# Implantation des langages de programmation

Jacques Malenfant

maître-assistant

Département informatique

# Table des matières

# Chapitre 1

# Introduction

Pourquoi étudier l'implantation des langages de programmation ? Nous retiendrons trois raisons :

1. Parce que l'implantation d'un langage de haut niveau implique une traduction d'un modèle de calcul de haut niveau vers un modèle de plus bas niveau, une opération qui peut apparaître de coût élémentaire dans le modèle de calcul de haut niveau aura en réalité un coût plus important lorsque réalisée sur du matériel. Bien que les micro-processeurs ont tendance à inclure des instructions facilitant et rendant plus performante la traduction des constructions de haut niveau, certains coûts sont incompressibles du fait de l'architecture standard de Von Neumann. Idéalement, tout ingénieur doit être conscient du coût des constructions des langages de haut niveau qu'il utilise de façon à écrire des programmes à la fois compréhensibles et efficaces.
2. Chaque implantation d'un même langage peut utiliser différentes *techniques* d'implantation qui feront que le coût d'une opération élémentaire du langage de haut niveau peut être en réalité très différent d'une implantation à l'autre. Par exemple, dans un langage à objets, un envoi de message est une opération élémentaire, mais la recherche de méthode qu'il implique peut avoir un coût linéaire dans la profondeur du graphe d'héritage si elle est implantée naïvement. Un bon ingénieur doit être en mesure de juger les implantations de langages en fonction des techniques qu'elles utilisent, et pour cela connaître ces techniques.
3. Finalement, pour comprendre le comportement réel d'un programme, il est souvent nécessaire d'avoir une compréhension de l'implantation du langage. Cela est particulièrement vrai lorsqu'il faut utiliser un dévermineur pour examiner l'état d'un programme suite à un arrêt pour faute majeure. L'information sur l'état du programme suit la plupart du temps la structuration utilisée par l'implantation du langage ; la connaître permet donc de naviguer efficacement dans cette information pour retrouver celle qui permettra de comprendre l'origine de l'erreur.

Aujourd'hui la quasi-totalité des matériels informatiques sont alignés sur un standard *de facto* qui est essentiellement une variante plus ou moins complexe de la machine de Von Neumann. Pour exécuter un programme, il faut donc le traduire du langage de haut niveau dans lequel il est écrit vers un programme équivalent écrit dans le langage machine seul reconnu par les processeurs physiques. Quelle que soit l'approche utilisée, la traduction d'un programme en langage de haut niveau vers un langage de plus bas niveau rend toujours explicites certains coûts qui n'étaient pas apparents dans le modèle de haut niveau.

Ces coûts sont souvent associés aux structures de données utilisées pour implanter en langage de bas niveau les opérations de haut niveau. En fait, l'abstraction joue parfaitement son rôle de libérer le programmeur des problèmes de bas niveau lorsque celui-ci écrit son programme. Par contre, l'abstraction ne doit pas mentir sur le coût réel des opérations, sinon elle peut mener à des programmes totalement inefficaces. Une bonne abstraction est donc celle qui libère le programmeur de l'écriture, souvent répétitive, d'opérations de bas niveau, mais qui lui indique aussi les propriétés fondamentales de son utilisation en termes de complexité en temps et en espace des opérations et des structures de données utilisées.

Pour comprendre ces propriétés, comme par exemple le coût associé à un appel de fonction, ou de méthode, il n'est pas nécessaire de comprendre toutes les subtilités de leur implantation réelle. Pour cette raison, notre approche dans ce cours sera intermédiaire. Nous verrons l'implantation de plusieurs techniques, mais dans un contexte idéal et simplifié, celui d'un interprète écrit en Scheme. En parallèle avec cet apprentissage par l'action, procédant par l'étude et la réalisation de variantes de cet interprète, nous étudierons plus théoriquement leur utilisation dans les processeurs de langage concrets.

Au chapitre 2, nous introduisons les concepts généraux des langages de programmation. Ce chapitre se veut un rappel des notions utiles pour comprendre le reste du cours. Au chapitre 3, nous introduisons les principales techniques utilisées pour implanter les langages impératifs, et qui forment en fait le socle commun à la plupart des langages de haut niveau : passage de paramètres, pile d'exécution et l'accès aux variables locales et non-locales. Le chapitre 4 s'intéresse aux langages de programmation fonctionnelle, principalement ceux pratiquant l'ordre applicatif du lambda-calcul, en opposition à l'ordre normal que nous n'évoquerons que brièvement. Nous couvrirons aussi dans ce chapitre la question de la gestion de la mémoire et de la récupération automatique de la mémoire allouée dynamiquement. Au chapitre 5, finalement, nous verrons les principaux aspects de l'implantation des langages à objets à travers deux cas, soient ceux de C++ et Smalltalk.

# Chapitre 2

# Langages de programmation

Dans ce chapitre, nous rappelons ce qui forme un langage de programmation et les différentes approches utilisées pour implanter les langages de haut niveau.

## 2.1 Qu'est-ce qu'un langage de programmation ?

Un langage de programmation vise à fournir un modèle de calcul de plus haut niveau, plus facile à comprendre. Cette facilité de compréhension vise à permettre :

1. l'écriture de programmes corrects,
2. l'entretien de ce logiciel par des modifications successives visant à corriger les fautes de programmation ou pour accomoder de nouveaux besoins des utilisateurs, et
3. la réutilisation de portions de programmes pour réaliser plus vite et de façon moins coûteuse de nouveaux programmes.

Un langage de programmation se définit en quatre grandes parties :

- Son vocabulaire, sa reconnaissance étant confiée à l'analyseur lexicographique.
- Sa grammaire, sa reconnaissance étant confiée à l'analyseur syntaxique.
- Ses règles sémantiques, la vérification de leur observance étant confiée à l'analyseur sémantique.
- Sa signification, dont la réalisation est obtenue par traduction du code de haut niveau vers du code machine.

Le vocabulaire d'un langage est constitué de constantes, de mots-clés, de symboles de ponctuation, d'identificateurs, etc. Leur description est souvent donnée sous forme d'expressions régulières. Des outils comme `lex` et `flex` transforment automatiquement une spécification des unités lexicales d'un langage sous formes d'expressions régulières en un analyseur lexicographique.

La grammaire d'un langage définit la forme des énoncés et des expressions. Sa description est souvent donnée sous forme d'une grammaire. Des outils comme `yacc` et `bison` transforment automatiquement une spécification de la syntaxe d'un langage sous forme d'une grammaire en un analyseur syntaxique.

La sémantique statique d'un langage exprime un certain nombre de règles sémantiques que doivent observer les programmes pour être considérés corrects. Typiquement, la congruence entre paramètres réels et paramètres formels d'une fonction est une de ces règles (même nombre d'arguments à l'appel de la fonction qu'elle a de paramètres dans sa définition). De même, la concordance des types fait partie de la sémantique statique. Il existe une grande variété de règles de sémantique statique, et par conséquent plusieurs outils sont utilisés pour la définir. Un des plus courants sont les grammaires attribuées.

La sémantique dynamique d'un langage exprime la signification en terme de résultat du programme lors de son exécution. Il existe plusieurs façons d'exprimer cette sémantique, les plus courantes étant des descriptions en langage naturel et des implantations concrètes (compilateur et/ou interprète). Mais de plus en plus, on voit apparaître des descriptions formelles, dont la sémantique opérationnelle structurelle, la sémantique dénotationnelle et la sémantique axiomatique. Nous utiliserons dans les pages qui viennent l'approche des interprètes et des descriptions en langage naturel.

Pour l'essentiel, nous supposons connus les principaux éléments constituant un langage :

- Constantes : nombres, caractères, chaînes de caractères, pointeurs.
- Types de données structurés : tableaux, structures, listes, etc.
- Expressions : arithmétiques, booléennes, etc. Elles sont exécutées pour leur résultat.
- Fonctions : l'abstraction de l'expression.
- Énoncés : affectation, boucles, alternatives, etc. Ils sont exécutés pour leurs effets sur les données.
- Procédures : l'abstraction de l'énoncé.

Par ailleurs, trois notions auront une importance majeure dans le reste du cours. Les deux premières sont la portée et la durée de vie des identificateurs. La portée des identificateurs et les constructions syntaxiques (fonctions, procédures, blocs, modules, objets, etc.) permettant de la limiter sont à l'origine de plusieurs des problèmes d'implantation que nous allons discuter. La fonction et la procédure paramétrise un calcul et les identificateurs utilisés comme paramètres formels et comme variables locales ont une portée qui se limite à l'intérieur de la fonction ou de la procédure. Leur durée de vie diffère cependant selon les langages. En Fortran IV, la durée de vie des variables locales est l'exécution entière du programme. En C et en Pascal, elle se limite à la durée de l'activation de la fonction qui les déclare et une nouvelle incarnation de chaque variable est créée pour chaque activation de la procédure (on dit qu'elles ont une durée de vie dynamique). Cela permet la récursivité. Les langages comme Scheme et Smalltalk permettent la création de *fermetures* capables de capturer les variables locales et pouvant être retournées comme résultat de la fonction. Ainsi, les variables ont une durée de vie illimitée. Ces phénomènes induisent des contraintes très précises en termes d'implantation, et dont les coûts ne sont pas les mêmes.

La troisième notion est celle de syntaxe abstraite. On distingue deux types de syntaxe pour décrire un programme. La syntaxe concrète définit l'agencement des unités lexicales qui forment les énoncés et les expressions du langage. La syntaxe concrète sert uniquement à analyser les programmes pour déterminer s'ils sont correctement construits. La syntaxe abstraite définit la structure du programme. Par exemple, comment une expression est formée de sous-expressions, celles-ci étant formées de l'application d'un opérateur sur des opérandes, ou d'un appel de fonction. La syntaxe abstraite se définit sous forme d'arbres et elle ne sert

qu'à exprimer la structure des programmes et le type des constructions utilisées.

De façon générale, on utilise la syntaxe concrète pour analyser les programmes, et au cours de l'analyse, on construit l'arbre de syntaxe abstraite représentant la structure du programme analysé. L'arbre de syntaxe abstraite est ensuite utilisé pour les phases subséquentes du traitement, c'est-à-dire l'analyse sémantique et la génération de code dans un compilateur. Lorsque l'on veut parler de l'ensemble des structures de programme que l'on peut écrire dans un langage, il est utile de parler de l'ensemble des arbres de syntaxe abstraite correctement formés. Pour définir l'ensemble de ces arbres, on utilise des grammaires abstraites.[1]

## 2.2 Implantation

Aujourd'hui la quasi-totalité des matériels informatiques sont alignés sur un standard *de facto* qui est essentiellement une variante plus ou moins complexe de la machine de Von Neumann. Ces processeurs se programment en *langage machine*, les programmes étant formés d'un séquence inintelligible de chiffres binaires. La programmation en langage machine est une tâche impossible pour un humain. Les langages d'assemblage, collection d'instructions mnémoniques directement traductibles en instructions du langage machine, permettent une programmation humaine, mais ils sont essentiellement tournés vers la machine, et non vers les problèmes à résoudre. De plus, chaque processeur ayant son propre langage machine et donc son propre langage d'assemblage, ses programmes ne sont pas exécutables sur les autres types de processeurs. Les langages de programmation de haut niveau, dont C, Pascal, Modula-2, Scheme, ML, C++ et Smalltalk en sont quelques exemples courants, visent à s'abstraire de la machine, et même de toute machine particulière.

Pour atteindre cet objectif, le langage de programmation offre un modèle de programmation permettant d'exprimer facilement les solutions aux problèmes. Il offre également des outils (types de données abstraits, etc.) et des opérations d'un haut niveau d'abstraction, ce qui évite de les reprogrammer dans chaque application. En résumé, l'implantation d'un langage de programmation consiste à réaliser un processeur de langage qui réorganise la machine sous-jacente et construit une nouvelle machine, virtuelle celle-là, sur laquelle les programmes écrit dans ce langage peuvent être exécutés. Bien sûr, une machine virtuelle, comme son nom l'indique, ne peut exécuter réellement les programmes. Seules les machines réelles peuvent le faire.[2] Il faut donc traduire les programmes du langage de haut niveau vers le langage machine seul reconnu par les machines réelles. Cette traduction implique le décodage des expressions et énoncés du langage de haut niveau, puis leur correspondance vers des instructions machines qui sont ensuite exécutées. La traduction est confiée à un *processeur de langage* et elle peut se faire de plusieurs façons :

- Le décodage peut être complètement entrelacé avec la correspondance vers le code machine et l'exécution de ce code, auquel cas on dire que le processeur de langage est un *interpréteur*.

[1] Cela est d'ailleurs source de confusion entre syntaxe abstraite et syntaxe concrète, puisqu'on utilise aussi des grammaires pour définir cette dernière, mais alors il s'agit de grammaires concrètes.

[2] Il est tout à fait possible, et cela a été fait à plusieurs reprises, d'implanter physiquement un processeur de langage de haut niveau. En pratique, cette approche n'a que rarement connu un succès commercial. Un des exemples les plus intéressants fut celui des machines Lisp de la compagnie Symbolics qui fournissaient des architectures dédiées à l'exécution du langage Lisp.

- Le décodage et la correspondance vers le code machine peuvent être faits avant l'exécution du programme, auquel cas on dira que le processeur de langage est un *compilateur*.
- Le décodage et la correspondance peuvent faire intervenir un *langage intermédiaire*, de plus bas niveau, qui lui est interprété, auquel cas on dira que le processeur de langage est un *compilateur interprétant*.
- Le décodage et la correspondance vers le code machine peuvent être faits au besoin, lors de l'exécution par interruption momentanée de cette exécution, qui est reprise après, auquel cas on dira que le processeur de langage est un *compilateur en flux tendu*.[3]

Il est rare que l'implantation d'un langage, par la conception de ce dernier, soit condamnée à utiliser l'une ou l'autre de ces approches. Il est tout à fait possible d'interpréter un langage aussi lié à l'idée de compilation que C. Par ailleurs, si Lisp, Scheme et Prolog, voire même Smalltalk, ont la réputation d'être des langages interprétés, en réalité des compilateurs très efficaces existent pour les trois premiers alors que le dernier est implanté selon la technique de compilation en flux tendu. Par contre, les techniques d'implantation diffèrent en fonction des propriétés de flexibilité qu'elles offrent au programmeur. L'interprète permet souvent d'être très réactif aux changements du code, car il n'a pas besoin de recompiler et relier l'ensemble du programme. Cette caractéristique se retrouve également dans l'approche des compilateurs à flux tendu. Par contre, la compilation globale, supposant un monde complet et fermé, permet des optimisations que les autres approches, surtout l'inteprétation, ne permettent pas. La performance de l'interprétation est bien sûr diminuée du fait que le décodage des expressions est fait à l'exécution et même lors d'une exécution particulière d'un programme, une expression sera décodée répétivement à chaque fois qu'elle doit être exécutée.

[3] *Just in time compiler.*

# Chapitre 3

# Programmation impérative

## 3.1 La machine de Von Neumann

Bien que de nombreuses entorses apparaissent de plus en plus, la machine de Von Neumann sert donc de modèle de base à la plupart des micro-processeurs courants. Son principe est simple, et bien connu :

- La machine est constitué de quatre composantes :
  1. une unité centrale arithmétique et logique chargée du décodage et de l'exécution des instructions en langage machine ; pour cette exécution, l'unité centrale dispose d'un petit nombre de registres généraux permettant de sauvegarder des valeurs d'une instruction à l'autre ;
  2. une mémoire centrale à accès direct (chaque cellule est adressable par un entier dénotant sa position relative, comme un vecteur) ; cette mémoire contient des données et des instructions (programmes en langage machine) et l'unité centrale peut lire et écrire des données depuis et dans chacune de ces cellules ;
  3. un canal d'entrée sur lequel l'unité centrale peut lire des données ;
  4. un canal de sortie sur lequel l'unité centrale peut écrire des données.
- L'unité centrale dispose également d'un petit nombre de registres spécialisés contenant des informations utiles à son fonctionnement. En particulier, il dispose d'un registre de programme (*program counter* ou PC) qui indique la cellule mémoire contenant la prochaine instruction machine à exécuter.
- Lorsque lancée, l'unité centrale exécute tant qu'une instruction spéciale (`halt`, par exemple) n'est pas atteinte, le traitement suivant :
  1. Charger l'instruction dans la cellule mémoire désignée par le PC.
  2. Incrémenter le PC de 1.
  3. Exécuter l'instruction chargée.

Ce qui caractérise l'état d'exécution d'un programme sur l'architecture de Von Neumann est le contenu de la mémoire, des registres de l'unité centrale et ces canaux d'entrée/sortie. L'effet d'une instruction se mesure en termes de modifications que son exécution fait à cet état d'exécution.

Ce modèle de bas niveau transparaît pour l'essentiel dans les langages impératifs (C, Pascal, Ada83, Modula-2, etc.). La réorganisation de la machine de Von Neumann réalisée par ces langages se limite à l'introduction de nouveaux types de données, à l'introduction de variables pour s'abstraire des emplacements précis en mémoire, à l'introduction de constructions linguistiques limitant la portée des identificateurs et l'abstraction de calculs (fonctions et procédures), et à l'introduction d'un flôt de contrôle dirigé par la syntaxe (la programmation structurée qui nous vaut aujourd'hui les constructions `if`, `while`, `for`, etc. plutôt que les `gotos` de naguère...).

Bien sûr, tous ces concepts sont extrêmements utiles, et ont fait en sorte que nos programmes impératifs sont baucoup plus compréhensibles. Reste néanmoins que l'exécution d'un programme impératif se comprend essentiellement selon le même schéma que la machine de Von Neuman : on exécute les instructions une à une et l'effet de chacune se comprend essentiellement en terme de modification qu'elle fait à la valeur des variables, qui nous abstraient de la mémoire et des registres.

Passons sur l'implantation des différents énoncés structurant le flôt de contrôle. Sachant que le jeu d'instructions typique d'un micro-processeur comprend des instructions de branchement conditionnel et inconditionnel, il est assez simple de comprendre comment la traduction peut se faire. Les astuces précises dépendent du jeu d'instruction disponible et est l'affaire des spécialistes de la phase génération de code machine du compilateur.

## 3.2 Appel de procédure

L'idée essentielle de la fonction et de la procédure est l'abstraction et la paramétrisation d'un certain calcul ou d'un certain traitement. Pour calculer la longueur de l'hypothénuse d'un triangle rectangle de côtés 3 et 4, on écrit[1] `(** (+ (** 3 2) (** 4 2)) 0.5)`, on peut abstraire et paramétriser ce calcul sous la forme d'une fonction :

¶rint (defun hypothenuse (a b) (** (+ (** a 2) (** b 2)) 0.5))

L'intérêt de ce genre d'abstraction est multiple. D'une part, elle permet, une fois écrite, de l'utiliser dans le reste du programme comme une opération de haut niveau sans plus avoir à se sourcier des détails de son implantation. Elle permet aussi de cacher l'implantation d'une opération en un seul endroit. Si cette implantation venait à devoir être changée, le changement se limitera à cette seule procédure ou fonction, et non à l'ensemble des points de programme où l'opération est utilisée. Les fonctions et procédures mettent en œuvre un principe de décomposition descendante facilitant la réalisation des programmes ; c'est la bonne vieille stratégie du «*diviser pour régner*». Enfin, elles permettent la création de bibliothèques d'opérations réutilisables d'un programme à l'autre.

Une procédure est constituée :

- d'un nom,
- d'une liste de paramètres formels, un paramètre formel étant un nom auquel est éventuellement associé un type dans les langages typés,
- d'une liste de déclaration de variables locales, et

[1] Le langage utilisée est librement inspiré de Scheme, mais surtout correspond assez bien à celui qui est défini par notre interprète un peu plus loin.

**TP 1.** Évaluation des expressions arithmétiques

Soit une machine à pile définie par les instructions suivantes :

| Inst. | Signification | Cond. | Résultat |
|---|---|---|---|
| `add` | STORE[SP-1] := STORE[SP-1] + STORE[SP]<br>SP := SP - 1 | (N,N) | (N) |
| `sub` | STORE[SP-1] := STORE[SP-1] - STORE[SP]<br>SP := SP - 1 | (N,N) | (N) |
| `mul` | STORE[SP-1] := STORE[SP-1] * STORE[SP]<br>SP := SP - 1 | (N,N) | (N) |
| `div` | STORE[SP-1] := STORE[SP-1] / STORE[SP]<br>SP := SP - 1 | (N,N) | (N) |
| `ldo` q | SP := SP + 1 ; STORE[SP] := STORE[q] | q valide | (N) |
| `ldc` q | SP := SP + 1 ; STORE[SP] := q | type(q) = T | (T) |
| `ind` | STORE[SP] := STORE[STORE[SP]] | (A) | (T) |
| `sro` q | STORE[q] := STORE[SP] ;<br>SP := SP - 1 | (T)<br>q valide | |
| `sto` | STORE[STORE[SP-1]] := STORE[SP] ;<br>SP := SP - 2 | (A,T) | |
| `halt` | arrête la machine | | |

À partir du code d'analyse des expressions arithmétiques qui vous est fourni et qui produit la syntaxe abstraite suivante, proposez une génération de code qui permet d'exécuter l'expression sur la machine à pile spécifié ci-haut.

$$
\begin{array}{rcl}
p & ::= & e \\
e & ::= & c \mid e + e \mid e - e \mid e * e \mid e \,/\, e \\
c & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

Considérez ensuite le langage obtenu en ajoutant les variables à ce petit langage des expressions arithmétiques.

$$
\begin{array}{rcl}
p & ::= & s^* \\
s & ::= & id = e \\
e & ::= & id \mid c \mid e + e \mid e - e \mid e * e \mid e \,/\, e \\
c & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
id & ::= & a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid \\
 & & r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z
\end{array}
$$

Pour le traitement des variables, il faut allouer à chaque variable un emplacement mémoire qui sera ensuite utilisé pendant la génération pour référer à la variable.

- d'un corps.

Une fonction est comme une procédure à ceci près qu'elle retourne une valeur ; dans les langages typés, un type de retour est associé à la déclaration de la fonction. Pour alléger le texte, sauf indication particulière nous allons parler de procédure pour désigner à la fois fonction et procédure dans le texte qui suit. L'appel d'une procédure est constituée :

- du nom identifiant la procédure à appeler, et
- d'une liste de paramètres réels, c'est-à-dire les valeurs qui seront désignées par les paramètres formels à l'intérieur du corps de la fonction.

Un programme déclare des procédures. Cette déclaration sert d'abord à associer un nom à la définition d'une procedure. Cette association permet également d'accumuler de l'information sur la procédure (nombre et type des paramètres formels, type du résultat pour une fonction, etc.) servant à vérifier que les appels apparaissant par ailleurs dans le programme se conforment à la définition. À l'exécution, un appel de procédure mène à l'activation de la procédure concernée. À la base, l'activation implique la rupture de la séquence d'énoncés en cours d'exécution par un branchement vers le code de la procédure appelée. À la fin de l'exécution de la procédure, le contrôle revient au point venant immédiatement après l'appel. Ces changements de contexte d'exécution s'accompagnent d'une gestion des variables accessibles, qui sont généralement différentes entre le point d'appel et l'intérieur de la procédure.

Grosso modo, lors de l'activation, il faut évaluer les paramètres réels pour les passer à l'appelé. Ensuite, il faut sauvegarder l'information sur le point d'appel de façon à la restaurer lors du retour. Cette information inclut principalement l'adresse de la prochaine instruction à exécuter au retour ainsi que les variables visibles et leur état respectif. Ensuite, il faut installer l'information nécessaire à l'exécution de la procédure, comme les variables locales. Enfin, il faut brancher à la première instruction de la procédure. Au retour, il faut récupérer la mémoire utilisée pour les informations de la procédure appelée, revenir à l'instruction suivante dans le contexte appelant, en prenant soin de récupérer le résultat si la procédure appelée est en fait une fonction.

L'ensemble de l'information nécessaire à l'exécution d'une procédure est rassemblée dans une structure de données appelée *bloc d'activation*. Le bloc d'activation contient :

- de l'information de contrôle interne dont la nature et la quantité dépend du langage (voir plus loin), comme par exemple l'adresse de retour dans le programme et l'adresse du bloc d'activation de l'appelant ;
- de l'information concernant le contexte appelant qu'il faudra remettre en place lors du retour, comme par exemple des valeurs de registres ;
- les paramètres réels de l'appel ;
- s'il s'agit d'une fonction, un espace destiné à contenir le résultat ;
- les variables locales à la procédure ;
- de l'espace pour les «variables» temporaires utilisées par les expressions à l'intérieur de la procédure.

À tout instant, le processeur doit connaître l'adresse du bloc d'activation courant. L'accès aux valeurs des paramètres réels, des variables locales et des variables temporaires est généralement réalisé par un adresse de type base plus déplacement, l'adresse du bloc d'activation servant d'adresse de base et le déplacement étant calculé pour chaque variable à l'intérieur du bloc (notez que cela peut se faire statiquement, dès la compilation).

## 3.3 Passage de paramètres

Dans la section précédente, nous avons volontairement évité plusieurs questions que nous allons maintenant aborder. En premier lieu, nous avons parlé des paramètres réels sans insister sur le *mode de passage* de ces paramètres. Plusieurs options s'offrent au concepteur de langage. Que doit-on passer ? La valeur du paramètre réel ? Son emplacement mémoire ? L'expression permettant de le calculer ? En fait, quatre grands modes de passage de paramètres ont été identifiés :

1. *Par valeur* : l'expression donnée en paramètre réel est évaluée et la valeur de son résultat est passée à la procédure pour être placée dans l'emplacement mémoire réservé au paramètre formel correspondant.
2. *Par référence* : l'expression donnée en paramètre réel est évaluée mais c'est l'emplacement mémoire de son résultat qui est passé à la procédure ; lorsqu'une affectation au paramètre formel sera faite dans la procédure, elle modifiera par indirection la valeur de l'emplacement mémoire passé en paramètre réel.
3. *Par valeur/résultat* : l'expression donnée en paramètre réel est évaluée et à la fois la valeur et l'emplacement mémoire de son résultat seront passés à la procédure ; la procédure s'exécute exactement comme en passage par valeur, sauf qu'au retour la valeur finale du paramètre formel est recopiée dans l'emplacement mémoire passé en paramètre.
4. *Par nom* : l'expression donnée en paramètre réel n'est pas évaluée immédiatement ; l'expression non-évaluée est passée à la procédure, et elle ne sera évaluée que lors de son utilisation dans la procédure.

### 3.3.1 Passage par valeur

De ces quatre formes, les deux premières sont les plus utilisées. C utilise par exemple le passage par valeur. En passage par valeur, soit une procédure `p` de paramètre formel `x` écrite en C :

```
int p(int x) {
  ...
  return ... x ...;
}

void main() {
  int i;
  ...
  i = p(x);
  ...
}
```

On peut simuler l'effet du passage par valeur en transformant `x` en variable globale[2], l'appel de cette procédure `p(z)` peut être vu de la façon suivante :

[2]Cet exercice ne vise qu'à caractériser l'effet du mode de passage des paramètres. Il néglige évidemment la question de la portée et de la durée de vie de la variable.

```
int x;

int p() {
  ...
  return ... x ...;
}

void main() {
  int i;
  ...
  x = z;
  i = p();
  ...
}
```

### 3.3.2 Passage par référence

L'appel par valeur est un mode de passage très courant et qui a l'avantage de la simplicité, de la prévisibilité des effets, et de l'encapsulation. Il est cependant restreint dans ce qu'il permet, et peut être coûteux lorsque les valeurs à passer sont des structures de données complexes (par exemple un tableau qui devra être recopié dans le paramètre formel, ce qui est coûteux en espace et en temps). Prenons le cas d'une procédure `swap` qui échange les valeurs de deux variables :

```
void swap(int a, int b) {
  int tmp;

  tmp = a; a = b; b = tmp;
}

void main() {
  int x, y;

  ...
  swap(x,y);
  ...
}
```

L'application du principe de la «traduction» proposée ci-haut montre bien que cette procédure en passage par valeur n'aura pas l'effet escompté :

```
int a, b;

void swap() {
  int tmp;

  tmp = a; a = b; b = tmp;
}

void main() {
```

```
  int x, y;

  ...
  a = x; b = y;
  swap();
  ...
}
```

Au retour de la fonction, on a bel et bien échangé les valeurs de `a` et `b`, mais pas celles de `x` et `y` qui demeurent inchangées. Par contre, on peut écrire `swap` en passage par référence, car les affectations dans la fonction vont alors directement modifier les emplacements mémoire des variables passées en paramètres réels. On peut donner en C un «traduction» simulant le passage par référence en utilisant les pointeurs :

```
int * a, b;

void swap() {
  int tmp;

  tmp = *a; *a = *b; *b = tmp;
}

void main() {
  int x, y;

  ...
  a = &x; b = &y;
  swap();
  ...
}
```

D'ailleurs, s'il faut écrire en C une procédure capable de modifier un paramètre réel, on utilisera les pointeurs. La procédure `swap` par exemple s'écrit et s'utilise comme suit :

```
void swap(int *a, int *b) {
  int tmp;

  tmp = *a; *a = *b; *b = tmp;
}

void main() {
  int x, y;

  ...
  swap(&x,&y);
  ...
}
```

### 3.3.3 Passage par valeur/résultat

L'appel par valeur/résultat est présent en Ada, et son utilité est probablement mieux comprise dans le contexte d'un appel à distance ($RPC$) dans un système réparti. Il peut aussi être simulé à l'aide de pointeurs. L'idée est de transformer chaque paramètre formel en deux variables, l'une du type du paramètre formel et l'autre du type pointeur sur le type du paramètre formel. On copie alors le paramètre réel dans la variable représentant la valeur du paramètre formel, puis on range le pointeur sur le paramètre réel dans la variable pointeur associée au paramètre formel. Au retour, les variables pointeurs sont utilisées pour recopier la valeur finale des paramètres formels dans les paramètres réels.

Il peut sembler *a priori* que le passage par référence et le passage par valeur/résultat sont parfaitement équivalents. Il n'en est rien. Des programmes peuvent avoir des résultats différents lorsqu'exécutés en passage par référence et en passage par valeur résultat. Le prochaine exemple vous en propose un. En général, ce genre de comportement se produit lorsque des alias[3] sont introduits. Il n'est pas toujours facile de s'en rendre compte dans les programmes complexes, et les erreurs qui s'en suivent peuvent être très difficiles à trouver.

**Exemple.** [Set96, pages 159–160] Soit le programme suivant :

```
int i, j;

void foo(int x, int y) {
  i = y;
}

void main() {
  i = 2; j = 3;
  foo(i,j);
  printf("%d %d\n", i, j);
}
```

En passage par référence, ce programme se termine avec la valeur 3 dans la variable `i`, alors qu'en passage par valeur/résultat, il se termine avec la valeur 2 dans `i`, comme le montre la simulation suivante :

```
int i, j;
int x, y;
int * px, py;

void foo() {
  i = y;
}

void main() {
  i = 2; j = 3;
  x = i; y = j;
```

[3] On a des alias lorsque plusieurs noms désignent le même emplacement mémoire. Cela peut arriver lorsqu'un pointeur est pris sur une variable ou encore, comme dans le cas suivant, lorsqu'une variable globale (donc visible dans la procédure est passée en paramètre réel.

```
  px = &i; py = &j;
  foo();
  *px = x; *py = y;
  printf("%d %d\n", i, j);
}
```

Dans la procédure `foo`, si l'affectation change bien la variable globale `i`, la recopie au retour lui redonne sa valeur initiale contenue dans le paramètre formel `x`. □

### 3.3.4 Passage par nom

Pour ce qui est du passage par nom, la difficulté principale tient au fait que l'expression dont l'exécution est retardée doit, au besoin, être exécutée dans le contexte de l'appelant. L'idée étant, pour l'essentiel, de substituer au paramètre formel le texte du paramètre réel, il ne faut pas que des variables du texte du paramètre réel soient capturées par des déclarations internes à la procédure appelée. Considérez le programme suivant [Set96] :

```
int p(int x) {
  int i;

  i = 1;
  return x + i;
}

void main() {
  int i, j;
  ...
  i = 3;
  j = p(A[i]);
}
```

Si on substitue simplement `A[i]` pour `x` dans la procédure `p`, la variable `i`, qui devrait référer au `i` déclaré dans `main`, va être capturée par le `i` déclaré dans `p`. Pour éviter ce problème, il faut trouver le moyen d'évaluer le texte `A[i]` à l'endroit où `x` est référencé dans `p`, mais dans le contexte de l'appelant, ici `main` (voir le TP2).

## 3.4 Un premier interprète

Notre premier interprète implante une variante grandement simplifiée des principaux choix de conception de Fortran IV, un langage qui date du début des années 60. L'interprète est lui-même simplifié puisque nous ne représentons pas explicitement le flôt de contrôle dans le programme interprété ; ce flôt de contrôle est implicitement pris en compte par le flôt de contrôle de l'exécution de l'interprète lui-même, ce qui est l'approche standard dans les interprètes. Cette simplification, couplée au choix de conceptions de Fortran IV fait en sorte que le bloc d'activation n'apparait pas explicitement pour le moment. On le réintroduira plus tard.

$p \in$ *Programme*
$d \in$ *Definition*
$i \in$ *Identificateur*
$e \in$ *Expression*
$c \in$ *Constante*

$p ::=$ `(prog` $d^*$ $e$`)`
$d ::=$ `(defun` $i$ `(` $i^*$ `)` $e$`)` | `(define` $i$ $e$`)`
$e ::=$ $c$ | $i$ | `(+` $e$ $e$`)` | `(-` $e$ $e$`)` | `(*` $e$ $e$`)` | `(/` $e$ $e$`)` | `(=` $e$ $e$`)` | `(>` $e$ $e$`)` |
`(if` $e$ $e$ $e$`)` | `(begin` $e^*$`)` | `(funcall` $i$ $e^*$`)` | `(set!` $i$ $e$`)` | `(while` $e$ $e$`)` |
`(print` $e$`)`

FIG. 3.1 – Syntaxe abstraite de notre mini-langage

La figure 3.1 présente la syntaxe abstraite de notre mini-langage. Un programme (forme syntaxique `prog`) est constitué d'une liste de déclarations de fonctions (forme `defun`) et de variables globales (forme `define`). Une fonction comporte un nom, une liste de paramètres formels et un corps. Nous ne prévoyons pas pour le moment de variables locales ; elles seront ajoutées par la suite (voir le TP2). Une variable globale possède un nom et sa valeur initiale est donnée par une expression. Les expressions sont en réalité un mélange d'énoncés au sens de la programmation impérative et d'expressions réelles retournant un résultat significatif. Les expressions artihmétiques et de comparaisons sont bien connues. L'expression `if` retourne comme résultat le résultat de son expression en partie *then* ou *else* selon le résultat de son expression test. L'expression `begin` permet de regrouper une séquence d'expressions, à exécuter l'une après l'autre, et le résultat est celui de la dernière expression. L'expression `funcall` représente l'appel de fonction. Les expressions `set!`, `while` et `print` sont plutôt des énoncés au sens classique du terme et à ce titre retournent une valeur spéciale indéfinie sur laquelle aucun calcul n'est possible.

### 3.4.1 Choix de conception et d'implantation

Fortran IV est un langage assez fruste du point de vue de son implantation. En particulier, il exige que l'allocation des emplacements mémoire de toutes les variables du programme soit faite à la compilation. Cette exigence a de nombreuses conséquences sur les capacités du langage. Puisque chaque paramètre formel de chaque procédure se voit allouer un et un seul emplacement mémoire, il ne peut y avoir en tout temps qu'une seule activation de chaque procédure. Cela exclut donc la récursivité.

Une seconde restriction est le fait que les fonctions doivent toutes être déclarées à la compilation. Il est interdit de déclarer des fonctions locales à d'autres fonctions. On a donc un univers plat où toutes les fonctions sont globales et où les seules variables visibles à une fonction sont ses propres paramètres formels et variables locales, et les variables et fonctions globales. Cette restriction, comme nous le verrons plus loin à la section 3.5.3, permet de passer les fonctions

```
(define (init-store)
  (vector 0 (make-vector store-size the-non-initialized-marker)))

(define (store address value st)
  (begin
    (vector-set! (vector-ref st 1) address value)
    st))

(define (store-list address value* st)
  (if (null? value*)
    st
    (begin
      (store address (car value*) st)
      (store-list (+ address 1) (cdr value*) st))))

(define (stored address st)
  (vector-ref (vector-ref st 1) address))

(define (allocate n st)
  (let ((return (vector-ref st 0)))
    (if (< (+ return n) (vector-length (vector-ref st 1)))
      (begin
        (vector-set! st 0 (+ return n))
        return)
      (error "store overflow" st))))
```

FIG. 3.2 – La mémoire

en paramètres réels et les retourner en résultat d'autres fonctions. Pour le moment cependant, il n'y a pas de possibilité d'appel calculé ; l'appel de fonction `funcall` exige d'utiliser directement le nom de la fonction à appeler. Les fonctions ne peuvent pas non plus être créées à l'exécution.

### 3.4.2 Gestion de la mémoire

La mémoire est essentiellement un vecteur d'emplacements mémoire accessibles directement à l'aide d'une adresse. Dans notre interprète, l'espace mémoire en tant que tel est représentée par un vecteur Scheme et les adresses sont les indices de ce vecteur. La mémoire, quant à elle doit savoir où allouer l'espace lorsque cela est nécessaire. On représente donc la mémoire comme un vecteur à deux entrées :

1. l'adresse de l'espace suivant où allouer, en considérant qu'on alloue l'espace linéairement et une fois pour toute (ce qui est juste ce qu'il faut dans le cas présent),
2. le vecteur d'emplacements mémoire ; un emplacement mémoire non-encore utilisé ou non encore initialisé est marqué par une valeur spéciale (`the-non-initialized-marker`).

La figure 3.2 présente l'implantation de la mémoire utilisée par notre interprète.

La correspondance entre un identificateur et la valeur qui lui est associée se fait en deux temps : un première correspondance se fait entre l'identificateur et l'adresse de l'emplacement

```
(define (initialize-env id*)
  (map (lambda (id) (cons id the-non-initialized-marker)) id*))

(define (extend-env id* value* env)
  (append (map2 (lambda (x y) (cons x y)) id* value*) env))

(define (lookup id env)
  (if (pair? env)
    (if (eq? (caar env) id)
      (cdar env)
      (lookup id (cdr env)))
    (error "No such binding" id)))

(define (update! id env value)
  (if (pair? env)
    (if (eq? (caar env) id)
      (begin (set-cdr! (car env) value)
             value)
      (update! id (cdr env) value))
    (error "No such binding" id)))
```

FIG. 3.3 – L'environnement

mémoire qui lui est alloué et une seconde se fait entre l'adresse et la valeur contenue dans l'emplacement mémoire. La première correspondance se fait à l'aide d'un environnement alors que la seconde est réalisée par la mémoire elle-même. En tout point de programme, l'environnement est tout simplement une association entre les identificateurs visibles en ce point de programme et les adresses des emplacements mémoire qui leur ont été alloués. Une référence à une variable se résout donc en recherchant dans l'environnement l'adresse de l'emplacement mémoire correspondant, puis en accédant l'emplacement mémoire dans la mémoire avec l'adresse trouvée.

La figure 3.3 présente l'implantation des environnements utilisée par notre interprète. La fonction `initialize-env` crée un nouvel environnement en liant chaque nom de variable de la liste passée en arguments à une valeur spécifique dénotant la non-initialisation. La fonction `extend-env` prend une liste d'identificateurs de variable et une liste de valeurs et ajoute les liaisons ainsi obtenues à l'environnement passé en argument. La fonction `lookup` recherche la valeur associée à l'identificateur `id` dans l'environnement `env`. La fonction `update!` change la valeur associée à l'identificateur `id` dans l'environnement `env` pour qu'elle soit la valeur `value`.

Notons que lorsqu'on compile un programme, la décision concernant l'allocation des emplacements mémoire est prise au moment de la compilation. La partie environnement est donc construite pendant la compilation et utilisée pendant la génération de code, mais à l'exécution le code généré contient directement les adresses des emplacements mémoires alloués ; l'environnement a disparu. En fait, l'environnement dans un compilateur est plutôt appelé *table des symboles* et il contient non seulement les emplacements mémoire alloués mais aussi l'information nécessaire aux vérifications de la sémantique statique du programme.

```
(define (eval-programme p)
  (let ((global-env (initialize-env (map (lambda (d) (snd d)) (snd p))))
        (st         (init-store)))
    (begin
      (eval-definition* (snd p) global-env st)
      (eval-expression (thd p) global-env st))))

(define (eval-definition* d* env st)
  (cond ((null? d*) env)
        (else       (begin
                      (eval-definition (car d*) env st)
                      (eval-definition* (cdr d*) env st)))))

(define (eval-definition d env st)
  (cond
    ((defun? d)  (eval-defun d env st))
    ((define? d) (eval-define d env st))))

(define (eval-defun d env st)
  (let* ((n-formal-par (length (thd d)))
         (base-address (allocate n-formal-par st))
         (new-env      (extend-env (thd d)
                                   (interval base-address n-formal-par)
                                   env))
         (fonction (lambda (args dyn-st)
                     (begin
                       (store-list base-address args dyn-st)
                       (eval-expression (fth d) new-env dyn-st)))))
    (update! (snd d) env fonction)))

(define (eval-define d env st)
  (let ((address  (allocate 1 st)))
    (store address (eval-expression (thd d) env st) st)
    (update! (snd d) env address)))
```

FIG. 3.4 – Évaluation de la partie définition du programme

### 3.4.3 Évaluation des définitions

L'évaluation des définitions se fait avant l'exécution du corps du programme. Chaque définition est évaluée en séquence, et l'évaluation de l'expression d'initialisation ne doit pas nécessiter l'évaluation d'une référence à une variable non encore initialisée. En pratique, cela veut dire qu'une définition de variable globale ne peut utiliser dans son expression d'initialisation que des variables déjà définies, alors que les définitions de fonctions, parce que le corps d'une fonction est évalué lors de l'appel et non lors de la définition, peuvent contenir des références à n'importe quel identificateur global, sans pour autant inclure d'appels récursifs directs ou indirects que nous avons déjà exclus.

Tout d'abord, l'environnement global est initialisé pour contenir des liaisons non-initialisées pour chacune des variables globales et chacunes de fonctions. C'est le résultat obtenu par

```
(define (eval-expression e env st)
  (cond
    ((constante? e)      e)
    ((identificateur? e) (stored (lookup e env) st))
    ((addition? e)       (let* ((v1 (eval-expression (snd e) env st))
                                (v2 (eval-expression (thd e) env st)))
                           (+ v1 v2)))
    ((soustraction? e)   (let* ((v1 (eval-expression (snd e) env st))
                                (v2 (eval-expression (thd e) env st)))
                           (- v1 v2)))
    ((multiplication? e) (let* ((v1 (eval-expression (snd e) env st))
                                (v2 (eval-expression (thd e) env st)))
                           (* v1 v2)))
    ((division? e)       (let* ((v1 (eval-expression (snd e) env st))
                                (v2 (eval-expression (thd e) env st)))
                           (/ v1 v2)))
    ((egal? e)           (let* ((v1 (eval-expression (snd e) env st))
                                (v2 (eval-expression (thd e) env st)))
                           (equal? v1 v2)))
    ((pg? e)             (let* ((v1 (eval-expression (snd e) env st))
                                (v2 (eval-expression (thd e) env st)))
                           (> v1 v2)))
    ((if? e)             (eval-if (snd e) (thd e) (fth e) env st))
    ((begin? e)          (eval-begin (cdr e) env st))
    ((set? e)            (begin (store (lookup (snd e) env)
                                       (eval-expression (thd e) env st)
                                       st)
                                the-undefined-marker))
    ((while? e)          (eval-while (snd e) (thd e) env st))
    ((print? e)          (begin (display (eval-expression (snd e) env st))
                                (newline)
                                the-undefined-marker))
    ((funcall? e)        (let* ((v1 (lookup (snd e) env))
                                (v2 (eval-expressions (cddr e) env st)))
                           (v1 v2 st)))))
```

FIG. 3.5 – Évaluation des expressions

l'appel à `initialize-env`. Ensuite, la mémoire est créée. Les définitions sont ensuite évaluées. Pour les variables, c'est-à-dire la forme `define`, il suffit d'évaluer l'expression d'initialisation dans le contexte de l'environnement et de la mémoire dans l'état où ils sont. Ensuite, un espace mémoire est alloué, dans lequel est rangée la valeur initiale, et dont l'adresse est liée à l'identificateur de la variable dans l'environnement global. Pour les fonctions, il faut créer une extension locale de l'environnement global qui contiendra les paramètres formels de la fonction. Les espaces mémoires nécessaires aux paramètres formels sont également alloués à ce moment. La fonction elle-même est créée comme une fermeture Scheme, une fonction recevant les paramètres réels et la mémoire lors de l'exécution, de façon à ranger lors de l'appel les valeurs des paramètres formels dans les espaces alloués statiquement pour évaluer le corps de la fonction dans le contexte de l'environnement étendu localement et de la mémoire

```
(define (eval-if e e1 e2 env st)
  (if (eval-expression e env st)
    (eval-expression e1 env st) (eval-expression e2 env st)))

(define (eval-while e e1 env st)
  (if (eval-expression e env st)
    (begin (eval-expression e1 env st)
           (eval-while e e1 env st))
    the-undefined-marker))

(define (eval-begin e* env st)
  (if (pair? (cdr e*))
    (begin (eval-expression (car e*) env st)
           (eval-begin (cdr e*) env st))
    (eval-expression (car e*) env st)))

(define (eval-expressions e* env st)
  (if (pair? e*)
    (cons (eval-expression (car e*) env st)
          (eval-expressions (cdr e*) env st))
    '()))
```

FIG. 3.6 – Évaluation des expressions spécifiques

dans son état lors de l'appel. Cette fermeture Scheme est liée à l'identificateur de la fonction dans l'environnement (pas dans la mémoire car on ne peut changer la valeur associée à un identificateur de fonction).

### 3.4.4 Évaluation des expressions

L'évaluation des expressions passe d'abord par la fonction `eval-expression` qui, jouant le rôle d'aiguilleur, fait la discrimination entre les différentes formes syntaxiques. Une constante s'évalue simplement à elle-même. En fait cela reflète le choix d'implantation qui est de représenter les constantes du langage défini par celle du langage de définition, c'est-à-dire Scheme. L'évaluation d'une référence à une variable se fait en recherchant dans l'environnement l'adresse de la variable, puis en allant chercher la valeur rangée dans l'emplacement mémoire correspondant. L'évaluation des expressions d'addition, de soustraction, de multiplication, de division, de comparaison d'égalité et de comparaison du plus grand est évidente ; il suffit d'appliquer l'opération correspondante de Scheme au résultat de l'évaluation des sous-expressions. Les expressions `if`, `begin`, et `while` sont déléguées à des fonctions spécifiques sur lesquelles nous revenons plus loin. L'évaluation de l'expression `set !` demande d'évaluer la sous-expression puis de ranger dans l'emplacement mémoire associée à la variable affectée le résultat de la sous-expression. Tout comme les expressions `while` et `print`, le résultat est une valeur indéfinie, implantée de la façon suivante pour être inforgeable dans le langage défini :

```
(define the-undefined-marker (cons 'undefined 'value))
```

L'expression `print` imprime le résultat de l'évaluation de sa sous-expression. L'évaluation

de l'expression `funcall` demande de récupérer la définition de la fonction dans l'environnement pour l'appliquer au résultat de l'évaluation de la liste d'expressions apparaissant en paramètres réels et à la mémoire courante. L'évaluation d'une liste d'expressions pour retourner la liste des résultats est assurée par la fonction `eval-expressions`.

La fonction `eval-if` réalise l'évaluation de l'expression `if` en utilisant simplement le `if` de Scheme ; ses sous-expressions sont évaluées selon les mêmes conditions que cette dernière. La fonction `eval-while` évalue répétitivement, par récursivité, le corps de la boucle aussi longtemps que la condition est vraie. La fonction `eval-begin` évalue chaque expression dans sa liste de sous-expressions pour retourner comme résultat le résultat de la dernière sous-expression.

**TP 2.** Variables locales et passage de paramètres par nom.

Le TP consiste à comprendre l'interprète défini dans la section 3.4 et à lui faire deux modifications :

1. Dans un premier temps, ajoutez au langage la possibilité de déclarer des variables locales dans les fonctions. Ces variables ne peuvent pas désigner des fonctions (nous verrons cela plus tard, au chapitre 4). Il faut modifier la syntaxe abstraite de la déclaration des fonctions, puis il faut modifier l'interprète en conséquence. Pour la syntaxe abstraite des déclarations, inspirez-vous des déclarations de variables globales.
2. Dans un deuxième temps, implantez le passage de paramètres par nom dans l'interprète. Rappelez-vous que le passage par nom exige deux choses : que l'évaluation du paramètre soit retardée jusqu'à son utilisation dans la fonction appelée, mais que cette évaluation doit se faire dans l'environnement de la fonction appelante.

## 3.5 Gestion de la mémoire pour l'activation de procédure

La gestion de mémoire proposée par notre premier interprète et utilisée par Fortran IV par exemple est très simple mais aussi fort limitée. Dans cette section, nous étendons le modèle vers la récursivité dans l'univers plat à la C, puis dans l'univers à structure de bloc à la Pascal.

### 3.5.1 Flôt de contrôle dans l'activation de procédure

Dans un langage séquentiel, le contrôle ne peut se trouver que dans une et une seule procédure à la fois. Lors la procédure `p` appelle la procédure `q`, `p` suspend son exécution pendant que `q` s'exécute. Lorsque `q` termine son exécution, le contrôle retourne à `p` qui reprend son exécution. En fait, en étudiant l'ensemble des appels dans un programme, on peut les décrire sous la forme d'un arbre, et voir le flôt de contrôle comme une exploration en profondeur d'abord de cet arbre. Le flôt de contrôle entre et sort des procédures selon une discipline dernier arrivé, premier servi, c'est-à-dire que le flôt sort d'abord de la dernière procédure où il est entré.

Ce comportement suggère que la gestion de mémoire en ce qui concerne l'allocation des blocs d'activation peut être réalisée par une pile, appelée *pile d'exécution*. Lorsque le flôt de

contrôle entre dans une procédure `p`, l'espace nécessaire à son bloc d'activation est alloué en sommet de pile. Peu importe les appels qui seront issus directement ou indirectement de `p`, on sait que le flôt de contrôle doit ressortir de ces procédures avant de quitter définitivement `p`. Le bloc d'activation de `p` se retrouvera donc alors en sommet de pile, et il suffira de le dépiler pour éliminer l'espace alloué devenu inutile.

La gestion de pile est très efficace. Désallouer consiste tout simplement à affecter une nouvelle valeur à la variable pointant vers le sommet de pile, ce qui se fait à coût unitaire constant. C'est pour cela que cette forme de gestion de mémoire est particulièrement appréciée. Lorsque nous avons introduit les blocs d'activation, nous avons signalé que l'on doit mémoriser dans le bloc d'activation le pointeur sur le bloc d'activation de l'appelant. La gestion en pile signifie simplement que ce pointeur, appelé *frame pointer* (`fp`) en anglais, peut servir de pointeur sur le sommet de la pile (dans la mesure où nous connaissons le déplacement maximal dans le bloc d'activation, on peut calculer facilement le «vrai» sommet de la pile). Désallouer le bloc en sommet de pile signifie donc tout simplement de ranger dans `fp` le pointeur sur le bloc d'activation précédent rangé dans le bloc d'activation à désallouer.

Cette gestion de la mémoire pour les blocs d'activation est tributaire des limitations sur le flôt de contrôle rencontré dans les langages impératifs normaux, comme C et Pascal. Elle est aussi dépendante de la durée de vie dynamique des variables locales dans ces langages. Nous verrons au chapitre 4 puis au chapitre 5 que ces restrictions sont levées dans les langages fonctionnels et à objets. Dans l'implantation de ces langages, il faut donc faire des entorses à la gestion sous forme de pile.

### 3.5.2 Portée des identificateurs

Dans les langages de programmation, on distingue deux grandes approches à la portée des identificateurs :

- *Portée dynamique* : un identificateur a une portée dynamique lorsque sa visibilité s'étend à l'ensemble des activations de procédures et fonctions découlant directement ou indirectement de l'activation de la procédure l'ayant déclaré. Dans un langage à portée dynamique, une référence à un identificateur est résolue en recherchant la première liaison de cet identificateur à partir du bloc d'activation courant et en descendant dans la pile d'exécution.
- *Portée lexicale* : un identificateur a un portée lexicale lorsque sa visibilité s'étend à l'ensemble des fonctions, procédures et constructions syntaxiques déclarés ou apparaissants à l'intérieur de l'unité (fonction, procédure, module, etc.) dans laquelle il est déclarée. Dans un langage à portée lexicale, une référence à un identificateur est résolue en recherchant le bloc dans la pile d'exécution correspondant à la plus récente activation de l'unité déclarant l'identificateur visible au point du programme où la référence apparait.

La portée dynamique a été utilisée pour les variables dans les premières implantations de Lisp, mais n'est plus guère utilisée aujourd'hui. Par contre, elle est toujours utile dans d'autres contextes. Par exemple, le traitement des exceptions l'utilise pour lier le nom d'une exception au traitant qui lui est associé dans le calcul en cours. Elle est aussi appropriée en intelligence artificielle pour faire des raisonnements hypothétiques, dans le contexte d'une recherche par retour arrière.

La portée lexicale est aujourd'hui la plus utilisée pour déterminer la visibilité des variables ou des noms de procédures. Le premier avantage de la portée lexicale est sa lisibilité. Consi-

dérons l'exemple suivant :

```
int x;

int foo() {
  return x;
}

main() {
  int x;

  printf("%i\n", foo());
}
```

En portée lexicale, celle utilisée par C, on peut facilement déterminer la liaison de la variable `x` visible dans la fonction `foo`. La règle veut que la liaison soit celle engendrée par la première déclaration de `x` dans le contexte de *déclaration* de `foo`. Il s'agit donc dans cet exemple de la variable globale `x`. Ce caractère purement syntaxique, *lexical* dirons-nous, permet de déterminer quelle sera la liaison utilisée uniquement en examinant le texte du programme, sans considérer l'exécution.

En portée dynamique, il est plus difficile de déterminer quelle sera la liaison visible. La règle dit que ce sera celle engendrée par la dernière déclaration rencontrée dans le contexte *d'exécution* de `foo`. Dans l'exemple, la dernière déclaration de `x` rencontrée est celle de la procédure `main`. Cette règle est facile à mettre en œuvre (mais pas nécessairement efficace !) pour l'ordinateur ; il suffit de chercher dans la pile d'exécution la liaison de `x` la plus récente. Pour le programmeur par contre, elle demande de suivre l'exécution du programme et de connaître les variables locales de chaque procédure appelée avant la référence de la variable.

En portée lexicale, trois grandes familles de langages impératifs existent : les langages à espace de noms plat, comme Fortran IV, les langages à espaces de noms en structure de blocs hétérogènes simplifiés, comme C, et ceux à espace de nom en structure de blocs, comme Pascal. Si les premiers sont beaucoup plus simples à implanter, ils offrent un mécanisme de contrôle de la visibilité des noms assez pauvre. C offre une structure de blocs où un différence est faite entre le niveau global et les niveaux imbriqués. Les procédures ne peuvent être déclarées qu'au niveau global ; il ne peut donc y avoir de procédures locales à d'autes procédures. La structure de bloc complète de Pascal donne un contrôle plus fin pour gérer la visibilité des noms, mais son implantation est nettement plus subtile, et il rend difficile le passage des fonctions en paramètre et leur retour comme résultat.

### 3.5.3 Univers hétérogène à la C

L'univers plat de C, engendrant un espace de noms plat, est caractérisé par les choix de conception et d'implantation suivants :

1. Toutes les procédures sont déclarées au même niveau et leur visibilité est globale. Prise négativement, cette restriction veut dire qu'il n'est pas permis de déclarer des fonctions locales à d'autres fonctions.

2. Un bloc lexical (délimité par `{` et `}`) peut inclure des déclarations de variables locales ; le corps d'une procédure est un bloc.
3. La récursivité est permise ; chaque activation d'une procédure doit donc avoir sa propre mémoire locale pour contenir les valeurs de ses paramètres formels et de ses variables locales pour cette activation.
4. Les pointeurs sur les fonctions sont des valeurs de plein droit ; il est possible de les passer en paramètre réel, de les ranger dans des variables, de les retourner comme résultat de fonctions.

Les deux premiers choix impliquent en fait qu'il n'existe en gros que deux types de déclarations. Les déclarations globales, qui peuvent lier un nom à une valeur de premier ordre (constantes de bases ou valeurs de types définis par le programmeur, c'est-à-dire des variables) ou à une procédure. Les déclarations locales (y compris les paramètres formels) ne peuvent lier un nom qu'à une valeur de premier ordre (y compris les pointeurs sur les fonctions). La troisième impose l'utilisation de blocs d'activation alloués à l'exécution (contrairement à l'allocation statique en Fortran IV). La dernière, comme nous le comprendrons mieux à la section suivante, ne pose pas de contraintes supplémentaires ; elle est plutôt le constat de la plus grande flexibilité qu'il était possible de laisser aux programmeurs étant donnés les choix précédents.

La simple dichotomie entre variables globales et locales rend la gestion de la mémoire plus simple. Les variables globales sont allouées statiquement et la génération de code peut lier chaque référence directement à l'emplacement mémoire alloué à la compilation. Pour les variables locales, il suffit de leur allouer un emplacement dans le bloc d'activation et de générer l'accès sous la forme base plus déplacement, comme nous l'avons discuté précédemment. Cette technique d'accès permet de générer le code lors de la compilation de façon à ce qu'il ne dépende que de la valeur d'un registre de base particulier (celui contenant l'adresse du bloc d'activation en sommet de pile). Cette technique permet donc de rendre la génération de code indépendante de l'allocation dynamique du bloc d'activation.

Le langage C permet d'imbriquer les blocs simples les uns dans les autres. Par bloc simple, on entend un bloc qui ne correspond pas à une procédure. Ce genre de blocs imbriqués n'est utilisé (rarement en réalité) que pour limiter la visibilité d'une variable par ailleurs locale à une fonction. La plupart des implantations de C allouent ces variables dans le bloc d'activation de la procédure englobante, tout en s'assurant à la compilation que les variables en question ne sont référencées que dans les blocs où elles sont effectivement visibles. L'autre option serait de les allouer uniquement à l'entrée du bloc déclarant et de les désallouer à la sortie.

### Un exemple de recherche binaire

La figure 3.7 présente un programme classique de recherche binaire dans un vecteur trié, écrit en C. La fonction `search` prend deux entiers `lo` et `hi` tels que la valeur recherchée est située entre les indices `lo` et `hi` dans le vecteur. L'idée de la recherche dichotomique est de situer de plus en plus précisément l'intervalle dans lequel se situe la valeur en coupant à chaquue appel récursif l'intervalle en deux. On calcul `k` la valeur médiane entre `lo` et `hi`. Trois alternatives sont alors possibles :

1. la valeur recherchée est égale à la valeur rangée à l'indice `k`, auquel cas on a trouvé,

```
#include <stdio.h>

int yes = 1, no = 0;
#define N 7
int X[] = { 0, 11, 22, 33, 44, 55, 66, 77 };
int T;

int search(int lo, int hi) {
  int k;
  if (lo > hi)
    return no;
  else {
    k = (lo + hi) / 2;
    if (T == X[k])
      return yes;
    else
      if (T < X[k])
        return search(lo, k-1);
      else
        if (T > X[k]) return search(k+1, hi);
  };
}

int main(void) {
  scanf("%d", &T);
  if (search(1, N))
    printf("found\n");
  else
    printf("not found\n");
  return 0;
}
```

FIG. 3.7 – Programme de recherche binaire dans un vecteur trié

2. la valeur recherchée est plus petite que la valeur rangée à l'indice `k`, auquel cas elle se trouve dans l'intervalle `lo`–`k` et on se rappelle récursivement en remplaçant la valeur de `hi` par celle de `k-1`, ou
3. la valeur recherchée est plus grande que la valeur rangée à l'indice `k`, auquel cas elle se trouve dans l'intervalle `k`–`hi` et on se rappelle récursivement en remplaçant la valeur de `lo` par celle de `k+1`.

La procédure `main` lit une valeur au terminal et lance la recherche entre les indices `1` et `N`, `N` étant la borne supérieure des indices.

Examinons la croissance de la pile d'exécution si on lance le programme en lui donnant la valeur 55 à rechercher. La figure 3.8 présente les états successifs de la pile. Au départ, le fond de la pile est initialisé d'une façon que nous ne décrirons pas. Supposons simplement qu'un certain nombre d'espaces sont requis pour ce «bloc d'activation» initial, en l'occurence deux espaces aux adresses 0 et 1. Le bloc d'activation de la procédure fonctionnelle `search` contient :

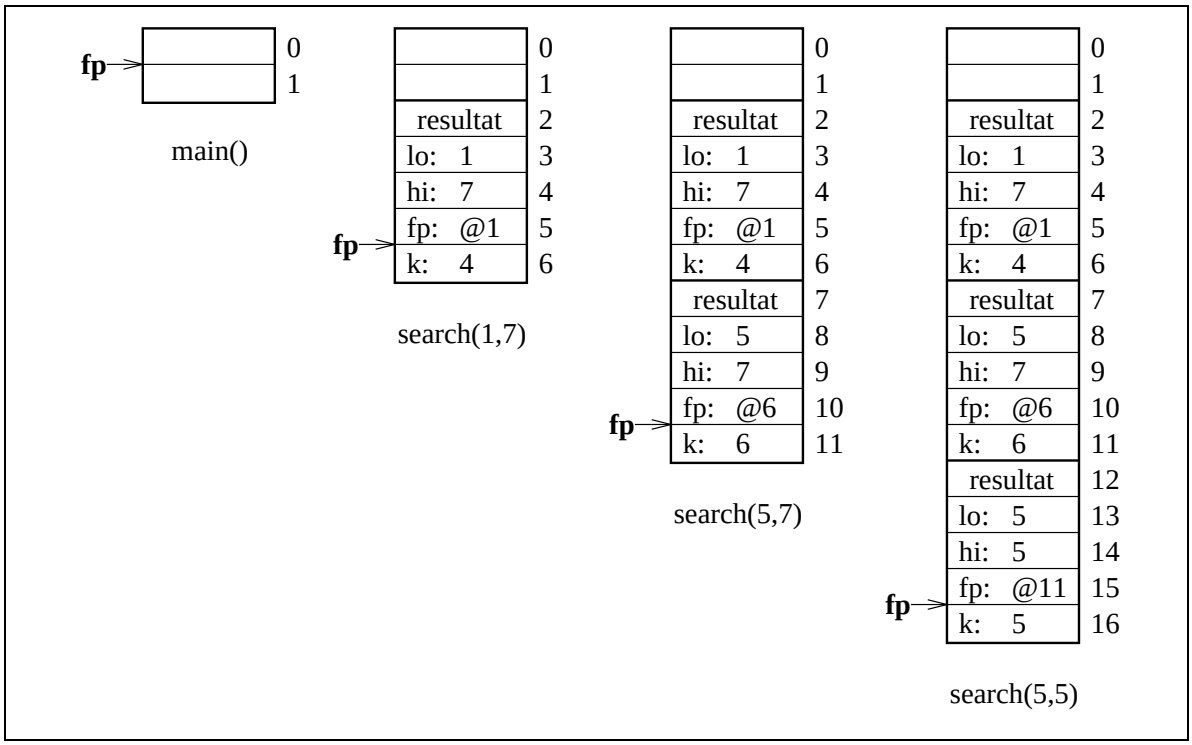



FIG. 3.8 – Pile d'exécution pour la recherche de la valeur 55

- un espace pour son résultat,
- deux espaces pour les deux paramètres formels `lo` et `hi`,
- un espace pour l'information interne en l'occurence la valeur du pointeur sur le bloc d'activation précédent qui est en fait la valeur précédente du registre `fp` (celui qui pointe sur le bloc en sommet de pile), et
- un espace pour la variable locale `k`.

Supposons qu'à tout moment, le registre `fp` contient l'adresse de l'espace mémoire venant immédiatement apres l'emplacement de l'ancienne valeur de `fp` dans le bloc d'activation en sommet de pile. Initialement, `fp` pointe sur l'adresse 1. Le premier appel à `search` engendre le premier bloc d'activation où on voit les valeurs `1` et `7` passée à `lo` et `hi`. La valeur courante de `fp` est sauvegardée dans le bloc d'activation et la valeur de `fp` est modifiée pour pointer sur l'adresse 6. Les adresses sont indiquées par le caractère `@` dans la pile. L'exécution de `search` calcule la valeur de `k`, c'est-à-dire `4`, et la range dans le bloc d'activation. Puisque la valeur recherchée `55` est plus grande que `T[4]`, c'est-à-dire `44`, `search` se rappelle récursivement avec `lo` passant à la valeur `5`.

Cet appel récursif engendre le second bloc d'activation. La valeur de `lo` dans cette nouvelle activation est `5` alors que celle de `hi` est `7`. La valeur de `fp` est rangée dans le bloc, c'est-à-dire `@6`, et l'exécution de la fonction est lancée. La nouvelle valeur de `k` est calculé (`6`) et rangée dans le bloc. Puisque la valeur recherchée `55` est plus petite que `T[6]`, c'est-à-dire `66`, `search` se rappelle récursivement avec `hi` passant à la valeur `5`.

Ce deuxième appel récursif engndre le troisième bloc d'activation où les valeurs de `lo` et `hi` sont toutes deux `5`. La valeur de `fp` (`@11`) est rangée dans le bloc et `fp` devient `@16`. La

valeur calculée pour `k` est `5` et on trouve à cet indice dans le vecteur la valeur recherchée. Cette activation de la fonction se termine et retourne donc la valeur de `yes`.[4] Cette valeur est rangée dans la partie résultat du bloc d'activation et on range dans `fp` l'adresse `@11` sauvegardée dans le bloc d'activation. L'activation précédente est relancée et elle copie simplement le résultat de l'appel récursif dans son propre espace résultat. Cette activation se termine et retourne le contrôle à l'appelant en rangeant la valeur `@6` dans `fp`. On se retrouve alors au niveau de la première activation de `search`, qui se termine elle aussi en retournant la valeur `yes` à `main`.

**TP 3.** Allocation dynamique des variables locales

Notre interprète fait une allocation statique des variables locales (comprenant les paramètres formels). Le but de ce TP est de le modifier pour que l'allocation soit dynamique. Faites cette modification en deux temps :

1. Dans un premier temps, modifiez l'allocation des variables locales pour qu'elle se fasse lors de l'appel de la procédure plutôt que lors de sa définition.
2. Dans un deuxième temps, intégrez la gestion des blocs d'activation en ayant un pointeur `fp` sur le bloc en sommet de pile et en liant les variables dans l'environnement à une adresse relative par rapport à `fp`. N'oubliez pas la gestion de `fp` lors des appels et retours de procédures (pour simplifier, faites en sorte que `fp` pointe sur le dernier emplacement pris par le bloc d'activation ; ainsi vous aurez directement l'information sur le vrai sommet de la pile).

### Optimisation de la récursivité terminale

Le lecteur curieux aura peut-être remarqué dans notre description de l'exemple de la recherche binaire un phénomène ouvrant à optimisation. En effet, lors du retour de la dernière activation de `search`, le résultat `yes` est passé à l'appelant, qui était en réalité l'activation précédente de `search` sur les paramètres réels `5` et `7`. Cette activation précédente ne fait à son tour rien d'autre que recopier le résultat `yes` dans son propre champ résultat de son bloc d'activation et retourne à son appelant. à nouveau un appel à `search` sur les paramètres réels `1` et `7`. Finalement, cette première activation de `search` se termine elle aussi en recopiant tout bonnement la valeur de `yes` dans son champ `resultat` pour le retourner à son propre appelant, c'est-à-dire la procédure `main`. S'il s'agit simplement de recopier le résultat d'appel en appel, ne pourrions-nous pas retourner le résultat directement au premier appelant de la fonction `search` ?

Cette idée, vue de façon très opérationnelle, est une manifestation de ce que l'on appelle *récursivité terminale*, et le court-circuit suggéré est obtenu, en plus d'un meilleure utilisation de l'espace, par *l'optimisation de la récursivité terminale*. Historiquement, on sait depuis longtemps que la récursivité et l'itération ont une puissance d'expression égale. Tout ce que l'on peut écrire sous forme de boucle peut s'écrire sous forme de récursivité, et *vice versa*. La transformation d'une récursivité en une itération demande cependant d'utiliser une pile (ce qui est obtenu dans l'implantation des fonctions récursives par la pile d'exécution elle-même). La grande différence donc entre itération et récursivité est le fait qu'une itération ne demande

[4] La partie retour n'est pas illustrée à la figure 3.8.

```
#include <stdio.h>

int yes = 1, no = 0;
#define N 7
int X[] = { 0, 11, 22, 33, 44, 55, 66, 77 };
int T;

int search(int lo, int hi) {
  int k;
L: if (lo > hi)
    return no;
  else {
    k = (lo + hi) / 2;
    if (T == X[k])
      return yes;
    else
      if (T < X[k])
        hi = k - 1;
      else
        lo = k + 1;
  };
  goto L;
}

int main(void) {
  scanf("%d", &T);
  if (search(1, N))
    printf("found\n");
  else
    printf("not found\n");
  return 0;
}
```

FIG. 3.9 – Programme de recherche binaire dans un vecteur trié avec optimisation de la récursivité terminale

qu'un espace constante (l'espace nécessaire aux variables affectées par l'itération) alors que la récursivité demande un espace linéraire dans le nombre d'appel récursif. Cet espace linéraire est nécessaire pour sauvegarder les valeurs des variables pour chaque appel récursif, car si l'appel récursif modifie la valeur d'une variable, il faut pouvoir retrouver la valeur initiale au retour de l'appel récursif.

Notez que cette dernière condition n'est vraie que si cette valeur initiale est encore utile lorsque l'appel récursif ramène le contrôle à l'appelant. Dans l'exemple de la recherche binaire, on a vu que l'appelant `search` n'a plus besoin de ses variables locales (et ses paramètres formels) lorsque l'appel récursif revient car il n'a plus auncu travail à faire autre que rendre son résultat. Le fait de ne plus avoir de travail à faire au retour de l'appel récursif est ce que l'on appelle un appel récursif terminal, c'est-à-dire que l'appel récursif est la dernière chose que la procédure fait lors de son activation.

L'idée de l'optimisation de la récursivité terminale est d'exploiter le fait que l'appelant n'a

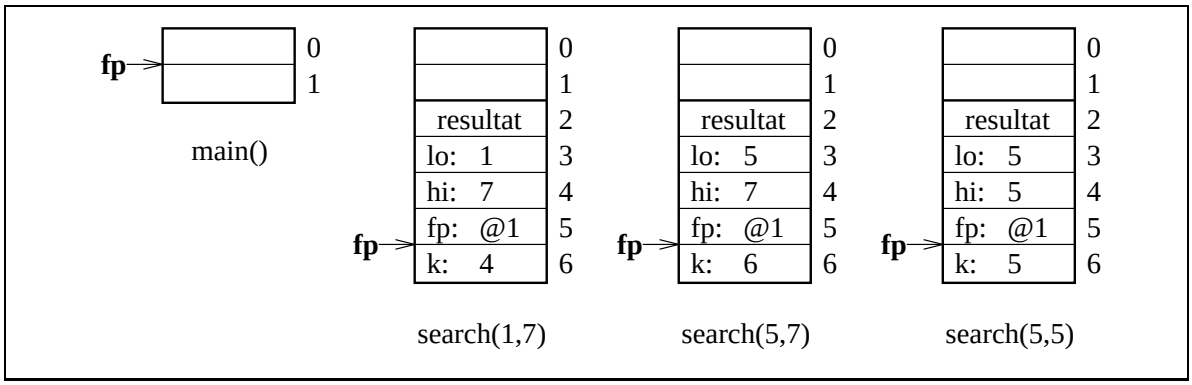



FIG. 3.10 – Pile d'exécution pour la recherche de la valeur 55 avec optimisation de récursivité terminale

plus besoin de ses variables au retour de l'appelé pour eliminer avant l'appel récursif l'espace nécessaire pour l'activation de l'appelant. Tout se passe comme si le premier appel à `search` disait à l'activation lancée par l'appel récursif : «Moi, je n'ai plus rien à faire avec le résultat que tu vas me retourner ; rend-le donc à ma place à mon appelant `main`.» Pour cela, l'appelant passe à l'appelé l'adresse de retour de son propre appelant.

Une fois comprise cette idée, remarquons que l'appel récursif va avoir besoin d'un bloc d'activation exactement semblable à celui de l'appelé ! De plus, on veut conserver la valeur de `fp` de l'appelant. L'optimisation de récursivité terminale réutilise donc le bloc d'activation de l'appelant pour exécuter l'appelé. De façon globale, un seul bloc d'activation est alors nécessaire pour exécuter tous les appels récursifs. On utilise donc un espace constant, ce qui devient aussi intéressant qu'une itération. Les langages fonctionnels ont été les pionniers de ce type d'optimisation parce qu'ils ne fournissent généralement pas d'énoncés d'itération, ce qui force les programmeurs à n'écrire que des récursivités. En promettant de faire l'optimisation de récursivité terminale, le langage Scheme, par exemple, rassure les programmeurs sur la performance en espace de leurs fonctions récursives terminales.

Techniquement, l'optimisation de récursivité demande d'abord d'identifier les fonctions récursives terminales. Ceci se détermine simplement en examinant le texte de la procédure à la compilation. Lorsqu'une procédure est récursive terminale, l'appel récursif ne s'exécute pas comme les autres appels dans le programme. En fait l'appel récursif consiste simplement à réutiliser le bloc d'activation en sommet de pile, ce qui se résume par le fait de ranger les valeurs des paramètres réels de l'appel récursif dans les espaces prévus à cet effet dans le bloc d'activation courant.

La figure 3.9 propose une simulation en C de l'optimisation de récursivité terminale appliquée au programme de recherche binaire. L'idée est de placer une étiquette (ici `L`) au début du corps de la fonction et de remplacer les appels récursifs terminaux par une simple affectation de la nouvelle valeur des paramètres formels et un branchement inconditionnel au début du corps de la fonction. Dans notre exemple, le branchement est mis à la fin du corps car le `return` rompra le flôt pour retourner lorsque la récusivité sera terminée, et toutes les autres braches du flôt d'exécution se terminent par un appel récursif terminal.

La figure 3.10 illustre l'effet sur la pile de l'optimisation de récursivité terminale. Un seul

bloc d'activation sert à toutes les activations de la fonction `search`. Seules les valeurs des paramètres formels et de la variable locale `k` changent d'une activation à l'autre.

#### Fonctions comme paramètre et résultat

Nous verrons au chapitre 4 que dans les langages fonctionnels, les fonctions sont des entités de plein droit. Est entité de plein droit dans un langage tout ce qui peut être créé à l'exécution, rangé dans une variable, passé en paramètre et retourné comme résultat de fonctions. Le principal problème des fonctions comme entités de plein droit est la capacité de les appelées dans un contexte tel qu'elle pourra s'exécuter normalement. Une fonction a besoin pour s'exécuter de son bloc d'activation, mais aussi d'un accès à toutes les variables non-locales qui sont référencées dans son corps.

En C, les fonctions ne sont pas des entités de plein droit car elles ne peuvent être créées à l'exécution. Par contre, il est possible de prendre un pointeur sur une fonction, et ce pointeur est une entité de plein droit. L'activation de la procédure ne pose pas de problème d'accès aux variables car, comme nous l'avons vu, une variable non-locale en C est nécessairement globale et ses références sont traduites en accès aux emplacements mémoires alloués statiquement pour ces variables. Nous verrons qu'en Pascal, les choses ne sont malheureusement pas si simples.

### 3.5.4 Structure de blocs à la Pascal

Les langages à structure de blocs à la Pascal (Modula-2, Ada, etc.) lèvent une restriction par rapport à C, nommément l'interdiction de déclarer des fonctions locales à d'autres fonctions. L'intérêt est bien sûr de découper une fonction complexe en plusieurs sous-fonctions et de masquer ces sous-fonctions. Souvent le traitement réalisé par ces sous-fonctions est sémantiquement incomplet au sens où les appeler isolément pourrait violer un invariant de données. Par contre, la gestion des accès aux variables et aux fonctions non-locales est plus complexe, et force, pour garder l'allocation des blocs d'activation selon la discipline de pile, à abandonner la possibilité de retourner les pointeurs sur les fonctions comme résultat de fonctions.

Le principe de portée des variables dans la structure de bloc est la portée lexicale. On peut redéfinir localement à un bloc les variables définies dans les blocs englobants, auquel cas c'est la déclaration la plus locale qui s'impose aux références à l'intérieur du bloc. La figure 3.11 propose un exemple d'utilisation de la structure de bloc en Pascal. Ce programme déclare une procédure `p`, à l'intérieur de laquelle est déclarée la procedure `q` qui, elle-même, contient la déclaration de la procédure `r`. Selon les règles de la portée lexicale et de l'imbrication des blocs :

1. La variable globale `n` est visible partout dans le programme.
2. Le paramètre formel `i` et les variables locales `x`, `y` et `z` de la procédure `p` ont une portée qui s'étend à toute la procédure `p` et à la procédure locale `q`.
3. Le paramètre formel `i` de `q` masque dans `q` le paramètre formel `i` de `p` et sa portée s'étend à toute la procédure `q` et à la procédure locale `r`.
4. Le paramètre formel `i` de `r` masque dans `r` le paramètre formel `i` de `q` et sa portée s'étend à toute la procédure `r`.

```
program m(input, output);
  var n : integer;

  procedure p(i: integer);
    var x, y, z : integer;

    procedure q(i: integer);

      procedure r(i: integer);
      begin (* r *)
        if i < 0 then
          z := 0
        else
          begin
            z := i;
            writeln(output, x, y, z);
            p(i - 1)
          end
      end; (* r *)

    begin (* q *)
      y := i;
      r(i - 1)
    end; (* q *)

  begin (* p *)
    x := i;
    q(i - 1)
  end; (* p *)

begin
  write('Donnez un petit nombre entier svp: ');
  readln(n);
  p(n)
end.
```

FIG. 3.11 – Exemple de structure de bloc en Pascal

Les variables `y` et `z` qui sont référencées dans les procédures `q` et `r` sont des variables non-locales allouées lors de la création du bloc d'activation de `p`. Pour retrouver leur valeur, il faut donc que les procédures `q` et `r` sachent retrouver le bloc d'activation correspondant à l'activation la plus récente de `p`. Plus généralement, dans un langage à structure de bloc, tout bloc d'activation doit connaître les blocs d'activation correspondant aux activations les plus récentes de ses blocs lexicaux englobants.

L'imbrication des blocs lexicaux crée une structure de niveaux qui est statique. En effet, la simple analyse du texte de programme permet d'attribuer un niveau à chaque bloc lexical. Le programme est au niveau 0, les procédures déclarées dans le programme sont au niveau 1, celles déclarées à l'intérieur des procédures précédentes sont au niveau 2, et ainsi de suite. Cette propriété a deux conséquences intéressantes :

1. On sait exactement à quel niveau d'imbrication chaque variable est déclarée, et donc on sait statiquement à quel niveau est declarée la variable accédée par chaque référence dans le programme.
2. La différence entre le niveau d'imbrication où une référence apparait et le niveau où la variable référencée est déclarée est connu statiquement et il est constant.

La technique la plus simple consiste à maintenir dans le bloc d'activation un pointeur, appelé *lien statique*, vers le bloc d'activation correspondant à l'activation la plus récente de son bloc lexical englobant. En suivant les liens statiques, on retrouvera l'un après l'autre les bloc d'activation successif des blocs lexicaux englobants. Dans notre exemple, le bloc d'activation de `q` a un lien statique vers le bloc d'activation le plus récent de `p` qui est son bloc lexical englobant. De même, le bloc d'activation de la procédure `r` a un lien statique vers le bloc d'activation le plus récent de `q` qui est son bloc lexical englobant.

La différence entre le niveau de référencement et le niveau de déclaration donne en fait le nombre de liens statiques qui devront être traversés pour trouver le bloc d'activation dans lequel la valeur de la variable est rangée. Cette différence sert également lors de l'appel d'une procédure à déterminer combien de liens statiques doivent être traversés pour trouver le bloc d'activation le plus récent correspondant au bloc lexical englobant. En fait, si la procédure appelée est déclarée dans la procédure appelante, le lien d'accès de l'appelé pointe sur le champ contenant le lien d'accès dans le bloc d'activation de la procédure appelante. Sinon, pour une différence $d$ entre les niveaux, la valeur du lien d'accès de la procédure appelée sera trouvée dans le champ atteint en traversant `d` liens d'accès à partir de l'appelant.

L'accès à une variable non-locale se fait donc de la façon suivante. Dans `q`, l'accès à la variable `y` se fait en parcourant un lien statique, ce qui nous mène au bloc d'activation de `p` ; le lien statique sert d'adresse de base à laquelle on ajoute le déplacement décidé pour `y` pour trouver l'adresse de cette variable. L'accès aux variables `x`, `y` et `z` dans `q` exige de suivre deux liens statiques. Le premier mène au bloc d'activation de `q`, le second au bloc d'activation de `p`.

La figure 3.12 présente l'état de la pile pour l'exécution du programme de la figure 3.11 si on donne en entrée la valeur 6. L'appel de la procédure `p` dans le corps du programme génère le premier bloc d'activation (pile 1). Ce bloc d'activation contient le paramètre formel `i` de valeur *6*, le lien sur le bloc d'activation précédent `cl` (pour «*control link*»), le lien sur le bloc d'activation le plus récent du bloc lexical englobant `al` (pour «*access link*»), et les trois variables locales `x`, `y` et `z` allouées mais non-initialisées. La procédure `p` s'exécute et range la valeur de `i` dans `x` puis appelle la procédure `q` en lui passant la valeur 5.

Cet appel de `q` génère le second bloc d'activation (pile 2) avec le paramètre formel `i` de `q` de valeur 5, le lien de contrôle valant `@4` et le lien statique valant également `@4`. La procédure `q` s'exécute et range la valeur de `i` (le paramètre formel local) dans la variable `y`. Pour accéder `y`, on traverse un lien statique et on trouve l'emplacement mémoire de `y` au déplacement `+2` par rapport au lien statique. Ensuite `q` appelle la procédure `r` en lui passant la valeur 4.

Cet appel à `r` génère le troisième bloc d'activation (pile 3). Passons sur les valeurs dans le bloc d'activation, et notons que l'accès à la variable `z` se fait en parcourant deux liens statiques : le premier de valeur `@10` nous amène à l'adresse du second, dont la valeur est `@4`. `z` au déplacement `+3` par rapport à l'adresse donnée dans ce deuxième lien statique. La procédure `r` rappelle alors `p`, une récursivité indirecte, avec la valeur 3.

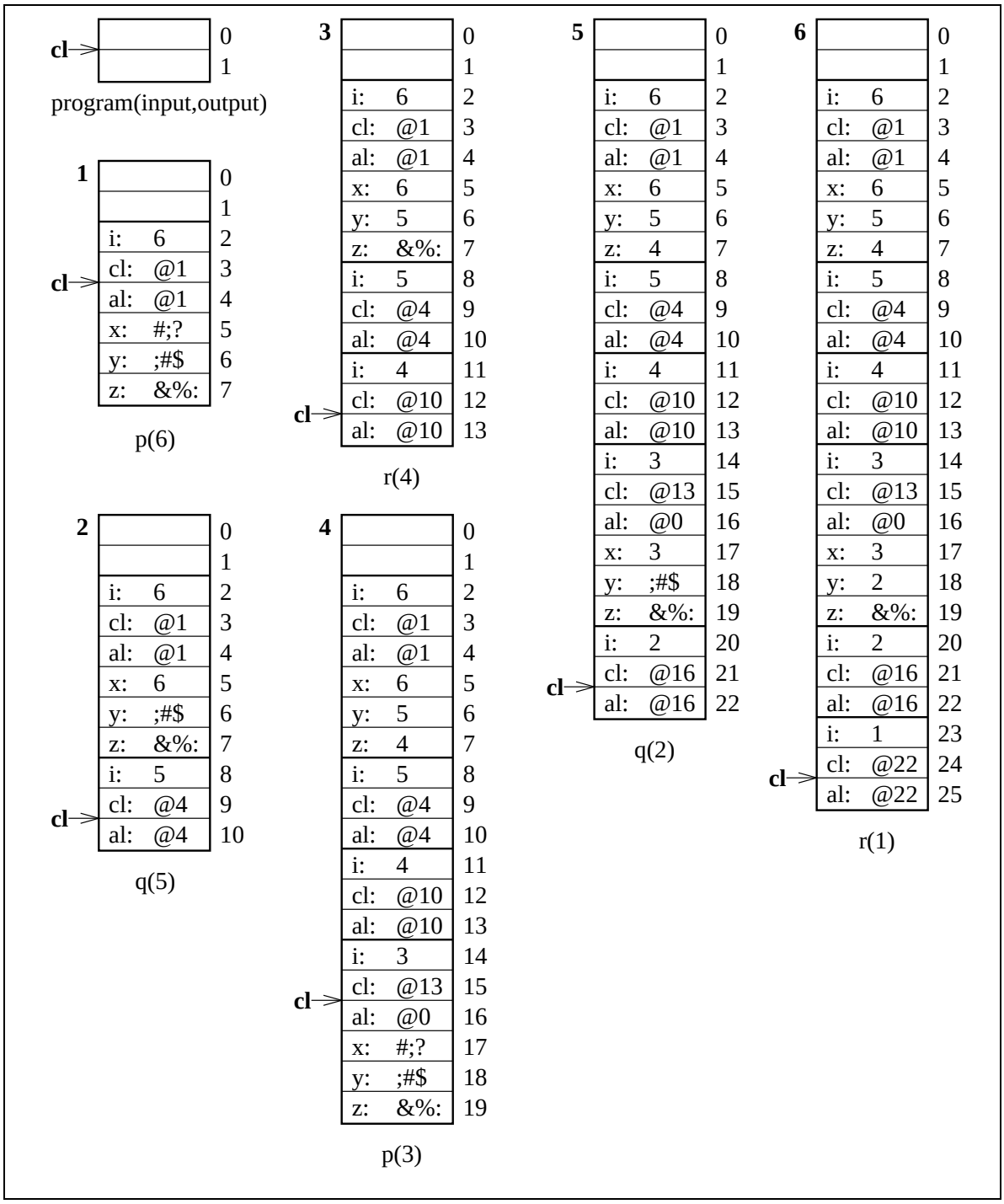



FIG. 3.12 – Pile d'exécution avec lien statique en Pascal (donnée : 6)

Cet appel à `p` génère le quatrième bloc d'activation (pile 4) qui devient le plus récent bloc d'activation correspondant au bloc lexical englobant de `q` et `r`. Ainsi, dans la suite de l'exécution, c'est sur ce bloc d'activation que devront revenir les accès aux variables `x`, `y` et `z` des activations subséquentes de `q` et `r`. Cela apparait dans la suite de l'exémple, dont

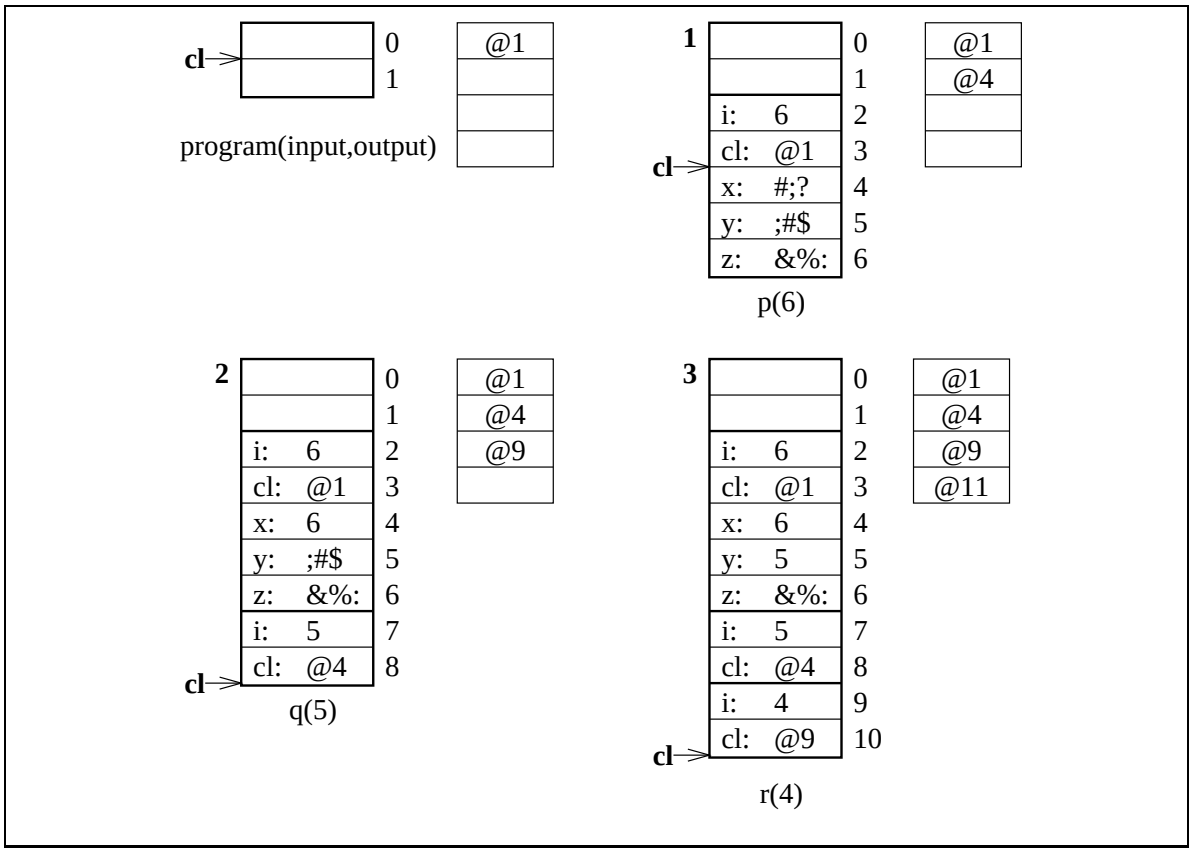



FIG. 3.13 – Pile d'exécution avec «*display*» en Pascal (donnée : 6, première partie)

l'exploration est laissée au soin du lecteur.

### La technique du «*display*»

Une autre conséquence du caractère statique de la structure en niveaux d'imbrication est que le nombre total de niveaux est connu statiquement et il est limité. Peut-on utiliser cette propriété pour améliorer l'accès aux variables non-locales ? Dans la technique des liens statiques, l'idée est de chaîner les blocs d'activation les plus récents de chaque niveau d'imbrication dans une liste. Une liste est une structure de données utile lorsque le nombre de valeurs à sauvegarder est variable, mais dans notre cas le nombre de niveaux est fixe pour un programme. Lorsqu'un nombre fixe de valeurs doivent être sauvegardées, le vecteur est une structure de données plus efficace car elle permet un accès direct à chacune de ces valeurs. Dans une liste, l'accès est linéaire dans la taille de la liste ; c'est bien ce que nous avons précédemment lorsqu'il faut parcourir les liens statiques pour retrouver le bloc d'activation contenant la variable non-locale.

Bien sûr, il n'est pas question de transformer la pile en vecteur, car le nombre de niveaux dans la pile entre le bloc d'activation de la procédure où la référence est faite et le bloc possédant la valeur référencée n'est lui pas fixe du tout. Dans notre exemple précédent, un nombre indéterminé d'appels récursifs à la procédure `q` pourraient avoir été faits avant d'appeler la

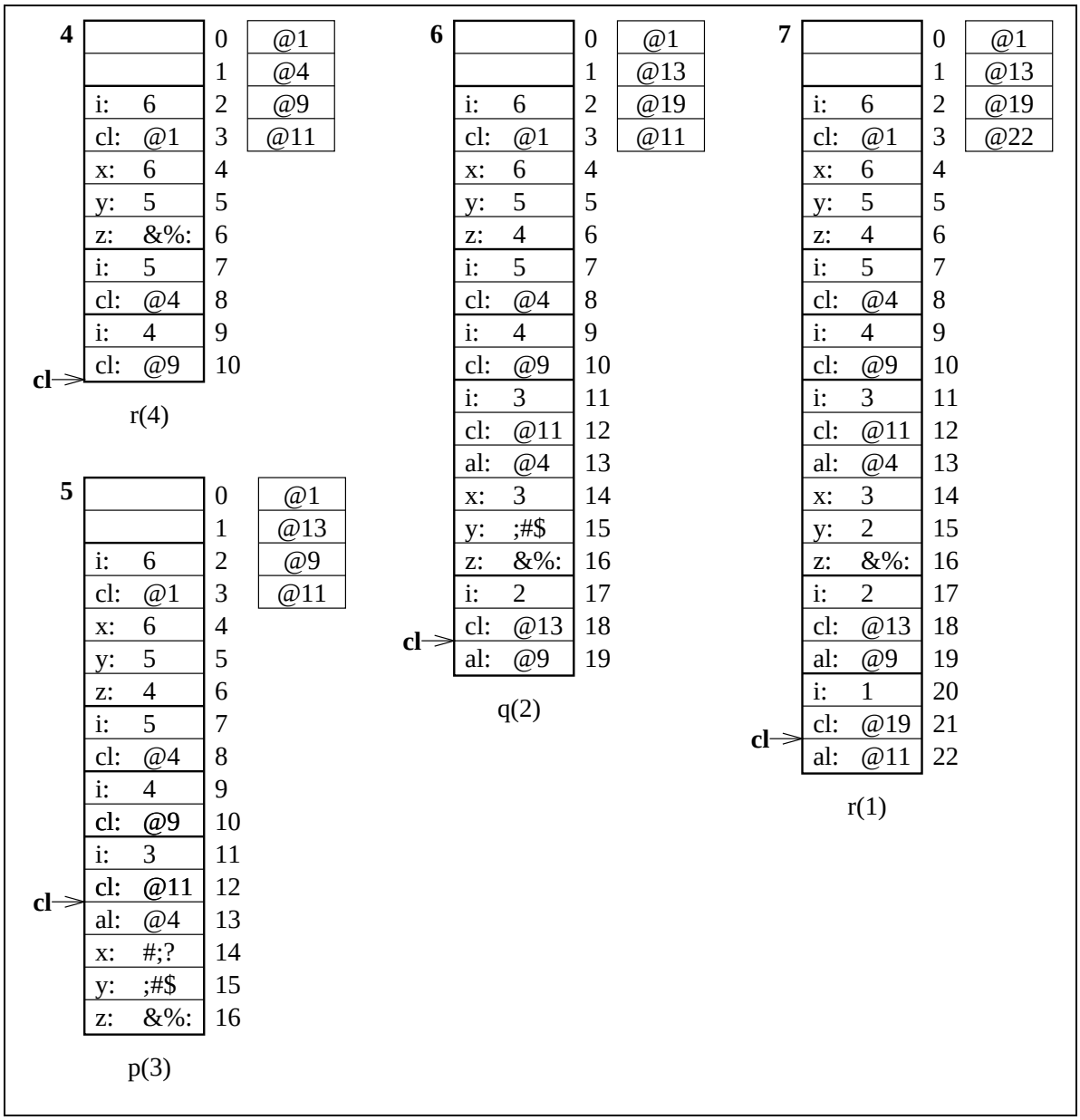



FIG. 3.14 – Pile d'exécution avec «*display*» en Pascal (donnée : 6, deuxième partie)

procédure `r`. La distance en nombre de blocs d'activation total entre `r` et `p` est donc inconnue au moment de la compilation.

L'idée de la technique des «*displays*» est plutôt de maintenir à coté de la pile un vecteur contenant les pointeurs à tous les blocs d'activation correspondant à la plus récente activation des procédures correspondant à chaque niveau d'imbrication. Ainsi, l'accès à une variable non-locale se fait par une indirection dans le vecteur utilisant le numéro du niveau de déclaration pour trouver le bloc d'activation, puis une autre indirection dans le bloc d'activation.

Les figures 3.13 et 3.14 présentent l'état de la pile d'exécution pour le même exemple que

précédemment. En fait pour les quatre premiers appels, la pile croît de la même façon que dans l'exemple précédent, à ceci près que les liens statiques («*access links*») se retrouvent dans le *display* plutôt que dans les blocs d'activation. L'accès à la variable `z` dans `r` se fait en indiçant le *display* par le niveau d'imbrication auquel `z` est déclarée, c'est-à-dire 1 (base 0). À l'indice 1 dans le *display*, on trouve l'adresse (`@4`) du bloc d'activation de `p` et on trouve l'emplacement mémoire de la variable `z` au déplacement `+2` par rapport à cette adresse.

À partir de l'appel récursif indirect à `p`, un nouveau phénomène apparaît. Puisque `p` est au niveau d'imbrication 1, le nouveau bloc d'activation remplace dans le `display` l'activation précédente de `p` à l'indice 1. Évidemment, il faut conserver quelque part l'ancienne valeur qui était dans le *display* pour la remettre en place lorsque cette nouvelle activation de `p` se terminera. On conserve donc cette valeur dans le bloc d'activation.

### Fonctions comme paramètre et résultat

Pascal est beaucoup plus contraignant que C en ce qui concerne le passage de fonctions en paramètre, et il interdit le retour comme résultat. En fait, Pascal ne possède pas de pointeurs sur les fonctions. Pour passer une fonction en paramètre, il faut directement utiliser son nom. Pourquoi ces restrictions ?

Toutes ces restrictions concourent en effet à deux propriétés importantes :

1. Une fonction `f` ne peut être référencée (par son nom) que dans la portée lexicale où elle est visible, et cela ne pourra donc se faire que si les procédures correpondant à ses blocs lexicaux englobants ont été activées.
2. La fonction à laquelle `f` est passée en paramètre est appelée à l'intérieur de l'activation du bloc lexical englobant `f`, et par le principe du dernier appelé premier terminé, cette fonction se terminera avant le bloc lexical englobant `f`.

Ces deux propriétés assurent donc que lorsque `f` sera effectivement appelée, si jamais elle l'est, ses blocs lexicaux englobants ont des activations présentes dans la pile auxquelles `f` pourra se référer si elle accéde à des variables non-locales. En effet, imaginons dans notre exemple précédent que `p` soit une fonction et qu'elle retourne la procédure `r` comme résultat. `p` retournant, son bloc d'activation est détruit. À quels emplacements mémoire pourrait-on trouver les variables `x`, `y` et `z` ? Ce genre de comportement irait à l'encontre de la discipline de pile. On dit que la fonction `r` serait exécutée hors de la durée de vie de son créateur, ou autrement dit qu'elle devrait avoir une durée de vie illimitée. Ceci est incompatible avec la structure de bloc et la discipline de pile. En C, il n'y a pas de problème car les seules variables non-locales sont globales et ont elles-mêmes une durée de vie illimitée. Nous verrons que les langages fonctionnels permettent à la fois la structure de bloc et la durée de vie illimitée des fonctions, mais cela en sacrifiant la stricte discipline de pile.

L'implantation du passage de procédures en paramètre en structure de bloc exige de tenir compte de la gestion de l'imbrication. En effet, lorsque la procédure passée en paramètre sera exécutée, il faudra être en mesure de retrouver les blocs d'activation de ses blocs lexicaux englobants. On ne peut pas déterminer leurs positions statiquement. Si on utilise la technique des liens statiques dans le bloc d'activation, il suffit de passer avec la procédure son lien statique. Cela est possible car on peut déterminer statiquement la position du bloc d'activation du bloc lexical englobant au moment où on passe la procédure en paramètre. En fait, cette

approche fonctionne bien sauf qu'il peut arriver qu'une nouvelle activation du bloc lexical englobant se produise avant que la procédure passée en paramètre ne soit effectivement appelée. Parce que le lien statique est déterminé au moment du passage et non au moment de l'appel, la sémantique d'un appel d'une procédure passée en paramètre est légèrement différente de celle d'un appel normal.

Si la technique utilisée est celle du *display*, c'est *grosso modo* toutes les valeurs du *display* qu'il faut passer en paramètre avec la fonction, et le déterminant au moment de l'appel, la même anecdote de sémantique s'applique aussi à ce cas.

**TP 4.** Fonctions locales et liens statiques.

Notre interprète ne permet pas la déclaration de fonctions locales à d'autres fonctions. Au TP 2, vous avez ajouté la possibilité de déclarer des variables locales à l'intérieur des fonctions. Étendez à nouveau le langage pour permettre la déclaration de fonctions locales et réalisez une implantation utilisant les liens statiques.
La présence de fonctions locales se manifeste d'abord par l'ajout d'un domaine syntaxique pour les déclarations de fonctions locales. Ce domaine est un extension du domaine ajouté au TP 2 pour les déclarations de variables locales.

$$
\begin{array}{rcl}
d & ::= & \texttt{(defun } i \texttt{ ( } i^* \texttt{ ) ( } l^* \texttt{ ) } e \texttt{)} \mid \texttt{(define } i \ e \texttt{)} \\
l & ::= & \texttt{(locfun } i \texttt{ ( } i^* \texttt{ ) ( } l^* \texttt{ ) } e \texttt{)} \mid \texttt{(local } i \ e \texttt{)}
\end{array}
$$

Pour réaliser ce travail, il faut :

1. Ajouter le lien statique dans les blocs d'activation. Pour cela, il est suggéré de l'ajouter immédiatement avant l'emplacement utilisé pour le pointeur de sommet de pile d'exécution `fp`.
2. Ajouter le traitement des fonctions locales, sans oublier de gérer leur niveau de bloc lexical. Au passage, il faut noter le niveau de déclaration de chaque variable dans l'environnement pour savoir les accéder lors de l'exécution.
3. Gérer le niveau lexical courant d'exécution. Il est important en tout point de l'exécution de connaître le niveau lexical auquel on s'exécute. D'une part, comme nous l'avons dit, la différence entre le niveau d'exécution et le niveau de déclaration d'une variable donne le nombre de liens statiques à traverser pour trouver le bloc d'activation contenant la valeur de cette variable. D'autre part, lors d'un appel de fonction, la différence entre le niveau lexical de l'appel et le niveau de déclaration de la fonction donne le nombre de niveaux de lien statique à traverser pour trouver le lien statique à mettre dans le bloc d'activation de la fonction appelée.

   Dans le programme de la figure 3.11, lorsque la procédure `p` appelle la procédure `q`, la différence entre le niveau d'appel (1) et le niveau appelé (2) donne -1, ce qui signifie que le lien statique de l'appelant doit pointer vers le bloc d'activation de l'appelé. Par contre, lorsque la procédure `r` appelle la procédure `p`, la différence est de 2. Il faut donc remonter de deux liens statiques et utiliser le lien statique du bloc d'activation ainsi trouvé pour le mettre dans le bloc d'activation de l'appelé. Voir la figure 3.12.

# Chapitre 4

# Programmation fonctionnelle

Dans ce chapitre, nous décrivons l'implantation de langages fonctionnels de la famille Lisp et Scheme. Ces langages se situent dans le prolongement naturel de ce que nous avons vu jusqu'à présent. Ils lèvent cependant plusieurs restrictions, ce qui nécessite l'adoption de nouvelles techniques d'implantation.

## 4.1 Qu'est-ce que la programmation fonctionnelle ?

La programmation fonctionnelle se caractérise d'abord par l'adoption de la fonction comme entité de plein droit. Ils permettent donc de créer des fonctions à l'exécution, de les passer en paramètre, de les rendre comme résultat et de les ranger dans les variables. La fonction devient donc une valeur comme les autres. Ce principe de base supporte un modèle de programmation faisant une place considérable aux fonctions d'*ordre supérieur*. Une fonction d'ordre supérieur est simplement une fonction qui en accepte une autre en paramètre. Derrière cette simplicité apparente se cache un mécanisme très puissant où les traitements les plus divers sont souvent modélisés comme l'application de fonctions à toutes les données d'une collection. Le prototype de la fonction d'ordre supérieur est `map` :

```
(define (map f list)
  (if (null? list)
    '()
    (cons (f (car list)) (map f (cdr list)))))
```

La fonction `map` prend en paramètre une fonction `f` et une liste `l` ; `map` applique `f` à chaque élément de `l` et retourne la liste des résultats ainsi obtenus. La paramétrisation de la fonction de traitement appliquée globalement sur les listes donne une puissance d'expression remarquable. Elle permet d'écrire très succintement ce qui prendrait des dizaines voire quelques centaines de lignes dans un langage impératif comme C, tout en produisant des fonction d'ordre supérieur réutilisables.

### 4.1.1 Le $\lambda$-calcul

Le fondement de la programmation fonctionnelle est le $\lambda$-calcul, qui est un outil formel permettant d'étudier systématiquement la notion de calcul dans un cadre minimaliste. Le $\lambda$-calcul pur représente un programme par un $\lambda$-terme pur et le calcul par la *réduction*. Un $\lambda$-terme pur est défini par la syntaxe abstraite suivante :

$$e \quad ::= \quad id \mid \lambda id.e \mid (e\ e)$$

Une expression ($e$) est soit une référence à une variable, soit une abstraction (fonction) ou encore une application. Le $\lambda$-calcul est dit *Turing-équivalent*, ce qui veut dire en termes plus simples qu'il permet d'exprimer n'importe quelle fonction calculable, tout comme la machine de Turing. Il peut paraître étonnant qu'avec une syntaxe aussi pauvre, ne contenant aucune constante par exemple, permette d'exprimer tout calcul. En fait, il a été montré que l'on peut exprimer sous forme de fonctions les constantes usuelles, comme les booléens et les entiers naturels. Par commodité cependant, on introduit généralement ces constantes directement de façon à travailler sur des $\lambda$-termes plus facile à lire et à comprendre. On peut considérer qu'un langage fonctionnel n'est rien d'autre qu'un $\lambda$-calcul pur auquel on a adjoint des constantes appropriées.

La notion de calcul est fondée sur la réduction, et en particulier la $\beta$-réduction. L'idée de la $\beta$-réduction consiste à appliquer une abstraction sur un argument en substituant le paramètre réel au paramètre formel dans le corps de l'abstraction. Pour une abstraction $\lambda x.e$ et un paramètre réel $e_1$, cette substitution est notée $e[e_1/x]$. Par exemple, l'expression $(\lambda x.x * x)5$ se $\beta$-réduit en $5 * 5$. Notons que l'opérateur $*$ ne fait pas partie du $\lambda$-calcul pur ; son introduction nécessite de prévoir une règle de réduction qui lui soit particulière pour exécuter la multiplication. Ces règles sont appelées *$\delta$-règles*.

Un calcul consiste à appliquer la $\beta$-réduction autant de fois que possible, passant d'un $\lambda$-terme à un autre jusqu'à ce que l'on obtienne un terme réduit, une *forme normale*. Un $\lambda$-terme peut posséder plusieurs sous-expressions réductibles, appelées *redex* (pour *reducible expressions*). Par exemple, le $\lambda$-terme $((\lambda x.x * x)((\lambda y.y + 1)10))$ possède deux sous-expressions réductibles : l'application de la première abstraction $(\lambda x....)$ ou l'application de la seconde abstraction $(\lambda y....)$. À chaque étape d'un calcul, il faut donc choisir un *redex* à $\beta$-réduire. Ce choix n'est pas anodin. Une question essentielle est de savoir si en faisant des choix de *redex* différents deux calculs peuvent réduire une même $\lambda$-terme à des formes normales différentes. Heureusement, ce n'est pas le cas. Le théorême de Church-Rosser nous assure qu'il existe au plus une forme normale pour tout $\lambda$-terme, mais que parfois il n'en existe pas. En fait, la $\beta$-réduction peut boucler sur certains $\lambda$-termes en fonction des choix de *redex* à chaque étape, ce qui est évidemment ennuyeux.

Il existe plusieurs stratégies de choix de *redex*, dont les plus connues sont l'*ordre normal* et l'*ordre applicatif*. En ordre normal, on choisit toujours le *redex* le plus à gauche et le plus externe. En ordre applicatif, on choisit toujours le *redex* le plus à gauche mais le plus interne. L'ordre applicatif correspond au mode de passage de paramètres par valeur que nous avons vu au chapitre précédent. En effet, on ne réduira une application que lorsque tous les arguments seront réduits, puisque ces derniers sont plus internes. L'ordre normal correspond lui au passage par nom puisque les arguments sont réduits après la réduction de l'application.

Il est démontré que l'ordre normal est complet au sens où si une forme normale existe pour un $\lambda$-terme, l'ordre normal nous permettra de la trouver. L'ordre applicatif n'est pas complet puisqu'il peut boucler sur des $\lambda$-terme possédant une forme normale. L'ordre normal est aussi appelé évaluation paresseuse puisque les *redex* sont évalués uniquement au besoin. Cependant, comme on l'a vu précédemment, l'implantation de la paresse dans un langage réel est coûteuse... Outre la liaison au contexte de définition que nous avons vu, il faut aussi éviter de réduire plusieurs fois un même *redex* auparavant propagé par $\beta$-réduction.

L'ordre normal et l'ordre applicatif ont donné naissance à deux grandes familles de langages fonctionnels : les langages paresseux dont Miranda et Haskell sont les principaux représentants, et les langages applicatifs, dans la tradition de Lisp, dont Common Lisp, Scheme et ML sont aujourd'hui les principaux représentants. Les langages paresseux, le plus souvent également fonctionnels purs en ce sens qu'il n'admettent pas l'affectation des variables, sont généralement implantés selon un approche dite de réduction de graphe. L'idée est de compiler le programme sous la forme d'un graphe d'expressions ; le calcul consiste alors à réduire ce graphe à une valeur, le résultat du calcul. Ce type d'implantation est aux antipodes de ce que nous avons vu jusqu'à présent. Nous n'en dirons donc pas plus ici.

L'implantation des langages applicatifs se situe plus directement dans la lignée des implantations de langages vus jusqu'à maintenant, à ceci près que certaines hypothèses de ces dernières sont remises en cause, comme nous le verrons dans la suite de ce chapitre.

### 4.1.2 Inférence de types

Le $\lambda$-calcul typé a aussi servi de fondement à l'étude de la théorie du typage dans les langages de programmation fonctionnels mais aussi maintenant des langages à objets. L'introduction de constantes dans le $\lambda$-calcul pose des problèmes car une mauvaise construction peut entraîner des erreurs. Si un $\lambda$-terme $3 + 5$ apparait légitime, $(0\ x)$ est syntaxiquement correct mais ne peut être réduit puisque 0 n'est pas une abstraction. Le $\lambda$-calcul typé cherche à détecter ces erreurs en associant à chaque $\lambda$-terme un type qui est celui de son résultat. Une abstraction reçoit un type fonctionnel, noté $\tau_1 \rightarrow \tau_2$ où $\tau_1$ et $\tau_2$ sont les types de l'argument et du résultat respectivement. Par exemple, une fonction acceptant un entier en argument et retournant un réel aura pour type `int` $\rightarrow$ `float`.

La notion de typage dans les langages fonctionnels a connu un fort retentissement lorsque Hindley et Milner ont indépendamment découvert qu'il est possible d'attribuer automatiquement les types à tout $\lambda$-terme par une analyse statique, appelée *inférence de type*, sans donc exiger de déclarations de la part du programmeur. De plus, le système de types de Hindley-Milner permet l'inférence de types polymorphiques fondée sur l'utilisation de variables de types. Prenons par exemple la fonction identité $\lambda x.x$. Quel est son type ? Si je lui soumets un entier, elle me retournera un entier, et si je lui soumets un réel, elle me rendra un réel. Son type est polymorphique et noté $\alpha \rightarrow \alpha$ où $\alpha$ est une variable de type. Ce type indique que pour tout type $\alpha$ de l'argument à la fonction identité, le type du résultat sera aussi $\alpha$.

Ce type de polymorphisme est appelé *polymorphisme paramétrique* et il se caractérise par le fait que la fonction polymorphique applique exactement le même algorithme quel que soit le type de ses arguments polymorphiques. La liaison tardive de la programmation par objets dont nous discuterons au prochain chapitre réalise plutôt un *polymorphisme ad hoc*, de même

nature que la surcharge des opérateurs, où une même fonction exécute un algorithme différent selon le type de ses arguments polymorphiques. Il est en effet évident que l'addition sur les entiers ne se fait pas de la même façon que l'addition sur les réels. On parle alors de surcharge de l'opérateur +. De la même façon, la plupart des classes Smalltalk possèdent une méthode `printOn :` mais dont les algorithmes sont bien sûr différents les uns des autres.

La procédure d'inférence de types dans le système de Hindley-Milner est extrêmement intéressante. Elle s'apparente à une preuve dans un certain système logique. Le génie de Hindley et Milner fut de proposer un système de type assez puissant pour permettre le polymorphisme paramétrique tout en assurant la décidabilité de la procédure d'inférence de type, c'est-à-dire le fait que tout terme sans erreur puisse être typé automatiquement. Aujourd'hui, on comprend mieux le défi qu'ils ont relevé puisque les chercheurs tentent actuellement d'obtenir un système de types pour les langages à objets assez puissant pour exprimer son polymorphisme ad hoc ; malheureusement, jusqu'à maintenant les systèmes proposés n'ont pu allier puissance et décidabilité. Faute de temps, nous ne nous étendons pas plus sur cette question.

## 4.2 Fermetures

Comme nous l'avons dit plus haut, la programmation fonctionnelle se caractérise par le fait que les fonctions y sont des entités de plein droit. La première chose que l'on puisse faire est la création de fonction à l'exécution. Considérons Scheme pour mieux fixer les idées. En Scheme, la forme syntaxique `lambda` permet d'écrire des expressions de création d'abstraction. La fonction créée par l'expression :

```
(lambda (a b c) (- (* b b) (* 4 (* a c))))
```

est nulle autre que celle calculant le discriminant d'un polynôme du second degré de coefficients `a`, `b` et `c`. L'exécution de cette expression crée la fonction. Nous remarquons immédiatement que Scheme impose que le code de la fonction soit explicitement donné lors de l'écriture du programme. Il n'est pas possible d'écrire une expression `(lambda (a b c) x)` où `x` représenterait une expression forgée à l'exécution. Cette restriction permet de compiler le code des formes `lambda`, comme n'importe quel autre code ; lors de l'exécution il ne restera plus qu'à créer la fonction comme telle, ce qui va nécessiter, comme nous allons le voir immédiatement de la lier à son contexte de création.

Tout comme dans les langages C et Pascal vu précédemment, Scheme utilise la portée lexicale des variables. Les trois principales formes de liaison de variables sont `let`, `let*` et `letrec`. Mais dans les trois cas, il s'agit de «sucre syntaxique», puisque ces trois formes peuvent être transformées en application de fonction :

`(let ((`$x_1$ `` `e_1`) ... (`$x_n$ `` `e_n`)) e)`

se transforme en

`((lambda (`$x_1$ ... $x_n$`) e)` $e_1$ ... $e_n$`)`

alors que

`(let* ((`$x_1$ `` `e_1`) ... (`$x_n$ `` `e_n`)) e)`

se transforme en

`((lambda (`$x_1$`) ... ((lambda (`$x_n$`) e) `$e_n$`) ...) `$e_1$`)`

et finalement

`(letrec ((`$x_1$ $e_1$`) ... (`$x_n$ $e_n$`)) e)`

se transforme en

`((lambda (`$x_1$ ... $x_n$`) (set! `$x_1$ $e_1$`) ... (set! `$x_n$ $e_n$`) e) '* ... '*)`

Dans chaque cas, nous en sommes donc réduits à étudier la liaison d'une variable à une valeur par le passage de paramètre lors de l'application d'une fonction. La portée lexicale impose donc que toute variable libre dans une fonction est définie par la liaison apparaissant le plus proche syntaxiquement dans les blocs lexicaux englobant. Les blocs lexicaux de Scheme sont les formes liant des variables donc les formes `lambda` ainsi que, par sucre syntaxique, `let`, `let*` et `letrec`. Dans l'extrait de programme suivant :

```
(let ((x 10) (y 20)) ((lambda (x) (+ x y)) 100))
```

La forme `let` introduit deux liaisons pour les variables `x` et `y` et forme le bloc lexical englobant pour son corps. Ce corps est formé d'une application du résultat d'une $\lambda$-expression à la valeur 100. Cette $\lambda$-expression introduit une nouvelle liaison pour la variable `x` qui, par le passage de paramètre est liée à 100. La variable `y` du corps de la fonction est libre dans la $\lambda$-expression ; la liaison utilisée pour trouver sa valeur est donc celle introduite par le bloc lexical englobant, donc le `let` liant `y` à 20. Le résultat de l'expression est finalement 120.

De cet exemple, il apparait clairement qu'une fonction doit être exécutée dans l'environnement de définition de cette fonction. Comment retrouver cet environnement ? L'idée est de capturer dans la représentation de la fonction son environnement de définition. Cette représentation de la fonction est justement appelée *fermeture*, car les variables libres de la fonction sont liées dans l'environnement de définition ; la fermeture est donc complète en ce qui concerne les liaisons de variables.

### 4.2.1 Un interprète pour un petit $\lambda$-calcul

Nous allons maintenant implanter un petit $\lambda$-calcul différant du $\lambda$-calcul pur que par l'extension des fonctions à plusieurs arguments, par l'ajout des constantes numérique et booléennes et de deux expressions spécifiques pour vérifier l'égalité de deux valeurs et pour additionner des nombres.

La figure 4.1 présente les fonctions principales de cette implantation. La fonction `inter` évalue une expression dans le contexte d'un environnement ; elle est chargée du décodage syntaxique et de l'aiguillage vers l'une des fonctions suivantes réalisant la sémantique du langage. La fonction `eval-constante` retourne simplement la constante elle-même comme résultat. En cela, nous utilisons directement les constantes du langage d'implantation, ici Scheme, pour représenter les constantes du langage implanté, ici notre petit $\lambda$-calcul.

La fonction `eval-variable` retourne la valeur qu'associe l'environnement courant à la variable en question ; nous utilisons les symboles Scheme pour représenter nos variables. La figure 4.2 présente les fonctions de gestion de l'environnement. Nous le représentons par une liste d'association de Scheme.

```
(define (inter exp env)
  (cond ((constante? exp)   (eval-constante exp env))
        ((variable? exp)    (eval-variable exp env))
        ((abstraction? exp) (eval-abstraction (cadr exp) (caddr exp) env))
        ((delta-regle? exp) (delta-regles exp env))
        ((application? exp) (eval-application (car exp) (cdr exp) env))
        (else (error "Syntaxe inconnue : " exp))))

(define (eval-constante valeur env)
  valeur)

(define (eval-variable variable env)
  (lookup variable env))

(define (eval-abstraction formels corps env)
  (make-fermeture formels corps env))

(define (eval-application exp-fn exps env)
  (appliquer (inter exp-fn env) (eval-lis exps env) env))

(define (appliquer fermeture arguments env)
  (inter (corps fermeture)
         (extend (environnement fermeture) (formels fermeture) arguments)))

(define (eval-lis exps env)
  (if (null? exps)
    '()
    (cons (inter (car exps) env) (eval-lis (cdr exps) env))))

(define (delta-regles exp env)
  (cond ((egalite? exp) (equal? (inter (cadr exp) env)
                                (inter (caddr exp) env)))
        ((addition? exp) (+ (inter (cadr exp) env)
                            (inter (caddr exp) env)))
        (error "Delta-regle inconnue : " exp)))
```

FIG. 4.1 – Interprète pour un petit $\lambda$-calcul

La fonction `eval-abstraction` se charge de l'évaluation de la forme `lambda` créant une fermeture. Les fermetures sont représentées par une liste dont les deuxième, troisième et quatrième éléments sont respectivement la liste des paramètres formels, le corps et l'environnement de définition de la fonction. La figure 4.3 présente les fonctions implantant le type de données *fermeture* pour notre interprète. L'évaluation d'une abstraction consiste donc simplement à créer la fermeture correspondante. On notera que la seule différence entre toutes les fermetures créées par une seule et même lambda-expression est l'environnement de définition de la fonction. C'est ainsi qu'une implantation plus efficace d'une fermeture peut compiler le code correspondant à la fonction, et créer une fermeture comme une paire dont le «car» pointe sur la fonction compilée (existant en un seul exemplaire) et le «cdr» pointe vers l'environnement de définition.

```
(define (lookup id env)
  (if (pair? env)
    (if (eq? (caar env) id)
      (cdar env)
      (lookup id (cdr env)))
    (error "Pas de telle liaison : " id)))

(define (extend env variables values)
  (cond ((pair? variables) (if (pair? values)
                               (cons (cons (car variables) (car values))
                                     (extend env (cdr variables) (cdr values)))
                               (error "Pas assez de valeurs" variables)))
        ((null? variables) (if (null? values)
                               env
                               (error "Trop de valeurs" values)))
        ((symbol? variables) (cons (cons variables values) env))))
```

FIG. 4.2 – Représentation de l'environnement

```
(define (make-fermeture formels corps env) (list 'fermeture formels corps env))

(define (fermeture? exp) (starts-with? exp 'fermeture))

(define (formels fermeture)
  (if (fermeture? fermeture)
    (cadr fermeture)
    (error "Pas une fermeture : " fermeture)))

(define (corps fermeture)
  (if (fermeture? fermeture)
    (caddr fermeture)
    (error "Pas une fermeture : " fermeture)))

(define (environnement fermeture)
  (if (fermeture? fermeture)
    (cadddr fermeture)
    (error "Pas une fermeture : " fermeture)))
```

FIG. 4.3 – Représentation des fermetures

La fonction `eval-application` évalue le sous-terme en position fonctionnelle (la première) puis sous-traite à la fonction `eval-lis` l'évaluation de la liste de sous-expressions données en paramètres réels. La sous-expression en position fonctionnelle doit nécessairement retourner une fermeture. L'application, réalisée par la fonction `appliquer`, évalue le corps de la fonction dans l'environnement obtenu par l'extension de l'environnement de définition de la fonction conservé dans la fermeture en liant chaque paramètre formel au paramètre réel correspondant.

Pour organiser notre interprète de façon modulaire, le traitement des formes syntaxiques étrangères au $\lambda$-calcul pur est sous-traité à la fonction `delta-regles`.

### 4.2.2 Environnements à durée de vie illimitée

Considérons l'expression suivante :

```
(lambda (x y)
  (if (>= x 0)
    (lambda (z) (+ z x))
    (lambda (z) (+ z y))))
```

La fonction extérieure retourne une fonction en résultat. La fonction retournée dépend du signe de `x` ; si `x` est positif ou zéro, la fonction retournée ajoute `x` à son argument, sinon la fonction retournée ajoute plutôt `y` à son argument. Que se passe-t-il lors de l'appel de cette fonction ? Que `x` soit positif ou négatif, le résultat retourné sera une fermeture dépendant des liaisons des variables `x` ou `y` introduite par l'activation de la fonction extérieure. Les variables `x` et `y` sont locales à la fonction extérieure, pourtant leurs liaisons doivent survivre à la fin de l'activation de la la fonction les ayant introduites.

Cet exemple académique montre que lorsqu'on permet de retourner des fonctions en résultat, la portée lexicale impose que les environnements aient une durée de vie illimitée et non dynamique. Dans l'interprète précédent, l'exécution de la fonction extérieure ajoute à l'environnement de définition les liaisons de `x` et `y` et cet environnement est capturé par l'exécution de l'une ou l'autre des formes `lambda`. C'est ainsi que se manifeste la survie de l'environnement à la fonction l'ayant créé.

Nous avons vu au chapitre précédente que Pascal ne permet pas de retourner les fonctions en résultat précisément à cause de ce problème. En Pascal, les variables locales sont allouées dans le bloc d'activation de la fonction les déclarant. Le bloc d'activation a une durée de vie dynamique, étant détruit à la fin de l'activation de la fonction. Toute capture d'une liaison sortant de la durée de vie dynamique de la fonction l'ayant créé y est donc proscrit. Pour permettre la capture de liaisons à durée de vie illimitée, il faut prévoir l'allocation de ces liaisons en dehors de la pile d'exécution.

### 4.2.3 Stratégies d'implantation

Pour l'essentiel, les techniques d'implantation utilisées jusqu'ici s'applique parfaitement au cas des environnements à durée de vie illimitée. Le passage de la liste d'association aux blocs d'activation puis l'ajout de lien statique pour gérer les environnements lexicaux englobants sont aussi utiisés. La principale différence est la survie potentielle des environnements lors du retour d'une fonction en résultat.

Une approche consiste à allouer les environnements dans le monceau plutôt que dans la pile d'exécution. C'est exactement ce que fait notre interprète précédent en utilisant une liste d'association pour représenter les environnements. L'allocation des paires par la fonction `cons` se fait effectivement dans le monceau. Il est cependant bien connu que l'allocation dynamique en monceau coûte plus cher que l'allocation en pile. Cela est d'autant plus vrai que la plupart des processeurs physiques offrent aujourd'hui des instructions spécialisées dans la gestion de la pile d'exécution, la rendant encore plus attractive. De plus, des optimisations bien connues sont basées sur l'hypothèse d'une gestion en pile. Comme la plupart des liaisons introduites dans un programme n'ont en réalité qu'une durée de vie dynamique, l'objectif des implanteurs

a été de maintenir l'allocation en pile la plupart du temps, et de recourir à l'allocation dans le monceau uniquement lorsque cela devient nécessaire à cause d'une capture pour une durée de vie illimitée.

Les liens statiques créent un environnement en profondeur où le coût d'accès aux variables non-locales est proportionnel à la profondeur de la déclaration de la variable par rapport au point de référence. Cette approche favorise la rapidité de création des environnements au détriment de l'accès. Une alternative consiste à créer des fermetures à environnements plats, capturant les liaisons nécessaires à la fermeture, et uniquement celles-là. Une technique apparentée est utilisée par exemple en Smalltalk pour les fermetures qui n'affectent pas de nouvelles valeurs aux variables capturées. Les valeurs des variables sont copiées dans la fermeture, ce qui donne un accès très rapide. Par contre, le coût de création de la fermeture est plus élevé. On remarque donc encore une fois qu'il est difficile d'avoir le beurre et l'argent du beurre...

## 4.3 Continuations

Considérons la fonction factorielle récursive bien connue :

```
(define fact
  (lambda (n)
    (if (zero? n)
      1
      (* n (fact (- n 1))))))
```

Il est possible d'utiliser une dérivation pour décrire le calcul réaliser par `fact`, par exemple sur le paramètre réel `4` :

```
    (fact 4)
⇒   (* 4 (fact 3))
⇒   (* 4 (* 3 (fact 2)))
⇒   (* 4 (* 3 (* 2 (fact 1))))
⇒   (* 4 (* 3 (* 2 (* 1 (fact 0)))))
⇒   (* 4 (* 3 (* 2 (* 1 1))))
⇒   (* 4 (* 3 (* 2 1)))
⇒   (* 4 (* 3 2))
⇒   (* 4 6)
⇒   24
```

Chaque appel à la fonction `fact` se fait dans un *contexte*. Si on considère le calcul de `(fact 0)`, il se fait dans le contexte `(* 4 (* 3 (* 2 (* 1 □))))` où □ représente l'expression `(fact 0)` ou plus précisément la valeur attendue par le contexte comme résultat de l'expression `(fact 0)`. Un contexte se représente facilement sous la forme d'une $\lambda$-expression :

```
(lambda (v) (* 4 (* 3 (* 2 (* 1 v)))))
```

où le paramètre `v` représente la valeur à insérer dans le contexte et où l'application de la fonction ainsi créée revient à relancer l'exécution du contexte. Soit `k` cette $\lambda$-expression, elle permet de traduire l'expression `(* 4 (* 3 (* 2 (* 1 (fact 0)))))` en `(k (fact 0))`. De fait, si nous généralisons, chacune des lignes ci-haut contenant un appel à `fact` sous la forme

(*k* `(fact` *n*`)`) avec les valeurs de *k* et *n* appropriée. Cette forme nous permet d'introduire une nouvelle définition de `fact` en style *passage à la continuation* (ou CPS pour *continuation-passing style*). Le style passage à la continuation est basé sur l'introduction de la continuation comme un paramètre explicite de chaque fonction, sur le fait de passer le résultat de la fonction directement à la continuation lorsque ce résultat est obtenue par exécution d'une expression et sur le fait de construire une nouvelle continuation à chaque fois qu'on appelle une fonction pour poursuivre le calcul. La fonction `fact-cps` prend donc un paramètre `k` pour la continuation explicite et gère explicitement cette continuation dans son corps :

```
(define fact-cps
  (lambda (n k)
    (if (zero? n)
      (k 1)
      (fact-cps (- n 1) (lambda (v) (k (* n v)))))))
```

Il est intéressant de noter que la transformation d'un programme en style direct (normal) en style passage à la continuation est automatisable. Mais qu'est-ce donc qu'une continuation ? Est-ce une simple fonction ?

Pour répondre à cette question, considérons une nouvelle forme de procédure appelée *procédure d'échappement*. Une procédure d'échappement est simplement une procédure dont le résultat, lorsqu'elle est appelée devient le résultat de l'ensemble du calcul en cours. Soit par exemple une procédure d'échappement `echappe-*` qui réalise la multiplication de ses arguments. Le résultat de l'expression :

```
(+ 2 (* 3 (echappe-* 4 5)))
```

est donc 20 puisque la multiplication de 4 et 5 donne 20 et, puisque `echappe-*` est une procédure d'échappement, le reste du calcul est tout simplement ignoré.

Par définition, une continuation est un contexte représenté par une procédure d'échappement. Plus généralement, en tout point de programme, la continuation représente le reste de l'exécution du programme en ce point. Dans les implantations que nous avons vus jusqu'à maintenant, la continuation est représentée par la pile d'éxécution. En effet, examinons l'accumulation des continuations intermédiaires dans le calcul de `(fact-cps 3 ik)` où `ik` est la continuation identité représentée par a procédure d'échappement `(lambda (v) v)`. Par dérivation, on obtient :

```
    (fact-cps 3 (λv. v))
⇒   (fact-cps 2 (λv. ((λv. v) (* 3 v))))
⇒   (fact-cps 1 (λv. ((λv. ((λv. v) (* 3 v))) (* 2 v))))
⇒   (fact-cps 0 (λv. ((λv. ((λv. ((λv. v) (* 3 v))) (* 2 v))) (* 1 v))))
⇒   ((λv. ((λv. ((λv. ((λv. v) (* 3 v))) (* 2 v))) (* 1 v))) 1)
⇒   ((λv. ((λv. ((λv. v) (* 3 v))) (* 2 v))) (* 1 1))
⇒   ((λv. ((λv. ((λv. v) (* 3 v))) (* 2 v))) 1)
⇒   ((λv. ((λv. v) (* 3 v))) (* 2 1))
⇒   ((λv. ((λv. v) (* 3 v))) 2)
⇒   ((λv. v) (* 3 2))
⇒   ((λv. v) 6)
⇒   6
```

L'imbrication des continuations successives fait clairement apparaître la discipline de pile que nous avons déjà rencontrée. Le fait est que les continuations peuvent être représentées par des procédures d'échappement ou comme des structures de données. Dans ce second cas, elles sont représentées par un enchainement de blocs d'activations attendant un résultat et représentant le point de programme où le calcul doit reprendre dans la fonction avec le résultat reçu de l'appelé.

Le principal intérêt du style de programme en passage à la continuation est le fait qu'il rende explicite le séquencement de tous les calculs. Chaque opération apparait alors explicitement avec son contexte d'exécution qui attend son résultat avant de poursuivre le calcul avec la prochaine opérations. Par exemple, les langages de programmation laissent la plupart du temps non-spécifié l'ordre d'évaluation des paramètres réels lors d'un appel de fonction. En style passage à la continuation, l'ordre d'evaluation doit apparaître explicitement. Ce facteur est d'ailleurs la raison pour laquelle la forme CPS est parfois utilisée comme forme intermédiaire faisant apparaître l'ensemble du flôt de contrôle explicitement. Par contre, toute cette information devenue explicite rend les programmes très lourds, ce qui limite une utilisation de ce style dans les programmes écrits par les humains (et non générés automatique par une transformation comme dans le cas de la forme intermédiaire dans les compilateurs).

L'utilisation du concept de continuation ne se limite pas au style passage à la continuation. Plutôt que de les faire apparaître ainsi, un langage peut fournir des primitives permettant de capturer la continuation courante en un point de programme et de les invoquer par la suite. Le principal intérêt d'une telle manipulation des continuations réside dans la capacité qu'elle donne d'exprimer toutes sortes de structure de contrôle. En particulier, elles ont été utilisées pour modéliser les structures de contrôle courantes comme l'alternative et l'itération mais aussi des structures de contrôle quasi-parallèle comme l'échappement lors d'erreur ou les coroutines. D'un point de vue puissance d'expression mais aussi de difficulté d'implantation, on distingue deux types de continuations : les continuations à durée de vie dynamique et les continuations à durée de vie illimitée. Nous les étudions maintenant tour à tour.

### 4.3.1 Continuations à durée de vie dynamique

Capturer les continuations permet de manipuler le flôt de contrôle[1]. Ces manipulations ont un coût si elles peuvent se faire sans aucune contrainte. Capturer une continuation veut dire capturer un état de la pile d'exécution. Cette capture ne détruit pas la continuation faisant partie du calcul, et si le calcul rend une valeur à cette continuation, elle va s'exécuter normalement. L'invocation explicite d'une continuation consiste à réactiver un état de pile en lui soumettant une valeur pour jouer le rôle de la valeur qu'il attend. Que se passe-t-il si par le flôt normal de contrôle cet état de pile a déjà été utilisé ? L'invocation explicite doit-elle déclencher une erreur ? Doit-on toujours assurer qu'une continuation capturée puisse être réinvocable à volonté ? Ces deux options se retrouvent dans les choix de conception représentées respectivement par les continuations à durée de vie dynamique et les continuations à durée de vie illimitée.

Nous étudions maintenant deux couples de formes syntaxiques permettant de capturer et d'invoquer des continuations à durée de vie dynamique.

[1] Cette partie du document est fortement inspiré du livre de Christian Queinnec (voir la bibliographie en fin de document).

### Formes `catch` et `throw`

Parmi les différentes primitives proposées, les formes `catch` et `throw`. La première permet de capturer une continuation :

`(catch` *étiquette formes ...*`)`

tandis que la seconde sert à l'invoquer :

`(throw` *étiquette forme*`)`.

La sémantique de la forme `catch` consiste à évaluer son premier argument `étiquette` dont la valeur est associée à la continuation courante. Cette liaison se fait dans un environnement, mais un environnement différent de celui des variables normales puisqu'il n'est pas question d'affecter une nouvelle continuation à une étiquette.

La sémantique de la forme `throw` consiste à évaluer son argument *étiquette*, à la valeur de laquelle doit avoir été associée une continuation dynamiquement englobante par une forme `catch`. Si on trouve une telle continuation, le corps de la forme `catch` qui l'a capturée est interrompu et sa valeur devient celle de la *forme* du `throw`.

En pratique, le `catch` marque un niveau de pile et un point dans l'exécution du programme, puis lance l'exécution de son corps. Pendant l'exécution de ce corps, il est possible de rencontrer une forme `throw` référant à cette marque. Le corps du `throw` est évalué et son résultat est directement transmis au niveau marqué pour y relancer l'exécution. L'utilisation typique de `catch` et `throw` vise à sortir rapidement d'un calcul profond, soit pour retourner directement le résultat soit pour déclencher une erreur le cas échéant.

Prenons l'exemple de la recherche d'un symbole dans un arbre binaire. La recherche se fait par une suite d'appels récursifs, et lorsque la valeur est trouvé, il est plus rapide de retourner directement la valeur au premier appelant que de la fair percoler lentement au fil des retours successifs des appels récursifs.

```
(define (find-symbol id tree)
  (define (find tree)
    (if (pair? tree)
      (or (find (car tree)) (find (cdr tree)))
      (if (eq? tree id) (throw 'find #t) #f)))
  (catch 'find (find tree)))
```

Comme son nom l'indique, la forme `catch` attrape la valeur que lui envoie `throw`.

### Formes `block` et `return-from`

La recherche de la liaison introduite par un `catch` lors d'un `throw` est un excellent exemple de portée dynamique. Il faut rechercher dans la pile la plus récente liaison de l'étiquette donnée par le `throw`. Les formes `block` et `return-from` introduisent une portée lexicale qui permet de déterminer quelle sera la continuation invoquée par un `return-from` en recherchant la continuation capturée par un `block` dans le texte du programme. La portée lexicale des étiquettes

offre exactement les mêmes avantages de lisibilité que la portée lexicale des identificateurs de variables.

La forme `block` a pour syntaxe :

`(block` *étiquette formes*`)`

alors que la forme `return-from` a pour syntaxe :

`(return-from` *étiquette formes*`)`

La forme `block` introduit une liaison lexicale de son étiquette avec la continuation courante ; l'étiquette de la forme `return-from` doit référer à une étiquette d'une forme `block` lexicalement visible.

Si on reprend l'exemple de la recherche dans un arbre binaire cette fois en utilisant `block` et `return-from`, on obtient :

```
(define (find-symbol id tree)
  (block find
    (letrec ((find (lambda (tree)
                     (if (pair? tree)
                       (or (find (car tree)) (find (cdr tree)))
                       (if (eq? tree id) (return-from find #t) #f)))))
      (find tree))))
```

Observons qu'il ne s'agit pas d'une banale traduction de l'implantation précédente avec `catch` et `throw` car la forme `return-from` doit se trouver dans la portée lexicale du `block`, ce qui n'est pas le cas du `throw` par rapport au `catch` dans la fonction précédente.

L'implantation du couple `block`/`return-from` est très efficace. La forme `block` mémorise la hauteur de pile et la lie à son étiquette ; la forme `return-from` fournit le résultat de sa forme interne comme résultat de l'ensemble de la forme du `block` correspondant et rajuste la hauteur de pile à celle liée à l'étiquette. Il suffit donc de quelques instructions machine dans chaque cas. Par rapport au couple `catch` et `throw`, il n'est plus nécessaire de faire la recherche dynamique de l'étiquette ni de vérifier qu'il existe bien une telle étiquette puisque ces deux opérations sont réalisées à la compilation.

### 4.3.2 Continuations à durée de vie illimitée et `call/cc`

Les continuations référencées par `catch` et `throw` ou par `block` et`return-from` ont une durée de vie dynamique. Elles ne sont donc référençables que si elles n'ont pas déjà été invoquées. Dans le cas du couple `block`/`return-from`, cette propriété est imposée par la portée lexicale de la liaison de l'étiquette. Dans le cas de `catch` et `throw`, l'invocation d'une continuation obsolète resultera en une erreur lors de la recherche de l'étiquette suite au `throw` erroné.

Scheme propose des continuations à durée de vie illimitée capturable à l'aide de la forme `call-with-current-continuation` (ou plus succintement `call/cc`). La spécification de `call/cc` est un peu complexe. Il s'agit donc d'une forme capturant sa continuation d'appel. Soit `k` cette continuation d'appel, on a donc :

$_k$`(call/cc ...)`

Une fois capturée la continuation d'appel, deux questions se posent au concepteur du langage : comment représenter la continuation et comment la fournir au programme ? La représentation utilise évidemment une fonction d'échappement. La fonction est l'outil naturel en programmation fonctionnelle. Pour ce qui est de fournir cette fonction au programme maintenant, les concepteurs de Scheme ont choisi de la passer en paramèter à une fonction d'un paramètre qui est fournie comme argument de la forme `call/cc`. On obtient donc :

$_k$`(call/cc (lambda (k) ...))`

On note que cette approche réifie la continuation comme une entité de plein droit dans le langage, c'est-à-dire la fonction. De plus, la fonction recevant la continuation en paramètre est évaluée dans le contexte de cette même continuation. La capture d'une continuation ne la fait pas disparaître... Si le calcul de la forme `call/cc` se faire sans rupture de contrôle, son résultat sera tout simplement passée à la continuation d'appel qui sera invoquée normalement.

Reprenons l'exemple de la recher de valeur dans un arbre binaire et réalisons-le avec la forme `call/cc` :

```
(define (find-symbol id tree)
  (call/cc
    (lambda (exit)
      (define (find tree)
        (if (pair? tree)
          (or (find (car tree)) (find (cdr tree)))
          (if (eq? tree id) (exit #t) #f)))
      (find tree))))
```

Dans cet exemple, l'intérêt des continuations à durée de vie illimitée n'apparait pas clairement, puisque la continuation est toujours invoquée dans la durée de vie dynamique de sa capture. L'exemple suivant illustre comment il est possible de coder une structure de contrôle récursive avec les continuations à durée de vie illimitée, et ce sans récusivité explicite. Il s'agit d'une nouvelle version de la factorielle :

```
(define (fact n)
  (let ((r 1) (k 'void))
    (call/cc (lambda (c) (set! k c) 'void))
    (set! r (* r n))
    (set! n (- n 1))
    (if (= n 1) r (k 'recurse))))
```

Ici la continuation enclôt la liaison de la variable `k` dont la valeur est la continuation elle-même. Cette circularité n'est pas très surprenante, puisque toute récurivité exige une forme de circularité quelque part. Il est à noter que si la continuation représente la suite du calcul au point de sa capture, ce qui inclut des liaisons de variables, elle ne va pas jusqu'à exiger que les valeurs de ces variables soient les mêmes au moment de l'invocation qu'elles ne l'étaient lors de la capture. C'est cette dernière propriété qui permet à l'exemple de la factorielle de

fonctionner correctement, car à chaque nouvelle invocation de la continuation liée à `k`, les valeurs de `r` et `n` changent de telle façon que le calcul va s'arrêter et retourner la factorielle de l'argument initial à `fact`.

L'implantation de continuations à durée de vie illimitée est plus coûteuse car elle force à abandonner le modèle d'implantation à pile par un modèle arborescent. L'arborescence de blocs d'activation se crée lorsque l'invocation d'une continuation capturé entraine le dépilement d'un bloc d'activation par ailleurs encore accessible parce que lié à une continuation capturée elle-même encore accessible. Le bloc doit donc survivre à la fin de l'invocation qui l'a créé, et la pile peut croître à nouveau créant la forme arborescente. L'implantation efficace d'une arborescence de blocs d'activation suppose la mise en œuvre de mécanismes d'implantation subtils.

Deux langages ont popularisé l'utilisation des continuations à durée de vie illimitée : Scheme et Smalltalk (voir le prochain chapitre). Des stratégies d'implantation ont donc été développées pour ces langages. Parce que ces stratégies utilisent des variantes sur l'allocation dynamique de mémoire et de la récupération automatique, nous étudions d'abord ce domaine avant de revenir à l'examen des principales stratégies pour l'implantation des continuations à durée de vie illimitée.

## 4.4 Allocation dynamique de mémoire et glanage de cellules

Les langages fonctionnels ont été les pionniers d'un usage de l'allocation dynamique d'espace qui soit le plus transparent possible au programmeur. Quiconque a déjà programmé une application ne serait-ce que de taille moyenne en C avec de l'allocation dynamique d'espace a été confronté au problème de la récupération manuelle de l'espace alloué. Lorsque des structures chainées complexes sont produites par allocation dynamique d'espace, il est notoirement difficile de libérer cet espace correctement. Il faut s'imposer une discipline d'acier pour éviter :

1. de tenter de libérer plusieurs fois le même bloc d'espace, ce qui résulte en une erreur du système d'allocation ;
2. d'oublier de libérer de l'espace, ce qui résulte en fuite de mémoire et amener l'arrêt du programme par manque de mémoire si le taux d'allocation est élevé ou encore si le programme s'exécute sur de longue période de temps ;
3. de référencer un bloc de mémoire déjà libéré, ce qui est couramment appelé le problème des références pendouillantes («*dangling pointers*»).

Ces erreurs sont reconnues comme très difficiles à corriger et elles sont une source majeure de manque de fiabilité des applications. Il est maintenant de plus en plus accepté que la gestion de la mémoire doit se faire automatiquement, c'est à dire de laisser le soin à l'implantation du langage de déterminer quels sont les blocs de mémoire alloués encore utiles au programme et de récupérer les autres pour réutilisation ultérieure.

Déjà en Lisp, la récupération automatique d'espace par un glaneur de cellules («*garbage collector*») faisait partie intégrante de l'implantation du langage. Les premières techniques de récupération automatique implantent la technique du marquage suivi d'un recyclage. En gros l'idée est la suivante. Le programme fait de l'allocation dynamique de mémoire, essentiellement pour Lisp à l'aide de la primitive `cons` qui alloue les paires pour représenter les listes. Tant

qu'il y a de l'espace disponible, le programme peut continuer à allouer. Lorsqu'il n'y a plus d'espace disponible, une phase de récupération automatique est lancée pour trouver de l'espace qui n'est plus utilisée et donc qui pourrait servir à satisfaire la nouvelle demande d'allocation.

La phase de récupération se divise elle-même en deux parties. Notons d'abord que toute cellule (paire cons) utile en un point donnée du programme doit pouvoir être atteinte à partir d'un nombre limité d'emplacements mémoire appelés *racines* par une séquence d'opérations `car` ou `cdr`. Les racines correspondent aux registres de la machine, aux variables globales du programme et aux environnements encore accessibles dans la pile d'exécution. La première phase de la récupération consiste à marquer toutes les cellules atteignables en suivant les pointeurs depuis les racines jusqu'aux feuilles de ce graphe de cellules. Le marquage se fait en étiquettant les emplacements mémoire visités (on utilise par exemple le bit de poid fort dans les pointeurs qui contribue rarement à une valeur utile de pointeur considérant les espaces d'adressage courants). Cette phase de marquage correspond à la traversée d'un graphe ; la principale difficulté de ce parcours vient du fait qu'il doit se faire en espace constant puisque par définition le glaneur de cellules intervient lorsqu'il n'y a plus de mémoire disponible. Une technique due indépendamment à Deutsch et à Schorr et Waite consiste à utiliser le renversement des pointeurs pour effectuer le parcours en espace constant (le marquage suffit pour déterminer si une cellule a déjà été visitée).

Lorque la phase de marquage est terminée, une phase de recyclage est lancée. Cette phase balaie toute la zone d'allocation dynamique de mémoire et remet dans la liste des cellules libres toutes les cellules non-marquées à la phase précédente.

Le recyclage de la technique du marquage–balayage est coûteux car il doit parcourir toute la zone d'allocation dynamique de mémoire. Si la quantité de cellules encore utile est nettement plus petite que la quantité de cellules à récupérer, et c'est souvent le cas en pratique, le temps passé à cette partie de la récupération tend à devenir prépondérant. Comment le diminuer ?

La technique de récupération par recopie échange mémoire contre temps d'exécution. L'idée est de diviser la zone d'allocation en deux parties égales et de n'en utiliser qu'une seule à la fois pour l'allocation. Lorsque l'allocateur ne trouve plus de mémoire disponible dans la zone courante, il lance une récupération qui consiste alors à copier dans la zone non-utilisée toutes les cellules encore utiles de la zone courante. À la fin de la récupération, le programme reprend son exécution en ayant simplement échangé les rôles des deux zones d'allocation de mémoire. L'avantage de la technique de recopie est de n'avoir à parcourir que les cellules encore utiles, et non les inutiles. Un second avantage est la compaction ; à la fin de la copie, la zone de mémoire utilisée est concentrée ce qui peut améliorer l'efficcité surtout dans les systèmes à mémoire paginée. Cette approche a cependant un coût. À tout moment, seule une moitié de la mémoire disponible peut être utilisée. De plus, le fait de diviser par deux l'espace disponible peut engendrer de plus fréquents appels au récupérateur d'espace, ce qui implique un coût global de récupération (sur toute l'exécution du programme) plus grand.

Une technique ancienne, qui abaisse le coût de récupération au prix d'une hausse du coût d'allocation et d'affectation, est celle des compteurs de référence. L'idée est d'associer à chaque cellule un compteur du nombre de variables ou d'autres cellules pointant sur elle. À chaque affectation, le compteur de référence de la cellule affectée est augmenté de un alors que celui de la cellule dont le pointeur est écrasé est diminué de un. Lorsque le compteur de références devient égal à zéro, la cellule est remise dans la liste libre. Cette technique peut devenir

avantageuse lorsque l'on ne peut se permettre de longues pauses à des moments aléatoires pendant l'eécution des programmes. La dispersion des coûts doit cependant être soigneusement étudiée pour démontrer son avantage. De plus, la technique des compteurs de références a le désavantage remarquable de ne pouvoir récupérer l'espace occupé par les structures cycliques.

En 1983, Lieberman et Hewitt on remarqué dans les programmes Lisp que (1) les nouvelles cellules ont tendance à pointer sur les anciennes et (2) que les nouvelles cellules ont tendance à avoir une durée de vie plus courte que celle des cellules plus anciennes. En effet, lorsqu'une cellule est ancienne, il y a de fortes chance qu'elle fasse partie d'une structure de données importante qui sera encore utile pour la suite du programme. Les cellules nouvelles contiennent souvent des résultats intermédiaires qui ont toutes chances de devenir rapidement obsolètes et inutiles.

Ces observations ont mené aux récupérateurs générationnels qui concentrent leurs efforts sur les cellules les plus récemment allouées. Un glaneur de cellules générationnel sépare l'espace en plusieurs zones $G_i$ appelées générations, où la génération $G_n$ est la plus jeune. Les cellules dans chaque génération ont toutes approximativement le même âge, et l'observation (1) nous assure que peu de pointeurs d'une cellule de la zone $G_i$ iront vers des cellules des zones $G_j$ avec $j > i$. La plupart du temps, il suffira de ne balayer que la zone $G_n$ qui, par l'observation (2) contient la plus forte proportion de cellules à récupérer. Si cela ne suffit, on élargit le balayage aux générations précédentes. Lorsqu'une cellule de la zone $G_n$ survit à plusieurs cycles de récupération, elle est copiée vers une zone de mémoire d'une génération plus ancienne. De plus, si une cellule d'une génération plus ancienne vient à pointer vers une cellule d'une génération plus jeune, cette cellule est notée pour servir de racine lors des cycles de récupération suivants.

En pratique les techniques générationnelles fonctionnent très bien, même si la gestion des générations est un peu lourde. Il a d'ailleurs été observé que d'excellents résultats peuvent être obtenus en utilisant uniquement deux générations. Notons finalement que des mesures expérimentales montrent que le temps total de récupération dépasse très rarement 5% de la durée totale de l'exécution d'un programme, ce qui est un coût somme toute modeste en comparaison des coûts associés à la gestion manuelle de l'espace alloué dynamiquement.

## 4.5 Stratégies d'implantation pour continuations à durée de vie illimitée

La stratégie de base et en même temps la plus simple consiste à allouer tous les blocs d'activation dans le monceau et de laisser au GC le soin de récupérer la mémoire lorsqu'ils ne sont plus utiles. Lors de la capture d'une continuation, une nouvelle liaison au bloc d'activation en somme de la pile la représentant est créée. Le récupérateur de mémoire ne recyclera donc pas l'espace occupée par les blocs d'activation de cette continuation tant que cette liaison reste visible dans le programme.

Le problème avec cette stratégie est que les blocs d'activation ont une durée de vie moyenne très courte et le glanage de cellules a un coût trop important pour gérer le cas moyen efficacement. Des mesures expérimentales montrent que si le GC est particulièrement efficace, cette stratégie peut être utilisée avec un interprète de code octal, mais n'est pas utilisable pour un compilateur en code natif. Des stratégies alternatives doivent donc être trouvées.

### Stratégie spaghetti

La stratégie spaghetti, due à Bobrow, est une variante de la stratégie GC où la zone d'allocation de mémoire est divisée en deux parties ; une partie est gérée par marquage et recyclage et l'autre par des compteurs de références. Les blocs d'activation sont alloués dans la zone à compteurs de références. Bien que complexe, la stratégie spaghetti[2] est plus efficace que la stratégie GC car elle gère plus efficacement les appels de procédure et leur retour ainsi qu'un ensemble d'opérations qui leurs sont reliées. En fait, dans la cas normal où tous les blocs d'activation ont une durée de vie dynamique, la zone gérée par compteurs de références se comporte comme une pile conventionnelle.

Les stratégies suivantes sont de variantes simplifiées de la stratégie spaghetti. En fait, elles ont une performance bien inférieure à la stratégie spaghetti dans certains cas exceptionnels, mais se comportent mieux dans les cas moyens parce qu'elles simplifient leur traitement.

### Stratégie du monceau

La stratégie du monceau alloue les blocs d'activation de façon similaire à la stratégie GC, mais elle les marque d'un bit pour savoir s'ils ont été capturés ou pas. Les blocs libres sont gérés dans une seconde liste libre à laquelle sont ajoutés les blocs lorsque la procédure les ayant créés retourne, sauf dans le cas où le bloc fait partie d'une continuation qui a été capturée. Lorsqu'une continuation est capturée, tous les blocs d'activation en faisant partie sont parcourus pour les marquer comme capturés. Les blocs marqués comme capturés sont gérés par le GC normal. L'invocation d'une continuation consiste simplement à y faire pointer le pointeur de sommet de pile du processeur.

Cette stratégie fonctionne particulièrement bien lorsque tous les blocs d'activation ont la même taille (ce qui peut arriver si l'environnement n'en fait pas partie...). Une variante à marquage incrémental des blocs capturés a été proposée par Danvy, au prix de quelques instructions supplémantaires lors du retour de procédure pour marquer le bloc auquel on retourne si le bloc duquel on arrive est lui-même marqué.

### Stratégie de la pile

Seuls les blocs d'activation ayant été capturés peuvent dépasser leur durée de vie dynamique normale. L'idée de la stratégie de pile consiste donc à allouer les blocs d'activation de la continuation courante dans une zone de mémoire gérée comme une pile standard. Lorsque la continuation est capturée, la pile est copiée dans la zone d'allocation dynamique. Lorsqu'une continuation est invoquée, elle est recopiée dans la zone gérée en pile. Théoriquement, cette stratégie doit exécuter les programmes ne capturant pas les continuations aussi efficacement que dans les langages sans `call/cc` ; en pratique il y a un coût caché en termes d'optimisations qui ne peuvent plus être faites parce que les continuations peuvent être capturées.

Des variations de la stratégie de pile sont utilisées dans la plupart des implantations de Scheme et de Smalltalk courantes. Dans certaines variantes, la zone de pile ne contient que

[2] probablement en référence à la complexité de la structure créé avec des pointeurs allant d'une zone à l'autre...

quelques blocs d'activation (les plus récents), les autres étant systématiquement gérés dans le monceau. La zone de pile devient alors une espèce de fenètre sur les blocs d'activation en cours d'utilisation. Une autre variante due à Ungar consiste à reporter la copie des blocs d'activation capturés au moment où la procédure les ayant créés retourne ou bien au moment où la continuation les ayant capturés est invoquée. Cette stratégie favorise les utilisations des continuations capturées dans leur durée de vie dynamique.

Un problème particulier à cette approche est l'affectation des variables apparaissant dans les blocs d'activation. Puisqu'il y a copie des blocs lors de la capture, il faut justement éviter d'allouer les variables affectées dans les blocs d'activation. Certains Scheme allouent ces variables systématiquement dans le monceau, ce qui en augmente considérablement le coût d'utilisation, une manière détournée d'encourager un style de programmation vraiment fonctionnel.

### Stratégie pile/monceau

la stratégie pile/monceau est similaire à la stratégie de la pile, sauf que lors d'une capture de continuation, la continuation est copiée et la zone de pile est vidée, et lors d'une invocation, la zone de pile est aussi vidée. Le processeur du langage doit donc distinguer le traitement du retour de procédure selon que le bloc est dans la pile ou dans le monceau. L'allocation d'un bloc après une capture se fait dans la pile, comme dans les variantes à fenètre sur les blocs courants de la stratégie de la pile.

L'avantage de la stratégie pile/monceau est de diminuer considérablement le coût de copie des blocs d'activation. En plus, une et une seule copie existe pour chaque bloc, ce qui permet de revenir à une allocation des variables affectées dans le bloc d'activation. L'invocation des continuations est alors très rapide. La capture d'une continuation dont la plupart des blocs ont déjà été capturés est aussi très rapide, ce qui est intéressant car on a observé dans les programmes une tendance à concentrer les captures en gerbes qui capturent et recapturent presque toujours les mêmes blocs. Le désavantage de cette approche est d'empêcher certaines optimisations du compilateur, ce qui s'avère parfois pénalisant.

### Stratégie pile/monceau incrémentielle

La stratégie pile/monceau incrémentielle est une légère variante sur la stratégie pile/monceau. Lorsque le retour d'une procédure amène vers un bloc d'activation qui n'est pas dans la zone de pile, une exception est levée et son traitement ocnsiste à copier le bloc dans la zone de pile. Cette variante impose un coût de copie mais facilite certaines optimisations des compilateurs.

## Évaluation

Des expériences menés sur une implantation particulière de Scheme montrent que la stratégie pile/monceau, si elle est implantée avec soin, peut donner des performances presqu'aussi bonne que l'allocation en pile conventionnelle sur des programmes ne capturant pas les continuations, et avoir d'excellentes performances lorsque les captures apparaissent en gerbes. Ceci dit, ces résultats méritent d'être évalués dans le contexte de leur éventuelle utilisation car pour

le moment les expériences se limitent à une seule implantation et à un seul jeu de tests.

# Chapitre 5

# Programmation par objets

La programmation par objets se divise en deux courants principaux. Le premier courant insiste sur les aspects génie logiciel ; il a été marqué par les langages Simula-67 et Eiffel par exemple. Le deuxième courant a plutôt été influencé par la flexibilité et l'adaptativité des langages comme Lisp ; il a été marqué par les langages Smalltalk et CLOS. Dans les deux cas, les différences fondamentales entre les langages à objets et les langages fonctionnels et impératifs tiennent à la représentation des objets et à l'implantation de l'envoi de message.

Le coût de l'implantation de l'envoi de message tient à la liaison tardive entre l'appel et la méthode réellement appelée, ou de façon équivalente à la liaison tardive de la classe d'instanciation du receveur du message. La méthode invoquée par un envoi de message dépend de la classe d'instanciation du receveur. Cette classe est rarement connue statiquement car le résultat de l'expression dénotant le receveur n'est pas toujours instance de la même classe. Même dans les langages à objets typés, une expression devant retourner une instance d'une classe A admet toujours que lui soit substituée une instance de n'importe quelle classe B sous-classe de A. Dans les deux sections de ce chapitre, nous étudions de façon générale les implantations de C++ et de Smalltalk, en insistant sur les aspects les plus novateurs par rapport à ce que nous avons vu jusqu'à présent.

## 5.1 C++

C++ est un langage qui a été proposé par Bjarne Stroustrup en 1984-85. Sa conception a commencé par un langage plus simple, appelé «C with classes», proposé dès le début des années '80. Aujourd'hui, il y a toujours des efforts de conception faits sur C++ car le langage est en voie de standardisation.

La machine virtuelle de C++ a été historiquement fortement influencée par le fait qu'au départ, C++ a été implanté à l'aide d'un préprocesseur traduisant un programme C++ en un programme C pour ensuite appelé un compilateur C se chargeant de produire le code exécutable. Cette approche revient à dire que la machine virtuelle de C++ est essentiellement la même que celle de C, sauf pour quelques constructions permettant d'implanter les aspects proprement objet de C++.

Cette approche tient aussi au biais fondamental qui a modelé le projet C++ qui était d'avoir

une implantation imposant le minimum de surcharge (*overhead*) de travail à l'exécution, en particulier pour les envois de message. De plus, C++ adopte la même vision classique du développement d'application que C, c'est à dire une phase de compilation préalable à, et bien détachée de l'exécution du programme. Cette approche a l'avantage de bien séparer la phase de développement de celle de l'exécution, mais il est moins flexible en termes d'ajouts ou de modifications faits au programme (comme on le voit en Smalltalk). La compilation séparée de sous-ensembles de programme permet de mitiger ce problème, sans totalement l'éliminer cependant.

Pour diminuer à un minimum la surcharge de travail fait à l'exécution, il n'y a pas de secret, il faut faire le maximum de liaisons statiques. C'est dans ce sens où C++ deçoit les spécialistes de la programmation par objets. Dans sa conception même, C++ tente d'éviter le plus possible la liaison tardive inhérente à la programmation par objets. L'existence même de fonctions non-virtuelles soulève des questions sur la nature «objet» de C++. En réalité, ces fonctions ne sont là que parce que leurs appels peuvent être liés statiquement, comme le sont les appels de procédures en C. Pas besoin de faire de «recherche» dynamique comme dans le cas des fonctions virtuelles. En réalité, c'est la conséquence du fait qu'historiquement C++ vient du langage «C with classes» où les fonctions virtuelles n'existaient pas dans ce langage et qui n'avait donc pas besoin de liaison dynamique pour être implanté.

### 5.1.1 Représentation des objets

C++ a donc d'abord été implanté par un préprocesseur produisant du C. Les classes apparaissaient alors comme une simple extension des structures où il devenait possible d'avoir des champs fonctionnels (i.e. contenant des fonctions). La déclaration d'une classe `A` s'apparente alors à la déclaration d'une structure alors que l'instantiation revient à créer effectivement une valeur du type structure `A` (statiquement ou dynamiquement).

Si on raffine un peu le modèle, il n'est pas utile que chaque objet ait effectivement un champ pour chaque fonction déclarée par la classe. En effet, les champs fonctionnels n'étant pas mutables, toutes les instances d'une classe ont exactement les mêmes fonctions. Un objet peut donc se contenter de contenir uniquement les variables déclarées par la classe. On peut les voir simplement comme des structures contenant toutes les variables déclarées par la classe (y compris celles déclarées par les superclasses).

**Exemple**

```
class B {
public:
 virtual char f() { return 'B'; }
         char g() { return 'B'; }
         char testF() { return f(); }
         char testG() { return g(); }
protected:
         int x;
};

class D : public B {
```

```
public:
        char f() { return 'D'; }
        char g() { return 'D'; }
protected:
        int y;
};
```

- Une instance de la classe `B` est (un peu) comme une structure où il y a un champ x entier.
- Une instance de la classe `D` est (un peu) comme une structure où il y a deux champs `x` et `y` entiers. □

### 5.1.2 Appel de méthodes

#### Liaison statique des fonctions non-virtuelles

Nous avons dit que les fonctions non-virtuelles peuvent être traitées comme une fonction C, en termes de moment de liaison entre appel et fonction activée. Le seul problème qui se pose est le conflit de nom puisqu'il y a effectivement surcharge du nom de fonction si de méthodes du même nom sont déclarées dans plusieurs classes. Pour un nom de fonction `f`, chaque classe peut potentiellement déclarer une fonction implantant `f`. Il faut donc lever l'ambiguïté au moment de la compilation. La façon de lever l'ambiguïté est très simple. Il suffit de donner un nom différent à chacune de ces fonctions `f` ! Eh bien, pourquoi ne pas préfixer le nom de chacune de ces fonctions par le nom de la classe qui les déclare ? En C++, tout se passe comme si c'était exactement cela qui arrivait.

**Exemple** (suite)

Dans les classes `B` et `D` précédentes, la fonction `g` est non-virtuelle et elle a des implantations différentes dans chacune des deux classes. Pour lever l'ambiguïté, il suffit de nommer `B : :g()` la fonction `g` de `B` et de nommer `D : :g()` celle de la classe `D`. Soient deux variables `b` et `d` déclarées de la façon suivante :

```
B b;
D d;
```

alors, un appel `b.g()` sera compilé directement en un appel à la fonction `B : :g()` alors qu'un appel à `d.g()` sera compilé directement en un appel à la fonction `D : :g()`. Ces appels s'effectueront donc comme de classiques appels de fonction C. De même, si la variable `b` est plutôt déclarée comme un pointeur sur une instance de `B`, mais qu'elle contient effectivement une instance de la classe `D`, comme suit :

```
B *b = new D;
```

l'appel `b->g()` sera tout de même compilé comme un appel à la fonction `B : :g()`, parce que `b` est déclaré contenir un pointeur sur une instance de `B`. □

#### Liaison dynamique des fonctions virtuelles

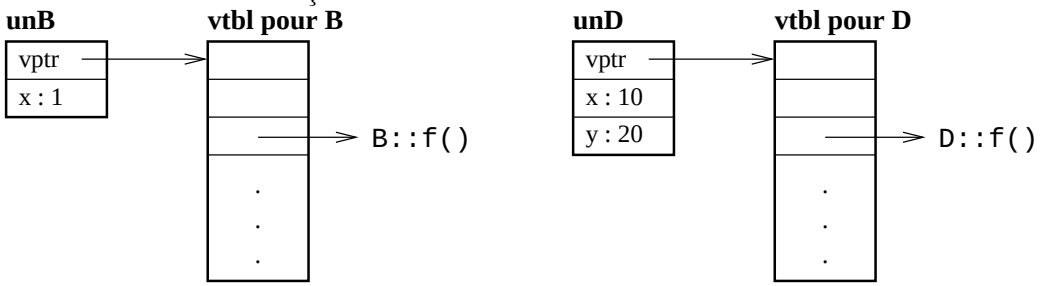
Pour les fonctions virtuelles, cependant, il n'y a pas le choix, il faut passer par une liaison tardive. Reste que C++ tient à réduire le coût de cette liaison tardive à un minimum. Grâce

au mécanisme de la table des fonctions virtuelles, C++ réussit effectivement à réduire le coût d'un appel de fonction virtuelle à deux indirections plus un calcul d'indice dans un tableau.

L'idée de cette approche consiste à donner un indice à chaque sélecteur ou nom de fonction virtuelle existant dans le programme (ou éventuellement dans une partie du programme compilée séparément). Supposons par exemple qu'il existe cent noms de fonctions virtuelles dans un programme, une approche naïve leur attribuerait les indices de 1 à 100 (ou plutôt 0 à 99). Ensuite, pour chaque classe, on construit une table de fonction virtuelle. Cette table est un vecteur de pointeurs sur des fonctions virtuelles. Dans la table associée à une classe, on met à l'indice corrspondant au nom de fonction virtuelle `f`, par exemple 2, la fonction qui doit effectivement être appelée si une instance de cette classe reçoit un message de sélecteur `f`. Enfin, on met dans chaque instance un pointeur vers la table de fonctions virtuelles associée à sa classe d'instantiation. Ainsi, un envoi de message pourra être compilé en deux indirections et un calcul d'indice sur un tableau.

**Exemple** (suite)

Voici comment seraient représentées les instances des classes `B` et `D`, en supposant que le nom de fonction virtuelle `f` reçoit l'indice 2 :




Grâce à cette représentation, un appel `b->f()` est compilé sous la forme `(*(b->vptr[2]))()`. `b->vptr` représente le pointeur sur la table de virtuelles de l'objet contenu dans `b`. On accède l'entrée 2 dans cette table qui correspond au nom de fonction virtuelle `f`. L'expression `*(b->vptr[2])` retourne donc le pointeur sur la fonction virtuelle qui doit être activée. L'activation se fait ici sans paramètres réels car la fonction est sans arguments. □

Le problème majeur de cette approche vient de l'espace nécessaire pour les tables de fonctions virtuelles. S'il y a cinquante classes dans le programme et une centaine de sélecteurs, cela nous donne 50 * 100 = 5000 mots mémoires pour représenter les tables de fonctions virtuelles. Et on ne parle là que d'un tout petit programme. L'image Smalltalk standard par exemple contient des centaines de classes et des milliers de sélecteurs.

Pour économiser l'espace, il faut trouver un moyen de compacter les tables de fonctions virtuelles. L'idée consiste à réutiliser un même indice pour deux sélecteurs (ou plus) si ces deux sélecteurs ne peuvent jamais entrer en conflit l'un avec l'autre. Ne pas entrer en conflit revient à dire que jamais une classe ne déclare en même temps de fonctions virtuelles correspondant à ces deux sélecteurs. Cette technique est appliquée et elle permet bien sûr des gains substantiels dans la taille des tables.

La question qui se pose est de savoir comment se fait l'attribution des indices aux sélecteurs dans cette approche de compaction. En fait, aussi surprenant que cela puisse paraître, ce problème d'attribution de numéro se modélise comme un problème de coloration de carte géographique. Ce problème s'énonce de la façon suivante : «Combien faut-il de couleur au

minimum pour colorier une carte géographique de telle façon que deux pays qui se touchent ne se verront pas attribuer la même couleur.» Ce problème a été longuement étudié par les théoriciens et on en connaît maintenant des solutions en passant par un représentation sous forme de graphes. Comme quoi, les problèmes théoriques qui ont l'air inutiles en pratique trouve parfois des utilisations inespérées !

## 5.2 Smalltalk-80

Le langage Smalltalk-80 est le troisième d'une lignée commencée en 1972. Ses prédécesseurs sont Smalltalk-72 et Smalltalk-76. Depuis 1983, des techniques très efficaces ont été développées pour implanter Smalltalk-80 (ci-après simplement Smalltalk). En effet, dès 1983, Deutsch et Schiffman avaient réalisé une implantation de Smalltalk qui était entre 5 et 10 fois plus lente que C optimisé. Aujourd'hui, la situation s'améliore encore.

Il faut cependant savoir que Smalltalk impose une vision de la programmation par objets qui veut que tout dans le langage soit des objets, de l'entier au descripteur de fichier, de la chaîne de caractères jusqu'au dévermineur (*debugger*), en passant bien sûr par les classes qui sont elles aussi des objets. Il n'y a, à proprement parler, aucune structure de contrôle en Smalltalk (boucle, alternative, etc.) ; l'unique moyen de déclencher des calculs est l'envoi de message. Toutes les structures de contrôle sont en effet réalisées à l'aide d'envoi de messages ! L'approche du «tout objet» fait en sorte qu'il a été possible d'implanter la plupart des outils de programmation de Smalltalk-80 (compilateur, fouineur ou *browser*, dévermineur, etc.) en Smalltalk-80 lui-même. C'est ainsi que dans l'image Smalltalk-80 (l'ensemble des objets existant au moment où on lance Smalltalk), on trouve le compilateur, le dévermineur, etc. Pas surprenant que la moindre image fasse plus de cinq mégaoctets dans ces conditions.

Par contre, c'est plutôt étonnant qu'on ait réussi à obtenir des performances qui se rapprochent autant de celles de C, toutes optimisations enclenchées. En fait, le présent texte vise à vous montrer, en gros, comment on en est arrivé à ce résultat. Il veut également tordre le coup à un mythe persistant voulant que Smalltalk soit un langage interprété. En réalité, il y a autant de compilation dans Smalltalk que dans C, et même avec des techniques beaucoup plus sophistiquées.

Il faut aussi mentionner que Smalltalk et les recherches faites autour de son implantation ont été pionniers des environnements graphiques d'aujourd'hui. Il ne fait pas oublier qu'au début des années 1980, les grands écrans à représentation binaire (*bitmap*) étaient à peu près inexistants, avec leur cortège de systèmes de fenêtres, etc. C'est en Smalltalk qu'est apparu le premier environnement de programmation à fenêtres. Les premiers Smalltalk avaient donc dû implanter tout le système de fenêtrage, qui était bien sûr écrit en Smalltalk. On raconte que Steve Jobs a adopté l'interface graphique du MacIntosh suite à une visite au laboratoire de Xerox à Palo Alto (Xerox PARC) d'où il était sorti complètement emballé. Ce n'est un secret pour personne que la convivialité du Mac est en réalité le résultat de l'utilisation (certains diraient pillage) des résultats de recherche de Xerox sur Smalltalk et autour de ce projet. Il n'y avait probablement qu'Apple cependant qui pouvait, au milieu des années 80, rendre cette technologie accessible au grand public. Heureusement que Xerox ne l'a pas empêché de le faire...

### 5.2.1 La genèse d'un mythe

Le principe fondateur de Smalltalk est donc que tout doit être objet. En termes de langage de programmation, on appelle *entité de plein droit* une entité qui est traitée comme toutes les autres entités du langage. En programmation fonctionnelle, on dit que les fonctions sont des entités de plein droit, car il est possible de les créer à l'exécution, de les ranger en mémoire (dans les variables), de les passer en paramètre ou des les retourner comme résultat d'autres fonctions. En Smalltalk, on dit que tout est traité comme des entités de plein droit parce que tout est objet, l'entité à la base du langage.

Ce principe va très loin. Normalement un objet est un vecteur de mots mémoire dans lesquels on va ranger les valeurs des variables d'instance de cet objet. De plus, le premier mot de l'objet contient l'identité (on peut le voir comme un pointeur, mais c'est une grossière simplification) de la classe d'instantiation de l'objet. C'est ce qui permet au système de retrouver la classe d'un objet lorsque ce dernier reçoit un message. Comment faire en sorte qu'un entier soit un objet dans ces conditions. Va-t-on conserver avec chaque entier un pointeur vers la classe `Integer` ? Ce serait un gaspillage terrible, et Smalltalk ne s'y résout pas. Il y a donc un truc, mais lorsque vous voyez dans un programme Smalltalk :

```
1 + 2
```

il faut savoir que cela est traité comme l'envoi à l'objet `1` d'un message `+` avec l'argument `2` !

Cela va encore plus loin. Pour exécuter les méthodes, Smalltalk a bien sûr besoin d'une pile d'exécution et de blocs d'activation. Même les blocs d'activation sont des objets ! On peut leur envoyer des messages pendant l'exécution du programme.[1] C'est d'ailleurs cela qui a permis d'écrire le dévermineur Smalltalk en Smalltalk. Pas besoin d'assembleur quand les blocs d'activations sont des objets du langage lui-même.

D'ailleurs, l'implantation de Smalltalk en entier est basée sur une machine virtuelle qui a déjà été entièrement spécifiée et écrite en Smalltalk. C'est ce qu'on va appeler par la suite la *v-machine*. Cette *v-machine* est en réalité à l'origine du mythe du Smalltalk interprété, un mythe qui a bien failli tuer Smalltalk.

Voici l'histoire. Par son principe de base (tout est objet), Smalltalk s'est compromis à faire en sorte que sa propre implantation soit le plus possible écrite en Smalltalk ; c'est la v-machine. La v-machine est un interprète de code octal qui propose une vision homogène de l'implantation de Smalltalk. Pour ceux qui connaissent, la technique d'implantation par code octal était très répandue dans les années 70. Pascal a été implanté comme cela, avec un code octal appelé pcode. L'idée est la suivante. On écrit un compilateur qui produit du code pour une machine virtuelle, en Pascal la p-machine. Pour obtenir une implantation Pascal sur une machine quelconque, il suffit d'écrire un interprète de pcode réalisant la p-machine et le tour est joué. L'avantage de cette technique est la réutilisation très facile de tout le compilateur produisant le pcode, ce qui est la partie la plus complexe. On peut alors très facilement porter

[1] Les blocs d'activation comme entités de plein droit permettent donc à un programme d'examiner son propre état d'exécution. Un ami a d'ailleurs implanter tout un système de gestion des exceptions pour Smalltalk écrit en Smalltalk. Pour ceux qui connaissent un peu la gestion des exceptions, l'accès à la pile d'exécution est crucial pour déterminer quel traitant (*handler*) d'exception est appelé lorsqu'un condition d'exception est détectée.

l'implantation de Pascal d'une machine à l'autre, simplement en réécrivant l'interprète de pcode.

Les premières implantation de Smalltalk procédaient ainsi. Un compilateur Smalltalk écrit en Smalltalk compile vers le vcode de la v-machine. Pour obtenir une implantation qui fonctionne, il suffit d'écrire un interprète de vcode, réalisation logiciel de la v-machine. Plus fort encore. Dans leur livre célèbre, Adele Goldberg et David Robson (deux pionniers de Smalltalk à Xerox Palo Alto Research Center), ont même proposé une spécification de la v-machine écrite en Smalltalk. Ce livre, intitulé «Smalltalk : the language and its implementation», était familièrement appelé le bouquin bleu (*the blue book*). Quand il est sorti, il existait à peu près que l'implantation de Smalltalk développée a Xerox PARC, et elle était difficile d'accès car réalisée sur des machines Xerox coûteuses et très peu répandues [2]. Dans sa dernière partie, les auteurs proposaient une spécification complète de la v-machine en Smalltalk, selon une approche en vogue à l'époque pour donner une sémantique précise à un langage de programmation.[3] Voyant qu'il etait simple d'implanter Smalltalk, on peut supposer que plusieurs groupes (compagnies, chercheurs) se sont lancés à faire des implantations procédant comme cela. Il suffisait d'écrire un interprète de vcode puis de rentrer le code du livre de Goldberg et Robson et le tour était joué... Le tour, ouais ! Parlons-en.

En réalité, ce ne pouvait être que catastrophique. Cette approche pour implanter Smalltalk ne pouvait mener qu'à des performances épouvantables. Est-ce pour cela que le *blue book* ne fut pas réédité ? Toujours est-il qu'il a été remplacé par le *purple book*, intitulé «Smalltalk : the language», où la partie implantation a en grande partie disparue... N'essayez donc pas de trouver le *blue book* dans les librairies, il a disparu depuis longtemps, et les exemplaires en circulation sont précieusement conservés par leurs propriétaires qui furent souvent les pionniers du domaine des objets.

### 5.2.2 La v-machine

La v-machine est construite à partir des éléments suivants :

- Un **compilateur** qui prend des méthodes écrites en Smalltalk et qui les traduit en code octal. Ce code octal est conservé dans des objets particuliers, instances de la classe `CompiledMethod`. Ces méthodes compilées sont ensuite insérées dans le dictionnaire de méthodes de la classe concernée (dictionnaires qui sont instances de la classe `MethodDictionary` bien sûr !).
- Un **interprète** de code octal qui peut exécuter le code produit par le compilateur.
- Une mémoire d'objets qui est chargée de conserver tous les objets existants dans le système pendant l'exécution, et qui sert à créer les images Smalltalk. En réalité, la mémoire d'objets est une mémoire virtuelle qui reconstruit la mémoire normale qui lui ajoute des fonctions d'accès propres aux objets (comme lire la valeur d'une variable d'instance dans un objet précis).
- Un ensemble de classes dont les instances sont des objets utilisés par l'interprète de vcode pour conserver et «exécuter» les programmes. Parmi celles-ci, il y a bien sûr les classes

[2] La v-machine était en réalité microcodée sur ces machines, ce qui lui donnait une performance acceptable.

[3] Les lispiens furent les pionniers dans ce domaine, en proposant des spécifications de Lisp écrites en Lisp. La question de savoir pourquoi une implantation d'un langage écrite dans la langage lui-même est suffisante pour être considérée complète et formelle dépasse largement le cadre du présent texte.

`CompiledMethod`, `BlockContext`, `MethodDictionary`, etc.

Il est important de comprendre que la v-machine fait partie intégrante du langage Smalltalk, au même titre que sa syntaxe par exemple. Les objets bloc d'activation, instances de la classe `BlockContext` sont des objets utilisés par la v-machine, mais puisque les programmeurs peuvent leur envoyer des messages, il faut absolument que l'implantation fournisse effectivement ces objets. C'est là le défi de l'implantation efficace de Smalltalk, car on va voir que pour optimiser Smalltalk, la seule solution consiste à le compiler directement pour la machine physique réelle. Comment concilier le fait que les objets Smalltalk représentant le code et son exécution soient toujours visibles tout en compilant le code pour une machine réelle ? C'est le problème majeur auquel on était confronté au début des années 80. Voyons comment les chercheurs se sont sortis du problème.

### 5.2.3 L'efficacité : l'implantation Deutsch-Schiffman(1982)

Bien qu'il existait des implantations à peu près efficaces de Smalltalk microcodées sur les machines Xerox au début des années 80, il y avait bien sûr beaucoup de recherche qui se faisait sur la façon d'implanter Smalltalk sur du matériel «de série» ordinaire. Parmi les approches explorées à cette époque, la révolution est venue des chercheurs Deutsch et Schiffman. L'originalité de leur approche venait du fait qu'ils voulaient conserver scupuleusement la vision «idéale» de Smalltalk, soit l'implantation par la v-machine tout en obtenant un grande efficacité. La plupart des chercheurs de l'époque devaient croire que nous en viendrions à sacrifier cette vision un peu idyllique des choses pour revenir à des techniques d'implantation plus répandues.

L'approche Deutsch-Schiffman, au contraire, propose une implantation alternative, de meilleure performance, mais qui respecte les hypothèses de base de la vision idéale de Smalltalk dont la spécification de la *v-machine* de même la spécifications des objets qui apparaissait dans l'édition originale du livre bleu de Goldberg et Robson. Le principe de base de cette approche est le suivant : il faut modifier dynamiquement la représentation des objets réalisant l'implantation (comme les blocs d'activation) de façon totalement transparente pour l'usager. L'usager doit continuer à croire que tout se passe comme dans la v-machine, mais en quelque sorte tous les coups sont permis pour optimiser les choses derrière son dos quand il ne regarde pas.

Voici les principaux «coups derrière le dos» proposés par l'approche Deutsch-Schiffman :

- Le code pour la *v-machine* est traduit dynamiquement dans du code directement exécutable par la machine physique sous-jacente. Ce code machine est conservé dans des systèmes de mémorisation (*caches*) entre les activations. Si à la prochaine activation de la méthode, le code compilé est toujours disponible, on l'utilise directement, sinon on le regénére si jamais on a dû l'effacer par manque d'espace.
- Les blocs d'activation sont représentés dans une forme orientée vers la machine (i.e. une pile comme en C ou Pascal) lorsqu'utilisés pour contenir un état d'exécution mais comme un objet lorsqu'ils sont utilisés comme tels. (Note : ceci suppose aussi une «traduction» dynamique entre les deux représentations.)
- Plusieurs systèmes de mémorisation sont utilisées pour accélérer la recherche dynamique de méthodes. Dans 90% des cas, un appel de méthodes requiert seulement une opération de comparaison en plus du traitement normal d'un appel de procédure.

L'article original de Deutsch et Schiffman parlait également d'une meilleure gestion des compteurs de références pour la gestion automatique de la mémoire, mais ceci est complètement obsolète maintenant à mon avis. Il est à noter que cette approche se base aussi sur l'idée suivante : la plupart du temps, l'exécution de Smalltalk se comporte comme celle d'un langage classique. C'est à dire qu'on envoie rarement des messages aux blocs d'activation. C'est dans ce sens-là qu'il est intéressant de faire une implantation selon les critères d'efficacité les plus rigoureux quitte à payer un fort coût lorsqu'un programme envoie un message à un bloc d'activation, car cela arrive si peu souvent qu'on sera à terme gagnant.

Voyons maintenant chacun des points dans l'ordre.

### Traduction dynamique du code v-machine

Cette traduction ne posait pas de problème théorique sérieux. Il s'agit, comme dans tout compilateur, de prendre du code virtuel et de le transformer en code réel d'une machine physique sous-jacente. Les implanteurs de compilateurs savent faire cela. Les problèmes que cela pose concernent plus les optimisations à faire étant donné le jeu d'instruction de la machine physique et leurs propriétés. Évidemmment, comme ce traducteur est appelé souvent pendant l'exécution, l'objectif premier est que la traduction soit la plus rapide possible.

### Traduction dynamique des états machine en états v-machine

Par état de machine, on veut dire toute l'information système nécessaire pour gérer l'exécution de la v-machine. Comme on n'exécute pas le code de la v-machine mais plutôt sa traduction en code réel, on doit permettre de regarder le code et les états de calcul de la machine physique sous sa forme v-machine. C'est très important car c'est la forme que présuppose le dévermineur Smalltalk (lui-même écrit en Smalltalk)[4].

On doit garder le code v-machine pour y référer. On doit aussi trouver une bonne correspondance pour toutes les informations de la machine comme le pc, l'adresse de retour, etc. Pour le pointeur d'instruction courante (*program counter, pc*). Ceci est fort difficile si on peut interrompre le programme n'importe où. En effet, il est possible qu'une instruction v-machine se traduise en plusieurs instructions de la machine physique. Où doit-on pointer si on s'interrompt en plein milieu de cette séquence d'instructions ? En fait, on fait des concessions. On accepte que l'observation des blocs d'activation ne peut se faire qu'à un sous-ensemble de points de programmes où on le considère interruptible et on précalcule les correspondances du code v-machine en code machine à l'avance.

### Représentations multiples pour les blocs d'activation et les fermetures

Les blocs d'activation de la machine réelle ne sont pas représentés de la même façon que les blocs d'activation de la v-machine. Les processeurs physiques imposent des contraintes sur la forme de leur bloc d'activation qui font qu'on ne peut pas imposer la forme des blocs de la

[4]En tous cas, c'était sûrement vrai à l'origine. Il n'est pas sûr qu'aujourd'hui, le dévermineur n'utilise pas directement les structures de données optimisées.

v-machine. Pour accéder réellement les blocs, il faudra donc traduire les blocs physiques en blocs réels.

En pratique, les blocs d'activation sont créés comme des blocs physiques (efficace et compact). Ce n'est que lorsqu'un bloc d'activation est accédé comme une entité de plein droit qu'il sera transformé en objet. Les blocs physiques sont dits volatiles. Lorsqu'un bloc est créé comme un objet, on lui donne une représentation conforme à la spécification de la machine virtuelle. Ces blocs «objets» sont dits stables (dans le sens où il ne disparaissent pas avec la fin de l'activation de la méthode, mais plutôt lorsque le programme ne détient plus de références sur eux). Enfin, une représentation hybride est utilisée lorsqu'un bloc d'activation est créé par l'activation d'une méthode mais qu'il peut être accédé comme un objet (on a un pointeur sur lui), il est représenté sous une forme hybride, c'est à dire une représentation traditionnelle avec un entête (*header*) contenant l'information nécessaire pour le voir comme un objet. Lorsqu'un message lui est envoyé le bloc hybride est converti en bloc stable.

Certaines expériences ont montré que sur des programmes Smalltalk courants, seulement 10% des blocs ont besoin d'être représentés sous une forme autre que volatile, ce qui tend à justifier l'approche de Deutsch-Schiffman qui est d'optimiser tout le temps, quitte à payer le prix fort si on accède aux objets faisant partie de la sep'cification de la v-machine. Si ces accès n'arrive que dans 10% des cas, l'optimisation sur les autres 90% va amplement compenser le coût plus élevé dans les premiers 10%.

### Systèmes de mémorisation pour la résolution des envois de messages

L'idée est relativement simple. Lorsqu'on envoie un message à un objet, une recherche est faite pour trouver la méthode applicable à ce message pour l'objet donné. Une fois qu'on a trouvé la méthode, on mémorise ce résultat au cas où il y aurait un autre envoi de message semblable à l'objet. On crée donc une grande table où il y a une entrée par sélecteur dans le système. Si j'envoie le message `foo` à un objet `o1` qui est instance de la classe `A`. Supposons que la recherche de la méthode applicable nous retourne la méthode `m`. On met alors dans la table à l'entrée correspondant à `foo`, la classe `A` et la méthode `m`. S'il y a un nouvel envoi de message `foo`, on compare la classe d'instantiation du receveur `o2` à `A` et si c'est aussi `A`, on applique directement la méthode `m`, sinon on refait la recherche et on remplace l'entrée de la table par les nouvelles classe et méthode trouvées.

Cette technique relativement évidente fonctionne bien, probablement parce que le principe de localité s'applique ici également. En effet, si un objet `o1` vient de recevoir un message `foo`, il est probable qu'il en recevra un autre très rapidement. C'est déjà le cas des appels récursifs après tout. Des expériences ont montré d'ailleurs que cette technique de mémorisation à un élément seulement couvrent 85 à 90% des appels sur des programmes Smalltalk courants.

**TP 5.** Implantation de `block` et `return-from`

À partir de l'interprète produit lors du TP 3, introduisez les formes `block` et `return-from` selon la syntaxe abstraite suivante (alternatives à ajouter à la syntaxe abstraite du langage) :

$$e \ ::= \ \texttt{(block}\ i\ e\texttt{)} \mid \texttt{(return-from}\ i\ e\texttt{)}$$

Rappelez-vous que la forme `block` doit lier son identificateur à la continuation courante qui sera invoquée par la forme `return-from` correspondante qui lui passera le résultt de ses propres formes. L'essentiel de ce TP consiste donc à identifier dans l'interprète ce qui représente la continuation courante du programme interprété pour la capturer et la lier à l'identificateur du `block` pour qu'elle soit récupérable lors du `return-from` correspondant.

**TP 6.** Aspects réflexifs de Smalltalk.

Exploration de la définition en Smalltalk des éléments servant à son exécution.

# Bibliographie


[ASS85] H. Abelson, G.J. Sussman, et J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press and McGraw-Hill, 1985.

[CHO88] W.D. Clinger, A.H. Hartheimer, et E.M. Ost. Implementation Strategies for Continuations. In *Proceedings of the ACM Conference on Lisp and Functional Programming '88*, pages 124–131. ACM Press, June 1988.

[CR91] W. Clinger et J. Rees (édité par) . *Revised$^4$ Report on the Algorithmic Language Scheme*, November 1991.

[FWH92] D.P. Friedman, M. Wand, et C.T. Haynes. *Essentials of Programming Languages.* MIT Press and McGraw-Hill, 1992.

[Lal90] R. Lalemant. *Logique, réduction, résolution.* Masson, 1990.

[Que94] C. Queinnec. *Les Langages Lisp.* InterÉditions, 1994.

[Set96] R. Sethi. *Programming Languages – Concepts and constructs.* Addison-Wesley, 2nd édition, 1996.

[Wat93] D.A. Watt. *Programming Language Processors.* Prentice-Hall, 1993.