objet est un atome
pair? : indifférent → booléen
détermine si un objet est un doublet
null? : indifférent → booléen
détermine si un objet est la liste vide
integer?: indifférent → booléen
détermine si un objet est un entier
char?: indifférent → booléen
détermine si un objet est un caractère ascii
equal?: indifférent x indifférent → booléen
détermine si deux objets sont identiques[1]

On remarque que nil est une liste plate, une liste d'entiers, une liste de mots… En effet, en logique, les deux propriétés suivantes sont considérées comme équivalentes:

(1) Tous les éléments de X sont des Y
(2) Il est impossible de trouver un élément de X qui soit autre chose qu'un Y

Par convention, si xxx désigne le type liste de yyy, xxx+ désigne le type xxx - {nil}.

1. On rappelle que le type de la fonction = est nombre x nombre → booléen

6.2 Liste et récursivité à droite

6.2.1 Concaténation de deux listes

Le lecteur réfléchira avec soin aux différences entre le type liste-ascii et le type mot. Les deux types permettent d'agglomérer des caractères mais le premier ne se prête absolument pas à une récursivité à gauche.

Définition 6.2 Concaténation

Concaténer deux listes consiste à construire la liste qui contient les éléments de la première suivis des éléments de la seconde.

Nous cherchons à programmer la fonction:

Problème

Concat :	liste x liste	→	liste
	m, n	→	la concaténation de m et de n

Il est clair qu'on peut s'inspirer de la version récursive à droite de la fonction Coller vue au § 4.3.1.

Analyse

1. Il s'agit donc d'exprimer Concat(m, ...) en fonction de Concat(cdr(m), ...):
   Concat(m, n) = cons(car(m), Concat(cdr(m), n))
2. L'ordre associé est fondé sur la longueur des listes.
3. L'équation récursive est mal typée pour m = nil
4. Concat(nil, n) = n

Algorithme

$$\text{Concat}(m, n) = \begin{cases} n \textbf{ si } \text{null?}(m) \\ \text{cons}(\text{car}(m), \text{Concat}(\text{cdr}(m), n)) \textbf{ sinon} \end{cases}$$

6.2.2 Quelques fonctions de base

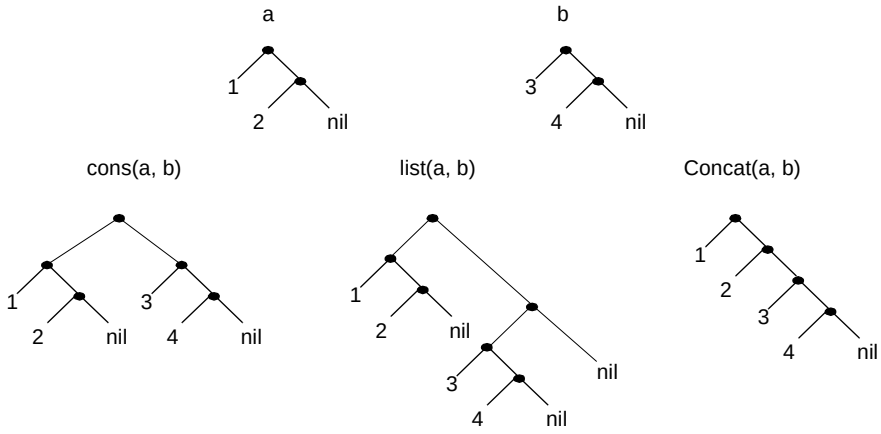
Un certain nombre de fonctions de base sur les listes sont prédéfinies en Scheme. Ainsi, on dispose de:

append:	liste…	→	liste
	effectue la concaténation des listes données en paramètre		
list:	indifférent…	→	liste
	construit une liste dont les éléments sont donnés en paramètre		
length:	liste	→	naturel
	calcule la longueur d'une liste		
member:	indifférent x liste	→	booléen
	a, l	→	*a est un élément de l*

6.2.3 Constructeurs de listes

6.2.3.1 Les constructeurs de base

On remarque qu'il y a plusieurs constructeurs de listes et qu'il importe de bien distinguer leurs fonctionnalités. Ainsi:



Du point de vue du typage, si on considère que a et b sont des liste-ent:

- Concat(a, b) est une liste-ent dont la longueur est longueur(a) + longueur(b)
- list(a, b) est une liste de liste-ent dont la longueur est 2
- cons(a, b) est une liste de [liste-ent ∪ entier] dont la longueur est 1 + longueur(b).

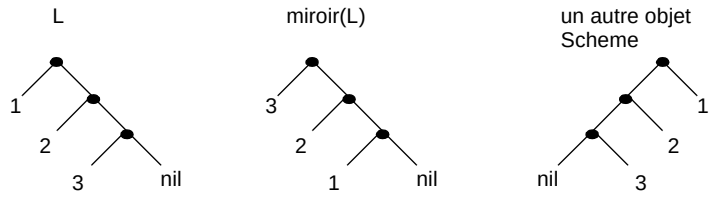
6.2.3.2 Du choix des bons constructeurs

Problème

Soit à écrire la fonction Miroir :

Miroir :	plate	→	plate
	m	→	la liste miroir de m

Le miroir d'une liste L est une liste qui contient, rangés dans l'ordre inverse, les mêmes éléments que L.



Nous allons, avant de donner la bonne solution, explorer un certain nombre de possibilités d'erreurs sur l'équation récursive qui consiste à exprimer Miroir(L) en fonction de Miroir(cdr(L)).

Analyse 1

1. Équation récursive (fausse):

Miroir(m) = cons(Miroir(cdr(m)), car(m))

2. L'ordre associé est fondé sur la longueur des listes.
3. L'équation récursive est toujours mal typée! En effet, le résultat de Miroir étant une liste, la fonction cons doit rendre une liste. m étant une liste plate, car(m) n'est pas une liste et l'appel de la fonction cons est mal typé (pour rendre une liste).

Analyse 2

On peut chercher à corriger l'erreur de typage en utilisant list(car(m)) à la place de car(m).

1. Équation récursive (fausse):
   Miroir(m) = cons(Miroir(cdr(m)), list(car(m)))
2. L'ordre associé est fondé sur la longueur des listes.
3. L'équation récursive est toujours mal typée: le premier paramètre du cons étant une liste plate, le second une liste plate, le résultat est une liste de [plate ∪ atome+] et non une liste plate.

Analyse

1. Équation récursive:
   Miroir(m) = Concat(Miroir(cdr(m)), list(car(m)))
2. L'ordre associé est fondé sur la longueur des listes.
3. L'équation récursive est mal typée pour m = nil
4. Miroir(nil) = nil

Algorithme

$$\text{Miroir}(m) = \begin{cases} \text{nil } \textbf{si } \text{null?}(m) \\ \text{Concat}(\text{Miroir}(\text{cdr}(m)), \text{list}(\text{car}(m))) \textbf{ sinon} \end{cases}$$

6.2.3.3 Construction pendant le parcours

Problème

Soit à écrire la fonction Elim :

Elim :	atome+ x plate	→	plate
	a, L	→	la liste obtenue en ôtant de l toutes les occurrences de l'atome a

Analyse

1. L'équation récursive comporte deux cas :

$$\text{Elim}(a, L) = \begin{cases} \text{Elim}(a, \text{cdr}(L)) \textbf{ si } \text{car}(L) = a \\ \text{cons}(\text{car}(L), \text{Elim}(a, \text{cdr}(L))) \textbf{ sinon} \end{cases}$$

2. L'ordre associé est fondé sur la longueur des listes.
3. L'équation récursive est mal typée pour L = nil
4. Elim(a, nil) = nil

Algorithme

$$\text{Elim}(a, L) = \begin{cases} \text{nil } \textbf{si } \text{null?}(L) \\ \textbf{sinon} \begin{cases} \text{Elim}(a, \text{cdr}(L)) \textbf{ si } \text{car}(L) = a \\ \text{cons}(\text{car}(L), \text{Elim}(a, \text{cdr}(L))) \textbf{ sinon} \end{cases} \end{cases}$$

6.3 Quelques algorithmes de tri

Les algorithmes de tri font partie de la culture informatique de base. Le problème type est le suivant:

Tri :	liste-ent	→	liste-ent$_{\text{triée}}$
	l	→	liste contenant les éléments de l rangés en ordre croissant

On imaginera qu'on peut appliquer les algorithmes que nous allons étudier à toute liste dont les éléments peuvent être comparés à l'aide d'une fonction <= adéquate (listes de caractères, listes de mots…).

6.3.1 Tri par insertion

Principe

Pour trier une liste m, on insère le car(m) à sa place dans le résultat du tri de cdr(m).

Problème

Soit à écrire la fonction TriIns :

TriIns :	liste-ent	→	liste-ent$_{\text{triée}}$
	m	→	liste contenant les éléments de m rangés en ordre croissant, *en utilisant l'algorithme de tri par insertion*

Sous-problème

Insérer :	entier x liste-ent$_{\text{triée}}$	→	liste-ent$_{\text{triée}}$
	a, m	→	liste obtenue en plaçant a à sa place dans m

Analyse (TriIns)

1. Équation récursive:

$$\text{TriIns}(m) = \text{Insérer}(\text{car}(m), \text{TriIns}(\text{cdr}(m)))$$

2. L'ordre associé est fondé sur la longueur des listes.

3. L'équation récursive est mal typée pour m = nil

4. TriIns(nil) = nil

Algorithme

$$\text{TriIns}(m) = \begin{cases} \text{nil } \textbf{si } \text{null?}(m) \\ \text{Insérer}(\text{car}(m), \text{TriIns}(\text{cdr}(m))) \textbf{ sinon} \end{cases}$$

Analyse (Insérer)

1. Équation récursive:

$$\text{Insérer}(a, m) = \begin{cases} \text{cons}(a, m) \textbf{ si } a < \text{car}(m) \\ \text{cons}(\text{car}(m), \text{Insérer}(a, \text{cdr}(m))) \textbf{ sinon} \end{cases}$$

2. L'ordre associé est fondé sur la longueur des listes.

3. L'équation récursive est mal typée pour m = nil

4. Insérer(a, nil) = list(a)

Algorithme

$$\text{Insérer}(a, m) = \begin{cases} \text{list}(a) \textbf{ si } \text{null?}(m) \\ \textbf{sinon} \begin{cases} \text{cons}(a, m) \textbf{ si } a < \text{car}(m) \\ \text{cons}(\text{car}(m), \text{Insérer}(a, \text{cdr}(m))) \\ \qquad\qquad\qquad\qquad \textbf{sinon} \end{cases} \end{cases}$$

6.3.2 Tri par extraction

6.3.2.1 Algorithme

Principe

On utilise une fonction Bulle capable, à partir d'une liste m, de fabriquer une liste contenant les mêmes éléments que m, le plus petit étant placé en tête. Pour ce faire, il suffit évidemment d'appliquer Bulle au cdr(m) et de comparer la tête de la liste ainsi obtenue avec car(m). Si on considère le développement d'une telle récursivité, on constate que l'élément le plus petit de m « remonte », tel une bulle, en tête de la liste.

Pour trier une liste m, on constate que, après application de la fonction Bulle, il ne reste plus qu'à trier le cdr du résultat.

Problème

Soit à écrire la fonction TriExtr :

TriExtr :	liste-ent	→	liste-ent$_{\text{triée}}$
	m	→	liste contenant les éléments de m rangés en ordre croissant, *en utilisant l'algorithme de tri bulle*

Sous-problème

Bulle :	liste-ent$^+$	→	liste-ent$^+$
	m	→	*une liste obtenue en remontant en tête de m le plus petit de ses éléments*

Analyse (TriExtr)

1. Équation récursive:

$$\text{TriExtr}(m) = \text{cons}(\text{car}(\text{Bulle}(m)), \text{TriExtr}(\text{cdr}(\text{Bulle}(m))))$$

2. On remarque que $|\ \text{Bulle}(m)\ | = |\ m\ |$, on peut donc adopter un ordre fondé sur la longueur des listes puisque $|\ \text{cdr}(\text{Bulle}(m))\ | < |\ m\ |$.

3. L'équation récursive est mal typée pour m = nil puisque Bulle n'est pas définie sur la liste vide.

4. TriExtr(nil) = nil

Algorithme

$$\text{TriExtr}(m) = \begin{cases} \text{nil } \textbf{si } \text{null?}(m) \\ \text{cons}(\text{car}(\text{Bulle}(m)), \text{TriExtr}(\text{cdr}(\text{Bulle}(m)))) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{sinon} \end{cases}$$

Analyse (Bulle)

1. Équation récursive:

$$\text{Bulle}(m) = \begin{cases} m \textbf{ si } \text{car}(m) < \text{car}(\text{Bulle}(\text{cdr}(m))) \\ \text{cons}(\text{car}(\text{Bulle}(\text{cdr}(m))), \text{cons}(\text{car}(m), \text{cdr}(\text{Bulle}(\text{cdr}(m))))) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{sinon} \end{cases}$$

2. L'ordre associé est fondé sur la longueur des listes.

3. L'équation récursive est mal typée si cdr(m) est vide (donc si m a pour longueur 1). Remarquons que, de part la spécification de Bulle, le cas m = nil n'a pas à être étudié.

4. Si m a pour longueur 1, son plus petit élément est bien en tête, donc

- Bulle(m) = m si cdr(m) = nil

Algorithme

$$\text{Bulle}(m) = \begin{cases} m \textbf{ si } \text{null?}(\text{cdr}(m)) \\ \textbf{sinon} \begin{cases} m \\ \quad \textbf{si } \text{car}(m) < \text{car}(\text{Bulle}(\text{cdr}(m))) \\ \text{cons}(\text{car}(\text{Bulle}(\text{cdr}(m))), \text{cons}(\text{car}(m), \text{cdr}(\text{Bulle}(\text{cdr}(m))))) \\ \quad \textbf{sinon} \end{cases} \end{cases}$$

6.3.2.2 Remarque sur le codage

Considérons un codage « littéral » de la fonction TriExtr:

```
; TriExtr :   liste-ent          ->    liste-ent-triée
;            m                 ->    liste contenant les éléments de m rangés
;                                     en ordre croissant, en utilisant l'algorithme
;                                     de tri bulle
(define TriExtr (lambda (m)
      (cond
            ((null? m)     nil)
            (#t            (cons (car (Bulle m)) (TriExtr (cdr (Bulle m))))))))
```

Le calcul (Bulle m) est appelé deux fois. On peut remédier à cet inconvénient de deux manières équivalentes:

1. En introduisant une fonction auxiliaire:

```
; Travail :   liste-ent+     ->     liste-ent-triée
```

```
;               b, dont le +petit elt est en tête
;                              ->      liste contenant les éléments de b rangés
;                                      en ordre croissant
(define Travail (lambda (b)
      (cons (car b) (TriExtr (cdr b)))))
; TriExtr :     liste-ent               ->      liste-ent-triée
;               m                       ->      liste contenant les éléments de m rangés
;                                               en ordre croissant, en utilisant l'algorithme
;                                               de tri bulle
(define TriExtr (lambda (m)
      (cond
            ((null? m)      nil)
            (#t             (Travail (Bulle m))))))
```

2. Ou encore, en utilisant une fonction anonyme (plus élégant):

```
; TriExtr :     liste-ent               ->      liste-ent-triée
;               m                       ->      liste contenant les éléments de m rangés
;                                               en ordre croissant, en utilisant l'algorithme
;                                               de tri bulle
(define TriExtr (lambda (m)
      (cond
            ((null? m)      nil)
            (#t             ((lambda (b) (cons (car b) (TriExtr (cdr b)))) (Bulle m))))))
```

6.3.2.3 Désignation temporaire

Dans l'exemple qui précède, l'usage d'une fonction anonyme Scheme permet de donner un nom (en l'occurrence b) au résultat du calcul de (Bulle m).

Nous incluons, assez librement, cette possibilité dans le langage de description. Par exemple :

$$\text{Bulle}(m) = \begin{cases} m \ \textbf{si}\ \text{null?}(\text{cdr}(m)) \\ \textbf{sinon} \begin{cases} \textbf{soit}\ b = \text{Bulle}(\text{cdr}(m))\ \textbf{dans} \\ \begin{cases} m \ \textbf{si}\ \text{car}(m) < \text{car}(b) \\ \text{cons}(\text{car}(b), \text{cons}(\text{car}(m), \text{cdr}(b)))\ \textbf{sinon} \end{cases} \end{cases} \end{cases}$$

Le langage Scheme admet une catégorie d'expression, l'expression *soit*, visant à rendre explicite l'usage d'une désignation temporaire. Le codage devient :

```
; Bulle :      liste-ent+      ->      liste-ent+
;              m               ->      une liste obtenue en remontant en tête de m le
;                                      plus petit de ses éléments
(define Bulle (lambda (m)
 (cond
      ((null? (cdr m))m)
      (#t     (let ((b (Bulle m)))
                  (cond
                     ((< (car m) (car b))    (cons (car m) b))
                     (#t                     (cons (car b) (cons (car m) (cdr b))))))))))
```

.

Définition 6.3 Expression Scheme: soit

Syntaxe

<expression> ::= <*notation*> | <*identificateur*> | <*appel*> | <*définition*> | <*fonction*> | <*conditionnelle*> | <soit>

<soit> ::= **(let (**<s-déf>**)** <expression> **)**
<s-déf> ::= **(** <identificateur> <expression> **)** |
(<identificateur> <expression> **)** <s-déf>

Sémantique

• L'expression
(let ((<i_1> <e_1>)…(<i_n> <e_n>)) <e>)
• est équivalente à
((lambda (<i_1>…<i_n>) <e>) <e_1>…<e_n>)

6.3.3 Tri pivot

Principe

On définit une fonction Pivot capable, à partir d'un entier p et d'une liste l, de construire deux listes contenant respectivement les éléments de l plus petits et plus grands que p.

Pour trier une liste l, on peut appliquer Pivot à cdr(l) en utilisant car(l) comme élément pivot. Il suffit ensuite de trier chacune des listes fournies par Pivot et de composer judicieusement les résultats de ces tris et car(l).

Problème

Soit à écrire la fonction TriPivot :

TriPivot : liste-ent → $\text{liste-ent}_{\text{triée}}$
m → liste contenant les éléments de m rangés en ordre croissant, *en utilisant l'algorithme du pivot*

Sous-problème

Pivot : entier x liste-ent → doublet de liste-ent
n, m → cons(a, b) où a (resp. b) est une liste contenant les
éléments de m inférieurs (resp. supérieurs ou égaux) à n.

Analyse (TriPivot)

1. Équation récursive:

$$\text{TriPivot}(m) = \begin{cases} \textbf{soit } P = \text{Pivot}(\text{car}(m), \text{cdr}(m)) \textbf{ dans} \\ \text{append}(\text{TriPivot}(\text{car}(P)), \text{list}(\text{car}(m)), \text{TriPivot}(\text{cdr}(P))) \end{cases}$$

2. Tant car(P) que cdr(P) ont strictement moins d'éléments que m, on peut donc adopter un ordre associé fondé sur la longueur des listes.

3. L'équation récursive est mal typée pour m = nil

4. TriPivot(nil) = nil

Algorithme

$$\text{TriPivot}(m) = \begin{cases} \text{nil } \textbf{si } \text{null?}(m) \\ \textbf{sinon} \begin{cases} \textbf{soit } P = \text{Pivot}(\text{car}(m), \text{cdr}(m)) \textbf{ dans} \\ \text{append}(\text{TriPivot}(\text{car}(P)), \text{list}(\text{car}(m)), \text{TriPivot}(\text{cdr}(P))) \end{cases} \end{cases}$$

Analyse (Pivot)

1. Équation récursive:

$$\text{Pivot}(n, m) = \begin{cases} \textbf{soit } P = \text{Pivot}(n, \text{cdr}(m)) \textbf{ dans} \\ \begin{cases} \text{cons}(\text{cons}(\text{car}(m), \text{car}(P)), \text{cdr}(P)) & \textbf{si } \text{car}(m) < n \\ \text{cons}(\text{car}(P), \text{cons}(\text{car}(m), \text{cdr}(P))) & \textbf{sinon} \end{cases} \end{cases}$$

2. L'ordre associé est fondé sur la longueur des listes.

3. L'équation récursive est mal typée si m a pour longueur 0.

4. Pivot(n, nil) = cons(nil, nil)

Algorithme

$$\text{Pivot}(n, m) = \begin{cases} \text{cons}(\text{nil}, \text{nil}) & \textbf{si } \text{null?}(m) \\ \textbf{sinon} \begin{cases} \textbf{soit } P = \text{Pivot}(n, \text{cdr}(m)) \textbf{ dans} \\ \begin{cases} \text{cons}(\text{cons}(\text{car}(m), \text{car}(P)), \text{cdr}(P)) & \textbf{si } \text{car}(m) < n \\ \text{cons}(\text{car}(P), \text{cons}(\text{car}(m), \text{cdr}(P))) & \textbf{sinon} \end{cases} \end{cases} \end{cases}$$

6.3.4 Tri fusion

Principe

On utilise une fonction Séparer capable de séparer une liste *l* en deux listes. Pour trier une liste *l*, on peut la diviser et trier chacune des 2 listes ainsi obtenues. Il suffit ensuite d'effectuer une *fusion* des 2 résultats.

Problème

Soit à écrire la fonction TriFusion :

TriFusion :	liste-ent	→	$\text{liste-ent}_{\text{triée}}$
	m	→	liste contenant les éléments de m rangés en ordre croissant, *en utilisant l'algorithme du tri fusion*

Sous-problèmes

Séparer :	*liste-ent*	→	*doublet de liste-ent*
	m	→	*cons(a, b) où a et b sont deux listes dans lesquelles sont répartis les éléments de m. Tant a que b ont strictement moins d'éléments que m si la longueur de m est supérieure à 1*
Fusion :	$\text{liste-ent}_{\text{triée}}$ x $\text{liste-ent}_{\text{triée}}$	→	$\text{liste-ent}_{\text{triée}}$
	a, b	→	liste triée contenant les éléments de a et de b

Analyse (TriFusion)

1. Équation récursive:

$$\text{TriFusion}(m) = \left[\begin{array}{l}\textbf{soit } D = \text{Séparer}(m) \textbf{ dans} \\ \text{Fusion}(\text{TriFusion}(\text{car}(D)), \text{TriFusion}(\text{cdr}(D)))\end{array}\right.$$

2. L'ordre associé est fondé sur la longueur des listes.

3. L'équation récursive est bien typée. En revanche, l'ordre n'est pas strict si m a 0 ou 1 élément[1].

4. TriFusion(m) = m si m = nil ou cdr(m) = nil

Algorithme

$$\text{TriFusion}(m) = \left\{\begin{array}{l} m \textbf{ si } \text{null?}(m) \\ m \textbf{ si } \text{null?}(\text{cdr}(m)) \\ \textbf{sinon } \left[\begin{array}{l}\textbf{soit } D = \text{Séparer}(m) \textbf{ dans} \\ \text{Fusion}(\text{TriFusion}(\text{car}(D)), \text{TriFusion}(\text{cdr}(D)))\end{array}\right.\end{array}\right.$$

Analyse (Séparer)

1. Équation récursive:

$$\text{Séparer}(m) = \left[\begin{array}{l}\textbf{soit } D = \text{Séparer}(\text{cddr}(m)) \textbf{ dans} \\ \text{cons}(\text{cons}(\text{car}(m), \text{car}(D)), \text{cons}(\text{cadr}(m), \text{cdr}(D)))\end{array}\right.$$

2. L'ordre associé est fondé sur la longueur des listes.

3. L'équation récursive est mal typée si m a pour longueur 0 ou 1.

4. Séparer(m) = cons(m, nil) si m = nil ou cdr(m) = nil

Algorithme

$$\text{Séparer}(m) = \left\{\begin{array}{l} \text{cons}(m, \text{nil}) \textbf{ si } \text{null?}(m) \\ \text{cons}(m, \text{nil}) \textbf{ si } \text{null?}(\text{cdr}(m)) \\ \textbf{sinon } \left[\begin{array}{l}\textbf{soit } D = \text{Séparer}(\text{cddr}(m)) \textbf{ dans} \\ \text{cons}(\text{cons}(\text{car}(m), \text{car}(D)), \text{cons}(\text{cadr}(m), \text{cdr}(D)))\end{array}\right.\end{array}\right.$$

Analyse (Fusion)

1. Équation récursive:

$$\text{Fusion}(a, b) = \left\{\begin{array}{l} \text{cons}(\text{car}(a), \text{Fusion}(\text{cdr}(a), b)) \textbf{ si } \text{car}(a) < \text{car}(b) \\ \text{cons}(\text{car}(b), \text{Fusion}(a, \text{cdr}(b))) \textbf{ sinon}\end{array}\right.$$

2. On peut opter pour l'ordre:

- $(a', b') \left\{\right.$ (a, b) si et seulement si $|a'| + |b'| < |a| + |b|$

3. L'équation récursive est mal typée si a ou b sont vides.

4. Fusion(nil, b) = b et Fusion(a, nil) = a

1. On aurait pu typer la fonction Séparer pour qu'elle n'admette que des listes ayant au moins 2 éléments. Dans ce cas, le contrôle de type aurait repéré les cas singuliers.

Algorithme

$$\text{Fusion}(a,b) = \begin{cases} b \textbf{ si } \text{null?}(a) \\ a \textbf{ si } \text{null?}(b) \\ \textbf{sinon} \begin{cases} \text{cons}(\text{car}(a), \text{Fusion}(\text{cdr}(a), b)) \textbf{ si } \text{car}(a) < \text{car}(b) \\ \text{cons}(\text{car}(b), \text{Fusion}(a, \text{cdr}(b))) \textbf{ sinon} \end{cases} \end{cases}$$

6.3.5 Comparaisons

6.3.5.1 Généralité sur le coût des algorithmes

Quand il s'exécute, un algorithme consomme des ressources (mémoire, temps d'utilisation du processeur, E/S…). Les deux principales unités de consommation sont le temps (utilisation du processeur) et l'espace (utilisation de la mémoire).

Il est facile de prendre conscience de la consommation en temps : il suffit d'avoir passé quelques minutes à attendre les résultats d'une exécution pour être sensibilisé à ce problème.

Il est à peine plus difficile de constater que l'exécution d'un programme prend de la place : si on essaye de calculer la factorielle de nombres de plus en plus grands on finit par obtenir le message d'erreur Recursion too deep : Stack overflow. Ce message signifie que l'ordinateur manque de place pour exécuter le programme qui lui a été confié.

Sur ce sujet, il est important de comprendre trois choses :

- La consommation d'un algorithme dépend de la taille des données sur lesquelles on l'exécute.
- On ne peut pas, dans le cas général, gagner sur les deux tableaux (temps et espace).
- Même s'il n'y a pas de petites économies, l'énergie passée à réfléchir sur les gains possibles doit l'être à bon escient : il est ridicule, voire dangereux, de passer des heures de travail et de rendre peu lisible un programme pour gagner une seconde sur un programme de deux heures. On s'intéresse donc à des ordres de grandeur significatifs.

Pour donner une mesure de la consommation de ressources d'un algorithme, on cherche d'ordinaire à évaluer la manière dont évolue une consommation particulière en fonction de la taille d'une donnée. Cette évaluation, qu'on appelle complexité de l'algorithme, peut être faite en moyenne, dans le meilleur des cas, dans le pire cas.

Remarque

La complexité d'un algorithme mesure son comportement dynamique (à l'exécution) et non la difficulté de sa conception : des algorithmes simples à écrire et à comprendre ont parfois des complexités apocalyptiques.

Les divers algorithmes de tri ont des complexités différentes. Dans le cadre fonctionnel, il est délicat de mesurer la complexité en espace (nous reve-

nons sur ce problème dans la conclusion de l'ouvrage). Pour ce qui concerne la complexité en temps des algorithmes de tri, on imagine sans peine que le nombre de parcours des listes (qui peut être évalué de différentes manières: nombre d'utilisations des fonctions cons, car et cdr par exemple) est significatif.

Dans la suite, n désigne la longueur de la liste à trier et nous cherchons à évaluer le nombre d'appels récursifs des diverses fonctions: à chaque appel les fonctions consomment un temps (cons, car, cdr...) qu'on peut, en première approximation, considérer comme une constante.

6.3.5.2 Complexité du tri par insertion

Il y a autant d'appels récursifs de la fonction TriIns que d'éléments dans la liste à trier. Pour chaque appel récursif, on réalise une insertion: il y a donc n insertions dans des listes triées.

Une insertion dans une liste triée de longueur p demande, si on imagine une répartition uniforme des entiers à trier, de parcourir, en moyenne, p / 2 éléments[1]. Comme les insertions sont faites dans des listes dont la longueur varie de 0 à n -1, p vaut en moyenne (n - 1)/2.

Le coût du tri est donc proportionnel à n(n - 1)/4: on dit que la complexité en temps est en $O(n^2)$: cela signifie que, quand n devient grand, le temps mis pour exécuter TriIns varie comme n^2. Plus précisément, si on appelle T(n) le temps d'exécution de TriIns sur une liste de longueur n, il existe une constante K et une valeur n_0 telles que pour tout $n > n_0$, $T(n) < Kn^2$

6.3.5.3 Complexité du tri fusion

Le coût moyen en temps T(n) du tri fusion d'une liste de longueur n comprend:

- Le coût $K_1 \times n/2$ de la séparation d'une liste de longueur n (n/2 appels récursifs de Séparer).
- Le coût $K_2 \times n$ de la fusion de 2 listes de longueurs n/2 (n appels récursifs de Fusion).
- Deux fois le coût du tri fusion d'une liste de longueur n/2.

On peut donc écrire, en posant $n = 2^k$:

$T(2^k) = 2T(2^{k-1}) + K2^k$
$T(2^{k-1}) = 2T(2^{k-2}) + K2^{k-1}$
...
$T(2^1) = 2T(2^0) + K2^1$
$T(2^0) = 1$

Donc:

1. Il est clair que si on cherche une complexité dans le pire cas (resp. meilleur cas) l'insertion demande de parcourir p (resp. 1) éléments.

$T(2^k) = 2^1T(2^{k-1}) + K2^k$
$T(2^k) = 2^2T(2^{k-2}) + 2^1K2^{k-1} + K2^k$
…
$T(2^k) = 2^iT(2^{k-i}) + 2^{i-1}K2^{k-(i-1)} + \ldots + 2^1K2^{k-1} + K2^k$
…
$T(2^k) = 2^kT(2^0) + 2^{k-1}K2^1 + \ldots + 2^1K2^{k-1} + K2^k$
$T(2^k) = 2^k + Kk2^k + K2^k$

Comme $k = \log_2 n$, on a finalement :

$T(n) = K_3 n\log_2 n + K_4 n$

On dit que la complexité moyenne du tri fusion est en $O(n\log_2 n)$.

On remarquera que, pour $n = 1000$, n^2 vaut 1 000 000 alors que $n\log_2 n$ vaut environ 10 000.

On montre facilement que, toujours en moyenne, la complexité en temps du tri bulle est en $O(n^2)$ alors que celle du tri pivot est en $O(n\log_2 n)$.

6.4 Conclusions

Les listes sont des objets fondamentaux en informatique : elle permettent d'agglomérer des informations dont on ne connaît pas, statiquement, le nombre. Dans un cadre « lispien » elles sont également le moyen le plus pratique pour agglomérer un grand nombre d'objets (dans un cadre impératif, les tableaux[1] sont souvent plus adaptés à cette fonctionnalité).

En tant que structure de base pour ranger des objets, les listes sont un outil indispensable pour mettre en œuvre des abstractions de données réalistes. Le lecteur pourra revenir sur le problème « contrôle de durées » du chapitre précédent et imaginer un certain nombre de traitements, utiles au dépouillement, qui nécessitent des listes de relevés.

Enfin, nous rappelons que de par sa définition, une liste se prête bien à un traitement récursif « à droite ». On peut ainsi modéliser le traitement standard par une fonction d'ordre supérieur[2] dont nous donnons le type et l'algorithme :

Parcours : liste-de-E x (E x R → R) x R → R

De nombreux algorithmes peuvent alors s'exprimer d'une manière élégante. Par exemple :

1. L'étude des structures de données appelées tableaux est abordée dans un autre module d'enseignement du Deug.

2. Une *fonction d'ordre supérieur* est une fonction qui admet en paramètre (ou qui fournit en résultat) au moins une autre fonction.

$$\text{Parcours}(L, \text{Composer}, \text{CasVide}) = \begin{cases} \text{CasVide} \textbf{ si } \text{null?}(L) \\ \text{Composer}(\text{car}(L), \text{Parcours}(\text{cdr}(L), \text{Composer}, \text{CasVide})) \\ \quad \textbf{sinon} \end{cases}$$

- Pour effectuer le et logique de tous les éléments d'une liste de booléens on peut utiliser l'appel Parcours (L, and, #t).
- Pour effectuer la somme de tous les éléments d'une liste d'entiers on peut utiliser l'appel Parcours (L, +, 0).
- Pour calculer le plus grand de tous les éléments d'une liste de naturel on peut utiliser l'appel Parcours (L, Max, 0).

Le lecteur est invité à réfléchir sur d'autres cas d'utilisation de la fonction Parcours.

6.5 Exercices

6.5.1 Constructeurs

Exercice 6.1 Fonction list, cons et Concat

Dessiner les objets suivants et préciser leur type:

```
list(1)
list(1, 2, 3, 4)
list(1,2,list(3, 4))
cons(1, cons(2,3))
cons(1, list(2, 3))
cons(Concat(list(1), list(2)), cons(1, cons(1, nil)))
Concat(list(1, list(2, 3)), list(cons(1, 2), 3, list(4)))
```

6.5.2 Listes plates

Exercice 6.2 Listes plates

1. Concevoir une fonction qui rend le dernier élément d'une liste plate.
2. Concevoir une fonction qui délivre le plus grand élément d'une liste d'entiers.
3. Rédiger une fonction SDP qui, à partir d'une liste d'entiers, calcule la somme des entiers pairs de la liste.
4. Rédiger une fonction EDP qui, à partir d'une liste d'entiers, en extrait la liste des entiers pairs.

5. Concevoir une fonction qui prend en paramètres deux listes d'entiers de même longueur, une liste de poids et une liste de valeurs et qui calcule la somme des valeurs pondérées par leurs poids respectifs.
6. Concevoir une fonction qui calcule le nombre d'occurrences d'un entier donné dans une liste d'entiers.
7. Concevoir une fonction qui ajoute un atome+ à la fin d'une liste plate.
8. Concevoir une fonction qui, à partir d'une liste d'entiers fournit en résultat une liste composée des sommes de 2 éléments consécutifs.

6.5.3 Listes et doublets

Exercice 6.3 Couples d'entiers

- Concevoir une fonction qui, à partir d'une liste d'entiers fournit en résultat la liste des couples d'éléments consécutifs.

Exercice 6.4 Listes et doublets

Dans cet exercice, le type liste-espèces désigne des listes de mots qui sont des noms de certaines espèces d'animaux. Par exemple (list "mammifère" "oiseau" "poisson" "insecte").

À l'aide d'une telle liste on peut coder une espèce par son rang (à partir de 1). dans l'exemple qui précède, le code de "poisson" est 3.

Le type liste-ac désigne des listes de doublets cons(mot, code) où mot est un nom d'animal et code un code d'espèce. Par exemple list(cons("lion", 1), cons("truite", 3), cons("mouche", 4), cons("moineau", 2), cons("chat", 1)).

1. Dessiner une liste-espèces et une liste-ac.
2. Concevoir la fonction E :

Espèce :	mot x liste-ac x liste-espèces	→	mot
	A, B, E	→	nom de l'espèce de A si A est associé, dans B, à un code correspondant à une espèce de E, "échec" sinon

Une liste du type liste-an est une liste de doublets cons(mot1, mot2) où mot1 est un nom d'animal et mot2 un nom d'espèce.

3. Concevoir une fonction ConsAN :

ConsAN :	liste-ac x liste-espèces	→	liste-an
	B, E	→	la liste des animaux de B associés à leurs espèces « décodées » à l'aide de E.

Le type base désigne une liste de doublets cons(mot, L) où mot est un nom d'espèce et L une liste de noms d'animaux.

4. Concevoir une fonction ConsBase :

ConsBase :	liste-ac x liste-espèces	→	liste-an
	B, E	→	une base où chaque espèce de E est associée à tous les noms d'animaux de B qui lui appartiennent.

6.5.4 Quelques types construits

Exercice 6.5 Vecteurs

1. Concevoir et implanter un type abstrait vecteur de taille n.
2. Concevoir une fonction calculant un produit scalaire.

Exercice 6.6 Ensembles d'entiers

1. Définir un type abstrait ensemble d'entiers muni, entre autres, des fonctions d'accès Appartient, Union, Intersection, Différence.
2. Proposer une mise en œuvre où un ensemble d'entiers est implanté par une liste plate sans doubles.
3. Proposer une mise en œuvre où un ensemble d'entiers est implanté par une liste plate triée sans doubles.

Exercice 6.7 Problème des huit reines

On cherche à disposer huit reines sur un échiquier 8 x 8 sans que 2 quelconques d'entre elles puissent se prendre.

- Concevoir une fonction qui délivre une solution au problème des huit reines.

6.5.5 Fonctions d'ordre supérieur

Exercice 6.8 Méthode dichotomique

- Rédiger, en utilisant la méthode dichotomique, une fonction permettant de déterminer un zéro d'une fonction f donnée (on supposera que la fonction a de bonnes propriétés).

Exercice 6.9 Itérants de listes plates

- Utiliser la fonction d'ordre supérieur Parcours pour coder les fonctions Longueur, Concat…

Exercice 6.10 Tris

- Adapter les tris du cours de sorte que la relation d'ordre puisse être choisie à l'appel.

Exercice 6.11 Composition de fonction

- Concevoir une fonction calculant f o g à partir de f et g.
- Concevoir une fonction calculant f^n = f o … o f, n fois, à partir de f et n.

6.5.6 Complexité

Exercice 6.12 Vérification expérimentale

Les temps d'exécution de certains algorithmes de tri sont clairement dépendants de la nature des données sur lesquelles ils s'exécutent. Il faut donc faire une étude de type probabiliste qui est au-delà des buts visés par ce cours.

Nous nous intéressons, dans cet exercice, à une étude expérimentale des temps d'exécution de tels algorithmes.

La fonction suivante, dont l'étude sort du cadre de ce cours, permet de calculer sommairement le temps en secondes pris par l'exécution d'une fonction f sans paramètres.

```
(define chrono (lambda (f)
    (let    ((h     (runtime)))
            (f)
            (/      (- (runtime) h)         100))))
```

Nous rappelons qu'en moyenne les tris par insertion et par extraction demandent un temps d'exécution proportionnel à n^2 alors que, pour les tris pivot et fusion, ce temps est proportionnel à $n \log_2 n$.

Pour obtenir des résultats expérimentaux, on utilisera la fonction:

```
(define UneListe (lambda (n)
    (cond
        ((= n 1)        nil)
        (t              (cons (random 1000) (UneListe (- n 1)))))))
```

1. Sachant que pour n = 128, TriIns demande 2 s et TriFusion 1 s, prévoir un ordre de grandeur du temps pour n = 1024. Vérifier expérimentalement ces prévisions.

On constate, que dans le pire cas, le tri pivot ressemble fâcheusement au tri par insertion.

2. Expliquer pourquoi et vérifier expérimentalement cette propriété.

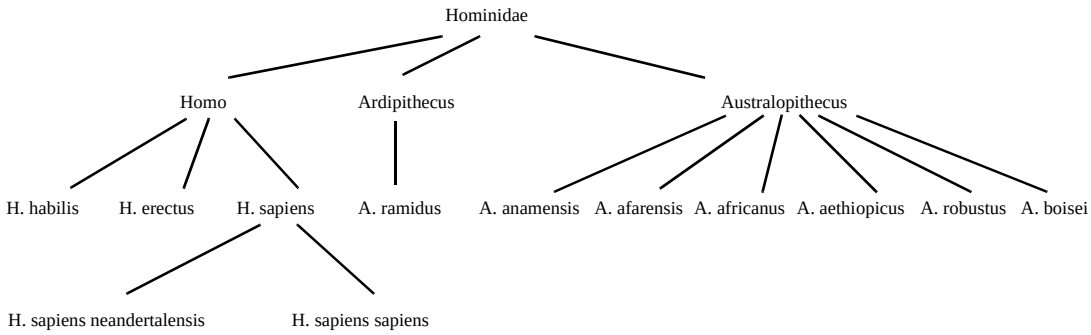
CHAPITRE 7

Arbres

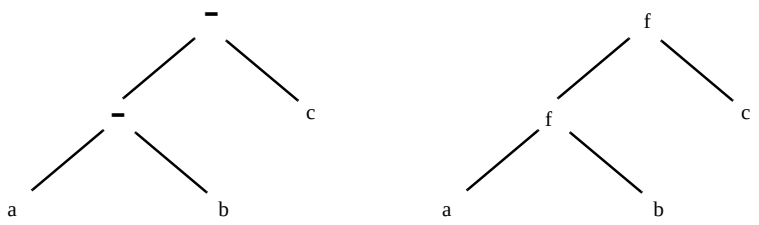
7.1 Introduction

7.1.1 Quelques exemples d'arbres

Le lecteur est prié d'admettre que la notion d'arbre (structure hiérarchique) est particulièrement importante en informatique. Nous n'en donnons qu'une définition informelle et, pour commencer, nous remarquons qu'il est commode de représenter une hiérarchie à l'aide d'un dessin:

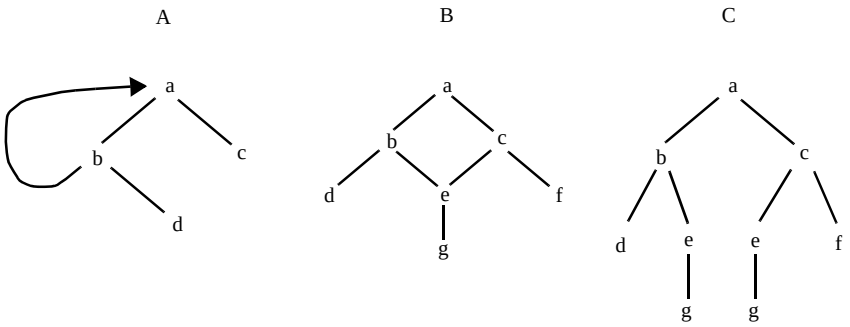


Cette représentation visuelle est particulièrement porteuse d'informations. Par exemple, les deux expressions a - b - c et f(f(a, b), c) ont une structure identique (les règles d'association du - font qu'on combine d'abord a et b et qu'on combine ensuite le résultat avec c) qui devient apparente dès qu'on représente ces expressions avec des arbres.



On remarquera qu'une liste plate est un cas particulier d'arbre (« plat » sur la gauche).

On notera enfin qu'il s'agit de hiérarchies strictes: il n'y a ni boucles ni partage de descendants: sur la figure qui suit, ni l'objet A ni l'objet B ne sont des arbres à notre sens alors que l'objet C, lui, est un arbre qui représente la même réalité que B.



7.1.2 Terminologie

Définition 7.1 Arbre et sous-arbres

Un *arbre* est une structure de données récursive finie comportant:

- une *étiquette* E, qui est une information quelconque,
- une *arité* N qui est un naturel,
- N *fils* qui sont des arbres.

Remarques

- Le fait qu'une telle structure puisse être finie provient de la possibilité d'avoir une arité nulle: dans ce cas il n'y a pas de fils.
- On peut donner facilement une représentation visuelle d'un arbre.

Définitions associées

- Un *sous-arbre* d'un arbre est soit lui-même, soit un sous-arbre d'un de ses fils.
- Un *sous-arbre propre* d'un arbre a est un sous arbre différent de a.
- Dans tout arbre il y a un (et un seul) sous-arbre qui n'est fils d'aucun autre: c'est la *racine* de l'arbre.
- Les arbres qui n'ont pas de fils sont appelés *feuilles*.

On appelle souvent *nœuds* (ou sommets) les sous-arbres d'un arbre. Un nœud est donc caractérisé par:

- Une *étiquette* (la valeur associée au nœud),
- Ses *fils* ou descendants immédiats,
- Son *arité* (le nombre de ses fils).

On utilise souvent le vocabulaire emprunté à la généalogie: frères, pères, ascendants, descendants…

Définition 7.2 Profondeur et hauteur

On dit qu'un nœud n est à la *profondeur* p dans un arbre a si le chemin qui va de la racine de a à n passe par p + 1 nœuds.

On dit qu'un nœud n est à la *hauteur* h dans un arbre a si le plus long chemin qui va de n à une feuille par h + 1 nœuds.

La *hauteur d'un arbre* est la hauteur de sa racine.

Remarques

- La racine de l'arbre est à la profondeur 0; ses fils sont à la profondeur 1…
- Dans un arbre, il existe au moins un nœud m de profondeur maximale; la hauteur de l'arbre est égale à la profondeur de m.
- Toutes les feuilles ont pour hauteur 0.

7.2 Arbres binaires

7.2.1 Définition générale et restrictions

Définition 7.3 Arbre binaire

Un *arbre binaire* est un arbre dont les nœuds sont d'arité 2 ou 0.

Remarque

- Par abus de langage on qualifie parfois un arbre dont les nœuds sont d'arité 2, 1 ou 0 d'arbre binaire. On sous-entend alors que tous les nœuds d'arité 1 ont été complétés par un fils d'arité 0 dont l'étiquette est vide.

Nous cherchons à définir un type abstrait Abin pour manipuler des arbres binaires.

Nous choisissons ici une approche simplificatrice en introduisant un arbre particulier noté Avide qui n'a pas de descendant. Un élément terminal de la hiérarchie sera donc représenté par un nœud dont les deux fils sont des Avides. Une conséquence de ce choix est que les seules feuilles des arbres binaires avec lesquels nous travaillons sont des Avide. Nous noterons Abin le type arbre binaire et Abin+ le type Abin - {Avide}.

Il convient de remarquer que le type Abin est un *cas très particulier d'arbre binaire.*

Le type Abin peut être défini ainsi:

type Abin **volet spécification**
accès

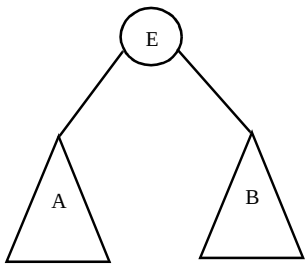
Avide:	Abin		
Abin?:	indifférent	→	booléen
	détermine si un objet est un Abin		
Avide?:	Abin	→	booléen
	détermine si un Abin est un Avide		
ConsA:	indifférent x Abin x Abin	→	Abin+
	construit un Abin+		
Etiq:	$Abin^+$	→	indifférent
Gauche:	$Abin^+$	→	Abin
Droit:	$Abin^+$	→	Abin
	ces trois fonctions délivrent les composantes d'un Abin+ qui portent leurs noms		

fin spécification

Pour la suite, les seuls arbres binaires que nous considérons sont des Abin. Dans la visualisation des arbres, nous nous autoriserons à ne pas représenter les Avide qui terminent la structure.

7.2.2 Parcours standard d'un arbre binaire

Parcourir un arbre c'est visiter tous ses nœuds. Se pose alors le problème du choix d'un ordre pour faire la visite et on a le choix entre au moins trois ordres assez naturels: les parcours *préfixe*, *infixe* et *postfixe*.



- *Préfixe*: visite de E, parcours de A, parcours de B
- *Infixe*: parcours de A, visite de E, parcours de B
- *Postfixe*: parcours de A, parcours de B, visite de E

Remarque
On remarquera l'analogie avec les diverses notations des expressions arithmétiques: la notation usuelle (infixe) et les deux notations dites polonaises, préfixée et suffixée. L'expression a - b - c s'écrit, en notation préfixée, - - a b c. Un des intérêts de cette notation est qu'elle dispense de l'usage des parenthèses: l'expression a - (b -c) s'écrit - a - b c.

Pour illustrer ces trois parcours, on peut imaginer trois fonctions dont le but est de calculer une liste plate des étiquettes des nœuds d'un Abin (les Avide

sont donc exclus de cette liste), la liste étant ordonnée dans l'ordre de la visite. Nous traitons le parcours préfixe de manière complète.

• Soit à écrire la fonction Préfixe:

Préfixe: Abin → plate
a → la liste des étiquettes de a visitées dans l'ordre préfixe

1. Équation récursive:
Préfixe(a) = append(list(Etiq(a)), Préfixe(Gauche(a)), Préfixe(Droit(a)))

2. Il est commode d'utiliser l'ordre:
- a' ⟨ a si et seulement si a' est un sous-arbre propre de a

3. L'équation récursive est mal typée lorsque a est un Avide. En effet, les fonction Etiq, Gauche et Droit ne sont pas définies.

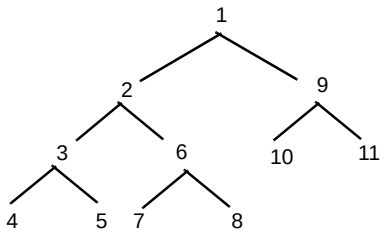
4. Lorsque a est un Avide, comme les Avide sont exclus (de par la spécification) du résultat du parcours, Préfixe(a) = nil.

On obtient alors l'algorithme:

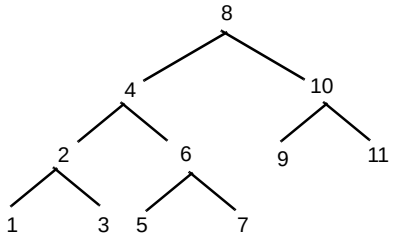
$$\text{Préfixe}(a) = \begin{cases} \text{nil } \textbf{si } \text{Avide?}(a) \\ \text{append}(\text{list}(\text{Etiq}(a)), \text{Préfixe}(\text{Gauche}(a)), \text{Préfixe}(\text{Droit}(a))) \\ \qquad\qquad \textbf{sinon} \end{cases}$$

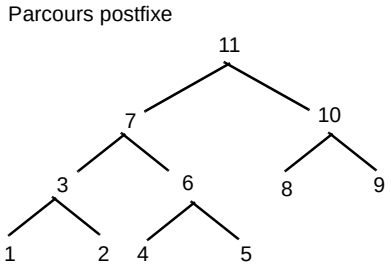
Dans les schémas qui suivent les étiquettes des nœuds ont été choisies afin que le résultat du parcours soit la liste '(1 2 3 4 5 6 7 8 9 10 11). Nous ne représentons pas les arbres vides qui devraient donc être systématiquement ajoutés aux dessins pour avoir une représentation fidèle de la structure abstraite choisie.

Parcours préfixe



Parcours infixe





7.2.3 Mise en œuvre du type arbre binaire

Un nœud comportant une étiquette et deux fils, il est naturel d'utiliser un triplet pour le représenter. D'où:

type Abin volet **mise en œuvre**
principe
- Avide est représenté par nil
- Un nœud (E, G, D) est représenté par cons(E, cons(G, D))

programmation des accès

Avide = nil

Avide? = null?

ConsA(E, G, D) = cons(E, cons(G, D))

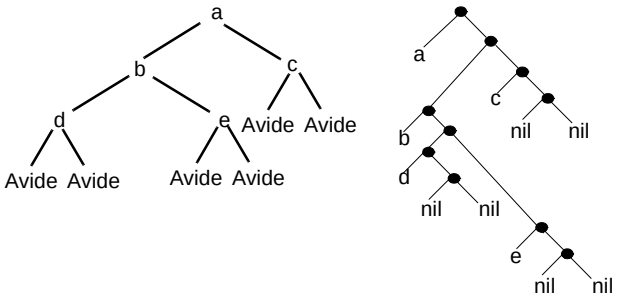
Etiq = car

Gauche = cadr

Droit = cddr

fin mise en œuvre

Pour bien comprendre les avantages de l'abstraction réalisée, il est intéressant de mettre en regard un arbre et sa mise en œuvre.



On réfléchira avec profit à d'autres possibilités de mise en œuvre.

7.2.4 Un exercice d'application

Nous notons ABE le type des arbres binaires dont toutes les étiquettes sont des entiers et ABEtrié un ABE dont tous les sous-arbres vérifient la propriété suivante:

$$(\forall a)(\text{etiquettes}(\text{descendantsgauches}(a)) \leq \text{etiq}(a) \leq \text{etiquettes}(\text{descendantsdroits}(a)))$$

Il en résulte qu'un parcours infixe d'un ABEtrié fournit la suite des étiquettes rangées dans l'ordre croissant. On peut tirer profit de cette propriété pour programmer un tri.

• Soit à écrire la fonction TriArbre:

TriArbre:	L-ent	$\rightarrow$	L-ent
	l	$\rightarrow$	la liste des entiers de l rangés dans l'ordre croissant.

• Faisons l'hypothèse qu'on dispose d'une fonction FaireABET:

FaireABET:	L-ent	$\rightarrow$	ABEtrié
	l	$\rightarrow$	un ABEtrié contenant tous les entiers de l

On peut alors écrire:

TriArbre(l) = Infixe(FaireABET(l))

• Si on fait maintenant l'hypothèse qu'on dispose d'une fonction Placer:

Placer:	entier x ABEtrié	$\rightarrow$	ABEtrié
	n, a	$\rightarrow$	un ABEtrié contenant n ainsi que tous les entiers de a

on peut écrire:

$$\text{FaireABET}(l) = \begin{cases} \text{Avide } \textbf{si } \text{null?}(l) \\ \text{Placer}(\text{car}(l), \text{FaireABET}(\text{cdr}(l))) \textbf{ sinon} \end{cases}$$

Le sous-problème Placer est un peu plus délicat. On arrive à l'équation récursive:

$$\text{Placer}(n, a) = \begin{cases} \text{ConsA}(\text{Etiq}(a), \text{Placer}(n, \text{Gauche}(a)), \text{Droit}(a)) \textbf{ si } n \leq \text{Etiq}(a) \\ \text{ConsA}(\text{Etiq}(a), \text{Gauche}(a), \text{Placer}(n, \text{Droit}(a))) \textbf{ sinon} \end{cases}$$

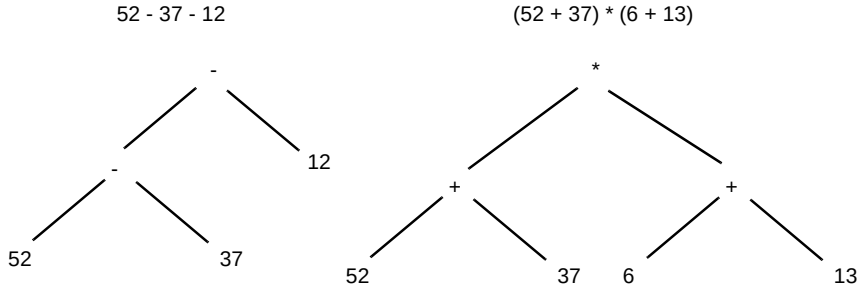
La méthode amène à traiter le cas où l'arbre est vide et on obtient:

$$\text{Placer}(n, a) = \begin{cases} \text{ConsA}(n, \text{Avide}, \text{Avide}) \textbf{ si } \text{Avide?}(a) \\ \textbf{sinon} \begin{cases} \text{ConsA}(\text{Etiq}(a), \text{Placer}(n, \text{Gauche}(a)), \text{Droit}(a)) \textbf{ si } n \leq \text{Etiq}(a) \\ \text{ConsA}(\text{Etiq}(a), \text{Gauche}(a), \text{Placer}(n, \text{Droit}(a))) \textbf{ sinon} \end{cases} \end{cases}$$

7.3 Quelques études de cas

7.3.1 Évaluation d'expressions arithmétiques

On peut représenter des expressions arithmétiques à l'aide d'arbres dont les étiquettes des nœuds sont des opérateurs et celles des feuilles des nombres entiers.



Pour manipuler de tels arbres nous définissons une abstraction adéquate, le type EA qui se décompose en deux sous-types, EA+ et EA-: EA- est le type des feuilles et EA+ = EA - EA-.

type EA **volet spécification**
accès

ConsFeuille:	entier	→ EA-
	construit une feuille à partir d'un entier	
Feuille?:	EA	→ booléen
	détermine si un objet est une EA-	
ValFeuille:	EA-	→ entier
	délivre la valeur entière d'une feuille	
ConsEA:	mot x EA x EA	→ EA+
	construit une expression à l'aide d'un opérateur et de deux opérandes	
Oper?:	EA	→ booléen
	détermine si un objet est une EA+	
Oper:	EA^+	→ mot
Gauche:	EA^+	→ EA
Droit:	EA^+	→ EA
	ces trois opérations permettent d'obtenir les composantes d'une EA+	

fin spécification

On réfléchira avec profit aux propriétés qui relient les diverses fonctions d'accès.

• Soit à écrire la fonction Eval:

Eval: EA → entier
e → la valeur de l'expression e.

Lorsqu'on évalue une EA, deux cas sont à prendre en compte: soit l'étiquette du nœud racine est un opérateur et on l'utilise pour combiner le résultat de l'évaluation de ses fils gauche et droit, soit l'étiquette du nœud est un entier, et sa valeur est celle de l'expression. D'où l'algorithme:

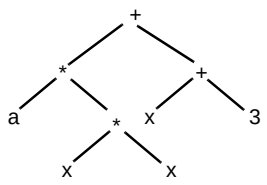
$$\text{Eval}(e) = \begin{cases} \text{ValFeuille}(e) \textbf{ si } \text{Feuille?}(e) \\ (\text{Eval}(\text{Gauche}(e)) + \text{Eval}(\text{Droit}(e))) \textbf{ si } \text{Oper}(e) = \textbf{"+"} \\ (\text{Eval}(\text{Gauche}(e)) - \text{Eval}(\text{Droit}(e))) \textbf{ si } \text{Oper}(e) = \textbf{"-"} \\ (\text{Eval}(\text{Gauche}(e)) \times \text{Eval}(\text{Droit}(e))) \textbf{ si } \text{Oper}(e) = \text{"*"} \\ (\text{Eval}(\text{Gauche}(e)) / \text{Eval}(\text{Droit}(e))) \textbf{ si } \text{Oper}(e) = \text{"/"} \end{cases}$$

Il faut adjoindre à ce programme une mise en œuvre du type EA. Pour mettre en œuvre le type EA on peut utiliser le type Abin+ vu au paragraphe précédent: les entiers auront deux fils qui sont des Avides.

type EA **volet mise en œuvre**
principe
Une EA est implantée par un Abin. Les Feuilles de l'EA correspondent à un Abin dont l'étiquette est un entier.
algorithmes des accès
ConsFeuille(i) = ConsA(i, Avide, Avide)
Feuille?(e) = integer?(Etiq(e))
ValFeuille = Etiq
ConsEA = ConsA
Oper?(e) = Mot?(Etiq(e))
Oper = Etiq
Gauche = Gauche
Droit = Droit
fin mise en œuvre

7.3.2 Dérivée d'une fonction simple

Nous considérons des expressions formées à l'aide des opérateurs + et *, de nombres et de lettres représentant des variables, et nous utilisons des arbres (EAV) pour les visualiser.



$(ax^2 + x + 3)$

On souhaite calculer la dérivée par rapport à une variable donnée d'une telle expression.

Dériv:	EAV x mot+	→	EAV
	e, x	→	la dérivée de e par rapport à x.

Comme dans l'exemple précédent, nous définissons une abstraction (le type EAV) propre à modéliser l'objet sur lequel on travaille.

type EAV **volet spécification**
accès

ConsFeuille:	mot+	→	EAV-
	construit une feuille avec une variable ou une constante		
Feuille?:	EAV	→	booléen
	détermine si un objet est une feuille		
ValFeuille:	EAV-	→	mot+
	donne la valeur d'une feuille		
Variable?:	EAV- x mot+	→	booléen
	permet de déterminer si la valeur d'une feuille est égale à une valeur donnée		
ConsEAV:	mot^+ x EAV x EAV	→	EAV+
	construit une EAV		
Oper?:	EAV	→	booléen
	détermine si une EAV est une EAV+		
Oper:	EAV^+	→	mot^+
Gauche:	EAV^+	→	EAV
Droit:	EAV^+	→	EAV
	ces trois fonctions extraient les composantes d'une EAV		

fin spécification

Compte tenu de cette abstraction, l'équation récursive de Dériv est simple, puisqu'il suffit d'utiliser les formules de dérivation d'un produit ou d'une somme:

(uv)' = u'v + v'u
(u + v)' = u' + v'

On peut introduire, afin d'améliorer la lisibilité, les fonctions:

Dprod:	EAV x EAV x mot	→	EAV
	u, v, x	→	la dérivée par rapport à x de (uv)
Dsom:	EAV x EAV x mot	→	EAV
	u, v, x	→	la dérivée par rapport à x de (u + v)
Plus:	EAV x EAV	→	EAV
	g, d	→	(g + d)
Mult:	EAV x EAV	→	EAV
	g, d	→	(g x d)

L'analyse des cas particuliers est simple puisque les éléments terminaux de la structure sont soit des constantes (dérivée 0), soit la variable x (dérivée 1). On obtient l'algorithme:

Plus(g, d) = ConsEAV(**"+"**, g, d)
Mult(g, d) = ConsEAV(**"*"**, g, d)

$$Dprod(u, v, x) = \begin{cases} \textbf{soit } du = Dériv(u, x), dv = Dériv(v, x) \textbf{ dans} \\ Plus(Mult(du, v), Mult(u, dv)) \end{cases}$$

$$Dsom(u, v, x) = Plus(Dériv(u, x), Dériv(v, x))$$

$$Dériv(e, x) = \begin{cases} \textbf{si } Feuille?(e) \begin{cases} ConsFeuille("1") \textbf{ si } Variable?(e, x) \\ ConsFeuille("0") \textbf{ sinon} \end{cases} \\ \textbf{sinon} \begin{cases} Dsom(gauche(e), droit(e), x) \textbf{ si } Oper(e) \textbf{= "+"} \\ Dprod(gauche(e), droit(e), x) \textbf{ si } Oper(e) \textbf{= "*"} \end{cases} \end{cases}$$

La mise en œuvre du type EAV ne pose aucun problème particulier.

type EAV **volet mise en œuvre**
principe

Une EAV est implantée par un Abin. Les Feuilles de l'EAV correspondent à un Abin dont l'étiquette est soit un mot, soit un entier.

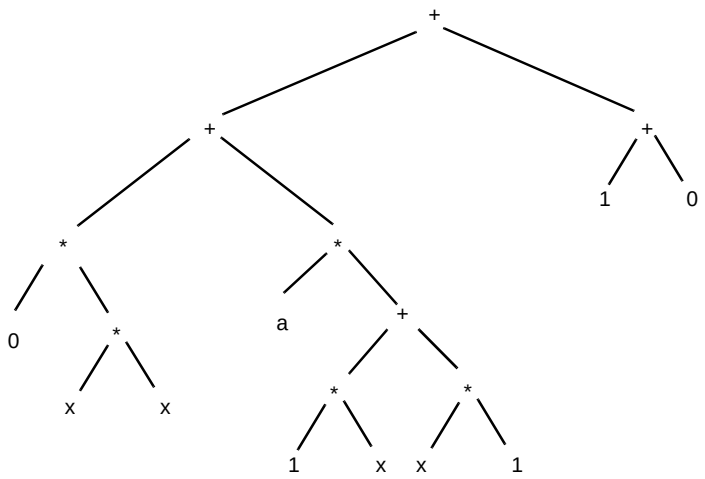
algorithmes des accès

ConsFeuille(i) = ConsA(i, Avide, Avide)
Feuille?(e) = Avide?(Droit(e))
ValFeuille = Etiq
Variable?(e, x) = Feuille?(e) ∧ Mot?(Etiq(e))
ConsEAV = ConsA
Oper?(e) = ¬Feuille?(e)
Oper = Etiq
Gauche = Gauche
Droit = Droit

fin mise en œuvre

7.3.3 Simplification d'une expression

Si on utilise la fonction qu'on vient d'écrire pour dériver l'expression de la figure du paragraphe précédent, on obtient un résultat juste, certes, mais particulièrement rébarbatif.



C'est un problème très difficile que de transformer une telle expression pour arriver à un résultat aussi compact que 2ax + 1. En revanche, il est tout à fait simple de procéder à quelques simplifications évidentes, en se contentant d'utiliser les règles:

0 + x = x
x + 0 = x
0 * x = 0
x * 0 = 0
1 * x = x
x * 1 = x

Simpl:	EAV	→	EAV
	e	→	une expression e plus simple, obtenue en utilisant au mieux les règles précédentes.

Il y a toutefois un piège à éviter. Pour l'illustrer, nous considérons, à titre d'exemple, la première règle, 0 + x = x.

Si on écrit, en prenant quelques libertés syntaxiques, l'équation récursive:

$$\text{Simpl}(g + d) = \begin{cases} d \textbf{ si } g=0 \\ \text{Simpl}(g) + \text{Simpl}(d) \textbf{ sinon} \end{cases}$$

on manque quelques simplifications, comme on peut s'en convaincre sur l'exemple donné plus haut: le fils gauche de la racine (+) a un opérande gauche qui est nul et qui n'est pas capté par l'équation que nous venons d'écrire. La bonne équation est donc:

$$\text{Simpl}(g + d) = \begin{cases} \text{Simpl}(d) \textbf{ si } \text{Simpl}(g)=0 \\ \text{Simpl}(g) + \text{Simpl}(d) \textbf{ sinon} \end{cases}$$

Pour exprimer cet algorithme sous une forme élégante et lisible, on peut introduire deux fonctions élémentaires chargées d'appliquer les règles de simplifications d'une somme ou d'un produit à des opérandes *déjà* réduits. En faisant appel à deux fonctions spécifiques pour traiter les opérateurs:

RèglePlus:	EAV x EAV	→	EAV
	g, d	→	l'expression obtenue en appliquant directement les règles de simplification d'une somme à l'expression g + d
RègleMult:	EAV x EAV	→	EAV
	g, d	→	l'expression obtenue en appliquant directement les règles de simplification d'un produit à l'expression g * d

On obtient facilement l'algorithme de la fonction Simpl, en prenant la précaution de simplifier *au préalable* les paramètres fournis aux deux fonctions RèglePlus et RègleMult.

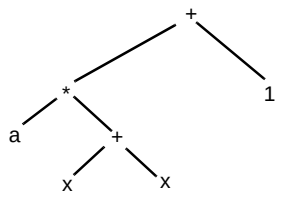
$$
\text{Simpl}(e) = \begin{cases} e \textbf{ si } \text{Feuille?}(e) \\ \textbf{sinon} \begin{cases} \text{RèglePlus}(\text{Simpl}(\text{Gauche}(e)), \text{Simpl}(\text{Droit}(e))) \\ \quad \textbf{si } \text{Oper}(e) \textbf{="+"} \\ \text{RègleMult}(\text{Simpl}(\text{Gauche}(e)), \text{Simpl}(\text{Droit}(e))) \\ \quad \textbf{si } \text{Oper}(e) \textbf{="*"} \end{cases} \end{cases}
$$

Les fonctions d'application des règles ne posent que des difficultés techniques: il suffit d'être conscient du fait que l'expression 0, par exemple, n'est pas représentée par l'atome 0, mais par une EAV.

$$
\text{RèglePlus}(g, d) = \begin{cases} d \textbf{ si } g\text{=ConsFeuille}(\text{"0"}) \\ g \textbf{ si } d\text{=ConsFeuille}(\text{"0"}) \\ \text{Plus}(g, d) \textbf{ sinon} \end{cases}
$$

$$
\text{RègleMult}(g, d) = \begin{cases} g \textbf{ si } g\text{=ConsFeuille}(\text{"0"}) \\ d \textbf{ si } d\text{=ConsFeuille}(\text{"0"}) \\ d \textbf{ si } g\text{=ConsFeuille}(\text{"1"}) \\ g \textbf{ si } d\text{=ConsFeuille}(\text{"1"}) \\ \text{Mult}(g, d) \textbf{ sinon} \end{cases}
$$

Sur l'exemple précédent le résultat produit est:



7.4 Conclusions

La structure d'arbre présente, par rapport à la structure de liste, la particularité de se prêter à des récursivités multiples. Pour le cas particulier de la structure d'arbre binaire (on peut généraliser aux arbres d'arité quelconque) il s'agit de composer un traitement récursif à gauche et à droite avec le traitement de l'étiquette du nœud. Suivant la place du traitement de l'étiquette on rentre dans le cadre d'une des trois grandes classes de parcours, préfixe (parfois appelé *descendant gauche-droit* ou *DGD*), infixe (*symétrique*), ou postfixe (*ascendant gauche-droit* ou *AGD*).

Le schéma général, paramétré par une fonction de traitement du cas d'arrêt (K) et par une fonction de composition des résultats récursifs (C), est le suivant:

$$f(a, K, C) = \begin{cases} K \textbf{ si } Avide?(a) \\ C(f(Gauche(a), K, C), Etiq(a), f(Droit(a), K, C)) \textbf{ sinon} \end{cases}$$

7.5 Exercices

7.5.1 Arbres binaires d'entiers

Exercice 7.1 Recherche dans un arbre binaire

- Concevoir une fonction Recherche permettant de déterminer si un entier donné est une des étiquettes d'un ABE.

Exercice 7.2 Recherche dans un arbre binaire trié

1. Concevoir une fonction Recherche permettant de déterminer si un entier donné est une des étiquettes d'un ABEtrié.
2. Concevoir une fonction Max qui recherche la plus grande étiquette d'un ABEtrié.
3. Concevoir une fonction NbEtiqSup qui calcule le nombre de nœuds d'un ABEtrié qui ont une étiquette supérieure à une valeur donnée.
4. Concevoir un type abstrait ABEtrié.

7.5.2 Arbres binaires

Exercice 7.3 Mobiles

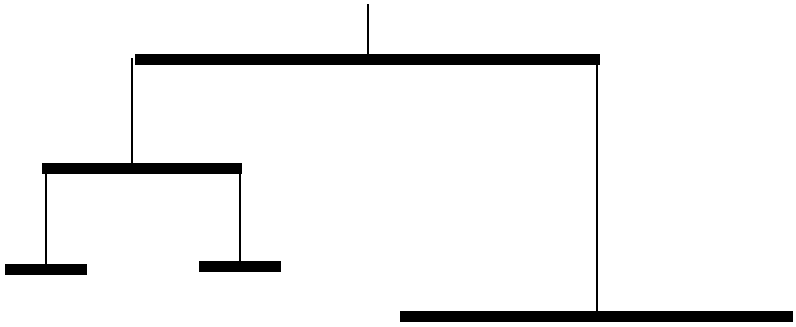
On considère des mobiles formés d'une tige principale aux extrémités de laquelle sont suspendus par leur milieu deux autres sous-mobiles. Chaque tige est caractérisée par son poids.

Un mobile est dit *équilibré* si toutes ses tiges sont horizontales.

On représente un mobile par un arbre binaire (type Abin) tel que:

- l'étiquette de la racine est le poids de la tige principale,
- le sous-arbre gauche représente le sous-mobile gauche,
- le sous-arbre droit représente le sous-mobile droit,

Le mobile du dessin est équilibré.



On le représentera par l'arbre suivant:

ConsA(9, ConsA(4, ConsA(2, Avide, Avide), ConsA(2, Avide, Avide)), ConsA(8, Avide, Avide))

- Ecrire une fonction Equil? qui teste si un mobile est équilibré.

7.5.3 Abstractions arborescentes

Exercice 7.4 Expressions booléennes

Une expression booléenne est formée à l'aide de lettres qui représentent des *variables* propositionnelles (variables dont les valeurs possibles sont vrai ou faux), de connecteurs (on prendra ici les connecteurs et, ou et non) et de parenthèses.

1. Rappeler les règles implicites de parenthésage; rappeler les tables de vérité des trois connecteurs.
2. Définir et mettre en œuvre un type EB pour représenter des expressions booléennes.
3. Concevoir une fonction Val qui, à partir d'une EB e et d'une liste L de doublets cons(variable, valeur booléenne) calcule la valeur de e.

Exercice 7.5 Expressions booléennes normalisées

Une expression booléenne normalisée EBN, est une EB telle que tout connecteur non ne porte que sur une variable booléenne.

1. Concevoir une fonction Neg qui à une EBN associe une EBN représentant sa négation.
2. Concevoir une fonction Normaliser qui, à partir d'une EB, calcule une EBN équivalente.

7.5.4 Étude de cas

Exercice 7.6 Jeu des animaux

Le « jeu des animaux » oppose un joueur et un savant. Le joueur « pense à un animal ». Le savant pose au joueur des questions dont la réponse ne peut être que oui ou non. Si le savant trouve l'animal choisi par le joueur il mar-

que un point. Dans le cas contraire, c'est le joueur qui marque un point, mais le savant demande alors au joueur de lui apprendre une question qui lui aurait permis de découvrir l'animal.

Au départ le savant ne connaît qu'une seule question « est-ce un mammifère? » et deux animaux, chat et moineau. On obtient des séquences de jeu comme celle qui suit.

Le joueur pense à un lapin.

S: Est-ce un mammifère?
J: Oui.
S: Je crois que vous pensez à « chat ».
J: Non.
S: A quoi pensiez-vous?
J: lapin
S: Donnez-moi une question pour distinguer « lapin » et « chat ».
J: Est-ce un rongeur?

Le joueur pense à une vache.
S: Est-ce un mammifère?
J: Oui.
S: Est-ce un rongeur?
J: Non.
S: Je crois que vous pensez à « chat ».
J: Non.
S: A quoi pensiez-vous?
J: vache
S: Donnez-moi une question pour distinguer « vache » et « chat ».
J: Est-ce un ruminant?
.....

- Programmer le savant.

CHAPITRE 8

Conclusions

8.1 Calculs sous-jacents et complexité

À l'occasion de la présentation de certains algorithmes de tri nous avons donné quelques éléments de réflexion sur la complexité en temps des algorithmes.

La complexité en espace (utilisation de la mémoire de l'ordinateur) est difficile à mesurer dans le cadre de la programmation fonctionnelle. On rappelle toutefois qu'un message d'erreur comme *Recursion too deep : Stack overflow* signifie que l'ordinateur manque de place pour exécuter le programme qui lui a été confié. Très souvent il s'agit d'un problème d'algorithme (qui boucle). Quand l'algorithme est juste c'est qu'il demande, compte tenu des données sur lesquelles l'exécution particulière a été faite, trop de place.

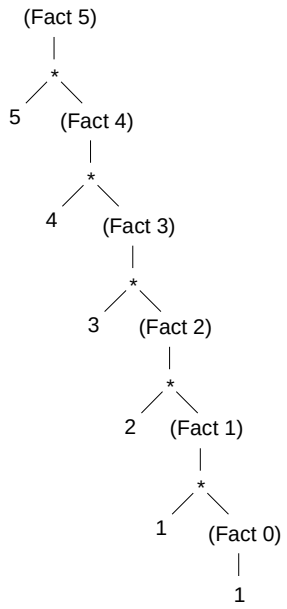
Temps d'une part et place (mémoire, espace… ces mots sont synonymes) d'autre part, sont donc les deux unités de consommation des programmes informatiques et on ne peut pas toujours gagner sur les deux tableaux.

8.1.1 Exécutions récursives et coûts

Considérons la fonction :

```
;Fact:      naturel      ->     naturel
;           n            ->     n!
(define Fact (lambda (n)
      (cond
            ((= n 0)     1)
            (t           (* n (Fact (- n 1)))))))
```

On peut modéliser les calculs engendrés par l'expression (Fact 5) à l'aide de l'arbre:



Cet arbre représente le calcul qu'effectue la machine. Ce calcul est ce que nous appellerons un processus récursif: la première multiplication en haut de l'arbre ne pourra être effectuée que lorsqu'on disposera du résultat du sous-arbre droit. Le développement de ce sous-arbre droit engendre donc une chaîne de calculs différés (on se souvient qu'il faudra, plus tard, multiplier 5 par quelque chose, multiplier 4 par quelque chose).

On souhaite généralement avoir une idée des performances des programmes qu'on rédige. Il faut donc savoir dans quelles unités on cherche à évaluer les dites performances (temps et espace).

- Nous admettrons, pour l'instant, que le temps est proportionnel au nombre de nœuds de l'arbre qui représente le calcul.
- Très intuitivement, dans le cas précédent, l'évaluation différée des * oblige à conserver les branches gauches de l'arbre. L'espace nécessaire est donc le nombre de nœuds qu'il faut conserver pour les évaluations différées.

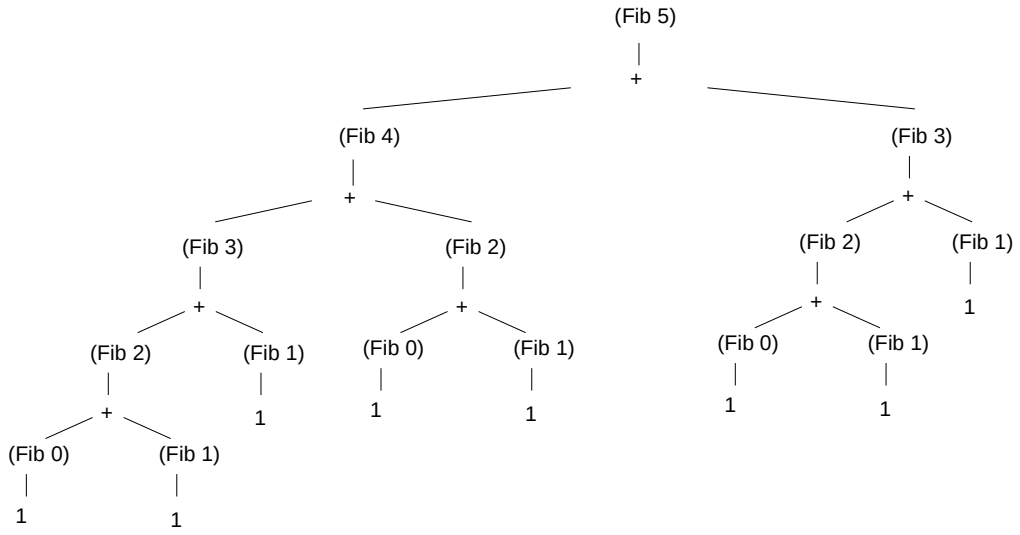
Pour ce qui concerne Fact, on dira donc que (Fact n) demande un temps et un espace en $O(n)$.

La suite de nombres dite de Fibonacci est la suite 0, 1, 1, 2, 3, 5, 8, 13, 21… qu'on peut définir par la règle:

$$Fib(n) = \begin{cases} 0 \textbf{ si } n = 0 \\ 1 \textbf{ si } n = 1 \\ Fib(n-1) + Fib(n-2) \textbf{ si } n > 1 \end{cases}$$

D'où la fonction:

```
;Fib:        naturel      ->     naturel
;            n            ->     n-ieme terme de la suite de Fibonacci
(define Fib (lambda (n)
      (cond
            ((= n 0)     0)
            ((= n 1)     1)
            (t           (+ (Fib (- n 1)) (Fib (- n 2)))))))
```



Du point du vue des performances on s'aperçoit que le nombre de nœuds de l'arbre n'est pas proportionnel à n: il croît de manière exponentielle avec n. Pour ce qui concerne l'espace, on constate que la longueur maximum d'une chaîne d'évaluations différées (en effectuant un parcours naturel de l'arbre) est celle de la plus longue branche. Le coût en espace est donc proportionnel à n.

8.1.2 Schémas itératifs

Considérons le programme suivant qui calcule une factorielle (nous discuterons plus tard de la manière dont ce programme a pu être conçu).

```
;Fact:        naturel      ->     naturel
;            n            ->     n!
(define Fact (lambda (n)
      (FactIterIter 1 1 n)))

;FactIter:    naturel x naturel x naturel    ->     naturel
```

```
;               --- voir plus loin ---
(define FactIter (lambda (f c n)
      (cond
            ((> c n)        f)
            (t              (FactIter (* f c) (+ c 1) n)))))
```

Si on développe un calcul, on obtient:

```
(Fact 6)
(FactIter   1     1     6)
(FactIter   1     2     6)
(FactIter   2     3     6)
(FactIter   6     4     6)
(FactIter   24    5     6)
(FactIter   120   6     6)
(FactIter   720   7     6)
720
```

On constatera, en réfléchissant sur l'arbre de calcul, que le coût en espace est constant: *il n'y a pas de chaîne d'évaluation différée*. En fait, le premier paramètre de FactIter contient, à chaque pas, toute l'information utile pour la suite. Un tel schéma de calcul est dit *itératif*.

Pour comprendre la manière dont on a pu concevoir un tel programme, cherchons à spécifier la fonction FactIter. On s'aperçoit qu'on ne peut pas y arriver simplement sans faire des restrictions sur ses paramètres. Modulo une restriction de ce type, on peut aboutir à:

FactIter: naturel x naturel x naturel → naturel
f, c, n → n!, **si f=(c-1)! et c <= n+1** (P)

La partie en gras (propriété P) de cette spécification est ce qu'on appelle une *pré-condition* (elle doit être vraie avant tout appel pour que la fonction produise le résultat attendu). Pour réfléchir sur un programme complet, il faut:

- Montrer que si la pré-condition de FactIter est vraie, alors la post-condition (le résultat est égal à n!) est vraie également.
- Vérifier que tout appel satisfait la pré-condition;

1. La pré-condition implique la post-condition

En examinant FactIter on a deux cas à considérer:

- Première branche du cond: c > n et (P) impliquent c = n + 1. Comme de (P) on a f = (c - 1)!, on en déduit que le résultat est n!.
- Deuxième branche du cond: comme c <= n on peut déduire que, si (P) est vrai pour l'appel en cours, la précondition de l'appel récursif est vérifiée (si on appelle f', c' et n' les paramètres de cet appel, on a f' = f * c, c' = c +1 et n '= n). Donc le résultat est bien n!.

2. L'appel initial satisfait bien (P).

Une autre manière de décrire le même phénomène consiste à adopter l'approche suivante: FactIter engendre une série d'appels que nous avons qualifiée de schéma itératif. Cette série d'appels correspond aux valeurs successives de c: 1, 2, 3… À tout schéma itératif on doit pouvoir associer une propriété *invariante* (ici f = (c - 1)! et c <= n +1 (on remarquera qu'on retrouve la pré-condition) et une *condition de terminaison* (ici, compte-tenu des valeurs prises par c, c = n +1). La satisfaction conjointe de l'invariant et de la condition de terminaison doit impliquer la propriété visée pour le résultat (ce qui est bien le cas, puisque le résultat f est alors égal à n!).

Appliquons ce mode de raisonnement pour construire une version itérative de la fonction Fib.

Idées

Le schéma sera gouverné par un paramètre c qui doit prendre successivement les valeurs 2, 3,…, n. Pour maintenir un invariant intéressant, on doit probablement disposer d'informations permettant de calculer Fib(c). En considérant la définition de Fib on est assez naturellement amené à retenir les valeurs Fib(c -1) et Fib(c -2).

D'où l'idée de retenir 4 paramètres a, b, c, n pour FibIter et l'invariant:

a = Fib(c -2), b = Fib(c -1), c <= n +1

Programme

```
;Fibiter:    naturel x naturel x naturel x naturel      ->    naturel
;            a, b, c, n                                 ->    n ieme terme de la
;                              suite de Fibonacci, sachant que :
;                              a est le c-2 ieme
;                              b le c-1 ieme
;                              c <= n+1
(define FibIter (lambda (a b c n)
      (cond
            ((> c n)          b)
            (t                (FibIter b (+ a b) (+ c 1) n)))))


;Fib:        naturel      ->    naturel
;            n            ->    n ieme terme de la suite de Fibonacci
(define (Fib (lambda (n)
      (cond
            ((= n 0)          0)
            ((= n 1)          1)
            ((> n 1)          (FibIter 0 1 2 n)))))
```

8.2 Schémas itératifs, variables et programmation impérative

Le style « itératif » de programmation n'a guère d'utilité en programmation fonctionnelle: il rend les algorithmes peu lisibles. On l'utilise « contraint et forcé » dans les deux cas de figure suivants:

- On utilise des outils non-fonctionnels du langage (effets de bord: voir annexe A).
- On a des problèmes de coût impossibles à régler autrement.

Notons sur le second point que la technologie du fonctionnel ayant fait des progrès, de nombreux compilateurs sont capables de découvrir, le cas échéant, la possibilité d'exécuter l'algorithme dans un style « itératif ».

En revanche, le style « itératif » prend tout son sens dans les langages impératifs qui offrent la possibilité d'utiliser et de modifier directement la mémoire: ce qu'on appelle variable en informatique (emplacement mémoire) est directement associé à la notion d'itération (boucle).

8.3 Perspectives et développements

Deux autres modules d'informatique proposés en Deug sont destinés à poursuivre cette initiation.

- Le module « méthodes et outils de l'informatique: approche impérative » qui introduit les notions de variable et d'itération et qui reprend les méthodes introduites dans ce cours sous l'angle de la programmation impérative.
- Le module « initiation au génie logiciel » qui aborde les problèmes posés par l'écriture de logiciels de taille significative et qui donne quelques notions sur des techniques mathématiques liées à l'informatique: la logique et la théorie des langages.

8.4 Conclusions

La programmation est un art difficile qui fait appel à beaucoup de technique et qui demande de la créativité. Il est illusoire de penser, qu'après une courte initiation, l'étudiant puisse appréhender le domaine dans sa globalité. Sur le fond, nous aimerions qu'il ait retenu deux maximes simples:

- Programmer, c'est d'abord réfléchir. L'approche qui consiste à aligner, au kilomètre, des lignes de code et à les tester vaille que vaille est non seulement ridicule, mais aussi dangereuse. Rien ne remplace une bonne analyse du problème et, parmi les outils d'analyse, l'abstraction joue un rôle fondamental.
- Un programme est un objet d'étude. On doit pouvoir raisonner sur le programme qu'on écrit (pour se prouver qu'il fait ce qu'on veut, pour comprendre ses dysfonctionnements éventuels, pour l'expliquer…). En ce domaine, la structure (ou l'absence de structure) du programme joue également un rôle privilégié.

Au niveau des acquis techniques, l'étudiant doit être capable de concevoir des abstractions simples (fonctions ou données) et doit avoir assimilé les principes de base de la récursivité.

ANNEXE A

Séquentialité et effets de bord

A.1 Introduction

En sus des aspects conceptuels et méthodologiques, l'apprentissage de la programmation nécessite également des outils techniques plus ou moins dépendants du langage utilisé.

Les outils techniques vus dans ce cours permettent d'écrire un programme Scheme (ie. une fonction) qui à un naturel n associe n!. En revanche, ils ne permettent pas d'écrire un programme qui:

- affiche à l'écran un message demandant de frapper un naturel au clavier,
- acquiert au clavier la valeur d'un naturel,
- calcule sa factorielle,
- affiche à l'écran le résultat.

Concevoir des programmes de dialogue avec l'utilisateur demande également des outils conceptuels et méthodologiques. Ces outils sont complexes et nous avons opté pour d'autres priorités. Toutefois, nous savons par expérience combien il est frustrant, pour le débutant, de ne pas utiliser d'entrées / sorties. Pour ces raisons, nous donnons dans cette annexe quelques éléments techniques pour réaliser des entrées / sortie en Scheme.

Le fait que ce discours soit donné en annexe marque toutefois notre volonté de considérer ces techniques comme marginales dans le cadre de la première initiation.

A.2 Définitions

A.2.1 Séquentialité

Nous enrichissons la définition d'une expression Scheme en introduisant une nouvelle construction du langage, la **séquence**.

Définition 1.1 Expression Scheme: séquence

Syntaxe

<expression> ::= <*notation*> | <*identificateur*> | <*appel*> | <*définition*> | <*fonction*> | <*conditionnelle*> | <*soit*> | <séquence>

<séquence> ::= **(begin** <expression$_1$> … <expression$_n$> **)**

Sémantique

- Les expressions de la séquence sont évaluées dans l'ordre.
- Le résultat est celui de la dernière expression.

Exemple
[1] (+ 1 1)
2
[2] (* 2 3)
6
[3] (begin (+ 1 1) (* 2 3))
6

On remarque que, si on se limite à ce qui a été présenté du langage Scheme, les expressions

```
(* 2 3)
(begin (* 2 3))
(begin (+ 1 1) (* 2 3))
(begin (- 4 1) (+ 1 1) (* 2 3))
…
```

donnent le même résultat et, par conséquent, on peut légitement penser que les n-1 premières expressions d'une séquence ne servent à rien! Pour utiliser la séquence, il faut donc disposer de fonctionnalités du langage qui la rendent utile.

A.2.2 Effets de bord

Ce qu'on appelle un effet de bord est une action de l'interpréteur qui laisse une trace durable. L'interpréteur Scheme, à la fin de l'évaluation d'une expression, affiche son résultat. C'est un effet de bord qui laisse une trace à l'écran pendant un certain temps. Il est important de noter que le moment où l'effet de bord est produit est important: dans une session Scheme, les résultats des évaluations successives sont affichés dans l'ordre.

On dispose, en Scheme, de fonctions réalisant des effets de bord; la séquence est alors un outil qui permet de gérer l'ordre de leurs occurrences. Ainsi, la fonction write, prédéfine, peut être décrite ainsi (par la suite, nous mettons systématiquement en gras ce qui, dans les spécifications, concerne les effets de bord):

write: indifférent → ∅
a → aucun résultat (indéfini), **mais, produit un effet de bord qui consiste à afficher, à l'écran, la valeur de a**

Ainsi,

[1] (+ 1 1)
2
[2] (* 2 3)
6
[3] (begin (write (+ 1 1)) (* 2 3))
2 6
[4] (begin (write (- 4 1)) (write (+ 1 1)) (* 2 3))
3 2 6

A.3 Des effets de bord particuliers: les entrées/sorties

A.3.1 Sorties

Les principales fonctions de sortie sont, outre write que nous venons de définir,

display: indifférent → ∅
a → aucun résultat (indéfini), **mais, produit un effet de bord qui consiste à afficher, à l'écran, la valeur de a**

Les fonctions display et write ne diffèrent que par l'allure de l'affichage: le format utilisé par display étant plus proche des habitudes visuelles du lecteur standard.

newline: ∅ → ∅
→ aucun résultat (indéfini), **mais, déplace, sur l´écran, le curseur au début de la ligne suivante.**

A.3.2 Entrées

Nous ne donnons qu'une fonction d'entrée, read.

read: ∅ → indifférent
→ **acquiert, au clavier, une valeur** et la délivre en résultat.

Exemple

[1] (begin (display "ici") (newline) (+ (read) 1))
ici
2 3

A.4 Méthodologie d'utilisation des entrées/sorties

On peut faire les pires programmes lorsqu'on utilise des entrées / sorties sans discernement. Nous donnons ci-après trois cas d'utilisation des entrées / sorties que nous considérons comme légitimes.

A.4.1 Acquisition des données et mise en forme du résultat

Le programme cité au paragraphe A.1 est tout à fait représentatif d'un schéma simple:

- acquisition de données,
- calcul,
- affichage du résultat.

Nous conseillons de limiter le dialogue avec l'utilisateur du programme à la première et dernière étape de ce schéma, le calcul étant purement fonctionnel. Nous donnons une première version du programme qui, quoique non satisfaisante, donne une idée du principe.

```
; Afficher     naturel x (naturel -> naturel)-> vide
;              n, Calcul                      -> indefini
;                              execute l'appel Calcul (n)
;                              affiche le resultat a l'ecran
(define Afficher (lambda (n Calcul)
      (begin
            (display "Pour ")
            (write n)
            (display "le résultat est ")
            (write (Calcul n))
            (newline))))
; Acquisition :      vide    ->      indifferent
;                            ->      acquiert, au clavier, une valeur et la delivre en
;                                    résultat
(define Acquisition (lambda ()
      (begin
            (display "Entrer un naturel: ")
            (read))))
; Prog :       vide    ->      vide
;      affiche à l'écran un message demandant de frapper un naturel au clavier,
;      acquiert au clavier la valeur d'un naturel,
;      calcule sa factorielle,
;      affiche à l'écran le résultat.
(define Prog (lambda ()
      (Afficher (Acquisition) Fact)))
```

Si on réalise le contrôle statique de type pour la fonction Prog, on constate que l'appel de Fact peut être mal typé. Rien, en effet, n'oblige l'utilisateur à entrer un naturel au clavier en réponse à l'invite de la fonction Acquisition. Pour pallier cet inconvénient, il suffit de modifier la fonction Acquisition.

```
; Acquisition :vide          ->      naturel
;                            ->      acquiert, au clavier un naturel
;                                    et le delivre en resultat
(define Acquisition (lambda ()
      (begin
            (display "Entrer un naturel: ")
            (Valider (read)))))
; Valider :   indifferent    ->      naturel
;             n              ->      n si n est un naturel
;                                    sinon, acquiert un naturel au clavier et le delivre
;                                    en resultat
(define Valider (lambda (n)
      (cond
            ((and (integer? n) (>= n 0))        n)
            (t                                  (begin
                                                      (display "incorrect")
                                                      (newline)
                                                      (Acquisition))))))
```

On remarque une récursivité croisée (Acquisition/Valider): rien n'assure que cette récursivité se termine (si l'utilisateur se révèle infiniment têtu, elle ne se termine d'ailleurs pas).

A.4.2 Mise au point

On peut utiliser, dans la phase de mise au point d'une fonction, des sorties qui permettent d'obtenir des éléments sur la nature de l'erreur. Cette facilité ne dispense pas de réfléchir au préalable.

```
; DeuxPn :   naturel     ->      naturel
;             n          ->      2^n
(define DeuxPn (lambda (n)
      (cond
            ((= n 0)        1)
            (t              (* 4 (DeuxPn (- n 2)))))))
```

Dans la tête du programmeur, cette fonction est sensée calculer 2^n. D'ailleurs, (DeuxPn 6) délivre 64. En revanche, (DeuxPn 5) boucle. Le programmeur sérieux aurait étudié la terminaison de sa récursivité (ce qui est une attitude raisonnable) et ne se serait pas mis dans ce mauvais pas. Ceci étant, nous ne sommes pas toujours sérieux… L'introduction d'une trace permet de saisir rapidement ce qui ne va pas.

```
; DeuxPn :   naturel     ->      naturel
;             n          ->      2^n
(define DeuxPn (lambda (n)
(begin
      (write n) (newline) (read)
      (cond
            ((= n 0)        1)
            (t              (* 4 (DeuxPn (- n 2)))))))))
```

A.4.3 Programmes dont l'unique objet est le dialogue

Certains programmes n'ont de sens que s'ils font des effets de bord! Pour ces derniers, il est difficile d'éviter des entrées/sorties assez profondément mélangées au code.

Nous en donnons un exemple représentatif qui est un programme de jeu.

A.4.3.1 Jeu de Nim

Le jeu de Nim est un jeu à deux joueurs 1 et 2. Au départ, ils sont devant un tas de n allumettes. Chaque joueur, à son tour, peut prendre 1, 2 ou 3 allumettes. Le perdant est celui qui prend la dernière allumette. On suppose que chacun des joueurs cherche à gagner.

A.4.3.2 Spécification du problème

On veut concevoir et coder un programme qui demande alternativement à chacun des joueurs ce qu'il joue, qui gère le tas d'allumettes et qui imprime le nom du perdant. Il est important de noter que le programme ne joue pas: il gère les divers éléments du jeu et empêche les joueurs de tricher.

On peut donner une spécification du problème et on remarque l'importance, dans cette spécification des effets de bord.

Nim: naturel x {1, 2}→ {1, 2}
n, i → le joueur vainqueur de la partie qui commence avec n allumettes, le joueur i ayant le trait. **Le programme doit gérer un dialogue avec les joueurs, pour demander ce qu'ils jouent, afficher le nombre d'allumettes restantes…**

A.4.3.3 Programmation

Il est clair que le programme Nim doit répéter deux actions: faire jouer un joueur (un coup) et changer le joueur qui a le trait. D'où l'idée d'introduire deux fonctions:

Coup: {1, 2} x naturel → {1, 2, 3}
i, n → nombre d'allumettes prises par le joueur i sachant que le tas comporte n allumettes, **La fonction coup affiche le nombre d'allumettes, demande au joueur de jouer et insiste jusqu'à ce que le coup joué soit légal.**

AutreJoueur: {1, 2} → {1, 2}
i → le numéro du joueur suivant.

On peut donner l'algorithme de Nim:

$$Nim(n, i) = \begin{cases} i \ \textbf{si} \ n = 0 \\ Nim(n - Coup((i, n), AutreJoueur(i))) \ \textbf{sinon} \end{cases}$$

D'où la programmation:

```
; Nim:     naturel x {1, 2}     ->     {1, 2}
;          n, i                 ->     le joueur vainqueur de la partie qui
;                  commence avec n allumettes, le joueur i ayant le trait.
```

```
(define Nim (lambda (n i)
      (cond
            ((= n 0)       i)
            (t             (Nim (- n (Coup i n)) (AutreJoueur i))))))
; Coup:      {1, 2} x naturel      ->      {1, 2, 3}
;            i, n                  ->      nombre d'allumettes prises par le joueur i
;                  sachant que le tas comporte n allumettes.
;                  La fonction coup affiche le nombre d'allumettes,
;                  demande au joueur de jouer et insiste jusqu'à ce que le coup
;                  joué soit légal.
(define Coup (lambda (i n)
      (begin
            (display "Joueur ") (write i)
            (display "(") (write n) (display "):") (newline)
            (Valider (read) i n))))

; Valider :    natuerl x {1, 2} x naturel          ->      {1, 2, 3}
;              c, i, n                             ->      valide la proposition de
;                        coup c faite par le joueur i devant un tas de n allumettes
;                        (rend c si c est correct, insiste auprès du joueur i jqa
;                        obtention d'une reponse correcte)
(define Valider (lambda (c i n)
      (cond
            ((and (integer? c) (<= c n) (<= c 3) (>= c 1))      c)
            (t                                                  (Coup i n)))))

; AutreJoueur:      {1, 2}          ->      {1, 2}
;                   i               ->      le numéro du joueur suivant.
(define AutreJoueur (lambda (i)
      (+ (remainder i 2) 1)))
```

Table des matières

Index

UNIVERSITE DE RENNES 1

UNIVERSITÉ DE RENNES 1 - Campus de Beaulieu - 35042 RENNES CEDEX - France

Tél : 02 99 84 71 00 - Télécopieur : 02 99 84 71 71 - mél : information@ifsic.univ-rennes1.fr

http://www.ifsic.univ-rennes1.fr