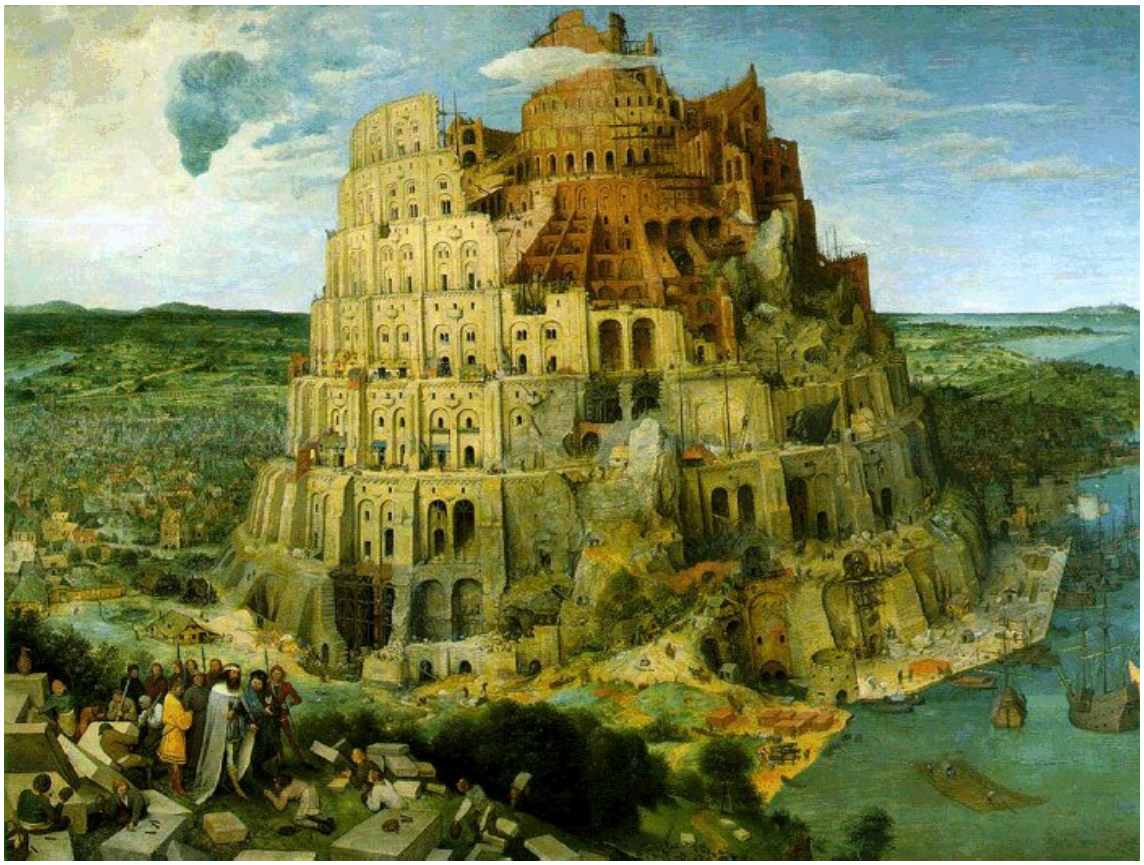
Compilateurs
et interprètes



Jerzy Karczmarczuk

Departement d'Informatique, Université de Caen

Caen, Janvier 2002

Table des matières

Chapitre 1

Introduction

1.1 Objectif du cours : Pourquoi apprendre la compilation?

La plupart des logiciels de haut niveau assure quelques possibilités de dialogue avec l'utilisateur et doit être capable de comprendre quelques instructions formulées dans un langage formalisé. Nous avons des tableurs ou des traitements de texte équipés avec des macro-processeurs puissants et capables de comprendre les formules mathématiques assez complexes ; systèmes graphiques ou gestionnaires de bases de données qui communiquent avec l'utilisateur dans des véritables langages de programmation de haut niveau (SQL), etc. Des éditeurs de texte évolués comme Emacs ou Word sont étendus par des interprètes des langages universels : Lisp ou Basic. Il y a des implantations des langages de programmation, comme par exemple Scheme conçues **spécialement** pour étendre les fonctionnalités des paquetages plus spécifiques. Scheme-Elk est utilisé dans un paquetage d'animation (AL) et dans un modeleur 3D : Sced. Une autre variante de Scheme (Guile) était utilisée dans le logiciel de traitement d'images GIMP comme son langage de scriptage. Les applications qui sont programmables en Scheme, doivent incorporer le compilateur et la machine virtuelle (interprète) appropriée. Les systèmes de mise en page (comme LaTeX) sont également des compilateurs – ils transforment le texte source, symbolique, en instructions de rendu graphique des textes et des images, «exécutées» ensuite par les pilotes des imprimantes, les pilotes PostScript résidant dans l'ordinateur, ou par des pilotes de sortie vidéo. Les *plug-in* des navigateurs Web comme Netscape sont des interprètes des langages spécifiques, comme Tcl/Tk, et le noyau de Netscape contient la machine virtuelle de Java et l'interprète du JavaScript, etc.

D'autre part il est évident que les systèmes d'interfaçage graphiques comme X-Window System, Microsoft Windows ou le système MacIntosh ont besoin de «scripts» (programmes exécutés par l'interface utilisateur ou l'interprète des commandes – le «shell») capables d'automatiser les tâches répétitives et de prendre quelques décisions dépendantes du contexte de l'intéraction. Comme il a été souligné ci-dessus, tous les éditeurs sérieux, comme Emacs sont programmables. Il ne s'agit pas d'ajouter simplement des macros au texte, mais de permettre à l'utilisateur de lancer l'exécution d'un autre programme, de récupérer le courrier, d'établir une communication vocale, de trier une petite base de données, etc. On a besoin des scripts pour intégrer plusieurs applications, par exemple une calculatrice symbolique avec un logiciel de rendu graphique comme Gnuplot, ou pour écrire les programmes CGI (exécutés sur le site serveur par une commande incluse dans une page HTML et lancée par un client distant). Des langages comme Perl, Tcl/Tk ou Python servent principalement à cela. À présent nous avons de nouveaux langages à objets comme Ruby, et aussi PHP. Donc, les interprètes/compilateurs sont vraiment omniprésents...

La programmation devient de plus en plus intégrée et visuelle. On a besoin de langages nouveaux pour décrire les scènes 3D et permettre l'intéraction avec le modèle (VRML, X3D, etc.). Même la description des *texture* constitue un langage de programmation (langage des «shaders»). Le monde XML évolue vite : nous avons déjà des langages de description de structures chimiques, ou de la phonologie et morphologie de la parole humaine.

On a souvent besoin de pouvoir décrire un *problème stratégique* dans un langage formalisé afin de préciser les modalités de sa solution automatique. On développe donc des *langages logiques*, des langages où il est plus facile de coder l'heuristique, l'apprentissage, l'accès aux bases de données déductives, la coopération entre experts/agents distribués, etc. Les langages «classiques» sont ici beaucoup moins commodes.

Enfin, on a besoin de langages (et de compilateurs spécialisés) pour le parallélisme, simulation, création musicale, communication multi-médiatique, organisation des hyper-textes, etc. Ajoutons à cela les langages de description de styles (CSS2 et ultérieurs), un langage de modélisation structurelle (UML), etc. Nous avons même des langages et les descriptions syntaxiques des systèmes biologiques évolutifs comme les systèmes de Lindenmayer, utilisés pour la modélisation des plantes. Il existe des langages style *dataflow* qui ne sont pas basés sur des phrases linéaires, mais où les programmes ont forme de graphes, et ces langages sont loin de l'abstraction académique : Khoros, Simulink, UML, ProGraph, Data Explorer, etc. sont des langages pratiques, voire même industriels.

Donc, notre programme dépasse le sujet contenu dans le titre : «*Compilation*». Nous allons parler de langages de programmation en général, de leur sémantique et de leurs styles, de techniques d'implantation des *machines virtuelles* et – si le temps nos permet – de l'interfaçage. Nous allons traiter des modèles concrets !

***Credo* religieux no. 1 :** On ne peut vraiment apprendre la compilation des langages de programmation, que si on en connaît quelques uns.

Les besoins des utilisateurs des ordinateurs évoluent, et les langages de programmation aussi. À présent ils sont plutôt orientés vers une bonne conceptualisation, lisibilité, sécurité, et surtout sur la puissance sémantique qui détermine notre liberté d'expression, que vers une utilisation extrêmement intense des ressources matérielles, comme des registres, transferts des octets entre les zones de mémoire, etc. Les assembleurs restent utiles à ceux qui envisagent le codage du noyau dur d'un système d'exploitation embarqué, où l'économie de mémoire et la vitesse sont essentielles, voire critiques.

Les techniques de génération du code présentés ici viseront plutôt des **machines virtuelles** – interprètes, que directement le silicium, car ainsi il sera plus facile d'aborder la sémantique et la pragmatique qui permettent d'évaluer et de comparer les langages. Le code de bas niveau **est important**, car, finalement, les ordinateurs marchent grâce aux instructions exécutées directement par le matériel, mais nous n'avons pas le temps de traiter tout, et surtout le domaine qui n'appartient plus à l'informatique au sens large, mais à l'ingénierie des architectures d'ordinateurs. Un étudiant en Informatique devra probablement un jour construire un petit compilateur et/ou interprète, car telles sont les tendances d'aujourd'hui. Par contre, les chances qu'il construise un compilateur dont le code-cible est l'assembleur de bas niveau, sont très faibles.

Ces notes contiennent de nombreux exemples du code écrit en Haskell – un langage fonctionnel pur. **La connaissance de ce langage est absolument incontournable pour pouvoir suivre ce cours.** Le polycopié contient donc une introduction à Haskell, mais il ne remplacera pas la documentation ! D'autres langages dont nous auront ***vraiment*** besoin :

- Scheme (ou Lisp), car ce langage constitue une base «classique» de programmation fonctionnelle, et grâce à la simplicité sémantique de son modèle : le calcul lambda, il sert souvent à implanter beaucoup d'autres paradigmes, notamment la programmation à objets, logique, etc., au moins à des buts pédagogiques.
- PostScript (ou Forth). Ces langages sont adaptés à des machines virtuelles à pile, relativement simples, programmables en code postfixe. Ceci sera notre machine virtuelle – cible privilégiée.

Un autre avertissement semble utile : les exercices inclus dans ces notes doivent être pris au sérieux. Ils font partie intégrale du cours, et quelques techniques de programmation ne seront expliqués qu'à travers eux. Quelques exercices ne seront pas accompagnés de réponses, et ceci peut suggérer que nous avons peut-être envie de les utiliser comme sujet d'examen…

Cette version de notes n'inclut plus la bibliographie ni la «Webographie» consacrée à la compilation, langages, etc. Notre polling effectué pendant 3 ans a démontré que les étudiants ne s'en servent pas *du tout*. Elle pourrait être utile, bien sûr, par exemple on peut trouver sur l'Internet presque tous les ingrédients du devoir (ou même le devoir complet), mais laissons cette recherche aux lecteurs. Si quelqu'un veut des références bibliographiques, prière de s'adresser persennellement à l'auteur.

1.2 Exercices

Q1. Quel est l'avantage de connaître plus de cinq (ou dix?) langages de programmation? Quel est le prix à payer? (Éviter la réponse triviale – on devient très savant, mais il faut du temps pour apprendre tout…)

R1. Pas de réponse unique. Trouvez vos réponses individuelles. Voici les nôtres :

- *Avantages.* Si un problème calculatoire, algorithmique, de représentation, etc. **très** complexe se pose, il est plus facile de trouver un style, un langage convenable, qui économise le plus le temps *humain*, même si l'implantation du langage est très inefficace. Ici, ce facteur d'économie peut atteindre plusieurs centaines : des heures plutôt que des semaines, et ceci peut récompenser aussi la «perte» du temps d'apprentissage. Ensuite, une bonne connaissance de plusieurs styles permet de choisir et d'inventer quelque chose de propice si le circonstances exigent la construction du compilateur d'un petit langage intégré à une application très spéciale.

  Il y a aussi le plaisir créatif indépendant de la vision strictement instrumentale des langages.

- *Handicaps.* On s'égare ! C'est un peu comme une tentative de pratiquer tous les sports à la fois. On risque de ne rien faire vraiment bien… Ensuite, on risque de confondre les langages et de coder des «monstres de Frankenstein» inutilisables. Cependant ce risque n'est pas très grand, si l'apprentissage est accompagnié par une raisonnable pratique.

  Aussi, on tombe facilement dans le piège du perfectionnisme et on commence à élaborer MNLP – *Mon Nouveau Langage de Programmation*. Ceci n'est pas mauvais *per se*, et peut aboutir à une thèse et à autres succès personnels, mais ce langage est *a priori* condamné à être oublié, sauf les cas extrêmement rares, car la concurrence est trop forte.

Q2. Mentionnez au moins une douzaine de conventions différentes utilisées pour dénoter des *commentaires* dans des divers langages de programmation.

R2. *Cherche et tu trouveras*. Mais n'oubliez pas Fortran ni Cobol.

Q3. Qui est l'auteur et le titre du tableau sur la première page de ces notes?

R3. Oui, vous avez gagné.

Q4. Fermez les yeux et citez au moins une cinquantaine de langages de programmation dont vous avez entendu parler.

R4. Toute l'idée ici est de ne pas tricher, et de pouvoir dire au moins 20 mots *a propos* de chacun de ces langages. Voici quelques suggestions : Basic, Pascal, C, C++, Simula, FORTRAN, Cobol, Lisp, (et Scheme), Smalltalk, Modula, Perl, Python, Sather, Ruby, Icon, Snobol4, TCL, SML, Haskell, CAML, Prolog, Mercury, Clean, Miranda, Eiffel, Algol, CLP, Sisal, Matlab, FORTH et PostScript, MetaPost, TEX, PL/I, Java, JavaScript, Ada, APL, Hope, Id, Self, Occam, SQL, PHP, Erlang, Awk, Life, plus une autre trentaine d'assembleurs, sans compter les langages graphiques comme UML, WiT, Simulink, ProGraph or Khoros, et sans compter les langages de description/marquage (plutôt que de programmation): VRML, SVG, X3D, HTML, etc.) Et encore des langages spécialisés, par exemple les langages de Calcul Formel : Maple, MuPAD, Magma, Axiom, GAP, etc.

Q5. Et maintenant, poses vous-même une question !

R5. OK, voici une proposition : ***Je n'ai pas beaucoup de temps, mais j'aimerais apprendre quelques langages de programmation élégants et puissants qui ne sont pas enseignés ici.*** Avez-vous une suggestion?

Pourquoi pas?

- O'Caml. C'est un langage fonctionnel moderne, typé, avec une couche orientée-objet très prononcé. Le compilateur est très efficace. Il a été utilisé à des buts assez compliqués, comme la construction d'un lanceur de rayons, où il a gagné (en fait, l'équipe d'Inria a gagné) avec d'autres langages au niveau de vitesse et facilité de programmation. C'est un produit français, ce qui facilitera votre accès à la documentation.
- Mercury. C'est un langage logique (ou, un peu : logico-fonctionnel), mais typé et *beaucoup* plus rapide que Prolog. Recommandé pour des personnes qui s'intéressent à l'intelligence artificielle.
- Python. Orienté-objet, petit et transparent, facile, et vraiment universel comme un langage de *scripting*. Il peut remplacer Perl partout.

Q6. Voici une anecdote historique. Quel est son rapport avec ce cours?

Quand la dynastie des Jagellons, rois de Pologne, s'est éteinte au 17 siècle, les Polonais ont eu une idée formidable et moderne : organiser une monarchie démocratique, avec les rois élus. Et ils en ont élu une douzaine, dont le premier, Alexandre d'Anjou, au bout de quelques mois s'est enfuit avec la caisse, pour rentrer en France et devenir Henry III. Mais les autres n'étaient pas toujours meilleurs. Après une fructueuse élection l'ambassadeur de Venise écrivit à son souverain, le Doge : «*Les Polonais ont élu un nouveau roi. Un personnage très intelligent et savant. Il parle couramment 7 langues ! Mais, malheureusement, il n'a rien à dire…* ».

R6. Aucun rapport. Tous les étudiants ont toujours beaucoup de choses à dire, même si cela ne se voit pas.

Chapitre 2

Classification générale des langages : survol de la tour Babel

2.1 Catégories classiques

Cette section contient une revue de plusieurs langages de programmation, leur comparaison et évaluation (très superficielle).

Il y a des visions très différentes du *processus calculatoire*. Le comportement de l'ordinateur qui pilote une navette spatiale n'est pas le même que celui de la machine qui aide un physicien-théoricien à trouver les tores de Kolmogorov-Arnold-Moser, bien que dans les deux cas on peut chercher le régime dans lequel la solution de quelques équations différentielles soit stable. Les buts pratiques sont différents. Il faut donner la préférence à : l'efficacité? la sécurité? l'intéractivité? la facilité du codage pour des non-spécialistes? Les mêmes questions se posent dans le domaine des langages et leur compilateurs. Un langage «ami de tous» n'existe pas, les différences sémantiques sont importantes et influencent le style et la syntaxe. Il n'y a pas de solutions-miracle. La création des nouveaux langages ne se terminera jamais.

Par convention on divise le monde des langages de programmation en quelques catégories non-exclusives. On parle par exemple des **langages impératifs, logiques ou fonctionnels**, mais la couche fonctionnelle existe dans presque tout langage, sauf les assembleurs les plus primitifs et quelques langages descriptifs (statiques), car cette couche fonctionnelle n'est rien d'autre que la capacité d'*évaluer les expressions en appliquant les opérateurs*. Un *vrai* langage de programmation possède plusieurs couches. Donc, le «catalogue» ci-dessous n'est pas du tout une classification des langages ! Au lieu de parler le «langages fonctionnels» nous aurions dû mentionner des *couches* sémantiques : impérative, fonctionnelle, logique, etc. Le langage est considéré fonctionnel si sa couche fonctionnelle prédomine, si elle est bien exposée syntaxiquement, et conditionne le style global des programmes écrits dans ce langage, ainsi que le modèle d'exécution du programme (le modèle de la machine virtuelle). Mais – répétons – tout langage impératif est d'habitude un peu fonctionnel, et la programmation par objets peut se faire en style procédural (impératif, comme en Smalltalk), fonctionnel (quelques extensions du Scheme , ou Haskell), ou même logique (extensions objet du Prolog).

Si on veut réellement en deux mots préciser les différences fondamentales entre ces catégories de langages on pourra formuler ceci de manière comme ci-dessous. Cependant, la couche fonctionnelle de la programmation sera couverte de manière *beaucoup* plus complète plus tard, car tout notre cours est basé sur des techniques fonctionnelles.

2.2 Programmation impérative

Un langage impératif est un langage de *commandes* (ou instructions, ou directives, ou actions, etc.) On modifie les variables (ou registres), on construit des itérateurs (boucles) ou autres structures de contrôle qui se réduisent aux branchements. Les valeurs des variables contituent l'*état* du système, et les instructions modifient cet état. Un registre particulier, le «compteur du programme» adresse l'instruction qui sera exécutée, et si le programme modifie explicitement la valeur de ce registre, ceci constitue le *branchement*. Sans branchements, le compteur du programme est auto-incrémenté.

C'est la catégorie «classique» des langages ; le code compilé est de bas niveau, adapté aux architectures des processeurs (architecture de von Neumann), et il doit être rapide. Exemples : C, Pascal, Ada, Fortran.

Dans le modèle de von Neumann le programme stocké dans la mémoire est une liste linéaire d'instructions. La machine exécute une boucle : après la modification du compteur du programme on continue. Il y a toujours une instruction à exécuter. **La machine ne s'arrête jamais**. Le branchement inconditionnel ou conditionnel : selon la valeur Booléenne d'un des registres, on effectue ou pas un *branchement*, le **goto**, une instruction dont l'argument est l'adresse d'une autre instruction est la structure de contrôle fondamentale.

Si l'adresse passée à **goto** précède l'adresse actuelle, on peut fermer une boucle classique. Mais un programmeur typique, intéressé par les résultats finaux ne doit pas traiter ce programme en C

```
while(x>2)
 {faire(x,g(x)); x=h(x);}
```

comme l'abréviation de

```
boucle:  z=x-2;
if(z<=0) goto bfin;
faire(x,g(x));
x=h(x);
bfin:      ...
```

car psychologiquement c'est totalement inutile. Il faut, bien sûr, comprendre la sémantique de la boucle, et non pas sa forme décortiquée de bas niveau.

Cependant les concepteurs et réalisateurs de compilateurs ne sont pas des programmeurs typiques. Ils doivent savoir traduire les constructions de haut niveau en concepts appartenant au modèle d'exécution. Ceci est vital, indépendamment des différences syntaxiques entre langages.

Alors, un petit récapitulatif. La machine virtuelle de plus bas niveau, un interprète impératif «plat» doit permettre

- L'adressage des zones mémoire contenant les instructions.
- Auto-incrémentation (exécution séquentielle des instructions) et modification dynamique du compteur des instructions, le branchement (goto).
- Au moins un mécanisme décisionnel (if→goto).
- L'adressage des emplacements des données (registres, variables).
- Récupération des données, leur transfert.
- Modification des données (primitives) par le processeur (p. ex. l'arithmétique).

Ceci n'épuise pas les notions appartenant au modèle impératif. Il faut ajouter au moins la possibilité de construire des procédures, c'est à dire d'automatiser le **goto** avec retour. Il faut donc pouvoir *stocker quelque part l'adresse d'une instruction*, de la traiter comme donnée.

En Fortran antédiluvien les procédures (subroutines) n'étaient pas récursives, et chaque procédure prévoyait *un* emplacement statique dans son segment de données pour y stocker l'adresse de retour de ce procédure au module appelant. Le code de l'instruction **`call f`** par **`g`** se compilait comme

- Récupérer l'adresse de la procédure appelée **g**.
- Récupérer l'emplacement du segment de données de **g** correspondant à l'adresse de retour, disons, le registre **RET**.
- Stocker dans ce registre la valeur du compteur-programme (l'instruction suivante à exécuter, appartenant à la procédure **f**).
- Exécuter **goto g**.

tandis que le retour de la procédure se réduisait à

- Récupérer la valeur stockée dans **RET**, et effectuer le **goto**.

Dans ce modèle toutes les données étaient statiques, et chaque procédure travaillait dans son «monde privé», ou, éventuellement dans une zone accessible globalement.

La possibilité d'opérer avec des procédures récursives implique la *protection de l'adresse de retour*. Le protocole standard, utilisé par pratiquement toutes les implantations des langages admettant la récursivité est basé sur la *pile des retours*. Au lieu de stocker l'adresse de retour dans une zone statique appartenant au module appelé, chaque appel réserve un segment du tableau géré par le système et structuré comme une pile. L'adresse de retour est stockée sur ce segment. Chaque retour détruit le dernier segment alloué.

Bien sur, les langages traditionnels, disposant des procédures parametrées prévoient également l'allocation d'un segment de données où on stocke les arguments et les données locales. Cette zone de mémoire est aussi structurée comme une pile. Conceptuellement ce segment est indépendant du flot de contrôle, même si la pile des retours et la pile des données souvent fonctionnent en synchronie. Nous verrons toutefois, qu'il est plus facile de construire des machines virtuelles si on garde l'indépendance des deux objets.

2.2.1 Co-procédures et quasi-parallélisme

Notons que les branchements et les boucles ajoutées aux appels procéduraux n'épuisent pas toutes les structures de contrôle disponibles dans quelques langages. Un mécanisme particulièrement intéressant pour la simulation de systèmes dynamiques est la *co-procédure* qui permet la réalisation de collaboration ***symétrique*** entre deux modules : comme dans un jeu avec deux partenaires qui jouent leur coups en alternance. Avec l'appel standard si module A appelle module B, il empile l'adresse de retour et continue l'exécution après le retour. Si B appelle de nouveau A, un nouveau empilement a lieu. Comment alors organiser un jeu binaire, où les deux modules représentent les joueurs : chacun joue à son tour, change l'état global du système (l'échiquier par exemple, où la trajectoire de la balle), et passe la main à l'adversaire. Comparez les deux dessins sur la Fig. (2.1).

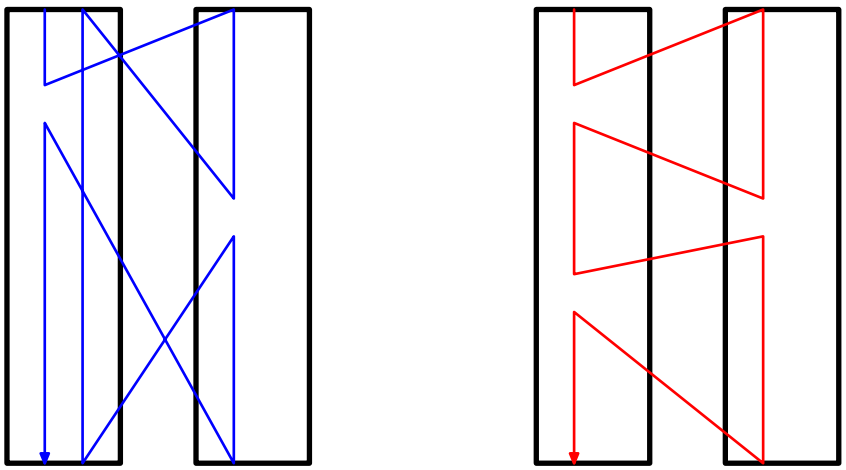


Fig. 2.1: Procédures et co-procédures

La co-procéduralisation consiste à respecter les règles suivantes :

- Si le module A n'a jamais été appelé, son lancement (appel) se déroule (pratiquement) de manière standard, comme s'il était une procédure normale.
- Si maintenant A veut passer le contrôle à B, il exécute le branchement, mais avant **stocke l'adresse de retour dans *sa propre* zone de données**, où dans une structure globale, spéciale, balisée et identifiable comme appartenant à A.
- Quand B veut «retourner» à A, il fait la même chose – branche à A après avoir stocké l'adresse de retour.
- Le re-lancement du module partenaire (l'exécution de l'instruction **resume**) est un branchement indirect, le code qui récupère le contrôle vérifie d'abord si une adresse de retour co-procédural n'a pas été stockée au préalable, et si c'est le cas, un branchement secondaire a lieu.

La technique de co-procéduralisation est très importante dans la simulation, et constitue une alternative aux flux (pipes) permettant d'établir une collaboration symétrique entre les modules du compilateur. Les co-procédures constituent aussi une variante de réalisation du parallélisme, et donc son implantation est importante pour la compilation des langages parallèles (et pour comprendre le fonctionnement des systèmes d'exploitation).

Ce qui sera intéressant pour nous est le fait que les co-procédures et même le branchement **goto** possèdent des modèles fonctionnelles (même si ces modèles ne sont pas toujours pratiquement efficaces). En fait, *les langages fonctionnels sont suffisamment puissants pour pouvoir modéliser les constructions impératives*. Mais ceci est loin d'être facile !

2.2.2 Exemples

La majorité des langages sur le marché est impérative : C, C++, Java, Ada, Oberon (une évolution de Modula), etc. Le modèle impératif se combine avec la programmation par objet de plusieurs façons différentes : C++, Eiffel et Smalltalk sont tous des langages impératifs à objets, mais assez différents. (Smalltalk est même *très* différent).

La popularité des langages impératifs est le résultat du fait que les assembleurs sont impératifs, et d'une illusion (justifiée historiquement) qu'un langage impératif est toujours plus efficace qu'un langage fonctionnel ou logique.

Ainsi, souvent les langages interprétés où l'efficacité brute du code est un facteur secondaire, comme les langages de programmation scientifique : Matlab, IDL, etc., ou les langages de calcul formel : Maple, Axiom, Magma, etc., sont des langages impératifs, dont la syntaxe ressemble à la famille Pascal, même si plusieurs parmi eux sont basés sur des machines virtuelles fonctionnelles, notamment sur l'interprète Lisp, ou pareil. La psychologie conservatrice a gagné sur la logique…

Les langages interprétés conçus pour écrire des scripts (et partiellement remplacer les langages de commandes du système d'exploitation) comme Perl sont aussi impératifs. L'auteur avoue ne pas comprendre pourquoi Perl est devenu si populaire. Sa structure syntaxique est laide, et son déboguage pénible. Mais sa gestion des chaînes de caractères et expressions régulières est très riche, et la coopération avec le système d'exploitation (gestion des fichiers et des processus) est mise au point. En tout cas, d'autres langages (par exemple Python) possèdent déjà presque toutes les fonctionnalités du Perl, et le langage PHP remplace actuellement les scripts CGI classiques.

Un de nos langages préférés : Icon qui a un goût fonctionnel très prononcé, et est un de très rares langages avec des structures de contrôle non-déterministes (ce qui l'approche aux langages logiques), a été bâti comme langage impératif qui ressemble à C, car son auteur, Ralph Griswold, un excellent pédagogue, croyait que ceci était mieux adapté à la psychologie humaine.

Une autre raison de cette popularité est la tradition académique, toujours présente ici ou là : on enseigne trop souvent la compilation comme un processus qui doit obligatoirement après toutes les optimisations générer un code linéaire impératif, style assembleur, exécuté par la machine «matérielle». Pas besoin de dire que dans de tels établissements les chances de créer un nouveau langage de programmation et de son compilateur sont plutôt minces…

***Credo* religieux no. 2 :** Ceux qui enseignent la compilation et passent 90% de leur temps à discuter la syntaxe des langages de la famille Pascal, et pour qui le seul code-cible est l'assembleur, iront tous en enfer. (Ou, peut-être, ils sont déjà là, sans le savoir…)

2.3 Programmation fonctionnelle

La machine qui exécute un programme fonctionnel «focalise son attention» sur le concept d'*expression* : objet qui engendre une *valeur*. Nous avons donc les constantes littérales : nombres, chaînes de caractères (considérées comme atomiques ou comme des listes de caractères), etc. Les fonctions seront elles aussi des valeurs, des ***objets fonctionnels***. Il existe également des variables qui donnent des *noms* aux valeurs, et leur usage implique l'existence de **l'environnement** permettant d'établir le rapport entre un nom et la valeur correspondante. Cet environnement replace partiellement la notion d'état impératif.

Dans un langage **purement** fonctionnel on n'a pas le droit de changer les valeurs des variables, une variable est synonymique avec sa valeur, un peu comme en mathématiques. La récursivité terminale (itérative) est le

seul moyen d'implanter les boucles, et lors de la nouvelle instance de cette «boucle», une *nouvelle* variable remplace l'ancienne[1]. On construit des fonctions dont les arguments (et les résultats) sont aussi des fonctions, ce qui permet la construction de structures de contrôle très compactes et beaucoup plus riches (continuations, monades, filtres) que celles dans des langages impératifs. Fonctions peuvent être composées, et – ce qui est essentiel pour tout programmeur – dans un langage fonctionnel on peut former ***fermetures*** (ang. *closures*) : objets fonctionels qui «attrappent» l'environnement dans lequel ils sont définis, pouvant ainsi stocker des données arbitraires. On verra plusieurs exemples de ces constructions.

On utilise fréquemment l'allocation dynamique de mémoire, et ces langages sont gourmands en mémoire. Exemples : Haskell, ML (variantes SML ou CAML), Hope, Erlang, et parties «pures» du Lisp ou Scheme. Cependant ce fait n'a rien à voir avec le fait que le langage soit fonctionnel. Java utilise l'allocation/déallocation dynamique de mémoire également.

Les nouveaux langages fonctionnels sont *statiquement typés*, comme C, Java ou Pascal, contrairement au Lisp, mais la discipline formelle sur laquelle repose les définitions du Haskell etc., permet *l'inférence automatique de types*! On n'a pas besoin de déclarations (sauf dans quelques cas ambigus, et pour la documentation).

Le fait que les objets fonctionnels soient des données comme les autres implique l'existence de fonctions anonymes. En Scheme ces deux définitions sont équivalentes :

```
(define (f x y) (sqrt (+ (* x x) (* y y))))

(define f
   (lambda (x y) (sqrt (+ (* x x) (* y y))))
)
```

On peut dire «la fonction **f**», mais **f** n'est qu'une variable dont la valeur est une fonction. Ceci est très différent de la situation en C où la procédure est attachée à son nom de manière irrévocable.

La nécessité d'opérer uniquement avec des données immuables et applications fonctionnelles n'empêche pas l'usage des variables locales, par exemple la construction

```
(let ((x (sin (/ pi 4)))
      (y (sqrt 2.0)))
  (* x (exp (+ x y)))
)
```

peut être reformulée comme

```
((lambda (x y) (* x (exp (+ x y))))
 (sin (/ pi 4))
 (sqrt 2.0)
)
```

La construction **letrec** (ou **let** en Haskell) est plus délicate, et sera discutée ultérieurement. Rappelons que **letrec** permet de définir des objets récursifs, et ceci implique que la variable définie *ainsi que sa définition* appartient au *même* environnement, ce qui empêche la translation en **lambda** comme ci-dessus.

En général, la suite de cette section recommandée au lecteur est l'annexe consacré au Haskell. Ici nous mentionnerons encore quelques généralités, et passons aux autres choses.

2.3.1 Programmation paresseuse

Scheme, comme la totalité des langages impératifs appartient à la catégorie des langages **stricts**, dont la définition informelle est la suivante : si la fonction f s'applique à ses arguments, p. ex., $f(x, y, z)$, l'ordre d'évaluation est le suivant : d'abord on évalue x, y, et z, et ensuite on applique f aux valeurs des arguments. Ces valeurs peuvent être des nombres, des références (pointeurs, etc.) des objets composites, etc., mais elles sont statiques. Si l'évaluation de, disons, y échoue à cause d'une erreur arithmétique ou autre, l'application de f n'aura jamais lieu. Tout appel récursif force l'évaluation de l'argument qui contient l'appel de la fonction récursive, donc on est *obligé* de stocker les valeurs intermédiaires et les adresses de retour sur une pile, sauf dans le cas de récursivité terminale.

[1] même si cette variable dénote un tableau de 1000000 éléments. La recopie intégrale d'une telle structure de données quand on change un seul élément serait extrêmement pénalisant. Il existe donc quelques astuces d'optimisation discutées plus tard

Mais Haskell (comme Clean, ou une version de Hope) est un langage **paresseux** (ou non-strict), où l'évaluation des arguments d'une fonction a lieu si, et seulement si, et *quand* la fonction a besoin de cet argument, quand il est effectivement utilisé. Avant cela l'expression-argument est compilée, transformée en un fragment de code appelé souvent le ***thunk***, et ce code est passé à la fonction appelante. Quand la fonction a besoin de l'argument, elle déclenche automatiquement l'exécution du thunk. D'habitude la valeur retournée par le thunk, le remplace, donc toute évaluation ultérieure n'a plus besoin d exécuter l'expression différée.

Si un langage est réellement fonctionnel, c'est-à-dire s'il permet la création des objets fonctionnels quelconques, la paresse peut être aisément implémentée. Par exemple en Scheme il existe une *macro-instruction* **`delay expr`**, qui transforme l'expresssion en thunk. La procédure **force** force l'évaluation du thunk. Regardez le programme suivant :

```
(define (integs n)
  (cons n (delay (integs (+ n 1)))))

(define ints (integs 0))

(define (tail l) (force (cdr l)))
(define (take m l)
  (if (= m 0) ()
      (cons (car l) (take (- m 1) (tail l)))))
```

La fonction **integs** est récursive ***sans clause terminale*** – elle représente une liste infinie de nombres à partir de **`n`**. Mais aucun débordément n'a lieu, la forme **delay** «protège» **`(integs (+ n 1))`** de déclencher une fuite en avant, qui doit forcément se terminer par le débordement de la pile. On construit ici un *objet différé*, un thunk qui rend sa valeur quand il est forcé. À ce moment-là ce thunk génère un nouveau chaînon dans la liste, et il cache derrière sa nouvelle instance, avec l'argument $n + 2$.

La réalisation du **delay** en Scheme est relativement simple : on transforme une forme quelconque **`expr`** en **`(lambda () expr)`**. Le thunk n'est rien d'autre qu'une fonction anonyme sans paramètres, dont l'évaluation produit le résultat souhaité. La vérité *complète* est, bien sûr, plus élaborée, le thunk est une structure auto-modifiable : la forme lambda après l'évaluation «écrase soi-même», et remplace son corps par le résultat.

Il *doit* être évident, que la présence de fonctions différées, qui peuvent être lancées dans un contexte quelconque, à n'importe quel moment, demande que les fonctions soit relativement pures, sans effets de bord, sinon le déboguage peur être impossible. Pour cette raison la programmation paresseuse est restreinte au monde de la programmation fonctionnelle.

En Haskell *toute* expression est différée, on n'a pas besoin de la forme **delay**, ni du forcing. Dans quelles circonstances ce protocole peut être *utilisé* dans la compilation?

1. D'abord, les fonctions paresseuses peuvent représenter les *structures de contrôle*. Une forme genre **`(if condition alors_instr sinon_instr)`** vérifie la condition toujours quand **if** est exécuté, mais ensuite on évalue une et une seule de deux expressions qui suivent. Leur représentation par thunks est assez naturelle.

2. Les listes ou autres structures paresseuses dans des programmes fonctionnels *remplacent les boucles, ou autres processus itératifs* : au lieu de faire quelque chose dans une boucle, on génère des instances nouvelles de manière paresseuse. Ceci est souvent plus facile à déboguer qu'un programme dynamique.

2.3.2 Continuations

Le concept de continuations est très important, et constitue un pont entre la programmation fonctionnelle et impérative. La continuation est le «futur» d'un calcul, c'est la réponse à la question «qu'est-ce qu'on fait à présent», après avoir évalué une expression. Les continuations peuvent modéliser les branchements, et aident à gérer le non-déterminisme.

Nous verrons des réalisations concrètes de continuations en Haskell, mentionnons ici deux contextes dans lesquels on voit les continuations dans la pratique de la programmation.

1. Le style CPS (*Continuation Passing Style*). Si dans un style applicatif classique l'expression $f(x)$ signifie : appliquer la fonction f à la valeur de x, l'expression CPS équivalente aura la forme $f(x, c)$, où c est *une autre fonction,* ***la*** *continuation de* f, qui récupère la valeur du résultat, et en fait quelque chose. Une continuation peut à la fin appeler une autre, ensuite une autre, etc., jusqu'à la fin du programme, quand on récupère la réponse finale.

   L'enchaînement des continuations dans un programme fonctionnel remplace l'enchaînement (sérialisation, séquentialisation) des instructions dans un programme impératif ! Les continuations réellement constituent le modèle fonctionnel du branchement **goto**.

   Une fonction peut constituer une partie de sa propre continuation, et ceci correspond aux appels récursifs. En général, si le style CPS est poussé à l'extrême, *toute* récursion devient terminale ! Ceci ne signifie pas que l'on peut *optimiser* tout appel récursif, en évitant l'usage de la pile, la pile (ou autre structure équivalente, comme une liste stockée sur le tas) servira toujours pour la sauvegarde des structures intermédiaires en cas de besoin, mais son usage devient plus explicite.

   Les continuations et le CPS sont des outils de construction de compilateurs assez populaires. Andrew Appel a écrit un livre entier consacré à la construction des compilateurs à l'aide des continuations.

2. Continuations «de 1-ère classe», ou **call-with-current-continuation** (ou **call/cc**) en Scheme. Ce concept est une structure de contrôle très puissante, presque la plus puissante qui existe dans le monde de programmation. Elle est malheureusement très rarement enseigné ***ici***. Le **call/cc** peut «attraper» la continuation *courante*, le futur du calcul qui se déroule au moment de son appel, et de l'«emballer» dans un objet fonctionnel, une donnée qui peut être réactivée plus tard.

   Ainsi, on peut stocker la continuation courante *avant* de déclencher un calcul très long et profond, et quand à l'intérieur de ce calcul on découvre qu'il n'a plus de sens, on relance cette continuation, ce qui fait abandonner tout et monter jusqu'à la surface du programme, à l'endroit qui a appelé **call/cc**.

   On peut aussi appeler **call/cc** à l'intérieur d'un calcul, exporter le résultat (la continuation) et faire d'autre chose. Plus tard on réactive ce calcul en relançant la continuation sauvegardée.

   Ce mécanisme est une version de haut niveau, structurée, et sémantiquement propre d'un mécanisme d'échappement en C connu sous le nom de **setjmp / longjmp**.

Les exercices qui suivent ce chapitre demandent la réalisation de quelques problèmes en Haskell, le lecteur doit donc – si tels sont ses besoins – lire l'annexe.

2.4 Programmation logique

Les langages logiques, comme Prolog ou Mercury se rapprochent un peu des langages fonctionnels, mais ici le concept fondamental est une *relation* entre deux objets, p. ex. entre un symbole et une valeur numérique ou structurale : ceci peut être considéré comme une affectation, mais logiquement c'est une équivalence.

Ces langages sont souvent non-déterministes, et offrent la possibilité de lancer la recherche d'une solution alternative d'un problème stratégique. On a pensé (la 5-ème Génération au Japon) que ces langages deviendront très populaires, car leur force d'expression est très grande. Malheureusement ce rêve a échoué, partiellement à cause de l'inefficacité des implantations. La situation évolue toutefois, et quelques implantations de Prolog, ou Mercury, qui sont des variantes des langages logiques *typés* gagnent du terrain. C'est la catégorie propice à la construction et gestion des bases de données, ou à la programmation par contraintes.

De plus en plus souvent on parle des langages hybrides (surtout logico/fonctionnels) avec la couche logique très importante : Life, Oz, Leda ou Opal. Le Prolog reste néanmoins le langage logique numéro 1. (En plus, il existe en plusieurs dialectes.)

Voici les traits caractéristiques de cette catégorie. Nous n'allons pas parler de structures syntaxiques, seulement souligner ce qui peut être intéressant de point de vue de la compilation.

- Le langage est statique, comme les langages fonctionnels. On n'a pas le droit de modifier une variable, elle se confond sémantiquement avec sa valeur. Toutefois en Prolog il existe le concept de *variable logique* non-instanciée : une variable qui n'a pas de valeur, mais qui occupe de la place, et qui peut être équivalencée à une autre variable.

- Les boucles classiques sont réalisées par des appels récursifs terminaux, mais il existe une autre catégorie de boucles *non-déterministes* qui utilise le *backtracking* : On récupère une solution, on la rejette (après l'avoir – éventuellement – sauvegardé de manière persistante), et on demande au système une ou plusieurs solutions alternatives (dont le nombre peut être infini).
- Le mécanisme décisionnel fondamental est l'*unification* (`=`) des termes simples et composites qui automatise la construction et l'analyse (décomposition) des données, par exemple l'unification combinée `Z=..[F,X,a]`, `Z=g(p(A,Y),Y)` instancie automatiquement les variables suivantes :

  ```
  F=g
  Y=a
  X=p(A,a).
  ```

  ce qui rend la programmation en Prolog très compacte. L'unification est une sorte d'équivalence. `p(A)=p(B)` ssi `A=B`. L'unification `f(x)=g(A)` **échoue**. La notion d'échec en Prolog est fondamental pour la construction de structures de contrôle. Quand l'échec se produit, le programme «retourne sur ses pas» au point, où il avait une décision non-déterministe à prendre. Il marque les chemins parcourus, et en choisit un autre. Cette technique est connue au moins depuis les temps de Thésée, Minotaure, et Ariane.
- La dépendance fonctionnelle entre données ($y = f(x)$) s'est généralisée en *relation* plus universelle, et la représentation linguistique d'une relation est un *prédicat* qui a la forme d'un terme, par exemple $r(x, y, [2, x])$. Un prédicat qui réalise une relation peut représenter une fonction, par exemple `plus(A,B,C)` qui modélise l'énoncé : `C=A+B`, ou peut dénoter une contrainte ou un attribut (propriété) : `negative(X)`, etc. Dans un Prolog interprété les prédicats sont souvent représentés par des termes (spécialement optimisés).

En fait, on a besoin de techniques un peu spéciales pour compiler le non-déterminisme et l'unification complète de manière *efficace*. Trop souvent les cours universitaires classiques de compilation ignorent ce domaine. Nous n'avons pas le temps de le traiter non plus, il faut cependant remarquer qu'un progrès formidable a été fait – grâce aux *continuations* et à la construction d'une machine virtuelle très simple, mais puissante : WAM – la machine abstraite de Warren, les implantations de Prolog jadis rares, sont devenues des exercices standard pour les étudiants. En particulier, la construction d'une machine non-déterministe à l'aide d'un langage fonctionnel paresseux, est un vrai plaisir intellectuel, et il existe au moins trois implantations de Prolog réalisés en Haskell.

2.4.1 Exemples des programme en Prolog

Le seul but de cette section est de permettre voir comment réaliser quelques structures sémantiques nondéterministes. Plus tard nous allons les coder en haskell. Construisons un programme en Prolog qui génère *toutes* les permutations des éléments d'une liste, par exemple `[a,b,c]` donne les $3! = 6$ permutations abc, acb, bac, bca, cab et cba. L'algorithme sera **présenté de manière non-déterministe** et cet exercice doit obligatoirement être bien assimilé par le lecteur. La compréhension du non-déterminisme est fondamentale pour la construction des analyseurs syntaxiques, car le *parsing* très souvent est non-déterministe.

Le raisonnement non-déterministe (logique) signifie simplement qu'on pose des questions, qui admettent plusieurs réponses. Le programme doit (éventuellement) trouver toutes. La *programmation* (ou le **style**) non-déterministe est basée sur le principe suivant : on analyse *une* solution, *quelconque*, arbitraire, inconnue, et à partir de cette solution (partielle) on génère une, ou plusieurs autres. Une vision un peu Science-fiction peut être utile : imaginez que l'ordinateur posé devant un problème non-déterministe qui possède deux solutions, déclenche le clonage du monde entier en deux exemplaires. Dans l'un de deux la machine fournit la réponse numéro 1, dans l'autre – la numéro deux. Si ces réponses provoquent autres clonages, on obtient une arborescence de taille arbitraire. Le processus est supervisé par un «démiurge» extérieur, capable de récupérer *toutes* les solutions et de les projeter dans le «monde réel». Répétons : le non-déterminisme ici est un *style de programmation*. La réalisation effective de ce «clonage» utilise le *backtracking*.

Construisons d'abord un prédicat *d'insertion non-déterministe* (rien à voir avec l'insertion discriminée, utilisée dans le tri par insertion). Cette insertion met un nouveau élément dans une liste *n'importe où*, par exemple à la tête, ou à l'intérieur. Appelons le prédicat correspondant `ins(X,Lst,Res)`.

```
nondetins(X,Lst,[X|Lst]).
nondetins(X,[Y|Q],[Y|R]) :-nondetins(X,Q,R).
```

La forme **[A|B]** est le même que **(A:B)** en Haskell. La première ligne (la première clause) du prédicat **ins** signifie que le résultat *peut* être obtenu par la mise de **X** à la tête de la liste. Si on rejette cette solution, c'est-à-dire si on en cherche une autre, l'argument **X** doit se trouver à l'intérieur de la nouvelle liste; alors la tête originale doit rester sur place. On la sépare, l'appel récursif insère **X** quelque part dans la queue (on obtient une **solution quelconque**), et il suffit de réinsérer la tête **Y**.

Le prédicat qui génère les permutations est encore plus simple. La permutation de la liste vide est toujours vide. Sinon, séparons la tête, trouvons *une permutation quelconque* de la queue, et réinsérons la tête, mais *n'importe où* dans le résultat. Voici le code :

```
permut([],[]).
permut([X|Q],R) :- permut(Q,R1),nondetins(X,R1,R).
```

Dans la pratique la machine non-déterministe opère de façon suivante : un résultat est généré, et éventuellement affiché, ce qui constitue une sauvegarde permanente – une valeur affichée ne peut ples être «oubliée». Ensuite la machine «oublie» tout son état interne, effectue le *backtracking*, efface les piles, remonte le graphe (arbre) décisionnel, et suit une autre branche.

L'analyse syntaxique possède le côté non-déterministe, et nous aurons besoin de coder de telles opérations. Mais nous ne voulons pas de «magie» trop évidente, il serait utile de pouvoir réaliser le non-déterminisme de manière classique. Bien sûr, on peut mettre toutes les solutions dans une liste.

2.4.2 Programmation par contraintes

Ce sous-domaine a évolué partiellement à partir des langages logiques, et s'est partiellement inspiré par des applications numériques. Il s'agit de rendre symétrique une relation, par exemple **plus(A,B,C)**. On peut imaginer que si la sémantique d'une telle clause représente l'assignation **C=A+B**, mais si les variables **A** et **C** sont connues, et **B** – inconnue, la machine «comprendra» qu'il s'agit de l'instruction **B=C-A**.

Les langages à contraintes (CLP, Bertrand, Eclipse) et plusieurs autres sont capables de résoudre *automatiquement* les équations numériques ou logiques (dans des domaines finis). L'importance de cette catégorie ne cesse pas d'augmenter. Les sous-systèmes de programmation par contraintes comme Garnet (ou ses successeurs) sont devenus incontournables dans la construction des interfaces utilisateur. (Et servent – par exemple – à placer automatiquement ou presque, les composantes : boutons, menus, zones texte etc. dans la fenêtre application, en respectant des contraintes géométriques). Les modéliseurs 3D exploitent aussi très intensément ce style. (Mais Garnet n'est pas un langage : c'est une puissante librairie d'interfaçage en Common Lisp. Il est actuellement obsolète, mais son successeur : Amulet existe, et il est utilisée dans le monde C++).

La *vraie* compilation (avec optimisation et linéarisation) de ces langages est en générale si difficile, que la plupart du travail est effectué lors de l'exécution du programme. L'interprète des contraintes doit être assez intelligent et disposer de plusieurs «solveurs» des équations, des paquetages numériques, des modules de parcours des graphes, etc. Cette catégorie de langages comme peu d'autres démontre que la zone de démarcation entre les compilateurs et les interprètes est vraiment floue…

Notre langage favori qui appartient à ce domaine, et qui a été exploité pour programmer quelques graphes inclus dans ces notes est MetaPost. Ce langage, construit par John Hobby, est une distillation du langage Metafont de Donald Knuth. Metafont a été conçu pour générer des familles entières de polices de caractères. Le créateur précisait quelques attributs géométriques de la police, parfois sous forme d'équations : «ces deux lignes doivent être parallèles», etc., et le paquetage construisait le jeu de caractères complet. Malheureusement, Metafont est resté inconnu, car combien de créateurs de polices y a-t-il dans le monde?

Cependant le *langage* est vraiment universel et facile. MetaPost est un macro-processeur permettant à l'utilisateur l'usage des structures lexico-syntaxiques comme **3x** ce qui signifie **3*x** dans des langages plus classiques, et où la boucle **for i=1 upto 9** n'est pas une construction primitive, mais une macro où on a défini

```
def upto = step 1 until enddef;
```

Un tel massacre syntaxique n'est pas possible dans les langages structurés classiques.

La *compilation des contraintes*, leur transformation en *code exécutable* nécessite une analyse profonde des relations syntaxiques. La forme **A+B+C=D*E** n'est plus une arborescence qui permet au générateur du code de «ramasser» les sous-expressions et de construire le résultat final, car on ne sait pas *a priori* quelles sont les sous-expressions connues. Cette classe de langages est un terrain formidable pour l'analyse sémantique profonde et pour les exercices en parcours des graphes.

2.5 Programmation par objets

La programmation OO, les objets, les messages et méthodes – tout ceci est devenu malheureusement un super-domaine rempli de slogans et défini de façon incongrue. Le lecteur connaît C++ et nous n'avons pas besoin de définir les concepts de base. Il faut rappeler que la technique est née avec le langage Simula, et a été développée de manière exhaustive dans le cadre du langage Smalltalk. (Simula 67 était d'ailleurs le premier langage populaire avec des co-procédures, très commodes pour l'implantation de la simulation de systèmes dynamiques.)

L'idée de bas niveau est simple : si nous avons la possibilité de construire les *records*, les données composites, nous pouvons prévoir qu'un ou plusieurs champs de nos données soient des *fonctions* qui «savent» comment traiter ces données (les autres champs) de manière spécifique, appropriée. La donnée s'appelle désormais *objet*. L'appel de la fonction qui est attachée à notre objet-donnée s'exprime comme *l'envoi du message à l'objet*. Ce message déclenche l'exécution d'une *méthode*. On voit qu'un bon langage fonctionnel permet aisément la construction de systèmes à objets. En effet, le nombre de paquetages OO en Lisp dépasse une centaine, leur construction est devenu un exercice pédagogique classique.

La vraie puissance des langages OO est la généricité – la possibilité de grouper des objets dans des **classes** qui partagent les mêmes fonctionnalités, et l'héritage : un mécanisme permettant de construire des sous-classes, de spécifier des objets un peu différents des autres, avec des méthodes particulières, mais qui peuvent automatiquement «hériter» le comportement de leurs «ancêtres» – d'autres objets, définis au préalable.

Les méthodes d'implantation de l'héritage sont nombreuses. Supposons que les objets x et y appartient aux classes X et Y, et ces deux classes définissent la méthode f. (Une de ces classes peut être la sous-classe de l'autre). L'appel «interne» (défini dans la classe du «récepteur») $f(\ldots)$ d'une méthode peut être réalisé de manière suivante :

- Le compilateur génère le code $f(self, \ldots)$, où *self* est le récepteur x ou y du «message» f.
- S'il n'y a pas d'ambiguïté, le compilateur, sachant quelle est la classe du récepteur, connaît la procédure attachée au nom f. L'appel est compilé normalement.
- Si le recepteur peut appartenir à X ou à Y, et on ne peut résoudre ce dilemme statiquement, l'appel ne peut être compilé directement. La méthode f devient «virtuelle». Ceci implique les surcharges spatiales et temporelles suivantes :
  - Chaque objet (structure de données) appartenant à une classe qui dispose de méthodes virtuelles possède un champ de plus – la référence vers un ***dictionnaire de méthodes virtuelles***, un tableau associatif stocké dans la classe de l'objet. (En fait, la classe en tant qu'objet «physique» **est** le dictionnaire des méthodes virtuelles).
  - La compilation de l'appel développé : $f(self, \ldots)$ contient un code indirect : on récupère le dictionnaire accessible par *self*, on décode l'objet procédural attaché à f, et on l'appelle.

La compilation des langages OO peut être donc assez facile, si toute décision est laissé à la machine virtuelle, mais elle peut contenir des optimisations extrêmement importantes, et partiellement à cause de cela les compilateurs de C++ sont très grands…

2.5.1 Quelques exemples

Les langages à objets sont si nombreux, qu'une litanie de noms ne servira à rien. Le langage dominant pour les grands programmes est C++ qui doit sa popularité principalement au fait que son ancêtre : le langage C est si populaire. C'est un langage riche et difficile à maîtriser. Pour construire un compilateur de C++ réaliste il faut une équipe de personnes très compétentes (cependant le *design* original est l'œuvre d'une personne : Bjarne Stroustrup. Ce même Stroustrup avoue publiquement qu'il est loin de maîtriser toutes les intrications du langage…).

Actuellement une bonne partie du «marché» C++ diverge dans la direction de Java, qui est (d'habitude) interprété, alors plus lent, et un peu plus pauvre au niveau de syntaxe (pas de surcharge des opérateurs, pas d'héritage multiple, etc.), mais qui est bien adapté à la construction des programmes sécurisés. Il faut noter que Microsoft s'est engagé récemment dans la construction d'une «plate-forme virtuelle» `.net`, la définition d'un noyau intermédiaire entre le hardware et des langages évolués, qui pourra faciliter la compilation de tous les

langages imaginables. Mais les spécialistes affirment, que ce noyau semble être vraiment bien adapté à Java, beaucoup moins aux langages fonctionnels (par exemple).

Comme il a été dit, les langages fonctionnels constituent une bonne plate-forme pour implanter les langages à objets. Parmi eux, la position privilégié par le nombre d'utilisateurs est occupée par CLOS (*Common Lisp Object System*). La syntaxe reste presque la même qu'en Lisp, ce qui n'est pas très clair, mais CLOS a ses inconditionnels. Lisp, Scheme, et autres langages de cette famille ont donné naissance à des langages à objets innombrables.

Les langages fonctionnels modernes reconstruisent ses systèmes de typage hiérarchique (qui réalise le polymorphisme restreint et l'héritage) de manière différente. Les langages comme Objective CAML, Haskell ou Clean méritent aussi d'appartenir un peu à la famille OO.

Depuis quelques années un autre langage à objets : Python fait une belle carrière. Le langage est simple et joli, et très transparent. On peut l'apprendre en quelques jours, et écrire des applications graphiques très performantes. Il remplace de plus en plus souvent le langage Perl. Python, grâce à sa transparence est utilisé dans quelques établissements comme le langage-modèle sur lequel les étudiants apprennent *l'implantation* des langages à objets.

Il faut mentionner ici le langage Eiffel, qui a ses partisans déclarés. Ce langage a été conçu par Bertrand Meyer, un Français (comme le nom du langage le suggère). Mais Eiffel est développé surtout aux États Unis.

Un autre langage pragmatique, sans trop d'ambitions théoriques est apparu récemment : Ruby, développé surtout au Japon. Encore plus facile que Python, mais – selon l'auteur de ces notes – moins intéressant. C'est un langage de genre «quick-and-dirty», programmation facile et rapide pour les gens qui ne veulent apprendre rien qui dépasse leurs objectifs immédiats.

Notons que l'ancêtre des langages OO – Smalltalk vit actuellement une visible renaissance. Il existe au moins 3 implantations commerciales sérieuses (comme Visual Works, disponible aussi gratuitement), et deux implantations sérieuses gratuites : GNU Smalltalk, qui permet de faire beaucoup d'expériences de programmation, et Squeak – une vraie merveille, avec une couche graphique étonnante, multi-plate-forme, et qui possède déjà plusieurs milliers de supporters. L'intérêt pour la compilation de ce langage (et ses implantations) est qu'une bonne partie de la machine virtuelle sous-jacente et du compilateur ont été écrites en Smalltalk, ce qui permet leur analyse et expérimentation.

Par contre, le *premier* langage qui mérite être appelé un langage à objets : Simula 67 a été complètement oublié. Ceci est dommage, car Simula était également le premier langage relativement populaire, qui permettait la programmation dans le *style co-procédural*, très intéressant et important pour la simulation des systèmes quasi-parallèles. Actuellement aucun langage populaire ne gère des co-procédures de manière si instructive et transparente.

2.6 Programmation pilotée par les événements

Ce modèle est vraiment différent de la «programmation classique», et jusqu'aujourd'hui il est *rarement* enseigné. Même si dans le cadre de la programmation OO on parle d'envoi de messages d'un objet à l'autre, il s'agit toujours d'un appel de type procédural, synchrone, avec l'empilement de l'adresse du module envoyant. Si l'objet X envoie le message à l'objet Y, cela veut dire qu'une fonction (méthode) dans la classe de l'objet Y est appelée par une fonction appartenant au contexte X, et c'est tout.

Par contre, la programmation par événements constitue réellement l'envoi des messages au sens intuitif, proche du modèle co-procédural. Quand vous envoyez un message, vous continuez votre travail là où vous l'avez suspendu. Au lieu de déclencher explicitement une activité de la part du module appelé, l'appelant place dans une *file globale d'événements* le message (descripteur d'une activité future) destiné à un ou plusieurs récepteurs, et poursuit l'exécution de son code.

La machine virtuelle d'un tel système dispose d'un «dispatcher» global qui lit la file des événements, décode ses éléments et appelle les procédures concernées. Tout ceci se déroule en *dehors de contrôle* de la part du programme utilisateur. Les événements peuvent être «artificiels», des messages de nature quelconque que remplacent les appels procéduraux, mais également «naturels», émis, par exemple, par la procédure système qui contrôle la souris et le clavier.

La programmation événementielle est devenu incontournable pour la création des interfaces graphiques et pour la simulation. Dans d'autres contextes, par exemple dans l'intelligence artificielle, on exploite des principes

analogues : la technique du «tableau-noir» (*blackboard*), où les modules-experts envoient indépendamment leurs propositions de solution d'un problème global vers une zone de ressources partagées.

Les langages spécifiquement orientés-événement sont rares. On peut mentionner ici le Lingo, le langage de *scripting* du paquetage multi-médiatique Director (aucun progrès récent. . .).

Cependant un nouvel élan a été donné à la programmation événementielle par les techniques de programmation graphique. Un de meilleurs systèmes de programmation scientifique – Matlab est équipé à présent avec un sous-système qui s'appelle Stateflow, et qui permet d'organiser graphiquement, dans un style très intuitif et élégant la totalité de transitions entre les éléments d'un système complexe. Stateflow est une évolution du Simulink, un langage graphique type *dataflow*, appartenant à la famille mentionnée ci-dessous.

2.7 Dataflow et langages graphiques/visuels

La programmation classique impérative est basée sur le paradigme de *contrôle*, le programme détermine la suite des événements et distribue les tâches aux modules subordonnés. La vision fonctionnelle ou logique n'est pas *très* différente de celle-là, bien que ciblée plus sur les relations entre données. Même la programmation par objets, qui semble se concentrer sur les données est orientée réellement vers l'exécution des procédures-méthodes ; seulement ces méthodes sont attachées aux données. Mais, justement, la vision orienté données semble être mieux adaptée à l'organisation de plusieurs logiciels qui soit modélisent le monde «réel», soit enchaînent des opérations complexes sur des structures complexes, par exemple effectuent la segmentation d'images.

La programmation *dataflow* est une autre approche. Chaque module (fonction, procédure, prédicat – choisissez le terme le plus convenable, dépendant du modèle d'implantation), peut être visualisé comme un «boîtier» avec un certain nombre d'entrées et de sorties.

Les entrées correspondent aux arguments d'entrée, et les sorties – aux arguments calculés par la procédure, ou à la valeur (éventuellement multiple) retournée par la fonction. Les boîtiers sont liés par des lignes comme dans un circuit électronique, et ainsi un système plus complexe est construit à partir de boîtiers plus primitifs. Tout paquetage de ce genre prévoit un nombre important de boîtiers primitifs : le module d'affichage du résultat (textuel ou graphique), le multiplexeur qui transforme plusieurs lignes élémentaires en une vectorielle (ou vice-versa), et les «prises» d'entrée et de sortie qui permettent la construction d'une librairie de boîtiers. Prenons un exemple, un bloc de simulation montré sur la Fig. (2.2). Les diagrammes peuvent être rédigés et parametrés.

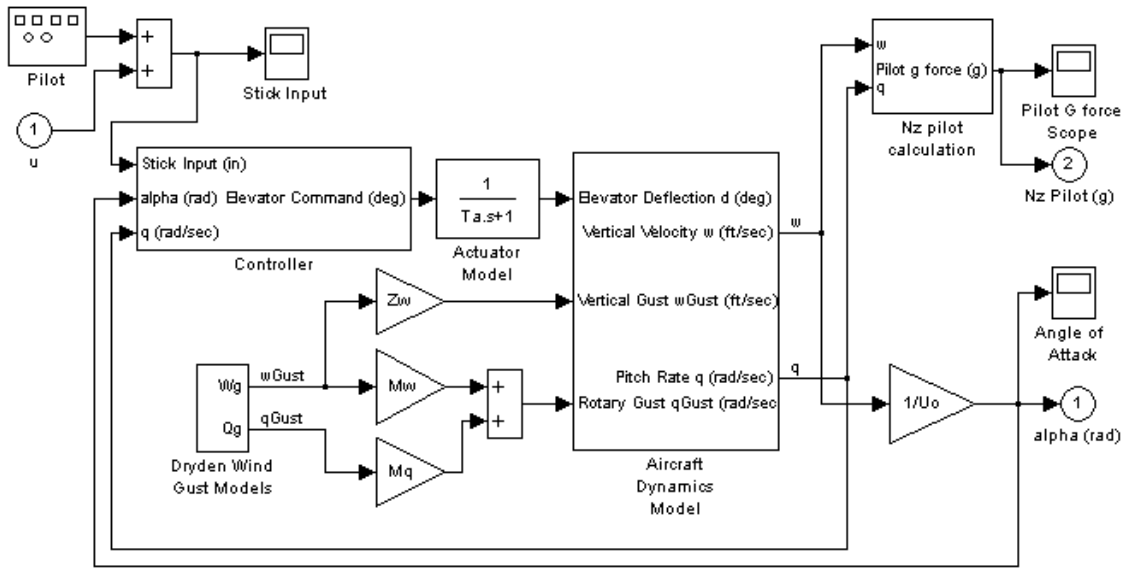


Fig. 2.2: Un diagramme Simulink

L'exécution d'un programme est pilotée par les données. Quand un bloc termine son travail, les données sont «pipelinées», injectées dans un autre bloc, qui à ce moment-là commence son exécution.

Le *compilateur* d'un langage *dataflow* est très particulier. Il s'agit de générer le code dont l'exécution est déclenchée par la présence des données, et ce code est naturellement quasi-parallèle, alors adapté aux architectures multi-processeurs, ce qui demande un système de simulation sur les architectures classiques. La tâche principale de la machine *dataflow* est d'établir la synchronisation entre les boîtiers, donc le code généré par le compilateur est assez particulier.

Le succès grandissant de cette famille de langages (jadis considérée comme une invention académique de quelques informaticiens imaginatifs, mais pas très pratiques), montre d'une part que la programmation *visuelle*, basée sur les techniques d'interfaçage modernes est très importante.

D'autre part – et ceci est très intéressant pour nous – la composition visuelle des modules logiciels complexes peut nous inspirer à modéliser ainsi un compilateur !

Simulink est un produit commercial, disponible (avec Matlab) sur presque toutes les plates-formes populaires.

L'institut INRIA a produit un paquetage gratuit qui ressemble beaucoup à Matlab – SciLab. Ce système possède également un langage *dataflow* – SciCos, un peu moins complet que Simulink, mais aussi très riche.

Il existent au moins trois grands paquetages de traitement de signaux et images basés sur *dataflow* qui sont facilement accessibles : SciCos, Khoros qui est commercialisé, mais dont la version pédagogique est gratuite (pour les étudiants, à titre individuel), et IBM Data Explorer distribué selon les règles du *Open Source*.

Ce catalogue aurait pu être beaucoup plus long et détaillé. Son but est de convaincre le lecteur d'une simple chose : la compilation n'est pas et ne sera jamais un domaine fini. Puisque les nouveaux langages, styles, protocoles, sémantiques et techniques de gestion de mémoire apparaissent tous les ans, les techniques de compilation doivent suivre ce développement.

2.8 Autres schémas de classification et paradigmes de programmation

Une autre classification est basée sur la vieille et mal comprise dichotomie : langage compilé – langage interprété. C++ est compilé, Perl ou Python sont interprétés. Lisp a été longtemps interprété, maintenant on trouve des compilateurs (pour Java et Perl aussi). Cette classification en principe est étrangère à notre philosophie. Dans les livres on trouve souvent un slogan douteux : «Un interprète exécute directement un programme *instruction par instruction* sans le traduire en code-machine, tandis qu'un compilateur effectue d'abord cette translation».

Tout langage est compilé **et** *interprété*. Compilé, car il faut traduire le texte-source d'un programme en code interne. Il faut reconnaître les lexèmes et les transformer en atomes, il faut bâtir les arborescences syntaxiques, appliquer les règles sémantiques spécifiques à chaque opérateur, optimiser le code, etc. On peut préciser que si la compilation va jusqu'au bout et produit un code exécuté directement par le processeur *matériel*, le langage est compilé, et s'il s'agit du code intermédiaire : «bytecodes», liste des pointeurs ou autres structures de données, alors le langage est interprété par une machine virtuelle de plus haut niveau. Mais il n'y a pas de ligne de démarcation distincte entre les deux mondes. Un programme Java peut intégrer *bytecodes* et procédures écrites en C. Dans un programme «interprété» l'instruction : le *bytecode* ou un pointeur après son décodage lance l'exécution d'une suite d'instructions en code machine. Dans un langage «compilé», mais orienté-objet, l'exécution d'une méthode virtuelle fait exactement la même chose. Parfois même en C ou Fortran il est avantageux pour le déboguage d'organiser l'architecture globale du programme comme une liste ou un tableau d'adresses de procédures primitives. Quelques librairies graphiques, par exemple OpenGL, codées en C et compilées, constituent les automates (*state machines*), qui sont des véritables interprètes.

En plus, l'exécution d'une instruction du code assembleur n'est rien d'autre que l'exécution d'un microprogramme cablé dans le silicium. Une micro-instruction se traduit par un «programme» style dataflow, mais où les données qui circulent sont des électrons, et les sommets du graphe représentant l'automate – des transistors. Toute machine virtuelle a un certain nombre d'instructions primitives qui sont exécutées «par magie», et cette magie est un programme de la machine en dessous, de plus bas niveau. Finalement on arrive au niveau des transitions quantiques, et cette magie n'a aujourd'hui aucune explication. Le mot «magie» sera donc utilisé assez souvent lors de ce cours, et il possède une signification technique et rationnelle : instructions etc. magiques appartiennent à la couche plus basse que celle qui est actuellement discutée.

Donc la dichotomie compilé-interprété est une question de niveau de réalisation de la machine virtuelle. Bref,

***Credo* religieux no. 3 :** on ne peut pas apprendre à construire les compilateurs, si on ne maîtrise pas la sémantique de la machine *conceptuelle* – ou le modèle qui exécute le programme.

La compilation contient une partie passive, analytique ; le compilateur doit *comprendre* le texte-source, attribuer une signification à tous les éléments du programme. Donc, la personne qui définit un langage, et qui construit l'analyseur, doit savoir ce qu'elle fait, et ceci constitue l'essence du *credo* no. 1. Le résultat de cette compréhension par le compilateur est la synthèse du code-cible par le module actif – le générateur du code, et cette synthèse est *l'explication* de ce qui a été compris. Mais on ne peut expliquer le programme à une machine ou à un humain, que si on connaît **son** langage.

Si on enseigne la génération du code assembleur sans préciser les détails de la sémantique, de la signification des instructions en assembleur, on n'enseigne qu'un rituel religieux. Si, comme nous le voulons – on se concentre plutôt sur le code-cible interprété par des machines virtuelles de niveau intermédiaire, assurant la portabilité et l'efficacité, comme Java ou PostScript, il faut alors obligatoirement savoir comment ces machines marchent, et la meilleure façon de l'apprendre est d'en construire quelques unes. Voici le sens du *credo* no. 3.

D'autres critères de classification des langages existent également. Parfois on distingue les langages universels et les langages dédiés, spécifiques à un domaine, comme les langages de requêtes de bases de données, ou quelques langages de calcul formel, riches en mathématiques et pauvres en organisations des données universelles et structures de contrôle. Mais une telle classification est toujours incomplète. Tout langage réputé universel sera trop pauvre pour quelqu'un, par contre, il y aura toujours des optimistes incurables, qui pensent qu'un langage de calcul formel comme Maple soit bon pour les lycéens (voir l'exercice)…

Finalement, récemment une prolifération très importante des *langages de spécification* a eu lieu. La complexité syntaxique du VRML ou SGML et ses variantes (par exemple MathML – langage de spécification des structures mathématiques), est sérieuse, mais ce ne sont pas des véritables langages de programmation, car le «code» est statique : la notion d'*état* y est absente, et aucun *flux de données* n'est généré. Le compilateur se réduit à un parseur, et à la construction d'une structure de données de haut niveau.

Les concepts vraiment universels dans ce domaine sont peu nombreux. Chacun doit comprendre intuitivement ce qui est une constante numérique, un caractère, ou une fonction. Mais le concept d'*adresse* n'est pas si universel, car appartient à la description de bas niveau, comme le pointeur. L'*instruction* est aussi un concept du monde impératif, absent dans les langages fonctionnels.

2.8.1 Types

Mais le concept de *type* même si pas tellement universel (le code assembleur peut voir toutes les données comme séquences de bits) est si universel, que nous allons consacrer beaucoup d'attention au *typage*. Ceci constitue une des bases de la sémantique des langages de programmation en général et facilite la compréhension de la compilation des langages orientés-objet. On divise donc les langages en typés dynamiquement : Scheme, Icon, Prolog, Perl, et typés staiquement (chaque variable se voit attribuer un type lors de la compilation) : C , Java, Haskell, etc. Les langages typés statiquement sont d'habitude plus efficaces (rapides), car on peut éviter beaucoup de tests durant l'exécution du programme.

On parle souvent du *polymorphisme* – la possibilité d'appliquer une fonction donnée à des arguments hétérogènes, mais il ne faut pas confondre les deux catégories suivantes :

- La surcharge des opérateurs, la possibilité de pouvoir écrire **`x+y`** pour **`x`** etc. entiers ou réels, chaînes alphanumériques, ou nombres complexes. Ici «**+**» est le nom commun à des opérations *différentes* qui peuvent ne rien avoir en commun.

- Le vrai polymorphisme sémantique, par exemple l'extraction du second élément d'une liste :

  ```
  (define (second l) (car (cdr l)))
  ```

  en Scheme, et

  ```
  second (_ : x : _) = x
  ```

en Haskell. La valeur retournée peut être de type absolument quelconque, car la fonction ne l'utilise pas, elle la transmet à son consommateur.

La surcharge (overloading) est un exercice relativement facile, c'est une question de renommage. L'implantation du vrai polymorphisme est plus délicate, car une fonction polymorphe doit se compiler et s'exécuter correctement sans savoir quel est le type de données, ou sa représentation dans la mémoire. Il faut souligner : la fonction *ne vérifie pas le type dynamiquement*, comme les fonctions arithmétiques en Lisp, mais l'ignore jusqu'à la fin. La représentation des données peut et doit cacher les détails.

2.9 Notre langage de travail

Nos critères de choix du langage proposé pour la présentation des algorithmes de compilation : Haskell, et la réalisation des machines virtuelles sont les suivants :

1. Syntaxe compacte et lisible ; pas trop des redondances, de surcharge syntaxique (c'est-à-dire : pas beaucoup de mots-clé et d'autres verbosités), mais relativement intuitive.

2. Outils de construction de *données* raisonnables, car les données plutôt que les procédures déterminent si un langage est approprié pour la construction des compilateurs ou des interprètes.

3. Accessible à tous, facile à apprendre, et suffisamment puissant pour démontrer quelques programmes non-triviaux sans avoir besoin de bibliothèques chargées séparemment, ou des fichiers-entêtes énormes. L'implantation du langage doit évidemment être gratuite et disponible sur toutes les plates-formes populaires.

4. Universel, capable de permettre la discussion (et l'implantation) de structures pertinentes à d'autres langages.

5. Universel dans un autre sens, sans spécificités difficilement traduisibles dans d'autres langages. On sait que les techniques du *parsing* non-déterministe s'expliquent et se réalisent aisément en Prolog qui est un langage non-déterministe. Mais ainsi nous n'apprendrons rien sur la *réalisation* de bas niveau de ce concept, et la traduction de notre parseur en C serait difficile.

   Également, si pour construire la couche objet (les méthodes, l'héritage…) du langage compilé on utilise les objets et les concepts du langage d'implantation, la technique peut être élégante et efficace (une bonne partie du compilateur-interprète de Smalltalk est écrite en Smalltalk), mais on sera «coincé» dans ce langage[2].

2.10 Exercices

Q1. Est-ce que le langage C++ est polymorphe? Justifier la réponse, éventuellement donner les exemples si elle est positive.

R1. Analyser les pointeurs, peut-être ici. Ailleurs on n'a pas beaucoup de chances pour trouver le vrai polymorphisme en C++. Mais analyser aussi les instructions d'entrée/sortie formatée. En tout cas, le vrai but de cette question est de forcer les lecteurs à se poser la question sur la vraie signification du mot *polymorphisme*, qui parfois a d'autres signification (comme dans le jargon des *design Patterns*…).

Q2. Pourquoi Maple (ou un autre langage de calcul formel) n'est forcément pas adapté à l'initiation à la programmation, p. ex. au Lycée?

R2. Bien sûr, nous n'attendont vraiment aucune réponse de la part des lecteurs, sauf si quelqu'un a déjà eu l'expérience avec un tel enseignement. Cet «exercice» est un peu anecdotique…

Le danger est le suivant : les élèves confondent très vite une *variable* au sens classique dans la programmation, et une valeur «non-déterminée», un symbole, disons x qui représente une valeur algébrique

[2] Cette propriété ne sera pas respectée entièrement : Haskell est un langage **paresseux**, et les structures paresseuses sont difficilement traduisibles en C ; ceci a déjà été commenté, et sera encore discuté plus tard.

manipulée par des moyens formels. La distinction devient floue, et ceci constitue un obstacle dans l'apprentissage des langages de programmation «normaux». À cause de cela Maple malgré ses avantages (convivialité, bon support graphique) ne doit pas être enseigné comme le *premier* langage ! Les jeunes qui débarquent en DEUG avec un tel bagage, ont la nécessité de «dé-apprendre» Maple, d'oublier une partie de leur initiation à l'informatique, sinon ils font des sottises pendant plusieurs semaines...

Q3. Essayer d'optimiser, de linéariser le code de la fonction factorielle à l'aide de continuations. (Bien sûr, on aura besoin de la connaissance du Haskell ici).

R3. La définition standard de factorielle :

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

subira la transmutation par continuations classique. Définissons

```
faccnt 0 cnt = cnt 1
faccnt n cnt | n>0 = faccnt (n-1) (\r -> cnt(r*n))
```

On voit que le futur de l'appel récursif de la fonction **`fac`** est la multiplication du résultat par n, et cette manipulation a été incorporée dans la continuation.

Ici la multiplication n'a pas été continuée, la modification est superficielle, et son objectif est de rendre la factorielle récursive terminale. Ceci est une optimisation différente de celle connue – l'ajout d'une variable-tampon. Tampon ici serait plus économique, mais dans de très nombreux cas la connaissance de continuations peut sauver beaucoup de temps.

Q4. Comment réaliser le programme qui calcule les permutations en Haskell? Ce langage est déterministe, alors le non-déterminisme sera simulé par le retour d'une *liste* de solutions individuelles. La liste vide symbolise l'échec : pas de solutions.

R4. La convention est donc la suivante. Si une fonction «classique» renvoie un objet **`x`**, les fonctions dans ce style (que nous pouvons appeler *monadique* pour des raisons qui seront expliquées plus tard) retournent **`[x]`** : le résultat stocké dans une liste. (On peut utiliser d'autres structures de données, par exemple des arbres, mais les listes sont suffisamment universelles).

Commençons par la transformation du prédicat Prolog d'insertion non-déterministe. Rappelons que la réponse non-déterministe était : mettre le nouvel élément **`x`** à la tête, **ou** séparer la tête existante **`y`**, *insérer* **`x`** *quelque part dans le reste*, et restaurer la vieille tête **`y`**. Cette dernière opération n'est pas triviale, car l'appel récursif (en italique ci-dessus) génère une liste de listes, un résultat non-déterministe.

Alors la première question importante se pose : comment appliquer une fonction (normale) à un objet non-déterministe? Il faut l'appliquer *à tous les éléments* de la liste. Nous utiliserons donc la fonctionnelle **`map`**, dont la définition doit être connue :

```
map fun [] = []
map fun(x:q) = (fun x) : map fun q
```

Voici donc la fonction d'insertion non-déterministe. le lecteur voudra la comparer avec la variante en Prolog :

```
ndins x l
 | l==[]     =  [[x]]
 | otherwise =  let (y:q)=l in
    (x:l) : map (y :) (ndins x q)
```

Passons aux permutations. La liste vide possède une permutation triviale. Sinon, on peut enlever la tête, trouver une permutation du reste, et réinsérer la tête n'importe où. Il faut donc répondre à la seconde question : comment appliquer une fonction *non-déterministe* à un argument déjà non-déterministe? D'abord on applique **`map`** de cette fonction à tous les éléments de la liste, mais ainsi le résultat est une liste, dont les éléments sont des listes de listes. Il faut enlever le «parenthésage interne» redondant, aplâtir la liste en concaténant les listes internes. Il existe l'équivalent de la fonction Lisp **`append`** en Haskell, l'opérateur de concaténation **`(++)`**. Nous aurons

```
permut [] = [[]]
permut (x:q) =
   flat (map (ins x) (permut q))
```

où **flat** peut être définie par **flat l = foldr (++) [] l**, et le réducteur choisi ici pour varier n'est plus **foldl**, mais **foldr**, récursif à droite, défini comme :

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

(Cette définion sera encore discutée.)

Observation très importante. Nous avons vu deux réducteurs d'opérateurs binaires sur des listes : **foldl** et **foldr**. On peut avoir – à juste titre – l'impression que **foldl** est plus efficace, car c'est une fonction récursive terminale. **foldl** applique l'opérateur entre la tête et une valeur initiale, et boucle sur la queue de la liste, tandis que **foldr** s'applique à la queue de la liste avant d'appliquer l'opérateur binaire à la tête et la réduction de la queue. Les deux réductions ci-dessous

```
l=[1,3,2,7,2,4,1,2,5,8,2]

a = foldr (+) 0 l
b = foldl (+) 0 l
```

donnent 37, mais **foldl** utilise la mémoire de manière plus économique.

Cependant la réalité n'est pas toujours si simple, car il ne faut pas oublier que Haskell est un langage paresseux, alors le deuxième argument de **f** dans **f x (foldr f z xs)** sera évalué seulement si **f** en a besoin. Ceci n'est pas toujours le cas. Voici la définition de la concaténation de deux listes par **foldr** :

```
l1 ++ l2 = foldr (:) l2 l1
```

qui est correcte même si la liste **l2** est infinie. Le programme suivant est parfaitement correct, et donne **[5, 7, 1, 1, 1]** quand on demande la valeur de **res**.

```
uns = (1:uns)

a = foldr (:) uns [5,7]
res = take 5 a
```

Mais n'essayez pas d'afficher **a** ou **uns**, car l'affichage ne se termine jamais. La fonction **take n l** prend les premiers **n** éléments de la liste.

Si on utilise les listes pour implanter le backtracking, parfois on a besoin de toutes les solutions, et dans ce contexte la sémantique paresseuse et **foldr** ne sont pas utiles. Mais parfois on cherche *la première* solution convenable parmi de très nombreuses, peut-être parmi un nombre infini de solutions possibles. La liste de solutions sera alors consommée de manière incrémentale, paresseuse, et évidemment **foldr** reste la seule variante qui ne fait pas exploser la mémoire.

Q5. Construire la fonction **powerset l** qui prend un ensemble (réalisé comme une liste), et qui renvoie *l'ensemble de tous les sous-ensembles* de son argument, en commençant par l'ensemble vide, et terminant avec l'argument lui-même.

R5. La stratégie est la suivante : on parcourt la liste **l** et on en construit par le choix non-déterministe *une sous-liste quelconque*. Ce choix consiste à : soit prendre un élément, soit le rejeter. Voici la solution en Prolog. Le prédicat **sousens(L,R)** construit **R** comme un sous-ensemble de **L**.

```
sousens([],[]).      % Pas d'autre possibilité
sousens([X|Q],R):-
    sousens(Q,L),    % et ensuite:
   (R=L;             % X rejeté. ";" est l'alternative
    R=[X|L]).        % X accepté
```

En Prolog quand l'utilisateur charge le fichier avec ce prédicat, et l'exécute en demandant l'évaluation de **`sousens([a,b,c,d],R).`**, le système répond **`R=[]`**, et il attend la réaction du programmeur qui peut l'accepter, ou taper le point-virgule qui redémarre la machine non-déterministe, et affiche **`[a]`**. Ainsi nous pouvons récupérer les réponses une par une, mais il est possible de les ramasser ensemble en tapant

```
bagof(Z,(sousens([a,b,c,d],Z)),L).  %% Ceci donne:

Z=_x2235,                           %% n'importe quoi, nom interne
L=[[],[a],[b],[a,b],[c],[a,c],[b,c],[a,b,c],[d],[a,d],
   [b,d],[a,b,d],[c,d],[a,c,d],[b,c,d],[a,b,c,d]]
```

En Haskell la stratégie sera exactement la même. On construit la solution partielle sans la tête, et ceci nous donne la moitié de la solution finale – tous les sous-ensembles qui ne contiennent pas la tête. Pour construire les sous-ensembles qui la contiennent il suffit de l'ajouter, et de concaténer les deux parties

```
sousens [] = [[]]
sousens (x:l) = let part=sousens l in
   part ++ map (x :) part
```

Question accessoire, obligatoire à tous ceux qui ont un minimum ϵ d'ambition : Prouver que la cardinalité du **`(sousens l)`** est égale à 2^n, où n est la longueur de **`l`**.

Q6. Construire la fonction **`take`**

R6. Ah, non, essayez vraiment vous-même. Si vous n'avez pas le courage, regardez le *Standard Prelude* de Haskell. Cette fonction est prédéfinie.

Q7. Est-ce que l'expression **`foldl (:) l1 l2`** est légale? Non? Pourquoi? Comment y remédier?

R7. L'erreur est déclenchée par le vérificateur des types. L'opérateur **`(:)`** n'est pas symétrique, son premier argument est un objet, et le second – une liste des objets du même type. L'expression incriminée applique cet opérateur dans mauvais sens. Par contre, ceci est légal : **`foldl (flip (:)) l1 l2`**. Qu'est-ce que cela donne?

Q8. Quel est le *type* de la fonction **`ins`**?

R8. Haskell nous dit :

```
ins  ::   Eq a => a -> [a] -> [[a]]
```

alors : deux arguments, le premier d'un type inconnu **`a`**, et le second – une liste composée des éléments du même type. Le résultat est une liste de listes. Le préfixe **`Eq a => ...`** signifie que Haskell a bien reconnu **`ins`** comme une fonction polymorphe, mais il a automatiquement restreint le type **`a`** à la classe de types qui admet la relation d'égalité. Pourquoi? Est-il possible d'enlever cette contrainte?

Q9. Construire une fonction en Haskell **`(comb k l)`** qui génère toutes les *combinaisons* de k objets parmi tous n éléments de la liste **`l`**. (Leur nombre est égal au coefficient binomial de Newton : $\binom{n}{k}$.

R9. L'algorithme repose sur un choix itératif : il faut parcourir tout l'ensemble de n objets et soit choisir (accepter) un objet, ou le refuser, comme avec *powerset*. Le nombre de choix positifs doit être égal à k. Si on rejette la tête, il faut choisir k éléments parmi les restants. Si on l'accepte, il faut encore en choisir $k-1$, et réinsérer la tête. le programme est d'une simplicité exemplaire :

```
comb k l =
 let cmb k n l
     | k<0       = []
     | k==0      = [[]]
     | k==n      = [l]
     | otherwise = let (x:q)=l in
        cmb k (n-1) q ++ map (x :) (cmb (k-1) (n-1) q)
 in cmb k (length l) l
```

Le résultat concatène les deux solutions partielles. Cette stratégie prouve accessoirement la validité du théorème

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \tag{2.1}$$

Et, pour comparer, voici la solution en Prolog :

```
comb(K,L,R) :- length(L,N),cmb(K,N,L).
cmb(K,_,_,[]) :- K<0.
cmb(K,_,_,[[]]) :- K=0.
cmb(K,N,[X|L],R) :- N1 is N-1,
   (cmb(K,N1,L,R) ; K1 is K-1, cmb(K1,N1,L,R1), R=[X|R1]).
```

où l'opérateur point-virgule dénote une alternative non-déterministe, ce qui est plus compact que l'écriture de deux clauses séparées. Nous rappelons qu'en Prolog on génère *une* solution «quelconque» et on a pas besoin de concaténer quoi que ce soit. L'opérateur **is** force l'évaluation arithmétique, l'opérateur **=** est l'unification qui peut attribuer la structure à droite à la variable à gauche, mais ne déclenche pas l'évaluation numérique.

Q10. Construire les combinateurs **`dupl`** et **`comp`** (le compositeur : **`(.)`**avec **`subs`** et **`const`**.

R10. Le duplicateur :

```
f x x = f x (id x) = subs f id x

==>    dupl f = subs f id
              = flip subs id f
```

et donc **`dupl = flip subs id`**. La définition de **`id`** est déjà connue. Et le **`flip`**? Rien de plus simple :

```
flip f x y = f y x
           = subs f (const x) y

==>          flip f = (subs f) . const
```

Le compositeur est un peu plus tordu :

```
f (g x) = (const f x) (g x) = subs (const f) g x

==>    comp f = subs (const f)
```

et l'élimination complète de **`f`**, afin d'obtenir une définition combinatoire semble être difficile. La dernière forme se réduit à… :

```
comp f = subs (const f) = (comp subs const) f
```

ou **`comp = comp subs const`**, ce qui mène nulle part…

Q11. Écrire une procédure de tri des listes en Haskell

R11. Construisons le tri rapide des listes (*quicksort*). Rappelons le principe de cet algorithme :

- On choisit un élément quelconque de la collection, le «pivot». Pour des listes il est naturel de prendre la tête, car elle est directement accessible.
- On partitionne la liste en deux sous-listes : le éléments petits, et éléments grands par rapport au pivot.
- On effectue (récursivement) le tri des deux sous-listes.
- On concatène les résultats, en mettant le pivot au milieu.

Voici le code qui utilise les compréhensions :

```
qsort [] = []
qsort [x] = [x]
qsort (x:l) =
  qsort [p | p<-l,p<=x] ++ [x] ++ qsort [g | g<-l,g>x]
```

Il n'est pas optimal. La liste-argument est parcourue et filtrée deux fois pour construire les sous-listes contenant les éléments grands et petits. On peut faire mieux, voici le code optimisé :

```
qsort1 [] = []
qsort1 (x:q) = qs q [] [] where
  qs (a:aq) p g | a<=x = qs aq (a:p) g
                |otherwise = qs aq p (a:g)
  qs [] p g = qsort1 p ++ (x : qsort1 g)
```

Cherchez d'autres optimisations, par exemple l'élimination de la concaténation par l'introduction d'un argument-tampon.

Q12. Construire l'algorithme qui génère *tous* les nombres premiers par la technique du crible d'Eratosthène.

R12. Le crible prend une séquence de nombres entiers, met à part le premier élément, et élimine (filtre) tous les multiples de cet élément du reste de la séquence. Ceci constitue une étape du filtrage. Si on répète la même opération sur la queue, et si on continue, à la fin seulement les nombres premiers auront le droit de rester dans la liste. Voici la procédure complète :

```
prims = sieve [2 ..]  where
 sieve (x:q) = x :
               filter (\m -> m 'mod' x /=0) (sieve q)
```

La liste **`[ 2 .. ]`** est une liste *paresseuse* infinie : 2, 3, 4, ... qui est une abréviation de **`intsFrom 2`** :

```
intsFrom n = n : intsFrom (n+1)
```

Essayez d'optimiser cette solution.

Q13. Le «Prélude Standard» de Haskell contient une fonction qui combine ensemble deux listes, élément par élément avec un opérateur binaire :

```
zipWith oper (x:xq) (y:yq) = oper x y : zipWith oper xq yq
```

Quel est le contenu de la liste paresseuse **`mystere`** définie ci-dessous

```
mystere = 0 : q where
  q = zipWith (+) uns mystere
  uns = 1 : uns
```

Pourquoi?

R13. Pas de réponse ici. Veuillez tester ce programme. Il a déjà été donné une fois comme sujet d'examen.

Q14. Regardez la définition de la fonction **`filter`**. On note que l'appel récursif **`filter p xq`** est effectué indépendamment de la condition **`p x`**. Est-ce possible alors d'optimiser cette fonction de manière suivante :

```
filter p [] = []
filter p (x:xq) = let rst = filter p xq
                  in if p x then p:rst else rst
```

R14. Oui c'est possible, mais il faut faire attention. La première solution est incrémentale, elle marche même avec les listes infinies, tandis que la solution «optimisée» dans ces circonstances déborde la pile. Elle est récursive non-terminale, et *stricte* : la queue est évaluée inconditionnellement. Donc ceci peut être une optimisation réelle si la liste est courte et si on sait que la totalité de la liste-source sera parcourue, et que la totalité du résultat doit être construite, et disponible en même temps.

Q15. Une technique d'approximation de la fonction sinus exploite la récursivité et la formule de triplication de l'angle :

$$\sin(3x) = 3\sin(x) - 4\left(\sin(x)\right)^3 \ . \tag{2.2}$$

Implanter cet algorithme en Haskell (Supposons que l'on doit implanter une librairie numérique standard pour Haskell…). Ensuite **optimizer** la solution de façon très profonde : la fonction doit être *itérative*. Pour simplicité (pour ne pas être obligé de réduire l'argument en utilisant les propriétés des fonctions trigonométriques) on suppose que l'argument se trouve entre 0 et π.

R15. Voici la solution triviale, qui arrête la récursion quand l'argument est très petit :

```
epsilon = 0.000001
monsin x | x<epsilon = x
         | otherwise = let z = monsin (x/3.0)
                       in z*(3.0 - 4.0*z*z)
```

L'optimisation consiste à observer qu'il suffit de réduire d'abord l'argument jusqu'à la valeur souhaitée en comptant le nombre de réductions : m, et ensuite d'appliquer la formule **`z*(3...`** m fois.

```
msin x = ms x 0 where
  ms x m | x>=epsilon = ms (x/3.0) (m+1)
         | otherwise = mq x m
  mq z 0 = z
  mq z m = mq (z*(3.0 - 4.0*z*z)) (m-1)
```

Q16. Les listes contenant des chiffres, p. ex. **`[1,2,0,8,7,2,6]`** représentent dans cet exercice des entiers, ici : 128726. Écrire une fonction qui prend une liste de ce genre, et qui la transforme en un entier numérique «normal». Écrire aussi une fonction qui effectue la transformation inverse, qui transforme un entier en liste.

R16. Cette exercice constitue notre première construction d'un *parseur*. Voici une construction sérieuse. Pour la lecture :

```
nombre l = nb l 0 where
  nb 0 tmp = tmp
  nb (x:xq) tmp = nb xq (10*tmp + x)
```

Pour l'écritue on utilisera la fonction prédéfinie en Hugs : **`divMod`** qui renvoie le résultat de la division Euclidéenne et le reste de cette division de ces arguments dans une paire.

```
affiche n = aff n [] where
  aff 0 buf = buf
  aff n buf = let (d,r) = divMod n 10
              in aff d (r:buf)
```

Q17. Et à présent construire une fonction capable d'ajouter deux nombres dans cette représentation «explosée». Ceci, bien évidemment, n'est pas la somme élément par élément, car il faut surveiller la retenue, et en général les listes peuvent avoir des longueurs différentes.

R17. La solution est un peu pénible car la retenue se propage de droite vers la gauche, il faut donc renverser la liste pour avoir l'accès direct à son dernier élément. La fonction auxiliaire **`addigit`** ajoute un seul chiffre à une liste.

```
dbase = 10

addigit c [] = [c]
addigit c (x:xq) =
  let m = c+x
  in if m<dbase then m:xq else (m-dbase) : addigit 1 xq

addlist l1 l2 = reverse (addl 0 (reverse l1) (reverse l2))  where
  addl c [] l = addigit c l
  addl c l [] = addigit c l
  addl c (x:xq) (y:yq) =
    let (d,r)=divMod (c+x+y) dbase
   in r : addl d xq yq
```

Q18. Est-il possible d'écrire l'algorithme d'addition *sans renverser les listes*? On suppose que les deux ont la même longueur (sinon on peut compléter la plus courte par des zéros), et on exploite de manière assez agressive l'évaluation paresseuse.

R18. *Cherche et tu trouveras*. C'est une solution courte, mais un peu bizarre : il faut «emprunter» la retenue des chiffres qui n'ont pas encore été traités.

Q19. Essayez d'exprimer les fonctionnelles **`map`** et **`foldl`** par **`foldr`**.

R19. Ceci est relativement simple, mais si on ne connaît pas les «trucs du métier», la solution est difficile à trouver. Voir le Prélude standard.

```
map f l = foldr (\a b -> f a : b) [] l

foldl f z l = foldr (\b g a -> g (f a b)) id l z
```

La construction du **`map`** est immédiate : l'élément initial est la liste vide, et le «pliage» consiste à ajouter à cet élément les applications de **`f`** à la liste initiale. D'ailleurs, on peut représenter cette fonction de manière plus compacte, en exploitant les combinateurs :

```
\a b -> f a : b   ≡   \a b -> (:) (f a) b   ≡
\a -> (:) (f a)   ≡   \a -> ((:) . f) a   ≡
((:) . f)
```

Cool, non?

Le **`foldl`** est un peu surprenant, car cette fonctionnelle doit être récursive terminale (itérative), tandis que **`foldr`** empile les résultats intermédiaires. Analysez cette solution pour voir dans quel ordre cette fonctionnelle réduit la liste : de droite ou de gauche.

Q20. Et comment implanter à travers **`foldr`** la fonction de filtrage :

```
filter _ [] = []
filter p (x:xq) | p x = x : filter p xq
                | otherwise = filter p xq
```

R20. `filter p = foldr (\a b -> if p a then a:b else b) []`

Q21. Trouver le type principal du **`foldl`**.

R21. Au travail, au travail !

Chapitre 3

Machines virtuelles et exécution des programmes par l'ordinateur

3.1 Entre compilation et interprétation

Dans l'archive des messages envoyés au *newsgroup* Usenet consacré à la compilation, une série de questions se répète assez souvent : on enseigne la construction de compilateurs. Quel *langage-cible* choisir? Ceci doit être un langage de bas niveau pour que l'exécution du programme soit rapide. Assembleur? Alors lequel? Un assembleur théorique, abstrait? Alors comment vérifier le code? Concret? Mais est-ce raisonnable de coincer les étudiants dans une architecture spécifique qui peut provoquer une perte de temps non-négligeable?

De plus, – comme nous avons déjà souligné – le développement de langages de programmation va dans un autre sens, Java, Python, Prolog, etc. utilisent des instructions *primitives* qui réalisent soit l'aiguillage indirect caractéristique des langages à objets (avec des méthodes virtuelles), soit le non-déterminisme logique qui ne peut être réalisé au niveau assembleur, car demande la possibilité de fournir plusieurs réponses à une question. La solution est de construire des *petits* interprètes de bas niveaux – des machines virtuelles intermédiaires entre le «matériel» (c'est à dire : les microprogrammes qui exécutent les instructions assembleur), et un langage évolué. (En tout cas il ne faut pas essayer de chercher trop d'affinités entre les significations du mot «virtuel» dans *machines virtuelles* et *méthodes virtuelles*...

En utilisant le code intermédiaire nous montrerons qu'il n'y a pas beaucoup de différences conceptuelles entre un interprète et un compilateur. En fait, le compilateur *est* un interprète qui «exécute» (ou évalue) le programme, toutefois le résultat n'est pas une séquence finale de valeurs numériques ou graphiques, mais *le code de plus bas niveau* qui produira cette séquence de valeurs à l'aide d'un autre interprète.

3.2 Expressions fonctionnelles et évaluation récursive

3.2.1 Interprète descendant en Scheme

Construisons un interprète capable d'évaluer une expression arithmétique, par exemple

$$1.0 - 2.0\left(2.0x + \frac{3.0}{y}\right) - 2.0z \tag{3.1}$$

Considérons cette expression comme une structure de données arborescente, obtenue par un analyseur syntaxique, ou codée explicitement par le programmeur en Lisp. L'arbre syntaxique de l'expression (3.1) aura la forme présentée sur la Fig. (3.1).

La première chose à établir est la classe de données primitives gérées par notre machine. Nous aurons les constantes réelles et quelques variables symboliques, considérées ici pour simplicité comme des abréviations des objets globaux. Ensuite il faut préciser la panoplie des opérateurs disponibles. Ici nous avons seulement les opérateurs arithmétiques *binaires* symbolisés par les caractères spéciaux standard. Si nous construisions notre interprète en Scheme ou autre dialecte de Lisp, nous pourrions traiter les feuilles et les nœuds internes comme des atomes, et écrire un évaluateur de listes imbriquées de genre

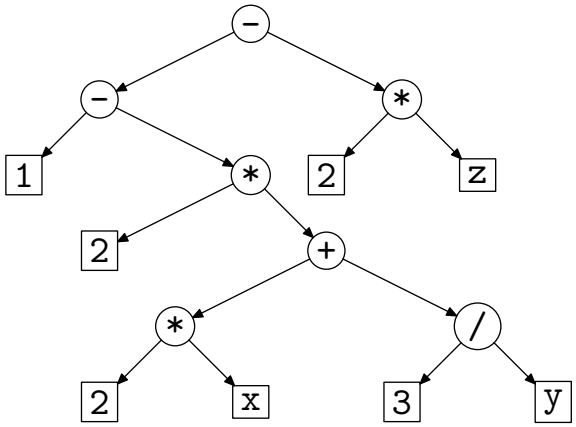


Fig. 3.1: Arborescence syntaxique

```
(-  (- 1 (* 2 (+ (* 2 x) (/ 3 y)))) (* 2 z))
```

La stratégie est évidente: si l'expression est une feuille, alors évaluons sa valeur, soit directement si c'est une constante, soit en cherchant son association (nom – valeur) dans l'environnement courant. Si l'expression est une liste (un nœud interne), évaluons récursivement les deux branches, et appliquons l'opérateur «par magie» (en fait par l'aiguillage : si l'opérateur est l'atome «**+**» appliquons la procédure d'addition, etc. – tout défini dans le langage d'implantation, qui correspond à la machine de plus bas niveau. Pour que cet exercice soit vraiment utile, il faut tenir compte de quelques généralisations possibles et quelques problèmes d'implantation :

1. Il serait utile d'avoir des opérateurs d'arité quelconque, notamment les fonctions unaires comme **`sin`**, **`exp`**, etc. Ceci est trivial, il suffit de récupérer tous les arguments et évaluer récursivement toutes les branches avant d'appliquer l'opérateur. Mais il faudra les stocker quelque part.

2. Pour un langage de programmation sérieux il est indispensable de pouvoir exploiter les fonctions définies par l'utilisateur. Nous allons traiter cette question en détail plus tard, mais le modèle adapté à notre machine descendante est simple, il n'est rien d'autre qu'un modèle du calcul lambda implanté déjà dans lespremières réalisations du Lisp. Supposons avoir défini

   `(cube x)` $\equiv$ **`(* x (* x x)`**

   Quand l'interprète trouve la branche **`... (cube 1.5) ...`** sur l'arbre en train d'évaluation, il peut vérifier que **`cube`** ne correspond à aucun opérateur magique, il doit alors être défini par une construction de genre

   ```
   (define cube  (lambda (x) (* x (* x x))))   ou en Haskell :
   cube = \x -> x * x * x
   ```

   L'interprète doit trouver dans l'environnement l'affectation de l'opérateur **`cube`**, comme de toute autre variable. Le protocole à suivre est alors le suivant :

   - On récupère les paramètres de l'opérateur (ici : **`x`**).
   - L'argument (ou les arguments) de l'opérateur sont évalués comme dans le cas de l'opérateur primitif.
   - Les valeurs – résultats de cette évaluation sont associées avec les paramètres dans l'environnement actuel, qui doit donc être dynamique, modifiable.
   - La forme λ est évaluée comme toute autre expression. Les paramètres sont associés avec leurs valeurs, et la procédure est effective.
   - Le résultat est récupéré, et les associations des paramètres sont détruites.

3. L'interprète conceptuel est «trop intelligent» et il sera lent, partiellement à cause de résolution dynamique des types de données. Haskell partage avec C le typage statique.

Le type de données – feuilles – utilisé ici est l'*union* (au sens connu en C) de nombres flottants et de symboles. Pour simplicité les symboles seront des chaînes, et en Haskell ces chaînes sont des *listes* de caractères.

```
type String = [Char]      Cette définition est standard
data Value = F Double | S String
```

où les constructeurs **S** et **F** sont des balises (*tags*) identifiant les variantes de cette alternative. Un opérateur est également un symbole. L'expression est une arborescence, dont le nœud interne contient le *tag*, l'opérateur, et deux branches. (Si nous voulions généraliser à des opérateurs d'arité arbitraire, il faudrait remplacer les deux branches par une liste de branches. Le type **String** en Haskell est prédéfini.)

```
type Opsymb = String
data Expr = L Value | A Opsymb Expr Expr
```

La recherche des valeurs symboliques dans l'environnement peut être réalisée à l'aide d'un tableau associatif

```
envir = [("x",2.5),("y",-1.0),("z",0.5)]
```

etc., avec la fonction de recherche correspondante **assoc :: String -> Double** dont la construction est laissée au lecteur. Plus tard il nous faudra construire des tables de symboles plus réalistes. La magie des opérateurs primitifs peut être réalisée par la fonction d'aiguillage (adaptée aux opérateurs binaires)

```
evalpr  :: String -> Double -> Double -> Double
evalpr op x y =
  case op of
    "+"  ->  x+y
    "*"  ->  x*y
    "/"  ->  x/y
    "-"  ->  x-y
```

qui traite des valeurs déjà décodées. Voici la fonction qui traite des valeurs générales, cherchées éventuellement dans l'environnement global :

```
decodval   ::   Value -> Double
decodval (F x) = x
decodval (S c) = assoc c envir
```

(Ici les déclaration de type sont redondantes, mais peuvent pendant le développement du programme guider l'œil du programmeur, et éviter quelques fautes.) L'évaluateur récursif est très simple :

```
eval  :: Expr -> Double
eval (L val) = decodval val
eval (A op v1 v2) = evalpr op (eval v1) (eval v2)
```

Passons à une machine plus efficace, et transformons le schéma ci-dessus en *compilateur* qui génère un code linéaire, destiné à une machine à pile, presque «professionnelle». Si le lecteur est intéressé par la vraie forme de notre expression exemplaire en Haskell, la voici :

```
A "-" (A "-" (L(F 1.0)) (A "*" (L(F 2.0))
     (A "+" (A "*" (L(F 2.0)) (L(S "x")))
         (A "/" (L(F 3.0)) (L(S "y"))))))
            (A "*" (L(F 2.0)) (L(S "z")))
```

donc, la lisibilité de Haskell par rapport à Scheme est une propriété relative, elle s'applique aux programmes et non pas aux données balisées…

Voici la définition de la fonction de recherche **assoc**, la plus primitive possible, linéaire. Nous suggérons au lecteur *formellement* de construire une version arborescente, dichotomique, sachant que les chaînes peuvent être ordonnées. Bien sûr, on peut stocker sur un tel environnement aussi des opérateurs !

```
assoc  :: String -> [(String,Double)] -> Double

assoc _ [] = error "Pas d'association !"
assoc ch ((sy,v):q) | ch==sy = v
                    | otherwise = assoc ch q
```

Les machines récursives, arborescentes sont plus simples que les machines linéaires de bas niveau, présentées ci-dessous. Elles sont parfois utilisées, car on peut les implanter en quelques lignes de code et insérer dans une application quelconque, à condition que les expressions évaluées soient courtes (et que le noyau de l'application sache gérer les listes et les arbres, donc, s'il dispose des procédures d'allocation et dé-allocation de mémoire, et si la récursivité est bien implantée).

3.3 Linéarisation du code et machines à pile

La surcharge de l'interprète ci-dessus est évidente, la structure arborescente des expressions exige son allocation dans le tas, avec les pointeurs (ou «handles» : poignées), et en plus, l'évaluation récursive encombre la *pile système* : la pile qui appartient à la couche d'implantation. Les deux inefficacités disparaîtront maintenant. Nous allons transformer l'expression arborescente en code linéaire postfixe, où l'opérateur suit ses opérandes. Point besoin de parenthèses, notre expression exemplaire devient

1	2	2	x	*	3	y	/	+	*	-	2	z	*	-

La transformation de l'arbre en code postfixe est assez banale, il suffit de modifier très légèrement la fonction `eval`. Sa structure reste essentiellement la même, seulement au lieu d'évaluer le nœud, la fonction génère le code pour les branches (alors elle s'appelle récursivement quand même, mais nous allons optimiser cette récursivité), ensuite elle concatène les deux codes, et à la fin elle stocke l'opérateur derrière le code correspondant aux branches. Les feuilles sont *presque* directement insérées dans le code. Mais ce «presque» est important : *un nombre n'est pas une instruction.*

L'évaluation de ce code a besoin d'une autre machine virtuelle, de plus bas niveau, qui n'aura même pas besoin d'être récursive. L'interprète parcourt le code «de gauche à droite». Si l'instruction courante contient une donnée, sa valeur sera empilée sur une pile (privée) ; si c'est un opérateur, les deux (pour les opérateurs binaires ; n pour les opérateurs n-aires) dernières valeurs sur la pile seront dépilées, l'opérateur appliqué, et le résultat empilé de nouveau. Ici la suite d'instructions sera : empiler 1, 2, x, multiplier $2 \times x$, empiler 3 et y, exécuter la division, etc.

Le reste est le codage. Mais le code postfixe symbolisé ci-dessus est *hétérogène*, il contient en vrac les données et les opérateurs, et ceci n'est presque jamais une bonne idée, car demande de la part de la machine virtuelle un peut trop – elle doit discerner *dynamiquement* entre ces classes d'objets avant interpréter chaque item du code, et ceci ralentit considérablement l'exécution. Ceci dit, les langages typés dynamiquement occupent une niche stable dans le monde de la programmation, et personne n'envisage l'abandon du Scheme à cause de cela. Une machine virtuelle postfixe, à pile, avec le typage dynamique existe en millions d'exemplaires dans le monde : il s'agit de l'interprète du langage PostScript qui pilote les imprimantes laser. En PostScript on mélange les données et les opérateurs, et la machine prend dynamiquement la décision d'empiler la donnée si elle figure «nue» dans le programme.

Nous allons faire quelques exercices basés sur ce modèle, mais pour la compilation générale la différence entre les données et les commandes est trop importante pour la traiter avec désinvolture.

Profitons de ce changement de chevaux, et introduisons quelques généralisations et quelques contraintes dans la machinerie.

3.3.1 Cahier des charges

1. Le système doit gérer les opérateurs primitifs d'arité quelconque, et leur liste doit être extensible.

2. Il est possible d'utiliser des fonctions non-primitives, dont le corps a le même statut que l'expression principale (code utilisateur), et il n'est plus exécuté par magie.

3. Le code est homogène, nous éviterons de mélanger les données et les commandes, même si ceci en principe est faisable. Nous voulons ainsi accélerer l'exécution, en évitant trop de décisions dynamiques de décodage.

4. Le code est stocké dans une liste chaînée. Une allocation plus statique, dans un tableau serait plus efficace, mais plus rigide, moins propice à l'apprentissage des idées générales. Cette optimisation sera discutée plus tard, elle est assez simple.

5. Les variables n'ont plus de noms symboliques (chaînes) à décoder, mais sont identifiés aux *indices*, ou *références*, adressant une table des symboles (l'environnement), ou – éventuellement – la pile des données locales. Ici également on peut utiliser une liste, un arbre ou un tableau de hachage pour accélerer l'insertion et le parcours.

6. Toutes les données circulent par la *pile de données*, structurée également comme une liste. La pile est conceptuellement hétérogène, on y stockera des nombres, mais aussi des objets procéduraux, exécutables. Ceci sera **essentiel** pour pouvoir gérer les structures de contrôle. Bien sûr, la pile syntaxiquement *est* homogène, tous les objets sont des données, mais définies avec variantes (balises).

7. La règle précédente suggère effectivement la solution du problème d'arité : *tout* opérateur prend en argument la pile, et retourne la pile modifié. S'il a besoin de dépiler 134 arguments, il s'en charge. Il peut aussi renvoyer au module appelant une valeur multiple, empilée. Ce problème est plus délicat avec une machine à registres fixes, car il faut les sauvegarder lors des appels.

8. Un objet exécutable de haut niveau est tout simplement une liste représentant son code. (Mais il faut que le consommateur d'un tel objet *sache* qu'il s'agisse d'un code ; ceci peut demander la présence d'une balise identifiante).

9. Pour éviter la nécessité de distinguer les listes-code vides des autres (et vérifier cela avant de décoder l'instruction suivante, ce qui décélère l'exécution), *aucune* liste ne sera vide. Tout code contiendra au moins une instruction spéciale, le **retour**, et cette instruction devra terminer tout programme (module). L'avantage est qu'une telle instruction peut se trouver au milieu d'un code, et précipiter sa terminaison : ceci permettra de concaténer plusieurs fonctions et blocs conditionnels ensemble.

Donc, les items du programme peuvent être des opérateurs primitifs, ou des opérateurs composites – objets exécutables de haut niveau qui remplacent les constructions λ du modèle précédent. Toutes les constantes et variables – les données présentes explicitement dans le code doivent se transmuter en opérations. En effet, elles seront remplacées par les *instructions d'empilement* de ces données.

Regardez la définition de la fonction **`evalpr`** ci-dessus. C'est un **switch** qui décode les opérateurs primitifs. Une question se pose : et si nous voulions ajouter d'autres opérateurs? Il nous faudrait recompiler cette partie de la machine virtuelle. Quand le nombre d'opérateurs est variable ou dépasse quelques dizaines, une telle stratégie est mauvaise. Nous pouvons naturellement construire un tableau associatif entre les noms et les exécutables, et utiliser une variante de la fonction **`assoc`**, mais ceci n'est pas une idée brillante, car l'exécution sera ralentie. Mais la machine virtuelle **n'a pas besoin de noms** des opérateurs (sauf pour le déboguage). La vitesse d'exécution augmente d'un facteur important (de plusieurs dizaines s'il fallait décoder les chaînes) si le code interprété ne contient plus des noms, mais directement les **références aux procédures exécutables**. Les noms des opérateurs sont décodés (transformés en indices) par le compilateur et non pas par l'interprète.

Très souvent les interprètes utilisent une stratégie intermédiaire : les *bytecodes*. Pas de noms symboliques, mais tout opérateur est représenté par quelques bits – un nombre entier, ou un symbole appartenant à un type énuméré, qui sert d'indice à un tableau d'aiguillage. Pas de recherche, pas de fonction **`assoc`**, mais l'indexage, précédé éventuellement par la séparation du bytecode du reste de l'instruction. Nous allons cependant exploiter encore une autre approche, utilisé dans FORTH, quelques variantes du langage Snobol4, Python, partiellement incluse dans le compilateur de Glasgow Haskell, et plusieurs autres langages, et pourtant rarement enseigné – le code «enfilé» (*threaded code*). Il n'y aura pas de bytecodes, mais le stockage direct des références aux objets exécutables, ou des références aux «boîtes» qui contiennent les références aux exécutables (*indirect threaded code*). Son *seul* désavantage est que la longueur du code augmente, car les bytecodes sont plus courts que les références (pointeurs).

Le nom «*threaded code*» est souvent utilisé dans un contexte plus spécifique : c'est un code qui n'a pas besoin d'une machine virtuelle globale (boucle centrale), puisque chaque opérateur connaît son successeur (ou successeurs dans le cas des conditionnelles). Nous allons adopter cette terminologie, et nous allons présenter les deux modèles d'interprétation : l'un où le flot de contrôle est piloté par une boucle centrale, et l'autre – *threaded code*, sans boucle, ***qui réalise une variante particulière de la philosophie CPS : ces paramètres supplémentaires sont des continuations***.

***Credo* religieux no. 4 :** Une machine virtuelle trop intelligente est une calamité désastreuse, comme un soldat qui pense trop. Les deux doivent exécuter des ordres simples sans réfléchir, sinon on gaspille le temps, et c'est la concurrence qui gagne la guerre.

***Credo* religieux no. 5 :** Une machine virtuelle stupide est bonne pendant la guerre. Mais quand la paix arrive et il faut développer la société, les soldats sans cervelles ne servent pas à grande chose.

3.3.2 Codage de la machine

Commençons par la spécification des données. Avant, les variables étaient stockées dans une liste, **`x`** était associé avec 2.5, etc. Les variables locales auront toujours besoin d'une structure dynamique (pile) pour stocker les valeurs, mais ici, *à titre pédagogique* imaginons que les variables sont globales et statiques. Un tableau **`envir`** peut remplacer la liste associative, et les noms ne sont plus que des indices. Le tableau en Haskell peut être construit par la fonction **`array`** qui prend deux arguments – une paire spécifiant la dimension (l'intervalle des indices), et une liste des associations : (indice,valeur).

```
import Array

envir = array (0,20) [(0,2.5), (3,-1.0), (4,0.5), ... ]
```

(Les fonctions sur des tableaux en Hugs sont importées optionnellement, d'où le mot-clé **`import`**.) Ces fonctions résident dans une librairie standard, mais qui pour des raisons d'efficacité n'est pas chargée automatiquement dans la mémoire.

Répétons que la transformation des noms de variables en indices n'est pas une tâche de la machine virtuelle, mais du compilateur, et ceci n'est pas actuellement notre problème.

Une donnée appartient au type **`Value`** :

```
type Indx = Int
data Value = F Double | I Integer | Ch Char | S String
                      | U | V Indx | Co Code
```

où nous avons enrichi la collection par les entiers, les chaînes, etc. La différence entre **`Integer`** et **`Int`** est secondaire, Haskell reconnaît les entiers très longs, de précision illimité, mais les indices des variables (**`V`**) seront des entiers normaux, de 4 octets appartenant au type **`Int`**. (Au cas où quelqu'un décide de modifier le protocole d'accès, nous avons introduit un synonyme des entiers standard : **`Indx`**.) Le type **`Code`** représente les fonctions utilisateur.

Ajoutons à cette collection encore un type «bidon» **`U`** qui représente une valeur «vide» (En Haskell il existe un type prédéfini pour cela : **`()`**, son usage est moins ambigu qu'un *symbole* **`nil`** ou la liste vide renvoyée par des procédures Lisp qui n'ont pas besoin de générer une valeur concrète).

Rappelons que l'affichage des objets complexes demande un peu d'attention de la part de l'utilisateur. Si nous voulons pouvoir afficher une valeur quelconque, il faut prévoir un ensemble de fonctions de conversion, par exemple :

```
instance Show Value where
  showsPrec p vl =
    case vl of F  x -> showString "Float: " . shows x
               I  x -> showString "Integer: " . shows x
               Ch x -> shows x
               S  x -> shows x
               U    -> showString "<rien!>"
               V  x -> showString "Variable: " . shows x
               Co x -> showString "[<code>]"
```

Passons au code qui est une liste d'items, et à la pile qui est une liste de valeurs. Les items du code peuvent être : RET, une opération sans arguments (mais extra arguments existent, la pile est *toujours* présente) ou une opération avec un argument supplémentaire, par exemple un code à exécuter, ou une valeur à empiler. La commande RET est introduite ici uniquement pour la présentation, pour varier un peule thème, nous allons l'abandonner vite.

```
type Code = [CodeItem]
type Stack = [Value]

type Op0 = Stack -> Stack            -- op. sans arg
```

```
type Op1 = Value -> Stack  -> Stack  -- 1 arg supp.

data CodeItem = RET | C0 Op0 | C1 Op1 Value
```

Il faut définir quelques opérations arithmétiques primitives, pour simplicité on définit seulement nos vieux opérateurs binaires. Définissons également une fonction qui empile une valeur, constante ou variable. Dans ce dernier cas il faut décoder la variable, utiliser l'environnement. La construction **`tbl!indx`** correspond à **`tbl[indx]`** dans d'autres langages.

```
type Opbin = Double -> Double -> Double

binop   :: Opbin -> Stack -> Stack
binop op ((F x):(F y):q) = F (op y x):q
add = binop (+)  :: Op0
mul = binop (*)
sub = binop (-)
dvd = binop (/)

load v stack = case v of
 F x  -> v:stack
 V i  -> (F (envir!i)):stack
```

Le code postfixe de notre expression exemplaire reste assez illisible :

```
cod = [C1 load (F 1.0), C1 load (F 2.0), C1 load (F 2.0),
       C1 load (V 0), C0 mul, C1 load (F 3.0), C1 load (V 3),
       C0 dvd, C0 add, C0 mul, C0 sub, C1 load (F 2.0),
       C1 load (V 4), C0 mul, C0 sub, RET]
```

mais la machine virtuelle devient très simple :

```
aeval (instr:rst) stack = case instr of
  RET       ->  stack
  C0 op     ->  aeval rst (op stack)
  C1 op val ->  aeval rst (op val stack)
```

et pour obtenir le résultat (qui est égal à -4.0) on construit, et on demande la valeur de

```
interp cod = xx where (xx:_) = aeval cod []
```

Nous avons promis de gérer les fonctions utilisateur, par exemple la fonction qui calcule le cube d'un nombre. Le contexte d'appel est simple, l'argument est empilé *avant* de lancer la fonction. Elle n'a donc besoin que de la pile. Mais il faut dupliquer la valeur deux fois pour pouvoir itérer deux fois la multiplication. Construisons donc quelques opérations primitives qui administrent la pile, et la fonction **`exec`** qui exécute un code utilisateur. Cette dernière est assez triviale : elle relance récursivement l'évaluateur.

```
dup p@(x:_) = x:p          -- duplique le sommet
exch (x:y:q) = (y:x:q)     -- échange deux dernières valeurs
pop (_:q) = q              -- détruit le sommet
exec (Co cod) stack = aeval cod stack  -- The Executioner

-- Le code du cube :
cube = Co [C0 dup, C0 dup, C0 mul, C0 mul, RET]

-- et le résultat :
res2 = interp [C1 load (F 5.0), C1 exec cube, RET]
```

est 125.0. Nous n'avons *rien* modifié dans la définition de la machine. L'usage de la pile pour passer les paramètres nous a libéré de la nécessité de stocker dynamiquement dans l'environnement les associations entre les paramètres et les arguments actuels. Mais si la construction du code correspondant à **`cube`** n'est plus manuelle, mais automatique, à partir d'une procédure avec paramètres, ce problème va resurgir, et le compilateur doit établir ces associations *avant* de construir le code final.

3.3.3 Mécanismes décisionnels

Notre machine est handicapée pour quatre raisons.

1. Le code exécuté est obligatoirement linéaire et déterminé statiquement *a priori*, la machine ne peut prendre aucune *décision*. Dans un modèle sérieux il faut augmenter la puissance sémantique de la machine par – au moins – l'équivalent de la construction **if-then-else**.

2. L'interprète reste trop complexe. Il est récursif, alors il utilise la pile *système*, présente dans l'application grâce au compilateur du langage d'implantation (Haskell, ou C...). Ceci n'est pas une bonne idée. La pile système n'étant pas contrôlée par le programme, le déboguage devient impossible, inefficace, ou tordu.

3. En principe toutes les structures de contrôle itératives (boucles) peuvent être réalisées par la récursivité terminale, mais nous sommes encore loin d'une telle optimisation. Toute tentative d'écrire une fonction récursive, terminale ou pas, est interprété de la même façon – appel **récursif** de l'évaluateur. Les boucles suffisamment longues se terminent toujours par le débordement de la pile système.

4. Le quatrième point est simple mais gênant : pour définir les fonctions récursives il faut pouvoir leur donner des noms. Il faut alors élargir le concept de variables – pas seulement numériques, mais aussi fonctionnels. (Et, bien sûr, d'autres objets – tableaux, listes, etc. aussi, mais cela viendra plus tard.)

Ajouterons ces modifications en douceur. Les procédures constituent des valeurs et peuvent être empilées. Construisons alors deux fonctions : **`iff`** de classe **`C0`** qui trouve sur la pile la condition et une procédure, qui sera exécuté si la condition est différente de zéro. La fonction **`ifelse`** (de la même classe **`C0`**, sans arguments extra) trouve sur la pile *deux* procédures – une si la condition est remplie, et l'autre en cas d'échec.

Faut-il introduire les Booléens? En principe oui, Haskell les utilise et ils n'alourdissent pas la structure de la machine. Mais nous pouvons aussi adopter le style du C, où le zéro est faux, et 1 (ou autre nombre différent de zéro) – vrai. Il faudra alors construire quelques opérateurs relationnels.

```
ifelse (pelse:pthen:(I cnd):q) =
  if cnd/=0 then exec pthen q else exec pelse q

iff (pthen:(I cnd):q) =
  if cnd/=0 then exec pthen q else q

biconvrt True  = I 1     -- Convertisseurs
biconvrt False = I 0

boolop  op ((F x):(F y):q) = biconvrt (op y x) : q

gt = boolop (>)
lt = boolop (<)
eq = boolop (==)
ne = boolop (/=)
```

La fonction **`msign`** qui calcule le signe d'un nombre entier : 1, 0 ou -1 aura la forme

```
msign = Co [C0 dup, C1 load (F 0.0), C0 gt,
            C1 load (Co [C0 pop, C1 load (F 1.0), RET]),
            C1 load (Co [C1 load (F 0.0), C0 eq,
                C1 load (Co [C1 load (F 0.0), RET]),
                C1 load (Co [C1 load (F (-1.0)),RET]),
                C0 ifelse, RET]),
            C0 ifelse, RET]
```

Un tel code, plein de balises et de constantes procéduralisées par l'empilement n'est pas écrit d'habitude par les humains. Le modèle, comme il a été dit, correspond au PostScript ou autres langages qui utilisent les machines à pile, comme Java, Smalltalk, quelques implantations du Pascal, etc. Mais le code est généré par le compilateur, et même en PostScript qui est une machine à pile «nue», programmée dans la notation postixe, demande seulement que l'utilisateur écrive

```
{dup 0 gt {pop 1}
          {0 eq {0} {-1} ifelse} ifelse}
```

L'ajout des tags, du RET, et des **load** est trivial, la seule vraie différence est le typage des nombres – PostScript reconnaît les nombres entiers et flottants dynamiquement. En C ou en Pascal la conversion est automatique. En Haskell elle est semi-automatique. Pour l'instant notre machine les considère comme des types complètement distincts. Laissons au lecteur la construction des opérateurs arithmétiques **iadd**, **imul**, etc., des relations arithmétiques **igt**, **ieq**, et éventuellement quelques autres «clones» entiers des opérations flottantes. En tout cas, grâce au balisage, la différence entre les nombres de types différents est explicite, visible, et on peut soit compiler l'opération adéquate, ou laisser la décision à la machine virtuelle.

Passons à la construction des fonctions récursives. D'abord, il faut rendre l'environnement plus souple. Définissons aussi un opérateur un peu plus général que **Op1**, et dont le premier argument est l'environnement. Ceci nous oblige à introduire une nouvelle classe d'opérateurs :

```
type Env = Array Int Value
type Op2 = Env -> Value -> Stack -> Stack

data CodeItem = RET | C0 Op0 | C1 Op1 Value | C2 Op2 Env Value
```

Ensuite il faut modifier un peu la fonction qui empile une valeur. La fonction **load** sera généralisée, mais simplifiée. Elle accepte un nouveau paramètre : **symtab**, le dictionnaire des symboles qui est notre tableau associatif. Si le second argument est une variable, la valeur correspondante est empilée, sinon la fonction empile l'objet même.

```
envir = array (0,3) [(0,F 2.5), (1,F (-1.0)), (2, F 0.5),(3, I 0)]
...
loadv symtab (V i) stack = (symtab!i) : stack
loadv _      v     stack = v : stack

...
envirn = envir // [(3,fac)]
loadn = loadv envirn
```

L'opérateur **(//)** accepte un tableau et une liste d'associations, et renvoie le tableau modifié par les associations. Ici l'environnement a été enrichi par un objet qui s'appelle **fac**, et qui – comme le lecteur soupçonne – est la référence de la fonction factorielle. Pour définir la fonction **fac** nous aurons besoin d'un exécuteur primitif modifié – de la fonction **execv** qui accepte aussi l'environnement.

```
execv env (V i) = exec (env!i)

fac = Co [C0 dup, C1 loadn (I 0), C0 ieq,
         C1 loadn (Co [C0 pop, C1 loadn (I 1), RET]),
         C1 loadn (Co [C0 dup, C1 loadn (I 1), C0 isub,
                       C2 execv envirn fact, C0 imul, RET]),
         C0 ifelse, RET]

fact = V 3
```

La dernière ligne boucle la liaison entre la définition du code et l'environnement. La liste **fac** n'est pas autoréférentielle, elle s'adresse par l'intermédiaire de l'élément 3 du tableau **envirn**. Le résultat de l'exécution de

```
interp [C1 loadn (I 6),C2 execv envirn fact, RET]
```

donne 720. La solution proposée n'est pas idéale. Les définitions des fonctions comme **loadn** doivent être locales, la construction **C2** est redondante, on peut définir localement des applications partielles **(execv envirn)** et éliminer la présence explicite de l'environnement dans la liste-code, mais ce sont des opérations cosmétiques.

En fait, il faut avouer que l'introduction de plusieurs classes d'opérations, notamment de **C1** qui contient un paramètre extra, ne constitue aucune nécessité. Nous l'avons fait uniquement pour pouvoir lier de manière «classique» une fonction et une donnée extra dont la fonction a besoin. Pour les programmeurs fonctionnels affranchis il suffirait de définir une fermeture appropriée, mais nous soulignons – encore une fois – que nous ne voulons pas exagérer avec les spécificités de la programmation fonctionnelle.

Rappelons encore qu'un programmeur en PostScript écrira

```
/fac {dup 0 eq {pop 1} {dup 1 sub fac mul} ifelse} def
```

pour définir la factorielle.

3.4 Gestion explicite de la pile des retours

Passons à l'élimination de la récursivité de l'évaluateur, ce qui mérite une section séparée. Nous pouvons laisser sans remords les appels récursifs terminaux de la boucle principale. Ce qui nous gêne est la fonction **`exec`** et ses variantes.

Nous avons besoin de la récursivité (empilement des adresses de retour), car quand **`aeval`** trouve un code interprété interne, il doit revenir au contexte précédent après le retour de la fonction **`exec`**. Mais nous pouvons gérer la pile des retours par la machine elle même. L'interprète aura un paramètre supplémentaire, une liste dont les éléments sont les codes à exécuter. Au début n'a qu'un seul code, l'«expression principale», ou le programme principal.

On peut accepter un protocole d'exécution «système», où la machine virtuelle ne s'arrête jamais, et ne rend aucune valeur. Elle joue alors le rôle d'un «dispatcher» (aiguilleur), d'un système d'exploitation qui envoie les tâches a ses processus-esclaves, mais qui ne fait rien d'autre. Si l'utilisateur veut dialoguer avec son programme, lire les résultats partiels, etc., tout doit être prévu par son code particulier (et les fonctions prédéfinies). Une telle convention n'est pas fonctionnelle, mais impérative par excellence. Elle est utilisée dans la pratique. Nous suivrons néanmoins une stratégie différente, la machine s'arrête et retourne le résultat d'évaluation du programme principal quand elle exécute l'instruction **`stop`**. Nous pouvons insérer ce code par défaut sur la pile des retours, mais l'essentiel est d'assurer que la machine ait *toujours* une instruction à exécuter. On sait très bien que la processeur matériel une fois mis en marche fait toujours quelque chose, il ne s'arrête jamais, même si le programme constitue une boucle morte. L'avion qui s'arrête en plein vol n'est plus un avion. . .

Comme précédemment, la mise à jour de notre machine nous permettra de réfléchir et d'enrichir sa sémantique, en simplifiant en même temps sa structure. Les stratagèmes principaux exploités ici sont les suivants.

- Il n'y aura plus d'opérateur reconnu spécialement par sa syntaxe : RET. Il est devenu un opérateur normal, comme les autres, et comme l'opérateur **`stop`**. Son rôle est de reprendre le calcul à partir de la pile des retours.
- *Tout* opérateur prend maintenant l'environnement et une valeur supplémentaire comme arguments, mais ces arguments peuvent ne pas être utilisés du tout. (Dans un langage paresseux ceci ne force pas leur évaluation, et ne coûte presque rien.) Ceci simplifie les classes des opérateurs.
- L'environnement est passé à la machine comme une variable globale, et tout **`CodeItem`** contient une donnée supplémentaire (**`U`**). On pourra optimiser cela aisément plus tard, mais ainsi la structure du code est plus régulière.
- Tout opérateur prend la pile des données, le reste du code actuel (la queue de la liste dont la tête est l'instruction exécutée) et la pile des retours comme arguments.
- Le résultat retourné est toujours un triplet : le code (instruction suivante) à exécuter, la nouvelle pile des retours, et la nouvelle pile des données. (Et on voit déjà que la généralisation suivante doit permettre également retourner un nouvel environnement dynamique, si la sémantique du langage permet p. exemple la réaffctation des variables globales).
- La machine «extérieure» est une boucle qui s'arrête quand le code à exécuter est vide. Cette liste vide est renvoyée par l'opérateur **`stop`**. On peut – bien sûr – optimiser ceci, comme il a déjà été signalé, et cet exercice est laissé au lecteur.
- L'opérateur **`ret`** ignore le code restant, et reprend le nouveau code de la pile des retours.
- L'opérateur **`exec`** empile le code restant sur la pile des retours, et assigne le nouveau code depuis la procédure de haut niveau qui sera exécutée.

Nous introduirons accessoirement quelques simplifications de notation, des opérateurs **`cnst`** et **`var`** qui empilent une constante ou une variable. Le nouveau programme sera copié dans son intégralité pour faciliter la lecture.

```
import Array

type Indx = Int
data Value = F Double | I Integer | Ch Char | S String
                      | U | V Indx | Co Code

type Code = [CodeItem]
type Stack = [Value]
type Env = Array Int Value
type Rstack = [Code]

type Op =
 Env -> Value -> Code -> Rstack -> Stack -> (Code, Rstack, Stack)
data CodeItem = O Op Value
```

Notez la simplicité du code et l'introduction du **`Rstack`**. Voici l'interprète principal complet et quelques opérateurs primitifs.

```
interp env (instr:code) = machine instr (code:[[stop]]) []
 where
 machine   ::  CodeItem -> Rstack -> Stack -> Stack
 machine (O op val) (rest:later) pile =
  let (ncode, nlater, npile) = op env val rest later pile in
  case ncode of
   []             -> npile        -- "stop" a été exécuté
   (ninstr:nrest) -> machine ninstr (nrest : nlater) npile

stop = C (\_ _ _ rt pile -> ([], rt, pile)) U
ret  = C (\_ _ _ (rt:demain) pile -> (rt, demain, pile)) U

-- Constante générique
cnst a x = O (\_ z rst later pile ->
              (rst, later, (z:pile))) (a x)

-- et ses variantes: flottante et entière.
dblc x = cnst F x
intc x = cnst I x

-- Empilement d'une variable (décodée)
var  v = O (\e z rst later pile ->
              (rst, later, ((e!v):pile))) (V v)
```

La première partie de l'exercice demande seulement la construction des opérateurs binaires (dans l'arithmétique flottante)

```
binop op =
 O (\_ _ rst later (F x:F y:q) -> (rst,later,(F (op y x):q))) U

add = binop (+)
mul = binop (*)
sub = binop (-)
dvd = binop (/)

envir = array (0,3) [(0,F 2.5), (1,F (-1.0)), (2, F 0.5),(3, I 0)]
cod = [dblc 1.0, dblc 2.0, dblc 2.0, var 0, mul, dblc 3.0,
       var 1, dvd, add, mul, sub, dblc 2.0, var 2, mul, sub, ret]

res1=interp envir cod
```

Passons aux procédures utilisateur et aux mécanismes décisionnels (**`Ifelse`**). Voici la définition du **`cube`** plus quelques fonctions accessoires.

```
dup = O (\_ _ rst later p@(x:q) -> (rst,later,x:p)) U
```

```
pop = O (\_ _ rst later (_:q) -> (rst,later,q)) U
exch = O (\_ _ rst later (x:y:q) -> (rst,later,y:x:q)) U

exec cod = O (\_ (Co pr) rst later pile ->
   (pr,rst:later,pile)) (Co cod)

cube = exec [dup, dup, mul, mul, ret]
res2=interp envir [dblc 5.0, cube, ret]   -- oui, cela donne 125
```

Notre code comme **[dblc 5.0, cube, ret]** contient seulement les structures de données (**CodeItem**), on ne voit ni fonctions, ni la pile des données ni la pile des retours. Mais le code est strictement fonctionnel, et en plus très facilement traduisible en code impératif. Avec **ifelse** il y a un petit problème ! Il serait commode – comme dans la version précédente – d'utiliser **exec** aiguillé par le conditionnel de plus bas niveau (**if-then-else**) de Haskell. Mais à présent les primitifs prennent en plus de la pile aussi le reste du code et la pile des retours. Comment les passer à l'autre primitive, sachant qu'**exec** *n'est pas une fonction* opérant sur la pile, mais un générateur de structures de données? Nous avons choisi une solution banale, mais les exercices discutent d'autres possibilités.

```
ifelse = C (\_ _ rst later ((Co pelse):(Co pthen):(I cnd):q) ->
  ((if cnd/=0 then pthen else pelse),rst:later,q)) U

-- les fonctions suivantes ne changent pas
biconvrt True  = I 1
biconvrt False = I 0
-- modification assez triviale
boolop  op =
 O (\_ _ rst later ((F x):(F y):q) ->
   (rst,later,(biconvrt (op y x):q))) U

gt = boolop (>)
lt = boolop (<)
eq = boolop (==)

proc x = cnst Co x         -- Comment empiler une procédure

msign = exec [dup, dblc 0.0, gt,     -- Signe d'un nombre
         proc [pop, dblc 1.0, ret],
         proc [dup, dblc 0.0, eq,
               proc [pop, dblc 0.0, ret],
               proc [pop, dblc (-1.0),ret],
               ifelse, ret],
         ifelse, ret]
```

Et finalement la factorielle :

```
-- Ops binaires entiers. Aucune élégance...
binop op =
 O (\_ _ rst later (I x:I y:q) -> (rst,later,(I (op y x):q))) U

addi = binop (+)
muli = binop (*)
subi = binop (-)
-- Ops Booléens (relationnels) entiers
booliop  op =
 O (\_ _ rst later ((I x):(I y):q) ->
   (rst,later,(biconvrt (op y x):q))) U

gti = booliop (>)
lti = booliop (<)
eqi = booliop (==)

-- Variante d'exec : exécution précédée par le décodage
```

```
execv n = O (\e (V i) rst later pile ->
    let (Co cod) = e!i in (cod,rst:later,pile)) (V n)

-- L'indice de la factorielle
fact = 3
envirn = envir // [(3,fac)]


fac = Co [dup, intc 0, eqi,
          proc [pop, intc 1, ret],
          proc [dup, intc 1, subi,
             execv 3, muli, ret],
          ifelse, ret]

-- test:
res6 = interp envirn [intc 6,execv fact, ret]
```

Notons que la syntaxe de Haskell est suffisamment souple pour qu'on puisse aisément vérifier la machine pendant sa construction, à condition de définir quelques abréviations. En C ceci n'est pas si simple...

3.4.1 Omission importante

Nous n'avons pas traité les affectations, ni les définitions des fonctions utilisateur *dans le programme* (ce qui peut être la même chose : l'association entre les noms et les objets). Pour le faire il faut faire des modifications suivantes :

- Ne pas passer l'environnement à la **`machine`** comme une variable globale, car elle risque de subir des modifications, mais comme un argument.
- Les opérateurs doivent également retourner l'environnement. Ceci fait déjà quatre arguments, ce qui détériore la lisibilité de la solution (mais on peut les emballer dans un record).
- L'opérateur **`(=)`** accepte une variable et une expression. On empile l'expression et on l'évalue, mais ensuite il faut *empiler l'adresse de la variable*, sans la décoder. Ceci est facile, on peut définir l'opérateur **`vaddr n = cnst V n`**.
- L'exécution de l'opérateur d'affectation dépile l'adresse de la variable et la valeur, et lance l'opération primitive **`(//)`** qui change le tableau d'associations. Le nouveau environnement remplace le précédent, la pile des données reste intacte.

3.4.2 Conseils pour les irrécupérables

Cette section est destinée aux lecteurs qui voudraient implanter une petite machine virtuelle selon notre modèle dans un langage impératif classique, comme C++. Les différences par rapport à Haskell sont les suivantes.

- On n'est pas obligé de respecter le protocole fonctionnel. En particulier la pile des données peut être une structure **globale**, et sa gestion peut utiliser les procédures séparées d'empilement et de dépilement. Ces opérations *modifieront* la variable globale.
- La même chose avec l'environnement qui ne sera pas seulement un tableau global (comme ici), mais qui peut être arbitrairement modifié par les modules de la machine (ce qui d'ailleurs sera le cas en présence des affectations).
- La pile et le code seront plutôt des tableaux que des listes. Le code est parcouru par une boucle **for** ou **while**, et les piles sont gérées par des indices spéciaux, qui adressent les sommets.
- Il faudra se débrouiller pour insérer dans le code les *pointeurs sur les fonctions* en C ou C++, d'établir un pont entre la machine virtuelle crée, et la couche sous-jacente, magique. Cette technique fait partie du cours du langage C, ou du cours de génie logicielle, mais ne sera pas traitée ici.

- Un tel programme en C++ peut et *doit* utiliser les techniques orientées-objet, en particulier
  - la surcharge des opérateurs arithmétiques doit être réalisée par les méthodes (fonctions génériques) correspondantes ;
  - les procédures polymorphes – vraiment polymorphes et non pas surchargées, comme la procédure d'empilement d'une donnée, doivent être définies dans une super-classe de toutes les données empilables.

3.5 Variante : *Indirect threaded code*

Montrons encore un autre modèle de la machine virtuelle … sans machine virtuelle. Jusqu'à présent nous n'avons pas touché la structure globale de l'interprète : il était toujours une boucle qui récupérait la nouvelle adresse (morceau de code) à exécuter, fourni par l'opérateur qui vient de terminer son travail. Rappelons que dans la première proposition, la machine «incrémentait le compteur» (passait à la queue de la liste avec le code) elle même, ce qui était trop rigide : les conditionnelles, les appels et les boucles demandaient un peu plus de souplesse.

Mais si à présent l'opérateur local (empilement, addition, etc.,) trouve son code successeur, pourquoi retourner au niveau de la machine uniquement pour ensuite passer la main à ce successeur? L'opérateur peut lui-même appeller son successeur par l'appel terminal. ***Ceci constitue la réalisation de bas niveau du concept des continuations, déjà mentionné***, et est une variante du «code enfilé». (l'attribut *indirect* résulte du fait que le code ne contient pas directement les opérateurs (pointeurs), mais des structures qui contiennent ces opérateurs). Voici, encore une fois, la machine complète. Elle est un peu différente de son prédecesseur. Pour simplicité, l'environnement global est absent, sa présence n'apporte rien de pédagogique.

La pile des retours peut être une simple liste, mais pour varier un peu, définissons une «liste privée», une structure linéaire construite par un opérateur **`(:>)`** défini par nous, avec un constructeur **`Empty`** qui remplace la liste vide. Voici la définition des valeurs, où nous avons ajouté aussi des listes (**`L [...]`**), et les vrais Booléens :

```
data Value = I Integer | F Double | S String | B Bool | U | Ch Char
           | L [Value] | V Int | C Code

type Dstack = [Value]                 La pile des valeurs
type Operator = Code -> Dstack -> Rtstack -> Value

data CodeItem = Op Operator Value
type Code = [CodeItem]

infixr 5  :>
data Rtstack = Empty | Code :> Rtstack          La pile des retours
```

Notez qu'un opérateur prend 3 arguments : le *code* dans lequel il se trouve, et les deux piles. Ce code sert uniquement à trouver le successeur de l'opérateur. L'opérateur renvoie une valeur comme son résultat (le sommet de la pile des données). Le code est composé de **`Codeitems`** qui sont des records possédant un opérateur et *toujours* une valeur extra, souvent **`U`** (et rappelons ici qu'une machine fonctionnelle aurait utilisée des fermetures assemblées par le compilateur).

La «machine» maintenant ne fait pratiquement rien, seulement initialise les piles. La fonction **`exec`** passe la main au premier opérateur présent dans le code. (Cette fonction n'est pas un opérateur utilisateur. L'opérateur utilisateur qui utilise directement **`exec`** s'appelera **`goto`**, et réalisera l'appel terminal).

```
interp code = exec code [] Empty

exec code@(Op op _ : _) = op code
  -- en fait: exec code pile retpile = op code pile retpile
```

La machine s'arrête en exécutant l'instruction **`stop`**. Voici sa définition, ainsi que la définition du branchement, l'instruction **`goto`** dont l'argument est une liste représentant le code :

```
stop = Op stfun U where
  stfun _ (v:_) _ = v
goto proc = Op (\(Op _ (C prc) : _) -> exec prc)
               proc
```

Attention : la fonction **goto** définie en Haskell *construit* le **CodeItem** correspondant, la structure **Op** dont le premier champ est une fonction anonyme qui lance **exec**, et le second – la procédure utilisateur qui sera exécutée.

Les définitions des opérateurs d'empilement, arithmétiques, et les autres deviennent maintenant plus complexes qu'auparavant, puisque chaque opérateur est obligé de localiser (ou construire) son successeur. Comme avant, les définitions en Haskell génèrent les **Op**-structures correspondantes. Commençons par les procédures d'empilement :

```
loadc v = Op ldfun v where
  ldfun (Op _ x : nxcode@(Op nxop _ : _)) p
     = nxop nxcode (x:p)

ldi n = loadc (I n)
ldf x = loadc (F x)
ldl l = loadc (L l)
ldcod cod = loadc (C cod)
```

etc. La déstructuration du premier argument de l'opérateur – le programme, sera souvent la même :

```
Op _ x : nxcode@(Op nxop _ : _)
```

ce qui peut être lu comme suit :

- l'argument anonyme qui suit **Op** est l'opérateur lui même (et donc, il n'a pas besoin de le spécifier);
- **x** est son paramètre extra ;
- **nxcode** est le code successeur, dont **nxop** est le premier opérateur.

les opérations typiques sur la pile : **dup**, **exch**, etc. possèdent toutes la même structure : la manipulation de la pile des données, et la construction du successeur. Nous pouvons faire une petite abstraction, et paramétrer nos opérations par leur «noyau», la fonction qui manipule la pile des données, et qui ne fait rien d'autre. Cette fonction : **action** parametrisera le manipulateur générique de la pile – **stackop**.

```
stackop action = Op (actfun action) U where
  actfun act (_ : nxcode@(Op nxop _ : _)) pile
  = nxop nxcode (act pile)

dup  = stackop (\p@(x:_) -> x:p)
pop  = stackop (\(_:q) -> q)            -- drop est reservé !
exch = stackop (\(x:y:q) -> y:x:q)
rot = stackop (\(x:y:z:q) -> y:z:x:q)
indx = stackop (\(I n : p) -> p!!(fromInteger n) : p)
under = stackop (\(x:y:q) -> y:x:y:q)
```

Rappelons que la notation **liste!!n** récupère le n-ième élément d'une liste : $n = 0$ récupère la tête. L'opérateur **indx** copie le n-ième élément de la pile sur son sommet. L'opérateur **rot** effectue la transformation **[x,y,z,...]** $\rightarrow$ **[y,z,x,...]**, etc. L'opérateur **under** est équivalent à **ldi 1, indx**, ou à **exch,dup,rot**.

Voici quelques opérateurs «standard» binaires, et unaires :

```
unop action = Op (actfun action) U where
  actfun act (_: nxcode@(Op nxop _ : _))
             (x : p) = nxop nxcode (act x : p)

expop = unop (\(F x) -> F (exp x))
sqrtop = unop (\(F x) -> F (sqrt x))     -- etc.
```

```
binop action = Op (actfun action) U where
  actfun act (_: nxcode@(Op nxop _ : _))
             (x : y : p) = nxop nxcode (act y x : p)

addi = binop (\(I x) (I y) -> I (x+y))
muli = binop (\(I x) (I y) -> I (x*y))
subi = binop (\(I x) (I y) -> I (x-y))
```

Nous avons défini déjà l'appel terminal. Voici un appel quelconque, et le retour. N'oublions pas que le retour est un opérateur normal, qui attend son successeur. La machine ne s'arrête pas.

```
call proc = Op callfun proc where
  callfun  (Op _ (C prc) : nxcode) pile rtpile
   = exec prc pile (nxcode :> rtpile)
ret = Op retfun U where
  retfun _ pile (code :> rtpile) = exec code pile rtpile
```

Finalement, passons aux mécanismes décisionnels. Définissons quelques relations arithmétiques (ceci est trivial), et les opérateurs **`iff`** et **`ifelse`**

```
eqi = binop (\(I x) (I y) -> B (x==y))
gti = binop (\(I x) (I y) -> B (x >y))
lti = binop (\(I x) (I y) -> B (x <y))

ifelse = Op ifefun U where
  ifefun (_ : nxcode)
         (C elcod : C thcod : B cnd : pile)
         rtpile | cnd =       exec thcod pile (nxcode :> rtpile)
                | otherwise = exec elcod pile (nxcode :> rtpile)

iff = Op ifun U where
  ifun (_ : nxcode@(Op nxop _ : _))
       (C thcod : B cnd : pile)
       rtpile | cnd =       exec thcod pile (nxcode :> rtpile)
              | otherwise = nxop nxcode pile rtpile
```

Avec la boucle **`while`** la procédure sera un peu différente. Comme avant, on prévoit le bouclage en mettant le code original (dont le premier opérateur est **`while`**) sur la pile des retours, mais d'abord il faudrait dépiler *une seule fois* la procédure qui sera répétée. Cette fois, pour varier (***et en contradiction avec PostScript !***) la procédure ne sera pas mise sur la pile, mais elle constitue un paramètre extra de l'opérateur.

```
while proc = Op whfun (C proc) where
  whfun this@(Op _ (C prc) : nxcode@(Op nxop _ : _))
        (B cnd : pile)
        rtpile | cnd  =      exec prc pile (this :> rtpile)
               | otherwise = nxop nxcode pile rtpile
```

Tout le reste ce sont des tests. Définissons le **`cube`**, et trois version de la factorielle : récursive, itérative (récursive terminale), et itérative avec **`while`**.

```
cube = C [dup, dup, muli, muli, ret]

fact = C [dup, ldi 0, eqi,
          ldcod [pop, ldi 1,ret],
          ldcod [dup,ldi 1,subi,call fact,muli,ret], ifelse, ret]

rtfact = C [ldi 1, exch, goto fctmp]
fctmp  = C [dup, ldi 0, eqi,
            ldcod [pop,ret],
            ldcod [dup, ldi 1, subi, rot, muli, exch, goto fctmp],
            ifelse, ret]
```

```
whfact = C [dup, ldi 1, exch, ldi 0, gti,
            while [under, muli, exch, ldi 1, subi, exch,
                   under, ldi 0, gti, ret],
            exch, pop, ret]

prog ff = [ldi 3, call cube, call ff, stop]

resa = interp (prog fact)
resb = interp (prog rtfact)
resc = interp (prog whfact)
```

Le résultat est 108888694504183521607680000000, mais les tests sous Hugs donnent un résultat paradoxal : la solution la plus efficace est la première, récursive en profondeur, ce qui est une calamité pour un informaticien orthodoxe…

La section suivante retourne au modèle précédent, avec la machine virtuelle en forme de boucle, et avec l'environnement. **[Dans la prochaîne version de ces notes ceci sera révisé !]**

Le *threaded code* vit actuellement une renaissance. Notez que son implantation est une réalisation de bas niveau du concept des *continuations* : l'enchaînement des opérations assemblées de manière à ce que chaque opération connaisse son successeur, ce qui élimine la nécessité d'un dirigeant global.

Mais attention ! Nous utilisons dans nos construction un langage paresseux, ce qui peut provoquer un comportement inattendu de la part de la machine. En fait, même un concept si simple que la récursivité terminale, est non-trivial, et risque de déborder le tas-système si on ne fait pas attention. Faites l'expérience suivante :

```
n = 1000000

add n t | n==0      = t
        | otherwise = add  (n-1)  (t+0.05)
res = add n 0.0
```

Cette fonction calcule le produit $n \cdot 0.05$. Mais sous Hugs même si le programme est accepté, le résultat de la tentative d'affichage : `res` peut être le suivant :

```
(3320991 reductions, 8302514 cells, 17 garbage collections)
ERROR: Garbage collection fails to reclaim sufficient space
```

car au lieu d'ajouter 0.5 au tampon, la fonction construit un *thunk* qui suspend cette addition. La réduction de tous ces *thunks* différés a lieu au moment de l'affichage. La conclusion est : ***un langage paresseux n'est pas bien adapté à la construction des interprètes (ou programmes de simulation, etc.) par des personnes qui ignorent les intrications de la sémantique non-stricte***.

Cependant le problème est très bien connu, et tous les langages paresseux raisonnables offrent à l'utilisateur la possibilité d'effectuer la réduction stricte des arguments. En Haskell il existe un opérateur `($!)` dont la sémantique est la suivante : `f $! x` est équivalent à `f x`, mais `x` est évalué d'abord. Donc, pour que le programme marche, il faut le construire comme suit :

```
add n t | n==0      = t
        | otherwise = add  (n-1) $! (t+0.05)
```

Ceci dit, Hugs prend quand même plusieurs secondes pour afficher le résultat, et encombre le tas avec des miettes, ce qui déclenche 9 fois le ramassage des miettes. Le même programme en Clean, un autre langage paresseux, mais qui découvre **automatiquement**, sans aucun opérateur spécial le fait que l'opération `(t+0.05)` est stricte, donne le résultat immédiatemment, en moins de 0.1 secondes…

Un compilateur d'un langage paresseux pour être compétitif doit être équipé d'un bon analyseur automatique de la nécessité d'évaluation stricte, et il existe pour Haskell aussi, mais Hugs n'a pas été conçu pour être très rapide. (Les mêmes tests prouvent que GHCi n'est pas très rapide non plus).

Credo **religieux no. 6 :** La paresse était toujours le moteur principal du progrès de l'Humanité. Mais il ne faut pas exagérer, cette paresse doit être consciencieuse…

3.5.1 Co-procédures

Ceci est une digression importante. Rappelons la Fig.(2.1) qui montre la différence entre les procédures et les co-procédures. Après chaque phase d'activité, chaque module co-procédural doit «réveiller» son partenaire (ou un de ses partenaires), et suspendre son activité en sauvegardant son état local qui sera restauré au moment du «réveil».

La réalisation de ce mécanisme dépend des détails. Si les modules A et B sont statiques et distincts (s'ils ne sont pas des clones, des instances de la même définition de procédure), le mécanisme est plus facile à implanter dans un langage impératif. On opère avec l'adresse de retour comme avec des procédures normales, mais cette adresse n'est pas stockée sur une pile. Quand A réactive B, stocke l'adresse de retour dans sa zone privée et **statique** de données. La réactivation commence toujours par le début (A sait qu'il faut réactiver B, mais il ne sait pas où l'autre module s'est arrêté). Tout module co-procédural possède un code préfixe qui s'exécute au moment de la réactivation. Ce code récupère l'adresse de retour et branche. Dans le jargon de programmation des processus parallèles ceci s'appelle le *context-switching*.

Mais si le système simule de très nombreux vaisseaux spatiaux qui essaient de se détruire, ces vaisseaux seront des instances d'un seul module générique : la classe des vaisseaux. Ils partageront *tous* le même code, qui ne peut donc contenir aucune zone privée. Il faut alors établir pour chacun un **contexte** local qui remplace la zone statique de données mentionnée précédemment, et qui constitue *l'état local* de chaque co-procédure. Ce contexte est une structure de données assez simple, qui contient des données privées de chaque instance, et qui peut naturellement être crée dynamiquement sur le tas système, de préférence par des techniques d'allocation structurées et orientées-objet.

Dans le monde fonctionnel l'état privé d'une fonction, son contexte modifiable de l'intérieur n'existe pas. Cependant, les co-procédures sont *en principe* réalisables (et dans un langage qui dispose de la primitive `call/cc` ceci est facile et naturel). Rappelons l'essentiel d'un appel procédural standard : L'opérateur `call`

- sauvegarde sur la pile des retours l'adresse qui suit sa position, et
- branche à son paramètre.

Pour les co-procédures la suite d'opérations peut être la suivante. L'opérateur `resume`

- sauvegarde sur la pile des retours l'adresse qui suit sa position,
- identifie sur cette pile la co-procédure qui doit être réveillée. Si celle-là n'a jamais été activée auparavant, le `resume` se réduit à un appel. Mais si elle se trouve déjà sur la pile des retours,
- le contrôle «retourne» à elle.

Ceci est problématique, car comment le module appelant A peut identifier une adresse interne déposée par le module B quand celui-ci a suspendu son activité? Et même si on stocke sur la pile des retours les adresses avec quelques balises d'identification, la recherche risque d'être onéreuse, il faudra peut-être récupérer l'adresse du retour co-procédural de l'intérieur de la pile (elle n'est plus une pile !).

En fait, la technique la plus universelle, lisible et portable consiste à maintenir

- un tableau spécial *modifiable*, ou chaque co-procédure dépose son adresse de réactivation au moment du `resume` d'un autre module – ceci est avantageux si le nombre de co-procédures est connu, et si les co-procédures de réveillent directement (elles connaissent l'indice attribué au partenaire réveillé), ou bien
- une *file* (p. ex. une liste) avec les adresses de réactivation. Ceci est utile si les co-procédures «s'endorment» (en plaçant son contexte à la fin de cette file), mais la réactivation est à charge d'un pilote global, le «système d'exploitation», qui réactive toujours le premier (ou le prioritaire).

Nous n'allons pas implanter les co-procédures grâce à ce mécanisme, car pour les tester il faut prévoir que les co-procédures *fassent* quelque chose de non-trivial, qu'elles génèrent des effets de bord permettant de voir leur exécution quasi-parallèle. Notre machine virtuelle n'est pas bien adaptée à cette sorte d'exercices. La compilation du parallélisme dépasse les bornes de notre cours…

3.6 Le compilateur : première tentative

Les sections précédentes avaient pour but définir de manière la plus précise le modèle d'exécution de notre programme interprété. À présent construisons le générateur de code linéaire à partir de l'arborescence syntaxique. Ceci est (en principe) très simple.

Rappelons les structures définissant les expressions arborescentes, notre expression exemplaire, et introduisons deux tables: une qui décode les noms des variables, et l'autre qui transforme les noms des opérateurs (chaînes) en opérateurs – générateurs des **CodeItems**.

```
Opsymb = String
data Expr = L Value | A Opsymb Expr Expr

expr = A "-" (A "-" (L(F 1.0)) (A "*" (L(F 2.0))
          (A "+" (A "*" (L(F 2.0)) (L(S "x")))
              (A "/" (L(F 3.0)) (L(S "y"))))))
                  (A "*" (L(F 2.0)) (L(S "z")))

symtab = [("x",0),("y",1),("z",2)]
optab  = [("-",sub), ("+",add),("*",mul),("/",dvd)]
```

Nous aurons besoin de la fonction **assoc**, un peu plus générale que celle qui décodait les variables. Ici la fonction **assoc** est polymorphe et cherche un objet quelconque associé à une chaîne.

```
assoc _ [] = error "Pas d'association !"
assoc ch ((sy,v):q) | ch==sy    = v
                    | otherwise = assoc ch q
```

Le compilateur des expressions est triviale :

```
compil  :: Expr -> Code
compil (L val) = case val of
  F x -> [dblc x]
  S v -> [var n] where n=assoc v symtab
compil (A op e1 e2) = let f=assoc op optab in
                        compil e1 ++ compil e2 ++ [f]
```

La récursivité en cascade, surtout associée à la concaténation des listes n'est presque jamais une bonne chose. L'usage de la pile système est intense, mais le désavantage principal, si on utilise les listes classiques chaînes pour stocker le code linéaire est le fait que le fonction de concaténation **(++)** (connue en Lisp comme **append**) se trouve dans la clause récursive. Or, on sait que **append** recopie entièrement son premier argument pour l'attacher au second. Notre compilateur produira donc des copies des copies des copies…

L'optimisation d'une linéarisation hiérarchique d'un arbre est très bien connue et appartient à la panoplie standard de techniques fonctionnelles (et selon l'auteur de ces notes, si les étudiants en Informatique arrivent au bout du deuxième cycle sans connaître ces techniques, ils ne méritent pas leur diplôme, et/ou leurs enseignants sont coupables d'une négligence inadmissible…). Ce sujet a été traité dans l'exercice sur le tri arborescent. Faisons la même chose ici. Ajoutons un argument-tampon **tmp** à la fonction **compil** et construisons la fonction **compapp** dont la définition (**conceptuelle**, non pas réelle !) serait

```
compapp expr tmp  =  (compil expr) ++ tmp
```

Répétons que ceci n'est pas sa vraie déclaration, mais sa sémantique. La vraie structure utilise le «unfolding» (dé-pliage) de la concaténation :

```
compil expr = compapp expr [] where
 compapp (L val) tmp = case val of
   F x  -> (dblc x):tmp
   S v  -> (var n) :tmp where n=assoc v symtab

 compapp (A op e1 e2) = let f=assoc op optab in
                        compapp e1 (compapp e2 (f:tmp))
```

l'existence de deux branches implique deux appels récursifs, mais le dernier est terminal, et la concaténation a été éliminée.

La compilation et l'optimisation des structures de contrôle : des conditionnelles, des boucles, éventuellement d'un **switch**, c'est un bon exercice pour le travail individuel.

3.7 Exercices

(Les exercices concernent la machine avec la boucle centrale, sauf si nous précisons explicitement qu'il s'agisse du *threaded code*.)

Q1. Important ! Ajouter dans quelques opérateurs primitifs (**dup**, **exch**, opérations arithmétiques, etc.) un vérificateur de la pile, qui déclenche une exception (fonction **error**) si la pile est vide ou trop courte.

R1. Les modifications sont purement techniques. Au lieu d'écrire

```
dup = O (\_ _ rst later p@(x:q) -> (rst,later,x:p)) U
```

nous définissons

```
dup = O (\_ _ rst later p ->
            case p of (x:q) -> (rst,later,x:p)
                      _     -> error "Pile vide !") U
```

etc.

Q2. Les définitions des opérateurs primitifs comme **dup** etc. dans la dernière version de la machine sont horribles. Il serait beaucoup plus simple de pouvoir écrire simplement

```
dup p@(x:_) = x:p
pop (_:q) = q
exch (x:y:q) = (y:x:q)
```

etc. Peut-on simplifier cette notation baroque avec le constructeur **C**, arguments inutiles, constructeur «bidon» **U** etc.?

R2. Oui, notre proposition est la suivante. Les primitifs seront générés comme les **binop**'s :

```
dup = oppgen (\p@(x:_) -> x:p)
pop = oppgen  (\(_:q) = q)
```

etc., par le générateur

```
oppgen f = O (\_ _ rst later p -> let np=f p in
  (rst,later,np)) U
```

Q3. Construire l'opérateur **while** (pour la machine en forme de boucle) qui doit trouver sur la pile des données une condition (nombre entier) et une procédure. La procédure est exécutée si la condition est vraie, et le processus se répète, sinon le programme continue. La procédure peut (et en général le fera) modifier la pile, mais elle doit obligatoirement mettre sur le sommet un objet qui jouera le rôle de la condition pour l'étape suivante.

R3. Ceci est un exercice intéressant, car **while** d'habitude est un opérateur primitif *récursif*, et nous nous sommes refusés le droit d'utiliser la récursivité dans le langage d'implantation. Cet opérateur simule donc la récursivité dynamique par la «récursivité statique» : il empile sur la pile des retours sa propre instance si la condition d'itération est vraie.

```
while = O (\_ _ rst later ((Co prc):(I cnd):q) ->
  if cnd==0 then (rst,later,q)
            else (prc, (proc prc:while:rst):later,q)) U
```

Voici la sémantique : si la condition est fausse, l'opérateur ne fait rien, sauf dépiler et ignorer la procédure-boucle. Sinon, la procédure sera exécuté comme le nouveau code, mais **while** modifie le programme à exécuter, et place *devant* le code restant de nouveau la même procédure, et soi-même.

Notez l'auto-référence de while. Ce n'est pas un appel récursif direct, car **while** est une structure de données, et non pas une fonction. Pourquoi ceci ne déclenche pas une exception? Normalement une

struct ne peut pas contenir elle-même (mais elle peut contenir un champ contenant le pointeur sur soi-même). En fait, la construction constitue *presque* un appel récursif, car la référence à **while** dans sa définition est cachée à l'intérieur d'une fonction, dans le corps de la forme lambda. Ce corps sera évalué lors de l'application de la fonction par la machine virtuelle.

Dans des langages fonctionnels stricts comme ML ou Scheme on peut simuler l'évaluation paresseuse en cachant les objets différés à l'intérieur des formes lambda. Ce sujet sera encore abordé, mais nous avons déjà vu que

- les structures de contrôle, comme **if-then-else** ou **while** sont en fait des fonctions paresseuses, et
- la réalisation de ces structures dans notre machine virtuelle passe par l'empilement des objets fonctionnels anonymes (les codes «then» et «else»).

Notre définition de **while** sera utilisée pour définir la fonction factorielle itérative qui calcule $n!$ comme $n(n-1)(n-2)\cdots 2\cdot 1$. L'opérateur **roll** tourne les trois derniers éléments de la pile un peu différemment de **rot**.

```
roll = O (\_ _ rst later (x:y:z:q) -> (rst,later,z:x:y:q)) U

zerop = exec [dup,intc 0,eqi,ret]
decr = exec [intc 1,subi,ret]

ifac = exec [zerop,
 proc [pop, intc 1, ret],
 proc [dup, decr, dup,
       proc [dup, roll, muli, exch, decr, dup, ret],
       while, pop, ret],
 ifelse, ret]

ifres7 = interp envir [intc 7, ifac, ret]
```

La dernière ligne produit [5040] sans protester.

Q4. Implanter la boucle **repeat** – un opérateur primitif qui trouve sur la pile des données la condition de continuation et une procédure, et qui exécute la procédure avant de vérifier la condition. Si elle est vraie, le processus se répète.

Implanter aussi l'opérateur **loop** qui boucle sans fin, exécutant la procédure trouvée sur la pile. Il faut être raisonnable, et implanter aussi un opérateur primitif **break** qui arrête une telle boucle. Ceci est un défi conceptuel, car comment désactiver l'opérateur **loop** de l'intérieur de la procédure exécutée?

R4. Non, pas de réponse ici. Ceci est *vraiment* un bon sujet d'examen.

Q5. Question difficile. Nous voudrions ajouter à notre machine virtuelle un module de déboguage. Chaque fois quand l'interprète exécute un opérateur, une information est affichée. (On peut donner à chaque opérateur un attribut textuel, par exemple son nom, et les fonctions d'empilement «écrivent un rapport» de leur activité).

Pourquoi c'est difficile? Parce que notre machine est fonctionnelle. Il n'y a pas d'effets de bord, alors le résultat est renvoyé à la fin, quand la fonction termine le travail, ce qui n'est pas bon pour le déboguage, surtout si l'évaluateur déclenche une exception en mi-chemin...

Cependant si le langage d'implantation est paresseux, et si une fonction produit une liste ou une chaîne qui sera éventuellement affichée, la création procède de manière incrémentale : une liste partielle peut être formée et affichée avant la catastrophe. Essayer de comprendre cette stratégie et de l'implanter.

R5. Ce problème sera discuté plus tard, dans le contexte des structures de contrôle *monadiques*.

Q6. La fonction **compil** est un exercice en parcours des graphes arborescents. Écrire une fonction simplifiée, qui n'a pas besoin de décoder les variables ou les opérateurs, mais place les objets (les feuilles ou les opérateurs) directement dans la liste de sortie. Essayer d'écrire cette fonction linéarisante *de manière combinatoire*, sans paramètres. Commencer au moins par l'élimination du tampon **tmp** grâce au combinateur **comp** (en Haskell : **(.)**).

R6. Commençons par le code original, récursif arborescent :

```
type Val = Int                    -- pour la discipline
type Arb = F Val | A Val Arb Arb

flat (F x) = [x]
flat (A x gauche droite) = flat gauche ++ flat droite ++ [x]
```

et la variante optimisée :

```
flat a = flatmp a [] where
 flatmp (F x) tmp = x:tmp
 flatmp (A x gauche droite) tmp =
   flatmp gauche (flatmp droite (x:tmp))
```

La fonction récursive interne peut être simplifiée. La clause terminale aura la forme **`flatmp (F x) = (x :)`**. La clause récursive subira les transformations suivantes :

```
(flatmp gauche) ((flatmp droite) (x:tmp)) =
((flatmp gauche) . (flatmp droite)) (x:tmp) =
( ((flatmp gauche) . (flatmp droite)) . (x :)) tmp
```

et ainsi le tampon est éliminé. On ne peut aller plus loin, car la sémantique de la fonction **`flatmp`** contient un discriminateur des types, et les combinateurs sont par nature polymorphes. Bien sûr, la définition : **`f x tmp = x:tmp`** se réduit à **`f = (:)`**, mais on ne peut pas réduire à la forme sans paramètres une fonction qui a deux clauses distinctes, il faut utiliser **if-then-else**. Laissons au lecteur le reste de cette conversion.

Le but de cet exercice est un peu différent : comparez la forme récursive arborescente avec la forme optimisée et réduite. Elles ont presque la même structure, avec le compositeur **`(.)`** replaçant la concaténation. Comparez ceci aux exemples d'affichage des objets de type **`Value`**, et à la construction de la fonction **`showPrec`**. Ceci nous aidera à composer les parseurs.

Q7. Optimiser (linéariser) la fonction **`flat`** grâce aux continuations.

R7. Définissons d'abord l'opérateur **`(:)`** continué

```
cc x y cnt = cnt (x:y)
```

Avec cet opérateur on peut convertir la concaténation

```
apc [] l cnt = cnt l
apc (x:q) l cnt = apc q l $ \r -> cc x r cnt
```

où l'opérateur standard **`($)`** est un «applicateur» : **`f $ x = f x`** mais de très faible précédence, ce qui économise l'usage des parenthèses. Notez que la concaténation standard :

```
[]    ++ l = l
(x:q) ++ l = x : (q++l)
```

est une fonction récursive non-terminale, qui dans le cas d'un langage strict, où les arguments sont évalués avant l'appel de la fonction, effectivement empile élément par élément la totalité du premier argument. Donc, si la pile système, qui limite la profondeur de la récursivité, est courte (ce qui est une bonne idée, car sinon c'est du gaspillage de mémoire presque toujours vide), on ne peut pas concaténer une liste très longue avec une autre.

Dans la version continuée la concaténation est récursive terminale, mais il y a un prix a payer : une **fermeture**, une fonction anonyme : **`\r -> cc x r cnt`** qui «attrape» **`x`**, **`cc`** et **`cnt`** est dynamiquement crée sur le tas système pour chaque **`x`**. Ainsi on est limité seulement par la mémoire dynamique du système, d'habitude beaucoup plus large que la pile. Mais la consommation physique des ressources *est* plus importante.

La conversion de la fonction **`flat`** est

```
flatc (F x) cnt = cc x [] cnt
flatc (A x g d) cnt = flatc g $ \a ->
                      flatc d $ \b -> apc a b $ \r ->
                                      apc r [x] cnt
```

ce qui peut être encore simplifié.

Q8. Implanter les affectations en suivant les conseils dans le texte.

R8. Désolé, mais les conseils ont été mis justement pour encourager un travail personnel de la part des lecteurs. Ceci est un bon sujet d'examen.

Q9. Pourquoi avons-nous forcé la présence de l'instruction RET ou de l'opérateur **`ret`** dans toute procédure? Vérifier que la liste est vide est très simple et facile, et le code devient plus court.

R9. Ne lisez pas la réponse ! Essayez *vraiment* de répondre à cette question. D'ailleurs, la réponse se trouve dans le texte de ce chapitre.

OK. Il existe deux raisons importantes.

- Imaginez que les procédures ne sont pas stockées dans des listes, mais dans des segments contigus d'un tableau-buffer global. La machine au lieu d'utiliser pour son paramètre-code la liste, qui en fait est représentée par le pointeur, prend l'indice (l'adresse) du segment correspondant. Comment séparer les procédures? Physiquement elles ne se terminent pas, car elles sont suivies par les autres. L'opérateur de retour remplit cette tâche.
- On peut envisager un retour conditionnel au milieu de la procédure. Le retour *est* une opération de base pour presque toutes les machines virtuelles de bas niveau.

Q10. (*Tu l'as voulu, Georges Dandin...*) Implanter le retour conditionnel.

R10. Bien sûr, on ne peut pas mettre une procédure **`[ret]`** sous **`ifelse`**, car une procédure ne peut pas sortir non-localement de son module appelant. Mais on peut parametrer l'opérateur **`ret`**. Au lieu de

```
ret  = O (\_ _ _ (rt:demain) pile -> (rt, demain, pile)) U
```

nous aurons

```
ifret  = O (\_ _ rst ltr@(rt:demain) (cnd:pile) ->
   if cnd/=0 then (rt, demain, pile) else (rst,ltr,pile)) U
```

qui ne fait rien si la condition n'est pas satisfaite.

Q11. Nous avons écrit: *une procédure ne peut pas sortir non-localement de son module appelant.* Généralement ceci est faux ! L'opérateur **`stop`** le fait. Écrire un opérateur **`break`** qui sort du bloc où il se trouve, mais également du bloc englobant. Ceci est un outil permettant d'arrêter les boucles sans fin.

R11. Rien de plus facile, regardez les définitions du **`ret`** et **`stop`**. On sait que le code qui sera exécuté après le retour normal est placé sur la pile des retours. Il suffit de l'éliminer.

```
break = O (\_ _ _ (viré:hop:après) pl -> (hop, après, pl)) U
```

Q12. Pourquoi dans notre version du *indirect threaded code* (mais les autres modèles de machine virtuelle donneraient des résultats pareils), la version récursive de la factorielle est plus efficace que les versions itératives?

R12. C'est l'effet de l'interprétation par Hugs. L'efficacité ici signifie le nombre de réductions effectué par Hugs, et non pas par le processeur matériel. Une opération primitive est équivalente à une autre, et un programe un peu plus long (à cause de la présence du tampon, et de sa manipulation) sera moins efficace. Les opérations primitives sur la pile machine ne sont pas onéreuses. Mais on pourra optimiser encore ce code !

Q13. Alors, comment optimiser la machine sans boucle, qui passe le contrôle par *threaded code?*

R13. En fait, ceci fait **partie intrinsèque de notre générateur du code** futur, et nous voulions aborder cette question plus tard. Mais nous pouvons le faire tout de suite. Une machine de plus bas niveau encore, plutôt style assembleur que PostScript ou FORTH peut éviter beaucoup d'appels/retours grâce aux branchements conditionnels. Plus concrètement :

- Les opérateurs **`ifelse`**, **`while`**, etc. ne demanderont plus le chargement du code à exécuter sur la pile des données. En effet, si ce code est produit par le compilateur *statiquement*, avant l'exécution du programme, il est inutile de le mettre dans le programme pour l'empiler et ensuite dépiler et passer à l'opérateur en question. Ce code (son adresse) peut être placé comme la paramètre de l'opérateur.
- Il n'y aura plus de *procédures* à exécuter, ce qui nous a obligé de sauvegarder la continuation sur la pile des retours. Plus d'appels, plus de retours !
- Les opérateurs **`ifelse`**, **`while`** ne seront plus des primitives de la machine, mais leur générateurs(**`ifelsegen`**, **`whilegen`**) construiront des séquences de code liées par les opérateurs de branchement **`goto`**, **`ifgoto`** (branchement en cas de succès) et **`ifnotgoto`** (branchement en cas d'échec). Voici, sur la Fig. (3.2) la structure du code réalisant **if** *condition* **then** *code then* **else** *code else*, et sur la Fig. (3.3) la boucle **while**.

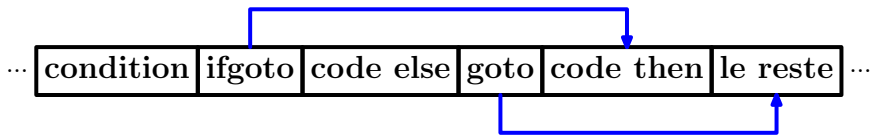


Fig. 3.2: If-then-else dans un code de bas niveau

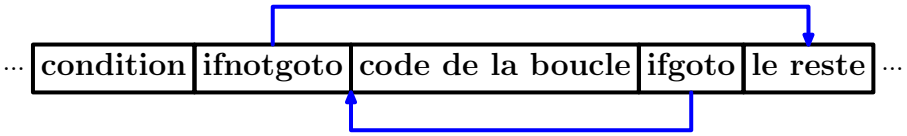


Fig. 3.3: Structure de la boucle while

Voici le codage de nouvelles primitives de branchement :

```
goto proc = Op (\(Op _ (C prc) : _) -> exec prc) (C proc)

ifgoto proc = Op ifgofun (C proc) where
  ifgofun (Op _ (C prc) : nxcode@(Op nxop _ : _)) (B cnd : pile)
    | cnd       = exec prc pile
    | otherwise = nxop nxcode pile
ifnotgoto proc = Op ifngofun (C proc) where
  ifngofun (Op _ (C prc) : nxcode@(Op nxop _ : _)) (B cnd : pile)
    | cnd       = nxop nxcode pile
    | otherwise = exec prc pile

ifelsegen cndlst thenlst elselst next  =
  cndlst ++ (ifgoto the : elselst ++ (goto next : the))
   where the = thenlst ++ next


whilegen cndlst prclist next =
  cndlst ++ ifnotgoto next : pr where
    pr = prclist ++ ifgoto pr : next
```

Ceci correspond aux diagrammes sur les Figures (3.2) et (3.3). Notez comment nous avons profité de la programmation paresseuse pour «boucler» une structure de données (une liste) et de la rendre auto-référentelle. Ceci est possible avec un langage strict par un algorithme à deux passes : d'abord on construit la liste avec des «trous», et ensuite on *modifie physiquement* le résultat, en remplissant les trous par les adresses voulues (en avant ou en arrière, mais toutes déjà connues). L'algorithme paresseux nécessite une seule passe.

à présent nous pouvons construire d'autres variantes de la immortelle factorielle. Voici le code récursif et le code avec la boucle **`while`** :

```
recfact = C (ifelsegen [dup, ldi 0, eqi]
                       [pop, ldi 1]
                       [dup,ldi 1,subi,call recfact,muli]
                       [ret])

whilfact = C (whilegen [dup, ldi 1, exch, ldi 0, gti]
                       [under, muli, exch, ldi 1, subi, exch,
                        under, ldi 0, gti]
                       [exch, pop, ret])
```

(mais n'essayez jamais d'imprimer ces listes, elles sont cycliques !) Notre nouvelle solution est «presque parfaite», plus économique que les précédentes, ce qui n'empêche que la solution récursive est plus efficace que celle avec la boucle, et toujours pour les mêmes raisons…

Q14. Question très délicate : peut-on construire le «code enfilé» en C ou C++.

R14. Réponse encore plus délicate : oui et non. Le problème est qu'en C le **goto** n'est pas dynamique ! Certes, il y a des pointeurs sur les fonctions, mais ils permettent d'*appeler* les fonctions, et non pas de brancher directement sur elles. Pour des raisons toujours un peu obscures, en C il n'y a pas de récursivité terminale optimisée ! (même si quelques compilateurs offrent cette option).

Le **goto** standard demande la présence d'une étiquette statique, nommée dans le programme. Mais il existe une extension GNU C (ce qui peut être considéré comme «presque standard»), permettant d'affecter des adresses variables à l'argument d'une instruction de branchement. Un tel programme marche :

```
main(int argc, char *argv[])
{void *labl; labl = &&l1;
 // ....
 goto *labl;
 cout << "ne sera pas affiché\n";
 goto lend;
 // ....
 l1:cout << "nous y sommes !\n";

 lend: cout << "fin du programme...\n";
 return 0;
}
```

Cependant, il reste impossible de transférer le contrôle d'une procédure à l'autre, les étiquettes doivent rester locales (cette restriction était relaxée en FORTRAN avec le *ASSIGNED GOTO*, mais cette structure était vraiment dangereuse). L'appel d'une fonction ce n'est seulement pas la sauvegarde de l'adresse du retour et le branchement, mais la création du *frame* pour l'instance d'activation de la procédure appelée, la mise à jour de la pile système, etc. Quelques personnes qui connaissent bien ce protocole et savent programmer en assembleur, ont su éliminer ces charges et construire un véritable *threaded code* en C (p. ex. les travaux d'Elliott Miranda sur Smalltalk), mais les résultats ne sont toujours pas portables.

Toutefois la situation n'est pas complètement désespérée si quelqu'un ne s'intéresse pas par l'efficacité de la solution, mais uniquement au concept de *threading*. Pour cela il faut restaurer la machine-interprète central, la boucle. Dans cette boucle on appelle une fonction et on place sa valeur de retour dans un registre global.

La fonction appelée termine son travail par l'opération **`JUMP(une autre fonction)`**. Mais **`JUMP`** n'est pas un **goto** (qui n'existe pas dans ce contexte), ni un appel standard, mais… – oui, vous avez deviné – le **retour**! On informe donc la boucle centrale quelle est l'adresse suivante, et la boucle appelle l'adresse stockée dans son registre global. Et la situation se répète, jusqu'à une instruction spéciale, disons **`STOP`**, qui provoque le **`break`**, ou déclenche une exception plus dramatique, p. ex. le vtlongjump.

En fait, ce protocole, pas très efficace mais portable, était la source de l'inspiration pour notre modele de machine virtuelle où chaque opérateur prépare l'adresse suivante à exécuter pour la boucle principale de l'interprète.

***Credo* religieux no. 7 :** Pour vivre longtemps, un langage de programmation doit être *mauvais*, comme C. Si un langage est bon, les gens s'y jettent pour l'améliorer encore plus, mais avec un langage mauvais la situation est désespérée, donc on le laisse en paix.

Chapitre 4

Les tâches et la structure d'un compilateur

4.1 Un peu d'anatomie et de physiologie

La «vraie» compilation est la génération du code. Le nom, d'ailleurs, est historique ; il signifiait le ramassage des morceaux du programme en une entité organique, exécutable. Les morceaux résidaient sur cartes perforées, et la compilation était moins une translation, que le montage d'une pile. À présent cette sorte de «compilation» est plutôt la tâche de *l'editeur des liens* que du compilateur…

Avant qu'un «ouvrage littéraire», le texte source d'un programme ne se transforme en quelque chose d'exécutable, en une application indépendante, ou un tableau interne exécuté par un interprète, il faut comprendre le texte du programme, l'analyser, le décomposer en unités primitives, et ensuite, à partir d'une description très précise et complète de ses éléments, de sa structure, et de sa sémantique on peut créer le code pour une machine-cible. Alors la compilation se décompose *par convention* en deux phases majeures :

- L'analyse,
- La synthèse

mais cette division n'implique pas que les deux phases soient *vraiment* distinctes, assez souvent elles se recouvrent partiellement, elles se chevauchent, et le compilateur bascule entre deux modes de travail plusieurs fois durant un cycle de compilation. De plus, la phase d'optimisation de code possède d'habitude des modules analytiques et synthétiques, et la table des symboles qui assure la correspondance entre les noms et les références, est gérée simultanément par les deux catégories de modules.

L'analyse est d'habitude divisé en étapes : lexicale, syntaxique et sémantique, ce qui facilite la conceptualisation et la modularisation de l'analyseur, et permet l'usage des outils simples pour des tâches simples. Par exemple, l'analyse lexicale peut s'appuyer sur les automates finis, sans mémoire, tandis que la structure récursive phrasale d'un langage nontrivial exige que l'analyseur syntaxique gère une pile. Mais – soulignons – cette séparation est vraiment conventionnelle, et pour nous elle sera secondaire. Une vision globale, homogène est pour nous plus importante que la modularisation. Un officier, par exemple un stratège du Quartier Général voit l'armée à travers les Forces, les Divisions, les Unités ; il distribue les tâches et organise la communication globale. (Et il ne demande pas qu'un sous-officier gagne seul une bataille, sauf s'il s'agit du John Rambo). Mais pour son supérieur politique, il existe *une* armée qui doit gagner la guerre, et il doit gérer cette armée de manière uniforme. Et vous pensez que le mot «uniforme» vient d'où? (Mais, plus sérieusement, la construction d'un analyseur monolithique qui comporte des éléments lexicaux et syntaxiques est plus facile).

Également la *synthèse* tout court couvre la synthèse d'un code intermédiaire, son optimisation (qui – comme nous l'avons dit – partiellement appartient à l'analyse) et – éventuellement – la sortie du code binaire final. Parfois on arrête la synthèse assez tôt et on exécute directement le code intermédiaire ; tel est le cas des interprètes classiques, comme les interprètes du Basic, Lisp, Prolog ou FORTH. (Mais presque jamais cette exécution ne se fait «instruction par instruction», ce que suggèrent quelques auteurs des livres anciens sur la compilation.)

Cette exécution ne doit pas forcément suggérer les calculs numériques suggérés par les machines virtuelles montrées précédemment. Il y a des compilateurs dont le code-cible sont des commandes graphiques qui permettent de programmer un dessin technique ou une scène d'animation. ***Ce*** texte a été formaté par un metteur en page spécialisé, LaTeX équipé avec un interprète de commandes (macros). Nous l'avons tapé en utilisant un éditeur textuel standard. Si nous voulions taper une formule mathématique relativement riche, comme, par exemple

$$\sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n^2+\frac{1}{2}}}$$

nous'écririons :

```
$$
\sum_{n=0}^\infty{{\alpha^n \over \sqrt{n^2+{1\over 2}}}}
$$
```

sans hésitation. La tâche de l'auteur humain est la *logique* de cette expression, sa structure correcte, sa signification, mais sa mise en page et les problèmes visuels – on les laisse à la discrétion du LaTeX. Il sait comment comprendre les mots comme **`$\alpha$`** et les «compiler» en α, il sait que les accolades délimitent les arguments d'une forme fonctionnelle, comme **`\sqrt{...}`**, il sait appliquer la récursivité. Finalement, le «code» généré – soit un fichier **`.dvi`**, soit un document en format PDF – contient les instructions de positionnement des entités graphiques sur les pages, les instructions de saut de page, etc., exécutées par l'interprète Acrobat Reader, ou l'interprète PostScript de l'imprimante (après une seconde compilation qui transforme le code précédent considéré comme intermédiaire, en PS).

Prenons cependant un exemple plus «classique». Voici une instruction d'affectation écrite en Pascal ou un autre langage de ce genre:

```
x := 5+alpha*3*(21 - x+3*alpha)-1.0
```

4.1.1 Le lexique

Ceci est un *texte*, une chaîne de caractères. Il faut tout d'abord reconnaître et séparer les mots, les unités lexicales, afin de pouvoir reconstruire les valeurs numériques comme «21» et de pouvoir reconnaître qu'il y a deux occurrences de la même variable **`alpha`**. C'est le rôle de l'analyseur lexical connu également sous le nom de *scanneur*. Le scanneur doit reconnaître les commentaires et les espaces sans signification, générer (ou préparer la génération) les nombres, les symboles identificateurs, et les mots-clés (il peut les distinguer, ou laisser cette tâche au module syntaxique), etc., et en général, transformer le texte en une suite de *lexèmes*. Ces lexèmes passent au module suivant, syntaxique, avec leurs *catégories lexicales*. L'analyseur souvent n'est pas intéressé par les noms concrets des variables ou par les valeurs numériques définies, mais uniquement par la catégorie des objets. Il suffit de savoir qu'un lexème représente un nombre flottant, pour pouvoir compiler le code qui traite ce nombre. (Le module de synthèse qui engendre les références aux données doit, bien évidemment, donner au programme compilé l'accès à la valeur numérique d'un objet, mais ceci viendra plus tard).

Les catégories classiques sont: identificateurs, constantes entières, réelles, Booléennes, etc., opérateurs binaires comme **`<>`** en Pascal ou **`!=`** en "C", mots-clé comme **`while`**, éventuellement aussi les séparateurs ou les terminateurs comme le point-virgule. Il faut y ajouter les parenthèses, crochets, accolades, etc. On appelle ces catégories de lexèmes des *jetons* (ou *tokens*).

Il faut parfois prendre des décisions délicates. Qu'est-ce que c'est : **`17alpha`**? En Pascal c'est une calamité, une erreur. En Lisp cela peut être *un* lexème normal, un identificateur. Dans la plupart d'autres langages un caractère alphabétique arrête le scanning d'un nombre, donc nous aurons deux lexèmes: **`17`** et **`alpha`**. Mais ensuite: est-ce légal, cette suite de lexèmes? Cette décision n'appartient plus à l'analyse lexicale, bien qu'un analyseur lexical peut diagnostiquer une faute lexicale si le lexème est terminé par un caractère considéré illégal (par exemple : un nombre suivi directement par une lettre). De très rares langages (comme par exemple le système MetaPost l'acceptent et traitent comme 17 multiplié par *alpha*, mais cette interprétation n'appartient non plus au scanneur, et dans d'autres langages le parseur ne l'accepte pas.

Une autre décision délicate est la gestion de *format* du document d'entrée. Fortran prévoyait une instruction par ligne, alors tout retour-chariot était un caractère spécifique – le terminateur. La plupart de langages plus modernes considèrent la fin de ligne comme tout autre espace blanc qui sépare les entités lexicales. Ceci implique l'usage intense de caractères de séparation *syntaxique* : les points-virgules. Cependant – pourquoi la philosophie de Fortran ou Basic dans ce contexte est pire? On note donc actuellement plusieurs réponses à la question concernant la signification du *layout*.

- En Matlab les points-virgules sont optionnels. On peut terminer une instruction par «`;`», mais si le terminateur est absent, si l'instruction se termine par fin de ligne, le résultat est automatiquement affiché, la présence du point-virgule bloque l'affichage. Ce caractère joue alors un rôle sémantique !

- Le langage de calcul formel Maple prévoit deux terminateurs : point-virgule, et deux points. *Le second* bloque l'affichage du résultat, le premier force l'affichage. (Le paquetage Matlab possède un module d'interfaçage avec Maple. Le travail simultané avec les deux systèmes est nuisible à la santé psychique de l'utilisateur. . .)

- Python considère la fin de ligne comme terminateur, mais conditionnel ; si la ligne suivante est indentée, elle est considérée comme continuation. Haskell suit la même philosophie, avec quelques différences. En général, les langages qui respectent des règles spécifiques d'indentation sont très lisibles, mais il faut être plus vigilant pour éviter des fautes dues à la négligence.

- Tout langage a ses propres techniques de déclarer qu'une ligne est la continuation de la précédente. En C on termine la ligne précédente avec *backslash*. En Matlab la ligne suivante commence par `...` , et Fortran classique demande que la colonne 6 de la ligne suivante contienne un caractère non-blanc.

- Un compilateur professionnel doit naturellement diagnostiquer les fautes, et signaler l'endroit où ellels ont été découvertes. Ceci suggère que le scanneur doit passer à l'étape suivante les lexèmes *avec leurs positions – les numéros de ligne et de colonne dans le document-source*. (Ces informations doivent survivre aussi l'analyse syntaxique, car parfois lors de l'analyse sémantique on découvre quelques fautes, par exemple une erreur de typage, et il est utile de pouvoir les localiser précisement, même si les erreurs de typage sont souvent «distribuées»).

Cette prolifération de protocoles lexicaux continue jusqu'aujourd'hui, et permet d'exprimer notre

***Credo* religieux no. 8 :** Les créateurs des langages de programmation sont tous des grands enfants et esprits artistiques, qui ne se refusent presque jamais le plaisir d'introduire des petites différences inutiles par rapport aux langages existants, et de démontrer ainsi leur originalité.

Mais aussi :

***Credo* religieux no. 9 :** Les enfants ont normalement une vie longue et joyeuse devant eux, tandis que les gens stables, ceux qui résistent aux changements, hmmmm. . .

Une chose est certaine. Le scanneur transforme le flot de caractères en unités atomiques, reconnaît leur catégories lexicales, et passe au module syntaxique les jetons, avec leur catégories. Accessoirement il construit la table des symboles où *chaque atome existe en un seul exemplaire* (sauf si le langage prévoit l'existence de plusieurs espaces de noms (*namespaces*).

L'analyseur lexical peut avoir plus d'intelligence que l'on ne lui accorde traditionnellement.

- Il peut reconnaître les caractères de manière parametrée. D'habitude il «sait» *a priori* distinguer les lettres, les chiffres, les parenthèses ou autres caractères spéciaux, l'espace blanc, etc., les attributions des catégories lexicales c'est fait statiquement. Mais ceci n'est pas la seule stratégie possible. On peut arbitrairement assigner la catégorie «lettre» à n'importe quel caractère, et ceci n'est pas un problème académique ! Si on travaille avec les alphabets plus riches que le notoire ASCII, et si on veut que les lettres accentuées, ou portant d'autres signes diacritiques (cédille roumaine ou turque, lettres "ś", ou "ż" polonaises), éventuellement les caractères en cyrillique ou arabe codés selon quelques standards particuliers, soient reconnues comme telles, la meilleure stratégie est de commencer par définir un tableau attribuant aux caractères leurs catégories. Ceci peut faciliter la reconnaissance des guillemets ou des chevrons : "`<< >>`" comme des parenthèses spécifiques (ces derniers sont obligatoires dans des documents Français officiels), ou forcer la prise en compte des espaces ou des fins de ligne. (Cette technique

est utilisée moins souvent qu'elle ne le mérite à cause de la monopolisation de l'informatique par la coquille culturelle anglo-saxonne, et par la prédomination de langages de programmation «classiques». Mais le standard Unicode commence à se faire reconnaître…).

- Au niveau de l'analyse lexicale on doit traiter les *macros* qui forcent le remplacement d'un lexème par une suite quelconque d'autres lexèmes. En présence des macros parametrés, éventuellement récursives, la collaboration entre la couche lexicale d'entrée et l'analyseur syntaxique se complique considérablement. (Et ceci rend le langage difficile non pas seulement au constructeur du compilateur, mais aussi pour les personnes qui apprennent ce langage ; ceci est le cas de MetaPost, Clean et partiellement aussi du Scheme).

Cependant, d'autre part, accorder à un module trop d'intelligence est toujours risqué. Les macros en C (les clauses **`#define`**) sont traitées par un programme spécial – le préprocesseur.

4.1.2 Syntaxe et Sémantique : introduction

Le module syntaxique construit les arborescences dérivées de la structure phrasale du programme.

Il ne faut pas croire que ces arbres sont toujours créés physiquement dans la mémoire de l'ordinateur, avec les pointeurs, etc. Sachant que la structure récursive des phrases, par exemple des expressions arithmétiques composites correspond souvent aux appels récursifs du parseur, qui s'appelle lui-même pour analyser une sous-expression parenthésée, l'arbre syntaxique peut être «virtuel», caché dans la **pile des instances** du parseur récursif (où dans la liste de **continuations** latentes). La construction de cette pile par les appels, et sa destruction par les retours constituent le parcours par ce graphe virtuel, et permettent la génération du code linéaire sans jamais utiliser de vrais arbres. Ceci est la stratégie appliquée par de nombreux compilateurs de Pascal, surtout les compilateurs rapides, qui génèrent le code final en une passe (comme le premier compilateur de Wirth et Amman, et les anciens compilateurs de Borland).

Cependant, parfois il est souhaitable de générer un code intermédiaire indépendant du parseur, il faut alors construire l'équivalent d'une structure arborescente dans la mémoire ou sur un fichier, avec l'adressage relatif. Ceci alourdit considérablement le compilateur, mais permet une *meilleure optimisation* du code final, et son assemblage à partir de modules compilés séparemment. Et ce n'est pas si mauvais pour la pédagogie de la compilation.

On n'est pas obligé d'accepter, mais on doit comprendre le

***Credo* religieux no. 10 :** Analyser et comprendre une structure textuelle équivaut à générer un objet correct à partir de cette structure. Le seul moyen de savoir si un soldat a compris un ordre est de vérifier qu'il l'a exécuté correctement.

Dans la théorie on peut réduire un analyseur à une machine, qui doit arriver finalement à un état (un nœud de son graphe d'états) terminal. Mais 95% de travail est la création du «tracé» de ce parcours par le graphe d'états, la construction du code, la mise à jour de la table de symboles, etc. Ce n'est pas la grammaire elle-même qui détermine l'utilité d'un langage de programmation, mais toute sa «décoration sémantique».

Voici quelques propriétés sémantiques des objets linguistiques dans le programme :

- Les nombres ne sont pas seulement des séquences de chiffres, mais possèdent des valeurs numériques.
- Un atome symbolique (identificateur) n'est pas une séquence de lettres, mais possède son *identité* propre, et sa catégorie : variable, nom de type, mot-clé, étiquette, macro, etc.
- Toutes les constantes, variables et expressions possèdent leur *types*, parfois statiques, et parfois dynamiques (type d'une variable est temporairement le type de sa valeur, si le langage n'est pas typé). Les types existent même dans des langages «non-typés» comme Scheme, sinon comment peut-on calculer une valeur? Quelles opérations appliquer?
- Les étiquettes «savent» à quel code elles se réfèrent (l'endroit-cible).
- Une instruction de branchement connaît sa ou ses destinations.
- Un fragment de code possède une longueur précise, et tôt ou tard aussi un emplacement (adresse) dans la mémoire.

- Une expression peut être marquée comme constante et pre-calculée par le compilateur avant l'exécution du programme.
- Une instruction peut être marquée comme inaccessible ("*dead code*"), et éliminée par l'optimiseur.

Tout ceci est si important, que nous ne pouvons détacher complètement l'analyse formelle de l'analyse sémantique. La sémantique aura pour nous plutôt un goût opérationnel que dénotationnel, elle sera liée conceptuellement plutôt à la synthèse du code qu'à son analyse.

4.1.3 Lex et Yacc – premiers commentaires

Lex est un populaire *générateur de scanneurs* qui appartient à la couche brevetée de l'Unix. Son remplaçant libre s'appelle Flex et il est presque totalement compatible avec Lex.

Yacc (*Yet Another Compiler Compiler*) est un générateur de parseurs LR(1), dont le clone GNU s'appelle Bison.

Les *générateurs d'analyseurs* sont des programmes qui lisent la description syntaxique (décorée) d'un langage de programmation, par exemple sous forme de productions BNF ou d'expressions régulières, et qui construisent un analyseur-hamburger, prêt à la consommation. Cet analyseur – scanneur ou parseur, est une fonction qui doit être insérée par le programmeur dans l'ensemble des sources de son application, et compilée avec.

Entre les années '70 et '80 il y avait une tendance d'affaiblir l'enseignement de la construction «manuelle» de parseurs, et jusqu'au aujourd'hui on trouve dans quelques livres et polycopiés un clivage entre la théorie du parsing ou la théorie des automates finis, très élaborée et complète, et les exemples pratiques de parseurs, qui sont trop souvent simplistes. Parfois les exemples sont primitifs et si mal codés (par exemple, on trouve des instructions conditionnelles **if-elseif-elseif…** avec plusieurs dizaines de clauses, ou des **switch** gigantesques), que ses auteurs auraient dû ajouter à leur textes des incantations magiques de genre : «ici on ne discute que la méthodologie générale de construction. Si vous avez un problème *réel* de parsing, prenez Yacc…». Voici les avantages qu'apportent les générateurs de parseurs :

- Le programmeur définit son langage de manière *statique*, en définissant la grammaire. La partie sémantique n'est pas si statique que ça, les procédures sémantiques doivent être codées explicitement aussi, mais on les attache *statiquement* aux productions de la grammaire.
- Le générateur effectue pour nous les tests de la validité de la grammaire. Le langage mal conçu sera rejeté dans plusieurs cas. Un parseur manuel d'habitude est plus fragile, et il a plus de chances d'être bogué.
- Le parseur est construit de manière modulaire. Les protocoles de communication avec autres modules du compilateur sont standardisés, et le travail en équipe est facilité.
- On trouve de très nombreux exemples de grammaires, et de scanneurs et parseurs réalisés avec les générateurs, ce qui facilite leur apprentissage, et on peut créer un nouveau langage et parseur en modifiant une réalisation existante.

Cependant, il ne faut pas oublier non plus quelques désavantages.

- Les générateurs figent le langage d'implantation, Lex et Yacc sont adaptés au C. Ceci n'est pas satisfaisant pour tout le monde. Il existe des clones de Yacc écrit en Pascal, ML (SML et CAML) et aussi en Haskell (Happy), mais ceci n'est pas *la* solution universelle de ce problème.
- Les protocoles de communication avec la table de symboles, générateurs de codes, etc. peuvent être considérés *trop rigides*. Adaptation *du reste* du compilateur aux protocoles de Yacc et Lex peut être un peu pénible.
- Le déboguage des procédures sémantiques peut être très difficile, car l'utilisateur ne contrôle pas directement le contexte de leurs appels : ils sont codés automatiquement. (Ce problème peut être présent dans les parseurs manuels aussi, mais un peu moins grave.)
- Le parseur généré est lourd, sans aucune élégance, et impossible à modifier. *Et on n'apprend pas beaucoup en construisant un analyseur par une machine automatique*.

La construction des générateurs évolue. Dans notre opinion personnelle, les générateurs sont indispensables si vous êtes un professionnel qui travaille sur la compilation de *plusieurs* langages en utilisant le même langage d'implantation.

Mais si vous, encore débutants, avez envie de faire un jour *un* compilateur, la construction manuelle du parseur va vous prendre moins de temps que la maîtrise et l'usage d'un générateur. Et, en général,...

***Credo* religieux no. 11 :** Le choix préférentiel entre la construction manuelle des parseurs, et l'usage des générateurs de parseurs, appartient au domaine des *credo* religieux.

4.1.4 Qu'est-ce que l'optimisation

Ce sujet sera (peut-être) abordé plus tard, mais quelques notions peuvent êtres utiles au lecteur tout de suite. En particulier il est utile de savoir que

***Credo* religieux no. 12 :** La meilleure stratégie d'optimisation est de ne jamais générer un code mauvais.

Cependant, générer directement le code optimal est extrêmement difficile. Parfois l'analyse globale du programme entier serait utile, mais le compilateur n'a pas de possibilité de mettre dans sa mémoire tout le code d'une grande application, et de plus, l'optimisation globale est *très* lente.

Respecter au pied de la lettre le *credo* no. 12 est difficile. Même si le programmeur fait attention, s'il ne génère jamais de structures inutiles ou redondantes, parfois le compilateur lui-même introduit des inefficacités en développant des macros. Parfois l'écriture d'un code efficace est en contradiction avec sa lisibilité ou sa modularité. On doit éviter l'évaluation multiple des expressions identiques, mais assignation de ces expressions aux variables locales alourdit le programme, donc l'optimisation automatique peut être très utile.

Voici quelques stratégies d'optimisation classiques.

1. Élimination du «code mort», (*dead code*) qui ne sera jamais exécuté, par exemple d'un fragment de code qui se trouve après un branchement obligatoire, et qui n'est pas étiqueté.

   Ceci favorise l'assemblage du code final à partir des morceaux stockées dans des structures dynamiques comme des listes (comme nous l'avons fait en construisant notre machine virtuelle à pile), même si cela alourdit le compilateur.

2. Élimination des sous-expressions communes, et génération des valeurs intermédiaires locales.

3. Pre-évaluation des constantes. L'expression `2*3` peut et doit être évaluée directement par le compilateur, et la valeur 6 insérée dans le code compilé.

   Cette optimisation peut être déclenchée par l'analyseur sémantique. Une constante numérique possède un *attribut* : «constante évaluable». L'application d'un opérateur primitif numérique (fonction prédéfinie, opération standard, etc.) aux objets constants génère un objet avec le même attribut. Ainsi le générateur de code peut «plier» (réduire) les branches de l'arbre syntaxique qui ont été balisées comme constantes.

4. Réduction des opérateurs. Si `x` correspond à une valeur entière, la multiplication par 2 (ou 4, ou 8, etc.), ou la division par une puissance de 2 peut être transformé en décalage des bits. La multiplication `2*x` peut-être transformé en `x+x` si le compilateur sait que l'addition est plus économique que la multiplication pour cette concrète machine-cible.

5. Dépliage des boucles, et autres *inlining* : remplacement des abréviations, comme des appels procéduraux ou des boucles, par le code explicite, répliqué. Ceci rend le code plus long (parfois beaucoup plus long), mais plus rapide.

6. Transformation des appels terminaux en branchements. Si f appelle g et retourne immédiatement après, peut-être il serait avantageux de sauter directement à g, qui retournera au contexte d'appel de f. (Mais ceci exige une gestion délicate des paramètres et des piles-système en général ; le débogage devient parfois inextricable, et la gestion des exceptions – très difficile).

   Pour des langages fonctionnelles comme Haskell (ou Scheme, même si ce dernier n'est pas fonctionnel pur) **ceci est essentiel et obligatoire !**, car il n'y a pas d'autres mécanismes d'itération.

7. Toute sorte d'optimisation de l'allocation des registres rapides (matériels, ou au moins dans des zones mémoire accessibles directement, sans passer par la pile, etc.

   Même pour notre machine virtuelle nous avons pu économiser un peu de temps en prévoyant que le sommet, le dernier élément de la pile n'est stocké qu'en cas de besoin, et normalement il occupe un registre statique (une variable) ; mais ceci est beaucoup plus avantageux pour les langages impératifs, où la notion naturelle de variable statique, modifiable sur place existe.

8. Re-arrangement du code. Parfois la modification de l'ordre d'exécution de quelques instructions permet mieux de sauvegarder et de réutiliser quelques valeurs dans des registres rapides. Ceci est une affaire complexe.

9. Évaluation partielle. Pour évaluer x^n où x et n sont des variables, il faut exécuter l'opérateur «puissance». Mais si le compilateur «sait» que $n = 3$, il peut réduire l'opération, et compiler $x \cdot x \cdot x$.

   (***Le sujet d'évaluation partielle est devenu très important, et mérite une discussion approfondie.***)

Il ne faut pas oublier que parfois l'optimisation du temps d'exécution est en contradiction avec l'économie de la mémoire (l'inlining en est un exemple).

4.2 Intégration d'un compilateur

La «dissection» d'un compilateur n'étant pas terminée, dans cette section nous essayons de discuter les méthodes d'intégration qui font *un* grand programme composé de plusieurs modules : scanneur, parseur, générateur du code, table des symboles, etc. Notre but est de sensibiliser le lecteur au problème de communication entre les parties d'un système de compilation. La richesse actuelle des langages de programmation et la possibilité d'enseigner la compilation à un niveau assez élémentaire, sont des résultats d'une bonne modularisation des compilateurs. Même si les phases d'analyse, synthèse et optimisation se chevauchent, on peut discuter séparément les phases, et on peut montrer comment intégrer le système sans introduire à son intérieur un chaos inextricable.

Question : comment assurer la transmission de l'information entre les étapes? Il serait ridicule d'effectuer d'abord toute l'analyse lexicale, construire un fichier coupé en lexèmes (par exemple : un mot par ligne), ensuite donner à manger ce fichier à l'analyseur syntaxique, etc. Le gaspillage est évident : si l'analyseur syntaxique risque de trouver une faute sur la deuxième ligne, le découpage du texte entier en lexèmes sert à rien.

Il faut assurer une communication *incrémentale*. Mais ici il existe plusieurs stratégies possibles.

4.2.1 Intégration procédurale

La plus classique est la démarche procédurale. Le générateur du code a besoin de la structure intermédiaire, alors il *appelle* le parseur qui lui doit la fournir. Le parseur est une fonction (elle peut s'appeler **`yyparse()`**, s'il s'agit d'un parseur engendré par Yacc), qui parcourt la liste des lexèmes actuellement disponibles.

Mais cette liste, peut-être, elle n'existe pas. Chaque fois quand le parseur a besoin d'un nouveau mot, il appelle le scanneur (la fonction **`yylex()`**), et ce dernier s'occupe de la lecture de la source. Cette démarche est la mieux connue, et les générateurs Lex et Yacc génèrent les modules adaptés à une telle technique. Les compilateurs traditionnels de Pascal (notamment les premiers compilateurs conçus par le créateur du langage, Niclaus Wirth, et codés en Pascal), l'exploitent aussi. Elle est également (ou mieux) bien adaptée à la construction de parseurs «manuels» descendants, car elle est intuitive, et correspond au style traditionnel d'apprentissage des langages de programmation.

L'intégration procédurale introduit naturellement des dépendances fonctionnelles entre les modules, et diminue ainsi la modularité. Le déboguage peut ne pas être facile. La modification du compilateur après avoir introduit des extensions dans le langage est d'habitude assez pénible, il faut vérifier tout.

4.2.2 Transducteurs de flux, ou «pipelining»

Nous avons souligné que la création d'un fichier intermédiaire peut ne pas être très économique. Il faut alors simplement – au lieu d'utiliser un fichier disque, faire passer l'information par un *flux* dynamique, par exemple

par un «pipe» Unix. On peut également utiliser des *listes paresseuses* qui sont réalisées différemment des «pipes», mais qui offrent des fonctionnalités analogiques, et qui sont très intensivement exploitées dans le domaine de la programmation fonctionnelle. (Par exemple les parseurs typiques en Caml utilisent des flux). Nous allons en profiter aussi, les listes en Haskell sont naturellement paresseuses.

Attention ! Les listes paresseuses qui simulent les *pipes* offrent à l'utilisateur la possibilité de traiter les fichiers de longueur quelconque comme des chaînes de caractères. Ceci n'est pas vraiment conseillé à tout le monde. Rappelons qu'en Hugs, l'implantation de Haskell avec laquelle nous travaillons, il faut écrire

```
import IOExts
...
texte = unsafePerformIO (readFile nom du fichier)
```

en important d'abord le module d'extensions **`IOexts`** comme ci-dessus. Rappelons que si le fichier est long, il faut *obligatoirement* écrire le programme de manière à ce qu'il consomme la liste de caractères itérativement, et oublie définitivement les segments contenant les caractères lus. Le compilateur doit être capable de le prouver formellement, sinon le flux se transforme peu à peu en une liste réelle, et finit par avaler toute la mémoire disponible. Il y a d'autres techniques d'interfaçage, mais traiter le contenu d'un fichier comme une chaîne quelconque est utile pour tester les programmes courts.

La dynamique de ce transfert d'information peut être visualisée de manière suivante : le scanneur génère une suite de lexèmes, et les injecte dans le flux. Mais le buffer du flux est très court et il se remplit immédiatement. À ce moment là, le scanneur est bloqué, et le consommateur du flux commence son travail, en transformant les lexèmes en arbres syntaxiques, etc. Quand le flux est épuisé, le parseur se bloque, et le scanneur redémarre.

Les deux modules travaillent donc en parallèle, ou plutôt quasi-parallèle, en temps partagé. On écrit les deux modules séparément, ils ne se communiquent pas directement, mais ils collaborent comme deux partenaires dans un jeu, en se «renvoyant la balle» : l'information concernant l'état du flux intermédiaire. Les deux modules ne sont plus des procédures qui s'appellent, mais des *co-procédures*.

4.3 Organisation de la table des symboles

Le dictionnaire de symboles est le cœur du compilateur, il est partagé par *tous* ses modules. Il contient les références aux chaînes et les attributs des symboles. Si le langage – comme presque tous – possède une structure de blocs lexicaux ou de fermetures, c'est à dire, permet la définition de variables locales (ou paramètres) à plusieurs niveaux, la table des symboles reflète la hiérarchie des blocs en train d'être analysés. La récursivité du parsing correspond à la structure arborescente du dictionnaire des symboles. Le terme «table des symboles» ne correspond pas à sa véritable structure…

Notre attitude vis-à-vis la table des symboles est un peu cavalière, car ses fonctions sont très simples, même si structurellement elle peut devenir une toile d'araignée. Nous construisons des listes d'associations, représentons les noms (mots-clés et identificateurs) par les chaînes de caractères, et, en général, le problème d'efficacité de stockage ne nous concerne vraiment pas. Cependant ceci *est* un problème sérieux.

La mémoire consacrée au stockage des données peut être morcelée en plusieurs milliers de segments sans pertes, à condition que tous ces segments occupent une zone contiguë, et que leurs adresses soient résolues statiquement. Mais si l'ensemble bouge, si allocations dynamiques sont fréquentes, si quelques segments deviennent inutilisables et doivent être retournés au *pool* géré par le système, cette gestion est compliquée, avec plusieurs pointeurs sur les pointeurs, ce qui est toujours un gaspillage. Il faut alors au moins utiliser l'allocation simple et statique partout où ceci est possible. On ne doit pas représenter les noms par les chaînes, mais plutôt réserver un long buffer, un tableau 1-dimensionnel, et y placer les chaînes de façon contiguë. Chaque objet placé dans le buffer est identifié par son indice et sa longueur, comme sur la Fig. (4.1).
Quand l'analyseur lexical identifie une nouvelle chaîne, il la met dans le buffer, et il construit une nouvelle entrée dans le tableau des paires (adresse,longueur). Les indices dans ce tableau sont les *seules* références de la chaîne dans le dictionnaire. Le dictionnaire lui même peut être le «prolongement horizontal» du tableau **`chtab`**, et les colonnes suivantes contiennent les attributs du symbole. Cependant, en général un symbole peut signifier plusieurs choses en dehors et à l'intérieur d'un bloc, ou si un langage prévoit plusieurs espaces de noms (p. ex. les identificateurs déclarés comme **`static`** dans un fichier contenant un programme en C).

Le dictionnaire est alors un autre structure de données, où le symbole (la référence d'un élément du tableau **`chtab`**, et non pas la chaîne) est l'attribut «nom» d'un objet lexical. Cette stratégie permet aussi de traiter de manière homogène les symboles qui dénotent les identificateurs du programme et les mots-clés. L'analyseur

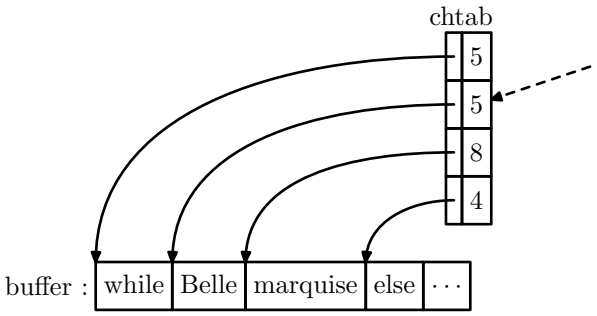


Fig. 4.1: Table de chaînes

lexical ne voit pas de différence entre eux, mais les mots clés (et quelques identificateurs standard) sont prédéfinis, et occupent une autre table dans le compilateur.

Le temps de vie et la portée des attributs sont très différents. Après l'analyse lexicale le compilateur peut jeter le buffer des chaînes, sauf s'il veut garder les noms littéraux pour le déboguage, ou si les programmes sont compilés par fragments, et le compilateur doit préserver l'information lexicale concernant les symboles exportés (globaux) pour l'éditeur des liens.

La valeur d'une constante est construite par l'analyseur lexical, et consommée finalement par le générateur de code qui génère l'instruction de chargement(`load`...). Le parseur peut totalement ignorer cet attribut. Mais si le compilateur optimise le code, la propriété «être constant» peut se propager depuis des constantes numériques vers les expressions sans variables, et le parseur peut déclencher une pre-évaluation sans invoquer le générateur. Dans ce cas il aura besoin de la valeur. Pour éviter trop d'ambiguïté, d'habitude l'optimisation est une phase séparée, liée plutôt au générateur du code, qu'à l'analyse, mais on voit clairement que la gestion du dictionnaire des symboles peut être compliquée. Accessoirement, quelques attributs, comme le type des données caractérisent pas seulement les symboles, mais les expressions entières, qui deviennent les «arbres décorés». Ceci montre que les attributs ne doivent pas être statiquement représentés comme des positions de la table des symboles, mais ajoutés selon les besoins comme des champs dynamiques.

La question vitale pour l'organisation de ce dictionnaire est la vitesse de recherche et/ou de l'insertion. On peut utiliser la recherche dichotomique dans un tableau trié, mais actuellement l'approche connue comme le *hachage* domine.

4.3.1 Techniques de hachage

L'idée générale du hachage ou du *hash-coding* consiste à transformer une chaîne de caractères par une manipulation locale et rapide en *indice*, permettant ainsi la recherche d'un atome sans être obligé de parcourir le dictionnaire (ou le tableau `chtab`) entier.

Tout objet informatique peut être considéré comme un nombre entier, puisque tout objet est une séquence de bits. Mais un tel nombre d'habitude est très long, et les tableaux de stockage des symboles doivent être raisonnables. La stratégie de hachage consiste donc à transformer un entier très long en court. Une possible solution serait de calculer ce nombre **modulo** N, où N est la taille du tableau indexé par les symboles «hachés». La division entière (Euclidéenne) est raisonnable et utilisée, mais elle est relativement lente, donc parfois on construit une somme pondérée des codes : $S = \sum_{k=0}^{n} \alpha C_k + \beta$, où C_k est le code du caractère, et n – la longueur de la chaîne. On peut utiliser le *exclusive or*, ou autres manipulations itératives. Leur propriété commune est de repartir de manière la plus homogène les indices résultants sur l'espace disponible. En effet, la fonction de hachage doit se comporter comme un générateur de nombres aléatoires repartis uniformément.

Une stratégie exploitée dans le compilateur de P. Weinberger est toujours réputée comme bonne : un nombre entier `h` initialisé à zéro est modifié dans une boucle par :

```
h = (h << 4) + ch.suivant;
```

mais si le nombre devient trop grand, on effectue un décalage des bits à droite (de 24 positions), et on applique l'opération `xor` entre le nombre et le résultat de ce dernier décalage. Les bits à gauche sont nettoyés. Finalement on calcule le reste de la division Euclidéenne par un nombre premier – la taille du tableau. Cette technique est mentionnée pour l'orientation générale du lecteur, et non pas comme une recette de cuisine.

Indépendamment du choix de la fonction de hachage, il y aura des *collisions* – deux symboles différents auront le même code. La fonction de hachage n'a aucune chance d'être réversible, car le nombre de symboles possibles est très grand, et le tableau de stockage est limité. Il existe deux catégories de solutions de cette difficulté :

- On choisit le premier emplacement libre dans le tableau, ou on répète le hachage paramétré par le dernier indice. (Ou on utilise une variante similaire, p. ex. on ajoute à l'indice haché une constante première par rapport à la taille du tableau, modulo cette taille. (Le nombre 1 est une possibilité). Si le tableau possède une case vide, elle sera trouvée. Quand le tableau est presque plein, l'efficacité de recherche diminue considérablement.
- Chaque élément du tableau contient une liste chaînée des symboles en collision. La liste peut être parcourue linéairement, ou par un moyen plus efficace, mais ces listes doivent être courtes. Si la fonction de hachage est bien choisie, et si le programmeur n'a pas choisi des noms très bizarres, toutes les listes auront (statistiquement) la même longueur. Si la longueur du tableau est 1000, la longueur de chaque liste sera d'ordre $N/1000$, où N est le nombre de symboles.

4.4 Exercices

Q1. Listez au moins douze conventions différentes de représenter les commentaires dans les programmes.

R1. Vous avez cherché vous même, n'est-ce pas?...

- `(* ... *)` ou `{ ... }` en Pascal.
- `/* ... */` en C.
- En C++ le précédent, ou `// ...` jusqu'à la fin de ligne. Le même style est utilisé aussi en Clean, et pour écrire les *shaders* Renderman..
- Les deux tirets `-- ...` jusqu'à la fin de ligne en Ada et Haskell. Mais ce dernier utilise aussi `{- ... -}`, qui peuvent être imbriqués.
- Le dièse : # en *shell*, Makefiles, Python, Tcl et Perl. Et aussi VRML, et fichiers RIB (Renderman).
- Le pourcent : % en Matlab, TEX, MetaPost, quelques implantation de Prolog, PostScript, etc.
- Le point-virgule `; ...` jusqu'à la fin de ligne en Scheme. (Aussi dans quelques assembleurs)
- Et en CAML? Comme en Pascal.
- Fortran? Le caractère "C" en première colonne.
- `<!-- ... -->` en HTML, etc.
- REM ou l'apostrophe en Basic.
- POVray? `//`.

On n'utilise plus le langage Algol 60 où le commentaire suivait le mot-clé **comment**, ou Algol 68 où le caractère «cent» était de rigueur. Mais on utilise toujours Lisp, et les premières versions prévoyaient des constructions de genre `(comment Belle marquise n'importe quoi, et encore)` dont la valeur retournée était NIL indépendamment de la couleur des yeux de la belle marquise.

Q2. Question accessoire qui ne concerne *directement* pas les compilateurs, mais importante pour le support d'exécution du code compilé : Comment réaliser les **files** (Structures FIFO) dans un langage fonctionnel? Comment réaliser fonctionnellement le parcours des arborescences en largeur?

R2. L'importance de ce problème doit être évidente. Les *pipes* sont des files ! La programmation événementielle ou pseudo-parallèle, les threads en ont besoin aussi.

Il faudra construire au moins deux fonctions, l'enfilement qui à partir d'un objet et une file construit une autre, et le défilement qui renvoie une paire : l'élément récupéré et la file restante. Une possibilité est l'usage des listes :

```
enfiler x q = q ++ [x]
defiler (x:q) = (x,q)
```

mais l'usage de la concaténation pour ajouter un élément est très inefficace (rappelons que la concaténation recopie son premier argument). En général, sans la possibilité de *modifier* des structures de données la situation semble désespérée, toute modification est obligée de reconstruire la nouvelle file sans abîmer la précédente, et pendant la compilation d'un grand programme les structures de données dans le compilateur sont assez volatiles.

D'habitude un programme – fonctionnel ou pas – n'utilise jamais en même temps une vieille structure : (pile, file ou autre chose) et la nouvelle. Si le programme est considéré comme l'enchaînement des opérations qui passent les structures de données construites à ses continuations, il est possible d'éliminer une bonne partie d'inefficacité. La technique ressemble beaucoup à l'optimisation canonique de la procédure qui renverse une liste à l'aide d'une variable-tampon (mais, ce qui est curieux, cette optimisation appartient aux «canons» de la programmation, et pourtant les files fonctionnelles sont très rarement enseignées...)

Une file **xs** est représentée par une *paire* de listes, **(ys,zs)**, telles que (conceptuellement !) **xs = ys ++ (reverse zs)**. On ajoute toujours le nouvel élément à la tête de **ys**, et on récupère le plus ancien du début la liste **zs** (alors, conceptuellement de la fin de **xs**). Quand la liste **zs** devient vide, on renverse **ys** ot on la substitue pour **zs**. Voici les fonctions de base

```
enfl x (ys,zs) = (x:ys,zs)
defl (ys,(z:zq)) = (z, (ys,zq))
defl (ys,[]) = defl ([],reverse ys)
```

De temps en temps le programme sera obligé à dépenser n unités de temps pour renverser la liste, où n est la taille moyenne de la file, mais l'efficacité globale de cet algorithme est raisonnable, la complexité moyenne est constante par un élément inséré ou enlevé (on l'appelle la *complexité amortie*.

Le parcours en largeur consiste à enfiler l'arbre, et itérer la manipulation suivante, avec la file comme argument : si la file est vide, le résultat est une liste vide ; sinon, défiler la racine. Si c'est une feuille, la mettre à la tête de la liste résultante. La queue est le résultat de l'aplâtissement de la file restante. Si la racine est un nœud intermédiaire, placer l'étiquette devant le résultat de l'aplâtissement de la file restante enrichie par l'enfilement de la branches gauche et droite.

```
data Arbr = F Int | N Int Arbr Arbr
type Lst a = [a]
data Queue a = Q (Lst a) (Lst a)
x= N 1 (N 2 (F 4) (F 5)) (N 3 (N 6 (F 8) (F 9)) (F 7))
enfl x (Q a b) = Q (x:a) b
defl (Q a (z:zq)) = (z, (Q a zq))
defl (Q a []) = defl (Q [] (reverse a))
flat ar = fl (enfl ar (Q [] [])) where
   fl (Q [][]) = []
   fl q = let (z,q1)=defl q in
          case z of (F i)     -> i : fl q1
                    (N i g d) -> i : fl (enfl d (enfl g q1))
r=flat x    -- donne [1 2 3 4 5 6 7 8 9]
```

La fonction n'est pas récursive terminale. Laissons au lecteur la tâche d'optimiser cet algorithme. Éliminer tout de suite la fonction **enfl**. Essayer d'incorporer également **defl** dans **fl**. Essayer d'évaluer la complexité de cet algorithme. Voir aussi l'annexe, section (B.6).

Q3. Comment stocker sur un fichier séquentiel les arborescences? Et les graphes quelconques, possiblement cycliques?

R3. La représentation séquentielle des arbres est bien connue des lecteurs : ce sont des listes (Lisp, Prolog etc.) Il suffit d'avoir les parenthèses qui jouent le rôle d'opérateurs : empiler/dépiler. **Cet exercice est important, il suggère comment peut marcher un simple parseur qui analyse les listes !** Comparer

cette stratégie avec la digression dans un de chapitres précédents, qui décrit la classe **Show** (procédures d'affichage en Haskell).

Les graphes acycliques sont équivalents aux arborescences avec duplication des nœuds partagés, et ils n'ont pas besoin d'une autre stratégie, mais naturellement pour l'efficacité il est envisageable de ne pas proliférer les expressions partagées. À l'envers, la réduction d'un arbre à un DAG est toujours souhaitée. Dans ce cas, *et même dans le cas cyclique* il suffit de stocker le graphe sous forme indirecte : choisir un ensemble de symboles spéciaux, p. ex. **#1**, **#2**, etc. et d'associer à chaque symbole son graphe, qui peut contenir à l'intérieur des occurrences de ces symboles.

Une autre stratégie consiste à «atomiser» la précédente. Chaque sommet du graphe aura son étiquette. Le fichier contient la liste des étiquettes suivie de la liste des arcs : paires d'étiquettes. On stocke un graphe représenté par sa matrice d'incidence.

Q4. Connaissez-vous la technique permettant de *simuler les structures de données par des fonctions* **pures** *(formes lambda?)*

Construire une fonction **cons** qui s'applique à deux objets, disons **x** et **y**, et qui construit un objet opaque, intraitable par quoi que ce soit, sauf par deux fonctions qui s'appellent (oui, vous avez déviné…) **car** et **cdr**. L'application du **car** à cet objet récupère **x**, et du **cdr** – **y**. On n'a aucun droit d'utiliser un constructeur de données, et les arguments de **cons** ne subissent aucune manipulation.

R4. Vous renoncez si vite? Tant pis. Voici la solution.

```
cons x y = trouNoir
  where trouNoir arg | arg=="car"    = x
                     | arg=="cdr"    = y
                     | otherwise = error "argument illégal"
car z = z "car"
cdr z = z "cdr"
```

Question accessoire pour déstabiliser les ambitieux. On a triché ici ! La fonction **trouNoir** n'est pas une «fonction pure», mais une fermeture qui profite des **x** et **y** comme des variables non-locales. Peut-on résoudre cet exercice sans variables non-locales? Ceci est un excellent sujet d'examen, mais soyons humains. Voici la solution, complètement triviale :

```
cons a b arg | arg=="car"    = a
             | arg=="cdr"    = b
             | otherwise = error "argument illégal"
```

En fait, on peut toujours remplacer un module fonctionnel avec des variables globales, par une fermeture réalisée par une application partielle. Cette transformation s'appelle «*lambda-lifting*» et appartient à l'abécédaire de la compilation des programmes fonctionnels.

Cet exercice figure dans nos notes ailleurs. trouvez-le ! Comparez l'autre solution avec celle là, discuté les différences concernant le typage !

Chapitre 5

Analyse syntaxique I – Techniques fonctionnelles

5.1 Grammaires et *parsing*

Ce cours exige et suppose une raisonnable connaissance de la théorie des grammaires et de la notation BNF (*Backus-Naur Form*). Notre notation sera classique, pour un langage non-contextuel, une production syntaxique aura (symboliquement) la forme

```
NonTerminal ::= UneChose  UneSéquence | AutreSéquence (Groupe composite)
```

etc., où à droite on a une séquence arbitraire des variables syntaxiques (objets nonterminaux), et de littéraux (terminaux). Les méta-caractères utilisés ici sont

- La barre verticale – l'alternative.
- Les parenthèses (pour le groupement), et les crochets (pour les regexps).
- Les apostrophes.
- l'assignation `::=` (plutôt que souvent utilisée ←).

Tous les autres caractères visibles, notamment les virgules, etc. sont des littéraux. Pour noter littéralement un méta-caractère, par exemple `'('` on le mettra entre apostrophes. Un apostrophe littéral s'écrit `\'`, et les symboles `\s`, `\n`, et `\t` dénotent l'espace, le saut de ligne et la tabulation. Pour distinguer les mots littéraux des variables syntaxiques, et ne pas encombrer les productions avec trop d'apostrophes, nous utiliserons la police proportionnelle pour des textes terminaux.

5.1.1 Exemple

Essayons d'écrire la grammaire définissant (de manière incomplète !) un programme en langage Prolog. Le lecteur doit se rappeler les exemples montrés dans des sections précédentes. Une liste ressemble à son homologue en Haskell, seulement au lieu d'écrire `(x:y)` la syntaxe Prolog est `[x|y]`.

Un programme (en fait : la définition d'un prédicat) en Prolog est une séquence de clauses, chaque clause possède l'entête, un «corps» qui peut être vide, et se termine par le point final. Le corps commence par l'opérateur d'inférence `:-`, suivi d'une séquence de prédicats, dont la structure est la même que celle du prédicat constituant l'entête. La syntaxe d'un prédicat est exactement la même, que la structure d'une donnée générale Prolog – le *terme*, ou plus spécifiquement : le terme atomique. Les termes dans la séquence peuvent être séparés par des virgules, ce qui dénote la conjonction («et») logique, ou les point-virgules, qui construisent l'alternative. La précédence de la virgule est plus grande. On peut utiliser les parenthèses pour le groupement, et il ne faut pas confondre les parenthèses et les méta-parenthèses.

```
Programme  ::= Entete Corps .
Entete     ::= TermeAtome
Corps      ::= ϕ | :- AltTerme
```

```
AltTerme   ::= SeqTerme | SeqTerme ; AltTerme
SeqTerme   ::= AtomeLog (φ | , SeqTerme)
AtomeLog   ::= TermeAtome | '(' AltTerme ')'
```

où naturellement ϕ dénote la chaîne vide. Notez la factorisation du préfixe **`TermeAtome`** dans la production décrivant **`SeqTerme`** ; la ligne précédente contient une construction structurellement identique, mais développée.

Passons aux données. Les termes Prolog sont des expressions qui peuvent être atomiques au sens : identificateurs ou nombres (***et ceci n'est pas la même chose que l'atomicité logique, l'absence des connecteurs***), fonctionnelles : **`f(x,2*y)`**, etc., contenir les opérateurs arithmétiques **ou autres opérateurs infixes que ne seront pas discutés ici**, et les listes.

```
TermeAtome ::= Terme
Terme      ::= AExpr | Liste | Tfunc | Atomic
Atomic     ::= Symbol | Number
Tfunc      ::= Symbol '(' Seq ')'
Seq        ::= φ | SeqTrm
SeqTrm     ::= Terme (φ | , SeqTrm)
Liste      ::= '[' Seq Queue ']'
Queue      ::= φ | '|' Terme
AExpr      ::= Atrm | Aexpr OpAdd Atrm
OpAdd      ::= + | -
Atrm       ::= Factor | Atrm OpMul Factor
OpMul      ::= * | /
Factor     ::= Primary | Primary ^ Factor
Primary    ::= Atomic | Tfunc | '(' Aexpr ')'
```

où ces définitions implicitement déclarent les quatre opérations arithmétiques comme associatives à gauche, et la puissance **`^`** comme associative à droite. Les atomes ne présentent pas trop de problèmes :

```
Number  ::= Sgn (Integ  | Float)
Integ   ::= Digit  | Digit Integ
Float   ::= Integ . Integ OptExp
OptExp  ::= φ | E Sgn Integ
Sgn     ::= φ | + | -
Digit   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Symbol  ::= Letter (φ | AlnSeq)
AlnSeq  ::= Alnum | AlnSeq Alnum
Alnum   ::= Letter | Digit
```

… et les lettres? Ah, non, écrire une clause avec 52 alternatives, sans compter les lettres accentuées, est un peu trop. On voit que l'approche *strictement* syntaxique, «extensionnelle», à la définition du langage a ses handicaps. Bien sûr, la notation BNF n'épuise pas la présentation des grammaires, et tous les lecteurs connaissent (ou doivent connaître !) les expressions régulières avec des règles spéciales *ad hoc*, par exemple

```
Integ   ::= Digit+
Letter  ::= [A-Za-z]
```

etc. La définition de **`Integ`** montre que la fermeture de Kleene (positive) est une simple abréviation, qui d'ailleurs implicitement contient l'associativité de la concaténation. La forme BNF peut être dans ce cas récursive à gauche ou à droite, selon la philosophie d'implantation. La définition de **`Letter`** est une définition syntaxiquement incomplète, elle assume que l'alphabet est figé et bien connu.

Rappelons-nous donc de la proposition introduite dans la section précédente : si les définitions des objets irréductibles utilisent obligatoirement une information sémantique extérieure, que cette information soit au moins souple. On peut donc définir une lettre comme caractère qui possède l'attribut «sémantique» «*lettre*». Il faut alors considérer que l'analyseur des lettres (quelconques) soit primitif. Ceci n'est pas possible directement en Lex, le seul moyen de procéder serait d'accepter un caractère quelconque, et procéder à sa reconnaissance dynamique par une procédure sémantique, ce qui est pénible, pratiquement inutilisable.

La notation classique aura aussi quelques difficultés pour décrire des opérateurs de précédence quelconque, ajoutés éventuellement dans le programme – le programme qui sera compilé et exécuté, et non pas dans l'analyseur. Ceci est possible en Haskell, ML, Prolog, etc. Partiellement pour cette raisons nous ne pouvons donner la définition complète du Prolog, cette définition aurait un niveau méta- trop important, et pas très lisible. Ce problème suggère fort aussi qu'une bonne partie de ce que nous appelons la sémantique, est en fait la syntaxe, mais avec des dépendances contextuelles importantes, et de longue portée. Pour qu'en Haskell l'expression `(x+1 <=> x-1)` soit légale, l'objet lexical `<=>` doit figurer dans une clause `infix` ou équivalente, et ainsi devenir un objet *syntaxique*. En Prolog c'est pareil, les opérateurs (préfixes, infixes et postfixes) doivent être déclarés par le prédicat primitif `op`.

En C toute occurrence d'une variable doit être accompagnée par sa déclaration, sinon ceci est une erreur *sémantique*, mais la déclaration elle-même est une simple structure syntaxique. Les exemples de ce type sont très nombreux. Bref, nous ne savons pas vraiment comment définir sans aucune ambiguïté le mot «sémantique» dans le domaine de la compilation, car le «sens» est souvent caché dans la *structure*...

La notation BNF standard manque un outil de reconnaissance parfois inestimable – la négation. Comment décrire un commentaire en C? Toute chaîne qui commence par `/*` et qui se termine par `*/`, mais avec la restriction de *ne pas* contenir la chaîne `*/` à l'intérieur. La construction d'un caractère «non-slash» par énumération : une alternative de tous les autres, est théoriquement correcte, mais pas pratique du tout. On peut construire une expression régulière correspondante, mais les regexps ne sont que des abréviations. Dynamiquement on procède de manière suivante : quand on trouve la première occurrence de `*/`, on s'arrête. Ceci est la solution du problème des commentaires, mais la négation est un concept plus général.

En général l'introduction de la négation dans une grammaire *est* possible, et le parseur correspondant, quand il trouve l'expression sous la négation, il génère un échec. Ceci – si on utilise une stratégie non-déterministe – provoque le*backtracking*, et la recherche d'une autre alternative. Par contre, quand ce parseur trouve le contexte qui *ne correspond pas* à l'expression sous la négation, l'échec se transforme en succès.

5.2 Stratégies du parsing

Par convention les techniques d'analyse syntaxique se divisent en deux grandes catégories :

- Techniques *descendantes*, et
- techniques *ascendantes*.

Si conceptuellement le rôle d'analyseur est de construir l'arbre syntaxique à partir de la représentation linéaire, textuelle, on peut construir cet arbre à partir de la racine, en ajoutant les branches et terminant le processus au niveau des feuilles, ou à l'envers, commencer par les feuilles, lier les feuilles par les branches, et terminer par la racine.

Les deux stratégies sont complémentaires. La stratégie ascendante est plus universelle, et peut être plus efficace, mais elle n'est pas très facile à implanter, et souvent beaucoup plus difficile à comprendre. Nous allons traiter en priorité la technique descendante, plus propice à la construction manuelle des analyseurs. Les parseurs générés par Yacc ou autres générateurs de ce type exploitent d'habitude la stratégie LR(1), ascendante.

5.2.1 Stratégie descendante

Dans la stratégie descendante un *parseur* peut être apparié à un *non-terminal*. Nous pouvons imaginer que le parseur est une fonction qui s'applique à un flux de données d'entrée, par exemple les caractères, et qui produit une arborescence. Le symbole non-terminal principal (le symbole de départ) de la grammaire entame l'analyse. Par exemple le parseur `Program` s'applique au flux d'entrée et construit le code (pas forcément final) du programme.

Si la production qui définit ce non-terminal utilise autres non-terminaux :

```
Program ::= Entete Corps .
```

la procédure `program` appelle les procédures `entete` et `corps`, qui appellent leurs «esclaves», etc. Bien sûr, les appels qui s'enchaînent séquentiellement : Le Corps doit suivre l'Entête – doivent consommer séquentiellement le flux d'entrée. En fait, si la grammaire est écrite, la construction d'un parseur descendant qui correspond à un non-terminal est relativement claire – elle correspond à la production qui définit ce non-terminal. Le programmeur doit assurer à peine deux choses :

- passer correctement le flux d'entrée d'un fragment à l'autre, et
- construir un résultat du parsing.

Nous verrons que la construction d'un parseur descendant à partir d'une grammaire, peut être une procédure presque méchanique. La complexité interne est gérée par les appels récursifs. On continue à descendre, et finalement un parseur primitif comme `lettre` consomme un caractère qui constitue la feuille d'une branche correspondant à un identificateur, etc. En remontant l'arbre des appels récursifs, l'arbre du parsing est construit physiquement, ou le parseur le linéarise directement, et engendre le code postfixe. Mais ceci constitue une manipulation sémantique, et ici le procédé est beaucoup moins automatique. Il est partiellement régularisé grâce au concept des *attributs* dont nous allons encore parler.

5.2.2 Techniques ascendantes

Un parseur ascendant n'utilise pas des appels récursifs, mais il est piloté par un nombre de structures de données spécifiques au langage – tableaux, piles, etc. qui contiennent des informations sur les enchaînements légaux des symboles. Si l'analyseur – par exemple – est en train d'analyser les expressions, et il se trouve devant la chaîne "`a*(1/x ...)+2...`, il trouve le premier lexème – l'atome `a`. Selon la grammaire, cet atome peut être réduit à un `Facteur`, mais pas tout de suite, car s'il est suivi par la parenthèse ouvrante, il fera partie d'un appel fonctionnel. Ceci n'est pas le cas dans l'exemple ci-dessus. Faut-il réduir le `Facteur` à un `Terme`? La réponse est *non*, le lexème suivant, l'opérateur de multiplication, suggère que si un `Terme` est formé ici, il sera composé de plusieurs Facteurs multipliés ensemble.

Ensuite le perseur trouve la parenthèse ouvrante, et il *sait* que ceci sera une expression primaire, parenthésée. Quand le parseur trouve l'opérateur additif `(+)`, il sait que l'expression sera composée de plusieurs termes, dont le premier vient d'être assemblé.

Pour que la stratégie fonctionne, le parseur doit pouvoir répondre à la question : *quelle production utiliser* pour la réduction éventuelle? On ne peut faire cela à l'aveugle, car la complexité de l'algorithme serait exorbitante. Il faut donner au parseur les moyens de «pilotage» – la possibilité de prendre de bonnes décisions sans connaître trop de choses sur la partie du flux d'entrée qui n'a pas été encore découverte. En particulier, les tableaux de pilotage répondent à la question : faut il consommer et stocker encore quelques items (effectuer l'opération **shift**), ou réduire quelques feuilles et branches déjà formées de l'arbre syntaxique, pour construir un arbre plus grand, et s'approcher ainsi de la racine (opération **reduce**).

Les détails seront précisés plus tard, à présent passons à la réalisation des parseurs descendants, en utilisant les stratégies *fonctionnelles* de codage. Nous verrons d'ailleurs, que notre stratégie ne sera pas complètement descendante, mais mixte.

5.3 Philosophie du parsing fonctionnel

Cette section montre comment construire les parseurs (surtout les parseurs descendants) dans un langage fonctionnel, par *composition*. Le but principal de cette stratégie est de construire les parseurs utilisables, relativement efficaces et économiques, de manière statique, déterminée par la syntaxe du langage. Nos parseurs ne seront pas des simples automates de reconnaissance, mais ils généreront le «code», les objets de sortie, on devra donc les équiper de procédures sémantiques. Chaque parseur sera donc un petit compilateur.

De plus – comme nous le verrons très tôt – nous envisageons de construire des parseurs *universels*, qui s'adaptent facilement à des structures syntaxiques fréquemment trouvées dans les langages de programmation traditionnels.

Pour l'instant nous ne distinguons pas entre l'analyse lexicale et phrasale – tout appartient à l'analyse syntaxique. Dans les deux cas les flux sont différents : flux de caractères, ou flux de mots ; les objets de sortie également : les mots ou les arbres. Cependant les méthodes de combinaison restent les mêmes, et une partie de l'analyse phrasale qui n'a pas besoin de récursivité, peut profiter des compositions itératives qui caractérisent les expressions régulières utilisées dans l'analyse lexicale. (Exemple : formation des listes ou blocs d'instructions demande uniquement la récursivité linéaire (et codage itératif).)

Ainsi l'exemple de la syntaxe de Prolog cache sans commentaires ces deux niveaux différents : lexical et syntaxique (phrasal). Pendant l'analyse lexicale on doit se poser la question : «que fait-on avec les espaces et les sauts de ligne?». Pour l'analyse phrasale ces objets n'existent plus (sauf si le *layout*, la fin de ligne et

l'indentation jouent un rôle sémantique ou syntaxique active, et sont stockés avec les symboles pour faciliter le déboguage).

5.3.1 Qu'est-ce qu'un parseur?

Commençons par une description simplifié, qui sera vite enrichie et complétée. Le contexte du parsing contient un flux d'entrée – les caractères pour le scanneur, ou déjà les lexèmes pour le parseur phrasal. On peut imaginer donc que le parseur est une fonction de type (*flux* → *objet*), mais nos parseurs sont plus localisés, ils ne sont pas obligés de consommer entièrement le flux. Donc, le flux restant fera partie de la valeur construite par l'analyseur, et correspond à l'«état» du système. Il faut le préserver pour que le successeur d'un module parseur puisse consommer le segment suivant. N'oubliez pas que notre construction est purement fonctionnelle, et que rien, et surtout pas le flot d'entrée, n'est caché «sous la moquette». Il n'y a pas de procédure «**scanf**».

Même si le langage et le parseur soient globalement parfaitement déterministes, un peu de non-déterminisme local est toujours présent, au moins intuitivement : quand on commence à consommer les chiffres, on ne sait pas encore si le résultat sera un nombre entier ou flottant. Parfois le parseur échoue et il faut annuler une décision précoce (faire le retour en arrière, ou *backtracking*). L'élimination totale du non-déterminisme est souvent possible, mais ceci sera discuté plus tard. Les parseurs parfaits n'ont pas beaucoup de valeurs pédagogiques, car ils sont trop compliqués. Il ne faut pas avoir peur de la situation où le parseur peut donner deux ou plusieurs réponses différentes (ambiguës), elles seront stockées dans une liste, et discriminées plus tard, selon le contexte.

Nous proposons alors l'introduction d'un type de parseurs universels, qui peuvent être spécifiés de cette manière :

```
type UParser c a = [c] -> [(a,[c])]
```

où **c** est le type des items sur le flux d'entrée (pensez que ce sont des caractères), et **a** – le type des objets de sortie (pensez aux lexèmes, ou aux arbres syntaxiques). Les flux seront réalisés par les listes, puisque les chaînes de caractères dans Haskell standard ne sont que des listes.

Attention !! La vraie définition du **UParser** introduite dans (5.4.1) sera un peu différente, nous allons baliser cette fonction.

Le parseur universel peut consommer un flux quelconque, et fournir un résultat quelconque. Il peut consommer un flux d'arbres syntaxiques, et créer un code postfixe, il peut donc aussi représenter un générateur de code piloté par une grammaire.

Dans la pratique une telle généralité sera nécessaire uniquement pour définir les combinateurs internes. Plus concrètement, nous aurons besoin de :

- un «tailleur» (**Cutter**) qui coupe un morceau *atomique* du flux et le fournit comme valeur. Par exemple – le premier caractère d'une chaîne. Bien sûr, ceci implique que le type **a** est égal à **c** ;
- des parseurs génériques adaptés à traiter les chaînes de caractères, pour pouvoir rapidement tester l'approche. Par exemple, **CScanner** parcourt un flux de caractères, et fournit un caractère, et **Scanner** parcourt une chaîne de caractères et renvoi un «<mot», aussi une chaîne. Plus tard nous construirons un parseur qui prend un flux d'atomes et qui en construit une arborescence.

5.3.2 Objectifs finaux

La grammaire du langage possède toujours un symbole non-terminal de départ, qui sera la racine de l'arborescence (physique ou conceptuelle) conçue par le parseur. Ce symbole peut être le **Programme**. Il existe une production qui *définit* ce programme, par exemple

```
Programme  ::== Déclarations Instructions
```

(ou dans d'autres langages : ensemble des clauses définissant les prédicats en Prolog, ou les définitions des classes et leurs méthodes en Java, etc.

Le compilateur lance alors (par exemple)

```
...                -- décl. initiales
code = Programme fluxDentrée
```

```
...
fluxSortie = transforme code
```

où la dernière instruction doit assurer la génération physique du code final. Le parseur **`Programme`** est exécuté, et appelle ses parseurs subordonnés, par l'exemple le parseur **`instruction`**, qui appelle le parseur **`variable`** à gauche, et le parseur **`expression`** à droite. Ainsi l'arbre syntaxique est bâti à partir de la racine et pour cela on appelle cette stratégie du parseur : *descendante*. (Comme tout le monde le sait, si un informaticien se trouve près de la racine d'un arbre, il descend pour arriver aux feuilles. Les gens normaux montent. Ainsi on a prouvé que les premiers singes qui ont descendu des arbres étaient des informaticinges qui voulaient s'élever vers le Progrès et l'Avenir...)

Un parseur n'est pas un compilateur complet, même s'il – comme nous avons envie de faire – génère le code grâce à son intégration avec les procédures sémantiques qui construisent son résultat. Il faut penser à d'autres choses :

- La génération du «prologue» – le code qui doit démarrer l'exécution du programme compilé. D'habitude ce code se trouve au début du programme compilé, et il est placé dans le fichier de sortie avant de démarer le parseur. (Ceci concerne le programme compilé indépendant, exécuté directement sous le contrôle du OS. De tels fragments s'appellent parfois *stubs*.)

- Nous avons menti un peu... Imaginez que la compilation échoue, et que le fichier contenant le programme compilé est incomplet, tronqué au millieu par une erreur. Un tel programme *ne doit pas s'exécuter du tout*, car il peut causer des dégats. Le prologue doit alors *avant* le démarrage du programme compilé vérifier son integrité, par exemple la présence du code "OKEY" à la fin du fichier, ou la cohérence d'autres dispositifs de sécurité, comme les codes CRC (*Cyclic Redundancy Check*).

- Si le programme compilé est prévu pour être exécuté par une machine virtuelle, le prologue est parfois absent, mais pas toujours. Même s'il n'a pas besoin d'un code d'initialisation, par exemple d'ouverture des canaux d'entrée/sortie standards, car l'interprète lui-même s'en occupe, le prologue peut contenir une importante bibliothèque de fonctions prédéfinies, l'ouverture des canaux non-standard, ou la spécification des ressources supplémentaires.

  Ainsi presque tout programme en PostScript généré par un traitement de texte ou un paquetage graphique ajoute à la tête de son document envoyé à l'impression un prologue avec plusieurs abréviations et comandes PS qui ne sont pas prédéfinies dans le pilote de l'imprimante, mais qui seront utilisées plusieurs fois.

- Le compilateur ajoute dans le programme compilé des procédures de gestion des buffers I/O, la gestion des exceptions «paniques», les procédures de récupération des signaux du OS (si le programme est compilé en code natif), les primitives d'allocation de mémoire, etc. Souvent ces procédures occupent 90% du code, si le programme est très court. Les programmes interprétés sont alors *beaucoup* plus courts que le programmes compilés, et c'est une raison pour laquelle la plupart des langages de *scripting* est interprété.

Revenons donc à nos parseurs. Quand on regarde la définition d'une grammaire, on voit que les définitions sont réellement descendantes, récursives, et qu'il faut s'arrêter au niveau des primitives. Mais on voit également des propriétés «méta-» de la grammaire : les mêmes stratégies de structuration s retrouvent plusieurs fois dans l'ensemble de productions. Par exemple, on trouve souvent

- les itérations ou séquences, type **`A ::= B | B A`**; De cette façon on construit les mots composés des lettres, les entiers construits des chiffres, les listes (en Lisp), etc.

  Une légère modification, la présence d'un lexème de séparation, style : **`A ::= B | B , A`** nous permet de construire les listes ou autres séquences, p. ex. les initialisations des tableaux en C) d'objets séparés par des virgules, blocs d'instructions séparées par les points-virgules, etc. La même stratégie s'applique à la reconnaissance des suites d'arguments des procédures. Une stratégie presque identique – à la déclaration des variables, ou à la déclaration des champs d'une **`struct`** ;

- plusieurs parseurs «presque primitifs» qui vérifient – par exemple – l'appartenance d'un caractère à la classe des majuscules, ou des symboles spéciaux. Un tel parseur consomme un item (caractère) et lance un prédicat de vérification. On peut évidemment construire quelque chose de plus abstrait ;

- parseurs d'objets «parenthésés» : listes en Lisp ou en Prolog, tableaux, blocs délimités par des accolades, suites d'indices entre crochets en Pascal, construction type **begin ... end** ou **repeat ... until**, appels fonctionnels avec la liste d'arguments entre parenthèses, etc.

Donc, quel que soit le langage, la structuration des parseurs composites suit souvent les mêmes règles. Nous allons alors commencer par la construction des briques génériques, universelles, qui seront réutilisés dans des parseurs concrets. Les définitions de ces derniers peuvent alors être *très* courtes.

5.4 Composition des parseurs fonctionnels

5.4.1 Premiers pas

Le contexte global du processus du parsing sera le suivant. Nous définiront un *opérateur de parsing universel* **`-*>`** utilisé pour appliquer un parseur à un flux de données :

```
infixl 0  -*>
...
unParseur -*> unFlot
```

En fait, nous pourrions définir directement les parseurs comme des fonctions qui agissent sur les flux d'entrée, mais nous préférons les voir comme les «objets» que l'on «applique». Donc, les définitions des types de ces objets, du parseur universel, du «Cutter» qui coupe un élément du flux, et d'un scanneur lexical, sont les suivantes :

```
newtype UParser c a = Pa ([c] -> [(a,[c])])

type Cutter a = UParser a a
type CScanner = UParser Char String   -- Analyseur lexical
```

Notez qu'un **`CScanner`** lit une chaîne (liste de caractères) : **`String`**, mais le type de son premier argument est **`Char`** – un *élément* du flux. Par contre, le type du résultat est de nouveau une chaîne.

Le mot-clé **`newtype`** en Haskell est une nouveauté. Pratiquement nous pouvons l'exploiter comme **`data`**, l'introduction d'un nouveau type algébrique, dont les instances sont identifiées par la balise **`Pa`**. Mais la sémantique est un peu différente, en fait, ceci ressemble un peu à un synonyme (**`type`**), qui se comporte comme la définition suggérée (et ensuite désavouée) dans la section (5.3.1). Quand le programme en Haskell est compilé, la balise **`Pa`** disparaît, et notre objet se comporte comme une fonction. Mais dans le code source, nous écrirons

```
Pa parseFun -*> flux = parseFun flux
```

Définissons d'abord trois parseurs primitifs. Le premier, **`fail`** ne consomme rien, et échoue toujours, c'est-à-dire, retourne la liste vide. Le second, **`return`**, est son dual : laisse le flux d'entrée intact, mais retourne une valeur spécifié *a priori*. Le troisième, **`item`**, est finalement un parseur qui fait quelque chose, il coupe le premier item du flux, et le retourne.

```
fail s = Pa (\inp -> [])

return :: a -> UParser c a
return x = Pa (\inp -> [(x,inp)])

item  :: Cutter a
item  = Pa (\inp -> case inp of
             []     -> []
             (x:xq) -> [(x,xq)])
```

Le parseur **`fail`** sera utilisé rarement. On en a besoin seulement dans des cas spéciaux, où on provoque l'échec volontairement. Plus souvent il sera généré par les circonstances : quand aucune autre possibilité ne marche. (Il est parametré pour des raisons qui seront expliquées un peu plus tard, mais son paramètre ici n'est pas utilisé ; dans d'autres circonstances il peut contenir un message diagnostic).

Le parseur **`return`** est incontournable : malgré sa simplicité d'est une des constructions les plus fondamentales dans notre présentation.

L'outil fondamental de *liaison*, la «colle» qui permet de combiner deux parseurs de manière séquentielle, est l'opérateur qui traditionnellement s'appelle **bind**, et que nous construirons comme un opérateur infixe **(>>=)**. Il agit sur des parseurs qui sont des fonctions (oublions la balise **Pa** ; elle est là, mais nous pouvont presque toujours traiter les parseurs comme des objets fonctionnels), donc il est une fonction d'ordre supérieur. On doit pour l'instant imaginer que la construction **(p >>= f)** est une généralisation assez évoluée de l'application **f(p)**.

Imaginons que **p** est un parseur, et **f** un «générateur de parseurs» – une fonction qui s'applique à une *valeur* (souvent : la valeur retourné par le parseur précédent), et qui retourne un parseur. Par exemple, **return** appartient à cette classe : il s'applique à une valeur quelconque, et produit un parseur qui génère cette valeur indépendamment du flux d'entrée. Un autre exemple peut être la construction d'un parseur qui vérifie et filtre une valeur concrète, passé comme argument à **f**. Encore un autre, très important : le parseur construit par la fonction **f** génère la valeur finale à partir du flux *et* l'objet passé comme paramètre. Les exemples seront très nombreux.

Le parseur **p** est une fonction qui renvoie une liste de valeurs appariées avec les segments non-consommés du flux : **[(v1,i1),(v2,i2),...]**. Le lecteur doit comparer ce comportement avec nos définitions de fonctions non-déterministes, comme la fonction d'insertion d'un élément «n'importe où» dans une liste. Le parsing est une opération qui peut être non-déterministe, même si tout non-déterminisme redondant est à éviter au nom de l'efficacité.

La fonction **f** récupère, élément par élément, les valeurs **v_k**, et pour chaque valeur rend un parseur. Bien sûr, l'itération : «élément par élément» est assurée par l'opérateur *bind*, non pas par la fonction. Ce parseur est appliqué au flux correspondant **i_k**. Le résultat est de nouveau une liste de paires valeur-flux. Cette application élément par élément peut être réalisée par la fonctionnelle **map**, mais le résultat final doit être aplati. Commençons par :

```
infixl 1 >>=
```

mais attention, cet opérateur est déjà prédéfini et constitue une fonction de «liaison» plus générale que la composition de parseurs. La fonction **fail** est prédéfinie aussi, tout ceci appartiend au monde des Monades. Mais pour l'instant nous allons les introduire comme s'ils n'étaient pas connus, ce qui ne dispense pas le lecteur de lire les Annexes !.

```
conc = foldr (++) []     -- Aplatisseur des listes de listes

(>>=)   :: UParser a c -> (a -> UParser b c) -> UParser b c
Pa p >>= f =
    Pa (\inp -> concat [(f v) -*> out | (v,out) <- p inp])
```

ou, si l'on préfère :

```
Pa p >>= f =
    Pa (\inp -> concat (map (\v out -> f v -*> out)
                            (p inp))
```

Récapitulons : le résultat de la construction **Pa p>>=f** est un parseur, une fonction qui agit sut un flux **inp**. Son fonctionnement est le suivant. **p** agit sur **inp** et produit une liste de résultats possibles, dont une instance est notée par **(v,out)**. La fonction **f** agit sur chaque instance, mais elle même peut créer plusieurs résultats, et on obtient ainsi une liste de listes. La fonction **conc** aplâtit le résultat.

5.4.2 Séquences, filtres, alternatives, itérations

Grâce au combinateur *bind* construisons à présent

- un parseur **seqp** qui représente la séquence de deux parseurs, et qui renvoie comme valeur la paire **(x,y)** si le premier produit **x**, et le second – **y**. Mieux : on n'est pas obligé de retourner un tuple **(x,y)**, mais nous pouvons appliquer une fonction – constructeur quelconque à **x** et **y**. Le parseur **seqp** sera donc parametré par ce constructeur ;
- un parseur filtrant **sat**, qui vérifie que le premier item du flux satisfait une condition logique ;
- une réalisation plus concrète du **sat** – un parseur vérifiant l'égalité entre la tête du flux, et une valeur donnée. (Ceci peut être utilisé pour la reconnaissance d'un séparateur lexical, d'un mot-clé, etc.)

- Les parseurs qui filtrent, et génèrent des lettres, chiffres, et caractères alphanumériques. Pour cela nous aurons besoin d'un combinateur très simple qui construit l'alternative de deux parseurs, en concaténant leurs résultats respectifs.

```
seqp cnstr p q = p >>= \x ->
                 q >>= \y -> return (cnstr x y)
alt (Pa p) (Pa q) = Pa (\inp -> p inp ++ q inp)

sat    :: (a -> Bool) -> Cutter a
sat p  = item >>=  \x ->
         if p x then return x else fail ""

lit c = sat (==c)
```

Avant de passer aux parseurs lexicaux observons que la construction **alt** n'est pas une bonne solution dans la plupart de cas intéressants, surtout là où l'alternative sert à définir une itération, ou dans le cas où les deux variantes sont visiblement incompatibles, p. ex. une lettre ou un chiffre. Nous pouvons définir une autre alternative, plus fréquemment utilisée, qui teste le premier composant et seulement s'il échoue, on applique le second :

```
xor (Pa p) (Pa q) =
  Pa (\inp -> let s=p inp
              in  if s==[] then q inp
                           else s)

infixl 0 #        -- pour notre confort
a # b = xor a b
```

Et voici les éléments primitifs d'un scanneur :

```
interval a b = sat (\x -> a <= x && x <= b)

digit = interval '0' '9'

lower  = interval 'a' 'z'
upper  = interval 'A' 'Z'

letter = lower # upper
alphanum = letter # digit
```

(Rappelons que la définition des lettres est un peu primitive, sans accents ni autres diacritiques. La généralisation à d'autres langues humaines est un joli exercice.)

Finalement, construisons un mot à partir d'une chaîne de caractères, et effectuons un test. Le mot (**word**) consomme les lettres par l'enchaînement des parseurs **letter**. Le parseur doit correspondre à la production suivante :

```
Word   ::= Letter Word'
Word'  ::= Letter Word' | φ
```

Le parseur **word** appelera une fonction interne **word'** qui doit itérer **letter**. Quand **letter** échoue, la clause alternative du **word'** renvoie la chaîne vide – l'objet terminal, auquel **word'** attache toutes les lettres précédentes. La différence entre **Word** et **Word'** est claire : **Word'** peut être vide, **Word** – jamais, c'est une fermeture positive.

Nous avons remarqué qu'une telle construction est assez typique, donc au lieu de construire ces parseurs, construisons d'abord un combinateur générique, capable d'itérer un parseur quelconque, et mettre les éléments partiels dans une liste. Nous allons donc exploiter le combinateur **seqp** parametré par le constructeur **(:)** :

```
infixr 1 +>
a +> b = seqp (:) a b

many  p = p +> many' p
many' p = p +> many' p  #  return []
```

et à présent il suffit d'écrire **`word = many letter`**. L'application

```
word -*> "Belle marquise ..."
```

retourne **`[("Belle", " marquise ...")]`**. Bien sûr, si on remplace **`(#)`** par **`'alt'`**, le résultat sera une liste qui contient **`"Belle"`** avec **`"Bell"`**, **`"Bel"`** etc.

La construction des nombres demande un autre protocole de combinaison des items, nous ne voulons pas construire une chaîne, mais combiner les chiffres selon l'algorithme qui a déjà été discuté :

```
nombre l = nb l 0    -- où l est une liste d'entiers entre 0 et 9
where
  nb [] tmp = tmp
  nb (x:xq) tmp = nb (10*tmp+x)
```

Il nous faudra généralise **`many`**, avec un constructeur arbitraire, et une valeur initiale du tampon aussi arbitraire. Mais il y a un autre problème. Supposons que l'on essaye de construire un itérateur générique comme ci-dessous :

```
iter constr tmp0 p = seqp constr p it  where
  it = seqp constr p it # return tmp0
```

et que le parseur des entiers soit défini

```
integ = iter accum 0 digit     -- où

accum x tmp = 10*tmp + ord x - ord '0'
```

Le résultat du test : **`integ -*> "7802 beaux yeux..."`** nous réserve une mauvaise surprise : **`[(2087," beaux yeux ...")]`**. La récursivité a été mal exploitée, le parseur interne **`it`** qui remplace **`many`** est erroné ! Il faut qu'il soit parametré par le tampon comme la fonction **`nb`**. Cette question sera abordée avec plus de détails encore deux fois : lors d'élimination de la récursivité à gauche, et quand nous allons parler des attributs et de l'analyse sémantique pilotée par la syntaxe.

Construisons d'abord un autre combinateur de séquentialisation, nommé **`seql`**. Il prend deux parseurs, **`p`** et **`q`**, et la fonction constructrice **`constr`** qui combine les deux résultats partiels, mais cette fois le second parseur **`q`** *possède un paramètre-tampon*. Il faut donc prévoir aussi sa valeur initiale. Voici la construction :

```
seqf constr tmp p q =
  p >>= \x -> q (constr x tmp)
seql constr tmp p q =
  seqf constr tmp p q # return tmp
```

Attention. Le parseur **`seqf`** peut échouer, **`seql`** non, mais il doit être utilisé dans un contexte où le parseur **`q`** n'échoue jamais. Si parseur **`p`** échoue, on retourne le tampon.

Le sérialiseur **`seql`** peut nous servir à présent à construire un itérateur correcte pour notre problème :

```
lmany constr tmp0 p = seqf constr tmp0 p lmany' where
 lmany' tmp = seql constr tmp p lmany'
```

Cette fois la construction **`integ = lmany accum 0 digit`** produit un parseur correcte. Notez que **`lmany'`** n'échoue jamais, comme nous l'avons demandé, car **`seql`** a toujours un résultat à rendre, mais qu'il commence par lancer **`seqf`**, donc, la tentative de trouver un nombre dans un texte qui ne commence pas par un chiffre, échoue au lieu de retourner 0. Ceci est raisonnable.

5.4.3 Sérialisation sans mémoire

L'analyse lexicale doit consommer et jeter les espaces blancs, tabulations, fins de ligne, etc. On peut éventuellement sauvegarder la position dans le flux d'entrée, mais on n'a jamais besoin d'accumuler les résultats partiels, donc l'usage des sérialiseurs **`seqp`** ou **`seql`** serait inefficace et inutile. Dans un autre contexte, par exemple si on analyse une liste qui commence par le caractère **`[`**, on doit le reconnaître (par **`sat`** ou **`lit`**), mais, encore une fois, on n'a pas besoin de stocker aucune information qui le concerne, car on en sait tout.

Dans le Prélude il existe un sérialiseur universel **`(>>)`** (que nous allons appeler : suite), défini (par défaut) par *bind* :

```
p >> q  = p >>= \ _ -> q
```

ou les deux argument seront des parseurs. Nous pouvons l'utiliser tel quel, ou éventuellement définir un autre, légèrement plus efficace, car plus spécifique :

```
Pa p >> Pa q = Pa (\inp -> concat [q out | (_,out) <- p inp])
```

Ceci suffit pour définir un parseur qui nous débarasse des espaces :

```
space = lit ' ' # lit '\n'
spaces :: CScanner
spaces = (space >> spaces) # return []
```

Notez la déclaration de type. Sémantiquement elle semble redondante, mais le système de types de Haskell nous le demande à cause de la restriction de monomorphisme. Sans cette déclaration, **spaces** serait trop polymorphe. Une autre solution est de figer le type ailleurs, par exemple en définissant

```
empty = return [] :: CScanner
spaces = (space >> spaces) # empty
```

5.4.4 Encore un exemple : listes Prolog

Construisons un parseur qui consomme un flux de forme

```
[alpha, b ,  [x,y,123], gg, [[v], hhh,[]], [klm,n|p]]
```

et qui construit une arborescence qui représente une liste – possiblement hétérogène – composé de mots (chaînes), entiers, ou autres listes. La liste peut se terminer par une «feuille» comme dans **[klm,n|p]**, ou être vide. Ceci est une raisonnable approximation des listes en Prolog. Nous voulons également que le parseur ignore les espaces. Il nous faudra définir d'abord la grammaire, et ensuite le type du résultat. (La grammaire ci-dessous est une légère modification de la grammaire présentée dans la section (5.1.1)).

```
List    ::= '[' Lseq Ltail ']'
Lseq    ::= φ | Lseqp
Lseqp   ::= Item | Item ',' Lseqp    ou, factorisé...
Ltail    ::= φ | '|' Item
Item    ::= Word | Number | List
```

Les structures Haskell qui représenteront les listes seront définies comme des arbres :

```
data Atom = N Integer | S String
data Tree = Nil | F Atom | L Tree Tree
```

et nous pouvons passer à la construction du parseur. Avouons cependant que l'exercice est un prétexte, grâce auquel nous voulons introduire quelques techniques un peu plus génériques de composition fonctionnelle des parseurs. tout d'abord, il est utile de généraliser le parseur **many**, comme nous l'avons fait avec **lmany**. Nous allons le paramétrer par le constructeur (pas forcément l'assemblage des listes par **(:)**), et par une valeur initiale arbitraire, compatible avec le constructeur. Ce parseur, **rmany** servira de définition de **many** par une simple instantiation :

```
rmany constr init p = seqp constr p w  where
   w = seqp constr p w #  return init

many p = rmany (:) [] p
```

Il est utile de pouvoir transformer sur place le résultat d'un parseur, par exemple de transformer un mot ou un entier en une feuille de notre arbre. ceci est trivial, voici un générateur convenable, qui prend le parser **p** et une fonction «normale» **f** et qui construit le parseur transformé :

```
transf f p = p >>= return . f
```

Notez l'ommission de «lambda» grâce au combinateur **(.)**.

Nous voudrions éliminer les espaces, donc il est utile d'augmenter tout parseur par un préfixe qui s'en charge. introduisons également quelques abréviations pour les littéraux «nettoyés» :

```
clr p = spaces >> p

clit c = clr (lit c)

virg = clit ','
bar = clit '|'
```

Les itérateurs comme **rmany** ou **lmany** demandent que la suite d'items soit contiguë, sans séparateurs, mais nous avons ici la virgule. Construisons donc un itérateur paramétré par un parseur-séparateur.

```
rmsep constr rest sep p = seqp constr p w  where
   w = seqp constr (sep >> p) w # rest
```

Il a été généralisé par rapport aux précédents par un autre aspect. *Au lieu de retourner une valeur initiale dans le cas d'échec de la boucle, il lance un parseur* **rest**, *qui s'en charge de procurer cette valeur*. Ceci sera très utile pour le parsing des formes **[a|b]**. Un autre parseur générique **brack** met un parseur **p** quelconque «entre parenthèses» filtrées par les parseurs **a** et **b** :

```
brack a p b = a >> p >>= \v-> b >> return v
```

Voici le reste de notre construction :

```
list = brack (clit '[') lseq  (clit ']')
lseq = lseq' # return Nil
lseq' = rmsep L ltail virg itm
ltail = clit '|' >> itm # return Nil
itm = clr (list
           # transf (F . N) integ
           # transf (F . S) word)
```

Notre construction prouve l'utilité pratique du formalisme. Les parseurs construits de cette manière seront plusieurs fois plus courts que les analyseurs construits par des générateurs en C. Ils *ne sont pas* tellement plus lents !

Attention ! La stratégie exploitée dans cette section : la combinaison de l'analyse lexicale et syntaxique dans un module, n'est pas idéale. Parfois, vous devez d'abord séparer les items lexicaux (et éliminer les espaces), construire un flux de lexèmes, et ensuite effectuer l'analyse syntaxique «pure». Ainsi, si on change la syntaxe du langage sans modifier sa couche lexicale, les modifications sont plus localisées.

5.5 Exercices

Q1. Construire l'instance **Show** pour la structure **Tree** de la section (5.4.4). Les arbres doivent être affichées comme les listes-sources, avec crochets, virgules et éventuellement avec la barre verticale. les balises **F**, **N** et **S** doivent être omises.

R1.
```
instance Show Atom where
  showsPrec _ (N i) = shows i
  showsPrec _ (S s) = shows s

instance Show Tree where
  showsPrec _ Nil = showString "[]"
  showsPrec _ (L a b) = showChar '[' . shows a . shl b
       where shl Nil  = showChar ']'
             shl (L x xs) = showChar ',' . shows x . shl xs
             shl (F x) = showChar '|' . shows x . showChar ']'
showsPrec _ (F a) = shows a
```

Q2. Réfléchir comment, au lieu de transformer une telle structure de données en chaîne, ce qui ajoute des guillemets à l'intérieur, définir une procédure d'affichage, qui trnasporte le résultat sur un fichier extérieur.

R2. J'ai dit : réfléchir…

Q3. Lire dans le Prélude standard et ***essayer de comprendre*** la classe **`Read`**.

R3. Ah, vous n'avez pas le temps? Bien, vous allez le regretter...

Q4. Alors, il est souhaitable de séparer l'analyse lexicale et syntaxique? Faites le. Construisez un parseur des listes Prolog qui passe d'abord par l'étape lexicale.

R4. La solution n'est pas immédiate, car nous devons préciser d'abord notre conception du *lexème*. Il faut – de préférence – transformer les chaînes (mots), les entiers, etc., ainsi que les séparateurs, les crochets, etc. en entités spéciales, appartenant à un type à part entière. Nous avons eu déjà ce problème, quand il fallait convertir les atomes (mots ou nombres) en feuilles de l'arbre final. Nous allons changer aussi la définition de l'arbre qui représente les listes hétérogènes.

La parsing reste néanmoins assez primitif. Commençons par l'analyse lexicale :

```
data Lexem = I Integer | W String | Op String | Spec Char
   deriving (Eq,Show)

liter c = lit c >> return (Spec c)
lbrack = liter '['
rbrack = liter ']'
barre = liter '|'
virgule = liter ','
wrd = transf W word
nbr = transf I integ
oper = transf Op (many opchar)
opchar = sat
(\c -> elem c ['+','-','*','/','=','<','>','&','!'])

lexs = lbrack # rbrack # virgule # barre # oper # wrd # nbr

scanner = spaces >> rmsep (:) (return []) spaces lexs
```

Il suffit de tester : **`scanner -*> "  Belle [Marquise, vos 123, []..."`**. Les balises **`I`**, **`W`** etc. jouent le rôle des spécificateurs de *catégorie lexicale* de l'item concerné. À présent ce sont ces balises qui pilotent l'analyse syntaxique. La fonction **`elem`** vérifie si un objet appartient à une liste. Construisez cette fonction, et comparez votre solution avec celle du Prélude. Vous serez peut-être surpris...

Le parseur proprement dit (analyseur et constructeur des arbres) ne subit presqu'aucune modification importante. Nous avons changé un peu le style du codage pour plus de variété.

```
data PTree = Void | Id String | Nb Integer | PL PTree PTree
   deriving Eq

spec c = lit (Spec c)          -- remplace litr
comma = spec ','               -- remplace virgule

atom = item >>= \x -> case x of
  (W a) -> return (Id a)
  (I a) -> return (Nb a)
  _     -> fail []

list = brack (spec '[') lseq  (spec ']')
lseq = lseq' # return Void
lseq' = rmsep PL ltail comma itm
ltail = spec '|' >> itm # return Void
itm = list # atom
```

En général, la séparation des deux phases est souhaitable, mais ajoute un peu de complication. Il faut traîner plusieurs systèmes d'identification des objets : leurs catégories lexicales et syntaxiques séparement. Finalement on risque de confondre les balises, de les oublier, etc., ce qui n'est pas dangereux, mais pénible.

Chapitre 6

Analyse syntaxique II – développement et optimisation

6.1 Analyse des expressions algébriques

Les vrais langages de programmation sont relativement simples sur le plan syntaxique. Bien sûr, rien n'est trivial, mais on est loin de la généralité traitée parfois dans la théorie des automates et langages. Le plus souvent on trouve des simples itérations, ou des structures imbriquées, et même si le langage contient des centaines d'opérateurs infixes différents (le langage Icon s'approche de ce terrible «idéal», et quelques programmeurs en Haskell également aiment bien les formes infixes privées très longues, comme `-#*>--*=>`), mais ceci n'augmente pas la complexité syntaxique du langage.

6.1.1 Premier essai, opérations Booléennes

Construisons un parseur pour la grammaire qui exprime la composition des opérations logiques. Les objets atomiques dans le flux d'entrée seront des caractères alphabétiques. Nous savons comment généraliser ceci.

```
Atome  ::= F | T | A | B | C      etc.
Expr   ::= Conj | Conj OR  Expr
Conj   ::= Prim | Prim AND Conj
Prim   ::= (NOT | φ) (Atome | '(' Expr ')' )
NOT ::= '~'
AND ::= '&'
OR  ::= '|'
```

Les opérations «et» et «ou» logiques sont associatives et symétriques (commutatives), donc nous nous sommes permis de définir leur composition en utilisant l'associativité à droite. Notons qu'une optimisation a été apportée ci-dessus : la factorisation de la négation.

Comme d'habitude, avant de construire le parseur il faut préciser quel est le résultat de l'analyse. Introduisons donc une structure arborescente qui représente des expressions Booléennes :

```
data Connect = AND | OR
data Arbool  = At Char | Cn Connect Arbool Arbool | Ng Arbool
```

Ceci est un peu différent par rapport aux listes Prolog, ici pas seulement les feuilles, mais les noeuds intermédiaires stockent aussi une information concrète : le connecteur «et» ou «ou». La balise **`At`** identifie l'atome, et **`Ng`** – la négation.

```
ou  = clit '|'
et  = clit '&'
non = clit '~'
lpar = clit '('
rpar = clit ')'
atomic = transf At (clr letter)
```

```
bexpr = conj >>=
  \u -> (ou >> bexpr >>= \v -> return (Cn OR u v)) # return u

conj  = prim >>=
  \u -> (et >> conj >>= \v -> return (Cn AND u v)) # return u

prim = (non >> prim' >>= return . Ng) # prim'
prim' = brack lpar bexpr rpar # atomic
```

et nous pouvons demander l'analyse de, disons, **`a&(b|t|~(x|a&b))|c&~f&(a|p)`**.

Notons l'usage des alternatives asymétriques. D'abord on essaie la clause la plus longue, et si elle échoue, on reste avec le segment initial. La même observation s'applique à la définition de la conjonction. Le parseur **`prim'`** est toujours positif, et **`prim`** peut contenir optionnellement la négation.

(Observons également un certain maniérisme notationnel : au lieu d'écrire de manière plus lisible **`\x -> return (Ng x)`**, nous avons abrégé cela à : **`return . Ng`**, ce qui n'est pas tellement clair pour les débutants. Mais il faut s'habituer à l'usage des combinateurs.)

Passons à l'optimisation et aux généralisations éventuelles. En fait, une optimisation (factorisation) a déjà été effectuée. Une solution un peu plus courte, mais moins efficace serait :

```
bexpr = conj >>= \u -> ou >> bexpr >>= \v -> return (Cn OR u v)
        # conj
```

ce qui correspond mieux à la grammaire d'origine, mais qui – si «**ou**» échoue, répète l'application du parseur **`conj`**.

On peut observer encore :

- La négation optionnelle existe aussi en arithmétique, et en général un parseur qui optionnellement fait une chose avant un autre parseur serait d'utilité générale.
- **`bexpr`** et **`conj`** ont la même structure compositionnelle, qui est d'ailleurs presque la même qu'en algèbre numérique. Comment en extraire tout comportement générique?

La préfixation optionnelle est simple, par exemple :

```
option constr opt p =
  (opt >>= \o -> p >>= \x -> return (constr o x)) # p

prim = option (\_ z -> Ng z) non prim'
```

Le méta-parseur **`option`** prend comme arguments le préfixe, le parseur principal, et une fonction de composition des résultats (un opérateur binaire).

Les observations restantes se réduisent à une simple constatation : les deux formes **`bexpr`** et **`conj`** se réduisent à l'itération associative à droite des composantes séparées par les opérateurs. Notre précedent parseur – itérateur avec séparateurs : **`rmsep`** ne convient plus, car le séparateur ne peut être plus ignoré. Voici donc un générateur des itérations opérationnelles à droite, et leur usage :

```
iterr cnstr3 op p = w where
 w = p >>= f
 f x = (op >>= \y -> w >>= \z -> return (cnstr3 y x z))
       # return x

bexpr = iterr (\_ x y -> Cn OR x y) ou conj
conj = iterr (\_ x y -> Cn AND x y) et prim
```

La construction du parseur **`iterr`** est un peu trop générale pour nos besoins : l'opérateur lui-même (l'argument **`op`**) peut apporter quelque chose de spécifique à la construction de l'arbre. Ici, si l'opérateur est «**ou**» nous savons *a priori* qu'il faut utiliser **`OR`**, etc, donc le premier argument de **`cnstr3`** n'est pas utilisé.

6.1.2 Arithmétique et problèmes avec la récursivité à gauche

Dans cette section nous nous occuperons (en outre) de la *normalisation de Greibach* – la transformation des règles récursives à gauche, en règles récursives à droite. Vu l'importance des expressions arithmétiques dans la programmation courante, la construction du parseur dans cette section doit être bien maîtrisée.

Nous voulons pouvoir analyser les expressions arithmétiques classiques. Éliminons la puissance, laissons seulement les quatre opérations de base, les appels fonctionnels de genre **`sin(x)`** et les parenthèses. La grammaire réduite aura la forme (déjà discutée)

```
Expr  ::= Trm | Expr OpAdd Trm
OpAdd ::= + | -
Trm   ::= Fctr | Trm OpMul Fctr
OpMul ::= * | /
Fctr  ::= Atome | Tfunc | '(' Aexpr ')'
Atome ::= Id | Nombre
Tfunc ::= Id '(' Seq ')'
Seq   ::= φ | ESeq
ESeq  ::= Expr (φ | , ESeq)
```

Elle est suffisamment riche, et elle contient des éléments déjà connus, comme les séquences itératives associatives à droite. Un **`Atome`** sera une chaîne de caractères, numérique ou alphanumérique. Nous permettrons l'occurrence d'espaces dans la chaîne d'entrée. Les espaces pourront entourer les opérateurs ou les virgules, et être placées derrière la parenthèse ouvrante ou devant la parenthèse fermante. Ils ne doivent pas couper les mots (ou nombres), et ne doivent pas séparer le nom de la fonction de la parenthèse ouvrante.

Ici nous avons cependant aussi les règles associatives à gauche, et ceci est une bombe à retardement. Toute application de ce parseur commence par l'appel de lui même, et la lecture du flux d'entrée ne progresse pas. Les règles récursives à gauche sont très dangereuses, et le bouclage récursif est leur conséquence directe indépendamment de la construction du parseur, fonctionnel ou pas. Ce problème touche *tous* les parseurs descendants. La «solution» ci-dessous n'est pas seulement inefficace, elle est tout simplement mortelle :

```
aexpr = aexpr >>= \u -> op >> trm >>= \v -> return (plus u v)
       # trm
```

La solution «officielle» du dilemme consiste à modifier la grammaire par la technique qui s'appelle la *normalisation de Greibach*

Si la grammaire contient une règle suivante

`S ::= S` α_1 **`| S`** α_2 **`| ... S`** α_n **`|`** β_1 **`| ... |`** β_m

où α et β sont des séquences quelconques, la normalisation consiste à remplacer cette production par

`S  ::=` β_1**`S'`** **`|`** ... **`|`** $\beta_m S'$
`S' ::=` α_1**`S'`** **`|`** α_2**`S'`** **`|`** ... α_n**`S'`** **`|`** ϕ

Par exemple, au lieu de la définition du terme additif :

```
Trm   ::= Fctr | Trm OpMul Fctr
```

nous aurons

```
Trm   ::= Fctr TrSeq
TrSeq ::= OpMul Fctr TrSeq | φ
```

Une telle transformation peut être faite automatiquement par un processus de pré-traitement de grammaire. Parfois une normalisation ne suffit pas, car la récursivité à gauche peut être indirecte. Il faut alors itérer le schéma ci-dessus pour tous les non-terminaux «dangereux».

Notons encore une fois que la nécessité d'éliminer les règles récursives gauches est caractéristique aux parseurs *descendants*. Les analyseurs *ascendants*, p. ex. les automates produits par Yacc, qui construisent une arborescence syntaxique à partir des feuilles, peuvent gérer ce problème, car ils ne sont pas récursifs.

La règle transformée nous rappelle quelque chose : c'est un cas déjà traité, une itération linéaire. La seule différence par rapport au cas des opérateurs associatifs à droite s'exprime par une réduction différente :

$$((x_1 \oplus x_2) \oplus x_3) \cdots \oplus x_n$$

(Ceci doit rappeler la différence entre les fonctionnelles **`foldr`** et **`foldl`**...).

Pour ne pas répéter le schéma Booléen, au lieu d'assembler l'arbre syntaxique «physique», notre parseur construira directement le code postfixe adapté à une machine virtuelle à pile (très simplifiée !), et l'arbre sera purement conceptuel.

Commençons par le définition du **`Code`** et d'une chaîne-exemple.

```
type Code = [CodeItem]
data CodeItem = I Int | S String | O String
ch6 = "alpha *( b-55/beta) - c*d-(165-(y - zzz ) )"
```

Le résultat du parsing doit être la liste dont la structure est

```
[alpha,b,55,beta,/,-,*,c,d,*,-,165,y,zzz,-,-,-]
```

(Pour l'instant nous ne discutons pas les expressions sous forme des appels fonctionnels (procéduraux). Ceci viendra un peu plus tard.) Commençons par les parseurs atomiques :

```
tlit c = transf (\x-> O [x]) (clit c)

add = tlit '+'
sub = tlit '-'
mul = tlit '*'
dyv = tlit '/'
opAdd = add # sub
opMul = mul # dyv

entier = transf (\x -> [I x]) (clr integ)
ident = transf (\x -> [S x]) (clr word)
atm = ident # entier
```

(Le nom **`div`** est prédéfini en Haskell.) Pour assembler le code postfixe à partir de deux opérandes et un opérateur, il suffit d'utiliser le constructeur suivant :

```
assembl op x y = x ++ y ++ [op]
```

(ce qui est horriblement inefficace et fait mal aux dents...), mais le vrai cheval de bataille est un ***itérateur à gauche*** **`iterl`**, dont la forme ressemble un peu à **`iterr`**, mais qui réduit la chaîne différemment, comme **`lmany`**. Observons que le parseur interne, qui itère l'argument **`p`** *est un parseur parametré, une fonction d'un argument* qui est le tampon. Voici l'itérateur et le reste de la construction :

```
iterl cnstr3 op p = p >>= pseq  where
  pseq tmp = (op >>= \y -> p >>= \z -> pseq (cnstr3 y tmp z))
             # return tmp

aexpr = iterl assembl opAdd trm
trm   = iterl assembl opMul fctr
fctr  = brack lpar aexpr rpar # atm
```

La construction d'un terme fonctionnel (appel genre $f(x, y+z)$)) a été différée, et se trouve dans la section des exercices, car il faut que le lecteur travaille un peu aussi. (C'était une blague. Mais est-elle vraiment drôle?...)

Pourquoi ce tampon? Regardons la normalisation de Greibach encore une fois, sous un aspect graphique. Prenons un terme composite **`a*b*c`**. Si la production est récursive à gauche, il sera analysé comme **`(a*b)*c`**, ce qui correspond à l'arbre dessiné sur la Fig. (6.1). Les productions utilisées par le parseur sont les suivantes

```
Trm   ::= Fctr TrSeq
TrSeq ::= OpMul Fctr TrSeq | ϕ
```

(où il faut noter le fait que la séquence **`Fctr TrSeq`** dans la définition de **`TrSeq`** *n'a pas été optimisée* à **`Trm`**.) L'Expression, le Terme, etc. sont des non-terminaux qui possèdent une valeur intuitive et visuelle importante : ils constituent des nœuds de l'arbre du parsing complet, ils forment des sous-arbres.

Or, **`TrSeq`** *n'est pas* un sous-arbre. La portée de ce nonterminal est la boîte pointillée : tout sauf le premier item. C'est une structure de données «incomplète». On peut considérer qu'une telle structure est un *objet fonctionnel* qui représente un sous-arbre après sa complétion par **son contexte gauche**. Le parsing de **`Trm`**

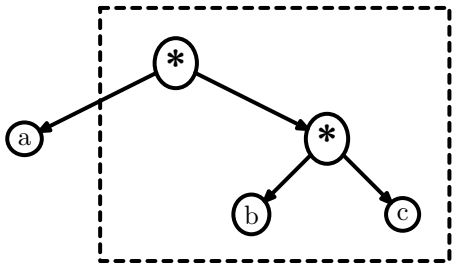


Fig. 6.1: Terme composite, associatif à gauche

lance le parseur **`Fctr`**, dont la valeur (disons : **`x`**, ici «a») sera justement le contexte gauche pour l'appel de **`TrSeq`** suivant.

`TrSeq` lance de nouveau son sous-parseur **`Fctr`** qui récupère le premier item restant – ici «b». La réduction $a \cdot b$ fournit à présent le contexte gauche à un appel ultérieur. Comparez avec les exercices, notamment avec la reconstruction d'un nombre entier à partir des chiffres.

6.1.3 Quelques optimisations

Les combinateurs sont simples et très souples, mais la transposition de la structure statique d'une grammaire vers le dynamisme des objets fonctionnels qui se cachent à l'intérieur des parseurs, peut générer des algorithmes peu efficaces, avec une sur-consommation de mémoire, et gaspillage de temps. Le retour de réponses multiples et le *backtracking* sont des sources évidentes du gaspillage. L'usage des outils comme **`PREMIER`** décrit dans une section ultérieure est souhaitable. Cependant :

***Credo* religieux no. 13 :** Optimisation d'un compilateur commence par l'optimisation de la grammaire.

Il faut donc au moins *factoriser* les alternatives, et éviter tout non-déterminisme inutile. Sachant que le vrai travail d'un parseur n'est pas l'analyse pure : acceptation ou rejet du texte, mais la construction du code intermédiaire, il faut connaître des techniques de programmation en général : optimiser les appels récursifs, *éviter la concaténation des listes en cascade !*, etc. Si possible : réduire directement, pendant la compilation, des expressions constantes. Cependant ceci est l'optimisation de l'application compilée, et non pas du compilateur. Ce sujet sera abordé lors de la discussion des attributs.

Il existe plusieurs autres stratégies d'optimisation, qui dépendent très fort du langage d'implantation. En particulier, les langages paresseux et les langages stricts se comportent différemment, et la programmation paresseuse favorise l'usage de la récursivité non-terminale, si l'appel récursif se trouve à l'intérieur d'un constructeur de données qui seront consommées incrémentalement. Ceci n'est pas restreint aux problèmes de compilation. Pour les langages strictes une telle approche peut déborder la pile, et il faut l'éviter.

6.2 Opérateurs de précédence et associativité quelconques

Cette section constitue une introduction aux grammaires d'opérateurs, et elle décrit une stratégie du parsing ascendante (mais attachée à nos parseurs descendants standard). Nous voulons implémenter le parsing des expressions un peu plus compliquées que celles vues jusqu'à présent. Rappelons, que si les opérateurs infixes qui figurent dans une expression sont soit tous associatifs à gauche, soit à droite, on peut construire les parseurs correspondants par les itérateurs – à gauche ou à droite, comme ci-dessous :

```
iterl cnstr3 opp p = shift where
 shift = p >>= pseq
 pseq ctx = (opp >>= \op -> p >>= pseq . (cnstr3 op ctx))
      # return ctx

iterr cnstr3 opp p = shift where
 shift = p >>= pseq
 pseq ctx = (opp >>= \op -> shift >>= return . (cnstr3 op ctx))
      # return ctx
```

où **`cnstr3 op x y`** construit un noeud dans l'arbre syntaxique, **`opp`** est le parseur d'un opérateur appartenant à la catégorie correspondante, et **`p`** est le parseur d'un élément de niveau inférieur (qui peut naturellement contenir l'appel aux itérateurs, donc on *peut* mélanger les opérateurs de deux associativités dans la même expression, mais pas au même niveau).

Mais le problème est le suivant : comment effectuer le parsing, si le langage permet la définition des opérateurs de précédence et associativité quelconque, qui peuvent figurer ensemble dans une expression? La grammaire n'est pas close. En principe nous pouvons clore le parseur en limitant le nombre de précédences différentes à, disons, 10 (comme en Haskell. Ainsi il est possible de faire une grammaire non pas à deux trois niveaux, comme notre langage algébrique : expressions – termes – facteurs, mais plus profonde, cependant ceci devient vite illisible.

Les techniques ascendantes générales ont été mentionnées, et elles méritent une discussion à part, voir section (8). Une technique ascendante facile s'appuie sur les grammaires dites de précédence, ou grammaires d'opérateurs. (En anglais : *operator precedence grammars*. Elles n'ont rien d'inhabituel, et plusieurs grammaires bien connues peuvent être réduites à la forme opérationnelle. Formellement, une grammaire de précédence se caractérise par deux exigences :

- il n'y a pas de production dont la partie droite est la chaîne vide ϕ , et
- aucune production ne contient deux non-terminaux adjacents. (Ceci est une affirmation forte qui risque d'être mal comprise ; il s'agit de prevenir la juxtaposition de deux données, elles doivent être séparées par des opérateurs qui sont considérés **ici** comme des terminaux.)

Ni grammaire pour les séquences

```
Seq  ::= φ | Item Seq
```

ni une description simplifiée des expression algébriques

```
E  ::= E Op E | '(' E ')' | Atm
Op ::= + | - | * | /
```

ne sont pas des grammaires de précédence. La forme **`E Op E`** contient même trois non-terminaux adjacents. Mais on peut la transformer en forme

```
E  ::= E + E | E - E | E * E | E / E | '(' E ')' | Atm
```

Ceci est contraire à la philosophie du parsing dirigé par la syntaxe, c'est à dire à la correspondance simple et immédiate entre l'analyseur et la grammaire. Lors de la conception du langage toute simplification et factorisation de la grammaire sont indispensables. Il *faut* extraire toute généricité de la grammaire, sinon les règles deviennent longues et illisibles. Le parseur doit également être compact, sinon son déboguage risque d'être pénible, et de plus, la construction d'un analyseur que vérifie dans un **switch** une vingtaine de cas particuliers qui structurellement sont presque des jumeaux, est très décourageant.

En décrivant la stratégie du parsing par les règles de précédence nous pouvons presque «oublier» les productions syntaxiques. Il n'y a plus de non-terminal *Expression* qui appelle*Terme* qui appelle *Facteur*, etc., et ainsi toutes les multiplications à l'intérieur d'un terme additif sont réduites avant de toucher à un opérateur additif. Ici nous dirons simplement qu'un opérateur additif a une précédence plus faible que celle d'un opérateur multiplicatif.

Plus concrètement : introduisons trois *relations de précédence*, existantes entre *certaines* paires d'objets **terminaux**, qui jouent le rôle d'opérateurs dans le langage : $\prec$, $\succ$ et $\doteq$. Si $a \prec b$ on dit que a a la précédence plus faible que celle du b. Par exemple, $+ \prec *$, ou $/ \succ -$. ***Attention,*** ces relations en principe peuvent ne pas respecter pas des propriétés de relations d'ordre connues en algèbre. En particulier, il est possible d'avoir dans un langage $a \prec b$ et $a \succ b$ en même temps. Il n'est pas sûr que $a \doteq a$.

Si dans le texte analysé le parseur découvre une séquence $\alpha\ \beta$ telle, qu'entre les terminaux α et β *aucune* relation de précedence n'a pas été définie, ceci est une erreur, la combinaison de ces deux terminaux est illégale. Exemple : deux nombres qui se suivent dans une expressions, ou deux mots-clés en dehors des structures de contrôle bien formées. En général le tableau des précédences est assez creux. Nous proposons alors une simplification de la stratégie de précédences générale.

L'ensemble d'objets terminaux d'un langage se divise en deux sous-catégories : les *données* et les *opérateurs*. On peut considérer qu'une donnée est un opérateur très spécifique, mais c'est inutile. Une donnée est un objet *qui ne figure pas dans la liste des opérateurs*.

Un opérateur dont la précédence relative par rapport à un autre est plus haute, est «plus fort», et attrappe en priorité l'argument entre les deux.

La stratégie du parsing opérationnel peut se réduire aux règles suivantes.

- On définit deux piles, la pile des données, et la pile des opérateurs.
- Initialement la pile des données est vide, et la pile des opérateurs contient un opérateur «bidon» (marqueur) de très faible précédence.
- En consommant le flot on trouve les données et les opérateurs. Une donnée est toujours empilée. Un opérateur est comparée avec le dernier opérateur empilé, et si ce dernier est plus faible, le nouvel opérateur est empilé. Si le nouveau est pluis fort, l'opérateur déjà empilé est réduit, avec ses argument, et un noeud est formé.
- Après chaque réduction on continue avec la réduction (peut être il faut réduire plusieurs opérateurs empilés).
- Quand l'expression se termine, on dépile le reste.

Commençons par la définition de nos arborescences syntaxiques, mais aussi d'un type qui définit l'opérateur : sa précédence, et associativité : gauche, droite ou aucune. Construisons aussi une lise d'opérateurs.

```
data Assoc = Lft | Rgt | Non deriving (Eq,Ord)
termop = ("$$$$",0,Non)      -- Opérateur bidon, faible

infops =[("^",8,Rgt),("*",7,Lft),("/",7,Lft),("quot",7,Lft),
         ("+",6,Lft),("-",6,Lft),("++",5,Rgt),
         ("<",4,Non),(">",4,Non),("<=",4,Non),("==",4,Non),
         ("elem",4,Non), termop]
```

Rappelons ge Haskell permet l'usage d'une fonction binaire quelconque comme l'opérateur, à condition de mettre son nom entre apostrophes inversés (*backquotes*). Donc, un opérateur est un tuple qui contient le nom, un attribut entier, et une propriété de type **`Assoc`**.

La fonction qui trouve un opérateur dans la liste **`infops`** (ou échoue) est une simple boucle :

```
findop x (p@(y,a,b):q) | x==y = Just p
                       | otherwise = findop x q
findop x [] = Nothing
```

mais nous pouvons envisager une stratégie plus efficace. En fait, on peut mettre les opérateurs dans une table des symboles globaux – hachée ou arborescente.

Voici la définition des arbres :

```
data Gentree = None | Lf String | Nod String Gentree Gentree
  deriving Eq

instance Show Gentree where
 showsPrec _ None = showString "()"
 showsPrec _ (Nod op a b) =
   showChar '(' . shows op . shows a . shows b . showChar ')'
 showsPrec _ (Lf a) = shows a

constrnod (op,_,_) x y = Nod op x y
```

(La dernière fonction est une abréviation). Pour simplicité, les feuilles sont des chaînes, laissons au lecteur de rétablir toute la structure algébrique déjà discutée, avec des nombres, variables, etc. Le parseur qui trouve un opérateur est :

```
infixop = spaces >> (many opchar) >>= \o ->
  let a = findop o infops
  in case a of
    Nothing -> fail ""
    Just oper -> return oper
```

Voici la définition d'un parseur primitif, atomique, et d'un objet «primaire» – atomique ou parenthésé. La généralisation aux cas plus sérieux est triviale, et en tout cas ceci a déjà été discuté.

Ajoutons à cela la définition de la fonction qui compare les précédences, et qui répond à la question si le premier argument est plus fort que le second.

```
atomp = spaces >> (many letter)
primp = spaces >> (brack lpar opexpr rpar # transf Lf atomp)

domin (_,pl,asl) (_,pr,asr)   -- nom n'a pas d'importance
  | pl >  pr = True
  | pl <  pr = False
  | pl == pr = (asl==Lft)
```

Finalement, le parseur principal. Le lecteur doit lire soigneusement sa définition et essayer de le comprendre.

```
opexpr = shift [termop] [] where
  shift opstack dstack = primp >>= \x -> pseq opstack (x:dstack)
  pseq opstack dstack =
        (infixop >>= reduce opstack dstack) # e_reduce opstack dstack
  reduce ops@(lastop:rops) dstack op
     | domin lastop op = reduce rops (cst lastop dstack) op
     | otherwise = w (op:ops) dstack
  e_reduce (lastop:rops) dstack@(top:_)
     | lastop == termop = return top
     | otherwise = e_reduce rops (cst lastop dstack)
  cst op (x:y:rdat) = (constrnod op y x) : rdat
```

Les précédences sont des «forces d'attraction» exercées par l'opérateur à sa gauche et à droite, selon l'associativité. D'autres convention que celle adoptée ci-dessus existent, par exemple, au lieu de préciser un attribut `Assoc` spécifique, nous pouvons affecter à chaque opérateur deux précédences : gauche et droite. Si la précédence droite est plus grande que la gauche, cela implique l'associativité à **gauche**. Le parseur est itératif.

La technique peut être généralisé à l'extrême : pratiquement toute la structure syntaxique peut être représentée par les opérateurs. `if`, `then` etc. peuvent être des opérateurs, et les parenthèses aussi ! Ceci demande des techniques décisionnelles assez compliquées. Par exemple, une parenthèse ouvrante «vue de gauche» a une précédence si forte, qu'elle est toujours empilée. Mais elle force la réduction de *tous* les opérateurs à droite, jusqu'à la parenthèse fermante (où les deux parenthèses s'annulent réciproquement, et ne génèrent aucun code). En général les techniques opérationnelles seules sont difficiles à déboguer, et elles sont utilisées éventuellement en combinaison avec des techniques plus orientées vers une grammaire fixe. Elles doivent être complétées par les procédures de vérification de légalité des constructions analysées. En général il ne faut pas s'appuyer trop si le langage est complexe. Il existe un algorithme qui lit une grammaire et qui en déduit les précédences des opérateurs, mais cette stratégie est rarement exploitée. Répétons : l'application la plus fréquente est la possibilité d'élargir la syntaxe des langages existants, s'ils prévoyaient les déclarations des opérateurs et de leur précédences.

6.3 Exercices

Q1. Compléter le parseur des expressions arithmétiques par le module qui reconnaît les appels fonctionnels : `fun(e1,e2,...,en)`.

R1. La **première** question qui doit être traitée est : quel est le résultat fourni par ce parseur? Nous devons étendre notre code-cible (arborescent ou postfixe) par un opérateur d'«appel» procédural. La partie analytique est si simple que nous la laissons au lecteur : identificateur et une liste d'arguments entre parenthèses. Rien de nouveau.

Q2. Comment optimiser la création du code postfixe par le parseur des expressions arithmétiques en évitant la création de nombreuses listes éphémères recopiées plusieurs fois par `(++)`.

R2. Ce problème a déjà été discuté. Si on construit une fonction d'aplatissement qui parcourt un arbre binaire et qui concatène le résultat de l'aplatissement avec un deuxième argument «tampon», la récursivité en

cascade se transforme en linéaire, et **(++)** disparaît. Mais la réalisation de cet algorithme n'est pas triviale. Ce «tampon» (la suite du code) doit être présent dans *tous* les parseurs.

Q3. Construire un scanneur de mots (alors un parseur vraiment *très* simple), qui ignore (mais pas totalement !) les espaces et les fins de ligne, et qui retourne les mots accompagnés par la *position* : le numéro de ligne et le numéro de colonne du premier caractère du mot. Ceci est indispensable pour le déboguage du programme.

R3. On peut reformuler le scanneur des mots en lui ajoutant un compteur spécial, incrémenté chaque fois quand un caractère est consommé. La construction explicite n'est pas compliquée, mais assez pénible : on compte toujours en consommant les lettres, mais on n'attache pas le compteur à chaque lettre du mot, seulement à la première.

La stratégie la plus universelle consiste à modifier le type décrivant le flux d'entrée : au lieu d'avoir une chaîne, l'objet de type **String**, nous définissons

```
type Flux = Fl (Int,Int) String
```

où les deux nombres entiers dénotent la ligne et la colonne courantes. Le parseur primitif que nous avons appelé **item** aura la forme :

```
item  = Pa (\(Fl (y,x) inp) -> case inp of
              []     -> []
              (x:xq) -> [(x, case x of
                    '\n' -> Fl (y+1,0)   xq
                    '\t' -> Fl (y,x+8)   x
                    _    -> Fl (y,x+1)   xq )] )
```

Bien sûr, le tabulateur peut être interprété différemment, et nous pouvons ajouter quelques caractères de contrôle, y compris le «backspace», mais c'est inutile.

On peut faire beaucoup de choses concernant la position, à condition que *toute* consommation du flux passe par ce parseur. Voici un parseur-observateur qui ne consomme pas le flux, mais qui rapporte la position actuelle.

```
posit = Pa (\imp@(Fl pos str) -> [(pos,imp)] )
```

Si maintenant un parseur quelconque, par exemple **word** consomme le flux et construit une chaîne spécifique, et s'il utilise le protocole conforme avec le parseur **item**, il nous suffit de déclarer

```
wordPos = posit >>= \pos ->      – juste avant le mot
          word >>= \wrd -> return (At pos wrd)
```

Nous ajoutons ainsi l'information positionnelle là où nous voulons.

Chapitre 7

Informations complémentaires sur les parseurs descendants

7.1 Diagrammes syntaxiques

Les productions BNF ne constituent pas le seul moyen de représenter les structures syntaxiques dans un langage. Les structures particulièrement simples, itératives, s'expriment mieux par les expressions régulières ou par les automates. Ces derniers, visualisés par ses graphes de transition, offrent une technique particulièrement intuitive et élégante, qui facilite la compréhension des règles par un lecteur humain. Le codage d'un automate est une autre chose, mais en général, le dicton folklorique : «une image vaut 1000 mots» mérite une attention.

Quand le langage Pascal est né, sa première définition syntaxique popularisée par son créateur Niclaus Wirth, allait à l'encontre des archetypes établis par le langage Algol 60. Au lieu de décrire tout en BNF, ce qui était toujours possible, les auteurs ont préféré de décrire tout à travers des graphes, ou des diagrammes de transition. Chaque symbole terminal ou non-terminal est un sommet du graphe syntaxique, l'enchaînement séquentiel entre deux symboles devient un arc, et l'alternative est la bifurcation d'un arc (plusieurs successeurs). Les itérations (productions récursives à droite) deviennent des boucles.

Les diagrammes apportent un peu d'esthétisme graphique au domaine de la compilation et facilitent la compréhension des structure syntaxiques, et nous allons montrer quelques uns, mais ils ne sont pas directement utiles pour un travail sérieux comme un outil de codage.

La Fig. (7.1) montre le diagramme pour le signe optionnel :

```
signe   ::= ϕ | + | -
```

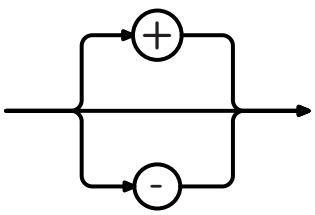


Fig. 7.1: Signe optionnel

Et voici, sur la Fig. (7.2) la construction du graphe qui représente une expression algébrique, d'abord récursive à gauche, et ensuite normalisée selon l'algorithme de Greibach, et optimisé.
Fig. (7.3) montre le résultat de la normalisation de Greibach. Il a fallu introduire un nouveau non-terminal **`trmSeq`**. Fig. (7.4) présente les optimisations de la nouvelle syntaxe.

7.2 Optimisation classique des parseurs descendants

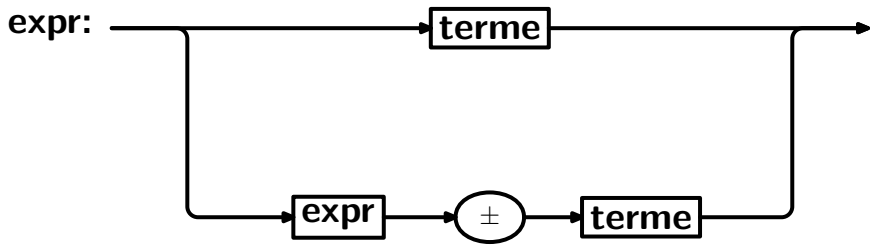


Fig. 7.2: Expression : **`terme | expr ± terme`**

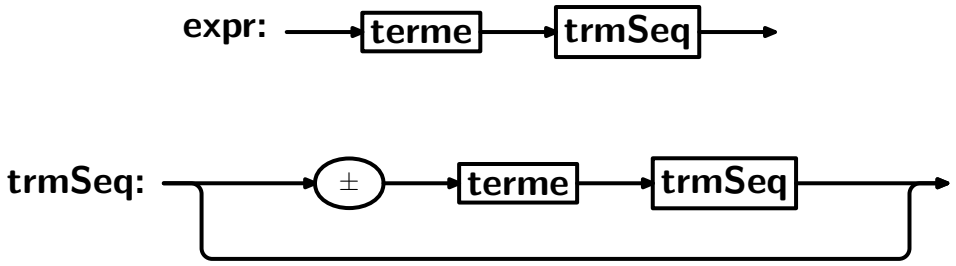


Fig. 7.3: Représentation graphique de la normalisation de Greibach

Dans les sections suivantes nous aborderons encore les techniques ascendantes LR. Mais la technique descendante, récursive, reste toujours la plus pédagogique et mieux structurée. Elle est irremplaçable pour la construction de petits parseurs pour des petits langages bien formalisés.

Mais la stratégie descendante est par nature non-déterministe, ce qui implique une certaine inefficacité, si le langage compilé est déterministe, et le non-déterminisme de l'analyse reflète uniquement le fait que l'information sur la structure phrasale ne soit pas transmise au parseur suffisamment tôt. Répétons : le non-déterminisme signifie simplement que le parseur n'est pas suffisamment prévoyant. L'usage de la pile récursive est plus intense que nécessaire.

Le *backtracking* éventuel doit être découvert très tôt, et les parseurs présentés dans cette partie de notes sont parfaitement utilisables dans les compilateurs sérieux. Il faut naturellement préparer manuellement ou automatiquement la grammaire – éliminer la récursivité à gauche, factoriser le préfixe gauche commun, etc. Dans presque tous les cas intéressants le non-déterminisme est éliminé (le parseur devient *prédictif*) après la reconnaissance du premier item sur le flux d'entrée.

Encore une fois : si le parseur réduit la chaîne par la production **`S ::= A | B`**, il essaye **A**, et si **A** échoue, alors le *backtracking* relance la variante **B**. Si l'échec de **A** se produit après la consommation de plusieurs

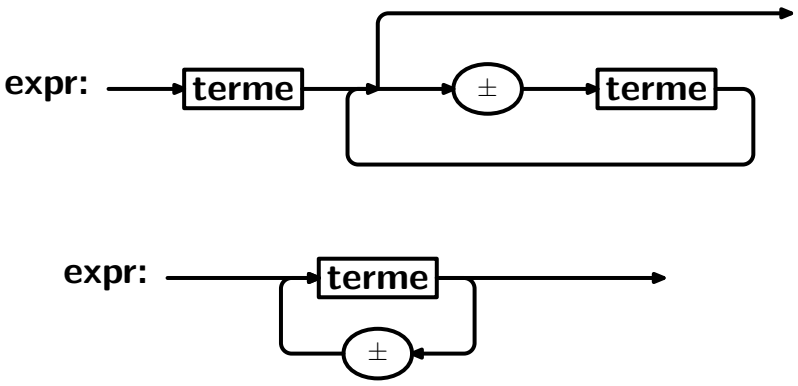


Fig. 7.4: Optimisations du diagramme : expr

lexèmes, et création de plusieurs morceaux d'arbre syntaxique, la stratégie non-déterministe est visiblement inefficace.

Cependant, si la factorisation a été faite correctement, si l'alternative plus longue précède la plus courte, etc., souvent la décision de basculer vers l'autre alternative est basée sur un simple test, ou il n'y a aucune différence entre l'alternative **`A`** ou **`B`**, et l'expression : **`if test(A) then A else B`**. (Toute alternative devient exclusive).

La gestion du non-déterminisme dans un langage fonctionnel est commode. On peut «consommer» la tête du flux d'entrée, mais rendre le même flux intacte à un autre module du parseur. Un parseur est un *objet* qui peut être combiné avec autres objets de la même catégorie (mais pas forcément du même type ; il suffit de regarder les exemples).

Les techniques de programmation impérative sont plus brutales. Il n'y a pas de flux d'entrée, mais une procédure de lecture qui consomme une partie du buffer d'entrée, et cette action est extérieure par rapport au programme, elle constitue un effet de bord. Il est difficile de restaurer le contexte précédent. On a élaboré alors la stratégie de *look-ahead* : la lecture de l'item suivant passe par un double tampon, le parseur a la possibilité de regarder un peu en avant (d'habitude un item suffit), et de reconnaître un objet sans le consommer. Ainsi l'ambiguïté, par exemple l'alternative – *un facteur, ou un produit de facteurs* constitue un terme additif arithmétique – est réduite quand le parseur voit que l'item après le premier facteur est/n'est pas un opérateur multiplicatif. Mais il n'a pas le droit de le consommer s'il s'agit d'un opérateur additif, car le parseur qui construit la somme de termes en aura besoin. La technique classique d'optimisation est basée sur les éléments decrits ci-dessous.

7.2.1 Élimination de la récursivité

En fait, elle ne peut être vraiment éliminée, les productions *sont* récursives. Il s'agit simplement de construir le parseur sous forme d'une procédure non-récursive, qui manipule explicitement toutes les piles indispensables pour sauvegarder les données et le fil de contrôle.

L'élimination de l'usage de la pile système en faveur de nos piles privées est une partie mineure de la stratégie. Ce qui nous intéresse est l'invention d'un «oracle» qui nous dira quelle production alternative appliquer, comment rendre le parseur *prédictif*? Il nous faudra introduire un *tableau de pilotage* du parseur, qui spécifie la production «éligible» pour le développement d'un non-terminal.

Pour cela on introduit deux tableaux accessoires : PREMIER et SUIVANT, qui déterminent le tableau de pilotage.

Un parseur descendant prédictif contient un tableau bi-dimensionnel $M[A, a]$, où A est un non-terminal, et a dénote un terminal (ou un marqueur de fin spécial, souvent noté comme \$). Il possède également une pile capable de stocker les symboles de la grammaire. Au début on y place le marqueur \$, on on le couvre avec le symbole de départ de la grammaire.

Le programme du parseur regarde X – le symbole au sommet de la pile, et a – le lexème d'entrée. L'action du parseur est alors determinée :

- Si $X = a = \$$, alors le parseur s'arrête.
- Si $X = a$, mais il est différent de \$, alors le parseur a reconnu un symbole littéral. X est dépilé et la lecture du flux d'entrée progresse d'un item.
- Si X est un non-terminal, le programme consulte $M[X, a]$. Ceci peut être une *production*, ou la signalisation d'erreur. Si la production a la forme **`X ::=`** $\alpha\beta$, X est dépilé, et il est remplacé par $\beta\alpha$. Une procédure sémantique construit le résultat du parsing.

Bien sûr, si le programme trouve un élément du tableau de pilotage qui correspond à une configuration illégale, l'analyse s'arrête et le parseur essaie de se calmer en cherchant un terminateur ou un autre terminal de synchronisation. Le tableau SUIVANT peut être utile dans ce contexte.

7.2.2 Tableaux PREMIER et SUIVANT

(Dans la littérature anglophone ils s'appellent : FIRST et FOLLOW.)

- Pour toute chaîne de terminaux et non-terminaux α qui après son développement se transformera en chaîne terminale, définissons une fonction PREMIER(α), et cette fonction définit l'ensemble de tous les terminaux qui peuvent se trouver au début de la chaîne réduite.

- Il est évident que le premier symbole de la chaîne α détermine cet ensemble, alors il suffit de construire un *tableau* PREMIER(X), où X est un symbole quelconque de la grammaire. Pour un symbole terminal P, PREMIER(P) se réduit à $\{P\}$. Si une production vide pour X existe, il faut ajouter ϕ à PREMIER(X).

- Nous avons menti. Si X peut se réduire à la chaîne vide, alors PREMIER(XY) dépend de PREMIER(Y), et n'est pas déterminé par seul X. Il faut continuer la construction.

- Pour un non-terminal X on trouve PREMIER(X) en regardant la production $X \rightarrow Y_1 Y_2 \ldots Y_k$, et en calculant PREMIER pour la chaîne à droite.

- Pour chaque non-terminal A on construit un autre tableau, SUIVANT(A), qui contient l'ensemble de tous les terminaux qui peuvent apparaître à droite de A dans une phrase, c'est à dire peuvent suivre le développement de A. S'il existe une dérivation (finale) : $S \Rightarrow \alpha A a \beta$, où a est un terminal, alors a appartient à SUIVANT(A).

- La construction de SUIVANT procède récursivement, en réduisant les productions. Si dans la production ci-dessus a n'est pas un terminal, on récupère son PREMIER.

- Si A figure dans la production $A \Rightarrow \alpha B$, tout dans SUIVANT(A) se retrouvera dans SUIVANT(B). Les règles sont vraiment simples.

Voici la construction du tableau de pilotage M. Il faut exécuter les opérations suivantes pour toute production **A ::=** α de la grammaire :

- Pour chaque terminal a dans PREMIER(α), ajouter **A ::=** α à $M[A, a]$.

- Si la chaîne vide ϕ se trouve dans PREMIER(α), il faut ajouter cette production **A ::=** α à $M[A, b]$, pour tout terminal b dans SUIVANT(A).

  Faire la même chose pour le marqueur $ – s'il se trouve dans SUIVANT(A), il faut ajouter la production courante dans $M[A, \$]$.

Tout élément du tableau qui n'est pas défini, signale l'erreur du parsing.

7.3 Exercices

Q1. Essayer de construire un tableau de précédences permettant de structurer les expressions conditionnelles **if … then … else …** en considérant tous les mots-clé comme des opérateurs.

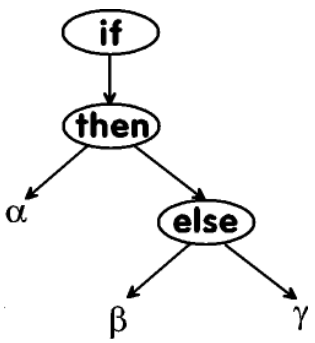


Fig. 7.5: Arbre d'une expression conditionnelle

R1. C'est facile, sachant que toutes les expressions algébriques, etc. doivent être liées par des opérateurs «plus forts». Le mot **if** est un opérateur préfixe, et les autres sont infixes. Mais on peut former des arbres alternatifs, p. ex. celui sur la Fig. (7.5), et l'autre, avec **then** et **else** échangés. Ce qui compte c'est la génération d'un code correcte, mais aussi la possibilité de reconnaître des erreurs syntaxiques de tout genre. Analysez cette question.

Q2. Construire les tableaux PREMIER et SUIVANT pour une grammaire classique qui décrit les expressions arithmétiques avec les 4 opérations et les parenthèses.

R2. Ceci est un bon sujet d'examen. Pas de réponse ici.

Chapitre 8

Stratégie ascendante d'analyse syntaxique

8.1 Idée générale

Nous avons essayé de démontrer que les parseurs construits par des techniques fonctionnelles combinatoires – qui formellement appartiennent à la catégorie des parseurs *descendants*, structurellement correspondent aux productions de la grammaire du langage. Ainsi, la construction du parseur est plus statique. La technique descendante (fonctionnelle ou autre) convient bien à la construction manuelle des parseurs. Rappelons nous pourquoi elle est descendante. Une production type

```
S :: a A B e
```

correspond à un parseur **S** qui s'applique au flux de données courant, et construit son fragment de l'arborescence syntaxique finale. Son nom S peut être considéré comme l'étiquette de la racine de cet arbre, et par cette racine la construction commence. Le parseur **S** appelle ses composantes **a**, **A**, etc., terminales ou non-terminales qui construisent les sous-arbres et les feuilles. La récursivité des définitions syntaxiques se traduit clairement par la récursivité des parseurs.

Une production qui contient des alternatives est équivalente à plusieurs productions «parallèles». La *définition* de la règle possède des alternatives, mais une dérivation concrète de l'arbre doit être unique, dans chaque contexte une seule alternative doit être applicable, sinon le langage est mal conçu.

Cependant, une autre **stratégie** analytique est également possible. Cette technique, qui gravite autour des symboles cryptiques comme grammaires LR(1), stratégies LALR etc. est reconnue par son universalité et efficacité dans le domaine de compilation des langages classiques. Cependant, elle est difficile à implanter «à la main», les parseurs construits par les méthodes ascendantes d'habitude sont des résultats du travail d'un générateur de parseurs, et ils sont lourds. Nous allons présenter ici deux variantes de la stratégie ascendante : les grammaires d'opérateurs, qui a l'avantage de pouvoir enrichir la syntaxe des langages existants, et la stratégie générale LR propice à la construction des générateurs de parseurs. ***Attention, les grammaires d'opérateurs seront à peine mentionnées – un exemple de leur usage dans le monde de parseurs monadiques se trouve dans un des chapitres précédents.***

L'idée générale de la technique ascendante est la suivante. Prenons encore une fois la production exemplaire écrite ci-dessus, mais complétons-la par la définition des composantes.

```
S  ::= a A B e
A  ::= A b c | b
B  ::= d
```

Si le flux d'entrée contient la chaîne «**abbcde**», un parseur ascendant commence la création de son arbre par les feuilles. Il déplace sa «tête de lecture» et consomme le lexème **a**. Cette opération s'appelle en jargon «*shift*» (décalage).

Avant de procéder, le parseur vérifie s'il peut faire quelque chose avec la feuille acceptée. S'il y avait une règle `X ::  a`, le parseur aurait pu construire le graphe **X** ← **a**, mais avec notre grammaire il est obligé de relancer le *shift* et consommer **b**. Cette fois il est possible de construire **A** ← **b**. Cette opération s'appelle

«*reduce*». Elle ne consomme rien, seulement remodèle l'état interne du parseur, en réduisant les sous-arbres déjà stockés sur une pile interne en leur nœud-père. Souvent une réduction est suivie par une autre réduction, et c'est ici que le compilateur déclenche l'exécution des procédures sémantiques.

«Virtuellement» notre chaîne est équivalente à **aAbcde**, et dont les deux premiers items se trouvent déjà dans le parseur, sur la pile. Il n'y a pas de production **X ::= a A**, alors le parseur est obligé d'exécuter le *shift*. L'item **b** suivant peut être réduit à **A**, comme avant, mais supposons qu'un oracle magique informe le parseur de la possibilité de faire une autre réduction, à condition de consommer encore quelques items. Ceci s'appelle la résolution du conflit *shift-reduce*. Quand le *shift* suivant résulte en séquence **aAbc** sur la pile, le parseur la réduit à **aA**. Le reste est immédiat : **d** se réduit à **B**, **e** est consommé, et la pile qui contient **aABe** donne **S** ce qui termine l'analyse.

Donc, toute difficulté pour implanter une telle stratégie d'analyse consiste à identifier la «poignée» (ang. *handle*), la chaîne d'items qui doit être consommée afin d'effectuer une réduction.

- Quelle production choisir?
- Comment résoudre les conflits entre *shift* et *reduce* simultanément possibles?

Il nous faudra aussi répondre à la question comment résoudre le conflit entre deux réductions alternatives possibles (conflit : reduce-reduce). Les techniques ascendantes utilisent les tableaux de pilotage, comme le parseur descendant prédictif décrit dans la section consacrée à l'optimisation des parseurs descendants. Ces tableaux de pilotage constituent l'oracle dont nous aurons besoin.

8.2 Grammaires d'opérateurs

Rappelons que formellement, une grammaire de précédence, qui constitue la base de la technique ascendante avec des opérateurs, se caractérise par deux exigences :

- il n'y a pas de production dont la partie droite est la chaîne vide ϕ , et
- aucune production ne contient deux non-terminaux adjacents.

La forme

```
E  ::= E + E | E - E | E * E | E / E | '(' E ')' | Atm
```

possède cette structure, mais pour enlever les ambiguïtés, chaque objet spécial comme **+** ou les parenthèses, doit être équipé avec des priorités (précédences) et l'attribut de l'associativité. Un tel objet portera le nom d'opérateur, et ce qui reste, ce sont des données : termes, atomes, etc.

Dans la section précédente nous avons défini la précédence des opérateurs comme *un* nombre entier, et l'associativité (gauche, droite ou aucune) était un attribut séparé. Mais les deux peuvent constituer une seule propriété, la précédence, avec deux champs : à gauche et à droite. Une donnée x qui se trouve entre deux opérateurs, par exemple $A\ x\ B$, après avoir été empilée, participe à la réduction suivante. Si la précédence droite de A et plus grande que la précédence gauche de B, la pile des données (contenant x) est réduite, sinon B est empilé. Si les précédences sont égales, on peut effectuer des opérations spéciales, par exemple réduire les deux opérateurs à la fois.

La construction d'un parseur complet basé sur ces principes constitue un joli exercice, mais le déboguage d'un tel parseur pose quelques problèmes...

Voici un perseur opérationnel basé sur cette stratégie. Il est plutôt rudimentaire, et il possède la même structure qu'un parseur déjà vu dans cec notes, dans la section (6.2). On commence par la définition du tableau de précédences. On considère que le marqueur $ initialize la pile. Notez comment représente-t-on les parenthèses.

```
optab=[("=",31,30),("+",49,50),("-",49,50),("*",59,60),
       ("/",59,60),("(",-1,10),(")",10,-1),("$",0,0)]

data Op = Nil | O String Int Int
marq = O "$" 0 0

parse l = shift [] [marq] (l ++ ["$"])
```

```
shift datstack operstack (c:q) = case (assoc c optab) of
  Nil -> shift ((F c):datstack) operstack q
  op  -> reduce op datstack operstack q

reduce op@(O nm lf ri) datstack opst@((O np l1 r1):oq) q
  | lf < 0   = shift datstack (op:opst) q
  | lf >r1   = shift datstack (op:opst) q
  | lf <r1   = let (a:b:dq)=datstack
               in reduce op ((Nd np b a):dq) oq q

  | lf==r1   = if nm=="$" then (head datstack,q) else shift datstack oq q
```

Tout le débogage, les actions sémantiques, etc., sont laissées à la discrétion du lecteur qui un jour trouve besoin d'exploiter cette stratégie.

8.3 Parseurs LR

Le parseur LR est un automate à pile, qui peut être codé une seule fois, indépendamment du langage. Ainsi Yacc en lisant la grammaire (décorée par les opérations sémantiques) *ne construit pas la procédure-parseur*. Cette procédure, ou plutôt son squelette, est *prédéfinie*, et stockée dans le générateur. Yacc construit des tableaux de pilotage, les «scripts» statiques contenant des instructions pour l'automate. Les tableaux sont ajoutés au squelette, «décorés» par les actions sémantiques prévues par l'utilisateur, et le parseur est généré en sa forme-source, qui doit encore être compilée par le compilateur C.

Les avantages de la stratégie LR sont nombreux, elle est universelle et efficace. Les erreurs sont découvertes relativement tôt. On peut prouver que la classe de langages reconnus par les techniques LR (basées sur les *grammaires* LR) est plus riche que celle décrite par les grammaires LL (et techniques descendantes). Bien sûr, *a priori* il n'est pas sûr si cette richesse est vraiment utile, des langages vraiment complexes et atypiques sont rares.

Mais la stratégie LR n'est pas facile à implanter, et nous allons seulement présenter la structure générale de l'algorithme. La construction complète d'un parseur LR est un peu longue. Nous ne voulons pas non-plus de décrire dans ces notes le générateur Yacc (ou Bison). Les lecteurs qui le veulent utiliser disposent de la documentation officielle, librement accessible partout.

Comme il a été dit, l'algorithme général du parsing ne dépend pas de la grammaire. Nous allons alors construire d'abord un automate général à pile, et la discussion de la construction des tableaux sera faite ultérieurement.

Le programme opère sur une pile qui peut stocker les symboles de la grammaire (terminaux, et – après leur réduction – les non-terminaux), ainsi que quelques symboles spéciaux qui représentent **l'état** de l'automate. Si s_k dénote un état, et X_i un symbole syntaxique, la pile contient la séquence $s_0 X_1 s_1 X_2 \cdots X_m s_m$, où s_m est le sommet. Dans une implantation concrète on peut éviter de stocker les symboles de la grammaire sur la pile ; ceci peut être restauré depuis le contexte, mais ainsi la présentation de l'algorithme (selon Aho, Sethi et Ullman) est plus facile.

Le tableau de pilotage est composé de deux parties, nommées conventionnellement : **ACTION** et **GOTO**. Les éléments du segment **ACTION** contiennent une parmi les quatre valeurs possibles :

- **`shift s`** – l'action de décalage. Ici **`s`** dénote le nouveau état de l'automate qui sera empilé ;
- **`reduce`**, action de réduction par la production $A \to \beta$;
- **`accept`** – la terminaison du parsing, et
- **`error`** – puisque personne n'est parfait...

Le tableau **ACTION** est indexé par l'état, et par un symbole terminal. Le programme du parseur après avoir consommé l'item suivant de son flux d'entrée, disons, a_i, consulte l'élément **`ACTION[`**s_m, a_i**`]`**, où – comme il a été dit – s_m est le sommet de la pile. Le flux contient les items en attente de consommation : $a_{i+1} a_{i+2} \ldots$.

Le tableau **GOTO** est indexé par l'état et un symbole syntaxique, et génère un nouvel état. C'est la fonction de transition de l'automate.

Si l'élément consulté contient **`shift s`**, l'automate passe à la configuration

$$(s_0 X_1 s_1 X_2 \cdots X_m s_m a_i s)$$

et procède à consommer a_{i+1} etc.

Si **`ACTION[`**s_m, a_i**`]`** est **`reduce`** $A \to \beta$, le programme vérifie la production en question et récupère r, la longueur (le nombre d'items) de β. Ensuite r états, et r symboles syntaxiques sont dépilés. L'état s_{m-r} devient le sommet, les états s_m, s_{m+1} etc. disparaissent, et les symboles X_m, X_{m+1} etc. se réduisent à A. Le flux ne change pas, le lexème suivant reste a_i, mais la configuration de la pile devient

$$(s_0 X_1 \cdots X_{m-r} s_{m-r} A s)$$

où s est le contenu de l'élément **`GOTO[`**s_{m-r}, A**`]`**. Ceci est presque tout. la signification des entrées **`accept`** et **`error`** est intuitive.

Exemple. Reprenons notre grammaire d'expressions arithmétiques, et plus concrètement l'ensemble de règles :

```
(1) E   ::= E + T
(2) E   ::= T
(3) T   ::= T * F
(4) T   ::= F
(5) F   ::= '(' E ')'
(6) F   ::= id
```

Le tableau de pilotage de cette grammaire aura la structure présente sur la Fig. (8.1).

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig. 8.1: Tableau de pilotage de la grammaire algébrique

Son analyse n'est pas difficile quand on comprend intuitivement la stratégie. Peut-être un exemple concret de parsing aidera le lecteur.

Effectuons l'analyse de **id * id + id**. Fig. (8.2) montre la séquence d'états et la pile de la machine.
Avant de passer aux tableaux, remarquons que nos parseurs, descendants, et ascendants LR partagent une inefficacité apparente générée par la structure de la grammaire. La séparation de l'expression en séquence de termes, du terme en facteurs, etc. définit correctement les précédences et l'associativité des opérateurs correspondants. Une grammaire style

```
Expr ::= Atome | Expr + Expr | Expr * Expr | Expr / Expr
```

etc. serait ambiguë. Mais la réduction d'un atome en facteur, ensuite en terme, qui finalement aboutit à la réduction en expression *est* un gaspillage de temps.

Les techniques classiques du parsing détestent toute ambiguïté. On peut traiter la grammaire mentionnée ci-dessus, si aux attributs standard, spécifiés par les productions, et alors par le contexte d'utilisation de tel out tel symbole syntaxique *on ajoute les précédences et l'associativité*. La combinaison de la technique LR régulière et les précédences permet la construction des parseurs rapides et efficaces, mais cette construction est délicate et difficile.

	Pile	Flux	Action
(1)	0	id*id + id $	shift
(2)	0 id 5	*id + id $	reduce: F → id
(3)	0 F 3	*id + id $	reduce: T → F
(4)	0 T 2	*id + id $	shift
(5)	0 T 2 * 7	id + id $	shift
(6)	0 T 2 * 7 id 5	+ id $	reduce: F → id
(7)	0 T 2 * 7 F 10	+ id $	reduce: T → T*F
(8)	0 T 2	+ id $	reduce: E → T
(9)	0 E 1	+ id $	shift
(10)	0 E 1 + 6	id $	shift
(11)	0 E 1 + 6 id 5	$	reduce: F → id
(12)	0 E 1 + 6 F 3	$	reduce: T → F
(13)	0 E 1 + 6 T 9	$	reduce: E → E+T
(14)	0 E 1	$	accept

Fig. 8.2: Exemple d'analyse LR

8.3.1 Construction des tableaux de parsing

Dans la littérature courante on présente trois méthodes différentes de construction des tableaux de pilotage, la méthode SLR (*simple* LR), la méthode dite canonique, et LALR – *lookahead* LR, la technique utilisée en pratique, qui permet de traiter quelques cas en dehors de la stratégie SLR, et qui partage avec elle une certaine simplicité du résultat. Pour un langage de complexité de Pascal le nombre d'états générés par SLR et LALR est de quelques centaines. La méthode canonique engendrera dans ce cas un automate à plusieurs milliers d'états.

Nous aurons besoin de la notion de «grammaire augmentée», qui ajoute à une grammaire donnée avec le symbole de départ S, une production extra : **`S'  ::= S`**, et où **`S'`** est le nouveau symbole initial. Quand cette nouvelle production est réduite, le parseur s'arrête. On n'a pas besoin d'action spéciale *accept*. En fait, les actions *error* sont redondantes elles aussi, on peut considérer l'erreur comme une réduction particulière, avec l'action sémantique qui doit «calmer» le compilateur, et synchroniser ses données (et, naturellement, écrire le diagnostique, et bloquer la génération du code ultérieur, si après la découverte de la faute, la compilation continue, pour découvrir d'autres éventuelles fautes).

Dans cette version de notes *nous n'allons pas* montrer la construction des tableaux de pilotage. Ceci n'est pas très compliqué, mais pénible, et redondant de point de vue de notre philosophie. Si le lecteur a *vraiment* besoin de la stratégie LR pour le parsing, il peut utiliser un de très nombreux générateurs. S'ils ne suffisent pas, prière de contacter l'auteur personnellement.

8.4 Exercices

Q1. Comment implanter les appels fonctionnels de genre **`f(x)`**, où la parenthèse ouvrante n'est plus un opérateur préfixe, mais plutôt infixe?

R1. Tout d'abord, il n'est pas sûr que dans ce contexte la parenthèse ouvrante doit être considérée comme un opérateur infixe entre deux données : le nom de la fonction, et les arguments. On peut toujours considérer un identificateur comme un opérateur, et toute forme **`x y`** comme l'appel fonctionnel équivalent à **`x(y)`**.

Alors, comment forcer l'interprétation d'un identificateur quelconque comme un opérateur? Il suffit d'attribuer par défaut à toutes les variables (noms) qui n'ont pas été déclarées comme des opérateurs, le statut d'opérateur de très haute précédence. Il sera donc toujours empilé, s'il suit un autre opérateur. (D'ailleurs, il doit être toujours considéré comme opérateur préfixe, car telle est la structure des appels fonctionnels standard). Mais dans la séquence **`id1 id2 + ...`** l'opérateur **`id2`** trouve déjà sur la pile un identificateur, et nous pouvons leur attribuer la propriété de parenthésage – après l'exécution de la réduction, les deux disparaîssent. Cette fois la procédure sémantique est plus élaborée que dans le cas des parenthèses, où le opérateurs disparaîssaient sans laisser des traces.

Ici la réduction peut former un appel **`(id1 id2)`**, et ensuite *le stocker à nouveau sur la pile des opérateurs, avec les mêmes attributs*. L'occurrence de la séquence **`id1 id2 id3`** construira l'appel **`((id1 id2) id3)`**.

Un opérateur de moindre précédence, l'addition par exemple, réduit la pile des opérateurs. Le dernier «objet fonctionnel» trouvé se transforme en un simple objet et passe à la pile des données. C'est tout.

Q2. Comment implanter à travers les grammaires de précédence des constructions comme **repeat** **`... instructions ...`** **until** **`condition`** en Pascal, ou **`do { ... } while ...`** en C?

R2. La situation est légèrement plus générale que celle décrite en cours, car **while** ou **until** dans les constructions mentionnées semblent être des opérateurs infixes, mais également les opérateurs de parenthésage. (En fait, **while** en C est simple, entre **do** et **while** on peut trouver une seule instruction, tandis que la construction **repeat** en Pascal est plus délicate.

Q3. Lire la documentation de Haskell et la partie du Prélude consacrée à l'affichage. essayer de comprendre la fonction **`showsPrec`**, qui est un «anti-parseur». Dans quelles circonstances nous pouvons l'utiliser pour nos messages diagnostiques, etc.?

R3. Lire la doc…

Chapitre 9

Sémantique

9.1 Grammaires attribuées et décorées

Notre philosophie d'analyse suit la règle : ***comprendre signifie construire***. Les analyseurs ne sont pas des simples machines à dire oui/non à la question : «est-ce une phrase correcte?», mais ils doivent retourner un résultat qui correspond à la phrase analysée.

Ils sont donc déjà des générateurs du code (au moins intermédiaire), et ceci signifie qu'une certaine dose de sémantique doit y être présente. Mais cette sémantique souvent est très pauvre, dans la réalité la sémantique est toujours contextuelle : le «sens» d'une sous-phrase **A** peut dépendre de son contexte **X** et **Y** dans une production **S ::= X A Y**, et de plus, quelques propriétés du résultat fourni par **A** lors de l'analyse peuvent dépendre du non-terminal **S** qui définit la production utilisée.

Commençons par l'introduction de la notion d'un *attribut* sémantique qui caractérise tout terminal et non-terminal du langage. En particulier tout symbole de la grammaire possède un ou plusieurs attributs, et toute construction syntaxique peut propager les attributs soit de droite à gauche : les propriétés de **A** déterminent les propriétés de **S**, soit à l'envers, de gauche à droite. Dans le premier cas on parle d'attributs *synthétisés*, dans le second – d'attributs *hérités*. Les attributs peuvent aussi se propager entre les symboles à droite d'une production, ils sont alors aussi appelés hérités.

Il faut préciser que la séparation entre la syntaxe et la sémantique, entre le parsing et l'analyse sémantique qui l'accompagne, est *un peu* une question de convention. Nous avons déjà *incorporé* des actions sémantiques dans nos parseurs. L'avantage de séparer cette partie d'analyse comme une catégorie conceptuellement indépendante nous permet surtout de

- Préciser la sémantique de manière statique et homogène ; parler de propriétés des objets syntaxiques indépendamment des programmes d'analyse. Quand la technique de construction de parseurs était principalement impérative, la sémantique cachée dans les procédures d'analyse était illisible. Ceci n'est plus le cas dans le cas des parseurs fonctionnels modernes, ou des parseurs écrits dans un langage logique, les actions sémantiques sont beaucoup plus claires.

- Grâce à cette description statique, nos actions sémantiques peuvent piloter de manière homogène un parseur construit par la technique LR, où nous spécifions les productions, mais nous ne construisons pas les procédures du parsing qui sont générées automatiquement.

  Ceci est très utile aux fans de Yacc, mais nous ne pouvons consacrer ici trop de temps à ce modèle.

Ceci est tout concernant l'«indépendance» de la sémantique et la syntaxe. Il va de soi que si la sémantique est traitée à travers des divers attributs, et ceux-si sont attachés aux symboles de la grammaire, et que les actions sémantiques sont pilotées par les productions syntaxiques, les deux parties de l'analyse sont intrinsèquement liées et inséparables. Passons à quelques exemples.

9.1.1 Valeurs des nombres

Rappelons les règles syntaxiques qui définissent un nombre entier par la concaténation des chiffres, ou N signifie un nombre (entier), et C – un chiffre :

```
N ::= C N
N ::= C
```

mais si à présent nous devons attacher des significations concrètes à tous les symboles ci-présents, il faut désambiguër la double occurrence du mot «**nombre**», puisque évidemment ***le*** **nombre** à gauche, et celui à droite ne sont pas identiques : celui à gauche contient un chiffre de plus. Traditionnellement nous pouvons indexer les non-terminaux ambigus. Nous allons aussi numéroter les productions.

```
(1) N0 ::= C N1
(2) N  ::= C
```

Le choix de la récursivité droite joue ici un rôle très important (et nous savons déjà que ce choix n'est pas judicieux…). À présent il faut affecter aux variables syntaxiques quelques attributs sémantiques. Sans doute, l'attribut principal est la *valeur numérique* d'un nombre. Considérons que chaque symbole de la grammaire est un record qui possède plusieurs «champs» – un pour chaque attribut. Ainsi, $N_0.v$ sera la valeur du nombre à gauche de la production (1).

Nous avons vu que le poids d'un chiffre dépend – évidemment – de la position dans la chaîne, et donc sa valeur *relative* dépend de la longueur de la chaîne à sa droite. Cette constatation nous permet d'établir la nécessité d'autres attributs : la longueur d'une chaîne, et la valeur relative d'un chiffre. Il devient alors évident, que les identités suivantes, attachées à la production (1) ont lieu :

$N_0.v = C.v + N_1.v$
$C.v = C.c \cdot 10^{N_1.l}$
$N_0.l = N_1.l + 1$

où $C.c$ est la valeur absolue du chiffre, son *code* numérique (ASCII-48 ou autre). La production (2) donne

$N.v = C.v$
$C.v = C.c$
$N.l = 1$

et si notre système de compilation est capable d'accepter de telles identités (ou instructions), il est également capable de synthétiser automatiquement les instructions sémantiques qui seront ajoutés au parseur. Ceci est fait par Yacc (avec sa syntaxe spécifique, et ses contraintes).

Mais nous savons que les règles

```
(1) N0 ::= N1 C
(2) N  ::= C
```

sont mieux adaptées à la construction de la valeur finale. Nous avons discuté la normalisation de Greibach, etc., mais cette fois nous sommes intéressés uniquement par la sémantique. La règle (1) spécifie : $N_0.v = 10 \cdot N_1.v + C.c$, et c'est tout, si le parseur (p. ex. ascendant LR) est capable de gérer la récursivité, il pourra s'occupper de la sémantique également.

Les attributs hérités apparaissent quand on introduit la normalisation, et les règles changent de forme :

```
(1) N  ::= C S
(2) S0 ::= C S1
(3) S  ::= φ
```

et la sémantique devient

$(1) N.v = S.v$
$(1) S.h = C.c$
$(2) S_0.v = S_1.v$
$(2) S_1.h = C.c + 10 \cdot S_0.h$
$(3) S.v = S.h$

où $S.h$ dénote son attribut hérité.

9.1.2 Constance

Si le parseur est capable de synthétiser les valeurs des nombres, il peut également se rendre compte que $7 \cdot 9 = 63$, et pre-compiler cette expression sans être obligé de passer au générateur du code un arbre syntaxique comportant la multiplication. Il peut «plier» (*fold*) toute expression considérée *constante* à condition de pouvoir

effectuer les calculs. L'attribut de constance se propage depuis des feuilles vers la racine, et la propagation est bloquée seulement par la présence des opérateurs inconnus, par exemple des fonctions définies par l'utilisateur (ou autres fonctions extérieures), ou des opérateurs, dont le comportement dépend des données disponibles lors de l'exécution du programme.

On voit ici que les langages fonctionnels qui interdisent la présence des effets de bord doivent permettre une optimisation plus agressive. Mais on voit aussi que les langages paresseux *par défaut* doivent abandonner une telle optimisation.

9.1.3 Temps de vie

Il s'agit d'optimiser les ressources, par exemple les registres rapides, ou la pile. Si le compilateur est capable de prouver qu'après avoir exécuté une instruction concrète, une ou plusieurs variables ne sont plus accessibles, il peut naturellement générer le code qui alloue les mêmes zones de mémoire à des variables différentes.

Dans le monde de la programmation fonctionnelle cette analyse peut avoir un autre «goût». Le langage Clean spécifie des objets à *accès unique*. Si x est un tel objet, et si le programme contient la création d'un nouvel objet $y = f\ x$, après cette création x n'est plus accessible, son rôle prend y. Le compilateur peut donc générer le code qui au lieu de créer un nouvel objet, modifie l'original.

9.1.4 Formatage 2-dimensionnelle des formules mathématiques

Présentons ici un exemple assez riche et instructif, mais un peu en dehors de notre vision de compilation comme d'un processus qui génère un code exécutable. Ici le texte source contient une expression algébrique classique, avec les 5 opérations arithmétiques traditionnelles, les parenthèses, et quelques fonctions comme la racine, etc. Nous pouvons y ajouter quelques fonctionnelles (au sens symbolique du terme) comme les sommes ou les intégrales.

Le «code-cible» de notre compilateur est une joli représentation graphique, bi-dimensionnelle de l'expression lue, formatée selon les règles de l'imprimerie. Ce formatage sera dirigé par la syntaxe. L'analyseur reconstruira à partir de la forme syntaxique les **attributs géométriques** de la phrase et de ses éléments : les dimensions des objets et leur position relative. La manipulation constitue une sorte de «anti-parsing»... L'objectif de cet exercice est de jouer un peu avec les attributs, et voir comment résoudre les problèmes de dépendance entre eux. Comme nous avons déjà dit maintes fois, l'élément la plus important dans la construction d'un parseur n'est pas la partie analytique, le module de reconnaissance (ils sont assez standardisés), mais le *résultat*, le «code» généré.

Ici, comme montre la Fig. (9.1), le résultat du parsing d'une expression, par exemple

```
a+b*(alpha/(beta-c*8^x) +f)
```

est une «boîte» géométrique, l'espace occupé par l'expression formatée. Le contenu de cette boîte est un

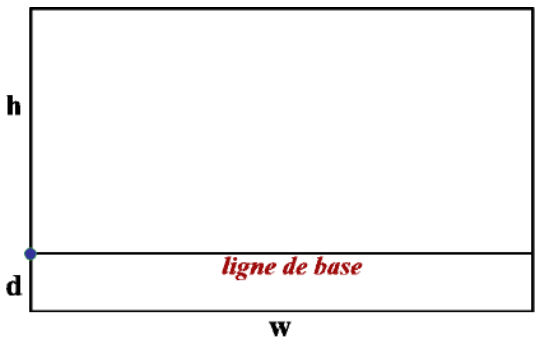


Fig. 9.1: Boîte de formatage des expressions

ensemble de boîtes imbriquées, qui finalement se terminent par des boîtes atomiques. Si nos ambitions avaient été plus grandes, nous aurions pu demander le formatage de la formule ci-dessus comme sur la Fig. (9.2), mais nous serons satisfaits aussi avec la version ASCII...

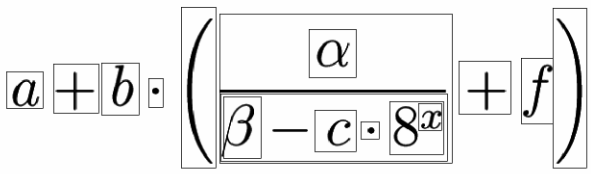


Fig. 9.2: Expression formatée professionalement, avec des boîtes

```
      /   alpha       \
a + b |---------- + f |
      |          x    |
      \beta - c 8     /
```

Rappelons que les détails du formatage ne nous intéressent pas vraiment. Nous allons donc utiliser une police de taille fixe, avec les parenthèses construites comme ci-dessus, etc. D'ailleurs, le parenthésage *est* une affaire délicate : pendant le parsing «algébrique» les parenthèses servent uniquement pour l'analyse, elles spécifient l'ordre des opérations, et c'est tout. Ici elles possèdent une forme visuelle, elles jouent un rôle aussi dans la «génération du code», et de plus, leur interprétation n'est pas évidente, parfois elles sont redondantes, comme dans `(a+b)+c`, parfois non, donc on ne peut pas toujours les considérer comme des éléments purement graphiques, si nous voulons que notre «compilateur» effectue un peu d'optimisation, et qu'il élimine des redondances.

Par convention une boîte quelconque est coupée horizontalement par la ***ligne de base***. L'extrémité gauche de cette ligne c'est le ***point de base***, qui détermine la position de la boîte par rapport à un répère. Les boîtes placées horizontalement à côté, alignent leurs lignes de base. On voit que en présence des fractions, une partie de la formule doit se trouver en dessous de la ligne de base.

Les boîtes possèdent donc les attributs géométriques suivants :

- La largeur w. Pour un atôme c'est le nombre de caractères qui le composent. Sinon, c'est la somme des largeurs des boîtes-composantes (horizontales).
- La hauteur h. Pour un identificateur, nombre, etc., la hauteur est égale à 1. La profondeur alors est égale à zéro. Sinon, c'est la somme des dimensions verticales de toutes les boîtes intérieures, au dessus de la ligne de base.
- La profondeur d. La dimension verticale sous la ligne de base est determinée par les dimensions du dénominateur de la fraction.

Ces propriétés sont synthétisés. On voit que la juxtaposition linéaire (p. ex., dans $a + b$ ajoute les largeurs des boîtes composantes, et prend le maximum de la hauteur et de la profondeur, comme des attributs de la boîte englobante. La puissance x^y peut stocker les deux boîtes composantes sur la diagonale, et une fraction $\frac{x}{y}$ construit la hauteur de la nouvelle boîte à partir du numérateur, et la profondeur – du dénominateur. La hauteur/profondeur des parenthèses s'adapte à la hauteur/profondeur de l'expression parenthésée.

On voit aussi quelques attributs hérités ! La position (d'affichage) des fragments dépend naturellement de la position de la structure complète. Ainsi dans $a + b$ la position «x»du b dépend de la largeur des éléments précédants. Dans une fraction nous aurons le décalage vertical, la position «y» du numérateur et deu dénominateur par rapport à la ligne de base, mais également, ce qui est normalement envisageable – le plus court de deux doit être centré. Naturellement, la longueur de la barre correspond à la plus grande de deux largeurs.

Le parseur construit les boîtes avec tous les attributs géométriques, et pour afficher le résultat sur une feuille ou sur l'écran, nous disposons de deux stratégies possibles ;

- Affichage «aléatoire», où chaque élément est ***dessiné***, positionné sur la page comme ses attributs le prévoient.
- Affichage séquentiel, par une imprimante ASCII. Dans ce contexte il faut ***trier*** les boîtes verticalement et ensuite horizontalement pour pouvoir placer les éléments une fois, dans l'ordre prévu, sans possibilité de reculer.

Si le paquetage peut opérer avec des polices arbitraires, modifier la taille des atomes, jouer avec la largeur des espaces blancs entre les items, la situation est plus complexe. Il faudra alors tenir compte du fait que les exposants et les indices utilisent la police plus petite, que le décalage vertical est assez complexe et il est basé sur quelques règles de typographie non-algorithmiques, etc.

9.2 Exercices

Q1. Compléter l'exercice d'affichage des expressions.

R1. Cet exercice est pour les ambitieux. On peut apprendre beaucoup de choses en lisant les livre de Donald Knuth consacré au TEX (*The TEXbook*, et *TEX: the program*).

Chapitre 10

Les types

10.1 Qu'est-ce qu'un type et quel est son rôle

Ce domaine est une exemplification paradigmatique de l'approche sémantique à l'analyse, et nous pourrions placer cette section dans le chapitre précédent. En effet, le **type** d'une expression est un des *attributs* les plus importants dans un langage algébrique. L'analyse des types est absolument indispensable pour pouvoir utiliser les opérateurs surchargés, et pour pouvoir reconnaître la légalité de presque toute construction de données.

Nous verrons dans cette section comment l'attribut de type se propage, comment les *déclarations* de types influencent-elles les instructions du langage, quelle est la différence entre les langages typés statiquement et dynamiquement, et comment réagir aux fautes de typage. D'abord il faut essayer de définir la notion de type.

Nous pouvons trivialiser la réponse en disant qu'un type est l'ensemble de toutes les valeurs qui peuvent être affectées à la variable appartenant à ce type, ou mieux : valeurs qui peuvent être traitées par le même algorithme détaillé, sans besoin de conversions, etc. Ceci est plus ou moins correct, mais conventionnel, ambigu, et pas tellement constructif. Il faut savoir vérifier les types par une procédure finie, et dont la complexité est faible (polynomiale, de préférence linéaire). Laissons donc la définition du mot comme quelque chose intuitive, mais formalisable.

Toute valeur dans un langage évolué a – conventionnellement – un type. Les constantes «naturelles» (numériques) peuvent être entières ou réelles, les constantes symboliques comme `#t` en Scheme ou `False` en Haskell appartiennent au type Booléen, et toute structure qui contient une balise particulière aurait dû être déclarée. Si le type est attaché à une valeur de manière explicite, visible (p. ex. par balisage), on parle de *typage dynamique*. On peut affecter une valeur (ou la référence à cette valeur) à une variable quelconque, et la variable elle-même n'a aucun type. C'est le cas du Scheme, Icon, Smalltalk, Python… La programmation est plus souple, mais l'exécution est plus lente, car la discrimination du type des valeurs est effectuée par le noyau exécuteur (la machine virtuelle). Les erreurs du typage sont donc reconnus pendant l'exécution du programme.

Par contre, les langages dont le typage est statique, cette discrimination est une affaire syntaxico-sémantique effectuée par le compilateur *avant* l'exécution. Les variables (y compris les variables décrivant les objets fonctionnels) sont *déclarées* ou leur type est *inféré* automatiquement. Ainsi toute erreur de typage est diagnostiquée avant l'exécution. Ceci ne signifie pas que la suite en C

```
double x;
x=7;
```

soit erronée, seulement que l'incompatibilité des types force le compilateur à ajouter au code quelques instructions de conversion. En général la règle syntaxique

```
var = expr;
```

doit déclencher la procédure sémantique, disons **`compatible`**$(\mathrm{var.type}, \mathrm{expr.type})$. La réponse peut être plus forte : *identique*, ce qui fait générer le code d'affectation sans aucun code supplémentaire, mais aussi affecte le type reconnu à l'instruction elle-même (rappelons qu'en C on peut écrire `x = y = 2*z`). Si les types sont compatibles, mais différents, la conversion est ajoutée.

L'expression `2*z` (ou toute autre expression numérique contenant un opérateur) est analysée de manière analogique. Si les types des opérandes sont identiques, l'environnement doit permettre de décoder l'objet syntaxique `(*)` selon son type. Si les types sont mixtes, une conversion vers le type plus étendu est nécessaire.

En C++, avec toute la panoplie de constructeurs définissables par l'utilisateur la conversion peut être ambiguë, et elle doit être pilotée consciemment. Voir la définition du langage.

10.1.1 Inférence automatique des types, système H-M

Notre expérience avec Haskell a prouvée qu'il est parfaitement possible d'avoir un langage typé statiquement, mais sans déclarations (sauf dans des cas spéciaux). Rien d'exotique, la section précédente montre clairement que le compilateur déduit le type des expressions et peut propager cet attribut en deux sens sans aucun problème. On peut se poser la question pourquoi des langages comme C ou Pascal forcent l'utilisateur à écrire les déclarations?

La réponse est historique. Ici nous présentons de manière superficielle le système de Hindley-Milner, le protocole d'inférence automatique de types utilisé par exemple dans les langages de la famille ML (CAML, SML, Hope, Miranda, etc.) La propriété commune de ces langages est : ils sont fonctionnels, la transparence référentielle est assurée, un symbole (variable) signifie une chose. Si on rejète tout polymorphisme, l'expression `2*x` *force* l'objet `x` à être entier. Si la fonction `f` possède le type $a \to b$, et elle est appliquée à un objet de type a, le résultat possède le type b. Mais si on écrit `y = f x` où on connaît les types de `x` et de `y`, le système H-M déduit le type de la fonction `f` sans aucune déclaration.

Plusieurs fonctions sont polymorphes, par exemple `head (x:_) = x`, applicables à toute liste non-vide. Le système d'inférence déduit le type de cette fonction : $a \to a$. L'argument et le résultat appartiennent au même type.

En général, l'inférence des types des expressions et des fonctions ressemble beaucoup à un raisonnement logique classique, où le flèche $\to$ qui symbolise le type fonctionnel peut être assimilée à la déduction logique. L'inférence qui gère une application fonctionnelle peut être lue : si a et $a \to b$, alors b. Aussi, a et b (pour a et b concrets, «vrais») impliquent $a \to b$. Et ceci permet d'affirmer qu'il n'existe *aucune fonction polymorphe* de type $a \to b$, car ceci n'est pas une tautologie, une vérité logique indépendante de l'assignation de a et b.

10.1.2 Structures composites

Avec la «curryification» des fonctions on n'a pas besoin de les traiter séparemment, les fonctions de type $a \to b \to c \to d$ appartiennent à la même catégorie des entités «logiques». Le système de typage automatique «sait» aussi que si `x,y` appartiennent aux types légaux a et b, l'objet `(x,y)`, le tuple, appartient à un type légal, disons $a \times b$.

Il existe un nombre de constructeurs standard, par exemple List, qui transforme un type légal a en un autre : List a. L'opérateur `cons` aura donc le type `cons :: a -> List a -> List a`. Les constructeurs définis par l'utilisateur entrent dans le même jeu.

Ainsi, la vérification du typage d'un programme devient **une démonstration logique**. Les constantes possèdent des types concrets, considérés «vrais». Aux variables dans le programme on affecte des variables logiques, et on effectue l'unification entre la gauche et la droite d'une assignation. C'est ainsi que l'on découvre que l'expression `(x:x)` est illégale, car implique que $a \equiv \text{List } a$, un *regressus ad infinitum*. Même si la programmation avec des listes nous a habitué à l'existence des structures «infinies» (p. ex. cycliques), les types infinis restent une calamité. Au lieu de parler d'une «démonstration logique» on peut voir le processus comme la solution d'une collection d'équation dans le domaine de types (ou on cherche évidemment les solutions finies...).

10.1.3 Quelques généralisations possibles

Le système de Hindley-Milner est un peu rigide. Sa version «canonique» pose même des obstacles au polymorphisme arithmétique standard, la possibilité de mélanger les entiers et les flottants dans une expression comme $23 + 17.8$. (Ainsi CAML prévoit des opérateurs arithmétiques flottants différents des opérateurs entiers...).

Il y a d'autres problèmes. La paire `(3,True)` est légale. Mais la compilation du programme

```
prog f = (f 3,f True)
res = prog id
```

échoue ! Cependant la forme suivante marche :

```
res = (let f = id in (f 3, f True) )
```

ce qui montre que trop de liberté peut bloquer le *typechecker*. Cependant, le polymorphisme a été inventé pour être commode et utilisable par des non-spécialistes. Ceci implique que les systèmes de typage continuent à évoluer jusqu'aujourd'hui, et constituent un domaine de recherche très actif.

En Haskell nous avons le *polymorphisme restreint* par le système de classes de types (discuté ailleurs). Clean introduit un peu de typage dynamique sur sa couche statique, pour éliminer quelques contraintes.

D'autre part un langage dynamique comme Lisp peut être équipé d'un compilateur qui vérifie le typage statiquement aussi. Dans le cas général ceci mène à rien, mais dans quelques cas il *peut* résoudre un problème de typage avant l'exécution, et générer un code beaucoup plus efficace. Le futur des compilateurs de Scheme appartient à cette catégorie.

Chapitre 11

Deux mots sur l'analyse lexicale

11.1 Qu'est-ce qu'un lexème

Techniques universelles, adaptables à tout problème d'analyse ne constituent pas la seule philosophie possible. Une autre stratégie : moyens simples pour des buts simples, a également ses avantages. Selon cette philosophie l'usage des techniques hautement récursives pour définir des structures régulières, itératives, comme les nombres ou les identificateurs, pour éliminer les espaces du texte, etc., n'est pas idéal. Pour construire les lexèmes simples comme des identificateurs ou des nombres entiers on n'a pas besoin des piles. L'analyse est strictement itérative, le parseur consomme les caractères dans une boucle jusqu'à l'occurrence d'un caractère inacceptable dans ce contexte. Il s'arrête, et la chaîne consommée forme le lexème en question (ou une procédure de recouvrement est lancée en cas de besoin).

Il est évident que pour promouvoir la modularisation du compilateur, une séparation complète de la couche lexicale et de la couche syntaxique pourrait être utile. Nous pourrions construire des scanneurs plus simples que les parseurs génériques, et aussi nous pourrions découpler complètement la partie bas niveau du tableau des symboles – le tableau des chaînes, de l'analyseur syntaxique, qui a besoin seulement des attributs des symboles, et jamais de leur apparence extérieure. Mais il faut se rendre compte d'une vérité souvent négligée.

***Credo* religieux no. 14 :** Il est évident pour tous, que ce qui est évident pour les uns, ne l'est pas pour les autres .

11.1.1 Catégories lexicales

Il faut d'abord régler les problèmes d'interfaçage du scanneur et de sa couche «magique» – la définition de l'alphabet, et les catégories lexicales. Très souvent ces définitions peuvent être statiques, par exemple on définit une *lettre* par une primitive qui vérifie le code ASCI : le caractère doit se trouver entre 65 et 90, ou entre 97 et 122. Un chiffre : entre 48 et 57, etc.

Cependant le monde actuel est multi-lingue, et il serait préférable de pouvoir définir la catégorie lettre de manière plus régulière, et *statique* et non pas par une procédure. Les démarches à suivre sont les suivantes.

- Définir l'alphabet : tous les codes et leur glyphes, c'est à dire *spécifier le codage*. Ceci n'est pas trivial, actuellement il existe plusieurs dizaines de codes qui convergent très lentement. Le standard ASCII pour les premiers 128 codes de l'alphabet s'est stabilizé, mais l'Unicode est loin d'être accepté partout.

- Construire un tableau *indexé par l'alphabet*. Chaque élément doit contenir la description du caractère correspondant – sa catégorie lexicale.

- Établir un nombre raisonnable de catégories lexicales : lettres, chiffres, chiffres étendus (p. ex. les lettres qui peuvent former des chaînes hexadécimales), parenthèses, caractères opérationnels, caractères d'échappement, etc. Déjà cette partie de la conceptualisation de l'analyse lexicale est loin d'être triviale : on voit que `Belle` contient une *lettre* `B`, mais le même caractère peut être (ailleurs) un chiffre hex.

  Il faut par exemple *bien* définir les caractères – espaces blancs.

- Décider ce qui est un identificateur, comment construire les opérateurs, etc. Si on décide de construire un scanneur lexical manuellement, il est ***très*** avantageux de ramasser tous les mots-clés (mots figés par la syntaxe du langage) et de les placer dans la table de symboles du parseur, de les traiter comme les identificateurs, et jamais par des procédures lexicales spécialisées. ceci economisera beaucoup de temps.

. . . et sans lever le pied, commencer à refléchir comment réagir aux fautes lexicales.

11.2 Expressions régulières

Cette section et la suivante décrivent leurs sujets de manière superficielle, les expressions régulières et les automates appartiennent au cours sur les langages et automates. Notre but principal est de donner quelques exemples *pratiques*, et d'élaborer une stratégie de construction des scanneurs.

Une expression régulière $e$sur un alphabet U est :

- la chaîne vide,
- un caractère (élément de l'alphabet) $a \in U$,
- la concaténation des expressions régulières $e_1 e_2$,
- l'alternative $e_1 | e_2$,
- et la fermeture de Kleene : e^*, qui est une abréviation

$$e^* = \phi \mid e \mid e\,e \mid e\,e\,e \mid \ldots$$

ou

$$e^* = \phi \mid e\,e^*$$

où la concaténation et l'alternative sont associatives, et la chaîne vide est l'opérande neutre pour la concaténation. L'alternative est symétrique, et l'ensemble est distributif comme en arithmétique :

```
a (b | c) = a b | a c
(a | b) c = a c | b c
```

On peut introduire d'autres abréviations comme e^+ :

$$\begin{array}{rcll} e^+ & = & e\,e^* & \text{ce qui implique} \\ e^* & = & \phi \mid e^+ & \\ e? & = & \phi \mid e & \end{array}$$

etc. Elles sont utilisées par Lex. Voici la description lexicale régulière des nombres sans signe (disons, en Pascal) :

```
num ::= digit+ (.digit+)? (E(+|-)? digit+)?
```

11.2.1 Automates

Cette expression correspond à l'automate fini présenté sur la Fig. (11.1).

Sur cette figure on a marqué seulement les états finaux légaux. Les transitions sont : **d** dénote un chiffre (**digit**), et **a** – un «autre caractère», qui termine le parsing.

La construction d'un scanneur comme un parseur combinatoire, ou comme un automate piloté par la matrice d'incidence de l'automate en question, tout ceci sont des questions secondaires sur le plan pratique. Il faut surtout s'assurer que

- Le lexème accepté est le plus long possible. Seulement quand il n'y a plus rien à faire, l'automate passe à son état terminal (dans le contexte discuté).
- Les trois issues possibles doivent immédiatemment discriminer le type du nombre reconnu.

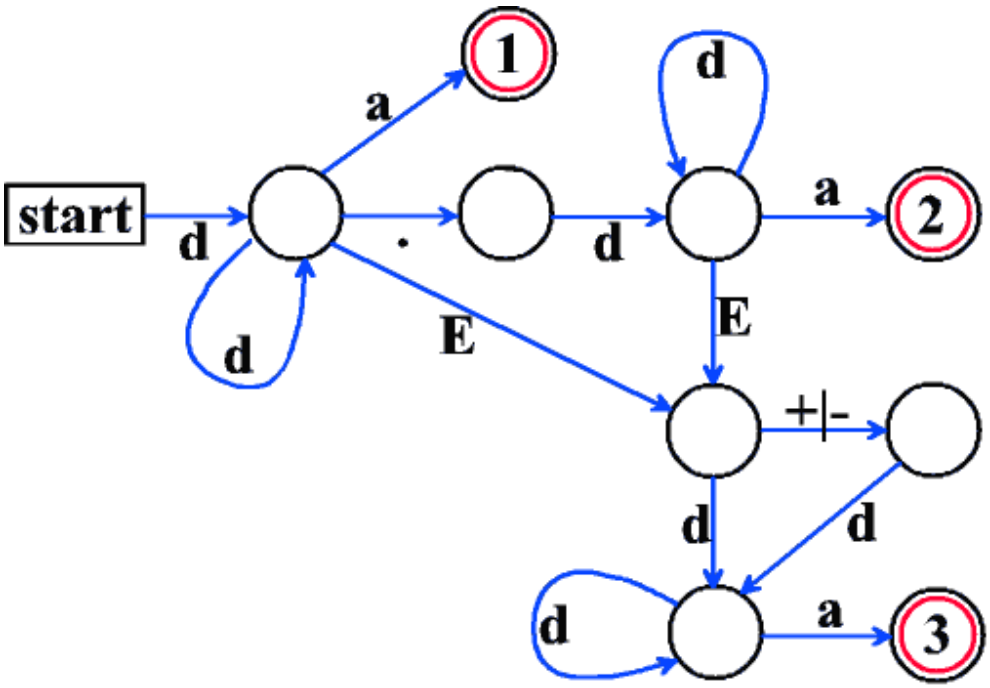


Fig. 11.1: Reconnaissance des nombres sans signe

- Une «bombe», par exemple une séquence **`23.=`** qui viole évidemment la grammaire peut être gérée par des procédures de recouvrement spécialisées. En général, après la découverte d'une faute lexicale, le scanneur doit faire tout pour terminer la construction du lexème.

  Il peut, ou même il doit considérer que le programme est erroné, et de ne plus générer aucun code, mais il peut compléter le lexème en formant **`23.0`**, et passer la main au parseur. Ainsi on pourra découvrir plusieurs fautes lexicales pendant une passe du compilateur.

Chapitre 12

Gestion de mémoire dynamique

12.1 Allocation du tas

Pour l'efficacité d'exécution il serait idéale de pouvoir adresser les structures de données de manière la plus directe, sans passer par les piles, tableaux d'indices, etc. Ceci en général est difficile (et moins portable). Mais même si nous pouvons opérer dans le programme avec les variables qui contiennent des pointeurs directs, un autre problème se pose : le dynamisme. Nous pouvons avoir besoin de quelques structures composites intermédiaires qui vivent pendant l'exécution d'un segment du programme, et deviennent inutiles. Il est donc souhaitable de pouvoir demander au système d'exploitation l'allocation d'un segment de mémoire et pouvoir le retourner au *pool* commun, utilisable par d'autres applications ou threads.

Tous les langages de programmation modernes permettent la *création dynamique* de structures de données. En C++ nous avons la commande **new**, et nous pouvons aussi détruire les structures. Mais la gestion manuelle est toujours un peu délicate. Qu'est-ce passe-t-il si nous détruisons la structure adressée par un pointeur, mais ce pointeur avait été copié dans une autre structure, utilisée par un autre fragment du programme? Pour éviter cette situation paradoxale, un des paradigmes de programmation moderne est : on ne retourne jamais au système des structures inutiles. On les «oublie», et le système (la partie système du *runtime*, le support logiciel qui n'est pas directement contrôlé par le programme) s'en charge pour récupérer la mémoire. Cette facilité est une des raisons de la carrière du langage Java. Les protagonistes de l'efficacité brute et du langage C ne peuvent pas nier que grâce à la gestion automatique de mémoire le nombre d'utilisateurs capables d'écrire des programmes très complexes a visiblement augmenté.

La technique sur un plan très général est suivante. Chaque application peut demander au système qui est le gestionnaire ultime de la mémoire, l'allocation d'une tranche de N octets. Le système – s'il est capable de rendre ce service, retourne au programme l'adresse de la zone allouée (sinon il déclenche une exception, ou, ce qui est plus commode – il retourne une valeur illégale, p. ex., zéro). Le système maintient la liste de zones/pages allouées et libres, et si le programme demande une zone ou retourne une zone, ces tableaux sont mis à jour. Les détails dépendent du langage, et d'autres détails.

Par exemple, quand le programme retourne au système une zone inutile, il passe aux procédures de récupération son adresse. Mais le système doit savoir *combien* d'octets récupérer ! On peut donc choisir deux stratégies :

- La première est universelle. Chaque *record* alloué et attribué au programme possède un champ caché – la longueur de la zone. Ce champ est utilisé pour la déallocation, le programme n'en a pas besoin.

- La deuxième est plus économique, mais beacoup plus difficile à implanter. Si le langage de programmation est typé, le *compilateur* connaît la longueur de toutes les structures. on peut donc (en principe) prévoir plusieurs procédures de récupération de mémoire, chacune spécifiquement adaptée à un type concret de données.

- On peut aussi envisager la variante orientée- objet de la stratégie précédente. Chaque classe de structures possède sa méthode particulière, un destructeur adapté. La différence de cette technique par rapport aux destructeurs en C++ est que dans ce langage les destructeurs sont exécutés quand la structure est retournée au système par l'utilisateur, tandis qu'ici il n'y a pas de «**delete**» ou autre instruction équivalente.

Il faut, indépendamment des détails, prévoir encore une procédure facile à implanter : le compactage de la mémoire libre. Supposons que la mémoire est partiellement allouée, comme sur la Fig. (12.1). En retournant

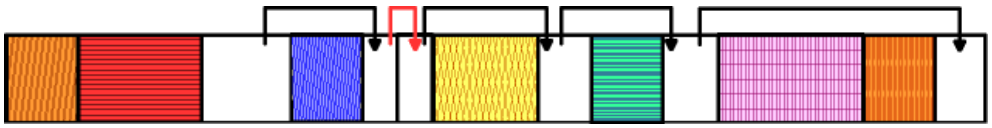


Fig. 12.1: Mémoire partiellement allouée

quelques objets au *pool* on produit des «trous» – zones libres. Le système normalement construit une liste des segments accessibles, qui peuvent être utilisées pour des allocations ultérieures. On voit que le lien rouge sur le dessin est redondant : le système doit compacter deux zones libres voisines en une plus grande.

Un problème *beaucoup* plus compliqué serait : comment compacter les zones libres non-contiguës par la relocation des structures allouées. Nous en allons parler plus tard.

12.2 Compteurs de références

Une technique classique, souvent enseignée avec C++, mais plutôt rarement implémentée correctement dans un contexte pédagogique, c'est la méthode qui prévoit que chaque record alloué possède un champ entier supplémentaire : un compteur des références. Ce compteur est initialisé à 1 au moment de la création de l'objet.

Si la structure S est affectée à une variable-pointeur p : `p=S;`, le système fait trois choses :

1. Vérifie la valeur précédente de p. Si la variable pointait sur une autre structure de ce type (ou compatible), son compteur de références est décrémenté. S'il devient égal à zéro, la structure est retournée au *pool* système.
2. L'adresse de S est passée au récepteur.
3. Le compteur de S est incrémenté.

Apparemment la technique est simple, efficace et sure. Mais il en faut voir quelques défauts :

- Le premier, bien reconnu depuis longtemps c'est l'impossibilité de gérer les références cycliques. Si une structure possède un champ-pointeur qui adresse elle même, l'affectation de ce pointeur incrémente le compteur de la structure, et même si toutes les variables extérieures ont «oublié» la structure en question, sa mémoire ne sera jamais récupérée.
- Un problème de discipline, qui, d'ailleurs, est commun, et frappe d'autres stratégies d'administration automatique de mémoire : le clonage. En utilisant le compteur des références il faut éviter à tout prix passer des structures composites *par valeur* au procédures, ce qui risque normalement de recopier tous les champs de l'objet. Le constructeur de recopie (en C++) ou une autre activité qui place la copie de la structure sur la pile, peuvent produire des effets très indésirables : modification incongrue de plusieurs exemplaires du compteur attaché à une structure. De préférence il faut passer tous les paramètres par référence (adresse).

12.3 Ramasse-miettes «marquage et balayage»

La technique de compteurs permet de retourner au système un objet inutile au moment de sa libération. Mais une autre stratégie devient à présent plus populaire : le *garbage collection* (GC), ou ramassage de miettes. La mémoire est allouée jusqu'à l'épuisement des ressources, et les structures oubliées résident dans la mémoire du programme. Quand le programme demande une nouvelle allocation et le système n'a plus de ressources, le programme entre dans la phase de ramassage, où la mémoire est analysée, les structures «vivantes» localisées et identifiées, et les structures mortes sont retournées au *pool* de la mémoire. Ce domaine a éloboré une certaine

terminologie. La «vraie» application, le module qui effectue les calculs et qui créé les structures de données s'appelle le *mutateur*, et le module GC – le *collecteur*.

Il existe deux techniques traditionnelles de GC :

- Marquage-balayage (*mark and sweep*, et
- Recopie complète.

Commençons par la première méthode. la Fig. (12.2) montre ce qui peut arriver à la mémoire, la présence d'un amas de structures liées avec les pointeurs, où chaque structure possède également d'autres informations (chaînes, nombres, etc.) Chaque structure doit prévoir la présence d'un bit supplémentaire, qui est inutilisé

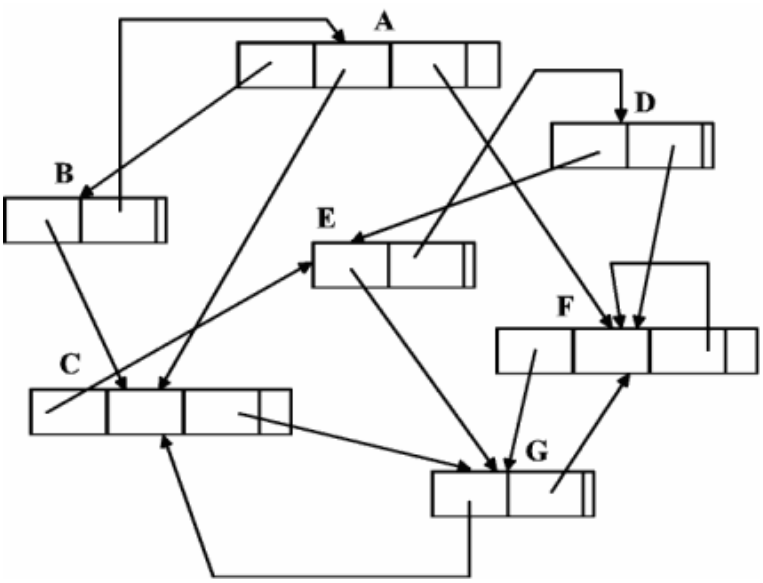


Fig. 12.2: Un «plex» dans la mémoire

pendant le fonctionnement normal du programme, et égal à zéro (par exemple). Quand la mémoire est épuisé et le *runtime* déclenche le GC, celui-ci doit avoir accès direct à un certain nombre d'objets, comme la pile système, les variables statiques, et autres données adressées directement dans le programme. Appelons cette collection le *noyau* des structures accessibles. **Toute** autre structure doit être accessible à partir du noyau en suivant les pointeurs.

Le GC *mark-and-sweep* fonctionne en deux phases. La première consiste à marquer (positionner le bit GC à 1) *toute structure accessible dans la mémoire*. Ceci peut être facilement réalisé par la récursion : Le GC accède à une structure, disons, A. Si elle a déjà été marquée, on ne fait rien et on retourne. Sinon, la structure est marquée, et le GC récursivement traite toutes les structures accessibles depuis A : les records B, C et F sur la Fig. (12.2).

En analysant B le GC n'a rien à faire avec A, il marque seulement C. Quand le contrôle retourne de B à A, la structure C est déjà marquée, rien à faire. Ainsi, en exploitant le parcours de ce graphe en profondeur, on marque toute structure vivante, accessible depuis le noyau.

La seconde phase est le balayage (*sweeping*). C'est une opération de bas niveau, qui ne respecte pas le typage des structures, et traite la mémoire comme un tableau contigu. Ce tableau est parcouru *linéairement*, le GC visite toutes les structures dans l'ordre des adresses. (Il doit donc savoir quelle est la longueur de chaque structure ; on ne peut pas se permettre d'avoir dans la mémoire des zones étrangères, gérées par un autre mécanisme ; ceci rend très difficile la co-existence du *garbage collector*, et de procédures externes, p. ex. des sous-programmes en C attachés à un programme en Haskell).

Pendant cette phase toute structure marquée est restaurée à son état d'origine (le bit GC est nettoyé). Les structures qui n'ont pas été marquées sont mortes, et peuvent être liées ensemble, en formant la zone libre.

Cette technique possède un défaut majeur – l'usage de la pile à cause de la récursivité peut devenir dangereux.

Le tas normalement est beaucoup plus volumineux que la pile système, et l'implantation naïve de cette technique de GC peut être défaillante.

12.3.1 Optimisation de Schorr-Waite

Un pseudo-code qui réalise le GC (la partie marquage) présenté ci-dessus serait

```
mark object =
 if object.gc!=Marked then
   object.gc=Marked
   for_each x=address_field(object) do
         mark x
```

et on peut envisager quelques optimisations, comme le remplacement de la récursivité par une pile privée, ou un parcours en largeur du graphe. Cette dernière possibilité sera discutée dans le contexte du GC copieur, ici nous présentons une très intéressante idée de Schorr et Waite, qui modifie temporairement le graphe de données dans la mémoire, et utilise les données elles-mêmes pour implanter une pile temporaire.

La description intuitive de l'algorithme est la suivante. Le GC en marquant un objet X garde toujours l'adresse de l'ancêtre A, l'objet qui contenait le champ x adressant l'objet en train d'être marqué. À présent nous devons descendre de X vers les fils de ce dernier, disons, à F adressé par le champ f, comme montre la Fig. (12.3).

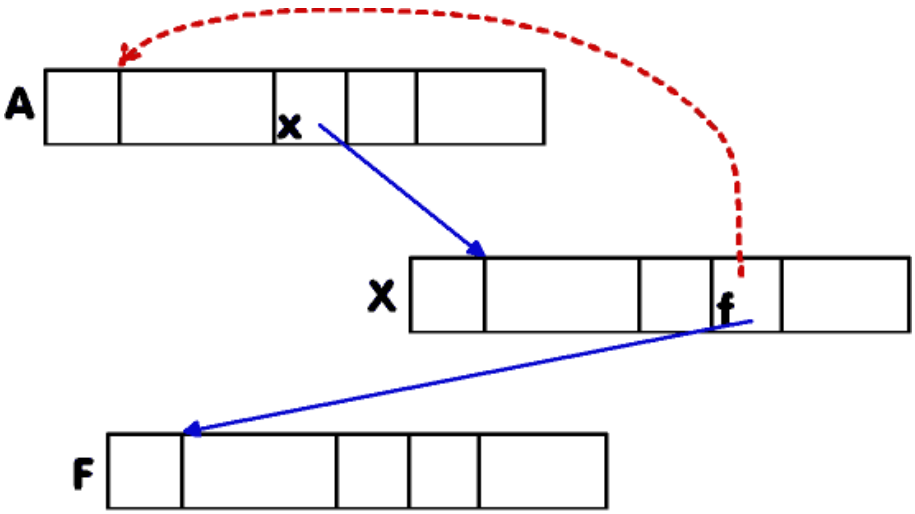


Fig. 12.3:

On descend par une simple ré-affectation de l'argument dans une boucle, sans aucun appel récursif, mais en descendant on mémorise A dans f. Après le marquage complet de F on réinsère son adresse dans f, en récupérant l'adresse de A. On peut alors marquer un autre champ, ou, si tous ont déjà été marqués, on remonte vers A. Voici un pseudo-code un peu plus discipliné. Il est conseillé de le lire et comprendre.

On organise très légerement différemment l'administration du marquage, la procédure suppose *toujours* que son argument n'a pas été marqué, la décision de ne rien faire dans le cas contraire est prise avant la «descente».

```
mark_SW object = msw object NIL where
 msw obj anc =
   obj.gc=Marked
   for_each f=address_field(obj) do
     if f.gc!=Marked then
        f=anc
        msw f obj
       ...
```

12.3.2 Problèmes avec le compactage de la mémoire

Un problème persistant de la stratégie de marquage-balayage est la *fragmentation de mémoire*. La *fragmentation extérieure* est le résultat de l'allocation et libération répétées, ce qui peut provoquer la création d'un grand nombre de petits trous dans la mémoire. La liste des segments libres risque de ne pas être contiguë, mais composée de petits segments séparées dans la mémoire par les zones allouées. Si le programme demande l'allocation d'un segment plus grand que le segment libre actuel, le système doit chercher plus loin. D'abord, ceci ralentit l'allocation. Mais on risque finalement de ne rien trouver, même si la quantité totale de mémoire libre est grande !

Il existe aussi un phénomène de *fragmentation interne*. Si un segment libre contient 20 octets, mais nous demandons, disons, 18 ou 19, il est hors de question d'allouer 18 octets de ce segment, et laisser 2 ou 1 octet inutilisable, car on ne peut pas l'attacher à la mémoire libre, on n'a pas suffisamment de place pour y mettre le pointeur sur le segment suivant. Donc, on alloue la totalité, 20 octets à la structure de données, même si quelques octets à l'intérieur ne seront jamais utilisées. On doit éviter que ce phénomène gaspille trop de mémoire.

Il existe toute une théorie d'allocation optimisée. On peut prendre le premier segment libre suffisamment grand. Ou, chercher le plus petit, mais encore convenable. Ou, chercher *le plus grand* restant, pour éviter la fragmentation interne, etc. Tout ceci sont des moyens semi-heuristiques de prévenir ou de retarder les problèmes causés par la fragmentation, mais le vrai rémède est le compactage de la mémoire. Il faut déplacer les objets vivants, les «glisser» dans la mémoire vers le bas ou vers le haut (pendant la phase de balayage), et – naturellement – ne pas oublier la mise à jour de tous les objets qui référencent la structure déplacée. L'algorithme de compactage qui marche avec le GC marqueur est assez compliqué, et il ne sera pas discuté ici. La technique de recopie nous donne le compactage gratuitement.

12.4 Ramasse-miettes copieur

Cette catégorie de ramasse-miettes est très différente, et est basée plutôt par un parcours du graphe de données en largeur, avec une file qui est réalisée par les structures de données elles-mêmes, pendant le processus de ré-arrangement. Elle a été inventée par Cheney.

On commence par la séparation de l'espace de travail en deux moitiés distinctes, dont une seule est utilisée (nous l'appelerons traditionnellement *from-space*), et l'autre reste en jachère (qui deviendra *to-space*). Ceci peut paraître un gaspillage inacceptable, mais, enfin, … la mémoire est devenue un article de consommation courante… On peut aussi prévoir l'allocation de la jachère sur le disque, ce qui économise la mémoire centrale, mais ralentit considérablement le processus GC.

La mémoire est allouée toujours depuis une tranche contiguë, il suffit de maintenir un pointeur sur la zone libre (ainsi que sa longueur restante, bien sûr). Quand la mémoire est épuisée, le programme est suspendu, et le GC transporte tous les objets vivants depuis *from-space* dans la jachère (*to-space*), qui devient ainsi la zone de travail. L'espace utilisé précédemment devient la jachère.

On commence par accéder aux objets accessibles directement (le noyau), par exemple l'objet A sur la Fig. (12.2). Il est copié intégralement dans la mémoire de la nouvelle zone de travail, suivi par les copies d'autres objets du noyau. Seulement la «surface» des objets est copiée, les pointeurs internes ne sont pas modifiés ! On commence à traiter les champs internes par la suite. Le GC connaît la structure de l'objet A, donc il reconnaît le champ qui adresse B. B est donc copié, et placé dans le nouvel espace, ici : suivant A. Après avoir terminé cette copie, le champ correspondant de A est mis à jour. On fait la même chose avec C et F, et la nouvelle zone de travail, *to-space* possède la structure comme sur la Fig. (12.4). Ayant épuisé les champs de A, on peut passer aux champs de B. Mais le premier pointe sur une structure qui a déjà été déplacé, donc il ne suffit pas de copier une structure, mais il faut mémoriser ce fait. Ainsi, au moment de la recopie, la structure interne de l'original est détruite, et remplacée par l'adresse de la copie. L'original doit également être marqué comme déplacé, ce qui suggère la présence d'au moins un bit de marquage (ou l'usage d'un tableau spécial extérieur, où on sauvegarde les adresses de toutes les structures déplacées), mais en fait ceci est inutile : si un objet dans *from-space* contient un pointeur appartenant à *to-space*, il a été déplacé et ce pointeur est sa nouvelle adresse, le *forwarding pointer*. Dans aucune autre circonstance on ne peut trouver une telle configuration.

En suivant ce *forwarding pointer* stocké dans A, on peut construire la vraie valeur du premier champ de B (la ligne pointillée).

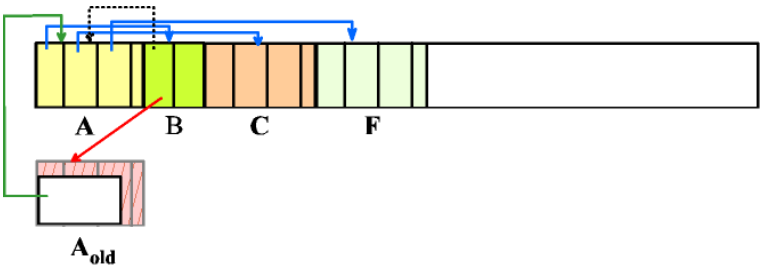


Fig. 12.4: Processus de recopie

On traite l'autre champ de B où le même phénomène se produit (record C qui a déjà été traité depuis A), et on passe à C, ce qui déclenche la recopie de E devant F.

Après avoir copié toutes les structures, l'espace de travail et la jachère changent de rôles, le *to-space* devient le *from-space* pour la callection suivante. Il est évident que cette technique produit toujours l'occupation contiguë de la mémoire. De plus, la complexité temporelle de cet algorithme est proportionnel au nombre de structures *vivantes*, tandis que la stratégie de balayage force le parcours par la totalité de la mémoire, et la complexité devient donc proportionnelle au nombre *total* d'objets.

12.4.1 Ramasse-miettes générationnel

Dans un programme réel suffisamment compliqué le «temps de vie» de structures dynamiques peut être très dispersé. Quelques structures temporaires sont allouées pour disparaître presque immédiatemment, les autres, hautement partagées deviennent permanentes, ou presque. Si on pouvait éliminer le traitement de ces données par le GC (ou au moins optimiser cette partie du ramassage), l'économie de temps pourrait être assez significative.

L'idée du GC *générationnel* est donc la suivante. On divise le tas (la partie exploitée dans le cas de l'algorithme de recopie) en quelques (2 - 5) zones de longueur variable, p. ex. G0 assez petite, G1 4 fois plus grande, G2, quatre fois plus grande que G1, etc. Au début toutes les nouvelles allocations ont lieu dans G0. Quand la mémoire doit être régénérée, on note dans les structures vivantes le fait qu'elles ont survécu un ramassage (donc, chaque record est équipé d'un compteur). Après quelques ramassages on transporte les structures toujours vivantes dans G1.

Le nettoyage de G0 est fréquent car la zone est petite. Il faut se rendre compte que les structures transportées dans G1 peuvent adresser les objets dans G0 (et *vice-versa*). Pour nettoyer G0 il faut donc parcourir G1, pour trouver les objets accessibles. Une fouille exhaustive de G1 serait mortelle, donc quand on déplace un objet de G0 dans G1, et cet objet référence les structures dans G0, on place son adresse dans une liste/tableau directement accessible. Normalement ceci **doit être rare**. Posséder la référence vers un objet plus stable, donc de G0 vers G1 est normal, mais la situation inverse ne doit pas être fréquente (un vieillard ne retient pas dans la mémoire les événements récents...).

De temps en temps il faut nettoyer G1, et – éventuellement – on fait la même chose, en transportant les objets durs à détruire dans G2.

12.4.2 GC pour les données «binaires»

On peut en principe poser la question suivante : que faire si dans le record alloué il n'y a pas de place pour le bit GC? Par exemple, si on alloue quelques octets qui seront remplis par un nombre flottant. Une solution serait de restreindre l'intervalle des nombres, et de ne réserver qu 47 bits pour la mantisse au lieu de 48. Dans le cas des entiers, utiliser seulement 31 bits et non pas 32. Le problème *principal* de cette stratégie n'est pas la perte de précision/capacité, mais la nécessité de «décortiquer» ce bit inutilisable lors de ***toute*** opération arithmétique !

On peut allouer un octet entier supplémentaire, ceci ajouterait $1/8$ d'allocation de mémoire aux nombres de type `Double`, mais une autre technique est possible aussi, surtout si toutes ces structures appartiennent au même type (même longueur d'allocation), et peuvent être allouées depuis une zone libre dédiée : on réserve un tableau externe, traité comme une suite de bits. Chaque bit correspond à un record, et ainsi on peut utiliser la

technique de marquage classique, seulement le bit GC n'est pas physiquement attaché à la donnée en question, mais séparé.

Si on décide de stocker tous les nombres flottants (de précision double : 8 octets par nombre) dans un tableau dédié, le tableau accessoire aura la taille 64 fois plus petite, ce qui est abordable.

12.4.3 GC en temps réel : ramassage incrémental

Parfois le fait que le ramassage arrête le programme, fige son action pendant un certain temps, est gênant. Peut-on nettoyer la mémoire par un *thread* parallèle (ou un processus lancé périodiquement par le programme principal) pendant un travail normal? Le problème est délicat, car pendant le GC la mémoire se trouve dans un état instable, parfois elle est «endommagée» (algorithme de Schorr-Waite qui renverse les pointeurs crée dans la mémoire des structures qui n'ont rien à voir avec le programme…; durant la phase de recopie qui insère les *forwarding pointers* aucune partie de la mémoire n'est utilisable…).

Il est difficile d'imaginer que l'on puisse dans un tel contexte faire une partie du ramassage (marquer partiellement les structures, par exemple). Cependant, si le modèle de marquage (bit : oui/non) est étendu à ***trois couleurs*** : blanc, gris et noir, où un noeud noir a été marqué (copié) entièrement, avec tous ces descendants, et un noeud gris a été traité, mais ses descendants pas encore, alors le ramassage incrémental devient effectif. Ceci est très important pour l'usage des langages évolués, qui exploitent l'allocation dynamique de mémoire dans le contexte de programmation en temps réel.

Les détails de l'algorithme sont un peu compliqués. En fait, il existe déjà plusieurs algorithmes incrémentaux, classés en deux catégories, les («vrais») algorithmes incrémentaux, où le collecteur opère quand l'application principale le demande, et le ramassage concurrentiel qui se déroule en «parallèle», en temps partagé avec le consommateur de mémoire (le mutateur).

On commence par une description abstraite du processus GC. Pour les deux stratégies décrites ici, le marquage et la recopie, on peut considérer que les nœuds possèdent une de deux couleurs, ils sont Blancs («normaux») et Noirs (visités : marqués ou copiés). La régénération de mémoire consiste à trouver tous les nœuds qui peuvent être «noircis» ; le marquage utilise une pile, et l'algorithme de Cheney – une file.

Ici les nœuds sont donc divisés en trois classes : les Blancs, les Noirs et les Gris.

- Les objets Blancs n'ont pas été encore visités, ni par l'algorithme de marquage, ni par la boucle de recopie. Le GC démarre avec tous les nœuds Blancs.
- Les objets Gris ont été touchés (marqués ou copiés), mais leurs descendants non, pas encore. Dans la stratégie de GC en profondeur (*mark-and-sweep*), ils sont accessibles par la pile de stockage des descentes récursives. Pour la stratégie de recopie, ce sont des records copiés, mais dont les fils n'ont pas encore subi le transfert.
- Les objets Noirs sont marqués avec tous les descendants immédiats. Ils ont quitté la pile, ou ils ont été recopiés avec leur famille descendante (les ancêtres et les descendants lointains peuvent rester encore Gris).

Un pseudo-code qui décrit le ramassage est très succint :

```
while(objets Gris existent)
  p = le premier objet Gris accessible
  for_each(champs f_i de l'objet p)
    if(p.f_i est Blanc) p.f_i <- Gris
  p <- Noir
```

La présence de la boucle intuitivement suggère que la stratégie décrite ici s'adapte mieux aux GC qui exploitent la recopie, et, en effet, ceci est plus populaire. Nous constatont maintenant, que

- Quand il n'y a plus de nœuds Gris, tous les objets Blancs sont morts.
- Tous les objets Gris doivent être physiquement accessibles (pile, ou file…).
- Finalement, aucun objet Noir n'a le droit d'adresser un objet Blanc ! Ce «racisme» est contagieux, comme nous le verrons dans quelques instants. Les objets changent de couleur en fonction de nœuds qui le voient.

Quand le collecteur travaille, me mutateur doit pouvoir créer de nouveaux objets. (De quelle couleur?[1])

Quand la procédure utilisateur stocke un pointeur sur une structure «normale», ancienne, Blanche A dans un champ d'un objet Noir B, il faut colorier Gris A et B. Le compilateur doit générer un code supplémentaire pour chaque affectation d'un pointeur, afin d'assurer un bon déroulement de cette opération.

Quaand le mutateur *accède* à un pointeur sur un objet Blanc, cet objet devient immédiatemment Gris. Le mutateur ***ne peut*** jamais opérer sur la référence à un objet Blanc ! Donc, les accès sont compilés aussi avec quelques instructions supplémentaires, ce qui est coûteux.

Quelques autres procédures entrent en vigueur si le système utilise la mémoire paginée, mais nous allons ommettre la discussion de ces problèmes.

C'est presque tout. À présent les deux «adversaires», le mutateur et le collecteur peuvent travailler ensemble. Essayons quand même d'y ajouter quelques précisions.

12.4.4 Algorithme de Baker

Cet algorithme augmente la puissance (et ralentit) de l'algorithme de Cheney, et il est compatible avec le GC générationnel.

Quand la mémoire est épuisée, les *from-space* et *to-space* basculent, et les objets dans le noyau sont déplacés. Ensuite le mutateur peut reprendre le travail. Mais chaque fois quand le mutateur demande l'allocation des nouvelles cellules, un processus quasi-parallèle force le scanning de quelques objets par le collecteur. Ils sont déplacés, et le nouvel objet alloué prend sa place à la fin de la zone d'allocation dans le *to-space*. Bien sûr, quand le nouvel objet est initialisé, les objets-cibles de ses champs sont (éventuellement) recopiés. ***Ainsi le mutateur n'adresse que le* to-space *; tous les pointeurs stockés appartiennent à la nouvelle zone du travail***.

Quand le mutateur accède à un objet, les instructions supplémentaires générées par le compilateur assurent le système qu'on reste dans le *to-space*. La découverte d'un pointeur «ancien» provoque immédiatemment la recopie de sa cible.

Le système GC est propre et lisible, mais son code est long. Nous n'allons pas le traiter plus, et on s'arrête ici. Tout le domaine de GC n'a pas dit son dernier mot.

[1] Les étudiants répondent trop souvent «Blanche», ce qui est complètement faux !

Chapitre 13

Macros et pre-traitement

13.1 Transformations source – source

Ce chapitre est plus important que l'on n'y pense. Le pre-traitement (*preprocessing*) est une étape qui précède d'habitude la compilation (translation en code intermédiaire/final) proprement dite. Pourquoi d'habitude? Souvent dans les livres on trouve une discussion très simplifiée, qui réduit les macros à la substitution textuelle des symboles par symboles ou séquences de symboles. Mais la vérité est infiniment plus riche.

La substitution textuelle des symboles est très utile, et l'usage des constantes et formes fonctionnelles symboliques (par **`#define`**) est un des traits caractéristiques du langage C Il est évident qu'une pre-définition de la constante PI=3.1415926536 économise le temps du programmeur et évite quelques fautes, et qu'une compilation conditionnelle selon la valeur de la variable **`BIG_ENDIAN`** peut rendre le programme plus portable.

Cependant, la transformation générale source → source est une technique de compilation universelle, qui doit être connue. Elle peut réaliser plusieurs objectifs.

- Une possibilité de faire un compilateur vraiment portable, indépendant de la plate-forme cible. On compile tout en C, ou un assembleur portable, dont un certain nombre existe sur le marché. Le résultat peut être moins efficace que la compilation native, mais ceci servira à rendre le langage plus populaire. Tels étaient les débuts de Haskell.

- Transformations des structures syntaxiques spécifiques à un langage, en formes universelles. Par exemple : transformation des boucles en récursivité terminale, ou vice-versa (selon les structures des langages source/cible). Nous allons commenter ceci plus soigneusement.

- Une possibilité de *bootstrapper* un compilateur, écrit dans le même langage qui sera compilé. La première phase consiste à écrire un compilateur simple, mais de le «compiler» (manuellement, ou par un macro-processeur qui est *beaucoup* moins complexe qu'un vrai compilateur) en C, ou autre langage universel, et ensuite utiliser le premier compilateur pour traiter la seconde version. Dans la pratique il faut prévoir plusieurs passes source-source.

La réalisation d'un macro-processeur intégré à un compilateur peut être plus ou moins facile selon la puissance du macro-langage souhaité, et le caractère du compilateur : est-ce un compilateur/interprète qui donne à l'utilisateur l'accès aux modules de compilation de l'intérieur du programme, comme en Scheme ou Smalltalk, ou c'est un compilateur «boîte-noire» comme le compilateur C?

La variante la plus simple, est la suivante :

- Le paquetage de compilation dispose d'un *dictionnaire de macros*. Ceci peut être une table de symboles indépendante de la table principale, mais il est possible, pour des raisons d'homogénéite, d'utiliser une seule. Ainsi tout symbole, macro, ou autre chose, possède une seule référence, et une macro peut être locale dans un bloc, comme tout autre symbole.

- Tout symbole – macro possède un attribut macro associé à une valeur qui est une liste de lexèmes *sans interprétation, sans structuration*. Autre possibilité : le système gère une liste d'associations séparée du dictionnaire, et cherche les valeurs dans cette liste. Les systèmes Scheme populaires utilisent soit un soit l'autre variante, selon l'option choisie par les concepteurs.

- Le scanneur remplace le symbole-macro par sa définition. Ceci signifie que l'analyseur doit pouvoir changer localement le flux d'entrée et de passer à la macro-définition, en sauvegardant le contexte précédent.

- Il est naturel de ne pas considérer le flux secondaire comme un flux de caractères, mais de lexèmes déjà construits avant, lors de la lecture de la macrodéfinition.

- En fait, nous avons menti encore une fois. Le préprocesseur du C permet qu'une macro soit réellement une liste des lexèmes, mais en Scheme le macro-développement est ***une transformation des expressions en expressions***. Si la ***forme*** **`(f x y)`** est reconnue comme une macro-expressions, c'est-à-dire si **`f`** possède l'attribut de macro, la forme entière est transformée en une autre forme, mais qui constitue une expression Scheme légale.

Dans la pratique la situation peut être encore plus complexe.

- Si les macros peuvent être récursives (par exemple, nous voulons précompiler quelques fonctions numériques, ou de traitement des listes), la sauvegarde du flux d'entrée doit utiliser une pile, avec toute la complication que cela implique.

- Macros récursives sont inutilisables, si le préprocesseur n'est pas capable de prendre des décisions de développement conditionnel, pour arrêter le dépliage des macros.

- Les macros doivent alors être parametrées, et ceci signifie que les macro-définitions constituent un langage dans langage. Le préprocesseur *constitue un interprète, une machine virtuelle integrée au paquetage de compilation*.

  Si, comme en C le macro-développement est une substitution textuelle (lexicale), le préprocesseur doit être équipé avec un petit parseur, et un «générateur de code» dont le résultat est le flux constituant la macrodéfinition. Le parseur doit au moins pouvoir reconnaître des formes fonctionnelles de type

```
#define abs(x)  ((x<0)?(-(x)):(x))
```

pour pouvoir remplacer **`x`** par l'expression appropriée lors du développement de **`abs(a-2/x)`**, et de ne pas confondre les «**`x`**».

Les macros comme le **`abs`** ci-dessus ont l'avantage de profiter de la surcharge des opérateurs de relation et du signe ; on n'est pas obligé de définir une macro pour les entiers, l'autre pour les complexes, etc. D'autre part, si l'argument d'une macro est une expression composite, il serait mieux de disposer d'un bon optimisateur de code, et en particulier d'un éliminateur des expressions communes. (Sinon, imaginez en C l'expression **`abs(XX)`**, où **`XX`** est une expression vraiment très large (plusieurs pages)...

Les macros restent une technique ambiguë. Les méthodologues des langages évolués modernes comme C++, où les macros ont été héritées du C, préconisent plutôt l'usage des fonctions *inline* que des macros, et leur usage habituel se restreint aux constantes symboliques et des fonctions très simples. En fait, dans Clean les macros sont des procédures *in-line*, les macro-définitions doivent être des *expressions* légales et bien typées. Les templates en C++ sont également une sorte de macros structurées et typées. D'ailleurs, dans les deux cas (Clean et C++), l'intégration du système de types rend impossible leur développement par un module séparé du compilateur.

La substitution textuelle est une source de plusieurs problèmes concernant la portée et le statut des identificateurs-noms des macros. Est-ce qu'une macrodéfinition doit être locale ou globale? Comment éviter le conflit des noms, sachant que le macro-développement est une transformation de la *source*? Peut-on dynamiquement assembler et créer des identificateurs nouveaux à partir des fragments : imaginez une macro-boucle **`for i=... x$i ...`**, qui crée les variables **`x1`**, **`x2`**, etc. Est-ce raisonable?

D'autre part, il existe des langages interprétés qui sont par excellence des macroprocesseurs, où il n'y a aucune, ou presqu'aucune différence entre les procédures et les macros. Tel est le cas du TEX et de MetaPost. Le langage JavaScript est aussi un macro-langage, les navigateurs comme Netscape ne génèrent pas de code intermédiaire, mais développent *in-line* les définitions des fonctions présentes dans le document.

13.2 Macros et langages-amibes

Des macros universelles peuvent en principe changer complètement l'apparence extérieure d'un langage. L'exemple le plus saugrenu vu par l'auteur de ces notes était la réalisation d'un langage de programmation BALM pour les machines CDC série Cyber. BALM était un langage relativement classique, fonctionnel, mais son compilateur-interprète était une merveille d'éclectisme : il avait l'apparence d'un programme en Lisp avec des listes, la récursivité, etc., et en vérité il a été écrit en assembleur. Chaque expression style Lisp, parenthésée, était une macro-instruction qui dynamiquement optimisait l'allocation des registres, préparait des étiquettes pour **goto**, gérait la pile des blocs lexicaux, etc. La puissance du macro-assembleur CDC était telle, que l'assemblage était dix fois plus lent que la compilation d'un programme en Pascal…

Cependant le langage qui peut être modifié à volonté manque tout simplement de stabilité. Il est très difficile à apprendre, et les programmes ne sont pas lisibles, ce qui va à l'encontre de l'idée même des macros.

Voici quelques exemples-type de l'enrichissement d'un langage par des macros. Les macro-formes en Scheme grâce auxquelles on définit de nouvelles formes spéciales (syntaxiques), par exemple des nouvelles structures de contrôle. Grâce à la couche macro en Scheme on peut simplifier la notation, ajouter des mécanismes de destructuration automatique des paramètres, ou même optimise le programme au niveau source. On ne sait pas comment compiler

```
(cond (a b c)
      (d e f)
      (else h i))
```

alors on commence par la réduction de cette forme en une cascade de **if**. Concrètement, DrScheme produit

```
(#%if a (#%begin b c)
        (#%if d (#%begin e f) (#%begin h i)))
```

Ainsi on peut réduire **let** à une forme contenant **lambda**, **while** à **let** nommé récursif-terminal, et celui-là, à des primitives de genre **letrec**, comme ici :

```
(let f ((a b))
       (g a) (if (p a) a (f (r a)))
==>>
((#%letrec-values
   (((f) (#%lambda (a) (g a) (#%if (p a) a (f (r a))))))
   f)
 b)
```

Ainsi le «vrai» compilateur, c'est-à-dire le générateur du code n'aura à traiter qu'un nombre limité de primitives.

Le compilateur de Scheme ayant découvert qu'un symbole `f` possède l'attribut macro, lance la procédure **expand-defmacro** de la forme entière `(f ...)`. On peut attacher à `f` une procédure de transformation quelconque, écrite en Scheme par l'utilisateur.

13.3 Exercices

Q1. Chercher (et trouver), ou construire la macro-procédure `when`, qui transforme en Scheme la forme

```
(when (a b) c d e f)
```

en

```
(#%if (a b) (#%begin c d e f))
```

R1. La voici :

```
(define-macro when
   (lambda (test . body)
     `(if ,test (begin ,@body))))
```

Notez, que la forme-cible contient **if** dans la définition, les primitives **`#%if`** etc., sont construites automatiquement.

Q2. Comment Scheme développe la forme **define**? On sait par exemple que

```
(define (a b) c d)
```

se transforme en

```
(#%define-values (a)
                 (#%lambda (b) c d))
```

mais comment gérer les **define** locaux à une fonction, ou les définitions des fonctions à nombre variable d'arguments?

R2. Cherchez la réponse vous-même. La deuxième question n'a pas de réponse unique. On peut prévoir des opérateurs, ou des formes lambda n-aires, mais on peut aussi – dans quelques cas – faire des macro-transformations du genre :

```
(* a b c d) ==>> (* a (* b (* c d)))
```

Chapitre 14

Modèles de code plus sophistiqués

14.1 Évaluateur eval-apply

Cette section est entièrement consacrée à la discussion de quelques machines virtuelles réelles, plus complexes que notre machine à pile, ou notre interprète arborescent. C'est un joli jeu de mots : machines virtuelles réelles, mais le sujet est très pratique et incontournable pour tous ceux qui veulent *vraiment* construire des compilateurs.

Il n'est pas difficile de trouver dans la littérature (par exemple sur l'Internet) la définition et l'implantation d'une *vraie* machine virtuelle pour Lisp ou autre langage fonctionnel. On les enseigne comme outils d'implantation, définitions de la sémantique des langages, et comme un terrain pratique por apprendre optimiser les algorithmes. Un de ces modèles, la machine SECD de Landin, qui est toujours un bon modèle de machine fonctionnelle stricte, a été élargie à maintes reprises : il existe une variante paresseuse (CASE), une machine avec des objets persistants, etc. Ces modèles servent également à implanter les langages à objets.

La première machine virtuelle, récursive arborescente, ressemble à un évaluateur Lisp classique conçu par MacCarthy autour de l'année 1960. Les modifications indispensables pour qu'elle s'approche de l'interprète réaliste de ce langage sont les suivantes. (Décrivons le dialecte Scheme pour au moins deux raisons différentes : le lecteur doit le connaître, et il est beaucoup plus simple et plus homogène que Common Lisp). Le passage ci-dessous est une répétition du modèle déjà discuté !

- Les opérateurs sont n-aires. Tout nœud (expression) intermédiaire (non-feuille) a la forme d'une *liste d'expressions*, dont la première joue le rôle d'opérateur. Le cas terminal est un atome, ou une forme **`lambda`** explicite.

- Si l'opérateur est un atome, il doit être associé (possède la valeur) à une forme primitive – une référence au code-machine, ou une forme **`lambda`** (déjà compilée, bien sûr).

- Si c'est une autre chose, la machine évalue cette expression jusqu'à sa réduction à un objet fonctionnel explicit. Est-ce que ceci peut être une macro? Évidemment, ***non***, le système de réécriture des macros aurait dû se débarasser de toutes les transformations de la source avant l'exécution.

- La machine évalue tous les arguments de l'opérateur : tous les éléments de la liste sauf le premier (qui a été déjà réduit et attend ses arguments). Ils peuvent être mis dans une liste temporaire, ou stockés dans un vecteur temporaire.

- Si l'opérateur est une primitive, la machine exécute la procédure magique correspondante.

- *L'heure de vérité arrive*. L'objet fonctionnel est une forme **`lambda`** avec paramètres, et un corps, qui est une expression. Les paramètres sont des atomes. (Les extensions syntaxiques possibles, qui déstructurent automatiquement les arguments ne seront pas discutées ici ; ceci appartient à la couche macro).

  – La machine construit des associations entre les paramètres (chacun des symboles), et les valeurs des arguments stockées au préalable dans une structure temporaire.

  - **Cette association n'est pas statique, destructrice, mais elle empile des nouvelles associations sur celles déjà existantes !** Si le paramètre d'une forme **`lambda`** s'appelle **`x`**, ceci n'a rien à voir avec la variable **`x`** globale. Des associations hiérarchiques, empilées, implémentent la localité des variables-paramètres.
  - La machine évalue (récursivement, c'est-à-dire en empilant la continuation du contexte actuel) le corps de la fonction, et ceci est le résultat de l'expression, passé au niveau appelant.
  - Les associations locales sont détruites.

- Exception au protocole précédent : l'évaluation de l'opérateur résulte non pas en objet fonctionnel, mais en une forme primitive, une structure de contrôle, par exemple en **`if`**. À ce moment là, la machine peut déclarer son incompétence, et passer à l'opérateur spécial directement le reste de l'expression, par exemple la liste

  ```
  (<condition>  <expression-then>  <expression-else>)
  ```

  et laisser tout le reste à l'opérateur.

- Le langage peut prévoir la possibilité de programmer les formes spéciales explicitement par le programmeur. Ceci était jadis une technique populaire en Lisp (les pseudofonctions de type FEXPR), mais est tombé en désuétude, et a été remplacé par les macros (qui elles aussi sont des formes spéciales). Dans ce cas les actions de la machine peuvent être les suivantes.
  - L'interprète reconnaît que l'opérateur est une forme spéciale de haut niveau, une forme **`flambda`**, ou quelque chose de ce genre.
  - Les éléments restants de l'expression sont associés aux paramètres de l'opérateur, **mais sans être évalués**.
  - Ceci implique que *le code* peut être considéré comme une *valeur*. Nous savons déjà comment le réaliser.
  - L'opérateur, et alors le programmeur et le langage dispose de la fonction primitive **`eval`** qui relance récursivement la machine, permettant d'évaluer une expression de l'intérieur du code.
  - Sachant que cette évaluation se réduit dans la plupart des cas à l'application d'un opérateur «normal» à une collection de valeurs, le langage dispose d'une autre primitive : **`apply`**, qui applique son premier argument aux éléments de son second argument qui est une liste.

Attention. *Tout* est là, décrit en français. Une demande de transformer ceci en code Haskell est un *excellent sujet d'examen*.

En fait le mot *tout* souligné ci-dessus ne correspond pas à la réalité. Plusieurs choses ont été «cachées sous la moquette», et laissées à la discrétion du *run-time* du langage d'implantation de l'interprète :

- La récursivité du **`eval`**, et donc tout le bagage de la gestion des piles système.
- La création des associations paramètre – valeur. On peut utiliser des listes normales, mais la gestion de ces associations doit être primitive, malgré la simplicité des opérations.

14.2 Machine SECD

La machine SECD de Landin, publiée pour la première fois déjà en 1964, précise quelques détails de manière plus disciplinée, et optimise légèrement le protocole **`eval-apply`**, car c'est une machine de plus bas niveau, plutôt proche de notre machine linéaire, mais moins intuitive, plus formelle. En tout cas, elle de plus bas niveau que l'interprète ci-dessus, et elle compte sur la couche pre-compilation qui transforme toute *valeur* en instruction : *charger la valeur*, etc. SECD a été conçue pour décrire la sémantique opérationnelle des langages fonctionnels, mais Landin lui-même s'est vite rendu compte qu'elle permet d'établir une correspondance entre le calcul lambda et le langage Algol 60, impératif par excellence. La machine SECD est l'ancêtre d'un modèle plus récent, CEK, qui exploite les continuations. (Voir aussi la machine FAM de Luca Cardelli, et plusieurs exemples élaborés par ANdrew Appel).

Elle possède quatre registres globaux :

1. **S** : (Stack) – la pile qui contient les résultats intermédiaires durant l'évaluation.
2. **E** : (Environment) – une pile des *frames*, qui contient les valeurs associées aux variables. Ceci combine la liste temporaire et la liste des association mentionnées ci-dessus.
3. **C** (Control list) : le code – une structure dont l'élément directement accessible correspond à l'expression évaluée, ou à l'instruction exécutée.
4. **D** (Dump) : zone de stockage d'autres registres utilisée quand la machine exécute une nouvelle procédure. Ceci correspond à peu près à notre pile des retours, mais est un peu plus générale.

La machine de Landin est un automate qui exécute des transitions

```
S E C D  ⟶  S' E' C' D'
```

Par exemple, l'empilement d'une constante aura la forme décrite par la fonction **`(trans s e c d)`** en Haskell

```
trans s e (LOAD (Const x) : c) d  = ((x:s),e,c,d)
trans s e (LOAD (Var i) : c) d  = ((assoc i e:s),e,c,d)
```

Simple, n'est-ce pas? Si l'on veut, on peut passer à **`trans`** les 4-uples comme 1 argument, et non pas les 4 séparément («curryfiés»), mais ceci est une modification cosmétique. Nous ne décrivons pas ici la structure du *langage* interprété par la machine SECD et le lecteur doit regarder les exemples d'une certaine distance. On peut, par exemple, éliminer le mot-clé LOAD, ou l'amalgamer avec **`Const`** ou **`var`** en construisant les **`CodeItems`** : **`LoadConst`**, **`LoadVar`**, etc. La liberté de préciser les détails, c'est-à-dire de définir **de manière précise** les *types* des objets concernés, avec leurs balises de reconnaissance correspondantes reste à la discrétion du programmeur.

Si le code contient une fonction, qui sera empilée pour être exécutée plus tard (p. ex. sous **if-then-else**, il faut se rendre compte que celle-là n'est jamais autonome (sauf pour les opérateurs «purs», p. ex. primitifs), mais peut être obligée de décoder ses paramètres et en général, le contexte de son empilement. Elle aura besoin de l'environnement *actuel*.

```
trans s e (LOAD (Fun f) : c) d = ((FunEv f e):s,e,c,d
```

où **`FunEv`** est une constructeur de données (cela peut être une simple paire, sauf pour le type qui doit être une valeur légale), qui stocke sur la pile la fonction *ainsi que son environnement*. Ce problème dans notre petite machine du chapitre 3 a été a peine signalé sans détails, et nos exemples comme le cube ou la factorielle étaient des fonctions pures

Les opérations primitives peuvent être traitées comme *tags* ou balises, et interprétées comme suit :

```
trans (x:s) e (CAR :c) d = (car x :s,e,c,d)
trans (x:y:s) e (CONS : c) d = ((y:x):s,e,c,d)
```

etc. Le lecteur pourra sans problème reconstruire autres opérations primitives. Pour varier un peu, l'instruction IF cette fois sera parametrée différemment. On la considère comme un opérateur unaire qui attend sur la pile la valeur de la condition. Mais le code conditionnel à exécuter *fait partie de l'opérateur*, constitue son paramètre (double), comme pour les opérateurs de chargement.

```
trans (cnd:s) e (IF cthen celse : c) d cxxxx (c:d)  where
   cxxxx = case cnd of  True  -> cthen
                        False -> celse
```

Bien sûr :

```
trans s e (RETIF : _) (c:d) = (s,e,c,d)
```

où RETIF (appelée parfois JOIN) est le retour du branchement conditionnel. Ceci est une version très simplifiée d'un retour général d'une procédure.

L'application d'une fonction «non-magique» mérite une discussion approfondie. L'instruction APPL attend sur la pile des données une fonction à exécuter , ou plutôt une fermeture **`FunEv`** qui a «attrapée» l'environnement actif au moment de la création de cette fermeture.

Le code est un peu symbolique, la liste **`[a1, a2,...,an]`** symbolise n arguments stockés sur la pile, où n correspond à l'arité de la fonction

```
trans (FunEv f e' : [a1,a2,...,an] : s) e (APPL : c) d =
  [] ([a1,a2,...,an] : e') f (s:e:c:d)
```

Le lecteur notera des particularités suivantes.

- Le code **f** s'exécute dans un environnement enrichi par les valeurs des paramètres. (Ici mises dans une liste, mais d'autres stratégies sont possibles).

- L'adressage des paramètres devient alors simple. Si la forme **lambda** possédait trois paramètres formels : **x**, **y** et **z**, la compilation du corps de cette fonction transforme la référence à **x** en variable 0, **y** en variable 1, etc. **Mais attention !** Et si la fonction faisait partie d'une forme englobante, et possédait des variables globales?

  Le protocole SECD donne la réponse à la question qui n'a pas été abordée lors de notre construction de la machine virtuelle : la co-existence des environnements locaux, hiérarchiques. On introduit la notion de **frames** ou instances de'activation. Le *frame* le plus proche, l'environnement local porte le numéro 0, et **z** n'est plus identifié comme la variable avec indice 2, mais son indice est (0,2). L'accès à la variable globale appartenant à l'environnement englobant immédiat utilisera l'indice (1,i), etc. La création de ces références est la tâche du compilateur qui transforme Lisp (ou autre langage) end code SECD.

  Il faut donc modifier l'interprétation de l'instruction **(LOAD (Var i))** !

- La pile de données est vidée. Ceci est différent de notre machine style FORTH, où la pile ne changeait pas. Dans notre machine une fonction pouvait faire absolument tout avec la pile de données : placer 6 valeurs, récupérer 15, bouleverser l'ordre dans toute la pile, etc. Ceci a ses avantages, et c'est grâce à cette liberté que le compilateur FORTH ou PostScript est simple, mais la programmation manuelle est une galère. La machine SECD est plus structurée. Pour les langages de type Lisp, chaque procédure doit consommer exactement le nombre d'arguments qui correspond à son arité prédéfinie. Ainsi, l'exécution de l'expression (symboliquement) **(f a1 a2 ... an)** procède de manière simple. La pile des valeurs incomplètes n'est pas concernée, l'expression prend une pile toute fraîche, mais les valeurs des arguments sont dûment stockées dans l'environnement. Un tel appel doit sauvegarder pas seulement la continuation du code, mais aussi l'environnement et la pile du contexte.

- Le retour aura la forme suivante :

  ```
  trans (x:s') e' (RET:_) (s:e:c:d) = (x:s,e,c,d)
  ```

La construction incrémentale des environnements hiérarchiques ne suffit pas pour construire des fonctions récursives. En effet, si **f** appelle soi-même, où va-t-elle trouver le décodage de la variable **f**? La machine SECD en enrichissant l'environnement lors de l'appel de **f** par **f** aura des difficultés pour attribuer un statut à cette variable. En construisant notre machine nous avons résolu le problème par une indirection statique : Le code accède à une variable, dont l'indice correspond à l'association contenant ce code. C'est une solution correcte, mais pas très élégante et dangereuse. Le code récursif doit être autonome, de ne pas dépendre d'un tableau d'associations des variables. Il doit alors être auto-référentiel.

De nombreux exposés de la machine SECD, notamment ceux qui présentent la construction de cette machine en Scheme, proposent une solution qui utilise la modification physique de l'environnement. (Avec les procédures de modification physique il est facile de faire des listes circulaires, et autres abominations.) La solution purement fonctionnelle très intéressante existe, mais elle est un peu «magique», basée sur la sémantique paresseuse, et elle ne sera pas discutée ici. Elle a été présente plusieurs fois intuitivement.

14.3 Exercices

Q1. Trouver sur l'Internet les définitions de quelques machines virtuelles théoriques et pratiques, comme CEK, l'interprète de Reynolds, la spécification de la machine de Java, description du noyau du langage Smalltalk, le sf Pascal de UCSD, le CASE, etc.

R1. Pas de réponse ici. Cet exercice n'a pas beaucoup d'utilité pour ceux qui veulent seulement passer l'examen, mais peut être très enrichissant.

Q2. Transformer les spécifications informelles de l'interprète **eval-apply** en code réel. Coder en Haskell cet interprète. Ajouter un nombre minimal de primitives et de fonctions d'interfaçage, et tester la machine.

R2. On ne peut priver le lecteur de son travail intellectuel en fournissant la réponse à cet exercice.

Chapitre 15

Omissions

15.1 Généralités

Aucun cours de compilation n'est complet. Les omissions peuvent être assez importantes, et elles peuvent être classés en deux catégories :

1. Incompatibilité avec la philosophie du cours. Ainsi, nous ne pouvions discuter les techniques de très bas niveau.

2. Priorités. On ne peut satisfaire tout le monde, et le temps est toujours limité.

Le seul moyen de pouvoir reconnaître les limitations de ce cours est de chercher ailleurs, suivre les *newsgroups* consacrés à la compilation, lire livres et articles. Il est évident qu'il sera alors difficile de distinguer entre les choses plus ou moins importantes.

La suggestion qui s'impose est alors : essayer de faire son propre compilateur. Pas trop ambitieux, mais réalisé dès le début jusqu'à la fin. Il n'est pas judicieux de commencer par l'écriture d'un parseur, ni par la construction d'une machine virtuelle. La technique la moins douloureuse est d'exploiter les paquetages et les langages déjà existants, et de faire un compilateur-interprète qui utilise Scheme, ou Smalltalk, ou Python – des langages qui offrent un bon support à la manipulation des structures de données représentant des programmes, et un niveau d'intéractivité élévé. Toutes les lacunes d'aprentissage deviendront vite assez visibles. Passons à quelques lacunes concrètes.

15.2 Grammaires et parsing

- Nous avons à peine glissé sur la surface de l'analyse lexicale, expressions régulières, automates finis… Ceci fait partie d'un autre cours, mais un cours complet de compilation doit traiter ceci également, de point de vue plus pratique.

- La construction des parseurs LR (p. ex. la construction LALR) doit un jour faire partie de ces notes. Mais nous n'envisageons jamais traiter la théorie d'automates.

- Peut-être une introduction raisonnable à Lex et à Yacc serait utile, avec quelques exemples raisonnables (même si ceci est très contraire à la philosophie de l'auteur…).

- Il faut plus d'exemples de la technique de transformation source-source.

- Nous n'avons pu traiter quelques dispositifs d'optimisation, notamment l'élimination des sous-expressions identiques (communes). Aussi : élimination du «code mort» (*dead code*) inaccessible, qui peut être généré par le développement automatique d'une macro non-optimisée. Aussi : élimination des variables (registres) redondantes. Tut ceci ce sont des exercices en parcours des graphes.

- Nous n'avons pas traité sérieusement les erreurs du parsing, et les dispositifs permettant de «calmer» le parseur qui se trouve dans un état inextricable, et de reprendre la compilation. Un compilateur qui panique après la première faute découverte, est un jeu académique.

- Finalement, nous n'avons pas traité les *namespaces* ni les modules, pratiquement incontournables d'une façon ou autre si on veut compiler des programmes relativement volumineux.

15.3 Sémantique et génération du code

- Nous n'avons pas discuté le filtrage – la compilation des définitions fonctionnelles où la reconnaissance des arguments se fait par *pattern-matching*. C'est un petit fragment de la compilation, mais vraiment fascinant. Il constitue d'ailleurs un formidable champ d'entraînement de la technique des continuations.

- La création et la compilation des fermetures n'ont pas été décrites suffisamment bien. En particulier le «lambda-lifting» et/ou la liaison entre les blocs lexicaux dans un programme méritent un peu plus de place.

- Parfois un changement de la grammaire permet de remplacer les attributs hérités par des attributs synthétisés, ce qui est plus commode pour le parsing ascendant (à l'envers, nous avons insisté sur l'opération de normalisation qui *introduit* les attributs hérités). En général, le langage des attributs est très riche et varié, une bonne maîtrise de ce domaine distingue un vrai expert en compilation des gens qui «ont entendu parler de»…

- Notre discussion des types est *loin* d'être satisfaisante, et la version suivante de ces notes verra le chapitre correspondant beaucoup plus épais. Nous présenterons un vérificateur de types concret et relativement complet (sauf si nous décidons de proposer cela comme le devoir obligatoire ; dans ce cas tous les ingrédients seront dûment discutés).

- Il faudra également dire quelques mots sur la *compilation modulaire*, séparée. Comment préparer l'information pour un éditeur des liens (et qu'est-ce que c'est, cet éditeur des liens), comment gérer les relocations des adresses, etc. Malheureusement les détails sont ici très dépendants de l'architecture-cible, et les solutions portables sont peu nombreuses (mais elles existent, Scheme Smalltalk et Python en sont des exemples).

  Il serait bien de pouvoir dire quelques mots concernant la *compilation des librairies*, statiques et dynamiques.

15.4 Modèles d'exécution

- Programmation logique et la machine de Warren (WAM). Peut-être aussi la «machine de continuations binaires» de Paul Tarau, le noyau du BIN-Prolog. La programmation logique malgré l'échec du projet Japonais de «V-ème génération» continue à progresser, et l'unification, ainsi que le non-déterminisme trouvent de plus en plus leur place dans les systèmes de solutions de contraintes, interfaces graphiques, etc. Il est utile de savoir comment compiler de tels langages.

- Programmation par objets et fonctions virtuelles. Ceci a été évoqué plusieurs fois, mais jamais de manière bien structuré. Un jour nous pourrons – peut-être – décrire la machine de Smalltalk, et une partie essentielle de la machine de Java.

  Le problème a deux «visages». Il faut savoir générer le code qui profite de l'héritage, qui évite toute indirection qui peut être résolu statiquement, et d'éviter trop d'«arithmétique des pointeurs». Il faut aussi optimiser les indirections dynamiques – l'accès aux méthodes virtuelles.

- Programmation paresseuse : la technique de réduction paresseuse des graphes doit être au moins mentionnée. Les modèles de Haskell et Clean sont faciles à comprendre et bien documentés.

- Des machines fonctionnelles (ou presque) vraiment efficaces et pratiques, notamment le noyau du CAML, et la machine FAM de Luca Cardelli.

- Programmation événémentielle. En fait il ne s'agit pas d'une machine virtuelle proprement dite, mais d'un protocole de collaboration entre les modules, qui peut être implanté en C, ou autre langage impératif. Le code d'un programme piloté par les événements est un code «normal», impératif ou fonctionnel. Mais c'est un code spécifique, qui insiste sur la synchronisation, qui gère les ressources partagées, qui est adapté aux interruptions, etc.

  Il y a au moins deux problèmes distinctes ici : l'interfaçage, la communication du programme avec le système d'exploitation (gestionnaire des ressources) et ses gestionnaires d'événements, et la structure du code.

15.5 *Run-time* et l'interfaçage

- Démarrage et arrêt. Tout programme qui va s'exécuter doit être chargé dans la mémoire par un module spécial du système d'exploitation. Ceci est une opération complexe. La gestion des processus, les opérations **fork** ou **exec** (sous Unix) sont discutées ailleurs, ce domaine n'appartient pas à un cours de compilation. Mais le programme compilé doit y être préparé, surtout s'il contient des adresses accessibles de l'extérieur (p. ex., s'il s'agit d'un module faisant partie d'une librairie dynamique : DLL ou SO, selon le système).

  Il faut donc dire quelques mots à propos de *stubs* qui démarrent l'exécution, et sur la présence dans le code de l'information symbolique, facilitant le débogage.

- Les exceptions et les dispositifs de débogage font la différence entre un langage utilisable et un projet strictement académique. Ceci n'a pas été traité en détail, mais évoqué plusieurs fois. Il faut plus d'exemples et la discussion approfondie de la sécurité de programmation. Concrètement, il faut qu'aucune erreur ne soit capable de mettre en danger la sécurité ni la logique de l'allocation de mémoire, les fichiers et le système d'interruptions.

- Nous n'avons pas parlé ni des systèmes de fichiers, leur bufferisation, leur sécurité, etc., ni de la gestion des «pseudo-fichiers» liés aux événements extérieurs, comme la lecture de la console. Ceci évidemment ne fait pas partie de la compilation proprement dite, mais il faut savoir générer le code qui accomplit de telles tâches, qui est capable de collaborer avec la couche-système du langage.

15.6 *Varia*

- Plus d'exercices ! (Et plus de solutions, n'est-ce pas?)

- Éliminer au moins 25% de bavardage.

Annexe A

Introduction à la *véritable* programmation fonctionnelle et à Haskell

A.1 Pratique de la programmation en Haskell

Ceci n'est pas un manuel ! Pour apprivoiser ce langage il faudra lire la documentation. Cette section est pour votre pratique, pour pouvoir commencer à écrire des programmes relativement compliqués. Bien sûr, les exemples seront très nombreux, et leurs sujets seront choisis en fonction de leur utilité pour la compilation et la construction des interprètes, mais aussi pour montrer quel est le véritable sens du concept de «programmation fonctionnelle» .

Nous allons utiliser un compilateur : *Glasgow Haskell Compiler* – GHC, et deux interprètes de Haskell – Hugs (écrit par Mark Jones, actuellement à Oregon), et GHCI (Glasgow Interpreter, plus compatible avec GHC) installés sur les plates-formes Linux. Il faut que las binairies se trouvent sur le PATH, et il faut lire la documentation. Les interprètes se lancent depuis la ligne de commande (**hugs** ou **ghci**), et on peut leur passer quelques options d'exécution. Nos exemples (avec quelques rares exceptions) ont été exécutés sous Hugs.

On peut intéractivement entrer les expressions, et Hugs affiche le résultat. Voici le transcript d'une courte session :

```
Hugs mode: Restart with command line option +98 for Haskell 98 mode

Reading file "\Lang\Hugs\lib\/Prelude.hs":

Hugs session for:
\Lang\Hugs\lib\/Prelude.hs
Type :? for help
Prelude> [1,2,4,5] ++ [2 .. 9]
[1,2,4,5,2,3,4,5,6,7,8,9] :: [Integer]
(292 reductions, 521 cells)
Prelude> ch x where
ERROR: Undefined variable "x"
Prelude> ch x where x=1.5; ch y = (exp y + exp (-y))/2
2.35240962 :: Double
(35 reductions, 132 cells)
Prelude> product [1 .. 120]
6689502913449127057588118054090372586752746333138029810295671352301633557244962989366874165271984981308157637893214090552534408589408121859898481114389650005964960521256960000000000000000000000 :: Integer
(2552 reductions, 6660 cells)
Prelude>
```

On note l'usage de l'opérateur de concaténation **(++)** et la forme abrégée de l'intervalle :**[a .. b]**. La totalité de l'expression doit être placée sur une ligne, les expressions incomplètes ne sont pas admises. On peut séparer les parties d'une définition par les points-virgules. On peut aussi utiliser les accolades pour définir des

«blocs», mais nous allons éviter ce style. Notons également le fait que Hugs peut opérer sur les entiers de longueur quelconque.

On peut entrer une définition fonctionnelle, mais elle doit faire partie d'une expression, comme ci-dessus. On ne peut pas écrire **`ch x = ...`** et espérer que Hugs accepte ceci comme une définition fonctionnelle ! Si nous voulons définir des fonctions persistantes, ou simplement écrire des définitions longues, il faut les mettre dans un fichier extérieur, et le charger. Hugs veut avoir la possibilité de lire *toutes* les définitions courantes, afin de pouvoir effectuer leur analyse globale. (Il faut admettre que ceci n'est pas commode).

Les commandes de chargement sont : **`:l ...`** pour *load*, ce qui annule les définitions précédentes (sauf le Prélude), ou **`:a ...`** (*add*), ce qui augmente l'environnement par des nouvelles définitions. Il existe aussi un système de modules en Hugs (et en Haskell en général), mais pour l'instant nous allons l'ignorer. La directive **`:?`** affiche la liste de toutes les directives :

```
Prelude> :?
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.

:load <filenames>   load modules from specified files
:load               clear all files except prelude
:also <filenames>   read additional modules
:reload             repeat last load command
:project <filename> use project file
:edit <filename>    edit file
:edit               edit last module
:module <module>    set module for evaluating expressions
<expr>              evaluate expression
:type <expr>        print type of expression
:?                  display this list of commands
:set <options>      set command line options
:set                help on command line options
:names [pat]        list names currently in scope
:info <names>       describe named objects
:browse <modules>   browse names defined in <modules>
:find <name>        edit module containing definition of name
:!command           shell escape
:cd dir             change directory
:gc                 force garbage collection
:version            print Hugs version
:quit               exit Hugs interpreter
Prelude>
```

et la directive **`:set`** permet de voir/changer plusieurs options d'exécution de l'interprète :

```
Prelude> :set
TOGGLES: groups begin with +/- to turn options on/off resp.
s    Print no. reductions/cells after eval
t    Print type after evaluation
f    Terminate evaluation on first error
g    Print no. cells recovered after gc
l    Literate modules as default
e    Warn about errors in literate modules
.    Print dots to show progress
q    Print nothing to show progress
w    Always show which modules are loaded
k    Show kind errors in full
o    Allow overlapping instances
u    Use "show" to display results
i    Chase imports while loading modules
m    Use multi instance resolution

OTHER OPTIONS: (leading + or - makes no difference)
hnum Set heap size (cannot be changed within Hugs)
```

```
pstr Set prompt string to str
rstr Set repeat last expression string to str
Pstr Set search path for modules to str
Estr Use editor setting given by str
cnum Set constraint cutoff limit
Fstr Set preprocessor filter to str

Current settings: +stfewui -gl.qkom -h3000000 -p"%s> " -r$$ -c40
Search path     : -PD:\Lang\Hugs98  --- etc. ---
Editor setting  : -E
Preprocessor    : -F
Compatibility   : Hugs Extensions (-98)
Prelude>
```

Les fichiers extérieurs comportant les définitions des fonctions (et des types, etc.) normalement doivent avoir le suffixe **.hs**. Si le fichier a été accepté et chargé, et si vous voulez le modifier, pour le recharger il suffit d'écrire **:r** (comme *reload*).

Les techniques présentées ci-dessous sont *fonctionnelles et pures* (sans effets de bord). Programmation dans un style impératif, avec des *effets*, qui «font» quelque chose, et non pas seulement évaluent une expression, est aussi possible, mais elle est assez particulière et difficile à maîtriser. Il faudra d'abord maîtriser bien le concept de typage en Haskell.

A.2 L'essentiel

Dans le style fonctionnel les notions d'un objet et de sa valeur se confondent. Il n'y a pas de *modifications* de valeurs (affectation : «**:=**»), et dans le même environnement la variable **x** signifie *toujours* la même chose. Il est évident, que pour pouvoir construire des nouvelles valeurs il faut savoir appliquer des opérations aux arguments, mais toute affectation, toute opération de genre **x:=x+1** est strictement interdite. En Haskell la définition **x = y** est une identification.

Cette propriété est essentielle pour la *transparence référentielle* des programmes fonctionnels. Le fait qu'une variable signifie (dans son contexte) *une* chose, permet une optimisation assez agressive da la compilation.

La même syntaxe, un peu généralisée sert à définir les fonctions, on n'a pas besoin de mots-clés, comme **function**, SUBROUTINE, etc. Pour définir le cube d'un nombre nous écrivons

```
cube x = x*x*x
```

Absence d'affectations n'empêche pas l'usage de variables locales qui s'identifient avec les expressions qu'elles représentent. Voici la définition du sinus hyperbolique en Haskell :

```
sh x = (y-1.0/y)/2.0
  where y=exp x
```

ou, alternativement

```
sh x = let y=exp x
       in  (y-1.0/y)/2.0
```

où **where** ou **let** sont de rares mots réservés. Il faudra s'habituer à une particularité syntaxique de Haskell (existant dans quelques autres langages, comme Clean ou Python : l'indentation remplace le parenthésage – si la ligne suivante est plus indentée que la précédente, ceci signifie la continuation. La même indentation dénote une définition collatérale, au même niveau, et une indentation plus courte termine la structure syntaxique précédente, sans besoin de parenthèses. On peut aussi utiliser les accolades pour construire les blocs, et les points-virgule pour séparer les entités syntaxiques, mais nous allons éviter leur usage.

La même définition en Scheme serait, bien sûr

```
(define (sh x)
  (let ((y (exp x)))
       (/ (- y (recip y)) 2)))
```

Dans les deux cas conceptuellement important est le fait que ces définitions de fonctions sont des abréviations des opérations plus primitives :

- Création d'un **objet fonctionnel**, d'une valeur qui représente la fonction, et
- Assignation de cet objet fonctionnel à une variable.

Des fonctions anonymes existent aussi, la forme **lambda** en Scheme

```
(lambda (x) (let ((y (exp x))) (/ (- y (recip y)) 2)))
```

se traduit en Haskell par

```
\x -> (y-1.0/y)/2.0 where y=exp x
```

ou, si vous voulez :

```
\x -> let y=exp x in (y-1.0/y)/2.0
```

Si la variable définie dans **`let`** n'est pas utilisée de manière récursive, cette structure peut être remplacée – comme nous le savons déjà – par l'usage d'une fonction anonyme. Voici donc encore une autre variante de la fonction ci-dessus :

```
\x ->
   (\y -> (y-1.0/y)/2.0) (exp x)
```

La définition

```
f x = g x
```

est équivalente à

```
f = \x -> g x
```

ce qui peut être réduit à

```
f = g
```

mais attention, *toute* la vérité de cette simplification : **`\x->g x`** $\equiv$ **`g`** est un peu complexe à cause du typage, et sera discutée ailleurs. La dernière forme (**`f=g`**) peut ne pas être acceptée par le compilateur, tandis que la première si.

On peut aisément passer la valeur fonctionnelle d'une variable à une autre, par exemple **`monsh = sh`**, et appliquer **`monsh`** à une valeur. Alors une fonction est une donnée comme toute autre, avec quelques particularités :

- D'habitude il n'est pas possible de comparer deux fonctions. Nous ne pouvons dire si deux fonctions sont égales.
- Nous pouvons *appliquer* une fonction, et cette application est une opération «magique», primitive de la machine. En fait la définition de l'application fonctionnelle est la partie la plus importante de la *définition* d'une machine virtuelle. Rappelons que le mot «magique» sera utilisé très souvent et sa signification est très concrète : une action magique signifie que sa sémantique et sa réalisation appartiennent à la couche plus basse que celle qui est actuellement discutée.

La partie la plus importante du code d'un langage fonctionnel est la possibilité de construire des procédures *parametrées*, et de pouvoir *nommer* les arguments. Mais les noms existent uniquement dans le programme-source, pour la machine virtuelle qui exécute le programme, ceci n'a pas d'importance. Par contre, la possibilité de construire dynamiquement un objet fonctionnel (une fermeture) pendant l'exécution du programme, est très important et délicat pour toutes les couches d'exécution. Cette possibilité n'existe pas en C.

Mais attention ! La construction dynamique des fermetures n'implique nullement leur *compilation dynamique* ! Les morceaux de code «pur» (sans références extérieures) doivent être compilés statiquement, se qui se forme dynamiquement, lors de l'exécution du programme, c'est la liason de ce code aux valeurs de variables non-locales.

Le trait syntaxique le plus caractéristique de Haskell est la présence des **applications partielles** (qui, d'ailleurs, constituent la manière la plus simple de création des fermetures).

Si la fonction **`f`** accepte deux arguments, et **`f x y`** est une expression correcte, la forme **`f x`** en est correcte également, et définit un *objet fonctionnel* qui peut s'appliquer à l'argument manquant. À quelques détails près, **`f x`** $\equiv$ **`\y -> f x y`**. En général Haskell obéit à l'*ordre normal* d'évaluation des fragments d'une expression, de gauche à droite, et il est toujours possible d'instaurer ou d'effacer les parenthèses à gauche :

```
f x y z = (f x) y z = (f x y) z = ((f x) y) z
```

En Scheme la forme `\y -> f x y` est la seule syntaxe possible (concrètement : `(lambda (y) (f x y))`). Mais en Haskell nous pouvons même écrire `(* 2)`, définissant ainsi une fonction d'un argument, qui le multiplie par 2. La forme `(2 *)` est légale aussi. La forme `(*)` parfaitement correcte aussi, est une variante fonctionnelle de l'opérateur de multiplication, appliquée de gauche comme toute fonction typique, avec le nom alphanumérique. On peut écrire `(*) x y`.

A.2.1 Récursivité et processus itératifs

L'algorithme de Newton permettant de calculer la racine carré d'un nombre y a la forme itérative suivante :

$$x_0 = 1, \qquad x_{n+1} = \frac{1}{2}\left(x_n + \frac{y}{x_n}\right). \tag{A.1}$$

La réalisation fonctionnelle d'un tel processus n'est pas évidente pour ceux qui ont été conditionnés par le langage C. La solution, très classique, exploite au maximum les propriétés d'abstraction offertes par un bon langage fonctionnel. Nous aurons

1. Une fonction qui génère la valeur suivante, ici

   ```
   nxt y x = (x + y/x)/2.0
   ```

   où la valeur `y` doit constituer un paramètre supplémentaire, car nous voulons éviter l'usage de variables globales spécifiques à un problème.

2. Un prédicat (fonction Booléenne) qui vérifie la convergence.

   ```
   cnv x xnx = abs(x-xnx) < epsilon
   ```

   où `epsilon` peut être une constante ou une variable globale.

3. Un itérateur qui à partir de deux valeurs, la précédente et l'actuelle, construit une séquence qui se termine quand la convergence est atteinte.

   ```
   iterf x_prec x_nouv fun condit =
     if condit x_prec x_nouv then x_nouv
              else iterf x_nouv (fun x_nouv) fun condit
   ```

   La solution est donnée par l'expression

   ```
   racine y = iterf 1.0 (nxt y 1.0) (nxt y) cnv
   ```

   Il suffit de lancer `racine 2.0` pour obtenir 1.4142… Notez l'usage de l'application partielle `(nxt y)` passée comme le paramètre `fun`. On peut éliminer l'un des deux appels à `nxt y` :

   ```
   racine y = let fun = nxt y in
     iterf 1.0 (fun 1.0) fun cnv
   ```

Nous devrons savoir implanter effectivement la **récursivité**, ce qui implique non seulement l'usage non-trivial des **piles** (des données et de contrôle), mais aussi l'optimisation de la **récursivité terminale** et les **exceptions** dans la gestion de la pile. Par contre, il faut éviter le piège d'implanter la récursivité en utilisant la récursivité, ce qui parfois est exploité pour modéliser l'interprète de Lisp en Lisp.

Il est à retenir également le fait que dans le monde fonctionnel on écrit *très souvent* des fonctions qui prennent d'autres fonctions comme paramètres, et les appliquent, comme notre itérateur. Mais à part cela la machine virtuelle fonctionnelle sous-jacente est la simplicité même. Elle doit savoir

1. décoder les valeurs des variables dans l'environnement ;
2. exécuter l'opération primitive «application» ;
3. posséder au moins une structure de contrôle primitive type **if-then-else** : application de f ou g selon la valeur d'un troisième objet, la condition;
4. gérer la mémoire, posséder un système des entrées/sorties, etc., tout ce qui appartient à la couche «système».

A.2.2 Évaluation paresseuse

Quel est le résultat de l'application fonctionnelle **`f(1/0)`**? (Supposons que le compilateur n'est pas très intelligent et que la singularité n'a pas été découverte avant son apparition lors de l'appel incriminé. Typiquement le programme déclenche une exception. Mais si la fonction **`f`** n'a pas besoin de son argument, par exemple

```
f x xtra = if xtra<0 then sqrt x else xtra

f (1/0) 7
```

le résultat peut être égal à 7, à condition que la fonction «avale» l'expression singulière sans déclencher une erreur.

Le protocole d'évaluation qui le permet, s'appelle *évaluation paresseuse*, ou *non-stricte*. Plus précisément, un langage est strict, si **toute** fonction **`f`** appliquée à l'expression $\perp$ qui symbolise un calcul qui ne peut se terminer, rend aussi $\perp$. La façon pratique de réaliser le protocole paresseux est le *passage de paramètres par nom*, et non pas par valeur. Quand l'application fonctionnelle **`(f x)`** est compilée, une partie du code est le résultat de la compilation de **`x`**. Ensuite...

- Dans un langage strict ce code est exécuté d'abord, et la valeur retournée ou sa référence est transmise au code qui exécute **`f`**.
- Le protocole paresseux demande que le code représentant **`x`** ne soit pas exécuté tout de suite. Il est transformé en une procédure anonyme et sans arguments qui traditionnellement porte le nom de *thunk*. Ce thunk est transmis à la fonction qui peut l'exécuter ou non. Il est exécuté quand la fonction utilise explicitement cet argument, quand sa valeur devient indispensable.

L'utilité de l'évaluation paresseuse repose sur le fait que ***les fonctions paresseuses peuvent représenter des structures de contrôle*** . Par exemple, l'expression $\mathbf{if}\ c\ \mathbf{then}\ A\ \mathbf{else}\ B$ peut être considérée comme l'application fonctionnelle de l'opérateur **`ifelse`**: **`(ifelse c A B)`**. Mais il est évident que **`A`** et **`B`** ne peuvent être évaluées avant de passer le contrôle à la fonction **`ifelse`**. Seulement une de ces expressions sera évaluée, selon la valeur de la condition. Donc, c'est la fonction **`ifelse`** qui doit «consciemment» demander l'évaluation soit de **`A`**, soit de **`B`**.

Ensuite, comme nous verrons plus tard, les listes paresseuses constituent une méthode très lisible et intuitive de représenter un flux (*stream*) – de caractères, de mots, etc., ce qui permet d'établir la communication entre plusieurs modules du compilateur. La transformation (ou *transduction*) des flux de données est devenu si paradigmatique, que même des langages fonctionnels stricts comme ML (notamment CAML) définissent des flux paresseux. Et les langages classiques comme C? Ici ce concept est réalisé par les *pipes* (Unix), mais les *pipes* sont des dispositifs purement techniques, sans trop de théorie sous-jacente. Les flux paresseux peuvent aussi modéliser avec une grande souplesse et clarté *des processus itératifs*, et ceci jouera un rôle très important dans notre cours.

Exemple. En Haskell le caractère **`(:)`** est l'opérateur connu en Lisp comme **`cons`** – le constructeur des listes. L'expression **`1 : 2 : []`** signifie le même que **`1 : [2]`** et **`[1,2]`**. Voici la construction d'une liste *infinie* **`1, 2, 3, 4, ...`**, qui peut être utilisée comme telle dans un langage paresseux : il suffit de *ne pas regarder* la queue de la liste pour qu'elle soit cachée dans son thunk, un «démon» procédural fini. Si on demande la valeur du second élément de la liste, le démon le génère et se cache derrière, etc. La construction de la liste a la forme suivante :

```
infint = intseq 1
 where
 intseq n = n : intseq (n+1)
```

Notez une propriété sémantiquement pratiquement illégale en C : la fonction récursive **`intseq`** n'a pas de clause terminale !

(Notez aussi **une fois pour toutes** qu'en Haskell la construction **`(1 : 2)`** est *absolument* illégale à cause du typage. Le second argument doit être une liste, non pas un objet quelconque. Pour former des tuples on utilise la notation **`(1,2)`**.)

L'évaluation paresseuse, surtout cascadée peur encombrer la mémoire avec des *thunks* ce qui peut être très indésirable. En général la paresse introduit une certaine pénalité d'exécution, et si on pouvait dans quelques cas critiques forcer l'évluation stricte desarguments d'une fonction, ceci augmenterait la vitesse d'exécution du programme. Ceciest possible grâce à la construction **`seq :: a -> b -> b`**.

La sémantique de cette construction est simple : **`seq a b`** $\equiv$ **`b`**, si **`a`** $\neq \perp$. Le Prélude définit la fonction **`$!`**, l'opérateur d'application stricte:

```
f $! x = x `seq` f x
```

associatif à droite, et de faible précédence.

A.2.3 Déstructuration automatique des arguments

En C, Pascal ou Lisp classique, y compris Scheme, les paramètres formels d'une procédure sont toujours des noms simples. Si l'argument actuel lors de l'appel est une expression composite : une liste, un record, etc., il faut lancer manuellement un sélecteur d'accès : CAR, CADDR, **`argum.champ`**, etc., pour pouvoir voir l'intérieur de l'argument.

Les langages fonctionnels de nouvelle génération (ainsi que quelques langages logiques, notamment la famille Prolog, et les nouvelles implantations du Lisp) offrent à l'utilisateur la possibilité de déstructurer les paramètres automatiquement via «*pattern-matching*» (filtrage). Si le paramètre formel dans la définition d'une fonction n'est pas un nom, mais une structure, par exemple **`(x:q)`** (en Haskell) ceci signifie que l'argument actuel sera une liste dont la tête est accessible dans la fonction par le nom **`x`**, et la queue s'appellera **`q`**. Le caractère _ dénote une variable anonyme, toujours différente des autres, et qui ne nous intéresse pas (elle ne sera pas utilisée par la fonction).

La construction **`nom@struc`** signifie : le paramètre s'appelle **`nom`** et il possède la structure **`struc`**. Voici la fonction qui duplique la tête d'une liste, p. ex. $[5,3,4,6] \rightarrow [5,5,3,4,6]$:

```
dtete l@(x:_) = x:l
```

A.2.4 Quelques exemples de programmes en Haskell

Le but de ces exemples est principalement pédagogique, il nous faut maîtriser ce langage.

Voici le tri par insertion d'une liste numérique. Le lecteur doit *bien* connaître la version Scheme de cet algorithme. Notez comment au lieu de proliférer les conditions **`if-then-else`**, le programme est décomposé en clauses. Notez aussi comment les alternatives sont représentées par les clauses internes, séparées par les barres verticales.

```
instri [] = []
instri (x:q) = inserer x (instri q) where
  inserer x [] = [x]
  inserer x l@(y:r) | x<y        = x:l       --(Nouvelle tête, sinon
                    | otherwise  = y : (inserer x r)   -- garde l'ancienne)
```

Le deuxième exemple sépare une liste en deux morceaux : les n premiers éléments, et le reste. La valeur retournée est une *paire* **`(prm,rst)`**, une structure primitive de Haskell, très commode. Il existent aussi des triplets **`(x,y,z)`**, etc., qui sont implantées de manière plus efficaces que les listes.

La première ligne de l'exemple (qui est une fonction prédéfinie) est sa *déclaration de type*. Elle est redondante, mais le typage explicite peut nous être utile plus tard, pour restreindre le polymorphisme, ou simplement pour la documentation. La notation est plus compacte qu'en C.

```
splitAt               :: Int -> [a] -> ([a], [a])

splitAt 0 xs           = ([],xs)              -- tête vide
splitAt _ []           = ([],[])              -- tout vide
splitAt n (x:xs)
  | n>0 = (x:xs',xs'')                        -- récursive
    where (xs',xs'') = splitAt (n-1) xs       -- sinon...

splitAt _ _            = erreur "neg. arg."
```

Le dernier exemple est l'usage d'une fonctionnelle de réduction des listes. Comment calculer la somme de tous les éléments de la liste `[x1, x2, x3, ... xN]`? La technique apprise à l'école maternelle est simple on ajoute le premier élément à la somme des éléments restants. Mais avec le produit nous ferons le même, seulement l'opérateur changera. La structure récursive du code reste la même, et il est de rigueur dans le monde fonctionnel de profiter de toute généricité, et de définir les fonctions permettant leur généralisation facile. Alors dans notre cas la somme est le résultat de la *réduction* (*folding* : «pliage») de l'opérateur `(+)` sur la liste.

```
somme l = foldl (+) 0 l
```

où le réducteur a la forme

```
foldl :: (a -> b -> a) -> a -> [b] -> a

foldl op z []      = z
foldl op z (x:xs)  = foldl op (op z x) xs
```

Naturellement le produit des éléments sera donné par `foldl (*) 1 l`.

A.3 Langage de base

A.3.1 Opérations

Après un avant-goût syntaxique et quelques exemples, soyons un peu plus formels. Récapitulons les ingrédients syntaxiques de base. Les expressions peuvent être des constantes (numériques ou autres), variables comme `alpha_23j`, etc. On peut utiliser la notation fonctionnelle : `fun arg1 15 arg3` etc., ou opérationnelle : `(p 'mod' 23)*12 + 7^u`, etc., avec les précédences des opérateurs arithmétiques similaires à d'autres langages.

Toute fonction binaire, p. ex. `funct` peut être utilisée comme opérateur infixe par la notation : `a 'fun' b`, et tout opérateur infixe peut être utilisé comme une fonction préfixe en l'entourant par les parenthèses : `(*) a b`. On peut définir les opérateurs utilisateur, et leur donner les précédences et l'associativité (gauche ou droite) grâce aux commandes `infixl, infixr` et `infix` (aucune associativité). Voici un fragment du Prélude :

```
infixr 9  .
infixl 9  !!
infixr 8  ^, ^^, **
infixl 7  *, /, 'quot', 'rem', 'div', 'mod', :%, %
infixl 6  +, -
-- infixr 5  :    -- Cette déclaration est figée dans le noyau du Hugs
infixr 5  ++
infix  4  ==, /=, <, <=, >=, >, 'elem', 'notElem'
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, 'seq'
```

Le nombre plus élévé signifie la précédence (la «force» de l'opérateur) plus grande. L'application fonctionnelle a par défaut la plus grande précédence de tous les opérateurs.

Pour définir un objet il suffit de le faire suivre par le caractère `=` (ce n'est pas un opérateur), et par sa définition. Si à gauche de l'affectation on trouve une forme plus complexe qu'une simple variable, c'est une définition fonctionnelle. Exemples :

```
pi = 3.1415926536

tg x = sin x/cos x
ff x y = sqrt (x*x+y*y)
```

Notons l'absence de parenthèses redondantes autour de paramètres. Haskell dispose de toute la panoplie de fonctions et d'opérateurs numériques standard. Les fonctions comme `sqrt`, `exp` etc. acceptent les arguments

flottants, et `sqrt x` où x est égal à 4, est illégal. Cependant l'expression `sqrt 4` est correcte et donne 2.0. Haskell automatiquement compile les constantes numériques, p. ex. 4 comme `fromInteger 4`, où `frominteger` est une fonction surchargée, qui effectue une conversion de son argument vers un autre type quelconque – le type qui est attendu par le contexte. Puisque la fontion $\sqrt{\ldots}$ attend un argument flottant, le résultat de la conversion sera flottant.

A.3.2 Types de données prédéfinis

En Haskell nous pouvons utiliser les nombres entiers de précision illimité : le type `Integer`, (les entiers courts standard existent aussi : `Int`), les flotants (`Float` et `Double`), les Booléens, `Bool` : `True` et `False`, et les caractères `Char` : `'c'` comme les données atomiques standard. Il existe aussi un objet «vide» : `()` (de type `()`).

Les objets composites standard sont :

- les «tuples» : `(a,b)`, `(a,b,c)`, etc., où les éléments peuvent être de type quelconque (aussi les tuples). Donc, les tuples constituent une collection infinie de types possibles. Si `a`, et `b` designent un type, alors `(a,b)` ainsi que `(a,b,c)` etc., sont des types légaux.
- Les listes : `[a,b,c,d]`, etc. Cette liste est équivalente à `a:b:c:d:[]` ; (l'opérateur `(:)` est l'équivalent du «cons» en Lisp). Tous les éléments d'une liste doivent appartenir au même type, p. ex. `a` – alors le type de la liste est `[a]`.

  Les chaînes alphanumériques : `"Belle Marquise"` sont équivalentes à des listes de caractères : `['B','e','l','l','e',' ','M','a','r','q','u','i','s','e']`. Leur type est `[Char]`, mais souvent on utilise une abbréviation `String`.

Plus tard nous verrons encore les «records», et les tableaux, mais nous allons utiliser de préférence les listes, et les structures «algébriques» définies par l'utilisateur.

Le langage dispose d'un outil syntaxique très élégant – les compréhensions. Pour transformer une liste de nombres en une liste de leur cubes on peut naturellement utiliser la fonctionnelle `map`

```
l = [1, 3, 5, 2, 4]
r = map cube l  where cube x = x*x*x
```

Ceci est la syntaxe standard. Avec les compréhensions nous écrirons

```
r = [cube x | x<-l]
```

Donc, les compréhensions remplacent le générateur `map` par une syntaxe plus lisible, mais elles peuvent également *filtrer* les éléments d'une liste. L'expression ci-dessous renvoie la liste des entiers qui ne soient pas divisible par 3 :

```
l=1 .. 20
r=[x | x<-l, x `mod` 3 /= 0]
```

Le résultat affiché est

```
[1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20]
```

Normalement on n'a pas besoin de déclarer les types des variables, fonctions, etc., mais ceci peut être utile pour la documentation, ou pour supprimer les ambiguïtés qui parfois peuvent se produire. Voici deux styles de déclaration de variables avec leur types :

```
x :: [Double]
x = [1.2, 2.6]

y = "Belle Marquise" :: String
```

Un type dérivé très important c'est le type fonctionnel. Si `a` et `b` sont deux types, la fonction qui prend un argument de type `a` et renvoie le résultat de type `b`, possède le type `a -> b`. Si la fonction `f`prend deux arguments, de type `a` et `b`, et son résultat est de type `c`, on écrira :

```
f :: a -> b -> c
```

La flèche peut être considérée comme un opérateur associatif à droite. On peut placer explicitement les parenthèses : `f :: a -> (b -> c)`. Cette convention est en accord avec le protocole de «curryification», selon lequel l'expression `f x y` est équivalente à `(f x) y`. Une «application partielle» d'une fonction binaire à son premier argument est légale, et dénote une fonction *unaire* qui attend le second argument. Concrètement, nous avons (presque) l'équivalence

```
f x   ≡   \y -> f x y
```

Important ! Analysez pourquoi l'associativité à gauche des applications fonctionnelles produit l'associativité à droite de la flèche qui dénote les types fonctionnels. (D'ailleurs, pour les débutants la flèche en Haskell est une source de confusion. Elle est utilisée aussi pour *définir les fonctions* – comme ci-dessus (et non pas seulement pour spécifier leur type), et pour la structure de contrôle `case`).

Les opérateurs infixes peuvent faire partie des «sections» : `(* 8)` ou `(8 /)`.

A.3.3 L'utilisation de l'évaluation paresseuse

La puissance de la paresse est visible quand la fonction en question est un *constructeur de données*, par exemple l'opérateur `(:)`. On peut construire facilement une liste infinie (cyclique) d'un ou plusieurs éléments. Voici comment construire des listes comme `[2,2,2,...]`, ou `[2,3,1,2,3,1,2,...]` :

```
repeat c = w where
  w = c : w
deuxs = repeat 2

cycle l = w where
  w = l ++ w
ccl = cycle [2,3,1]
```

où on aurait pu définir : `repeat c = c :  repeat c`, mais cette dernière définition malgré sa simplicité est moins efficace. La raison est très simple. La première variante génère une liste cyclique, l'objet `w` est autoreférentiel, et la représentation de cette liste dans la mémoire est très compacte. La deuxième variante construit une paire dont la tête est égale à `c`, mais la queue est une fonction dont l'appel génère la même liste.

Voici un autre exemple, la liste `[1,2,3,...]` :

```
entiers = ent_de 1 where
  ent_de n = n : ent_de (n+1)
```

Le trait caractéristique des fonctions récursives paresseuses est la présence de récursivité *ouverte*, sans la clause terminale. Ceci n'est pas gênant, car une consommation incrémentale d'une liste infinie «voit» uniquement le segment initial, et la queue infinie est «virtuelle», représentée par un thunk qui se «dé-virtualise» au fur et à mesure quand le programme progresse.

Les fonctions récursives ouvertes n'épuisent pas nos surprises. L'évaluation paresseuse permet également de créer des *données récursives*. Un exemple a déjà été vu : les listes cycliques. Mais regardons une autre définition récursive de la liste des entiers :

```
entiers = 1 : (entiers <+> uns) where
  uns = 1 : uns
  (x : xq) <+> (y : yq) = (x+y) : (xq <+> yq)
```

Ici l'opérateur `(<+>)` sert à ajouter deux listes, élément par élément. Il peut d'ailleurs être défini par une fonction prédéfinie `zipWith` qui applique un opérateur quelconque à deux listes, élément par élément :

```
zipWith _ [] [] = []
zipWith op (x:xq) (y:yq) = op x y : zipWith op xq yq

(<+>) = zipWith (+)
```

Cette définition des `entiers` est loin d'être triviale. C'est une liste dont la tête est égale à 1. La queue est le résultat d'addition des `entiers` et la liste infinie des 1. Apparemment ceci n'est pas faisable, mais on sait quelle est la tête du résultat: $1 + 1 = 2$. Donc, deux éléments sont définis : `entiers = [1,2,...]`. Si le

second élément de la liste est égale à 2, le second élément de sa queue est égal à 3, ce qui établit la valeur du troisième élément, etc.

Nous voyons donc que l'évaluation paresseuse permet de remplacer un processus itératif par une structure de données «auto-génératrice».

Un autre usage des listes paresseuses est la consommation de flots qui modélisent les fichiers dans le programme. On peut considérer qu'un fichier lu est une liste très longue de caractères. Ils sont «virtuellement» tous présents dans le programme, et on peut consommer itérativement ce flot. Dans la réalité ils sont lus incrémentallement. On n'a pas besoin de lire les items dans une boucle, on *lit le fichier une seule fois, dans un seul endroit dans le programme*, et on traite le contenu comme s'il était une structure interne.

Attention ! La construction d'un tel programme peut être délicate. Si on consomme un flot paresseux par une structure itérative, mais on oublie de «libérer» les éléments lus et traités, la liste entière se forme physiquement dans la mémoire, et on risque le débordement du tas. C'est une de nombreuses raisons pour lesquelles dans les modèles sérieux d'entrée/sortie en Haskell on procède différemment. Mais la fonctionnalité existe, et elle est bonne pour l'entraînement. Il faut d'abord charger le fichier **`IOExts.hs`** qui réside dans la librairie d'extensions du Hugs. Ensuite la définition :

```
chaine = unsafePerformIO (readFile "nom_du_fichier")
```

place – paresseusement – dans **`chaine`** la totalité du fichier lu.

A.4 Structures de contrôle

Ici Haskell est très austère. Nous disposons de fonctions et d'opérateurs infixes, et nous pouvons créer des applications partielles, et fonctions anonymes, p. ex., **`\x -> 6*x`** (ce qui est équivalent à la section **`(6*)`**). Le nombre de dispositifs syntaxiques qui facilitent la création d'un programme complexe est limité.

- Il existe naturellement l'expression conditionnelle **`if`** α **`then`** β **`else`** γ. Elle n'est pas très souvent utilisé, car les «gardes», et le filtrage des paramètres sont souvent plus commodes.

- On peut créer des variables (et les fonctions) locales à une fonction par le bloc **`let`** *définitions locales* **`in`** *expression*. Les blocs **`let`** peuvent être imbriqués à volonté, et les définitions locales peuvent être récursives (y compris indirectement). L'alternative à **`let`** est, bien sûr, la construction **`where`**. (Il faut seulement souligner que la présence de **`where`** en cascade, à plusieurs niveaux, n'est pas lisible, et dans les version précédentes de Hugs ceci était interdit).

  Les définitions locales donnent un autre moyen de créer des fermetures non-triviales. La forme

  ```
  f x = let
          g y = x+y
        in  g
  ```

  utilisée dans le contexte : **`h = f 5.0`** affecte à **`h`** une fonction qui ajoute 5.0 à son argument. La constante 5.0 a été «attrappée». (En fait, cet exemple est trop simple, on peut le transformer en : **`f x = (+ x)`** ou même en **`f = (+)`**. Mais vérifier le typage !

- Il existe une conditionnelle à choix multiples :

  `case` *expression* **`of`**
  valeur1 **`->`** *resultat1*
  valeur2 **`->`** *resultat2*
  `...`

A.4.1 Clauses, gardes et filtrage

Comme nous avons déjà mentionné, Haskell partage avec Prolog la possibilité de *filtrer* automatiquement les valeurs et les formes des arguments des fonctions, ce qui permet d'éviter la prolifération des conditionnelles. La factorielle récursive peut être définie ainsi :

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

au lieu d'expliciter :

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

On peut exploiter encore un autre style, avec des «gardes», formes conditionnelles qui ont la forme un peu similaire au **case**, mais où ce ne sont pas des valeurs, mais des conditions Booléennes qui déterminent les branches à suivre. Exemple :

```
fac n | n==0 = 1
      | n>0  = n * fac (n-1)
      |otherwise = error "Factorielle ; argument négatif"
```

Le mot **otherwise** est un synonyme de **True**.

Le filtrage des paramètres permet de déstructurer automatiquement un argument composite. Si dans une définition fonctionnelle le paramètre n'est pas un identificateur, mais une forme composite, sa structure doit correspondre à l'argument actuel, et les identificateurs présents dans cette forme sont instanciées avec les éléments correspondants de l'argument. Voici la précédure qui renverse une liste (avec accumulateur) :

```
reverse l = rev l [] where
  rev [] b = b
  rev (x:xq) b = rev xq (x:b)
```

Le filtrage en Haskell n'est pas l'unification complète, comme en Prolog. Il n'y a pas de «variables logiques» non-instanciées. Chaque identificateur peut apparaître une seule fois dans le *pattern* (forme), et même une seule fois dans la liste qui spécifie les patterns pour tous les paramètres d'une fonction.

Si on veut *nommer* le paramètre, comme dans d'autres langages de programmation, et avoir en même temps sa structure, on utilise la notation **nom@forme**. Voici une fonction qui parcourt une liste de paires : **[(nom1,val1),(nom2,val2),...]**, et qui retourne la paire dont le premier élément est égal à **z**. En cas d'échec on renvoie un message d'erreur (plus tard nous apprendrons comment sortir d'une telle situation de manière plus douce).

```
cherche _ [] = error "Échec de recherche"
cherche z (e@(x,_) : rst) | x==z = e
                          | otherwise = cherche z rst
```

Notons l'usage des variables anonymes (_) là, où leur instanciation est inutile.

Même si les sélecteurs des éléments des structures de données composites classiques : **head** et **tail** pour les listes, **fst**, **snd** pour les paires, etc. sont prédéfinis, on les utilise rarement, grâce au filtrage structurel.

A.4.2 Fonctions d'ordre supérieur

Dans un langage fonctionnel sérieux, les fonctions sont des données comme les autres. Elles peuvent être stockées, passées comme paramètres des autres fonctions, ou constituer le résultat d'une opération. L'usage des fonctions qui opèrent sur d'autres fonctions est un trait très typique de la programmation fonctionnelle. Le style fonctionnel préconise la définition et l'usage de **fonctions génériques**, universelles : les itérateurs, filtres, opérateurs de composition et autres combinateurs. Une partie obligatoire du cours de Scheme est la définition et l'usage de la fonctionnelle **map**. En Haskell elle aura la forme

```
map fun [] = []
map fun (x:xq) = fun x : map fun xq
```

mais nous connaîtrons encore beaucoup d'autres. Dans un langage fonctionnel *pur* l'usage des fonctionnelles remplace les structures de contrôle typiques comme les boucles avec l'accumulation, etc. Par exemple, pour trouver la somme des éléments d'une liste on peut écrire

```
somme [] = 0
somme (x:xq) = x + somme xq
-- ou, avec l'accumulateur
somem l = sm l 0 where
  sm [] n = n
  sm (x:xq) n = sm xq (n+x)
```

mais pour les fondamentalistes fonctionnels il est de rigueur l'usage de la fonctionnelle universelle **`fold`** (elle existe en deux variantes prédéfinies : **`foldl`** (l – *left*) récursive terminale, et **`foldr`**(r – *right*), incrémentale, adaptée à la programmation paresseuse, et déjà présentée). Voici leur définitions et l'usage :

```
foldr f z []      = z
foldr f z (x:xq)  = f x (foldr f z xq)

foldl f z []      = z
foldl f z (x:xq)  = foldl f (f z x) xq

somme l = foldl (+) 0 l
```

Pour combiner élément par élément deux listes nous avons **`zipWith`** (définie précédemment).

Voici un «itérateur» typique, la fonction **`iterate f x`** qui construit la liste infinie **`[x, f x, f(f x), f(f(f x)),...]`**.

```
iterate f x = x : iterate f (f x)
```

Une telle liste peut être ensuite parcourue par un autre itérateur-filtre, qui – par exemple – cherche à satisfaire une condition satisfaite par un élément, ou une convergence de cette suite. Voici comment chercher la solution numérique, itérative de l'équation $\exp(x) = 3x$.

```
eps = 0.000001
x0 = 1.0
fun x = (exp x)/3.0

converg (x:xq) = cvg x xq where
  cvg x (y:yq) | abs(x-y)<eps = y
               | otherwise = cvg y yq

solution = converg (iterate fun x0)
```

On peut utiliser aussi une fonction de filtrage universel, qui élimine de la liste tous les éléments qui ne satisfont pas la condition Booléenne **`p`**:

```
filter p [] = []
filter p (x:xq) | p x = x : filter p xq
                | otherwise = filter p xq
```

Plus tard nous connaîtrons ecore plusieurs autres fonctionnelles. Rappelons encore qu'en Haskell la fonctionnelle **`map`** et les filtres prennent souvent la forme de *compréhensions*, formes de genre **`[expr | gen_et_filtr]`**. Au lieu d'écrire **`map f l`** nous pouvons coder la forme un peu moins compacte, mais très lisible :

```
[f x | x <- l]
```

Si accessoirement nous voulons filtrer le résultat par le prédicat **`p`**, nous écrirons

```
[f x | x <- l, p x]
```

On peut générer l'expression à gauche de la barre à partir de plusieurs listes, et utiliser plusieurs filtres. La forme **`[(x,y)|x<-[1 .. 3],y<-[2 .. 4]]`** engendre **`[(1,2), (1,3), (1,4), (2,2), (2,3), (2,4), (3,2), (3,3), (3,4)]`**. Hugs et GHC permettent aussi l'usage des compréhensions «parallèles» :

```
[(x,y)|x<-[1..3]|y<-[2..4]] → [(1,2),(2,3),(3,4)]
```

A.5 Types définis par l'utilisateur

On peut affirmer que la richesse du système de typage est le trait fondamental des langages de programmation modernes, et en particulier des langages fonctionnels. Nous allons aborder ici la construction de *types algébriques*, équivalentes aux records (balisés) avec des variantes. Dans la notation qui sera désormais intensément exploité le type prédéfini **`Bool`** est algébrique, défini par

```
data Bool = True | False
```

Le mot-clé **data** introduit la définition d'un nouveau type. À droite du signe d'affectation on trouve l'énumération des ***constantes symboliques*** qui représentent les valeurs appartenant à ce type. On peut définir d'autres types de ce genre, p. ex. :

```
data Ordering  = LT | EQ | GT
data Couleur   = Rouge | Blanc | Bleu | Vert | Marron | Arc_enCiel
```

(Le premier est prédéfini dans la librairie standard). Les constructeurs des constantes doivent être distincts. Par convention les constantes symboliques en Haskell commencent par la lettre majuscule.

Les types structurés sont parametrés par les types des composantes. Voici comment on peut définir les nombres complexes comme des paires de deux réels :

```
data Complex = Complex Double Double
```

où l'identificateur **Complex** dénote simultanément le nom du type, et de son constructeur (le seul ici). Pour construire un nombre complexe concret il suffit d'écrire **Complex 1.3 8.7** etc. Les structures de ce genre sont correctement traitées par le mécanisme de filtrage des entêtes. Voici la définition de la fonction qui multiplie deux nombres complexes :

```
cmult (Complex a b) (Complex x y) = Complex (a*x-b*y) (a*y+b*x)
```

Les types en Haskell peuvent être **polymorphes**, parametrés par des types «variables», inconnus. Le constructeur n'est pas forcément une constante alphanumérique, il peut être aussi un opérateur infixe. Voici la création des fractions *num/den*. nous ne voulons pas figer le type des composantes. Il peut être **Integer**, mais aussi **Int**, ou, peut-être représenter le polynômes, pour qui l'arithmétique classique comme l'addition, la multiplication et la division Euclidéenne avec reste, est définie comme pour les entiers. Nous écrirons donc

```
infix 7  :%

data Fraction a = a :% a
```

où **a** dénote ce type inconnu. Plus tard nous apprendrons comment restreindre la catégorie de types «éligibles» pour le numérateur et le dénominatur (pour éviter la formation de fractions composés de chaînes de caractères, ou des chauve-souris). Une variable de type rationnel peut être défini par :

```
frct = x :% 8 :: Fraction Int
```

En fait, la vie est plus difficile. En Hugs l'utilisateur *n'a pas le droit d'utiliser l'opérateur* **(:%)**! La raison est très simple : ce constructeur permet de former impunément des fractions réductibles comme **2 :% -4**. Donc cet opérateur est «caché» (il n'est pas *exporté* du Prélude Standard), et l'utilisateur peut former les fractions avec un autre opérateur, par exemple **8 % 6**. Ce dernier opérateur force la simplification de la fraction.

Dans un programme de type compilateur ou interprète il est souhaitable parfois d'opérer sur des objets de type «*donnée numérique*» (éventuellement plus riche, comprenant les chaînes de caractères) de façon homogène : une valeur est une valeur, pour l'analyse syntaxique il est souvent redondant de préciser si cette valeur est entière ou réelle. On peut aisément introduire des types «multiples» grâce au balisage. Voici un exemple :

```
data Value = I Integer | F Double | S String | O Operator | Err String
```

où – par exemple – la variante **Err** dénote une donnée illégale, avec le diagnostic d'erreur, et **O** – un objet fonctionnel, désigné ici par l'abbréviation **Operator**.

On aura besoin de données structurales récursives, comme des listes ou des arbres. Voici leur possible implantation/spécification, avec deux types d'arbres binaires – l'un stocke l'information dans les feuilles, et l'autre dans des noeuds intermédiaires :

```
data List a = Nil | Cons a (List a)

data LTree a = Leaf a | Node (LTree a) (LTree a)
data NTree a = Empty | Nd a (NTree a) (NTree a)
```

Bien sûr, le type **List** est redondant, nous avons déjà des listes standard. Les arbres N-aires peuvent être construits à l'aide des listes, par exemple

```
data RTree a = Lf a | RNode [RTree a]
```

Voici le codage d'une procédure de tri arborescent qui utilise les **NTrees**. On place les éléments de la liste triée dans un arbre binaire, en suivant la branche gauche si l'élément est plus petit que la racine, et la branche droite s'il est grand. On répète cette procédure récursivement, jusqu'au niveau des feuilles.

```
tsort l = flatten (instree l Empty) where
  instree [] arb = arb
  instree (x:xq) arb = instree xq (ins x arb)

  ins x Empty = Nd x Empty Empty
  ins x (Nd y g d) | x<=y = Nd y (ins x g) d
                   | otherwise = Nd y g (ins x d)

  flatten Empty = []
  flatten (Nd x g d) = flatten g ++ (x : flatten d)
```

A.5.1 Types-synonymes

Il existe en Haskell deux manières de spécifier un type qui est équivalent à un type existant, mais dont la notation est différente (p. ex. constitue une abbréviation). La forme **type** définit un simple synonyme, p. ex.,

```
type String = [Char]    -- Ceci est prédéfini
type Binop a = a -> a -> a
```

On peut aussi définir un type de données dont la représentation *interne* est *exactement la même* que de données d'un autre type, mais qui est considéré comme vraiment distinct. Si nous déclarons

```
newtype Binop a = Op (a -> a -> a)
```

(notez la présence de la balise **Op**), alors la représentation interne des objets de ce type sera identique aux opérateurs binaires **(a -> a -> a)**, sans aucune perte de mémoire ou de temps pour la reconnaissance de la balise, mais le compilateur traite une telle définition pratiquement comme **data**, c'est-à-dire comme la définition d'un nouveau type distinct.

A.5.2 Introduction à l'inférence automatique des types

Nous avons déjà mentionné le fait que Haskell ne demande pas la présence des déclarations de type (sauf dans des cas spéciaux, discutés ultérieurement). Normalement le compilateur découvre le type d'une fonction assez facilement en regardant sa définition (et parfois son usage). Prenons la définition de la fonction **tsort**, et essayons de dériver son type, et les types de ses fonctions locales.

La fonction **flatten** prend un argument de type **Ntree a** (avec a qui n'est pas spécifié), et qui retourne une liste. Cette liste contient **(x : ...)**, alors c'est la liste des éléments du même type : **[a]**. Donc, la déclaration explicite serait :

```
flatten :: (NTree a) -> [a]
```

De même, il est évident que

```
ins :: a -> NTree a -> NTree a
```

Mais la définition de **instree** est un peu plus délicate. Le premier argument est une liste, mais le second n'est pas spécifié du tout. Le compilateur doit regarder la définition de **ins** pour trouver enfin que

```
instree :: [a] -> NTree a -> NTree a
```

Finalement : **tsort :: [a] -> [a]**. La vérité est plus complexe, le système nous dira :

```
tsort :: Ord a => [a] -> [a]
```

ce qui signifie que la fonction transforme des listes, mais ses éléments doivent obligatoirement appartenir à un type qui admet la comparaison (l'applicabilité de l'opérateur (<)). Ces contraintes seront discutées plus tard.

Parfois les fonctions sont beaucoup plus polymorphes, et leur type est très difficile à découvrir «à l'oeil nu». Prenons la fonction **foldr** dont la définition a la forme déjà bien connue

```
foldr f z []      = z
foldr f z (x:xq)  = f x (foldr f z xq)
```

On peut procéder de manière suivante. la fonction prend trois arguments, et renvoie un résultat, donc le type le plus général sera

```
foldr :: a -> b -> c -> d
```

Mais il y a des contraintes. La première clause établit l'équivalence entre le type du second paramètre et le résultat. Par ailleurs, le troisième paramètre est une liste. Donc, on peut écrire :

```
foldr :: a -> b -> [c] -> b
```

La deuxième clause montre que **`f`** est un opérateur binaire, dont le premier argument est du type **`c`**, et le second est le type du résultat renvoyé par **`foldr`**, ce qui est également le type de son résultat. Finalement

```
foldr :: (c -> b -> b) -> b -> [c] -> b
```

(Les noms actuels des variables **`a`**, **`b`** etc. n'ont pas d'importance). On peut facilement découvrir que

```
map :: (a -> b) -> [a] -> [b]
```

et que le combinateur **`subs`** défini ci-dessous se présente comme

```
subs f g x = f x (g x)
subs :: (a -> b -> c) -> (a -> b) -> a -> c
```

Le type le plus général d'un objet polymorphe, découvert par ce procédé sera appelé son type principal

A.6 Intermezzo : exemples fonctionnels très spécifiques

A.6.1 Combinateurs de Curry

L'importance de cette section sera reconnue un peu plus tard. Son objectif est de montrer comment définir des fonctions en faisant l'*abstraction des paramètres* (dont la forme triviale se réduit à l'équivalence entre la définition **`f x = g x`** et **`f = g`**.). La construction de programmes complexes de nature fonctionnelle peut utiliser en fait un seul mécanisme principal : les compositions de fonctions. Les compositions deviendraient plus simples à maîtriser, à comprendre et à implanter si nous pouvions éliminer au moins partiellement le lest des données inertes – les paramètres présents dans les définitions. Il faut avouer que le résultat d'une «optimisation combinatoire» peut ne pas être très lisible, mais ceci n'est pas grave, car elle peut être une opération interne du compilateur, et son résultat ne sera jamais vu par un humain (sauf si on adore la «folie combinatoire» ; un langage combinatoire concret, Unlambda a été conçu spécialement comme une blague. C'est le plus illisible langage fonctionnel existant).

Un **combinateur** de Curry est une fonction pure dont le rôle est de «coller» d'autres fonctions ensemble, ou de modifier leur comportement de manière universelle. Nous verrons comment les combinateurs facilitent la construction des continuations.

Certains combinateurs, proposés déjà par le mathématicien Haskell Curry, sont considérés standard. Parmi eux nous avons l'identité :

```
id x = x
```

qui dans le langage traditionnel des combinateurs s'appelle **`I`**, mais nous préférons utiliser ici le lexique de Haskell pour pouvoir implanter les combinateurs sans problèmes.

Le combinateur suivant est la «constante» qui ignore son deuxième argument :

```
const x y = x
```

Ensuite il y a le combinateur qui échange les deux arguments d'une fonction :

```
flip f x y = f y x
```

Le combinateur qui duplique un argument :

```
dupl f x = f x x
```

Finalement, parmi les combinateurs très simples il y a le compositeur de fonctions :

```
comp f g x = f (g x)
```

Tous ces combinateurs sont prédéfinis en Haskell ; le dernier est un opérateur infixe : `(.)`. Nous aurons besoin encore d'un autre combinateur de Curry : `subs` considéré standard, et très important.

Voici la définition combinatoire de la fonction qui calcule le carré de son argument : `sqr x = x*x`. Nous réécrivons :

```
sqr x = (*) x x = dupl (*) x
```

et alors `sqr = dupl (*)`. Pas de paramètres ! Bien sûr, la vraie sémantique repose quand même sur la présence d'une fonction concrète, la multiplication. Les combinateurs seuls *ne sont capables de réseoudre aucun problème concret !* Ils constituent à peine un outil de structuration.

Voici le cube exprimé de manière combinatoire : `cube x = x*x*x = (*) x ((*) x x)` :

```
(*) x ((*) x x) = ((*) x) (((*) x) x)
    = comp ((*) x) ((*) x) x = dupl comp ((*) x) x
    = comp (dupl comp) (*) x x
    = dupl (comp (dupl comp) (*)) x
```

et il suffit d'abstraire le `x` pour obtenir une version combinatoire. Le lecteur ne doit pas craindre, des manipulations de ce genre ne sont presque jamais effectuées par les humains (mais elles constituent des jolis sujets d'examen).

Un combinateur très important, appartenant au «canon» de Curry est le «substituteur» :

```
subs f g x = f x (g x)
```

Il peut être réduit aux combinateurs introduits ci-dessus, mais il est préférable de le laisser comme primitif. Il peut générer les autres, par exemple avec `subs` et `const` on peut générer l'identité :

```
(subs const const) x = const x (const x) = x
```

En fait, presque toute imaginable composition fonctionnelle peut se réduire à ces deux combinateurs (dans les ouvrages théoriques ils s'appellent **K** (`const`), et **S** (`const`). Bien sûr, l'identité doit dans la pratique rester primitive, pour des raisons d'efficacité.

Les combinateurs constituent une des techniques de compilation des langages fonctionnels ! Si on arrive à transformer une expression arbitraire à l'enchaînement de combinateurs, ceci permet de définir une machine virtuelle très simple qui réduit cette combinaison au résultat final. Cette machine est très compacte.

***Credo* religieux no. 15 :** Depuis des siècles les théoriciens travaillent pour réduire toute complexité de la Nature aux combinaisons de formes simples. Un jour on trouvera des combinateurs communs pour spécifier le chant de Barbara Hendrics, la choucroute Alsacienne, et le débogueur de Microsoft Windows. Mais ceci probablement ne rendra pas l'Humanité plus heureuse.

A.6.2 Arithmétique de Peano-Church

Nous sommes tous habitués à la présence des nombres dans des programmes. Les nombres sont considérés comme des objets primitifs, internes, irréductibles, et pour des raisons d'efficacité implantés à un niveau très bas. Mais sur le plan formel les langages fonctionnels sont capables de modéliser l'arithmétique complète et minimaliste sans vraiment avoir besoin de connaître la représentation de nombres !

Construirons donc l'arithmétique des nombres entiers qui est basée sur quelques axiomes assez primitifs de Peano – l'existence d'un objet spécifique, le *zéro*, et l'existence de la fonction *successeur* qui permet de passer d'un nombre à . . . son successeur (quelle surprise). Ces objets seront considérés comme **complètement abstraits et opaques**. Aucune fonction *arithmétique* ne doit demander à notre *zéro* sa carte d'identité. Par contre, pour tester le modèle et pour afficher quelques résultats, nous allons concrétiser ces abstractions. **Le deuxième but de cet exercice est de montrer que les constructions récursives dans un langage évolué ne se réduisent pas à de simples appels d'une fonction par elle même !**

Alors, le seul moyen de spécifier un nombre dans ce monde – un ***numéral de Church*** est de le paramétrer par nos abstractions. Si, disons, `nombre` représente un nombre légal, il aura une définition de genre

```
nombre s z = ...
```

où **z** et **s** représentent le *zéro* et le *successeur*.

La première définition concerne le *modèle* de zéro dans notre espace de nombres, le **nombre** zéro. Bien sûr, il ne doit pas dépendre d la fonction *successeur*, et nous pouvons écrire

```
zero s z = z
```
Attention, on changera le nom zero en zer

Le nombre **un** (ou, de préférence **one** pour ne pas mélanger l'Anglais et le Français) est le résultat de l'application du successeur au zéro, donc la définition suivante est légale :

```
one s z = s z
```

et, en fait, nous pouvons «concrétiser» (en faire un objet tangible) un successeur universel, en définissant une fonction Haskell

```
succ n s z = n s (s z)
```

(Rappelons que `succ n s z` peut et doit être lu : `(succ n) s z`, ou, si on préfère :

```
succ n = \s z -> n s (s z)
```

On peut définir d'autres instances concrètes de nombres :

```
two s z = s (s z)
three s z = s (s (s z))
```

etc., mais ceci est un peu ennuyeux. Mieux serait de définir

```
two = succ one
three = succ two
four = succ three
```

etc., mais cette solution n'est pas tellement plus intéressante. . .

Passons donc aux définitions des opérateurs arithmétiques comme l'addition, multiplication et l'exponentiation, mais d'abord préciseront comment tester le système et afficher quelque chose. Il suffit de modéliser le *zéro* et le *successeur* par des objets directement visibles, par exemple

```
zz = 0
ss = (1 +)

shown n = n ss zz
```

mais ce n'est pas la seule possibilité. Nous pouvons utiliser le «système unaire» de manipulation des nombres, comme chez les anciens. Le nombre n sera affiché comme une suite de n étoiles :

```
zzz = ""
sss = ('*' :)
showx n = n sss zzz
```

(Un méta-commentaire est nécessaire. Le mot `zero` dénote un objet prédéfini en Haskell, et pour tester le modèle il est préférable de donner un autre nom au numéral de Church correspondant, p. ex. `zer`. Peut-être un jour les processeurs de notre langage préféré vont accepter des lettres accentuées !)

L'addition est facile à trouver quand on réalise que la sémantique d'un numéral de Church n est d'appliquer n fois son premier argument à son second argument. Ceci implique (ou, au moins, suggère)

```
add n1 n2 s z = n1 s (n2 s z)
```

On applique d'abord **n2** fois le successeur, et ensuite encore **n1** fois, au résultat précédent. Il est trivial de prouver que `add n zer` est égal à **n**, et que `succ (add n1 n2)` $\equiv$ `add n1 (succ n2)`, ce qui complète la démonstration. ***(Essayez de faire cette démonstration formellement).***

À présent nous pouvons définir avec joie : `three = add two one`, `four = add two two`, `five = add two three` etc., mais il est toujours difficile d'aller ainsi très loin. Définissons donc la multiplication des numéraux de Church, et il nous sera utile de reconnaître que si le type du *zéro* abstrait est **a**, alors le type du *successeur* est `a -> a`, et c'est **exactement** le type d'une application partielle `nn succ`, où **nn** est un numéral quelconque, et **succ** – un successeur quelconque.

Alors `nn succ` est un «super-successeur», un successeur appliqué *nn* fois. À quoi? Bien sûr, à l'argument manquant. Alors, pour multiplier **n1** par **n2**, nous allons utiliser **n1** pour appliquer à *zéro* n_1 fois le super-successeur lié à n_2 :

```
mul n1 n2 s z = n1 (n2 s) s z
```

et nous pouvons définir `six = mul two three`, `eighteen = mul three six`, etc. ***Notez que ni l'addition ni la multiplication ne sont pas des fonctions primitivement récursives !*** Néanmoins l'addition est définie par l'itération des successeurs, et la multiplication par l'itération des additions.

A.6.3 Nombres de Peano-Church et combinateurs de Curry

Les définitions des opérations introduites ci-dessus : le successeur (modèle), l'addition et la multiplication peuvent être abrégées grâce au style combinatoire, simplifications des arguments à droite, et l'usage des combinateurs standard comme `flip`, `const` et la composition `(.)`. On voit que

```
zer s z = z ≡ (flip zer) z s = z    ou
zer = flip const
```

Le modèle de 1 est encore plus simple :

```
(one s) z = s z ≡ one s = s    ou
one = id
```

La simplification de l'addition ne va pas si loin :

```
add n1 n2 s z = n1 s (n2 s z) ≡ add n1 n2 s z = ((n1 s) . (n2 s)) z
ou
add n1 n2 s  = n1 s . n2 s
```

et l'élimination de `s` n'est pas très trivial (même si possible. Essayez, pensez au combinateur `subs`...) La simplification du successeur modèle est très simple également :

```
succ n s z = n s (s z) ≡ (n s) . s    ou
succ n s = n s . s
```

mais une forme alternative : `s . n s` est vraie aussi, car nous pouvions partir de la définition

```
succ n s z = s (n s z)
```

Ces formes deviennent combinatoires instantanément, grâce au combinateur `subs`. Cependant, la simplification de la multiplication est étonnante :

```
mul n1 n2 s z = n1 (n2 s) z ≡ (n1 . n2) s z    ou
mul n1 n2 = n1 . n2
```

et dans un langage très soutenu nous dirions que dans le monoïde des fonctions la multiplication est la composition !

A.6.4 L'exponentiation

Pour calculer la puissance n^m il faut itérer m fois la multiplication de n par soi-même. Le niveau d'abstraction monte, et, paradoxalement, la définition de simplifie :

```
pow n m s z = m n s z    ou
pow n m = m n
```

Une dérivation *ab ovo* de cette définition n'est pas très difficile, mais elle ne sera pas donnée ici. Observons seulement que dans la théorie des ensembles la notation A^B dénote **l'ensemble de toutes les fonctions** $B \to A$. Donc, le résultat n'est pas accidentel. Observons aussi que pour les ensembles finis le cardinal de cet ensemble satisfait $|B \to A| = |A|^{|B|}$. Ceci peut être prouvé comme suit. Pour tout élément b de B il existe $|A|$ possibilités de trouver un élément a de A et de former une instance de la fonction $B \to A$, à savoir : $b \to a$. Mais il y a $|B|$ possibilités de choisir b, donc le nombre total d'instances devient $|A|^{|B|}$.

Si le lecteur a toujours des doutes, donnons une preuve inductive de notre résultat. Vérifions que $n^0 = 1, n^1 = n$, et $n^{(m+1)} = n \cdot n^m$. La démonstration utilise la simplification combinatoire sans trop de commentaires :

`pow n zer s` ≡ `zer n s` ≡ `s`

donc, son agissement sur *zéro* donne 1.

`pow n one` ≡ `pow n id` ≡ `id n` ≡ `n`

et finalement

`pow n (succ m)` ≡ `succ m n` ≡ `n . (m n)`

Cet exercice montre clairement quelle est la puissance du raisonnement abstrait dans le domaine aussi concret que l'arithmétique.

A.6.5 Soustraction

Comment décrémenter un numéral de Church? Le problème n'est pas trivial, et sa solution est *forcément* inefficace. Notre «théorie» est close, et nous n'avons aucun moyen de passer au prédécesseur d'un nombre. Mais nous pouvons construire une opération d'incrémentation spécifique qui agit sur des *paires* d'objets, et qui sauvegarde l'original, comme ci-dessous :

```
s2 s (a,b) = (b, s b)
```

En agissant avec ce sucesseur spécial n fois sur `(z,z)` on obtient le le nombre n concrétisé, *avec son prédécesseur*. Voici la définition complète de la décrémentation, avec un peu de simplification de `s2` :

```
dec n s z = p where
  s2 (_,b) = (b,s b)
  (p,q) = n s2 (z,z)
```

Pour la soustraction il suffit d'itérer l'opérateur `dec`. Ainsi on doit reconnaître que `dec (dec (dec ten))` donne un nombre équivalent à `seven`. Laissons les détails de la construction au lecteur. Analysez également sa complexité ; il faut tenir compte que l'opérateur `dec` est onéreux : la forme `(dec n)` lance *l'incrémentation* n fois.

A.7 Exercices

Q1. Optimiser la procédure `tsort` , ou au moins analyser en détail sa complexité, et découvrir toutes les sources d'inefficacité.

R1. Des programmes comme ça sont très fréquents dans la littérature «pédagogique» consacrée à la programmation fonctionnelle. On découvre facilement la structure et le sens de l'algorithme. Mais tel quel, il est très inefficace, même si nous avons déjà effectué une légère optimisation, en construisant la fonction `instree` de manière récursive terminale. (La version encore pire aurait la forme

```
instree [] = Empty
instree (x:xq) = ins x (instree xq)
```

même si syntaxiquement elle est plus simple).

Une optimisation *évidente* consiste à supprimer la concaténation `(++)` du `flatten`. Rappelons que la concaténation recopie son premier argument ! Avec une variable-tampon on élimine cette création de copies éphémères, et de plus on transforme la fornction en itérative :

```
flatten Empty tmp = tmp
flatten (Nd x g d) tmp = flatten g (x : flatten d tmp)
```

(Il faut ajouter le tampon `[]` à l'appel de `flatten` par `tsort`).

Malheureusement la source principale d'inefficacité est ailleurs : Chaque insertion d'un élément dans l'arbre reconstruit complètement la branche concernée, de la racine, jusqu'à la feuille qui sera insérée. Dans un langage impératif on aurait modifié physiquement l'arbre en remplaçant le pointeur sur `Empty` par la référence de la nouvelle feuille, ce qui n'est pas possible directement ici.

Cependant une solution qui construit la structure d'un seul coup existe, mais elle est suffisamment compliquée pour abandonner sa présentation. (Elle utilise la programmation paresseuse de manière assez agressive ; trouver le programme `arbmin` dans le texte).

Q2. Démontrez *formellement* la validité des opérations arithmétiques sur les numéraux de Church, par exemple la commutativité et l'associativité des additions et des multiplications.

R2. Ah, quel beau sujet d'examen.

Q3. Montrez qu'un véritable langage fonctionnel n'a pas besoin de structures de données, elles peuvent être construites comme des fermetures.

R3. Présentons ici un exemple. Essayez d'en trouver autres. Voici une collection de définitions un peu bizarres :

```
cons a b s = s a b

car x = x const
cdr x = x (flip const)
```

Définissons, par exemple `x = cons 17.0 "Belle Marquise"`. La valeur de `x` n'a aucune représentation visuelle, c'est un objet fonctionnel, opaque. Mais l'information stockée à son intérieur est récupérable. Voici la réduction de l'appel

```
car x  →  cons 17.0 "Belle..."  const  →  const 17.0 "Belle..."
       →  17.0
```

Vérifiez que `cdr` marche également, et que structures plus complexes peuvent être composées par `cons`.

Q4. Quels sont les types principaux de fonctions standard (prédéfinies dans le Prélude Standard) ci-dessous :

```
flip f x y      = f y x

until p f x     = if p x then x else until p f (f x)

foldr f z []      = z
foldr f z (x:xs)  = f x (foldr f z xs)
```

R4. L'analyse des types est importante pour le déboguage, alors considérez cet exercice comme important. Et ce qui est *vraiment* important c'est le raisonnement, non pas la réponse.

(a) `flip` prend trois paramètres, et donc son type le plus universel, «amorphe» sera `flip :: a -> b -> c -> d`. Mais le premier paramètre est une fonction qui s'applique aux deux autres, et retourne le résultat qui sera également le résultat du `flip`. Nous aurons donc `f :: c -> b -> d`. Ceci est équivalent à `a`. Finalement : `flip :: (c->b->d)->b->c->d`.

(b) Dans `until` la forme `(p x)` est une condition Booléenne. Sur `x` nous ne savons rien, sauf qu'il est argument de `p` et également de `f`. Mais en regardant le *forme* de la définition, sa cohérence, on découvre que le résultat de `f x` est le même que celui de `x`. Donc : `until :: (a->Bool) -> (a->a) -> a -> a`.

(c) Le résultat retourné par `foldr` est le résultat de `f`, qui est une fonction de deux arguments. Mais son second argument est de même type que son résultat, alors `f :: a -> b -> b`.
Le premier argument de `f`, le `x` appartient à la liste qui est le troisième argument de `foldr`, donc `foldr :: (a->b->b) -> b -> [a] -> b`.

Q5. Quel est le type principal de la fonction `sd` :

```
sd g x l@(y:q) = (g x : l) : map (g y :) (sd g x q)
```

R5. Toujours le même raisonnement. Trois paramètres, dont le premier est une fonction. `sd :: (a->b) -> a -> [c] -> [d]`.

Le résultat `g x` a le même type que `l` (ou `q`). Sur `x` nous ne savons rien, mais le type de `y` est identique. Le résultat final doit être une liste, car il est le second argument du `map`. On trouve facilement les contraintes manquantes : `c` est égal à `b`, et donc aussi à `a`. En regardant l'argument gauche du `(:)` dans le résultat, on découvre la réponse finale `sd :: (a->a) -> a -> [a] -> [[a]]`.

Annexe B

Introduction à la programmation en Haskell (II)

B.1 Surcharge des types

Même des langages qui n'implémentent pas le polymorphisme permettent la surcharge des opérateurs. Par exemple `(+)` en C et en Pascal sert à ajouter les nombres réels et entiers, et le compilateur sait réagir convenablement dans le cas d'une expression mixte : le fait est reconnu, et l'argument entier est converti en réel, soit directement si c'est une constante, ou par la coercition : l'expression `n*x` est (par exemple) compilée comme `toReal(n)*x`. Ceci signifie que la reconnaissance des types est une partie très importante de l'analyse effectuée par le compilateur.

Il ne faut pas confondre la surcharge, appelée parfois le «polymorphisme ad hoc», et le vrai polymorphisme (paramétrique). Dans le dernier cas, une fonction est définie totalement indépendamment du type de son argument, par exemple dans la définition

```
dupl f x = f x x
```

la fonction `dupl` ignore jusqu'au bout le type de `x`, et elle «sait» seulement que `f` est un objet applicable à `x`. un tel polymorphisme n'existe pas en C, il peut être simulé à l'aide des templates.

Par contre, la surcharge des opérateurs arithmétiques peut être considéré comme une *abréviation*, nous avons simplement le même nom pour plusieurs functions. Le compilateur transforme le `(+)` en `addInteger` ou `addReal`, etc. Dans un langage typé statiquement comme C++ c'est presque tout. Il n'y a aucune pénalité d'exécution, car le code compilé contient seulement des fonctions spécifiques (sauf si une fonction est virtuelle, bien sûr).

En Haskell la situation est plus complexe, à cause de l'absence des déclarations du type. Nous pouvons définir une fonction qui «hérite» la surcharge, par exemple la puissance :

```
cube x = x*x*x
```

Si l'utilisateur, après avoir chargé cette définition tape : `cube`, le système va naturellement protester, car il ne sait pas afficher une fonction. Mais, il va dûment répertorier le type de cet objet «`cube`» comme `Integer -> Integer`. Pourquoi `Integer`? Parce que les objets – non pas des fonctions, mais des «choses» sans paramètres – qui existent sur le «top-level» de l'interprète – et c'est exactement ce qui a été compris par Hugs – doivent avoir un type précis. Cependant, si on demande le type de `cube` par la directive `:t cube`, la réponse sera

```
cube :: Num a => a->a
```

lu comme suit : `cube` est une fonction d'un paramètre qui rend le résultat appartenant au même type que le paramètre ***à condition que ce type*** `a` ***appartienne à la classe des nombres*** (où la multiplication est spécifiée comme un opérateur surchargé). `Num` et autres classes seront discutées en détail dans la section suivante, (B.2). Une restriction particulière de Haskell actuel, fait que le compilateur prend une décision inopinée, et considère qu'un argument numérique complètement inconnu soit entier...

B.1.1 Surcharge automatique des constantes numériques

Haskell ne prévoit pas de conversion automatique (ceci en général aurait généré des ambiguïtés), mais il existe une exception. Les constantes numériques entières ou réelles, p. ex. 7 ou 3.1416 ne sont pas compilées directement, mais *forcées* à se comporter comme des données surchargées. Le nombre 7 sera compilé comme `fromInteger 7`, et s'il se trouve dans un contexte flottant, p. ex. si on tape : `7*2.5`, on obtient 17.5 sans problèmes.

Les constantes réelles `X` se transforment en `fromDouble X`. Ceci permet en Haskell existence de l'arithmétique mixte, sans forcer l'utilisateur à placer explicitement les fonctions de conversion.

Il faut avouer que cette méthode n'est pas très efficace, la conversion automatique en C ou Fortran est plus rapide. Cependant, ces langages ne sont pas polymorphes, et demandent toujours des déclarations explicites, donc il y a toujours un prix a payer.

B.2 Classes de types

Haskell réalise – à sa manière – un de paradigmes de la programmation orientée-objet, à savoir une surcharge dynamique, la possibilité de compiler des fonctions en (une certaine) indépendance des types des arguments. Qu'est-ce le symbole «`Num`» découvert dans la section précédente? Il dénote le nom d'une **classe**.

Dans des langages à objets connus, comme C++ (ou Smalltalk) il y a une «collusion» entre les notions de ***classe*** et ***type***. En Haskell ces deux notions sont très différentes. Haskell opère avec des ***classes de types***. Un type et jamais une donnée, peut appartenir à une classe. Un type peut d'ailleurs appartenir à plusieurs classes (et ceci n'a presque rien à voir avec l'héritage multiple classique ; on peut cependant trouver une affinité entre cette propriété, et plusieurs «interfaces» d'un objet en Java).

Par exemple, les entiers et les flottants (et les rationnels, et les complexes, et tout ce qui vous pouvez définir dans un cadre arithmétique) appartiennennt à la classe des «nombres» `Num`. La classe précise quelles sont les «fonctions virtuelles» susceptibles à agir sur les objets de type qui appartient à la classe donnée. Si nous voulons définir une méthode surchargée, p. ex. `lg`, applicable aux entiers et aux chaînes, et qui returne soit la longueur de la chaîne, soit l'entier lui même, nous écrirons :

```
class Privée a where
  lg :: a -> Integer
```

et ensuite nous précisons les implantations concrètes, définies dans les *instances* de la classe définie :

```
instance Privée Integer where
  lg x = x
instance Privée Int where
  lg x = toInteger x
instance Privée [a] where
  lg x = lng x where
          lng [] = 0
          lng (_:q) = 1+lng q
```

Bien sûr, l'exemple n'est pas très sérieux, normalement les fonctions surchargées qui portent le même nom doivent faire des choses similaires.

Dans le Prélude standard nous trouverons :

```
class (Eq a, Show a) => Num a where
  (+), (-), (*)  :: a -> a -> a
  negate         :: a -> a
  abs, signum    :: a -> a
  fromInteger    :: Integer -> a
  fromInt        :: Int -> a
```

où on note la clause `(Eq a, Show a) =>` qui préfixe la déclaration de la classe `Num a`. Elle signifie que le type générique `a` peut appartenir à la class `Num`, à condition, qu'il appartienne déjà aux classes `Show` (l'existence des fonctions d'affichage ; c'est un accident historique cette restriction, elle n'a pas beaucoup de sens mathématique) et `Eq`, où on précise la fonction surchargée `(==)` – l'égalité entre deux structures de données quelconques.

La classe `Eq` est simple, mais intéressante :

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  x == y   = not (x/=y)
  x /= y   = not (x==y)
```

On y trouve pas seulement la déclaration de l'égalité et d'inégalité : **(/=)**, mais aussi deux définitions «concrètes» de ces opérateurs, qui visiblement ne peuvent servir à rien, car elles sont cycliques…

Mais elles *peuvent* être utiles. Si nous définissons une nouvelle structure de données, par exemple les nombres complexes définis comme

```
data Complex = C Double Double
```

il suffit de préciser l'égalité :

```
instance Eq Complex where
  C x y == C a b = x==a && y==b
```

et l'inégalité est définie *par défaut* en accord avec les définitions *dans la classe* (plutôt que dans l'instance). Ainsi, dans la classe **Num** nous retrouverons les définitions suivantes :

```
x - y           = x + negate y
negate x        = 0 - x
```

ce qui nous permet de définir soit la négation, soit la soustraction, et l'autre nous est fournie automatiquement. Nous pouvons donc définir

```
instance Num Complex where
  C x y + C a b = C (x+a) (y+b)
  C x y * C a b = C (x*a-y*b) (x*b+y*a)
```

et la soustraction est définie aussi.

B.2.1 Restrictions sur les types

Définissons une autre structure de données, une «fraction rationnelle générique»

```
data Fract a = R a a
```

où – intuitivement – nous voudrions que **R 7 9** représente la fraction $7/9$. Mais nous n'avons pas spécifié le type des fractions comme **R Integer Integer**, car nous pouvons un jour essayer de construire des fractions à partir des polynômes, sachant que les polynômes admettent l'arithmétique similaire à celle des entiers (addition, multiplication, division Euclidéenne, PGCD, etc.). Donc, le type des composants reste générique. Alors la définition des instances aura la forme suivante :

```
instance Num a => Num (Fract a) where
  R x y + R a b = simplifie (R (x*b+y*a) (y*b))
```

La clause **Num a =>** est nécessaire pour informer le compilateur que l'on ne peut assembler les fractions à partir de n'importe quoi. Le numérateur et le dénominateur doivent être d'un type compatible avec la classe **Num**.

Lisez le Prélude Standard de Hugs. Il contient plusieurs dizaines d'exemples de classes et d'instances qui peuvent vous inspirer.

B.2.2 Classes de constructeurs

Le système de types dans un langage fonctionnel inclut naturellement les types fonctionnels. Si l'expression **f a b** appartient au type **Double**, et si la variable **b** est **Integer**, l'expression **f a** appartiendra au type **Integer->Double**. Ceci nous savons déjà.

Mais ainsi les expressions **C** ou **R** qui sont les **constructeurs** des types **Complex** ou **Fract** appartiennent aussi aux types spécifiques :

```
C :: Double -> Double -> Complex
R :: a -> a -> Fract a
```

(on peut les considérer comme des applications partielles) et nous pouvons nous poser la question : a-t-il un sens préciser des classes *et donc, des opérations surchargées* pour ces constructeurs? En effet, il existe une raison. On trouve parfois des opérations similaires au sens qu'elles font «des choses pareilles» pas seulement indépendamment de type des arguments, mais également ***de leur structuration***. Avec le type **`Complex`** il n'y a rien à faire, mais si le type est paramétré, comme **`Fract`** – si. Nous pouvons préciser que **`Fract`** tout court appartienne à une **classe de constructeurs**.

Autre exemple. Nous connaissons la fonction **`map`** :

```
map f [] = []
map f (x:xq) = f x : map f xq
```

qui applique une fonction à tous les éléments d'une liste. Mais nous pouvons vouloir appliquer une fonctions à tous les éléments d'un tableau ou d'un arbre, par exemple

```
data Arbre a = Nil | Noeud a (Arbre a) (Arbre a)

tmap f Nil = Nil
tmap f (Noeud x g d) = Noeud (f x) (tmap f g) (tmap f d)
```

et il serait utile de pouvoir appeler cette fonction «**`map`**» aussi. Ceci est possible en Haskell (avec une légère modification du nom, une telle fonction est prdéfinie, et s'appelle **`fmap`**). D'abord on définit la classe :

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

Notez que le type – argument de la classe, **`f`** figure dans un contexte **`(f a)`** dans le type de **`fmap`**. Ceci signifie que **`f`** est un *constructeur de types*, un moyen de structurer des types composites. Nous pouvons maintenant déclarer :

```
instance Functor Arbre where
  fmap = tmap
```

et nous trouvons dans le Prélude la déclaration

```
instance Functor [] where
  fmap = map
```

Ici la forme **`[]`** *n'a rien à voir avec une liste vide, la notation est un peu accidentelle !* Ceci est un *constructeur de listes*, le type qui aurait pu s'appeler par exemple «**`List`**». Le nom **`[]`** est historique, il faut s'habituer, et de ne pas le confondre avec la *donnée* **`[]`**....

Nous pouvons construire d'autres instances de cette fonctionnelle. Par exemple il est utile de prévoir un type «peut-être» qui spécifie (avec un balisage) les objets résultants de l'appel d'une fonction qui peut «échouer», et rendre «rien». Dans le Prélude nous trouverons :

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord, Read, Show)
```

(Pour l'instant acceptez l'existence de la clause **`deriving`**. Elle est très utile et elle sera discutée ultérieurement.) Si une fonction «normale» retourne **`x`**, une fonction étendue, qui peut échouer, retourne **`Just x`** (en cas de succès).

Si nous voulons qu'une fonction appliquée par le **`map`** généralisée à **`Nothing`** ne fait rien, et se comporte «normalement» dans le cas contraire, nous définissons :

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

Pour le parsing nous aurons besoin d'une autre classe de constructeurs, la classe **`Monad`**, qui sera décrite plus tard.

B.2.3 Fonctions d'affichage

Tout langage de programmation doit permettre la lecture des données et l'affichage des résultats. Les langages fonctionnels évitent les effets de bord, traditionnellement associés aux procédures d'entrée et de sortie. Dans cette section nous allons traiter seulement une partie du problème : la conversion entre des objets quelconques et leur représentation extérieure, visuelle, concrètement : les chaînes de caractères.

En général le problème est difficile : comment afficher un objet quelconque? La procédure **`printf`** en C utilise des primitives qui reconstruisent les chaînes depuis des entiers ou des flottants, mais dans les cas plus complexes, l'utilisateur doit savoir ce qu'il veut. La situation en Haskell est similaire.

Il existe une fonction universelle, surchargée **`show`** qui transforme son argument en chaîne. Une autre fonction : **`shows x s`**, «affiche» (transforme en chaîne) l'argument **`x`**, et concatène le résultat avec la chaîne **`s`**, ce qui permet d'enchaîner l'affichage de manière plus efficace. Au lieu d'écrire **`show x ++ show y ++ show z`**, on peut écrire **`shows x (shows y (shows z ""))`**, ou **`(shows x . shows y . shows z) ""`** – construire des *combinateurs d'affichage* efficaces qui évitent de recopier les chaînes.

Quelques fonctions de base comme **`show`**, **`showList`**, et **`showsPrec`** sont définies comme membres de la classe **`Show`**, et les instances de **`Show`** pour les caractères, et les nombres sont prédéfinies. La fonction **`showsPrec`** est capable de gérerl'affichage de structures hiérarchiques, avec quelques règles de précédences et avec l'insertion de parenthèses. Le lecteur doit lire les définitions dans le Prélude.

La fonction **`showsPrec`** pour des types nouveaux doit être définie par l'utilisateur. Il faut déclarer au compilateur de Haskell que le type de données soit «présentable» (qu'il appartient à **`Show`**, ainsi il profitera pleinement du polymorphisme des opérations d'affichage.

Il existe également la classe **`Read`**, qui spécifie les fonctions de conversion entre les chaînes et des objets quelconques. En fait la fonction **`read`** constitue un parseur primitif integré dans Haskell. Il est suffisamment général pour être pratiquement utilisé comme un scanneur lexical. Encore une fois, le lecteur est invité à lire la documentation du langage.

B.3 Modules

Les langages respectables d'aujourd'hui doivent permettre la séparation de la source d'une grande application en morceaux : **modules**, qui communiquent, et peuvent profiter des définitions présentes ailleurs, mais qui gardent une certaine intégrité, et – par exemple – la possibilité de définir des fonctions privées, invisibles de l'extérieur, ce qui évite la collision des noms. Bien sûr, pour langage compilé vers un code binaire avec la génération des applications autonômes, le système doit assurer le traitement indépendant des modules. Chaque module qui veut utiliser des fonctions, constantes ou types définis ailleurs, doit les explicitement *importer*. Chaque module qui veut rendre visible une partie de ses définitions, les *exporte*.

Haskell demande que tout fichier contenant une librairie-source d'entités exportables (fonctions, types…) commence par le mot-clé **`module`** suivi par le nom et la liste de modules/entités exportés, par exemple

```
module Prelude (
   map, (++), concat, filter, head, last, tail, init, null,
   length, (!!), foldl, foldl1, scanl, scanl1, foldr, foldr1,
   scanr, scanr1, iterate, repeat, replicate, cycle,
   ...
   )
```

Si la liste est vide, par défaut cela signifie que *tous les noms* (et leurs définitions) sont exportés. (Un module qui n'exporte rien ne sert à rien). Par défaut un module ne ré-exporte pas des entités qu'il a importé.

Ensuite nous avons la liste d'importation et les définitions locales. On peut importer un module partiellement, précisant par exemple que quelques définitions sont «cachées». Voici l'entête d'un tel module (en fait c'est un module utilitaire réel écrit pour gérer differemment les nombres ; toute la classe **`Num`** disparaît et les opérateurs arithmétiques sont rédéfinis) :

```
module Math  where
import Prelude as Pp hiding ((+),(-),(*),Num,(/),(^),Fractional,
fromDouble,negate,abs,signum,recip,fromInt,fromInteger,Floating,
sin,cos,exp,log
...
)
```

```
infixr 8 ^
infixl 7  *, /, #, >/
...
```

En cas de besoin on peut toujours accéder à un objet **xxx** appartenant au Prélude par la notation **Pp.xxx**.

Le nom du module doit correspondre au nom du fichier dans lequel il se trouve. Les modules peuvent être réciproquement récursifs.

Nous ne pouvons pas donner l'information complète sur les modules. Il est possible de changer de nom d'entités importés ou de «qualifier» quelques noms à l'aide du mot-clé **qualified**, ce qui permet de gérer mieux les espaces contenant les noms conflictuels.

Il est possible d'exporter partiellement une structure de données : seulement son nom (et les fonctions qui manipulent ces données), mais sans constructeurs. Ainsi, dans le module qui importe cela, ce type de données devient *abstrait*. On ne connaît pas sa structure, on ne peut pas «manuellement» construire ses instances, seulement utiliser les fonctions importées. Ceci augmente la sécurité de programmation.

Ainsi les fractions rationnelles définies dans le Prélude à l'aide de l'opérateur **(:%)** : **7:%9** est la fraction $7/9$ – peuvent être manipulées, mais l'utilisateur n'a pas le droit de construire, disons, **4:%8** ou **3:%(-7)**, car de telles fractions sont mal simplifiées. Le constructeur **(:%)** n'est pas exporté, par contre, le Prélude exporte un autre opérateur, **%** qui force toujours la simplification de la fraction, qui vérifie que le dénominateur est différent de zéro, etc.

B.3.1 Clause `deriving`

Pour quelques procédures surchargées relativement universelles comme l'affichage, l'égalité, ou l'ordre, Haskell permet d'ajouter un peu d'automatisme, ce qui évite un travail pénible de définition de toutes les instances. Si nous avons une structure de donnée composite, par exemple

```
data Value a = Rien | D Double a | A (Value a) | T (Value a) (Value a)
```

on peut attendre que l'égalité soit définié «naturellement» : **Rien==Rien** et à rien d'autre, que **A x==Ay** si **x==y**, etc. Les structures comparées doivent être homologues. La clause

```
data ...              deriving Eq
```

demande à compilateur la construction des fonctions d'égalité et inégalité correspondantes.

On peut aussi dériver **Show**, **Ord**, et **Read**. L'affichage dérivé utilise les noms des constructeurs. Ceci peut «dévoiler» les constructeurs que nous avons voulu cacher dans un module !

La classe **Read** est la réalisation d'un petit parseur dans la couche standard de Haskell : elle permet la lecture de structures de données définies par l'utilisateur.

B.4 Continuations: du fonctionnel à l'impératif

Le lecteur peut – à juste titre – avoir l'impression, que si on élimine des langages impératifs l'évaluation des expressions, ce qui constitue la couche fonctionnelle, et si on reste avec le flux de contrôle, les branchements, les boucles, etc., ceci n'a plus rien à voir avec le monde fonctionnel, et que les techniques de compilation deviennent très asymétriques : un programme impératif est ***réel***, correspond au code assembleur exécuté par le processeur, tandis que le code fonctionnel, la réduction et l'évaluation d'un graphe qui représente une expression, force sa ***translation*** en code linéaire, impératif.

Cependant le progrès récent dans le domaine de compilation et l'arrivée de nouveaux compilateurs est partiellement le résultat du progrès dans la compilation *fonctionnelle* qui est beaucoup plus statique, mathématiquement précise, et facile à comprendre.

Nous avons déjà mentionné le fait qu'une fonction paresseuse peut réaliser une structure de contrôle, un module de code qui pilote localement l'évaluation d'une ou plusieurs sous-expressions, et ainsi permet de prendre des décisions sélectives (on n'évalue pas la clause **else** si la condition **if** était vraie et la clause **then** reste la seule en vigueur ; pour la structure **case** une seule branche peut et doit être évaluée).

Or, la sérialisation du code, la notion de séquence, possède sa forme fonctionnelle aussi, et s'appelle la **continuation**. Brièvement, la continuation d'une expression (rappelons que dans le monde fonctionnel l'espression c'est le *seul* objet intéressant) est le «futur» du calcul, ou l'opération qui sera exécutée immédiatement après. dans l'expression $f(g(x), h(y, x))$ qui en Haskell sera écrite comme

```
f (g x) (h y x)
```

La continuation de l'expression $g(x)$ est l'évaluation de $h(y, x)$ (si l'évaluation des arguments procède de gauche vers la droite ; en Scheme cet ordre n'est pas spécifié) car il faut évaluer le second argument de **f**, et finalement la continuation de h est f. Ainsi le code sera linéarisé.

La réalisation concrète et détaillée des continuations par le compilateur sera discutée ultérieurement. Ici il suffit de préciser qu'une telle opération peut être, et *est* automatique, et suggérer l'approche suivante : *toute* fonction est modifiée, et ***prend un paramètre supplémentaire***. Ce paramètre est justement la continuation, une fonction d'un argument. Si la fonction originale retournait simplement une valeur, la fonction «continuée» passe cette valeur à sa continuation. Plus concrètement, au lieu de discuter l'évaluation de **(f x)**, nous allons transmuter **f** en une «fonction continuée» **f_cont**, telle, que

```
f_cont x cnt = cnt (f x)
```

où la fonction **cnt** représente la continuation, le futur d'évaluation de **(f x)**. On peut abstraire «la continuation d'une fonction normale» en définissant :

```
f_cont = clift1 f

clift1 f x cnt = cnt (f x)
```

Nous affirmons que

- le processus de construction de fonctions «continuées» : transformation dite CPS : *continuation passing style*, peut être automatisé dans la plupart de cas, et
- le code résultant ressemble plutôt à l'assembleur qu'à l'arborescence représentant une expression composite, hiérarchique.
- Son optimisation est beaucoup plus facile.
- Grâce aux continuations on pourra de manière fonctionnelle, **et alors simple, lisible et statique** définir les branchements, structures itératives, voire même des structures de contrôle non-déterministes.

Prenons comme exemple l'évaluation de l'expression $\sqrt{x^2 + y^2}$. Une définition fonctionnelle classique d'une fonction qui effectue ce calcul serait

```
fn x y = sqrt (x*x + y*y)
```

Rien à ajouter, même si nous pouvons manuellement convertir ce code en postfixe. Supposons néanmoins que la fonction **sqrt** a été transmutée par **clift1** en sa forme continuée : **sqc = clift1 sqrt**, et que nous avons défini les opérateurs binaires continués également :

```
clift2 op x y cnt = cnt (op x y)
add = clift2 (+)
mul = clift2 (*)
```

La fonction continuée **fnc** peut être définie comme

```
fnc x y cnt =
  mul x x (\a ->
  mul y y (\b ->
  add a b (\c ->
  sqc c cnt)))
```

La continuation de la première multiplication est le calcul de **y*y**. Cette opération n'a pas besoin du résultat **a** de la multiplication précédente, mais on le garde pour l'avenir. Il sera utilisé par **add**. On doit noter une ressemblance entre l'exemple ci-dessus et les instructions impératives :

```
a = x*x
b = y*y
c = a+b
resultat = sqrt c
```

En fait, les continuations permettent d'établir un pont entre la programmation fonctionnelle et une machine à registres. Grâce à elles on peut établir l'ordre d'exécution des opérations (évaluations) dans le programme, *sans imposer cet ordre au niveau du méta-langage*. Nous définissons un langage de manière dénotationnelle, précisons sa syntaxe et sa sémantique *statiquement*, sans jamais aborder le problème du «temps». Mais la construction des relations «X est la continuation de Y» permet d'enchaîner l'exécution des instructions, et ainsi nous pouvons créer (partiellement) un langage impératif sans quitter le style fonctionnel.

Les continuations peuvent naturellement être utilisées dans un programme quelconque. Tout enchaînement d'applications fonctionnelles dans un module peut exploiter cette stratégie. Mais attention : si une expression normale se transforme en «continuée», c'est-à-dire en fonction qui attend un argument – la continuation, et si cette continuation est une fonction continuée comme **`mul`**, le résultat est encore une fois une expression continuée, un objet fonctionnel. Quand est'ce que le résultat *final* sera enfin récupéré?

La réponse est : à la fin du programme (module ou son fragment) ainsi sérialisé. Il faudra appliquer une **continuation terminale**, par exemple la fonction **`id`**, définie par **`id x = x`**, qui récupère le résultat. Naturellement cet appel de la fonction **`id`** possède aussi une continuation, mais cette continuation **pour le programmeur** est implicite : elle peut appartenir à la boucle principale de l'interprète (le dialogue avec l'utilisateur), où le résultat est affiché. Ou bien, ceci peut être la fin logique du programme qui s'arrête, et sa continuation est une des procédures du système d'exploitation, par exemple le *shell* qui fait avec le résultat ce qu'il veut. En tout cas la chaîne de continuations explicites doit être «cassée» pour voir le résultat (p. ex. numérique), sinon *tous* les objets crées sont des fonctions.

Ceci n'est pas notre dernière rencontre avec les continuations (sauf si le lecteur lit cet annexe après avoir assimilé le reste de ces notes).

B.5 Les tableaux

Tout langage sérieux doit permettre la construction de structures composites adressables rapidement : les vecteurs, même si souvent les listes sont plus commodes pour coder les algorithmes récursifs. Pour pouvoir utiliser les tableaux en Hugs il faut préfixer le fichier contenant le programme par **`import Array`**.

Les tableaux en Haskell sont un peu plus abstraits que dans d'autres langages, et leur usage demande une certaine expérience. On ne peut pas facilement modifier un élément d'un tableau, et ceci souvent décourage les débutants. Haskell définit un type générique **`Array a b`** où **`a`** est le type des indices, et **`b`** – le type des valeurs stockées dans le tableau. Les indices peuvent être des entiers, caractères, un type énuméré, etc., – tout type qui appartient à la classe **`Ix`** (qui ne sera pas discutée ici). ***Important :*** les paires (n, m) sont des indices légaux, si n et m sont des indices. Ainsi on peut construir des tableaux multi-dimensionnels.

Un tableau normalement est construit d'un coup, à partir d'une liste d'associations par la fonction **`array :: (a,a) -> [(a,b)] -> Array a b`** : , par exemple

```
v = array (1,100) ((1,1) : [(i, i * a!(i-1)) | i <- [2..100]])
```

ce qui montre l'usage des compréhensions et de la paresse pour construire des tableaux da façon incrémentale. Bien sûr, à droite, dans la liste des association chaque indice doit apparaître une et une seule fois. On voit aussi que l'opérateur **`(!)`** est utilisé pour indéxer les éléments. La fonction **`listArray`** demande uniquement l'intervalle des indices et une liste des éléments, et construit les associations elle-même.

L'absence d'une «instruction» de genre **`A[k]:=A[k]+1`** etc. devient moins gênente si on apprend quelques astuces du métier, p. ex. l'usage des tableaux cumulatifs. Un tableau en Haskell est stocké avec son paramétrage, en utilisant les fonctions comme **`bounds`**, **`indices`**, **`elems`** ou **`assocs`** on peut récupérer l'intervale des indices, ou les listes : des indices, des éléments, ou des associations, et de reconstruire avec ces informations un autre tableau.

L'opérateur infixe **`(//) :: Array a b -> [(a,b)] -> Array a b`** prend un tableau et une liste des associations, et construit un autre tableau, avec les éléments modifiés selon le second argument.

Comme il a été mentionné ci-dessus, la liste des associations qui définit les éléments du tableau ne doit pas avoir des indices répétés. Mais on peut relaxer cette contrainte. Il faudra donc répondre à la question : qu'est-ce que l'on met dans l'élément concerné, dont l'indice figure plusieurs fois? La réponse est : on *combine* ces éléments en utilisant la ***fonction d'accumulation***.

La fonction **`accumArray :: (b->c->b) -> b -> (a,a) -> [(a,c)] -> Array a b`** combine (p. ex. ajoute) toutes les contributions venant des indices répétés, en utilisant l'opérateur – le premier

argument de **accumArray**. Le second argument est la valeur initiale utilisée dans la combinaison. Cette fonction utilise un opérateur plus simple **accum** qui est une sorte de **fold** pour les tableaux. En Haskell nous avons aussi des fonctionnelles **map** et **ixmap** applicables aux tableaux, mais les détails doivent être cherchés dans la documentation.

B.6 Exercices

Q1. Quelle est la différence entre ces deux définitions :

```
cube x = x*x*x
cub = \x -> x*x*x
```

R1. **cub** a été défini sans paramètres, comme un objet lambda. La restriction monomorphique du Haskell précise que **cub** est une fonction du type **Integer->Integer**. Le type de **cube** a été discuté en détails : **Num a => a -> a**. Mais si on demande le type de **\x->x*x*x** on obtient de nouveau **Num a => a -> a**. Pourquoi?

Q2. Quelle est la réponse du système si après avoir chargé la définition de la classe **Privée** et de ses instances, on demande **lg 67**.

R2. Vérifiez, et analysez la réponse.

Q3. Définir une «file abstraite», une entité qui se comporte comme une file, avec les procédures typiques : ajouter un élément, enlever un élément (en retournant l'élément et la file restante), et la vérification si la file est vide. La file doit être générique (ses éléments sont de type quelconque), et la définition se trouve dans un module qui cache son implantation, p. ex. par une paire de listes, comme discuté dans la section (4.4).

R3. Ceci apparemment n'a rien de particulièrement intéressant, mais regardez la définition de l'égalité. Elle est mauvaise ! Si pour implanter les files on utilise une liste double, la paire **([a,b,c],[d,e])** est équivalente à **([a,b],[d,e,c])**, et à **([],[d,e,c,b,a])**etc.

```
module Queuedef(Queue, qadd, qinit, qvide, qdel)
where

data Queue a = Q [a] [a] deriving (Eq, Show)

qinit () = Q [] []

qvide (Q [] []) = True
qvide _ = False

qadd x (Q a b) = Q (x:a) b

qdel (Q a (x:b)) = (x,Q a b)
qdel (Q [] _) = error "File vide"
qdel (Q l _) = qdel (Q [] (reverse l))
```

La construction de la vraie opération d'égalité est un joli sujet d'examen.

Q4. La gestion des arbres est bien adapté au codage fonctionnel, les algorithmes typiques sont récursifs. Malheureusement parfois il faut parcourir plusieurs fois la même structure, ce qui n'est jamais très efficace. Un exemple de ceci est le suivant. Comment *parcourant un arbre binaire une seule fois* construire un autre arbre binaire dont la structure est identique, mais où les valeurs stockées sur les feuilles sont replacées toutes par le minimum des valeurs présentes dans l'arbre original.

Apparemment ceci est impossible. Il faut parcourir l'arbre une fois pour récupérer le minimum, et ensuite lancer le **map** généralisé pour reconstruire la réponse structurellement.

(Cet exercice est plutôt une devinette, car les chances que les lecteurs trouvent *seuls* la réponse, sont plutôt dérisoires, même si l'auteur de ces notes rêve qu'un jour *un* étudiant aura la volonté de lire la littérature consacrée aux techniques fonctionnelles. . .)

R4. La réponse existe grâce à la programmation paresseuse, qui permet de créer des programmes «circulaires», avec des références réciproques (croisées) des données. Les références croisées des fonctions n'ont rien d'inhabituel, il s'agit de la récursivité indirecte. Mais si une donnée A a besoin de B et vice-versa, la situation semble inextricable. Pourtant, la solution ci-dessous marche :

```
data Arb a = Feuille a | Noeud  (Arb a) (Arb a)

arbmin arbre = res where
 (res,x) = abm arbre x            -- dépendance circulaire. Qu'est'ce que le "x"?
abm (Feuille y) x = (Feuille x,y)
abm (Noeud ga dr) x = (Noeud tg td, min xg xd) where
  (tg,xg) = abm ga x
  (td,xd) = abm dr x
```

La fonction auxiliaire **`abm`** simultanément calcule le minimum (le second membre du tuple qui deviendra le «**`x`**»), et construit l'arbre, en propageant **`x`** jusqu'au niveau des feuilles. Ensuite ce **`x`** est inseré. Mais dans la programmation paresseuse l'arbre n'est pas vraiment construit physiquement, le résultat **`res`** contient le générateur de cet arbre, un *thunk* qui contient la référence à **`x`**. Quand nous **demandons** la valeur de **`res`** (p. ex. son affichage), c'est à ce moment-là que le *thunk* est exécuté, et l'arbre est formé, avec **`x`** correct.

Cet algorithme, publié par Richard Bird, est un exemple canonique de l'usage de la programmation paresseuse pour éviter de traverser plusieurs fois la même structure. Mais il ne faut pas avoir des illusions : en termes d'usage de mémoire cet algorithme est très onéreux (prolifération des *thunks* ; ceci introduit également une visible surcharge temporelle).

Q5. Une fonction produit une liste très longue de nombres apparemment aléatoires entre 0 et 1. Écrire un programme qui calcule l'histogramme de cette liste : un tableau a de 100 éléments dont chaque élément $a!k$ contient le nombre d'occurrences des nombres x appartenant à l'intervalle entre $k/100$ et $(k+1)/100$

R5. Par exemple :

```
x0=0.17
f x = 4.0*x*(1.0-x)
lst = take 20000 (iterate f x0)

a = accumArray (+) 0 (0,99) [(k,1)|k<-[floor(100.0*x)|x<-lst]]
```

Annexe C

Intermezzo monadique

C.1 Introduction

Cette section est une digression de nature un peu plus générale, qui touche quelques problèmes universels de la sémantique des langages de programmation. Son but immédiat est d'expliquer encore une fois l'opérateur **`(>>=)`**, et d'établir un pont entre lui, les continuations, et le non-déterminisme, mais elle est **beaucoup plus générale et plus importante.** Peut-être cette section est la plus importante de tout le cours, pour la culture générale d'un futur informaticien, indépendamment de la compilation elle même. (c'est pourquoi nous l'avons mis dans les annexes). Elle référence la section consacrée aux parseurs combinatoires ; en particulier l'opérateur *bind* **`(>>=)`** est considéré déjà un peu famillier.

Les stratégies de composition fonctionnelle à l'aide de l'opérateur **`(>>=)`** sont des exemples des **monades**. Leur discussion sera très superficielle, mais ce concept est une petite révolution paradigmatique dans le monde de programmation fonctionnelle, et il a influencée énormément le domaine de la programmation logique, et les stratégies de compilation. Selon l'opinion personnelle de l'auteur de ces notes, les personnes qui ignorent les monades n'ont pas le droit d'affirmer qu'elles connaissent (ou qu'elles enseignent) la programmation ***fonctionnelle***.

Superficiellement la programmation fonctionnelle est très proche des mathématiques : on a quelques objets formels (les valeurs appartenant à des types spécifiques), et on en construit d'autres, en appliquant des fonctions. Mais un ordinateur est une créature moins mathématique. Il **fait** quelque chose, et nous voulons à présent formaliser ce «travail».

Introduisons donc un concept très général, celui de l'*activité*, ou – si on le veut – du *programme* abstrait. En anglais on emploie le mot "*computation*" dans ce contexte, et nous pouvons essayer d'utiliser le mot «calcul». Une **monade** est un constructeur de types de données, qui transforme une *valeur* quelconque en calcul. Le cas le plus simple des monades est la Monade Identité, qui *ne fait rien* avec la valeur, qui la retourne telle quelle. Une *donnée quelconque*, ou plutôt un *type* (c'est à dire un ensemble de données) quelconque appartient à ce monde monadique trivial.

Si le lecteur ne sait toujours pas du tout ce que c'est une monade (dans le contexte trivial), il voudra lire le *Bourgeois gentilhomme* de Molière, et en particulier la discussion du concept de *prose*.

Oui, la programmation fonctionnelle est prosaïque... Un programme fonctionnel est une fonction qui s'applique à une valeur (elle peut être multiple), et qui est formée par la composition, ou l'enchaînement des fonctions plus simples. Les fonctions s'enchaînent comme d'habitude, $x \to f(x) \to g(f(x)) \equiv (f.g)(x) \to \ldots$. Il n'y a *rien* d'autre. Même les structures de contrôle, type **if-then-else** conceptuellement sont des fonctions[1], mais paresseuses, comme il a été déjà dit :

```
ifThenElse True  oui _   = oui
ifThenElse False _   non = non
```

Pour finaliser la partie triviale de l'exposé du monde monadique :

- Un calcul est une valeur. Une valeur est un calcul.

[1] en Clean **if** *est implémenté comme une fonction*, ce qui implique que l'usage des «gardes» est plus efficace !

- Donc, la création d'un calcul à partir d'une valeur, est une fonction «normale», qui transforme valeurs en valeurs. Le seul «travail» est l'***application fonctionnelle***. Donc –
- Le manipulateur qui permet d'enchaîner les calculs est cette l'application fonctionnelle («normale»), et les compositions des fonctions permettent de présenter l'enchaînement des applications de manière un peu plus abstraite : $f(g(x)) \rightarrow (f.g)(x)$.

C.1.1 Et les monades moins triviales?

En voici une liste qui est loin d'épuiser le sujet :

1. Un calcul peut déclencher des exceptions (ou erreurs), par exemple on peut diviser par zéro. Qu'est-ce passe-t-il alors? En tout cas, si le «résultat» d'un tel calcul est l'argument d'une autre fonction arithmétique, celle-ci ne peut faire rien de raisonnable, elle peut éventuellement propager l'erreur.

   On peut dire : cette fonction ne sera jamais appelée, car «le système» quand il découvre une erreur il abandonne l'expression arithmétique, et branche sur un chemin de récupération. Mais comment? Pour un constructeur de compilateurs il n'ya pas da magie, nous *devons* pouvoir gérér un tel contexte nous mêmes. Un calcul sera ici une expression «conditionnelle», soit «normale», soit «anormale», exceptionnelle (ne pas confondre avec la conditionnelle standard !).

2. Nous voulons tracer l'exécution, forcer le compilateur à ajouter à une expression un message diagnostique qui contient le nom de fonction appelée, les arguments et le résultat. Le calcul doit combiner les valeurs et les messages.

3. Nous voulons faire le *benchmarking* du programme. Pour chaque opérateur appelé, un compteur global est incrémenté, et à la fin nous pouvons savoir combien d'opérations ont eu lieu. Le calcul est la valeur combinée avec ***l'opération*** de changement d'état (incrémentation d'un compteur).

   Cette monade – la transformation d'état – est très importante et aborde plusieurs questions d'interfaçage. Les fenêtres, buffers, etc., tout ceci a besoin de la notion d'état qui n'existe pas en mathématiques pures.

4. Nous voulons implanter et exploiter la stratégie/style CPS. Un calcul n'est pas une valeur, mais une «valeur continuée», un objet fonctionnel dont l'argument est la fonction – continuation qui récupère la valeur en question.

5. Si notre programme lit ou écrit des informations sur les flots extérieurs (fichiers), ces opérations constituent des «effets de bord» dans les langages impératifs. Nous définirons un calcul approprié qui permet de parler de I/O sans quitter le monde pur de la programmation fonctionnelle. Lamonade I/O est prédéfinie, et elle est très intimement liée avec la monade (générale) de transformateurs des états.

6. Dans la programmation logique une expression peut retourner plusieurs valeurs (plusieurs réponses à une question). Nous pouvons utiliser les listes paresseuses pour stocker ces réponses. Un calcul ici – la monade non-déterministe – est un moyen de récupérer ces réponses et de combiner ensemble des procédures non-déterministes.

C.1.2 Monades arbitraires et combinateurs

Cette section est générique, universelle, indépendante des détails. Nous avons dit qu'une monade est un constructeur de types de données, équipé, bien sûr, d'un certain nombre de fonctions de traitement. **Mais ces fonctions sont très générales. Les monades ne permettront de résoudre aucun problème concret, seulement de le structurer**. Imaginons donc, que pour les valeurs de type `a` il existe un moyen de construire un nouveau type `T a`, parametré par `a`, qui représente le calcul qui peut fournir une valeur de type `a`.

La lettre `T` est ici purement symbolique ; nous aurons *plusieurs types monadiques différents*, parametrés par plusieurs types de base. Il y aura des monades privées, construites par nous même, comme la monade du parsing, et les monades-système, comme la monade I/O (entrées et sorties).

La fonction générique fondamentale qui «injecte» une valeur dans un calcul, c'est-à-dire qui définit un calcul susceptible de rendre cette valeur, est la fonction `return`. Pour la monade triviale `return x = x`, ou `return = id`. Dans le cas général le type de cette fonction est

```
return :: a -> T a
```

Il est utile d'ajouter à l'ensemble d'opérations primitives qui concernent les monades aussi la fonction **`fail`** de type **`fail :: String -> T a`** qui symbolise l'échec d'un calcul. L'argument-chaîne peut servir à transmettre un message diagnostique.

Le combinateur principal permettant d'enchaîner les calculs est l'opérateur ***bind*** : **`(>>=)`**. Son type principal est :

```
(>>=)  :: T a -> (a -> T b) -> T b
```

c'est-à-dire, que **`m >>= f`** possède l'interprétation suivante :

- Le *calcul* **`m`** est effectué. Il rend (normalement) une valeur, disons **`x`**. (Dans le cas nondéterministe «une» valeur ici signifie *une valeur quelconque*; le **`bind`** doit les récupérer toutes).
- La fonction **`f`** est appliquée à **`x`**, et produit – non pas une valeur, mais **un autre calcul.** les calculs mènent aux calculs, qui mènent aux calculs, qui… On ne sort pas si facilement du monde monadique quand on y entre une fois.

(Pour la monade triviale : **`m >>= f`** ≡ **`f m`**.) En général, si on «plonge» dans le monde des objets et fonctions monadiques, on y reste, **tout** est monadique, et pour sortir de la chaîne composée par *bind* il faut faire une opération spéciale, qui pour chaque monade peut être très différente. Pour la monade triviale on ne fait rien ; pour les parseurs on les applique à un flot. Pour la monade IO qui décrit le système des entrées/sorties en Haskell *il n'y a pas d'issue !* (sauf quelques extensions sémi-légales, déconseillées aux débutants. Ceci ne doit pas empêcher le lecteur de dormir. La monade I/O représente un «programme principal», et personne n'est malheureux à cause du fait que l'on ne sort pas du programme principal). Pour la monade non-déterministe la sortie est ambiguë… Nous y reviendrons.

Pour un type monadique **`T a`** il sera souvent utile de disposer d'une fonction surchargée **`fmap`** qui applique une fonction «normale» à tous les éléments d'une structure, et reconstruit une structure conforme. Rappelons que **`fmap`** généralise la fonction **`map`** définie pour les listes, et que la surcharge est spécifiée par la classe **`Functor`**.

La forme syntaxique souvent présente dans des programmes monadiques est

```
m >>= \x -> ... faire qqchose avec x ...    ...  return uneValeur
```

Parfois on n'a pas besoin de **`x`**. Il existe une version tronquée de *bind*, l'opérateur **`(>>)`** que nous allons appeler ***suite*** (ou ***ensuite*** ; en anglais il est parfois appelé : *then*, mais ce nom risque de provoquer des malentendus. Son type est

```
(>>)   :: T a -> T b -> T b
```

c'est-à-dire : prendre deux calculs, un après l'autre, et les combiner en un seul calcul composite. Le type du résultat est le type du second argument.

Les monades sont universelles, surchargées, donc elles sont réalisées en Haskell par une classe, la classe des constructeurs : **`Monad`**. Voici sa déclaration :

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a

  -- Minimum à impléménter : (>>=), return. Le reste, par défaut :
  p >> q  = p >>= \ _ -> q
  fail s  = error s
```

À présent nous pouvons construire quelques instances, implémenter *bind* etc., et définir aussi des fonctions spécifiques pour chaque type monadique, les «vrais» acteurs dans le programme, ceux qui «font» quelque chose d'intéressant.

C.2 Exemples de monades non-triviales

Voici encore une fois une collection de monades populaires, qui représentent des «calculs» très divers. Les détails seront élaborés dans des sous-sections.

Il faut d'abord avouer que les monades *sont spécifiques* à la programmation fonctionnelle. En C++ on peut les implanter – en principe, mais elles seront redondantes. Mais C++ est un langage «impur», avec une sémantique qui est plus intuitive que formelle, avec beaucoup de «bricolage» de bas niveau.

Nous avons décidé de présenter ce cours dans le cadre fonctionnel ***parce que la programmation fonctionnelle est simple, bien structurée, et formalisable***. Les programmes sont faciles à analyser (et à déboguer), et grâce aux fonctions d'ordre supérieur et le polymorphisme, ils sont très compacts. Mais on peut craindre, que les langages fonctionnelles sont plutôt limités, les concepts comme affectation des variables ou les branchements — tout ce qui appartient à la couche inéluctable de la programmation de bas niveau, et donc doit être traité par un cours de compilation — restent en dehors. Ceci est une impression erronée. À l'envers, nous pouvons dans le cadre fonctionnel découvrir le «sens» logique des concepts comme les branchements, ou les effets de bord liés aux opérations d'entrée-sortie. Les monades sont ici très utiles.

Il y a des monades pour le parallélisme. Nous connaissons déjà les monades du parsing et dans une section ultérieure nous allons décrire le système I/O monadique de Haskell. Les variables «mutables» (p. ex. les tableaux modifiables) s'expriment aussi par les monades. La connaissance de cette partie de la sémantique des langages de programation est devenue presque incontournable.

Enfin, nous pouvons avoir besoin d'une fonction d'«aplatissement» des structures monadiques. Si une fonction construit l'objet monadique de type `T a` d'une valeur appartenant au type `a`, on peut imaginer son application à un objet qui *est déjà monadique*. On obtient quelque chose comme `T (T a)` pour son type, et il faut réduire le résultat au type `T a` de nouveau. Ceci sera très utile dans le cas de listes – la monade non-déterministe.

C.2.1 Exceptions

Le premier cas non-trivial de monades est la monade – disons – *Peut-être* qui permet de définir les erreurs, ou les exceptions. (Présentées de manière très simpliste ! Les vraies exceptions sont plus riches.)

La programmation fonctionnelle classique semble triviale si tout va bien, mais si on demande $1/0$, ou `sqrt(-4.6)` (dans le domaine des réels), ou la tête d'une liste vide, etc. – qu'est-ce passe-t-il? On peut toujours dire : «Ça bombe», mais ceci est une non-réponse. Il est évident, qu'en construisant un compilateur il faut savoir répondre constructivement à une telle question. Que cela bombe, pourquoi pas, mais c'est à nous de définir l'explosif, sa portée, et les moyens de désamorçage. Un code qui n'est pas capable de gérer les erreurs est mortellement dangereux !

***Credo* religieux no. 16 :** Un programmeur lambda peut croire en démons ou dieux qui se mêlent dans ses affaires et qui prennent les décisions dans des situations inextricables. Un concepteur et réalisateur des compilateurs et/ou des interprètes doit être 99.9% athée. Pourquoi pas à 100%? Parce que la «magie» des opérations primitives, et l'architecture physique du processeur restent toujours là. Mais ces opérations doivent être vraiment primitives, et sûres.

Notre «calcul» sera défini par le type `Maybe` (prédéfini) :

```
data Maybe a = Just a | Nothing
     deriving (Eq, Ord, Read, Show)
```

La transformation d'une valeur en calcul a la forme

```
return x = Just x
```

N'oublions pas, que cette définition doit avoir lieu dans une instance de la classe `Monad` :

```
instance Monad Maybe where
  return  = Just
  ...
```

L'enchaînement, c'est à dire la généralisation de l'application, est plus élaborée :

```
  Just x  >>= k = k x
  Nothing >>= k = Nothing
  fail s        = Nothing
```

et il nous reste d'ajouter l'amorceur qui sera déclenché en cas de besoin. On ne permettra jamais que la couche magique, celle de la machine de plus bas niveau, prenne cette décision, donc *notre* opérateur de division doit vérifier le diviseur avant d'essayer la division primitive. Si **x** et **y** sont des calculs, et **primDiv** – l'opération de division primitive (magique), la division sera (par exemple) définie comme

```
x / y = x >>= \a ->
        y >>= \b ->
        if b/=0 then return (a/b)
           else fail
```

Bien sûr, on peut imaginer l'élargissement de ce système, par exemple l'introduction des paramètres qui discriminent entre de différents cas de «rien», ce qui permettra d'enrichir les messages diagnostiques, ou définir *plusieurs classes d'exceptions*. Ce système permet également de désamorcer la bombe, d'intercepter l'exception et de la transformer en une autre valeur. C'est ainsi que les formes syntaxiques de genre

```
try
  { calcul dangereux }
with
  excpt1 -> secours1
  excpt2 -> secours2
  ...
```

peuvent être réalisées. (Ceci n'est pas un programme Haskell valide !) Si le calcul se termine bien, la valeur monadique **(Just v)** est retournée, sinon **try** neutralise la bombe en appelant les secouristes. L'opération **try** est capable de briser le cercle enchanté d'enchaînement monadique.

Ce qui est le plus important ici est le remplacement *de toute application fonctionnelle normale* **(f x)**, sauf – naturellement – dans le cas du **try** – par **(x >>= f)** (si **x** est déjà un calcul ; au début on lance **return z** pour créer le premier calcul depuis la valeur initiale **z**).

En fait, le type **Maybe** est aussi un **Functor**. Voici la définition dans le Prélude :

```
instance Functor Maybe where
  fmap f Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

Le *bind* et **fmap** sont très liés, et ceci n'est pas un accident. Cette monade est presque fonctionnelle pure, et la suite **(>>)** n'a pas beaucoup de sens.

C.2.2 Monade non-déterministe

Cette fois le calcul correspond à un ensemble (**ordonné !**) de valeurs, qui intuitivement représente le choix parmi eux. L'implantation réaliste dans des cas sérieux, où le programme peut générer des millions d'alternatives (traitées incrémentalement) n'est possible qu'avec un langage paresseux, mais si on le choix entre 10 – 100 possibilités, un langage strict convient également. La monade en question peut être définie par

```
type Calcul a = [a]
```

L'enchaînement des calculs définies de cette manière nous est déjà connu. Chaque valeur alternative élémentaire soumise à une nouvelle transformation génère une nouvelle liste de possibilités. Donc, globalement, la valeur monadique non-déterministe (la liste) est d'abord «mappée» par la fonction de transformation, et ensuite la liste de listes est aplatie.

```
fail _ = []
return x = [x]          – Une seule possibilité
[]    >>= f = []
(x:q) >>= f = f x ++ (q >>= f)
```

ou, plus simplement : **((>>=) = concat . map)**, et le monde de la programmation non-déterministe (au niveau des données) est à notre disposition. Cependant ni les monades ni la programmation paresseuse ne peuvent nous enseigner à *penser de manière non-déterministe*, de formuler nos problèmes calculatoires de cette façon. N'oubliez pas de lire les exercices de ce chapitre.

Répétons : pour faire un programme qui «fait» quelque chose, il faut programmer cette action, définir des fonctions spécifiques. Les monades apportent «de la colle» pour assembler des programmes plus grands. Dans

le contexte de nos opérations non-déterministes, là où un programme peut rendre une réponse parmi deux, nous allons concaténer les deux résultats.

Exemple.
Le prédicat d'insertion non-déterministe en Prolog, qui permet de mettre un objet dans une liste sur n'importe quelle position, est défini comme

```
ndins(X,L,[X|L]).
ndins(X,[A|Q],[A|R]) :- ndins(X,Q,R).
```

L'appel `ndins(a,[b,c,d],R).` doit engendrer : `[R=[a,b,c,d]; R=[b,a,c,d]; R=[b,c,a,d]; R=[b,c,d,a];`. La traduction «aveugle» et complètement erronée de Prolog en Haskell se réduit au changement de syntaxe :

```
ndins x l = (x:l)
ndins x (a:q) = a : ndins x q
```

Mais les deux clauses ne s'excluent pas, elles sont réellement alternatives coexistantes. On doit donc concaténer les résultats :

```
ndins x l@(a:q) = (x:l) ++ (a : ndins x q)
```

C'est toujours faux !

- La valeur `(x:l)` est *une réponse*, et non pas un objet monadique (réponse multiple). Il faut la remplacer par `return (x:l)`.
- L'opération `(a : arg2)` est «normale», et demande que arg2 soit une valeur standard, et non pas un objet monadique, ce qui est le cas ici, puisque c'est le résultat du `ndins`. Il faut utiliser *bind*, et ne pas oublier de «monadiser» le résultat.

Donc, enfin, on obtient :

```
ndins x l@(a:q) = return (x:l) ++ ((ndins x q) >>=(\g -> return (a:g)))
ndins x [] = return [x]
```

où nous avons complété la définition par la clause manquante. (En Prolog elle est redondante, car la première et seulement la première clause produit un résultat. En Haskell le filtrage échoue et provoque une erreur).

Le reste est l'optimisation. On sait que `[a]++b` ≡ `a:b`. On sait aussi que le *bind* peut être formulée par `map` :

```
l >>= f = concat (map f l)
```

(***Exercice :*** Prouvez-le). La fonction `concat` aplâti la liste : `[[a,b], [c],[d,e,f]]` → `[a,b,c,d,e,f]`. Mais la construction

```
concat (map (\g -> [(a:g)]) (ndins x q))
```

d'abord ajoute les crochets internes, pour ensuite les enlever, ce qui est équivalent à

```
map (\g -> (a:g)) (ndins x q)
```

Et finalement

```
ndins x l@(a:q) = (x:l) : map (a:) (ndins x q)
ndins x [] = [[x]]
```

ou, si on veut :

```
ndins x l = (x:l) : case l of
              []    -> []
              (a:q) -> map (a:) (ndins x q)
```

Avec l'insertion non-déterministe nous pouvons générer la liste de toutes les permutations des éléments d'une liste-source. l'algorithme est le suivant : on enlève la tête, on trouve toutes les permutations des éléments restants, et pour chaque résultat on réinsère la tête sur toutes les positions possibles. En Prolog nous aurions

```
perm([],[]).
perm([X|P],R):-perm(P,L),ndins(X,L,R).
```

Voici la traduction aveugle, syntaxique :

```
perm [] = []
perm (x:p) = ndins x (perm p)
```

Et la vérité :

```
perm l@(a:q) = (perm q) >>= ndins a
perm [] = return []
```

où il ne faut pas confondre `[]` et `return[]` !

Les exercices contiennent d'autres exemples, et il en reste encore beaucoup pour l'examen...

C.2.3 Monade du tracing

Nous avons proposé un exercice : ajouter à notre machine virtuelle à pile un débogueur qui dûment affiche les opérations exécutées. Ceci est une technique assez simple a réaliser dans un programme impératif quelconque. Après chaque instruction on ajoute une commande d'affichage qui peut répertorier les valeurs des variables locales, arguments, etc. Cette technique est simple et utilisée, même s'il s'agit du bricolage.

Comment le réaliser dans un programme fonctionnel? Comment ajouter une option de déboguage *dans une expression*? Les usagers du Lisp etc. connaissent la réponse : le support *runtime* (l'ensemble des primitifs de gestion de mémoire et autres ressources pendant l'exécution du programme), peut brancher la magie de déboguage. Par exemple on exécute `(debug fun1 fun2 fun3)`, et l'interprète change son mode interne d'évaluation – chaque appel à `fun1`, `fun2` ou à `fun3` est intercepté, et la machine déclenche une activité accessoire, un effet de bord se produit – le message est affiché. On n'a pas besoin de changer le programme !

Mais alors comment ce dispositif a-t-il été réalisé? L'implantation la plus primitive consiste à *redéfinir les fonctions tracées*, ce qui peut être fait manuellement. Par exemple si `fun1` est une fonction de deux arguments, nous pouvons écrire en Scheme

```
(define old_fun1 fun1)
(define (fun1 x y)
  (display "Fun1 appelée avec arguments ")
  (display x) (display " et ") (display y) (newline)
  (let ((res (oldfun x y)))
    (display "Résultat de Fun1 : ") (display res)
    res)
))
```

Une telle manipulation peut être largement automatisée par quelques macros appropriées. Ceci est plus difficile dans un langage typé, et semble impossible dans un langage fonctionnel pur ou aucune possibilité de mettre `(display ...)` n'existe.

Les monades suggèrent une solution fonctionnelle et bien structurée. On définit

```
type Calcul a = (a,String)
return x = (x,"")
```

Le calcul est donc une paire qui contient la valeur, mais qui y associe une chaîne diagnostique. Toute fonction conforme avec le protocole de l'évaluation tracée, peut ajouter sa contribution à la chaîne finale, si elle est enchaînée par

```
(x,msg) >>= fun  = (y,msg ++ suiv)
    where (y,suiv) = fun x
```

et pour la bonheur totale nous pouvons ajouter une fonction qui transforme une simple chaîne diagnostique ou autre en monade

```
out x = ((),x)
```

en injectant une valeur nulle. (Ceci doit rappeler le parseur `posit` qui construisait une valeur artificielle à partir du flot d'entrée, mais sans le consommer – le parseur récupérait seulement l'information positionnelle).

Et, encore une fois : ***Les monades ne fournissent seules aucun mécanisme de traçage !*** elles permettent à peine à l'utilisateur de structurer son programme de manière non-triviale, en exploitant la généricité des fonctions d'ordre supérieur.

Bien sûr, il nous reste de faire le «lifting» des opérations «normales», qui au lieu de produire une simple valeur, créent les paires monadiques, en ajoutant au résultat le message approprié. Par exemple

```
out (shows "J'applique la fonction FUN à " . show xx)
  >>= return(fun xx)
```

Ce qui est fort intéressant et même étonnant, est la possibilité de construire la chaîne d'affichage à l'envers, en concaténant `suiv ++ msg` à la place de la construction ci-dessus. Essayez de le faire dans un programme impératif…

Une remarque importante : une telle stratégie du tracing convient bien à la programmation paresseuse ; si le «programme principal» n'est rien d'autre que l'affichage de la chaîne tracée, cette chaîne sera construite de manière incrémentale, et affichée lors du déroulement du programme ; un programme strict aurait d'abord empilé tout dans une zone interne de stockage, ce qui pourrait déborder la mémoire. ***Le traçage impératif est vraiment différent***.

Que cela a-t-il à voir avec la compilation? La réponse est immédiate : Notre compilateur peut être parametré par des options de déboguage, et ces options modifieront les définitions monadiques de `(>>=)` effectueront le *lifting* de quelques fonctions, etc., mais *la structure du code compilé restera globalement la même*.

D'autre part, ce mécanisme peut être utilisé aussi pour ajouter le traçage à une machine virtuelle. Une autre solution monadique du problème de traçage serait d'imbriquer le programme dans la monade IO qui sera dsicutée prochaînement.

C.2.4 États et transformateurs

Cette monade est d'une très grande importance, car elle nous approche de la programmation impérative, classique, avec les «effets de bord» et la sérialisation des instructions. Elle est également indispensable pour la discussion des parseurs, générateurs du code, etc.

Rappelons-nous la définition de la machine virtuelle à pile, et la structure du code interprété. Un `CodeItem` était un objet, une donnée qui cachait à l'intérieur un objet fonctionnel, qui agissait sur les piles de données et des retours, et éventuellement sur l'environnement. Nous avons mentionné que la structure de la machine virtuelle : la gestion des piles etc. resterait la même si on codait tout en un langage impératif, mais dans ce cas les piles et le tableau-environnement feraient partie d'un état *global* du système, et ces données globales auraient été soumises aux modifications par des effets de bord.

Ceci suggère l'approche suivante à l'émulation de la programmation impérative par un programme fonctionnel. Le calcul est un *transformateur des états*. Dans un programme impératif l'état est une notion globale, implicite. Dans un programme fonctionnel il est un *objet* explicit, une donnée.

Attention ! Chacun peut faire ses propres monades et son modèle d'état, comme nous l'avons fait avec les parseurs. Mais sur le plan pratique l'efficacité de quelques monades est basée sur le fait que l'état correspondant est implémenté comme primitif, et optimisé par le compilateur. C'est le cas de la monade IO (entrées/sorties).

Chaque valeur est combinée avec l'état du système. Chaque fonction appliquée à une valeur fait quelque chose avec l'état, et retourne la nouvelle valeur-résultat, ainsi que le nouvel état. En fait, *le calcul lui-même fait quelque chose avec l'état*, la *valeur* de la monade triviale, une simple donnée, se transforme en un objet actif. Notre «calcul» peut finalement «faire» quelque chose. Le type monadique correspondant peut être spécifié comme

```
type T a = State -> (a,State)
```

où `State` est un type qui symbolise l'état. (Il peut être très varié ; les états utiles pour le parsing : flot d'entrée, éventuellement l'inrmation positionnelle, et les états-compteurs utilisés pour le *benchmarking* : compteurs, sont très différents.) Ce type monadique est assez loin de ce que nous considérons comme une «valeur». Même en exécutant une opération arithmétique, par exemple l'addition de deux nombres, ceci est une construction fonctionnelle : deux fonctions produisent une troisième – le résultat. *Ce n'est qu'applicant ce résultat à un état, que l'on récupère la valeur résultante et l'état final !*

La fonction **`fail`** dans le cas général n'existe pas, c'est une exception. On peut – selon le cas – prévoir un état «erreur» spécial, ou déclencher une erreur-système, ou, si cette monade est embarquée dans une autre, comme nos parseurs qui combinent la transformation d'état (consommation du flux) avec le non-déterminisme (réponses multiples) – on peut propager l'échec vers cette dernière.

La fonction **`return`** est évidente, on retourne une valeur et on ne touche pas l'état :

```
return x = \st -> (x,st)
```

Le combinateur *bind* est fort intéressant. Ici l'état peut changer deux fois, d'abord comme le résultat du premier argument de **`(>>=)`**, et ensuite par la fonction qui s'applique au premier résultat. Voici le combinateur en question :

```
m >>= f = (\s_in -> let (x,s_int) = m s_in
                       (y,s_fin) = (f x) s_int
                   in  (y,s_fin)
```

Le paramètre **`s_in`** symbolise l'état initial, les autres sont : intermédiaire et final. Le combinateur *bind* peut être un peu simplifié, la dernière clause dans **`let`** est redondante, nous voulions visualiser explicitement l'enchaînement, en montrant la transmission des valeurs. Ceci doit être comparé avec le changement du flux d'entrée par la composition des parseurs.

Pour donner un simple exemple, construisons un programme pour le *benchmarking*. L'exécution (application) de toute fonction incrémente un compteur spécial de 1. À la fin nous pouvons récupérer le compteur et évaluer pratiquement la complexité d'un programme en termes du nombre d'opérations. L'état est donc un entier – la valeur du compteur.

```
type State = Integer              – Le compteur
type Calcul a = State -> (a,State)
return x = \t -> (x,t)          – Ne change pas l'état
tick     = \t -> ((),t+1)       – "Tick !" sans retour de valeur
```

où la dernière fonction devra être incorporée dans l'ensemble des fonctions de base du système. La fonction **`tick`** joue un rôle un peu similaire à la fonction **`posit`** dans le monde des parseurs – elle récupère une propriété de l'état. Mais ici cette propriété n'est pas transformée en valeur. Pour cela nous pouvons avoir un autre combinateur :

```
time = \t -> (t,t)
```

Les fonctions de base, les opérateurs arithmétiques, manipulateurs des listes, et toute cette panoplie présente dans un programme typique doit subir un «lifting», car *toute* fonction typique doit maintenant gérer l'état. L'opérateur **`(>>=)`** ne fait pas de miracles, il permet seulement d'enchaîner les calculs, et de «cacher» la gestion du compteur. De même, l'opérateur **`(>>=)`** dans le domaine des parseurs cache la gestion du flux d'entrée ; la définition du parseur devient plus abstraite et ressemble plus à une production syntaxique, c'est tout.

Dans l'exemple ci-dessus, si dans le formalisme standard, une fonction **`f`** agit sur **`x`** et produit **`y`**, le *lifting* de **`y = f x`** est

```
y = tick >> return(f x)
```

si à cet instant-là on n'a pas besoin de la valeur du compteur.

C.2.5 Monade CPS

Dans cette section nous allons traiter le *Continuation Passing Style*. Nous avons déjà mentionné les continuations – le concept qui permet de répondre à la question : «qu'est-ce qu'on fait après avoir terminé l'évaluation en cours». L'affinité et la ressemblance entre les continuations CPS classiques – les relais des fonctions et la passation des résultats, – et l'enchaînement typique pour les compositions monadiques, est très, très grande. Ce problème est particulièrement important, car les continuations nous donnent le moyen de formaliser les structures de contrôle impératives (branchements) et permettent leur compilation sans sortir du cadre fonctionnel.

Mieux encore, dans quelques langages comme Scheme, même si les continuations restent normalement invisibles (implicites), comme dans d'autres langages de programmation, il existe un objet fonctionnel : **`call/cc`**

(ou **`call-with-current-continuation`**), qui permet d'«attrapper» la continuation courante, le futur contenant *toutes* les actions qui devraient être exécutées par la suite (au moment de l'appel de **`call/cc`**), et donner à l'utilisateur la possibilité de relancer le système à partir de ce «moment».

Ceci permet d'implémenter les co-procédures et le processus parallèles, et aussi le *control backtracking* – un autre visage du non-déterminisme logique : la possibilité de effectuer plusieurs actions alternatives, et non pas seulement rendre une réponse (valeur) multiple. Mais cette problématique ne sera pas discutée en cours, elle est trop complexe.

Le rapport entre le CPS et les monades est spécifié par les clauses suivantes.

- Pendant l'exécution du programme tout objet (valeur) généré par une fonction «attend son consommateur». Nous pouvons noter cela comme

  ```
  return x = \cnt -> (cnt x)
  ```

  où la fonction dénotée ici par le paramètre **`cnt`** consomme la valeur, et produit une **`Réponse`**. Ce qui peut être une **`Réponse`** sera commenté un peu plus tard (ceci peut être, bien sûr, une valeur quelconque. La continuation finale peut être le combinateur **`id`** qui sort de la chaîne monadique.) Voici donc le type monadique :

  ```
  type Calcul a = (a -> Réponse)  -> Réponse
  ```

  Encore une fois : une valeur initiale est injectée dans le programme, et se transforme en Calcul quand on lui attribue sa continuation. Donc, le type **`Calcul`** est fonctionnel. Ce qui peut paraître un peu bizarre est le fait que *dans* la section monadique du programme on ne récupère jamais une valeur de type injecté par **`return`**, car *toute* fonction est continuée ! Les valeurs construites par des fonctions continuées défilent à travers la chaîne de continuations, mais on ne récupère quelque chose que quand on sort de cette chaîne.

  Si le lecteur se sent mal à l'aise à cause de cela, il doit rester calme. Dans un programme en C quelconque, la situation est pire, car normalement on ne récupère *aucun* résultat, on peut seulement profiter de quelques effets de bord liés aux instructions d'affichage (et l'affectation des variables).

- Nous pouvons donc attribuer à l'objet final un type **`Réponse`** qui n'est pas réductible *dans* le programme. Ceci peut être l'affichage final de la réponse, une chaîne. Bien sûr, on peut construire un *fragment* d'un programme réel de cette façon, on n'est pas obligé à suivre cette philosophie jusqu'au but. La Réponse peut être un nombre ou une chaîne quelconque, ou tout autre objet qui peut être affiché, ou traité par des fonctions en dehors de la chaîne des continuations.

  Mais un compilateur peut appliquer cette stratégie au pied de la lettre dès le début jusqu'au codage de l'arrêt du programme. La **`Réponse`** est alors envoyée à l'application appelante, par exemple le système d'exploitation. cette réponse peut être alors le code d'arrêt, ou une chaîne, éventuellement le descripteur d'un fichier.

- Si nous voulons appliquer une fonction, son résultat doit être également «lifté» aux valeurs continuées, comme toute fonction qui se trouve à droite de l'opérateur *bind*. Au lieu d'avoir **`y = f x`**, nous allons opérer avec un objet plus compliqué que **`f`**, disons **`g = lift f`**, **`g :: a -> Calcul b`**, où :

  ```
  (lift f) x cnt = cnt (f x)
  ```

  et si un objet **`m`** est déjà une «valeur continuée>, un calcul, et **`g`** est une fonction déjà «liftée», monadique, alors elle sera appliquée par *bind* de manière suivante :

  ```
  m >>= g    =   \cnt -> m (\r -> g r cnt)
  ```

  D'abord **`m`** est lancé, et appliqué à une continuation intermédiaire, qui récupère la valeur **`r`** de **`m`**, et lui applique la fonction monadique **`g`**. Si **`m`** est primitif (**`return x`**), et **`g`** est le résultat direct du lifting de la fonction normale **`f`**, présenté ci-dessus, alors *bind* se réduit a

```
(return x) >>= (lift f)   =
   \cnt -> (\c -> c x) (\r -> cnt (f r)) =
   \cnt -> (\c -> c x) (cnt . f) = \cnt -> (cnt . f) x =
   \cnt -> cnt (f x)
```

comme il fallait espérer.

***Credo* religieux no. 17 :** Ceux qui s'intéressent par la sémantique des langages de programmation, et qui ignorent les monades, sont des dinosaures anté-diluviens. Ceux qui refusent de les enseigner en affirmant qu'elles sont trop difficiles pour les étudiants, sont des dinosaures post-diluviens.

C.3 Système I/O de Haskell

Nous avons placé cette section dans le chapitre monadique, car le système des entrées/sorties de Haskell est la quintessence de l'approche monadique à l'implantation d'un langage de programmation. Les monades IO de Haskell sont un peu différentes des autres : ***elles sont internes, implantées par des primitives système, et très bien optimisées.***

Attention ! Cette section ne dispense pas les lecteurs de la lecture de la documentation de Haskell. Nous ne pouvons pas traiter la totalité du sujet, et les ommissions peuvent devenir gênantes un jour (p.ex. le jour d'examen...

Ceux qui veulent *comprendre* (en non pas seulement utiliser) le système d'entrées/sorties de Haskell peuvent imaginer qu'il existe une structure de données spéciale : le «Monde» (extérieur), qui appartient à un *état*, comme le flux d'entrée dans la construction des parseurs. Les fonctions IO de Haskell effectuent des opérations sur cette structure et produisent l'effet combiné : le résultat qui appartient au programme utilisateur, et un Monde modifié, qui va rester caché. Ce fait, de ne pas permettre au programme d'accéder directement au Monde n'a rien de spécial – en C on n'accède pas aux descripteurs de fichiers ni au tampon d'écran (sauf si on fait de la programmation système de bas niveau).

Mais dans un langage fonctionnel ce protocole possède une spécificité : on peut imaginer que le programme manipule le Monde comme n'importe quelle autre structure, sans aucun effect destructeur. On prend le Monde_0, et on rend Monde_1, une autre structure. Cependant le Monde existe en un seul exemplaire, et physiquement aucune création du monde n'a lieu, on retourne l'original modifié (fichiers lus, positionnés, ou écrits, écran rempli d'objets graphiques, etc.) Le protocole monadique empêche que le programme puisse accéder en même temps à l'original et au Monde modifié, et aucune situation paradoxale ne peut avoir lieu.

La totalité des opérations d'entrée/sortie concerne un type monadique spécial `IO a`, où `a` est le type du résultat (lu ; pour l'écriture souvent `a=()`, l'écriture ne rend aucun résultat utilisable).

Le lecteur doit déjà être préparé à la spécificité des programmes qui effectuent les opérations de lecture/écriture : ***la totalité du programme est imbriquée dans une chaîne monadique IO, car quand on entre dans cette chaîne, on ne peut plus sortir...*** (Bien sûr, ceci n'est pas vrai si on fait des tests intéractifs sous Hugs, cette restriction concerne les programmes en Haskell compilés, par exemple par le compilateur GHC).

Les opérations `return` et `(>>=)` sont primitives. Mais, même si plusieurs autres opérations pour des raisons d'efficacité sont primitives aussi, l'essentiel de la magie et limité, l'écriture d'une chaîne peut être réalisé comme l'enchaînement des actions sur les caractères, etc.

C.3.1 Notation «do»

Haskell possède une extension syntaxique qui jusqu'à présent a été bien cachée des lecteurs : le bloc «do» qui facilite la programmation monadique dans un style qui ressemble le codage impératif. Au lieu d'écrire

```
putStr "Entrez une chaîne : "  >> getLine >>= \l ->
    return (traitement l)
```

(où `putStr` affiche une chaîne, et `getLine` lit une ligne depuis le flot d'entrée standard), on peut se permettre à formuler ceci comme ;

```
do  putStr "Entrez une chaîne : "
    l <- getLine
    return (traitement l)
```

En général, la forme **do {***instructions***}** contient une séquence d'*instructions* séparées par le point-virgule, où chaque instruction est un appel fonctionnel parfaitement normal, ou une *déclaration* **let** (sans la partie **in ...**, ou une forme : **variable <- expression**.

Voici la traduction du bloc **do** en Haskell plus habituel.

```
do {e}                    ⇒  e
do {e; reste}             ⇒  e >> do {reste}
do {p<-e; reste}          ⇒  let tmpf p = do {reste}
                                 tmpf _ = fail "..."
                             in  tmpf
do {let declars; reste}   ⇒  let declars in do {reste}
```

Les point-virgules et les accolades sont redondants si on exploite correctement le *layout* (indentation), mais ces lexèmes sont souvent conseillés pour la lisibilité du programme.

C.3.2 Flots standard

Les opérations permettant d'écrire quelque chose sur la console standard sont les suivantes :

```
putChar    :: Char -> IO ()       -- affiche un caractère
putStr     :: String -> IO ()     -- ou une chaîne
putStrLn   :: String -> IO ()     -- ajoute la fin de ligne
print      :: Show a => a -> IO ()
```

Voici un programme Haskell complet :

```
main = print ([(n, 2^n) | n <- [0..19]])
```

qui affiche (0,1), (1,2), (2,4), (3,8), etc. Si on travaille sous l'interprète Hugs on peut tester intéractivement les commandes I/O. Cependant Haskell est un langage compilé, et si on veut utiliser un compilateur comme GHC ou NHC, il lui faut préparer un «programme principal». Ceci est la variable **main** qui appartient au type **IO ()**. Dans sa définition nous pouvons ouvrir les fichiers, les écrire, etc.

La fonction **print** utilise **show** pour convertir un objet quelconque en chaîne, et ensuite l'imprime.

La lecture du flot standard utilise les fonctions suivantes :

```
getChar     :: IO Char                  -- lecture d'un caractère
getLine     :: IO String                -- ou d'une ligne
getContents :: IO String                -- la totalité de l'entrée
interact    :: (String->String)->IO ()
readIO      :: Read a => String->IO a   -- lecture d'une donnée quelconque
readLn      :: Read a => IO a
```

La classe **Read** spécifie un petit parseur capable de lire les structures de données Haskell, si – bien sûr – l'utilisateur définit leur forme extérieure dans l'instance de **Read** correspondante.

L'opération **interact** est très intéressante, elle permet une conversation intéractive entre l'utilisateur et le programme. Elle est très simple :

```
interact f = getContents >>= (putStr . f)
```

Son argument est une fonction qui transforme une chaîne en une autre chaîne. Ceci peut être une fonction très complexe, par exemple un parseur. Mais comment on peut intéragir si le programme veut d'abord consommer la totalité du flot d'entrée? L'astuce consiste à exploiter la paresse. La fonction **getContents**, comme il a été dit, consomme *tout* du flot d'entrée. Mais le résultat est une chaîne, donc une liste paresseuse. Si l'utilisateur n'a momentanément besoin que d'un seul caractère, les autres ne seront pas lus (sauf si le système d'exploitation précipite quelques manipulations, en forçant – par exemple – la lecture d'une ligne entière).

Le programme

```
main = interact (filter isAscii)
```

lit tout, mais supprime tous les caractères non-Ascii. la fonction **getLine** est définie comme suit :

```
getLine = do c <- getChar
             if c == '\n' then return ""
                          else do s <- getLine
                                  return (c:s)
```

C.3.3 Fichiers

La généralisation de la lecture/écriture à d'autres fichiers n'est pas plus complexe que dans d'autres langages de programmation. Il faut simplement passer aux fonctions correspondantes le nom du fichier (une chaîne, souvent pour la lisibilité transformé en un synonyme, p. ex., **`Path`**), ou – éventuellement – un descripteur obtenu par l'opération d'ouverture. Les fonctions standard sont

```
type FilePath =  String

writeFile     ::  FilePath -> String   -> IO ()
appendFile    ::  FilePath -> String   -> IO ()
readFile      ::  FilePath             -> IO String
```

Pour lire un fichier et faire quelque chose avec son contenu, on écrira

```
readFile "mon_fichier.txt" >>= \s -> traitement s ...
```

il faut néanmoins garder toujours à l'esprit que le traitement ne peut sortir de la chaîne monadique. À la fin on peut éventuellement mettre **`return "Au revoir"`**.

Ceci est tout dans ces notes, mais la description complète est plus longue. Pour des utilisations sérieuses il faut maîtriser au moins

- la gestion d'erreurs ;
- la possibilité d'utiliser les fonctions écrites en C ;
- les sorties graphiques, les extensions spécifiques à Windows, à Unix, etc.

C.4 Exercices

Q1. Quel est le type principal de la fonction **`tf`** :

```
tf f p = p >>= return . f
```

R1. Ayant seulement le **`return`** et le *bind*, nous ne pouvons pas savoir dans quelle monade se situe le problème, mais il s'agit sans doute d'une monade qui spécifie le type du paramètre **`p`**. Peut-être on voit un peu plus comme ça :

```
tf f p = p >>= \x -> return (f x)
```

car ici le type du résultat et de l'argument de **`f`** deviennent plus lisibles. Soit **`a`** le type de **`x`** (le type de base de la Monade), et **`m`** – le type de constructeur monadique qui spécifie **`p`**. Alors la réponse est immédiate : **`ft ::  Monad m => (a->b) -> (m a) -> (m b)`**.

Q2. Construire explicitement une fonction qui joue le rôle de la structure **`try ...  with`** pour la monade **`Maybe`** généralisée un peu : **`Nothing`** sera parametré par une chaîne, en accord avec le **`fail`** monadique standard.

R2. La programmation fonctionnelle paresseuse est ici indispensable. nous construisons la fonction **`tryWith m secours`** qui lance le calcul **`m`** et le fournit au module appelant, mais qui neutralise la «bombe», en exécutant la fonction **`secours`** qui prend un paramètre – le message envoyé par l'expression qui a découvert l'erreur. La fonction doit être définie à un niveau assez bas, le même que celui de l'opérateur **`(>>=)`**.

```
tryWith m secours = case m of
  c@(Just x)  -> c
  Nothing s   -> secours s
```

Q3. Construire en Haskell une fonction qui calcule le *powerset* : l'ensemble de tous les sous-ensembles, d'une liste. La liste **`[a,b,c]`** doit générer : **`[[], [a], [b], [c], [a,b], [a,c], [b,c], [a,b,c]`**, au total 2^n, où n est la longueur de la liste. (Ceci est le cardinal de l'ensemble de fonctions : **`Bool -> elemDeListe`**).

R3. La stratégie non-déterministe consiste à parcourir la liste, et pour chaque élément soit le retenir, soit jeter. En Prolog la solution sera :

```
pset([],[]).

pset([X|L],[X|R]) :- pset(L,R).
pset([X|L],R)     :- pset(L,R).
```

Les deux dernières clauses sont évidemment non-conflictuelles, et engendrent la solution multiple. En Haskell nous pourrons tenter

```
pset [] = [[]]      -- Il ne faut pas se tromper !
pset (x:l) = let r=pset l
             in l ++ map (x:) l
```

où nous avons simplifié l'expression donnée par la traduction mécanique :

```
  l ++ concat (map (\c->return (x:c)) l)
```

Q4. Construire une fonction qui génère toutes les combinaisons de m éléments d'une liste de longueur n.

R4. Ceci ressemble au problème **`powerset`**, mais le choix est restreint. La solution Prolog : **`comb(M,Liste,Res)`** est légèrement plus complexe que **`pset`** :

```
comb(0,_ []):-!.
comb(M,[X|L],[X|R]):-M1 is M-1,comb(M1,L,R).
comb(M,[_|L],R):-comb(M,L,R).
```

Les deux dernières clauses sont naturellement non-exclusives, donc la solution fonctionnelle sera

```
comb 0 _ = [[]]
comb m p@(x:l) = map (x:) (comb (m-1) l) ++ comb m l
```

où les optimisations ont été effectuées presque sans réfléchir. Et d'ailleurs, cette manque de réflexion génère ici une petite catastrophe. ***La définition n'est pas complète ! Essayez de la corriger avant l'examen.***

Index