Table des matières

CHAPITRE 1

Introduction aux paradigmes des langages de programmation

1.1 Introduction

La programmation est une discipline riche et les langages de programmation pratiques sont généralement assez compliqués. C'est pourquoi les langages de programmation devraient supporter de nombreux paradigmes. Un paradigme de programmation est une approche de la programmation d'un ordinateur basée sur une théorie mathématique ou un ensemble cohérent de principes. Chaque paradigme soutient un ensemble de concepts qui le rend le meilleur pour un certain type de problème.

Les langages courants populaires ne supportent qu'un ou deux paradigmes distincts. C'est regrettable, car les problèmes de programmation distincts nécessitent des concepts de programmation distincts pour être résolus proprement, et ces un ou deux paradigmes ne contiennent souvent pas les bons concepts. Un langage devrait idéalement supporter de nombreux concepts de manière bien pondérée, afin que le programmeur puisse choisir les bons concepts chaque fois qu'il en a besoin sans être encombré par les autres.

Dans ce chapitre, nous introduisons le cours avec un aperçu de différents paradigmes de programmation et un bref historique des langages de programmation en donnant beaucoup plus d'informations sur de nombreux paradigmes et concepts.

1.1 Paradigmes, langage et concepts

Le *paradigme* peut être appelé comme méthode pour résoudre un problème ou faire une tâche. Le paradigme de programmation est une approche visant à résoudre un problème en utilisant un langage de programmation ou on peut aussi dire qu'il s'agit d'une méthode pour résoudre un problème en utilisant des outils et des techniques qui sont à notre disposition suivant une certaine approche. Il y a beaucoup moins de paradigmes de programmation que de langages de programmation. C'est pourquoi il est intéressant de se concentrer sur les paradigmes plutôt que sur les langages.

Un langage de programmation est un formalisme artificiel dans lequel les algorithmes peuvent s'exprimer. Malgré son caractère artificiel, ce formalisme reste un *langage*. Son étude peut faire bon usage des nombreux concepts et outils développés au siècle dernier en linguistique (qui étudie à la fois les langues naturelles et artificielles).

La figure 1.1 montre le chemin des langages vers les paradigmes et les concepts. Chaque langage de programmation réalise un ou plusieurs paradigmes. Chaque paradigme est défini par un ensemble de concepts de programmation, organisés en un simple langage de base appelé *langage* de noyau du paradigme. Il existe un très grand nombre de langages de programmation, mais beaucoup moins de paradigmes. Mais il y a encore beaucoup de paradigmes.

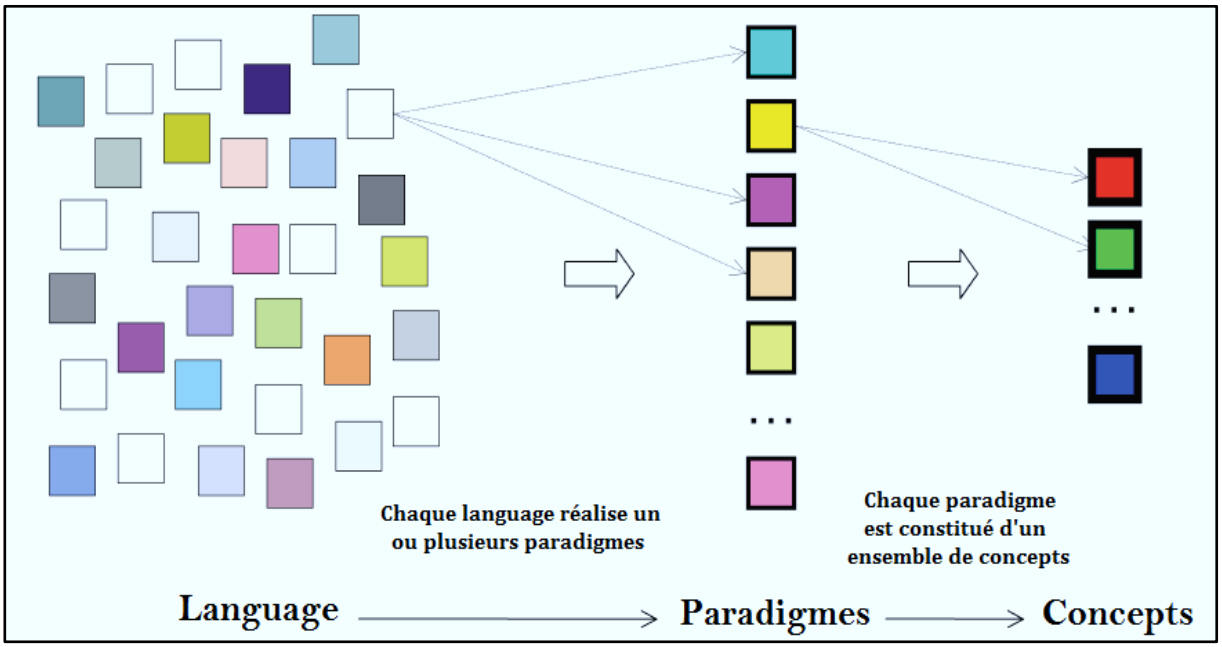

Figure 1.1 *: Langages, paradigmes et concepts*

1.2 Pourquoi étudier les langages de programmation ?

Les langages de programmation sont au cœur de l'informatique, les étudiants qui se sont familiarisés avec une ou plusieurs langages de haut niveau sont généralement curieux d'apprendre d'autres langages, et de savoir ce qui se passe "sous le capot". L'apprentissage des langages est intéressant. C'est aussi pratique.

D'une part, une bonne compréhension de la conception et de l'implémentation d'un langage peut aider à choisir le langage le plus approprié pour une tâche donnée. La plupart des langages sont meilleures pour certaines choses que pour d'autres. Peu de programmeurs sont susceptibles de choisir Fortran pour le calcul symbolique ou le traitement des chaînes de caractères, mais les autres choix ne sont pas aussi évidents.

Fortran ou C pour les calculs scientifiques. PHP ou Ruby pour une application web . ADA ou C pour les systèmes embarqués. Visual Basic ou Java pour une interface utilisateur graphique. De nombreux langages sont étroitement liés.

1.3 Évolution historique des langages de programmation

Les langages de programmation actuels sont le produit de développements qui ont commencé dans les années 1950. De nombreux concepts ont été inventés, testés et améliorés en étant incorporés dans les langages de programmation successifs. À quelques exceptions près, la conception de chaque langage de programmation a été fortement influencé par l'expérience des langages antérieurs. Le bref aperçu historique suivant résume l'ascendance des principaux langages de programmation.

FORTRAN est le premier véritable langage impératif de haut niveau, développé par le groupe de John Backus en 1957 et conçu pour des applications de type numérique-scientifique. À une époque où la programmation se faisait uniquement en langage assembleur et où la plus grande préoccupation était l'efficacité des programmes, la conception d'un langage de haut niveau ne pouvait pas ignorer les performances du code compilé. La conception de FORTRAN a également fait passer la performance en premier et, par conséquent, les caractéristiques d'une machine de référence physique spécifique (l'IBM 704) ont été prises en compte dans la conception. Cependant, contrairement aux langages précédents, FORTRAN était déjà un langage de haut niveau au sens moderne du terme dans sa première version. En fait, cette première version contenait de nombreuses constructions qui étaient plus ou moins indépendantes d'une machine spécifique. Il convient notamment de souligner que FORTRAN a été le premier langage de programmation à permettre l'utilisation directe d'une expression arithmétique symbolique.

ALGOL signifie, plus qu'une language, une famille de languages impératifs introduite à la fin des années 1950. Ces langages, même s'ils ne sont pas devenus de véritables succès commerciaux, ont été prédominants dans le monde universitaire dans les années 60 et 70 et ont eu un impact formidable sur la conception de tous les langages de programmation successifs. Nombre des concepts et constructions que l'on trouve dans les languages modernes ont été introduits, ou expérimentés, pour la première fois dans les language de la famille ALGOL. Ce dernier, a été conçu comme un langage universel, adapté à l'expression des algorithmes en général, plutôt qu'à l'utilisation dans des types d'application spécifiques.

LISP (LISt Processor) a été conçu en 1960 par un groupe dirigé par John Mc- Carthy au MIT (Massachusetts Institute of Technology) et a été l'un des premiers langages conçus spécialement pour les applications non numériques. C'est un langage conçu pour manipuler des ex-pressions symboliques (appelées s-expressions) qui sont

essentiellement des listes et qui sont typiquement utilisées dans l'intelligence artificielle. Parmi les applications du LISP, il y a eu les premières tentatives de l'implémentation de programmes de traduction automatique de textes.

COBOL comme LISP, ce langage a été conçue dans les années 60 et son nom est aussi un acronyme : COBOL signifie COmmon Business Oriented Language.

Les programmes COBOL sont composés de 4 "divisions". Dans la "division de procédure", on écrit le code pour les aspects algorithmiques du programme. Dans la "division des données", les descriptions des données sont écrites. La "division environnement" contient la spécification de l'environnement externe au programme fourni par la machine physique sur laquelle le langage est implémenté. Enfin, la "division identification" contient les informations utilisées pour identifier le programme (nom, auteur, etc.).

Simula, un autre descendant d'ALGOL60. Développé à partir de 1962 au Centre de calcul norvégien par K. Nygaard et O.J. Dahl, Simula est une extension d'ALGOL. Elle a été conçue pour des applications de simulation d'événements discrets, c'est-à-dire pour l'implémentation de programmes qui simulent des situations de charge et de file d'attente afin de pouvoir mesurer des paramètres fondamentaux (temps d'attente moyen, longueur de la file d'attente, etc.).

Smalltalk comme tous les langages de programmation "classiques" des années 1970. Développé dans les années 1970 par Alan Kay au Xerox PARC (Palo Alto Research Center),

Smalltalk présente une nouvelle façon d'intégrer dans un langage de programmation des mécanismes d'encapsulation et de dissimulation d'informations en utilisant les concepts de classe et d'objet (précédemment introduits par Simula) et des règles de visibilité précises pour les classes (les méthodes sont publiques, les variables d'instance sont privées).

C est parmi les nouveaux langages des années 1970, les plus importants, est un langage conçu par Dennis Ritchie et Ken Thompson au laboratoire AT&T Bell. Initialement conçu comme un langage de programmation système pour le système d'exploitation Unix, le C est rapidement devenu un langage à usage général, même si la programmation système reste un de ses domaines d'application les plus importants et notons que le nom vient du fait que le langage en 1972 a été le successeur d'un langage appelé B, qui était à son tour une version réduite du langage système BCPL.

Pascal a été développé vers 1970 par Niklaus Wirth comme développement et simplification d'ALGOLW et a été le langage éducatif le plus utilisé jusqu'à la fin des années 1980. Le nom est en l'honneur du mathématicien (ainsi que du physicien, philosophe et écrivain) Blaise Pascal qui, pour aider un père surchargé par un travail difficile dans le cadre de l'administration de la Normandie, a conçu en 1642 une "machine arithmétique", précurseur des machines de calcul mécaniques.

L'une des principales raisons du succès et de la renommée du Pascal est qu'il a été le premier langage qui, préfigurant Java et ses bytecodes de près de 20 ans, a introduit le concept de code intermédiaire comme instrument de portabilité des programmes.

ML est né sous le nom de Meta Langage (d'où son nom) pour un système semi-automatique destiné à prouver les propriétés des programmes et a été développé par le groupe de Robin Milner à Edimbourg, à partir du milieu des années 1970.

En ML, comme en LISP, un programme est constitué d'un ensemble de définitions de fonctions. Diverses constructions impératives ont été ajoutées à la partie purement fonctionnelle du ML, en particulier l'affectation (qui est limitée aux "cellules de référence" qui peuvent être considérées comme des variables modifiables qui utilisent le modèle de référence).

PROLOG a également été défini dans les années 1970. C'était le premier langage de programmation logique et il est toujours disponible aujourd'hui dans différentes versions et implémentations. Si, comme nous l'avons dit, certaines idées sur la programmation logique remontent aux travaux de Kurt Gödel et Jacques Herbrand, les premières bases théoriques solides ont été publiées par A. Robinson qui, dans les années 60, a apporté une contribution essentielle à la théorie de la déduction automatique. Robinson a en effet fourni une définition formelle de l'algorithme d'unification et a défini la *résolution*, un mécanisme de déduction qui utilise l'unification et permet la preuve des théorèmes en logique du premier ordre.

C++ la première version du C++ a été définie par Bjarne Stroustrup en 1986 aux Laboratoires Bell d'AT&T après plusieurs années de travail (et après la définition de divers autres langages) sur la manière d'ajouter des classes et un héritage au langage C sans nuire à l'efficacité et sans compromettre la compatibilité avec le langage C existant. Le langage C devait rester un sous-ensemble du langage C++ et, en tant que tel, devait être acceptable pour le compilateur C++. À ces objectifs principaux, s'est ajoutée l'amélioration du système de types du C.

ADA est une autre language importante définie dans les années 1980. Le projet de définition d'Ada a été parrainé par le ministère américain de la défense. Le langage Ada a été défini d'une manière un peu inhabituelle, en commençant par un concours entre plusieurs groupes de concepteurs, à la fois universitaires et industriels, pour la conception d'un nouvel formalisme qui satisferait les exigences du ministère de la défense. La pétition a été remportée par Jean Ichbiah en 1979 avec un langage basé sur Pascal qui comprenait de nombreuses nouvelles constructions nécessaires à la programmation de systèmes temps réel et embarqués, ainsi que d'autres types de systèmes. La proposition d'Ichbiah comprenait des types de données abstraites, le concept de tâche, des mécanismes de temporisation et des mécanismes d'exécution simultanée de tâches.

JAVA : Le langage orienté objet Java a été développé par un groupe (l'*équipe Green*) dirigé par Jim Gosling de SUN. Le projet initial, commencé en 1990, avait pour but de définir un langage, basé sur une nouvelle implémentation de C++, à utiliser dans de petits appareils informatiques de puissance relativement limitée, connectés à un réseau et à utiliser en liaison avec une télévision qui devait fournir le périphérique d'entrée/sortie. Ces appareils devaient implémenter une sorte de navigateur à utiliser pour la navigation sur le réseau, mais sans utiliser toute la technologie nécessaire à cette tâche.

La figure 1.2 illustre le développement historique de la majorité des langages de programmation.

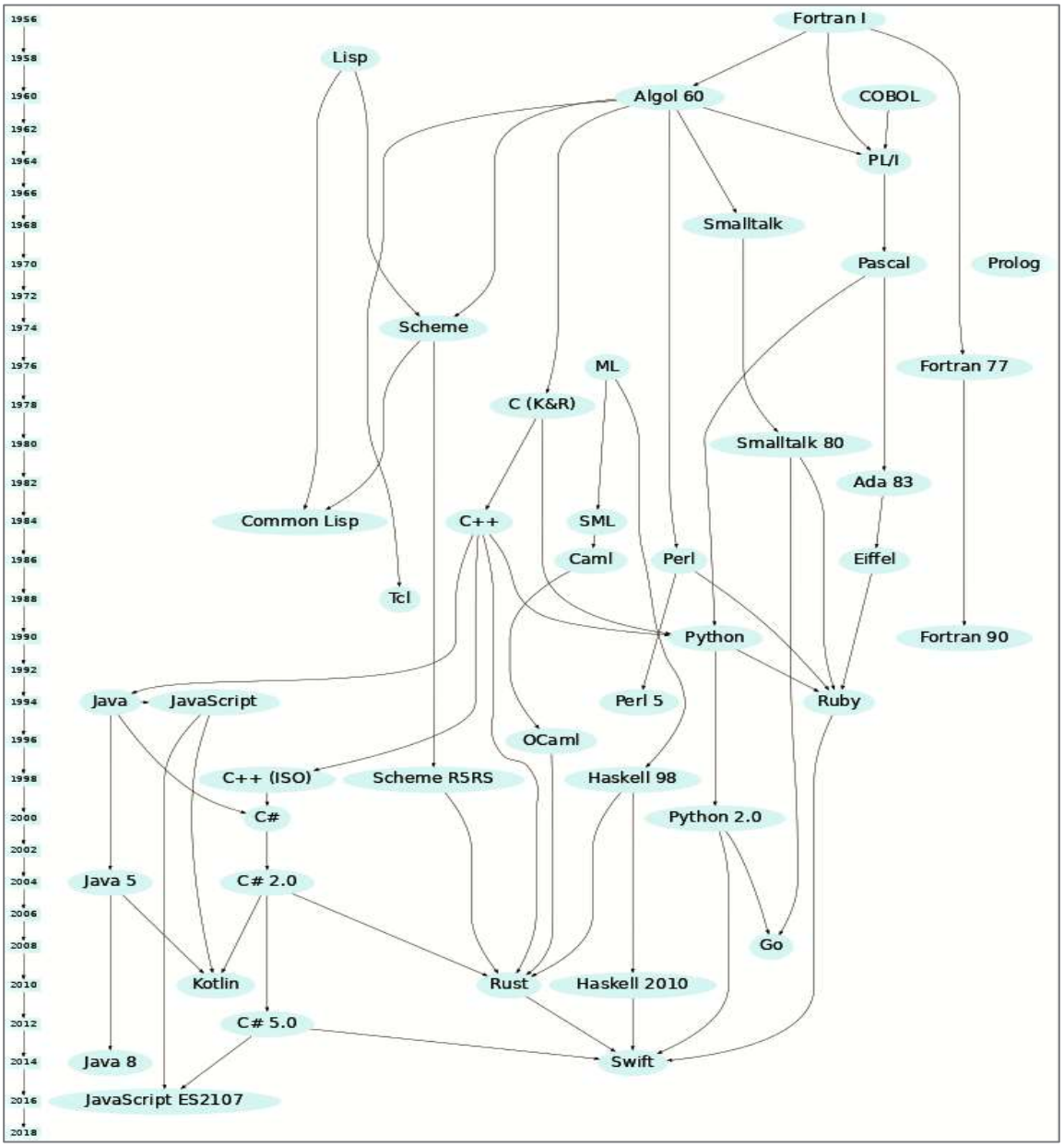

Figure 1.2 : *Historique des langages de programmation*

Presque tous les langages mentionnés jusqu'à présent ont pris en charge **la programmation impérative**, qui se caractérise par l'utilisation de commandes et de procédures qui mettent à jour les variables.

La programmation concurrente est caractérisée par l'utilisation de processus concurrents. Cependant, d'autres paradigmes sont également devenus populaires et importants.

La programmation orientée objet est basée sur des classes d'objets. Un objet a des composantes variables et est doté de certaines opérations. Seules ces opérations peuvent accéder aux composantes variables de l'objet. Une classe est une famille d'objets ayant des composantes variables et des opérations similaires. Les classes s'avèrent être des unités de programme pratiques et réutilisables, et tous les principaux langages orientés objet sont équipés de riches bibliothèques de classes.

Les concepts d'objet et de classe trouvent leur origine dans le SIMULA, un autre langage de type ALGOL. SMALLTALK était le premier langage purement orienté objet, dans lequel des programmes entiers sont construits à partir de classes. Le C++ a été conçu en ajoutant des concepts orientés objet au C. JAVA a été conçue en simplifiant radicalement le C++, en supprimant presque tous ses défauts. Bien qu'il s'agisse principalement d'un simple langage orienté objet.

La programmation fonctionnelle est basée sur des fonctions plutôt que sur des types tels que les listes et les arbres. LISP était le premier langage fonctionnel, qui a démontré à une date remarquablement précoce que les programmes de significant peuvent être écrits sans avoir recours à des variables et à des affectations. HASKELL et ML sont des langages fonctionnels modernes. Ils traitent les fonctions comme des valeurs ordinaires, qui peuvent être passées comme paramètres et renvoyées comme résultats d'autres fonctions.

La programmation logique est basée sur un sous-ensemble de logique de prédicats. Les programmes logiques déduisent des relations entre les valeurs, par opposition au calcul des valeurs de sortie à partir des valeurs d'entrée. PROLOG était le premier langage logique, et reste le plus populaire. Dans sa forme logique pure.

Le langage de script est un langage de programmation de haut niveau qui est interprété plutôt que compilé à l'avance. Un langage de script peut être un langage de programmation général ou se limiter à des fonctions spécifiques utilisées pour améliorer le fonctionnement d'une application ou d'un programme système, tels que JavaScript, Perl, Tcl et Python.

1.4 Taxonomie des paradigmes de programmation

La figure 1.3 présente une taxonomie de tous les principaux paradigmes de programmation, organisés dans un graphique qui montre leurs liens. Cette figure contient beaucoup d'informations et récompense un examen attentif. Il y a 27 cases, chacune représentant un paradigme comme un ensemble de concepts de programmation. Sur ces 27 boîtes, huit contiennent deux paradigmes avec des noms différents mais le même ensemble de concepts. Une flèche entre deux cases représente le ou les concepts qui doivent être ajoutés pour passer d'un paradigme à l'autre. Les concepts sont les éléments primitifs de base utilisés pour construire les paradigmes. Souvent, deux paradigmes qui semblent tout à fait différents (par exemple, la programmation fonctionnelle et la programmation orientée objet …) se distinguent par un seul concept.

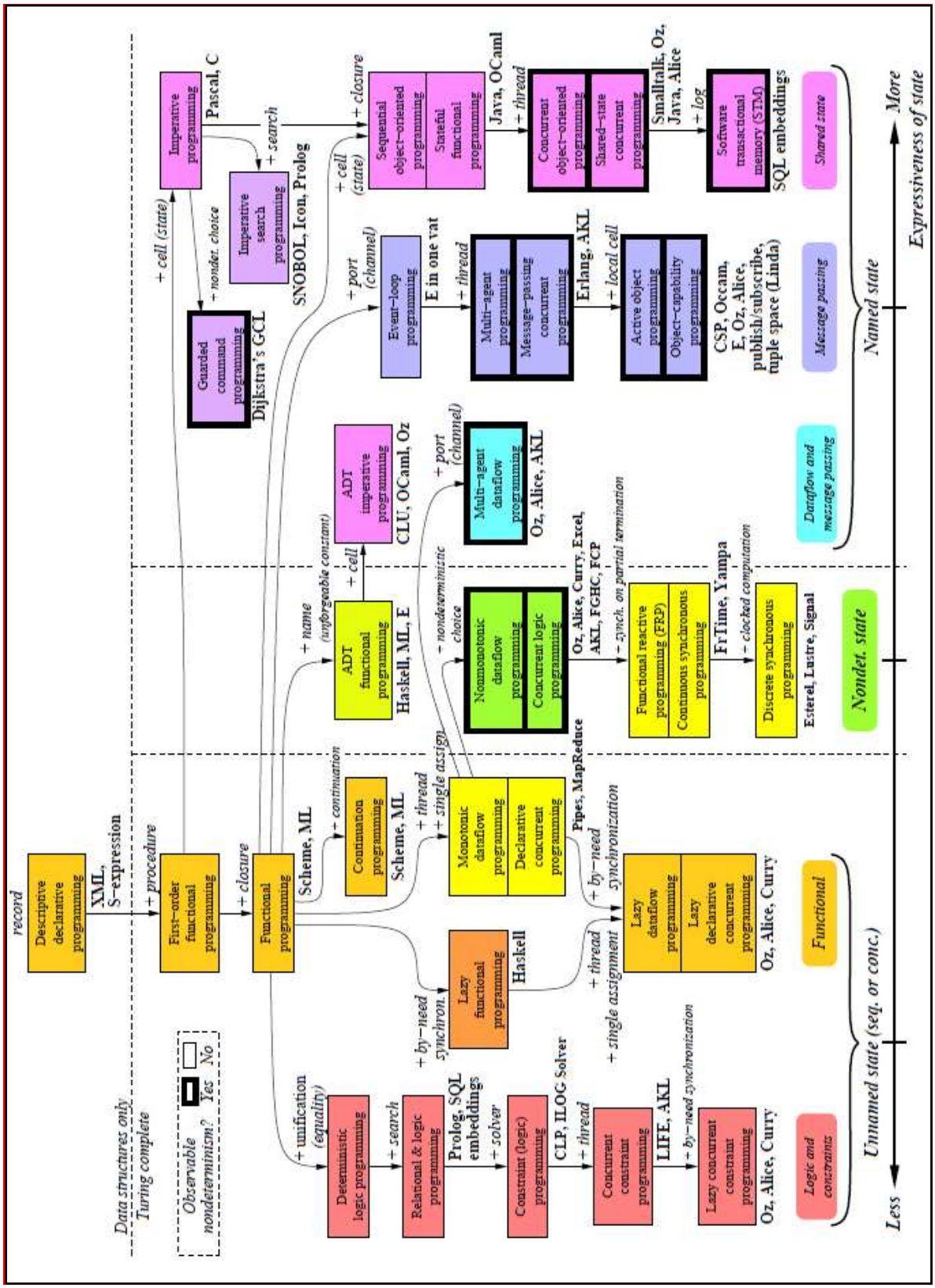

Figure 1.3 : *Taxonomie des paradigmes de programmation*

1.5 Niveaux de description des langages de programmation

Tout langage de programmation a une *grammaire*, une sémantique et une pragmatique. Nous trouvons que les langages naturels ont également une syntaxe et une sémantique, mais la pragmatique est propre aux langages de programmation.

La grammaire est la partie de la description de la langue qui répond à la question : quelles sont les phrases correctes ? Une fois que l'alphabet d'une langue a été défini dans un premier temps (dans le cas d'une langue naturelle, par exemple, l'alphabet latin de 22 ou 26 lettres, l'alphabet cyrillique, etc.), la composante *lexicale,* qui utilise ce phabet, identifie la séquence de symboles constituant les *mots* (ou *jetons*) de la langue définie. Lorsque l'alphabet et les mots ont été définis, la *syntaxe* décrit les séquences de mots qui constituent des expressions légales. La syntaxe est donc une relation entre les signes. Entre toutes les séquences de mots possibles (sur un alphabet donné), la syntaxe choisit un sous-ensemble de séquences pour former des phrases de la langue proprement dite.

La sémantique est la partie de la description du language qui cherche à répondre à la question "que signifie une phrase correcte ? La sémantique attribue donc une *signification* à chaque phrase correcte. Dans le cas des langage naturelles, le processus d'attribution de la signification peut être très complexe ; dans le cas des langage artificielles, la situation est plutôt plus simple. Il n'est pas difficile de supposer, dans ce cas, que la sémantique est une relation entre les signes (phrases correctes) et les significations (entités autonomes existant indépendamment des signes qui sont utilisés pour les décrire). Par exemple, la signification d'un certain programme pourrait être la fonction mathématique calculée par ce programme. La description sémantique de ce langage sera construite à l'aide de techniques permettant de fixer la fonction que le programme calculé.

La pragmatique est la partie d'une description du langage qui se demande "comment utiliser une phrase significative". Les phrases ayant le même sens peuvent être utilisées de différentes manières par différents utilisateurs. Des contextes linguistiques différents peuvent nécessiter l'utilisation de phrases différentes; certaines sont plus élégantes, d'autres plus anciennes ou plus dialectiques que d'autres. La compréhension de ces mécanismes linguistiques n'est pas moins importante que la connaissance de la syntaxe et de la sémantique.

1.6 Conclusion

Dans ce chapitre d'introduction, nous avons vu ce que l'on entend par paradigmes de programmation, et les sujets englobés par ce terme : Paradigmes, langages et concepts. Nous avons brièvement passé en revue l'histoire des langages de programmation. Nous avons également vu comment les principaux paradigmes ont évolué : la programmation impérative, la programmation orientée objet, la programmation concurrente, la programmation fonctionnelle, la programmation logique et les scripts.

CHAPITRE 2

Concepts de base

2.1 Introduction

Ce chapitre explique les concepts de base qui sous-tendent presque tous les langages de programmation : valeurs et types, variables et stockage, liaison et portée, procédures et paramètres. L'accent est mis dans ce chapitre sur l'identification des concepts de base et leur étude individuelle. Ces concepts de base se retrouvent dans presque tous les langages.

2.2 Valeurs et types

Une *valeur* est toute entité qui peut être manipulée par un programme. Les valeurs peuvent être évaluées, stockées, transmises sous forme d'arguments, renvoyées sous forme de résultats de fonctions, etc.

Les différents langages de programmation supportent différents types de valeurs : C prend en charge les entiers, les nombres réels, les structures, les tableaux, les unions, les pointeurs vers les variables et les pointeurs vers les fonctions. (Les nombres entiers, les nombres réels et les pointeurs sont des *valeurs primitives* ; les structures, les tableaux et les unions sont des *valeurs composites*).

JAVA supporte les booléens, les entiers, les nombres réels, les tableaux et les objets. (Les booléens, les entiers et les nombres réels sont des valeurs primitives ; les tableaux et les objets sont des valeurs composites).

ADA prend en charge les booléens, les caractères, les énumérateurs, les entiers, les nombres réels, les enregistrements, les tableaux, les enregistrements discriminés, les objets (enregistrements marqués), les chaînes de caractères, les pointeurs vers des données et les pointeurs vers des procédures. (Les booléens, les caractères, les énumérants, les entiers, les nombres réels et les pointeurs sont des valeurs primitives ; les enregistrements, les tableaux, les enregistrements discriminés, les objets et les chaînes de caractères sont des valeurs composites).

2.2.1 Types scalaires

Les types scalaires (ou simples) sont les types dont les valeurs ne sont pas composées d'agrégats d'autres valeurs.

2.2.1.1 Booleans

Le type de valeurs logiques, ou booléens, est composé de :
Les valeurs : Les deux valeurs de vérité, le *vrai* et le *faux*.
Opérations : une sélection appropriée parmi les principales opérations logiques : conjonction (et), disjonction (ou) et négation (non), égalité, exclusive ou, etc.

S'il est présent (C par exemple n'a pas de type construit de cette manière), ses valeurs peuvent être stockées, exprimées et indiquées. Pour des raisons d'adressage, la représentation de la mémoire ne consiste pas en un seul bit mais en un octet (ou éventuellement plus si un alignement est nécessaire).

2.2.1.2 Caractères

Le type de caractère est composé de:

- *Valeurs* : un ensemble de codes de caractères, fixés lors de la définition du language ; les plus courants sont l'ASCII et UNICODE.
- *Fonctionnement* : dépend fortement du language; on trouve toujours l'égalité, la comparaison et un moyen de passer d'un personnage à son successeur (selon le codage fixe) et/ou à son prédécesseur.

Les valeurs peuvent être stockées, exprimées et indiquées. La représentation stockée sera constituée d'un seul octet (ASCII) ou de deux octets (UNICODE).

2.2.1.3 Entiers

Le type de nombres entiers est composé de

- *Les valeurs* : Un sous-ensemble fini d'entiers, généralement fixé lors de la définition du langage (mais dans certains cas, il est déterminé par la machine abstraite, ce qui peut être la cause de certains problèmes de portabilité). En raison de problèmes de représentation, l'intervalle [-2t , 2t - 1] est couramment utilisé.
- *Opérations* : les comparaisons et une sélection appropriée des principaux opérateurs arithmétiques (multiplication "addition, soustraction", division entière, reste après division, exponentiation, etc.)

Les valeurs peuvent être stockées, exprimées et indiquées. La représentation en mémoire consiste en un nombre pair d'octets (généralement 2, 4 ou 8), sous la forme d'un complément à deux. (Certaines languess prennent en charge les nombres entiers de longueur arbitraire).

EXEMPLE : Types d'entiers en JAVA et C

Considérez les déclarations JAVA suivantes :

```
int countryPop ;

long worldPop ;
```

La variable countryPop pourrait être utilisée pour contenir la population actuelle de n'importe quel pays (puisqu'aucun pays n'a encore une population supérieure à 2 milliards d'habitants). La variable worldPop pourrait être utilisée pour contenir la population totale du monde. Mais notez que le programme échouerait si le type de worldPop était **int** plutôt que **long** (puisque la population totale du monde dépasse maintenant les 7 milliards).

2.2.1.4 Réels

Le type dit réel (ou nombres à virgule flottante) est composé de :

- *Valeurs :* un sous-ensemble approprié des nombres rationnels, généralement fixé lors de la définition du langage (mais il y a des cas où il est fixé par la machine abstraite spécifique, ce qui affecte profondément la portabilité) ; la structure (taille, granularité, etc.) d'un tel sous-ensemble dépend de la représentation adoptée.
- *Opérations* : comparaisons et sélection appropriée des principales opérations numériques (addition, soustraction, multiplication, division, exponentiation, racines carrées, etc.)

Les valeurs peuvent être stockées, exprimées et indiquées. La représentation de la mémoire comprend quatre, huit et dix octets, en virgule flottante, comme le prévoit la norme IEEE 754 (pour les langages et les architectures à partir de 1985).

2.2.1.5 Complexe

Le type dit complexe est composé de :

- *Valeurs* : un sous-ensemble approprié des nombres complexes, généralement fixé par la définition du langage; la structure (taille, granularité, etc.) de ce sous-ensemble dépend de la représentation adoptée.
- *Opérations* : comparaisons et sélection appropriée des principales opérations numériques (somme, soustraction, multiplication, division, exponentiation, prise de racines carrées, etc.)

Les valeurs peuvent être stockées, exprimées et indiquées. La représentation consiste en une paire de valeurs à virgule flottante.

2.2.1.6 Enumérations

En plus des types prédéfinis, tels que ceux introduits ci-dessus, on trouve également dans certains langages différentes façons de définir de nouveaux types. Les énumérations et les intervalles sont des types scalaires qui sont définis par l'utilisateur. Un type d'énumération consiste en un ensemble fixe de constantes, chacune étant caractérisée par son propre nom.

2.2.1.7 Intervalles

Les valeurs d'un type d'intervalle forment un sous-ensemble contigu des valeurs d'un autre type scalaire (le *type de base de l*'intervalle). Voici deux exemples en Pascal (qui a été le premier langage à introduire les intervalles ainsi que les énumérations) :

```
type Bingo = 1..90 ;

SomeDwarves = Grincheux...Dormeur ;
```

Dans le premier cas, le type Bingo est un intervalle de 90 éléments dont le type de base est le type entier. L'intervalle SomeDwarves est formé à partir des valeurs Grumpy, Happy et Sleepy et a Dwarf comme type de base..

2.2.1.8 Types discrets

Les types booléen, caractère, entier, énumération et intervalle sont des exemples de *types* types *discrets*. Ils sont dotés d'un concept d'ordre total bien défini et, surtout, possèdent un concept de prédécesseur et de successeur pour chaque élément, sauf pour les valeurs extrêmes. Les types ordonnés sont un choix naturel pour les indices vectoriels et pour les variables de contrôle d'itérations définies.

2.2.2 Types composites

Les types non scalaires sont dits *composites* car ils sont obtenus en combinant d'autres types à l'aide de *constructeurs* appropriés. Les types composites les plus importants et les plus courants sont

- *Enregistrement* (ou structure), un ensemble de valeurs en général de types différents.
- *Array* (ou vecteur), une collection de valeurs du même type.
- *Ensemble* : sous-ensembles d'un type de base, généralement des types ordinaux.
- *Pointeur* : valeurs l qui permettent d'accéder à des données d'un autre type.
- *Types récursifs* : types définis par récursion, utilisant des constantes et des constructeurs ; les cas particuliers de types récursifs sont les listes, les arbres, etc.

2.2.2.1 Enregistrements

Un enregistrement est une collection formée d'un nombre fini d'éléments (en général ordonnés) appelés *champs,* qui se distinguent par leur nom. Chaque champ peut être d'un type différent des autres (les enregistrements sont une structure de données *hétérogène*). Dans la majorité des langages impératifs, chaque champ se comporte comme une variable du même type.3 La terminologie n'est pas, comme d'habitude, universelle. En Pascal, on parle de records, en C (C++, ADA, etc.), on parle de *structures* (struct) ; en Java, elles n'existent pas car elles sont subsumées par la notion de classe.

EXEMPLE : Structures en C ++

Considérez les definitions C++ suivant:

```
enum Month {jan, feb, mar, apr, may, jun, jul, aug, sep, oct,
 nov, dec} ;
struct Date { Mois m ; octet d ;
} ;
Ce type de structure a l'ensemble des valeurs :
Date = Mois × Octet = {jan, feb,... ., dec}× {0,..., 255}
Ce type de modèle date de manière encore plus grossière que
 son homologue ADA.
Le code suivant illustre la construction d'une structure :
struct Date someday = {jan, 1} ;
Le code suivant illustre le choix de la structure :
printf("%d/%d", someday.m + 1, someday.d) ;  someday.d =29;
someday.m = feb ;
```

2.2.2.2 Tableaux

Un tableau (ou vecteur) est une collection finie d'*éléments de* même type, indexés par un intervalle de type ordinal. Chaque élément se comporte comme s'il était une variable de même type. L'intervalle de type ordinal des indices est le type d'indice, ou le type des indices du tableau. Le type des éléments est le type de *composant,* ou même, avec un peu d'imprécision, le *type de tableau*. Comme tous les éléments sont du même type, les tableaux sont des types de données homogènes.

Commençons par l'un des exemples les plus simples que nous puissions produire en C :

```
int V [10] ;
```

Les crochets après le nom de la variable indiquent que nous avons affaire à un tableau formé de 10 éléments, chacun d'entre eux étant une variable de type entier (dorénavant, nous dirons simplement : un tableau de 10 entiers). Le type d'élément est

donc int, tandis que le type d'index est un intervalle de 10 éléments ; selon la convention adoptée dans de nombreux langages (C, C++, Java, ADA, etc.), cet intervalle commence à 0. Dans ce cas, le type d'index est donc l'intervalle 0 à 9.

Tous les langages permettent de définir des tableaux *multidimensionnels*, c'est-à-dire des tableaux indexés par deux ou plusieurs indices :

EXEMPLE : Tableaux bidimensionnels ADA

Considérez les définitions ADA suivantes:

```
type  Xrange  is range  0  .. 511;
type  Yrange  is range  0  .. 255;
type Window is array (YRange, XRange) of Pixel;
type de tableau bidimensionnel possède l'ensemble de
valeurs suivant :
Window = Yrange × Xrange → Pixel = {0,..., 255}× {0,...,
511}→ Pixel
```

Un tableau de ce type est indexé par une paire d'entiers. Ainsi, w(8,12) accède à la composante de w dont l'indice est la paire (8, 12).

2.2.2.3 Fonctions

Chaque langage de programmation de haut niveau permet de définir des fonctions (ou des procédures). Peu de langages traditionnels permettent la dénotation des types de fonctions (c'est-à-dire leur donner un nom dans le langage). Soit f une fonction définie comme :

```
T f(S s){. }
```

Il est de type S -> T ; plus généralement, une fonction avec en-tête

```
T f(S1 s1, ..., Sn sn){...}.
est de type S1 ×- - - ×Sn->T.
```

Les valeurs d'un type fonctionnel sont donc dénotables dans tous les langages mais sont rarement exprimables (ou mémorisables). Outre la définition, la principale opération autorisée sur une valeur de type fonctionnel est l'*application,* c'est-à-dire l'invocation d'une fonction sur certains arguments (paramètres réels).

2.2.2.4 Les ensembles (Sets)

Certains langages de programmation (en particulier Pascal et ses descendants) permettent de définir des ensembles de types, dont les valeurs sont composées de sous-ensembles d'un type de base (ou univers), généralement limité à un type ordinal. Nous nous sommes basés sur la syntaxe du Pascal, mais en l'adaptant quelque peu :

```
set of char S;
set of Dwarf N;
```

Nous avons une variable S qui est un sous-ensemble des caractères et une variable N qui contiendra un sous-ensemble du type Dwarf. Les langages qui attribuent des ensembles fournissent une syntaxe appropriée pour attribuer un sous-ensemble spécifique à une variable, par exemple :

```
N = (Grincheux, somnolent) ;
```

Les opérations possibles sur les valeurs d'un ensemble de type set sont le test d'appartenance (le test qu'un élément appartient à un ensemble) et les opérations habituelles de la théorie des ensembles de l'union, de l'intersection et de la différence ; le complément n'est pas toujours fourni.

2.2.3 Types récursifs

Un type récursif est un type composite dans lequel une valeur du type peut contenir une (référence à une) valeur du même type. Dans de nombreux langages, les types récursifs peuvent être assimilés (et le sont en fait) à des enregistrements dans lesquels un champ est du même type que l'enregistrement lui-même.

2.2.3.1 Listes

Une *liste* est une séquence de valeurs. Une liste peut avoir un nombre quelconque de composants, y compris aucun. Le nombre de composants est appelé la *longueur de la* liste. La liste unique sans composants est appelée *liste vide*.

EXEMPLE : Listes en JAVA

Les déclarations de classe JAVA (unidiomatiques) suivantes définissent les listes d'entiers :

```
class IntList {
public IntNode first;
public IntList (IntNode first) {
this.first = first;
}
}
class IntNode {
public int elem;
public IntNode succ;
public IntNode (int elem, IntNode succ){
this.elem = elem; this.succ = succ;
}
}
```

La classe IntNode est défini en soi. Ainsi, chaque objet IntNode contient un composant IntNode nommé succ. Cela pourrait sembler impliquer qu'un objet IntNode contient un objet IntNode, qui à son tour contient un autre objet IntNode, et ainsi de suite pour toujours ; mais tôt ou tard, l'un de ces objets aura son composant succ réglé sur **nul**.

2.2.3.2 Chaîne

Une *chaîne* est une séquence de caractères. Une chaîne peut comporter un nombre quelconque de caractères, y compris aucun. Le nombre de caractères est appelé la *longueur de la* chaîne. La chaîne unique sans caractères est appelée *chaîne vide.*

Dans un langage *de* programmation qui supporte les listes, l'approche la plus naturelle consiste à traiter les chaînes de *caractères* comme des *listes de caractères*. Cette approche rend toutes les opérations de liste habituelles automatiquement applicables aux chaînes de caractères.

2.2.4 Expressions

Une expression est une entité syntaxique dont l'évaluation soit produite une valeur, soit ne se termine pas, auquel cas l'expression est non amendée.

La caractéristique essentielle d'une expression, celle qui la différencie d'une commande, est donc que son évaluation produit une valeur. Les exemples d'expressions numériques sont familiers à tous : par exemple, 4+3*2 est une expression dont l'évaluation est évidente. De plus, on peut voir que, même dans un cas aussi simple, pour obtenir le résultat correct, nous avons fait une hypothèse implicite (dérivée de la convention mathématique) sur la précédence des opérateurs. Cette hypothèse, qui nous dit que * a la priorité sur + (et que, par conséquent, le résultat de l'évaluation est 10 et non 14), spécifie un aspect de contrôle pour l'évaluation des expressions. Nous verrons ci-dessous d'autres aspects plus subtils qui peuvent contribuer à modifier le résultat de l'évaluation d'une expression.

2.2.4.1 Expressions conditionnelles

Une *expression conditionnelle* calcule une valeur qui dépend d'une condition. Elle comporte deux ou plusieurs sous-expressions, parmi lesquelles une seule est choisie pour être évaluée.

Le choix est bien sûr fondamental en matière de calcul. Tous les langages de programmation fournissent des expressions conditionnelles, ou des commandes conditionnelles, ou les deux, et les concepts sous-jacents sont similaires. JAVA, C et C++ et fournissent des ***expressions if***. HASKELL fournit à la fois des expressions if et des ***expressions case***.

2.2.4.2 Expressions itératives

Une *expression itérative* est une *expression* qui effectue un calcul sur une série de valeurs (généralement les composants d'un tableau ou d'une liste), donnant un résultat.

EXEMPLE : Compréhensions de listes en HASKELL

Étant donné une liste de caractères cs, la compréhension de la liste HASKELL suivante convertit toutes les lettres minuscules en majuscules, ce qui donne une liste de caractères modifiée :

```
[si c'est la minuscule c, alors c majuscule c
| c <- cs]
```

Le *générateur* "c <- cs" lie c à chaque composante de s à tour de rôle. Si la valeur de s est ['C', 'a', 'r', 'o', 'l'], cette liste de compréhension donnera ['C', 'A', 'R', 'O', 'L'].

2.2.4.3 Littéraux

Le type d'expression le plus simple est un *littéral*, qui désigne une valeur fixée d'un certain type.

EXEMPLE : Littéraux

Voici quelques exemples typiques de littéraux dans les langages de programmation:

```
365    3.1416   faux   '%'   "Quoi ?"
```

Ceux-ci désignent respectivement un entier, un nombre réel, un booléen, un caractère et une chaîne de caractères.

2.2.4.4 Constructions

Une *construction* est une expression qui construit une valeur composite à partir des valeurs qui la composent. Dans certains langages, les valeurs des composants doivent être littérales ; dans d'autres, les valeurs des composants sont calculées en évaluant des sous-expressions.

EXEMPLE : Construction de tableaux en ADA

Dans le code ADA suivant :

```
size: array (Month) of Integer :=(31,
 28,31, 30, 31, 30, 31, 31, 30, 31,
30, 31);...
if is_leap(this_year) then
size(feb) := 29;
end if;
```

Une construction de tableau est utilisée pour initialiser la taille de la variable. Dans les constructions ADA, les valeurs des composants peuvent être calculées, mais dans cet exemple, elles sont littérales. (Il serait bien d'utiliser une expression conditionnelle pour calculer la composante feb, au lieu du littéral 28, mais ADA ne fournit pas d'expressions conditionnelles).

2.2.4.5 Appels de fonction

Un *appel de fonction* calcule un résultat en appliquant une procédure (ou méthode) de fonction à un ou plusieurs arguments. L'appel de fonction a généralement la forme "*F(E)*", où *F* détermine la procédure de fonction à appliquer, et l'expression *E* est évaluée pour déterminer l'argument.

Dans la plupart des langages de programmation, *F n*'est que l'identifiant d'une fonction spécifique. Cependant, dans les langages qui traitent les fonctions comme des valeurs de première classe, *F* peut être toute expression produisant une fonction. Par exemple, l'appel de fonction HASKELL :

```
(if .. . then sin else cos)(x)
```

applique soit la fonction sinusoïdale, soit la fonction cosinusoïdale à la valeur de x.

2.3 Variables et stockage

Une *variable* est un conteneur pour une valeur, et peut être inspectée et mise à jour aussi souvent que souhaité. Les variables sont utilisées pour modéliser des objets du monde réel dont l'état change au fil du temps, comme la date du jour, le temps actuel, la population mondiale ou les performances économiques d'un pays.

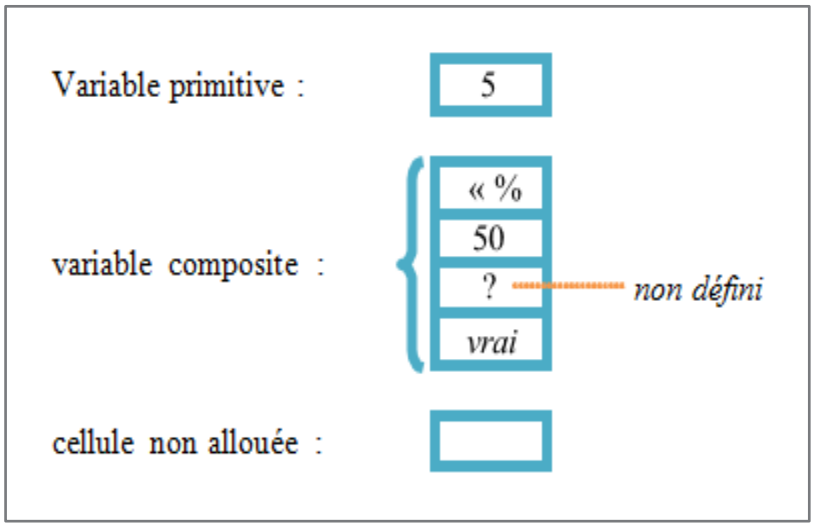

Figure 2.1 : *Un modèle de stockage abstrait.*

- ✓ Un stock est un ensemble de *cellules de stockage*, chacune d'entre elles ayant une *adresse* unique. Chaque cellule de stockage a un *statut* actuel, qui est soit *alloué* soit *non alloué*. Chaque cellule de stockage allouée a un *contenu* actuel, qui est soit une *valeur stockable*, soit indéfinie.

Nous pouvons imaginer chaque cellule de stockage comme une boîte, comme le montre la figure 2.1.

Dans le cadre de ce modèle de stockage, nous pouvons considérer une variable comme un conteneur constitué d'une ou plusieurs cellules de stockage. Plus précisément :

- ✓ Une *simple variable* occupe une seule cellule de stockage allouée.
- ✓ Une *variable composite* occupe un groupe de cellules de stockage contiguës allouées.

2.3.1 Les variables simples

Une *variable simple* est une variable qui peut contenir une valeur stockable. Chaque variable simple occupe une seule cellule de stockage. L'utilisation de variables simples est un aspect tellement élémentaire de la programmation que nous avons tendance à les considérer comme allant de soi. Cependant, il est utile de consulter reflect pour savoir exactement ce qui se passe lorsque nous déclarons, inspectons et mettons à jour une variable simple.

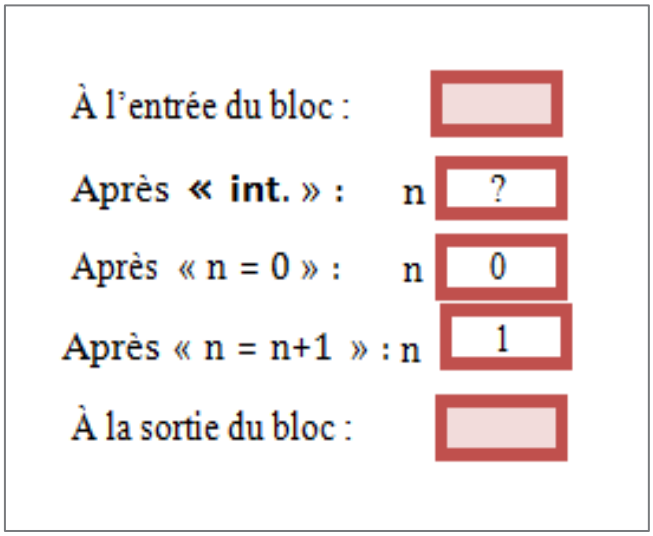

Figure 2.2 : *Stockage pour une variable simple.*

EXEMPLE : Variable simple en C

Considérons la variable simple déclarée dans le bloc C suivant :

```
{ int n;
n = 0;
n = n+1;
}
```

1. La déclaration de variable "**int** n **;** " change le statut d'une cellule de stockage non allouée en allouée, mais laisse son contenu undefined. Dans tout le bloc, n désigne cette cellule.
2. L'affectation "n = 0" fait passer le contenu de cette cellule à zéro.
3. L'expression "n+1" prend le contenu de cette cellule et en ajoute un. L'affectation "n = n+1" (ou "n++") ajoute un au contenu de cette cellule.
4. À la fin du bloc, le statut de cette cellule redevient non attribué. La figure 2.2 montre l'état et le contenu de cette cellule à chaque étape.

À proprement parler, nous devrions toujours dire "le contenu de la cellule de stockage désigné par n". Nous préférons généralement dire de manière plus concise "la valeur contenue dans n", ou même "la valeur de n".

2.3.2 Les variables composites

Une *variable composite* est une variable de type composite. Chaque variable composite occupe un groupe de cellules de stockage contiguës.
Une variable de type composite a la même structure qu'une valeur de ce type. Par exemple, une variable d'enregistrement est un tuple de variables composantes ; et une variable de tableau est une correspondance entre une plage d'indices et un groupe de variables composantes. Les variables composantes peuvent être inspectées et mises à jour de manière sélective.

EXEMPLE : Enregistrement en C++ et variables de tableau

Considérez les déclarations C++ (ou C) suivantes :

```
struct Date {
int a, m, j ;
} ;
Date d'aujourd'hui ;y
```

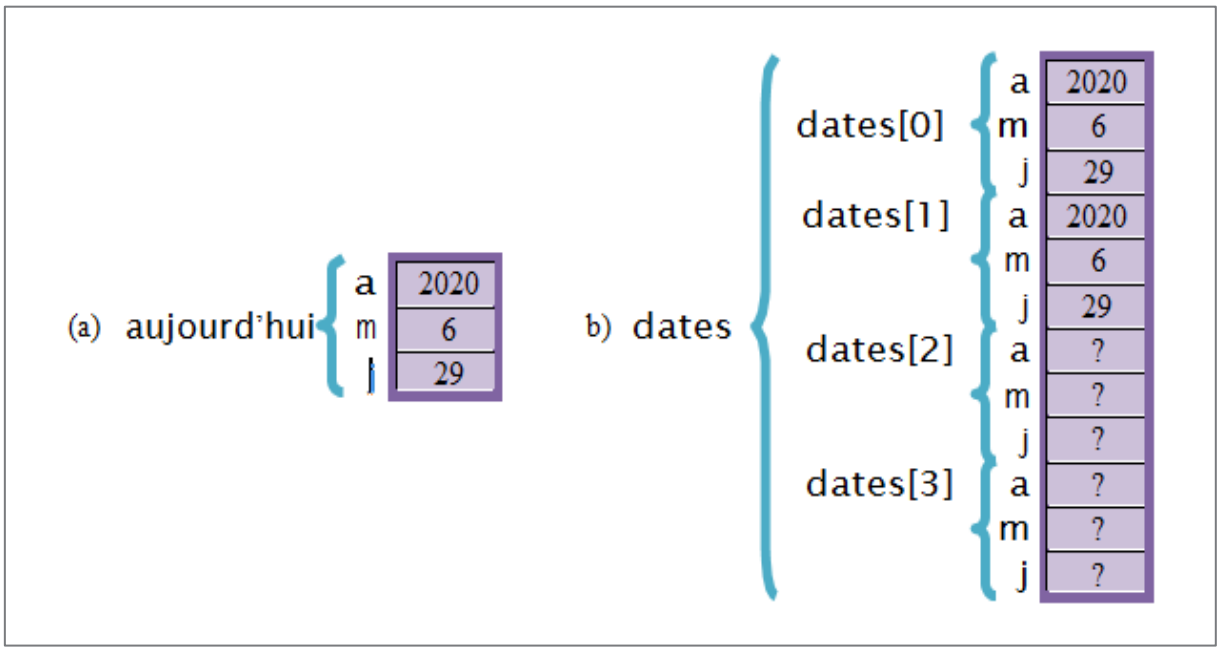

Figure 2.3 : *Stockage pour (a) une variable d'enregistrement, et (b) une variable de tableau*

Chaque valeur de date est un triple composé de trois valeurs **int.** De même, une variable Date est un triple composé de trois variables **int. La** figure 2.3 (a) montre la structure de la variable aujourd'hui, et l'effet des affectations suivantes : aujourd'hui.m = 6;aujourd' hui.j = 26; aujourd'hui. a = 2020 ;

2.3.3 Mise à jour totale ou sélective

Une variable composite peut être mise à jour soit en une seule étape, soit en plusieurs étapes, une composante à la fois. La mise à jour *totale d'*une variable composite consiste à la mettre à jour avec une nouvelle valeur (composite) en une seule étape. La mise à jour *sélective d'une variable composite signifie la mise à jour d'*un seul composant.

EXEMPLE : Mises à jour total et sélectives en C++

Considérez les déclarations suivantes :

```
struct Date {
int a, m, j;
} ;
Date d'aujourd'hui, demain ;
```

La mission suivante :
demain = aujourd'hui ;
copie toute la valeur d'aujourd'hui dans la variable de demain. En d'autres termes, elle copie le contenu des trois cellules de stockage d'aujourd'hui dans les trois cellules de stockage de demain. C'est un exemple de mise à jour totale.
La mission suivante :
demain.j = aujourd'hui.j + 1 ;
met à jour une seule composante de demain, en laissant les autres composantes intactes. C'est un exemple de mise à jour sélective.

2.3.4 Durée de vie (Lifetime)

Le concept de vie est important d'un point de vue pragmatique. Une variable ne doit occuper des cellules de stockage que pendant sa durée de vie. Lorsque la variable est détruite, les cellules de stockage qu'elle occupait peuvent être désaffectées, et peuvent ensuite être affectées à un autre usage. Ainsi, le stockage peut être utilisé de manière économique.
Nous pouvons classer les variables en fonction de leur durée de vie :

La durée de vie d'une *variable globale* est la durée d'exécution du programme.

La durée de vie d'une *variable locale* est une activation d'un bloc.

La durée de vie d'une *variable de tas* est arbitraire, mais elle est limitée par la durée d'exécution du programme.

La durée de vie d'une *variable persistante* est arbitraire, et peut transcender la durée d'exécution d'un programme particulier.

2.3.4.1 Variables globales et locales

Une *variable globale* est une *variable* qui est déclarée pour être utilisée tout au long du programme. La durée de vie d'une variable globale est la durée d'exécution totale du programme : la variable est créée lorsque le programme démarre, et est détruite lorsque le programme s'arrête.

Une *variable locale* est une *variable* qui est déclarée dans un bloc, pour être utilisée uniquement dans ce bloc. La durée de vie d'une variable locale est une activation du bloc contenant la déclaration de cette variable : la variable est créée à l'entrée du bloc, et est détruite à la sortie du bloc.

EXEMPLE : Durée de vie des variables globales et locales en C++

Considérez le programme C++ (ou C) suivant :

```
int g;
void main () {

  int x1;  float  x2;
    .. .  P();  .. .  Q(); .. .
}
void P () {
    float y1; int y2;
    .. . Q(); .. .
}
void Q () {
    int z;
   .. .
}
```

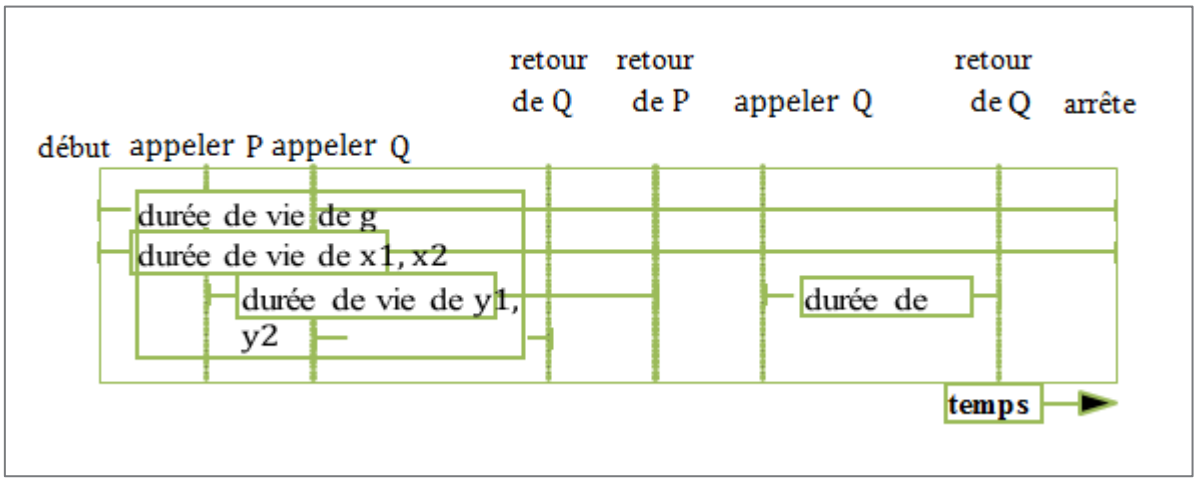

Figure 2.4 : *Durée de vie des variables locales et globales*

Les blocs de ce programme sont les corps de procédures main, P et Q. Il y a une variable globale, g. Il y a plusieurs variables locales : x1 et x2 (en main), y1 et y2 (en P) et z (en Q).

Remarquez que le principal appelle P, qui à son tour appelle Q ; plus tard, le principal appelle directement Q. La figure 2.4 montre les durées de vie de la variable globale et des variables locales.

2.3.4.2 Variables persistantes

Une *variable persistante* est une *variable* dont la durée de vie transcende l'activation d'un programme particulier. En revanche, une *variable transitoire est une variable dont la durée de vie* est limitée par l'activation du programme qui l'a créée. Les variables globales, locales et de tas sont transitoires.

EXEMPLE : Variable persistante en ADA

Considérez les déclarations de type ADA suivantes :

```
type Country is (AT, BE, DE, DK,
 ES, FI,FR,  GR, IE,  IT,  LU,  NL,
 PT,  SE,  UK);
type Statistics is record
population: Integer; area:
Float;. .
end record;
type StatsTable is array (Country) of
 Statistics;
```

Supposons qu'un file séquentiel nommé stats.dat comporte des éléments de type Statistiques, une composante pour chaque valeur du type Pays.
La déclaration suivante instancie le package générique Ada.Sequential_IO

```
papackage Stats_IO est nouveau Ada.Sequential_IO (
Element_Type => Statistiques) ;
```

Le package Stats_IO qui en résulte fournit un type File_Type, dont les valeurs sont séquentielles files avec des composants Statistiques, ainsi que des procédures pour ouvrir, fermer, lire et écrire un tel files.

2.3.5 pointeur

Un *pointeur* est une référence à une variable particulière. En fait, les pointeurs sont parfois appelés des *références*. La variable à laquelle un pointeur se réfère est appelée le *référent du* pointeur.

EXEMPLE : Pointeurs ADA

Dans ADA, chaque valeur de type "**accès** *T*" est soit un pointeur vers une variable de type *T*, soit un pointeur nul.
Considérez les déclarations suivantes :

```
type IntPointer is access IntNode;
type IntNode is record
elem: Integer; succ: IntPointer;
end record; p: IntPointer;
```

Chaque valeur de type "**access** IntNode" est un pointeur vers une variable de type IntNode. Si p est un pointeur (mais pas nul), la variable d'accès "p. **all**" donne la variable d'enregistrement vers laquelle p pointe, et

"p. **all**.elem" (ou plus précisément "p.elem") sélectionne l'élément composant cette variable d'enregistrement.

2.3.6 Commandes

Une *commande* est une construction de programme qui sera *exécutée* afin de mettre à jour les variables. Les commandes sont une caractéristique des langages impératif, orienté objet et concurrent. Les commandes sont souvent appelées "*statement*", mais nous éviterons d'utiliser ce terme dans ce livre car il signifie quelque chose d'entièrement différent en logique (et en anglais).

2.3.6.1 Commandes séquentielles

Comme les commandes mettent à jour les variables, l'ordre dans lequel les commandes sont exécutées est important. Une grande partie de l'effort de programmation dans un langage impératif concerne le flux de contrôle, c'est-à-dire la garantie que les commandes seront exécutées dans un ordre approprié.

Le flux de contrôle séquentiel est le plus courant. Une *commande séquentielle* spécifie que deux (ou plusieurs) commandes doivent être exécutées en séquence. Une commande séquentielle peut être écrite dans le formulaire :

C1 ; C2

ce qui signifie que la commande C1 est exécutée avant la commande C2. Le contrôle séquentiel flow est disponible dans tous les langages impératifs, et il est si familier qu'il n'est pas nécessaire d'en discuter plus avant ici.

2.3.6.2 Commandes conditionnelles

Une *commande conditionnelle* comporte deux ou plusieurs sous-commandes, dont une seule est choisie pour être exécutée.

La forme la plus élémentaire de commande conditionnelle est la commande ***if***, dans laquelle un choix entre deux sous-commandes est basé sur une valeur booléenne. La commande if se trouve dans tous les langages impératifs, et ressemble typiquement à ceci :

si (*E*) C1 **sinon** C2

Si l'expression booléenne *E* donne *vrai*, C1 est choisi ; si elle donne *faux*, C2 est choisi. La commande if est généralement abrégé lorsque le second sous-commande est un saut:

si (*E*) *C* ≡ **si** (*E*) *C*

autre ;

La commande if peut être généralisée pour permettre de choisir entre plusieurs sous- commandes:

si (*E1*) C1

sinon si (*E2*) C2

. . .

sinon si (*En*) *Cn*

sinon *C0*

Ici, les expressions booléennes *E1, E2,..., En* sont évaluées de manière *séquentielle,* et le premier *Ei* qui donne *vrai* entraîne le choix de la sous-commande correspondante *Ci.* Si aucun *Ei* ne donne vrai, on choisit *C0 à la* place.

2.3.6.3 Commandes collatérales

Une *commande collatérale* spécifie que deux (ou plus) commandes peuvent être exécutées dans n'importe quel ordre. Une commande collatérale peut être écrite dans le formulaire :

```
C1 , C2
```

où C1 et C2 doivent être exécutés, mais sans ordre particulier. Dans la commande collatérale suivante :

```
m = 7,  n = n + 1 ;
```

Les variables m et n sont mises à jour indépendamment, et l'ordre d'exécution n'est pas pertinent.

Une commande collatérale peu judicieuse le serait :

```
n = 7,  n = n + 1 ;
```

L'effet net de cette commande collatérale dépend de l'ordre d'exécution. Supposons que n contienne initialement 0.

```
Si "n = 7" est exécuté first, n contiendra finalement 8.
Si "n = 7" est exécuté en dernier, n finira par contenir 7.
Si "n = 7" est exécuté entre l'évaluation de "n + 1" et
l'attribution de sa valeur à n, n finira par contenir 1.
```

2.3.6.4 Commandes itératives

Une *commande itérative* (communément appelée *boucle*) comporte une sous-commande qui est exécutée de manière répétée. Cette dernière sous-commande est appelée le *corps de la boucle*. Chaque exécution du corps de la boucle est appelée une *itération*.

Nous pouvons classer les commandes itératives en fonction du nombre d'itérations fixés:

itération Indéfini: le nombre d'itérations n'est pas fixé à l'avance.

Itération défini : le nombre d'itérations est fixé à l'avance.

L'itération indéfini est généralement fournie par la ***commande en cours***, qui consiste en un corps de boucle *C* et une expression booléenne *E* (la *condition de boucle*) qui contrôle si l'itération doit se poursuivre. La commande en cours d'exécution ressemble généralement à ceci :

```
tand que (E) C
```

La signification de la commande **tand que** peut être défini par l'équivalence suivante:

```
tand que (E) C ≡ si (E) {
C
tand que (E) C
}
```

Ce site définition indique clairement que l'état de la boucle dans une commande en temps réel est testé
avant chaque itération du corps de la boucle.

Notez que cette définition de la commande en cours est récursif. En fait, l'itération n'est qu'une forme spéciale de récursivité.

2.3.6.5 Affectations

La commande d'affectation a généralement la forme "*V* = E; " (ou "*V* := E; " dans ADA). Ici, *E est* une expression qui donne une valeur, et *V* est un accès à une variable qui donne une référence à une variable (c'est-à-dire son adresse de stockage). La variable est mise à jour pour contenir la valeur. (Si la variable est une composante d'une variable composite, l'effet est la mise à jour sélective de cette variable composite).

Des types de missions plus générales sont possibles. Une *mission multiple*, généralement écrite sous la forme "V1 = . . . = Vn = E;", permet d'attribuer la même valeur à plusieurs variables. Par exemple, l'assignation multiple suivante attribue zéro à deux variables, m et n :

```
m = n = 0 ;
```

Certains langages de programmation (notamment JAVA, C et C++) permettent de combiner des opérateurs binaires tels que le " + " avec l'affectation. Par exemple, la commande suivante :

```
n += 1 ;
```

(qui équivaut à " n = n+1 ; ") augmente la valeur de la variable n..

2.4 Liaison et portée

Une *liaison* est une association fixée entre un identifiant et une entité telle qu 'une valeur, une variable ou une procédure. Une déclaration produit un ou plusieurs liants. La déclaration constante dans l'exemple ci-dessous n de la valeur 7. La déclaration de variable dans cet exemple lie n à une variable nouvellement créée.

EXEMPLE : Effets des déclarations en ADA

Considérons l'expression ADA "n+1", qui utilise l'identifiant n.

Si n est déclaré comme suit:

n : **constant** entier:= 7;

puis n désigne l'entier sept, de sorte que l'expression 'n+1' est évaluée en ajoutant un à sept.

• Si n est déclaré comme suit :

n : Nombre entier;

puis n désigne une variable entière, de sorte que l'expression ''n+1'' est évaluée en ajoutant une valeur à la valeur courante de cette variable

• Si n est déclaré comme suit : n : **constante** Entier := 7 ;

alors n désigne l'entier sept, donc l'expression "n+1" est évalué en ajoutant de un à sept.

• Si n est déclaré comme suit : n : Entier ;

alors n désigne une variable entière, donc l'expression "n+1" est évaluée en ajoutant un à la valeur actuelle de cette variable.

La *portée d*'une déclaration est la partie du texte du programme sur laquelle la déclaration est effective. De même, la *portée d*'un engagement est la partie du texte du programme sur laquelle l'engagement s'applique.

Dans certains des premiers langages de programmation, la portée de chaque déclaration était l'ensemble du programme. Dans les langages modernes, la portée de chaque déclaration est influencée par la structure syntaxique du programme, en particulier la disposition des *blocs*.

2.4.1 Structure du bloc

Un *bloc* est une construction de programme qui délimite le champ d'application des déclarations qu'il contient. Chaque langage de programmation a ses propres formes de blocs :
Les blocs d'un programme C sont les commandes de bloc ({ ... }), les corps de fonction, les unités de compilation (source files), et le programme dans son ensemble.

Les blocs d'un programme JAVA sont les commandes de bloc ({ ... }), les corps de méthode, les déclarations de classe, les packages et le programme dans son ensemble.
Les blocs d'un programme ADA sont les commandes de bloc (**declare** ... **begin** ... **end ;**), les corps de procédure, les packages, les tâches, les objets protégés et le programme dans son ensemble.

La figure 2.5 montre les trois types de structure de bloc. Chaque case représente un bloc. Les portées de certaines déclarations sont indiquées par des ombres.

Dans un langage à *structure monolithique par blocs* (figure 2.5-a), le seul bloc est le programme entier, de sorte que la portée de chaque déclaration est le programme entier. En d'autres termes, toutes les déclarations sont globales. C'est ce que l'on trouve dans les anciennes versions du COBOL.
La structure de bloc monolithique est la plus simple possible, mais elle est beaucoup trop grossière, en particulier pour l'écriture de gros programmes. Les programmeurs doivent veiller à ce que toutes les déclarations aient des identifiants distincts. C'est très gênant pour les grands programmes développés par une équipe de programmeurs.

Dans un langage à *structure de blocs flat* (figure 2.5-b), le programme est divisé en plusieurs blocs qui ne se chevauchent pas. Ceci est illustré par FORTRAN, dans lequel les organes de procédure ne peuvent pas se chevaucher, mais chaque organe de procédure agit comme un bloc. Une variable peut être déclarée à l'intérieur d'un corps de procédure particulière, et son champ d'application est ce corps de procédure. La portée de chaque variable globale (et la portée de chaque procédure elle-même) est l'ensemble du programme.

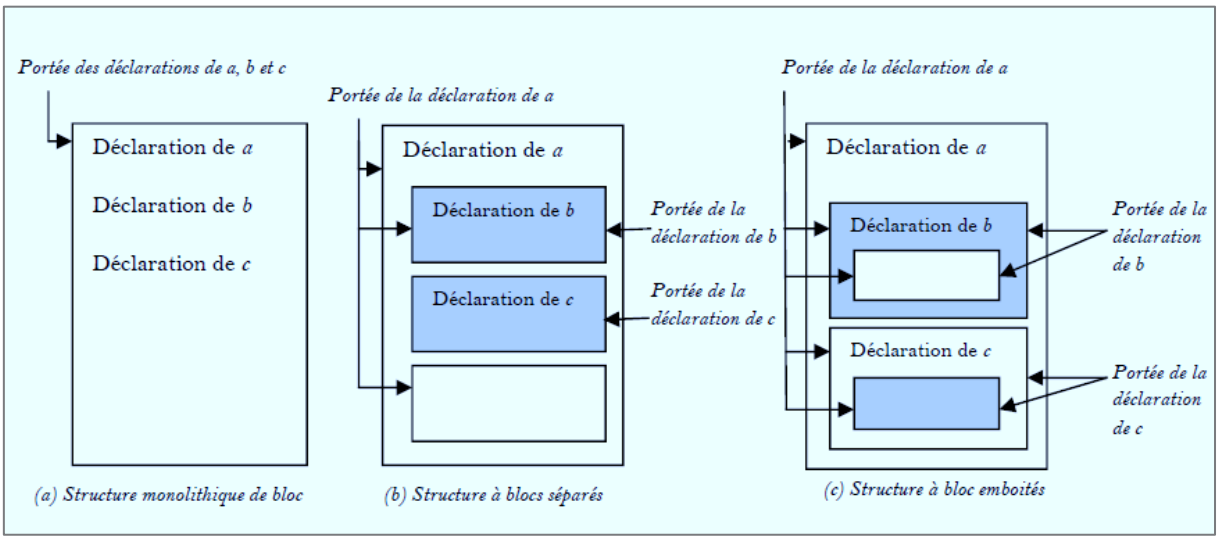

Figure 2 .5 : *structuration de programmes en blocs.*

Dans un langage à *structure de blocs imbriqués* (figure 2.5-c), les blocs peuvent être imbriqués dans d'autres blocs. Ceci est illustré par les nombreux langages de type ALGOL.. Le langage C a une structure de blocs imbriqués restrictive, dans laquelle les corps de fonctions ne peuvent pas se chevaucher, mais les commandes de blocs peuvent être librement imbriquées dans les corps de fonctions. JAVA a une structure de blocs imbriqués moins restrictive, dans laquelle les corps de méthodes et les classes internes peuvent être imbriqués dans une classe.

L'avantage de la structure en blocs imbriqués est qu'un bloc peut être situé n'importe où un identifiant doit être déclaré.

2.4.2 Portée et visibilité

Considérez toutes les occurrences de l'identificateur dans un programme. Nous devons distinguer deux types d'occurrences de l'identificateur:

Une *occurrence contraignante* de l'identificateur *I* est une occurrence où *je suis* lié à une entité *X*.

Une *occurrence appliquée* de *I* est une occurrence où l'on fait usage de l'entité *X à* laquelle *j'*ai été lié. À chaque occurrence appliquée de ce type, nous disons que *je désigne X*.

Dans un programme bien formé, chaque occurrence appliquée de *I* correspond exactement à une occurrence contraignante de *I*.

Mais que se passe-t-il si le même identificateur est déclaré en deux blocs *imbriqués* ? Cette possibilité, qui est illustrée dans la figure 2.6, est une conséquence de la structure des blocs imbriqués.

Considérons deux blocs imbriqués, de telle sorte que le bloc intérieur se trouve dans le champ d'application d'une déclaration de l'identificateur *I* dans le bloc extérieur :

Si le bloc intérieur *ne* contient *pas de* déclaration de *I*, alors les rences de *I* appliquées à l'intérieur et à l'extérieur du bloc intérieur correspondent à la même

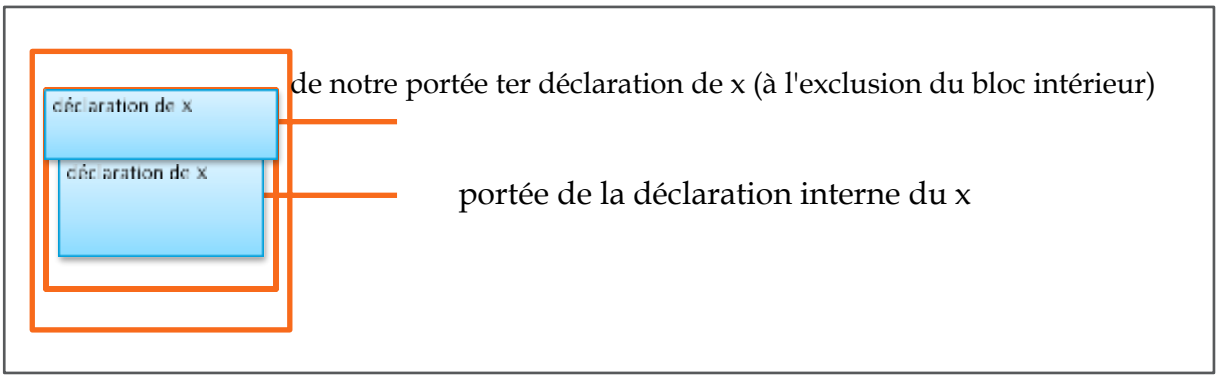

Figure 2.6 : *Cacher*.

déclaration de *I*. On dit alors que la déclaration de *I est visible dans* les blocs extérieur et intérieur.

Si le bloc intérieur contient une déclaration *de I* (comme dans la figure 2.6), alors toutes les occurrences de *I* appliquées à l'intérieur du bloc intérieur correspondent à la déclaration intérieure, et non extérieure, de I. On dit alors que la déclaration extérieure de *I est cachée* par la déclaration intérieure de *I*.

2.4.3 Règles de portée

Dans la plupart des langages modernes, la portée d'une liaison est déterminée de manière statique, c'est-à-dire au moment de la compilation. En C, par exemple, nous introduisons un nouveau champ d'application à l'entrée d'une sous-routine. Nous créons des liaisons pour les objets locaux et désactivons les liaisons pour les objets globaux qui sont "cachés" par les objets locaux du même nom. À la sortie d'une sous-routine, nous détruisons les liens pour les variables locales et réactivons les liens pour tous les objets globaux qui ont été cachés. Ces manipulations de liaisons peuvent à première vue apparaître comme des opérations d'exécution, mais elles ne nécessitent l'exécution d'aucun code : les portions du programme dans lesquelles une liaison est active sont entièrement déterminées au moment de la compilation.

2.4.3.1 Portée statique

Dans un langage à portée statique (lexicale), les liens entre les noms et les objets peuvent être déterminés au moment de la compilation en examinant le texte du programme, sans tenir compte du flux de contrôle au moment de l'exécution. Généralement, la liaison "courante" pour un nom donné se trouve dans la déclaration correspondante dont le bloc entoure le plus étroitement un point donné du programme,

bien que, comme nous le verrons, il existe de nombreuses variantes sur ce thème de base.

La règle de portée statique la plus simple est probablement celle des premières versions de base, dans lesquelles il n'y avait qu'une seule portée globale. En fait, il n'y avait que quelques centaines de noms possibles, chacun d'entre eux étant constitué d'une lettre éventuellement suivie d'un chiffre. Il n'y avait pas de déclarations explicites ; les variables étaient déclarées implicitement du fait de leur utilisation.

EXEMPLE : Code C pour illustrer l'utilisation de variables statiques

```
/*
Placez dans *s un nouveau nom commençant par la lettre « L » et continuant
avec la représentation ASCII d'un entier unique. Le paramètre s
est supposé pointer vers un espace suffisamment grand pour contenir un tel
nom; courtes ints utilisées ici, 7 caractères suffisent.
*/
void label_name (char *s) {
static  short int n; /* C garantit que les locaux statiques
sont initialisés à zéro */ sprintf (s, "L d\0", ++n); /* "print" formaté
sortie à s */
}
Comme exemple de l'utilisation de variables statiques, considérez le code
dans cet exemple.
```

Pour illustrer l'utilisation de variables statiques, examinons le code dans cet exemple.

Variables statiques dans C Le sous-routine label_name peut être utilisé pour générer une série de noms de chaînes de caractères distincts : L1, L2, ...Un compilateur peut utiliser ces noms dans sa sortie en langage assembleur.

2.4.3.2 Portée dynamique

Dans un langage à portée dynamique, les liaisons entre les noms et les objets dépendent du flux de contrôle au moment de l'exécution, et en particulier de l'ordre dans lequel les sous- programmes sont appelés. Par rapport aux règles de portée statique examinées dans la section précédente, les règles de portée dynamique sont généralement assez simples : la liaison "actuelle" pour un nom donné est celle rencontrée le plus récemment lors de l'exécution, et non encore détruite par le retour de sa portée.

2.4.4 Déclarations

Une *déclaration* est une construction qui sera *élaborée* pour produire des liaisons.

Toutes les déclarations produisent des liaisons; certaines ont des effets secondaires, comme la création de variables. Nous utiliserons le terme *définition* pour une déclaration dont le *seul* effet est de produire des liaisons.

2.4.4.1 Déclarations de type

Une *déclaration de type* lie un identifiant à un type. On peut distinguer deux types de déclaration de type. Une déclaration de type *définition* lie un identifiant à un type existant. Une déclaration de *nouveau type lie un* identifiant à un nouveau type qui n'est équivalent à aucun type existant.

2.4.4.2 Déclarations de variables

Une *déclaration de variable*, dans sa forme la plus simple, crée une seule variable et lie un identificcateur à cette variable. La plupart des langages de programmation permettent également à une déclaration de variable de créer plusieurs variables et de les lier à différents identifiants.

Une *variable qui renomme définition* lie un identifiant à une variable *existante.* En d'autres termes, elle crée un *alias*. ADA prend en charge le renommage des variables définitions, mais peu d'autres langages le font.

EXEMPLE : Déclarations de variables en ADA

La déclaration de variable ADA suivante :

```
count: Integer;
```

crée une variable entière, et lie le compte identifiant à cette variable. La déclaration de la variable suivante :

```
count: Integer := 0;
```

fait de même, mais initialise également la variable à zéro. La dé fi nition de renommage de variable ADA suivante:

```
pop: Integer renames population(state);
```

lie l'identificateu pop à une variable entière existante, à savoir une composante de la population de la variable tableau. En d'autres termes, pop est maintenant un alias de la variable composante population(state). Par la suite, si on le souhaite, pop peut être utilisé pour accéder à cette variable composante de manière concise et à efficacement
:

```
pop := pop + 1 ;
```

2.4.4.3 déclarations de constants

Une *déclaration de constante* lie un identifiant à une valeur constante. Une déclaration de constante a généralement la forme "**const** *I* = E **;** ", et lie l' identifiant *I* à la valeur de l'expression *E* ; le type de la constante est déterminé par le type de E. Une autre forme est "**const** *T I* = E **;** ", qui indique explicitement que le type de la constante est *T*.

Dans certains langages de programmation, *E* doit être une expression qui peut être évaluée au moment de la compilation. *E* peut être exprimé en termes de littéraux, d'autres constantes, d'opérateurs, etc.

Dans les langages plus modernes (dont C++, JAVA et ADA), le *E* est une expression arbitraire. En particulier, *E* peut accéder à des variables ou à des paramètres, auquel cas il doit être évalué à l'exécution.

2.4.4.4 Déclarations collatérales

Une *déclaration collatérale* compose des sous-déclarations qui doivent être élaborées indépendamment les unes des autres. Ces sous-déclarations ne peuvent pas utiliser des liants produits par les autres. La déclaration collatérale fusionne les liens produits par ses sous-déclarations. Les déclarations collatérales sont peu courantes dans les langages impératifs et orientés objet, mais elles sont courantes dans les langages fonctionnels et logiques.

EXEMPLE : Déclarations collatérale en ML

Voici une définition de la valeur collatérale en ML :

```
                    val e = 2.7183
                    and pi = 3.1416
combine des sous-déclarations indépendantes des identificateurs e et
pi. Cette déclaration collatérale produit les liaisons suivantes :

                    { e→the real number 2.7183,
                    pi→the real number 3.1416 }
La portée de ces liaisons commence à la fin de la déclaration
collatérale. Ainsi, nous ne pouvait pas étendre la déclaration
collatérale comme suit :
                    val e = 2.7183
                    and pi = 3.1416
                    and twicepi = 2 * pi (* illegal! *)
Nous utiliserions plutôt une déclaration séquentielle :
                    val e = 2.7183
                    and pi = 3.1416;
                    val twicepi = 2 * pi
```

2.4.4.5 Définitions de procédure

Une *définition de procédure* lie un identifià une *procédure*. Dans la plupart des langages de programmation, on peut lier un identifiant à une procédure de fonction ou à une procédure proprement dite.

EXEMPLE : Définitions de procédure en C++

```
                    bool even (int n) {
                    return (n % 2 == 0);
                    }
lie l'identifiant même à une procédure de fonction qui teste si son
argument entier est égal.
La définition suivante de la procédure C++ :
                    void double (int& n) {
                    n *= 2;
                    }
lie l'identificateur double à une procédure appropriée qui
double son argument, une variable entière.
```

2.4.4.6 Déclarations récursives

Une *déclaration récursive* est une *déclaration* qui utilise les liens qu'elle produit elle-même. Une telle construction est importante car elle nous permet de définir les types récursifs et les procédures.

2.4.4.7 Déclarations séquentielles

Une *déclaration séquentielle* compose des sous-déclarations qui doivent être élaborées les unes après les autres. Chaque sous-déclaration peut utiliser les liants produits par toute sous-déclaration *antérieure*, mais pas ceux produits par les sous- déclarations *suivantes.* La déclaration séquentielle fusionne les liens produits par ses sous-déclarations. Les déclarations séquentielles sont supportées par la plupart des langages impératifs et orientés objet.

EXEMPLE : Déclaration séquentielle en ADA

Dans la déclaration séquentielle ADA suivante :

```
count: Integer := 0;
procedure bump is begin
count := count + 1;
end;
```

L'identifiant count est déclaré dans la première sous-déclaration et utilisé dans la seconde. La déclaration séquentielle produit les liaisons suivantes:

```
{ count → une variable de caractère, bump → une procédure propre
}
```

2.4.5 Blocs

Un *bloc* est une construction de programme qui délimite le champ d'application des déclarations qu'il contient. Si nous permettons à une expression de contenir une déclaration locale, nous avons une *expression en bloc*. Si nous permettons à une commande de contenir une déclaration locale, nous avons une *commande de bloc.*

2.4.5.1 Expressions en bloc

Une expression en *bloc* est une forme d'expression qui contient une déclaration locale (ou un groupe de déclarations) *D* et une sous-expression *C*. Les liaisons produites par *D ne* sont utilisées que pour évaluer *E*.

EXEMPLE : Expressions en bloc en HASKELL

Supposons que les variables a, b et c contiennent les longueurs des trois côtés d'un triangle. L'expression de bloc HASKELL suivant obtient l'aire de ce triangle :

```
let
   s = (a + b + c)/2.0
in
   sqrt(s*(s-a)*(s-b)*(s-c))
```

2.4.5.2 Commandes en bloc

Une *commande en bloc* est une forme de commande qui contient une déclaration locale (ou un groupe de déclarations) *D* et une sous-commande *C*. Les liaisons produites par *D ne* sont utilisées que pour l'exécution de *C*.

EXEMPLE : Commande en bloc de JAVA

Supposons que x et y sont des variables entières, et que leur contenu doit être trié de telle sorte que x contienne le plus petit entier. Nous pouvons facilement implémenter ceci en utilisant une variable auxiliaire, déclarée localement à une commande de bloc. En JAVA :

```
si (x > y) {
    int z = x ;
    x = y ;
    y = z ;
}
```

La portée de z n'est que la commande de bloc, et la durée de vie de la variable désignée par z est l'activation de la commande de bloc.

2.5 Abstraction procedural

En programmation, l'abstraction fait allusion à la distinction que nous faisons entre *ce que* fait une unité de programme et *comment* cette unité de programme fonctionne. Cela permet de séparer les préoccupations des programmeurs qui utilisent l'unité de programme et celles du programmeur qui la met en œuvre.

2.5.1 Procédures appropriées

Une *procédure appropriée* comprend une commande à exécuter et, lorsqu'elle est appelée, elle met à jour les variables. Le programmeur d'application observe uniquement ces mises à jour, et non les étapes par lesquelles elles ont été effectuées.
Une définition de procédure correcte en C ou C++ a la forme :

vide *I* (*FPD1, ... , FPDn*) *B*

où *I* est l'identifiant de la fonction, les FPDi sont les déclarations formelles de paramètres, et *B* est une commande de bloc appelée *corps* de la procédure. La procédure sera appelée par une commande de la forme "*I(AP1, ... ,APn)*;", où les *APi sont des* paramètres réels. Cet appel de procédure provoque l'exécution de *B.*
Notez qu'une définition de procédure propre à C ou C++ n'est qu'un cas particulier de définition de fonction, qui se distingue uniquement par le fait que le type de résultat est *nul*.

EXEMPLE : Définition de la procédure appropriée en C++

Considérons la définition suivante de la procédure C++ :

```
void sort (int a[], int l, int r) {
    int minpos; int min;
    for (int i = l; i < r; i++) {
        minpos = i; min = a[minpos];
        for (int j = i+1; j <= r; j++) {
            if (a[j] < a[i]) {
                minpos = j; min = a[minpos];
            }
        if (minpos != i) {
            a[minpos] = a[i]; a[i] = min;
        }
    }
}
```

Cela lie le tri des identificateurs à une procédure adéquate. Le programmeur d'application estime que le résultat d'un appel de procédure tel que "'sort(nums, 0, n-1) ;" sera de trier les valeurs en nums[0], . . . *nums[0], ...*, nums[n-1] en ordre croissant. L'exécutant estime que le corps de la procédure utilise l'algorithme de sélection-tri.

L'implémentateur pourrait plus tard substituer un algorithme plus efficace, tel que le tri rapide:

```
void sort (int a[], int l, int r) {
    int p;
    if (r > l) {
        partition(a, l, r, p);
        sort(a, l, p-1);
        sort(a, p+1, r);
    }
}
```

Toutefois, le point de vue du programmeur de l'application serait inchangé.

2.5.2 Les Fonctions

Une *procédure de fonction* (ou simplement une *fonction*) incarne une expression à évaluer. Lorsqu'elle est appelée, la procédure de fonction donne une valeur connue sous le nom de *résultat*. Le programmeur d'application n'observe que ce résultat, et non les étapes de son calcul.

Une définition de fonction en C ou C++ a la forme :

T I (FPD1, ... ,FPDn) B

où *I* est l'identifiant de la fonction, les *FPDi* sont des déclarations *formelles de paramètres, T est* le type de résultat et *B* est une commande de bloc appelée **corps** *de la* fonction. *B* doit contenir au moins un retour de la forme ''*retour E* ; '', où *E est* une expression de type *T*. La procédure de la fonction sera appelée par une expression de la forme "*I(AP1, ... ,APn)*'', où les *APi* sont des *paramètres réels*. Cet appel de fonction provoque l'exécution de *B*, et le premier retour exécuté dans *B* détermine le résultat.

EXEMPLE : Définition d'une fonction en C++

La définition suivante de la fonction C++ :

```
float power (float x, int n) {
   float p = 1.0;
   for (int i = 1; i <= n; i++)
      p *= x;
   return p;
}
```

Lie la puissance à à une procédure de fonction avec deux paramètres formels (x et n) et un résultat type **float**. Cela permet de calculer la nième puissance de x, en supposant que n est non négatif. Notez que l'utilisation de variables locales et itération pour

calculer le résultat de la fonction. La définition suivante de la fonction C++ utilise plutôt la récursion :

```
float power (float x, int n) {
    if (n == 0)
        return 1.0;
    else
        return x * power(x, n-1);
}
```

2.5.3 Paramètres et arguments

Si nous transformons simplement une expression en une procédure de fonction, ou une commande en une procédure correcte, nous obtenons généralement une procédure qui effectuera toujours le même calcul chaque fois qu'elle sera appelée.

Pour exploiter pleinement la puissance du concept de procédure, nous devons *paramétrer* les procédures relatives aux entités sur lesquelles elles opèrent.

2.5.3.1 Mécanismes des paramètres de référence

Un *mécanisme de paramètres de référence* permet de lier directement le paramètre formel *FP* à l'argument lui-même. Cela donne lieu à une sémantique simple et uniforme du passage des paramètres, adaptée à tous les types de valeurs dans le langage de programmation (pas seulement les valeurs de la première classe). Les mécanismes des paramètres de référence apparaissent sous plusieurs formes dans les langages de programmation :

Dans le cas d'un *paramètre constant*, l'argument doit être une valeur. *FP* est lié à la valeur de l'argument lors de l'activation de la procédure. Ainsi, toute inspection de *FP* est en fait une inspection indirecte de la valeur de l'argument.

Dans le cas d'un *paramètre de procédure, l'*argument doit être une procédure. *FP* est lié à la procédure d'argumentation lors de l'activation de la procédure appelée. Ainsi, tout appel à *FP* est en fait un appel indirect à la procédure d'argumentation.

Dans le cas d'un *paramètre variable*, l'argument doit être une variable. *FP* est lié à la variable argument lors de l'activation de la procédure. Ainsi, toute inspection (ou mise à jour) de *FP* est en fait une inspection (ou mise à jour) indirecte de la variable argument.

Un inconvénient des paramètres variables est que l'alias devient un risque. L'*alias* se produit lorsque deux ou plusieurs identificateurs sont simultanément liés à la même variable (ou un identificateur est lié à une variable composite et un second identificateur à l'une de ses composantes). L'alias tend à rendre les programmes plus difficiles à comprendre et à raisonner.

EXEMPLE : Alias en C ++

Considérons la procédure C++ suivante avec trois paramètres variables :

```
void pour (float& v1, float& v2, float& v) {
// Donné deux pots contenant respectivement v1 et v2 litres d'eau,
// déterminer l'effet de verser v litres du premier pot dans le second.
 v1 -= v;
v2 += v;
}
```

Supposons que les variables x, y et z aient actuellement des valeurs de 4,0, 6,0 et 1,0, respectivement. Ensuite, l'appel " pour(x, y, z) ; " mettrait à jour x à 3.0 et y à 7.0, comme nous le ferions attendre. Mais maintenant, l'appel " pour(x, y, x) ; " mettrait x à jour à 0,0 mais laisserait y à 7,0, de manière inattendue.. Pour comprendre le comportement de ce dernier appel, il faut savoir que les versions v1 et v sont toutes deux les pseudonymes de x, et donc les uns des autres.

Pourquoi ce comportement est-il inattendu ? Lorsque nous lisons le corps de la procédure, nous avons tendance à supposons que v1 et v désignent des variables distinctes, de sorte que l'affectation "v1 -= v ; " s'actualise Presque toujours, l'hypothèse selon laquelle des identificateurs distincts dénotent des variables distinctes est justifiée. Mais très occasionnellement (en présence d'un alias), cette hypothèse n'est pas justifiée, et puis nous sommes pris par surprise.

2.5.3.2 Copier les mécanismes des paramètres

Un *mécanisme de paramètre de copie* permet de copier une valeur dans et/ou hors d'une procédure. Le paramètre formel *FP* désigne une variable locale de la procédure. Une valeur est copiée dans *FP* lors de l'appel de la procédure, et/ou est copiée hors de *FP* (vers une variable d'argument) à son retour. La variable locale est créée lors de l'appel de la procédure, et détruite à son retour.

Il existe trois mécanismes possibles de paramétrage de la copie :

Un paramètre de copie (également appelé *paramètre de valeur*) fonctionne comme suit. Lorsque la procédure est appelée, une variable locale est créée et initialisée avec la valeur de l'argument. À l'intérieur de la procédure, cette variable locale peut être inspectée et même mise à jour. (Cependant, toute mise à jour de la variable locale n'a aucun effet en dehors de la procédure).

Un paramètre de sortie (également appelé *paramètre de résultat*) est une image miroir d'un paramètre d'entrée. Dans ce cas, l'argument doit être une variable. Lorsque la procédure est appelée, une variable locale est créée mais non initialisée. Lorsque la procédure revient, la valeur final de cette variable locale est attribuée à la variable argument.

Un paramètre de copy-in-copy-out (également appelé *paramètre de valeur-résultat*) combine les paramètres de copy-in et de copy-out. Dans ce cas également, l'argument doit être une variable. Lorsque la procédure est appelée, une variable locale est créée et initialisée avec la valeur actuelle de la variable argument. Lorsque la procédure revient, la valeur final de cette variable locale est attribuée à la variable d'argument.

EXEMPLE : Mécanismes de paramètres de copie en ADA

Considérez les procédures ADA suivantes :

```
type Vector is array (1 .. n) of Float;
procedure add (v, w: in Vector;
sum: out Vector) is
(1) begin
    for i in 1 .. n loop
        sum(i) := v(i) + w(i);
    end loop;
(2) end;
procedure normalize (u: in out Vector) is
(3) s: Float := 0.0;
begin
    for i in 1 .. n loop
        s := s + u(i)**2;
    end loop;
        s := sqrt(s);
    for i in 1 .. n loop
        u(i) := u(i)/s;
    end loop;
(4) end;
```

Supposons également que le mot clé **in** *signifie un paramètre de copy-in*, que **out** *signifie un paramètre de copy-out*, et que **in out** *signifie un paramètre de copy-in-copy-out*.

Si a, b et c sont des variables vectorielles, l'appel de procédure " add(a, b, c) ; " fonctionne comme suit. Au point (1), les variables locales nommées v et w sont créées, et les valeurs de a et b sont attribuées à v et w, respectivement. Toujours au point (1), une variable locale nommée sum est créée mais n'est pas initialisée. Le corps de la procédure met alors à jour la somme. Au point (2), la valeur finale de sum est attribuée à c.

L'appel de procédure " normalize(c)" fonctionne comme suit. Au point (3), la variable locale u est créée, et la valeur de c est attribuée à u. Le corps normalize met alors à jour u. Au point (4), la valeur finale de u est attribuée à c.

2.6 Conclusion

Dans ce chapitre, nous avons étudié les valeurs des types primitifs, composites et récursifs supportés par les langages de programmationous, ensuite nous avons introduit un modèle de stockage simple qui nous permet de comprendre le comportement des variables en comparant les durées de vie des variables globales, locales, tas (heap) et les variables persistantes, puis nous avons étudié les formes de commande trouvées dans les langages de programmation et nous avons étudié les concepts de liaisons et de portées, après nous avons étudié les fonctions et les procédures appropriées, enfin nous avons étudié les paramètres et les arguments.

2.7 Exercices

1. Écrivez des types récursifs pour représenter des listes d'entiers de longueur quelconque, (a) en C ou C++, et (b) en ADA.
2. Choisissez un langage de programmation (tel que C, C++ ou JAVA) qui ne prend pas en charge les constructions de structures/enregistrements et de tableaux sans restriction. Concevez une extension à ce langage pour permettre de telles constructions. Permettez à vos constructions d'avoir des sous-expressions arbitraires (pas seulement des littéraux) des types appropriés. Permettez à vos constructions d'être utilisées partout où les expressions sont autorisées (pas seulement dans les déclarations de variables). Soyez attentif aux interactions possibles entre vos extensions et d'autres parties du langage.
3. Considérez le programme C suivant :

```
void main () {
 int f;
 f = fac(3);
 }
 int factorial (int n) {
 int p, i;
 p = 1;
 for (i = 2; i <= n; i++)
 p *= i;
 return p;
 }
```

   - Faites un diagramme, similaire à la figure 2.4, montrant la durée de vie des variables locales dans ce programme. (Notez que le paramètre formel n est, en fait, une variable locale de la fonction factorielle).

4. Considérez la procédure ADA suivante:

```
procedure multiplier (m, n: in out Integer) is
begin
m := m * n;
put(m); put(n);
end;
```

- Notez que le mécanisme de paramètre "copy-in-copy-out" est utilisé ici (puisque Integer est un type primitif). Supposons que i contienne 2 et que j contienne 3. Montrez ce qui est écrit par les appels de procédure suivants : multiplier(i, j) ; multiplier(i, i) ;
- Supposons maintenant que le mécanisme des paramètres variables ait été utilisé à la place. Montrez ce qui serait écrit par chacun des appels de procédure ci-dessus. Expliquez toute différence.

5. Considérez le programme C suivant :

```
int add (int i) {
return i + d;
}
void p () {
const int d = 1;
(1) print(add(20));
}
void q () {
const int d = 2;
(2) print(add(20));
}
```

- Si la portée du language est dynamique, qu'est-ce qui serait imprimé aux points (1) et (2) ?
- Si le langage est statiquement délimitée, que se passe-t-il ?

6. Considérez l'implémentation d'une procédure *P* avec un paramètre formel *FP* de type *T*.

- Supposons que le mécanisme du paramètre "copy-in-copy-out" soit utilisé. Qu'est-ce exactement est passé à *P*, une valeur ou une adresse ? Que se passe-t-il lorsque *P est* appelé ? Que se passe-t-il dans *P* lorsqu'il inspecte *FP* ? Que se passe-t-il dans *P* lorsqu'il met à jour *FP ? Que se* passe-t-il si quelque chose se produit lorsque *P* revient ?

CHAPITRE 3

CONCEPTS AVANCES

3.1 Introduction

Dans ce chapitre, nous alons examiner certains concepts plus avancés : abstraction de données (types abstraits, packages et classes), abstraction générique (ou modèles), systèmes de types polymorphisme, séquenceurs (y compris les exceptions) et la concurrence (primitives, régions critiques conditionnelles et moniteurs). Ces concepts plus avancés se retrouvent dans les langages les plus modernes.

3.2 Abstraction de données

L'abstraction de données est devenue essentielle au génie logiciel. L'abstraction fournie par les modules qui présente au moins trois avantages importants :

1. Il réduit la *charge conceptuelle* en minimisant la quantité de détails auxquels le programmeur doit penser à un moment donné.
2. Elle permet de *contenir les défauts* en empêchant le programmeur d'utiliser un composant de programme de manière inappropriée, et en limitant la partie du texte d'un programme dans laquelle un composant donné peut être utilisé, limitant ainsi la partie qui doit être prise en compte lors de la recherche de la cause d'un bug.
3. Il offre un degré d'*indépendance* important entre les composants des programmes, ce qui permet d'en attribuer plus facilement la construction à des personnes distinctes, de modifier leurs implémentations internes sans changer le code externe qui les utilise, ou de les installer dans une bibliothèque où ils peuvent être utilisés par d'autres programmes.

3.2.1 Unités de programme

Une *unité de programme* est toute partie nommée d'un programme qui peut être conçue et implémentée de manière plus ou moins indépendante. Une unité de programme bien conçue a un seul objectif et possède une interface de programme d'application simple. Si elle est bien conçue, une unité de

programme sera probablement *modifiable* (capable d'être changée sans forcer des changements majeurs dans d'autres unités de programme), et est potentiellement *réutilisable* (capable d'être utilisée dans de nombreux programmes).

3.2.1.1 Package

Un *package* est un groupe de plusieurs composants déclarés dans un but commun. Ces composants peuvent être des types, des constantes, des variables, des procédures, ou encore toute entité pouvant être déclarée dans le langage de programmation.

Nous considérerons ici un simple *Package* dont les composants sont tous publics, c'est-à-dire visibles pour le code d'application qui utilise le Package.

EXEMPLE : Package en ADA

Le package ADA suivant regroupe des déclarations de types, de constantes et de variables :

```
package Earth is
    type Continent is (
        Africa, Antarctica, Asia, Australia,
        Europe, NAmerica, SAmerica);
    radius: constant Float := 6.4e3; -- km
    area: constant array (Continent) of Float := (
        30.3e6, 13.0e6, 43.3e6, 7.7e6,
        10.4e6, 24.9e6, 17.8e6); -- km2
    population: array (Continent) of Integer;
end Earth;
```

Le package Earth produit l'ensemble de liaisons suivant :

```
    { Continent → the type Continent,
    radius → the real number 6.4×103,
    area → the array value {Africa→30.3×106, . . .},
    population → an array variable of type (Continent→Integer)}
where:
    Continent = {Africa, Antarctica, Asia, Australia, Europe,
    NAmerica, SAmerica}
```

Le code d'application suivant utilise certains éléments du package Earth :

```
for cont in Earth.Continent loop
      put(Float(Earth.population(cont)
      / Earth.area(cont));
end loop;
```

Ici, Earth.Continent désigne un composant type du package Earth, Earth.population désigne une composante variable, et Earth.area désigne une composante valeur.

3.2.1.2 Encapsulation

Pour que son API reste simple, un package ne rend généralement visible que certains de ses composants au code d'application qui l'utilise ; ces composants sont dits *publics*. Les autres composants ne sont visibles qu'à l'intérieur du package; ces composants sont dits *privés* et ne servent qu'à soutenir l'implémentation des composants publics. Un package avec des composants privés cache des informations qui ne sont pas pertinentes pour le code d'application. Cette technique pour réaliser une API simple est appelée *encapsulation*.

3.2.2 Types abstraits

Un *type abstrait* est un type dont l'identifiant est public mais dont la représentation est privée. Un type abstrait doit être équipé d'un groupe d'*opérations* pour accéder à sa représentation. Les opérations sont généralement des procédures et des constantes. Les valeurs du type abstrait sont defined pour être juste les valeurs qui peuvent être générées par l'utilisation répétée des opérations.

3.2.3 Objets et classes

Un *objet* contient des informations d'état (données, représentées par d'autres objets) et des opérations (code). Chaque objet est une instance d'une *classe,* qui détermine les données que l'objet conserve comme informations d'état et les messages que l'objet comprend. Le *protocole de* la classe est l'ensemble des messages que ses instances comprennent. Les objets interagissent en s'envoyant des *messages* entre eux. Ces messages sont comme des appels de procédure ; les procédures sont appelées *méthodes*.

3.2.3.1 Les classes

Une classe est un modèle pour un ensemble d'objets. Elle établit ce que seront ses données (type et visibilité) et fixe le nom, la signature, la visibilité et l'implémentation de chacune de ses méthodes. Dans un langage avec classes,

chaque objet appartient à (au moins) une classe. Par exemple, un objet tel que celui de la figure 3.1 pourrait être une instance de la classe suivante :

```
class Counter{
    private int x;
    public void reset(){
        x = 0;
    }
    public int get(){
        return x;
    }
    public void inc(){
        x = x+1;
    }
}
```

On peut voir que la classe contient l'implémentation de trois méthodes qui sont déclarées publiques (visibles par tous), tandis que la variable d'instance x est privée (inaccessible de l'extérieur de l'objet lui-même mais accessible dans l'implémentation des méthodes de cette classe).

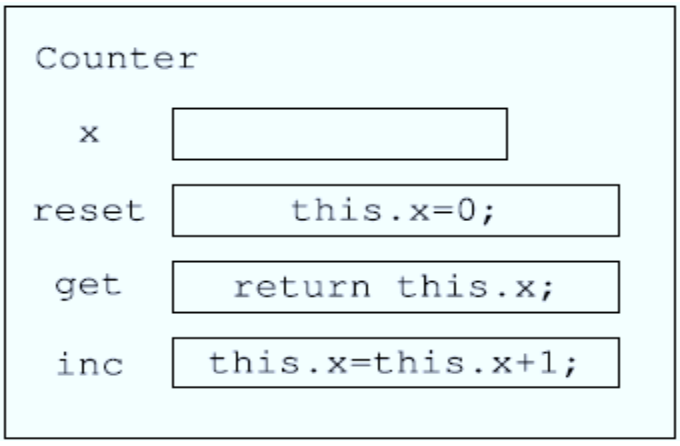

Figure 3.1 : *Un objet pour un compteur(Counter)*

Les objets sont créés dynamiquement par l'*instanciation* de leur classe. Un objet spécifique est attribué dont la structure est déterminée par sa classe. Cette opération diffère fondamentalement d'un langage à l'autre et dépend du statut linguistique des classes. Dans Simula, une classe est une procédure qui renvoie un pointeur vers un enregistrement d'activation contenant des variables locales et des définitions de fonctions (par conséquent, les objets qui sont des instances d'une classe sont des fermetures). Dans Smalltalk, une classe est linguistiquement un objet qui agit comme un schéma pour la définition de l'implémentation d'un ensemble d'objets.

En Java et C++ , les classes correspondent à un type et tous les objets qui instancient une classe A sont des valeurs de type A. En adoptant ce point de vue en Java, avec son modèle de référence pour les variables, nous pouvons créer un contre-objet :

```
Counter  c = new Counter  () ;
```

Le nom c, de type Counter , est lié à un nouvel objet qui est une instance de Counter . Nous pouvons créer un nouvel objet, distinct du précédent mais avec la même structure, et le lier à un autre nom :

```
Counter  d = new Counter  () ;
```

A droite du symbole d'affectation, on peut voir deux opérations distinctes : la création des objets (affectation de la mémoire nécessaire, nouveau constructeur) et l'initialisation (invocation du *constructeur* "classe", représenté par le nom de la classe.

Nous pouvons certainement supposer que le code des méthodes est stocké une seule fois dans sa classe et que lorsqu'un objet doit exécuter une méthode spécifique, ce code est recherché dans la classe dont il est une instance. Pour que cela se produise, le code de la méthode doit accéder correctement aux variables de l'instance qui sont différentes pour chaque objet et ne sont donc pas toutes stockées ensemble dans la classe mais à l'intérieur de l'instance (comme le montre le graphique de la figure 3.2). Dans la figure, les méthodes de la classe Counter font référence à ses variables d'instance en utilisant le nom suivant Nous avons déjà vu que lorsqu'un objet reçoit un message demandant l'exécution d'une méthode, l'objet lui-même est un paramètre implicite de la méthode. Lorsqu'une référence est faite à une variable d'instance dans le corps d'une méthode, il y a une référence implicite à l'objet qui exécute actuellement la méthode. D'un point de vue linguistique, l'objet courant est généralement désigné par un nom unique, généralement self ou this. Par exemple, la définition de la méthode inc pourrait s'écrire comme suit. Ici, la référence implicite à l'objet courant est rendue explicite :

```
public void inc(){
this.x = this.x+1;
}
```

Dans le cas d'un appel à une méthode par this, le lien entre la méthode et le code auquel elle se réfère est déterminé de manière dynamique. C'est un aspect important de ce paradigme.

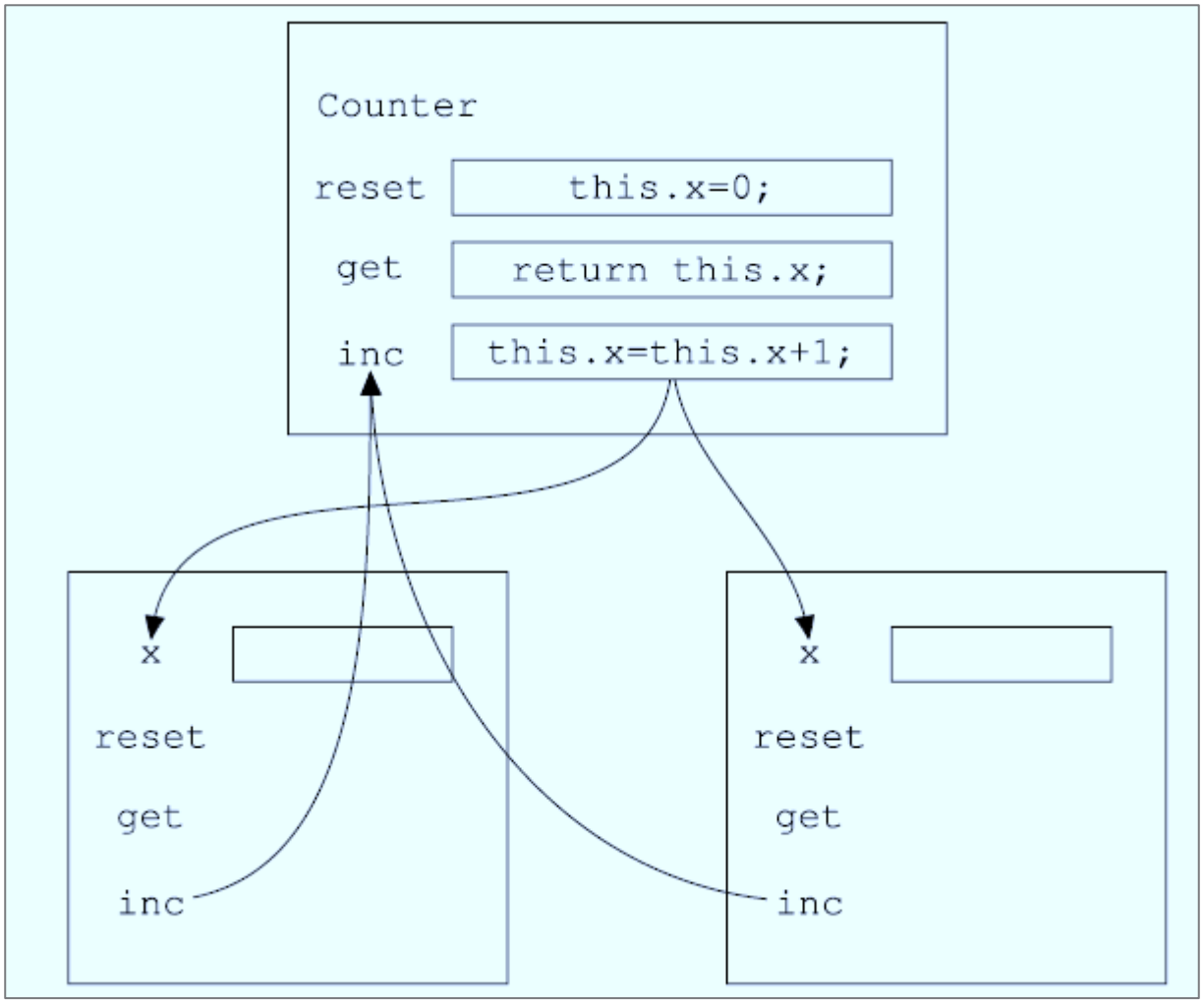

Figure 3.2 : L'implémentation des méthodes se fait à l'intérieur des classes

3.2.3.2 Classes d'abstraction

Les classes abstraites servent à fournir des interfaces et peuvent recevoir des implémentations dans des sous-classes qui redéfinissent (dans ce cas, définissent pour la première fois) la méthode qui manque d'implémentation. Les *classes abstraites* correspondent également à des types et le mécanisme qui fournit des implémentations pour leurs méthodes génère également des sous-types

EXEMPLE : Classe et sous-classes *abstraite en* JAVA

Il est plus naturel de considérer les points, les cercles et les rectangles comme des cas particuliers de *formes*. S'il est logique de construire des points, des cercles et des rectangles, il n'en va pas de même pour un objet de forme.

La classe abstraite JAVA suivante saisit la notion de forme :

```
abstract class Shape {
  // Les objets de forme représentent différentes sortes de formes dans le plan xy.
  protected double x, y;
  // Le centre de cette forme est représenté par ses coordonnées cartésiennes (x,y).
  public double distance () {
  // Retourne la distance du centre de cette forme à partir de (0, 0).
  return Math.sqrt(x*x + y*y);
  }
  public final void move (double dx, double dy) {
  // Déplacez cette forme par dx dans la direction x et par dy dans la direction y.
  x += dx; y += dy;
  }
  public abstract void draw ();
  // Dessinez cette forme sur l'écran.
  }
```

Chaque objet Shape aura des composantes variables x et y, et sera équipé de méthodes appelées distance, move et draw. Les méthodes de distance et de move sont ici definées de la manière habituelle, le premier étant remplaçable. Cependant, la méthode draw ne peut raisonnablement pas être defined ici (que ferait-elle ?), il s'agit donc d'une méthode abstraite.

3.2.3.3 Sous-classes et héritage

Une sous-classe peut redéfinir les méthodes de sa superclasse. Mais que se passe-t-il lorsque la sous-classe ne les redéfinit pas ? Dans ce cas, la sous-classe hérite des méthodes de la superclasse, en ce sens que l'implémentation de la méthode dans la superclasse est mise à la disposition de la sous-classe.

L'héritage est un mécanisme qui permet de définir de nouveaux objets en se basant sur la réutilisation d'objets préexistants. L'héritage permet la réutilisation de codes dans un contexte extensible. En modifiant l'implémentation d'une méthode, une classe met automatiquement la modification à la disposition de toutes ses sous-classes, sans aucune intervention de la part du programmeur.

- **Héritage et visibilité**

Une sous-classe est un client particulier de la superclasse. Elle utilise les méthodes de la superclasse pour étendre les fonctionnalités de celle-ci, mais doit parfois accéder à certaines données non publiques. De nombreux langages introduisent donc une

troisième vue d'une classe : une pour les sous-classes si la sous-classe a accès à certains détails de l'implémentation de la superclasse, la sous-classe dépend de manière beaucoup plus étroite de la superclasse. Toute modification de la superclasse nécessitera une modification de la sous- classe. D'un point de vue pragmatique, cela n'est raisonnable que si les deux classes sont "proches" l'une de l'autre,

❖ **Héritage simple et multiple**

Dans certains langages, une classe peut hériter d'une seule superclasse immédiate. La hiérarchie d'héritage est donc un arbre et on dit que le langage a un héritage unique (ou simple). D'autres langages, en revanche, permettent à une classe d'hériter des méthodes de plus d'une superclasse immédiate; la hiérarchie d'héritage dans ce cas est un graphe orienté acyclique et le langage a un héritage multiple.

Seuls quelques langages supportent l'héritage multiple (parmi lesquels Eiffel et C++), car il présente des problèmes qui n'ont pas de solution élégante, que ce soit au niveau conceptuel ou au niveau de l'implémentation. Les problèmes fondamentaux sont liés aux conflits de noms. Nous avons un conflit de noms lorsqu'une classe C hérite simultanément de A et B, qui fournissent tous deux une implémentation pour des méthodes ayant la même signature. Voici un exemple simple :

EXEMPLE : Classe et sous-classes en C

```
class A{
        int x;
        int f(){
             return x;
        }
}
class B{
        int y;
        int f(){
            return y;
        }
}
class C extending A,B{
       int h(){
          return f();
        }
}
```

3.2.3.4 Interfaces

Une *interface* est une unité de programme qui déclare (mais ne fait pas define) les opérations que certaines autres unités de programme doivent être definées. La relation de sous-type est introduite en Java en utilisant soit la clause extends (pour

définir les sous- classes), soit la clause implements lorsqu'une classe est déclarée sous-type d'une ou plusieurs interfaces.

En Java, une interface est une sorte de classe abstraite incomplète dans laquelle seuls les noms et les signatures de méthodes sont inclus-ils ne comprennent pas les implémentations. La relation d'héritage est introduite avec la clause "extends" chaque fois que la sous-classe ne redéfinit pas une méthode et utilise donc une implémentation de la superclasse. Il convient de noter qu'il n'y a jamais d'héritage d'une interface car l'interface n'a rien à hériter. Le langage contraint chaque classe à avoir une seule superclasse immédiate (c'est-à-dire qu'une seule superclasse peut être nommée dans un extends), mais permet qu'une seule classe (ou interface) implémente plus d'une interface :

EXEMPLE : Héritage et sous-types en Java

```
interface A{
    int f();
}
interface B{
    int g();
}
class C{
    int x;
    int h(){
       return x+2;
    }
}
class D extends C implements A,B{
    int f(){
       return x;
    }
    int g(){
    return x+1;
    }
}
```

Java a donc un héritage unique. La hiérarchie d'héritage est un arbre organisé par des clauses d'extension. En outre, la relation d'héritage est, en Java, toujours une sous-hiérarchie de la hiérarchie des sous-types.

3.3 Abstraction générique

3.3.1 Unités génériques et instanciation

Une *unité générique* est une unité de programme qui est paramétrée par rapport aux entités dont elle dépend. L'*instanciation d*'une unité générique génère une unité de programme ordinaire, dans laquelle chacun des paramètres formels de l'unité générique est remplacé par un argument.
Une unité générique peut être instanciée aussi souvent que nécessaire, générant ainsi sur demande une famille d'unités de programmes similaires. Ce concept met un outil très puissant entre les mains des programmeurs. Une seule unité générique peut être instanciée plusieurs fois dans le même programme, évitant ainsi la duplication du code du programme. Elle peut également être instanciée dans de nombreux programmes différents, ce qui facilite sa réutilisation.

3.3.1.1 Classes génériques en C++

Le C++ prend en charge à la fois les fonctions génériques (appelées *modèles de fonctions*) et les classes génériques (appelées *modèles de classes*). Nous nous concentrerons ici sur les classes génériques.

EXEMPLE : Classe générique en C++

Le générique de C++ suivant encapsule un type abstrait dont les valeurs sont des files de caractères délimitées. La classe est paramétrée en fonction de la capacité de la file d'attente.

```
template
      <int capacity>
class File {
      // Un objet File d'attente représente une file d'attente dont les
     // caractères et dont la longueur maximale est la capacité.
private:
      char elems[capacity];
int front, rear, length;
     // La file d'attente est représentée par un tableau cyclique, avec
      // Eléments stockés soit en élévation [front..rear-1]  soit en
or in
     // elems [front..capacity-1] et elems [0..rear-1].
public:
      File ();
     // Construire une file d'attente vide.
      void add (char e);
      // Ajouter l'élément e à l'arrière de cette file d'attente.
```

```
                    char remove ();
                    // Retirer et rendre l'élément de tête de cette file d'attente.
                }.
```

3.4 Systèmes de type

Un *système de types* consiste en un mécanisme permettant de définir des types et de les associer à certaines constructions linguistiques, d'une part, et en un ensemble de règles pour l'*équivalence des types, la compatibilité des types* et l'*inférence des types, d*'autre part. Les constructions qui doivent avoir des types sont précisément celles qui ont des valeurs, ou qui peuvent se référer à des objets qui ont des valeurs. Ces constructions comprennent des constantes nommées, des variables, des champs d'enregistrement, des paramètres et parfois des sous-routines ; des constantes littérales (par exemple, 17, 3.14, "foo") ; et des expressions plus compliquées qui les contiennent. . Dans un langage avec des variables ou des paramètres polymorphes, il peut être important de distinguer le type d'une référence ou d'un pointeur et le type de l'objet auquel il se réfère : un prénom peut se référer à des objets de différents types à différents moments.

3.4.1 Polymorphisme

Le polymorphisme permet à un seul corps de code de fonctionner avec des objets de types multiples. Il peut ou non impliquer la nécessité d'une vérification du type d'exécution. Tel qu'il est implémenté dans Smalltalk, Lisp et les différents langages de script, le typage entièrement dynamique permet au programmeur d'appliquer des opérations arbitraires à des objets arbitraires. Ce n'est qu'au moment de l'exécution que l'implémentation du langage vérifie que les objets implémentent effectivement les opérations demandées. Comme les types d'objets peuvent être considérés comme des paramètres implicites (non spécifiés), on dit que le typage dynamique prend en charge le *polymorphisme paramétrique implicite*. Malheureusement, bien qu'il soit puissant et simple, le typage dynamique entraîne un coût d'exécution important et retarde la notification des erreurs.

Dans les langages orientés objet, le *polymorphisme des sous-types* permet à une variable *X* de type *T de se* référer à un objet de n'importe quel type dérivé de *T*. Comme les types dérivés sont nécessaires pour prendre en charge toutes les opérations du type de base, le compilateur peut être sûr que toute opération acceptable pour un objet de type *T* sera acceptable pour tout objet auquel se réfère *X*.

3.4.2 Classification des types

La plupart des langages proposent des types intégrés similaires à ceux pris en charge par la plupart des processeurs : nombres entiers, caractères, booléens et nombres réels (en virgule flottante). Les booléens (parfois appelés *logiques*) sont généralement

implémentés sous forme de quantités sur un octet, 1 représentant le vrai et 0 le faux. Le C est inhabituel par son absence de type booléen : alors que la plupart des langages s'attendent à une valeur booléenne, le C s'attend à un entier ; zéro signifie faux, et tout autre nombre signifie vrai.

Les caractères ont traditionnellement été implémentés en quantités d'un octet également, en utilisant généralement (mais pas toujours) le codage ASCII. Les langages plus récents (par exemple, Java et C#) utilisent une représentation sur deux octets conçue pour s'adapter (la partie la plus utilisée) au jeu de caractères *Unicode.*

3.4.2.1 Types numériques

Quelques langages (par exemple Fortran) font la distinction entre différentes longueurs d'entiers et de nombres réels ; la plupart ne le font pas et laissent le choix de la précision à l'implémentation. Quelques langages, dont C, C#et Modula-2, fournissent à la fois des entiers signés et non signés (Modula-2 appelle les *cardinaux entiers* non signés). Quelques langages (par exemple Common et Lisp Scheme) fournissent un type rationnel intégré, généralement implémenté comme une paire d'entiers qui représentent le numérateur et le dénominateur. Quelques langages (par exemple, Fortran, Common Lisp et Scheme) fournissent un type complexe intégré, généralement implémenté comme une paire de nombres à virgule flottante qui représentent les coordonnées cartésiennes réelles et imaginaires ; d'autres langages les supportent comme une classe de bibliothèque standard.

Plusieurs langages prennent en charge les types *décimaux* qui utilisent un encodage en base 10 pour éviter les anomalies d'arrondi dans l'arithmétique financière et humaine. Les nombres entiers, les booléens et les caractères sont tous des exemples de types *discrets* (également appelés types *ordinaux*).
La plupart des langages de script prennent en charge des entiers de précision arbitraire ; L'implémentation utilise plusieurs mots de mémoire le cas échéant.

3.4.2.2 Types d'énumération

Les énumérations facilitent la création de programmes lisibles et permettent au compilateur de détecter certains types d'erreurs de programmation. Un type d'énumération consiste en un ensemble d'éléments nommés. En Pascal, on peut écrire :

EXEMPLE : Énumérations en Pascal

```
type jour de la semaine = (soleil, lun, mar, mer, jeu, ven, sam) ;
```

Les valeurs d'un type de dénombrement sont ordonnées, de sorte que les comparaisons sont généralement valables (lun < mar), et il existe généralement un mécanisme permettant de déterminer le prédécesseur ou le successeur d'une valeur de dénombrement (en Pascal, demain := succ(aujourd'hui)).

3.4.2.3 Types composites

Les types *composites* sont généralement créés en appliquant un *constructeur de type à* un ou plusieurs types plus simples. Les types composites courants comprennent les enregistrements (structures), les enregistrements de variantes (unions), les tableaux, les ensembles, les pointeurs, les listes et les fichiers. Tous, à l'exception des pointeurs et des listes, sont facilement décrits en termes d'opérations mathématiques sur les ensembles (les pointeurs et les listes peuvent également être décrits mathématiquement, mais la description est moins intuitive).

Les enregistrements (structures) *s*'apparentent aux *tuples* mathématiques ; un type d'enregistrement correspond au produit cartésien des types de champs. Un enregistrement est constitué d'un ensemble de *champs*, chacun d'entre eux appartenant à un type plus simple (potentiellement différent). Les enregistrements de *variantes (unions)* diffèrent des enregistrements "normaux" en ce sens qu'un seul des champs (ou ensembles de champs) d'un enregistrement de variantes est valable à un moment donné. *Les ensembles*, comme les énumérations et les sous-ensembles, ont été introduits par Pascal. Un type d'ensemble est le jeu de puissances mathématiques de son type de base, qui doit souvent être discret. Une variable d'un type d'ensemble contient une collection d'éléments distincts du type de base.

Les tableaux sont les types composites les plus couramment utilisés. Un tableau peut être considéré comme une fonction qui fait correspondre les membres d'un type d'*index* aux membres d'un type d'*élément*. *Les tableaux* de caractères sont souvent appelés *chaînes de* caractères et sont souvent supportés par des opérations spéciales qui ne sont pas disponibles pour d'autres tableaux.

Les listes, comme les tableaux, contiennent une séquence d'éléments, mais il n'y a aucune notion de cartographie ou d'indexation. Une liste est plutôt définie de manière récursive comme une liste vide ou une paire composée d'un élément de tête et d'une référence à une sous-liste. Alors que la longueur d'un tableau doit être spécifiée au moment de l'élaboration dans la plupart des langages (mais pas toutes), les listes sont toujours de longueur variable. Les *fichiers* sont destinés à représenter des données sur des dispositifs de stockage de masse, en dehors de la mémoire dans laquelle résident d'autres objets de programme. Comme les tableaux, la plupart des fichiers peuvent être conceptualisés comme une fonction qui fait correspondre les membres d'un type d'index (généralement un nombre entier) aux membres d'un type de composant.

Les pointeurs sont des valeurs L. Une valeur de pointeur est une *référence* à un objet du type de base du pointeur. Les pointeurs sont souvent, mais pas toujours,

implémentés sous forme d'adresses. Ils sont le plus souvent utilisés pour implémenter des types de données *récursifs.* Un type *T* est récursif si un objet de type *T* peut contenir une ou plusieurs références à d'autres objets de type *T*.

3.4.2.4 Types subrange

Une *subrange* est un type dont les valeurs composent un sous-ensemble contigu des valeurs d'un type de *base* discret (également appelé type *parent).* En Pascal et dans la plupart de ses descendants, on peut déclarer des sous-gammes d'entiers, de caractères, d'énumérations, et même d'autres sous-gammes. En Pascal, les sous-gammes ressemblent à ceci :

EXEMPLE : Sous-gammes en Pascal

```
type test_score = 0..100;
workday = lundi.. vendredi;
```

Dans Ada, on écrirait

EXEMPLE : Sous-gammes en ADA

```
type test_score est un nouvel intervalle entier de 0..100;
subtype workday est le jour de la semaine du lundi au
vendredi ;;
```

La partie de la définition en ADA relative à la portée s'appelle une *contrainte de* type. Dans cet exemple, test_score est un type *dérivé,* incompatible avec les entiers. Le type workday, en revanche, est un *sous-type contraint* ; les jours ouvrables et les jours de la semaine peuvent être plus ou moins librement mélangés. La distinction entre les types dérivés et les sous-types est une caractéristique précieuse de ADA.

3.4.3 Orthogonalité

L'orthogonalité est tout aussi importante dans la conception des systèmes de types. Un langage très orthogonale tend à être plus facile à comprendre, à utiliser et à raisonner de manière formelle. Pour caractériser un énoncé qui est exécuté pour son ou ses effets secondaires, et qui n'a pas de valeurs utiles.

3.4.4 Vérification du type

La vérification du type est le processus qui consiste à s'assurer qu'un programme obéit au type de langage des règles de compatibilité. Une violation des règles est connue sous le nom de *conflit de type*. On dit qu'un langage est *fortement typée* si elle interdit, d'une manière que l'implémentation de le langage peut faire respecter, l'application de toute opération à tout objet qui n'est pas destiné à soutenir cette opération. Un langage est dit

statiquement typé s'il est fortement typé et si la vérification du type peut être effectuée au moment de la compilation. Au sens le plus strict du terme, peu de langages sont statiquement typées. En pratique, le terme est souvent appliqué aux langages dans lesquelles la plupart des vérifications de type peuvent être effectuées au moment de la compilation, et le reste au moment de l'exécution.

3.4.4.1 Compatibilité des types

La définition de la compatibilité des types varie beaucoup d'un langage à l'autre. ADA adopte une approche relativement restrictive : un ADA de type S est compatible avec un type T attendu si et seulement si : S et T sont équivalents, l'un est un sous-type de l'autre (ou les deux sont des sous-types du même type de base), Les deux sont des tableaux, avec le même nombre et le même type d'éléments dans chaque dimension.

- **Coercition**

Chaque fois qu'un langage permet d'utiliser une valeur d'un type dans un contexte qui attend d'autre part, l'implémentation linguistique doit effectuer une conversion automatique et implicite vers le type attendu. Cette conversion est appelée "*coercition de type*", une coercition peut nécessiter un code d'exécution pour effectuer une vérification sémantique dynamique, ou pour effectuer une conversion entre des représentations de bas niveau.

EXEMPLE : Coercition en C

```
short int s ;
non signé long int l ;
char c ; /* peut être signé ou non signé -- dépend de l'implémentation */
float
f ; /* généralement IEEE simple précision */
double d ; /* généralement IEEE double précision */
...
s = l ; /* les bits de poids faible de l sont interprétés comme un nombre
signé. */
longueur de s ;
dynamique. */
l = s ; /* s est le signe prolongé à la plus grande longueur, alors ses bits sont
interprétés comme un nombre non signé. */
s = c ; /* c est soit prolongé par un signe, soit prolongé par un zéro à la

le résultat est alors interprété comme un numéro signé. */
f = l ; /* l est converti en virgule flottante. Puisque f a moins de des bits
significatifs, une certaine précision peut être perdue. */
d = f ; /* f est converti au format long ; aucune précision n'est perdue. */
```

```
f = d ; /* d est converti au format plus court ; la précision peut être perdue. Si
la valeur de d ne peut être représentée avec une précision unique, la
Le résultat est indéfini, mais il ne s'agit PAS d'une erreur sémantique. */
```

3.4.4.2Équivalence de type

Il existe deux manières principales de définir l'équivalence de type. L'équivalence *structurelle* est basée sur le *contenu* des définitions de type : en gros, deux types sont identiques s'ils sont constitués des mêmes composants, assemblés de la même manière. L'équivalence de *nom* est basée sur *l'occurrence lexicale des définitions de type : en gros,* chaque définition introduit un nouveau type..

L'équivalence structurelle stipule que deux types sont équivalents si, après avoir remplacé tous les identificateurs de type par leurs définitions, on obtient la même structure. Cette définition est récursive, car les définitions des identificateurs de type peuvent elles-mêmes contenir des identificateurs de type. Elle est également vague, car elle laisse ouverte la question de savoir ce que signifie "même structure". Tout le monde s'accorde à dire que T1, T2 et T3 sont structurellement équivalents. Cependant, tout le monde n'est pas d'accord sur le fait que les enregistrements nécessitent des noms de champs identiques pour avoir la même structure, ou que les tableaux nécessitent des plages d'index identiques. Dans l'exemple ci-dessous, T4, T5 et T6 seraient considérés comme équivalents à T1 dans certaines langages mais pas dans d'autres :

```
type
    T4 = array[2..11] of real; -- même longueur
    T5 = array[2..10] of real; -- indice compatible de type
    T6 = array[blue .. red] of real; -- incompatible
    --      type d'index
```

Le test d'équivalence structurelle n'est pas toujours trivial, car des types récursifs sont possibles. Dans l'exemple ci-dessous, les types TA et TB sont structurellement équivalents, tout comme TC et TD, bien que leurs extensions soient infinies.

```
type
    TA = pointer to TA;
    TB = pointer to TB;
    TC=
    record
    Data : integer;
    Next : pointer to TC;
    end;
    TD =
```

```
Record
Data : integer;
Next : pointer to TD;
end;
```

L'équivalence de nom indique que deux variables sont du même type si elles sont déclarées avec le même nom de type, comme un nombre entier ou un type déclaré. Lorsqu'une variable est déclarée à l'aide d'un *constructeur de type* (c'est-à-dire une expression qui donne un type), son type reçoit un nouveau nom interne pour des raisons d'équivalence de nom. Les constructeurs de type comprennent les mots *tableau*, *enregistrement* et *pointeur vers*. Par conséquent, l'équivalence de type indique que T1 et T3 ci-dessus sont différents, tout comme TA et TB. Il y a différentes interprétations possibles lorsque plusieurs variables sont déclarées à l'aide d'un seul constructeur de type, comme T1 et T2 ci-dessus. ADA est assez strict ; il appelle T1 et T2 différents. La norme actuelle pour le Pascal est plus indulgente ; elle appelle T1 et T2 identiques. Cette forme d'équivalence de nom est également appelée *équivalence de déclaration*.

3.4.4.3 Type Inférence

Le résultat d'un opérateur arithmétique a généralement le même type que les opérandes. Le résultat d'une comparaison est généralement booléen. Le résultat d'un appel de fonction a le type déclaré dans l'en-tête de la fonction. Le résultat d'une affectation (dans les langages dans lesquelles les affectations sont des expressions) a le même type que le côté gauche.

- **Types Composite**

L'inférence de type devient un problème lorsqu'une opération sur des composites donne un résultat d'un type différent de celui des opérandes. Les chaînes de caractères fournissent un exemple simple. En Pascal, la chaîne littérale "abc" contient un tableau de types [1..3] de caractères. En ADA, la chaîne analogue (dénommée "abc") est considérée comme ayant un type incomplètement spécifié qui est compatible avec tout tableau de trois éléments de caractères. Les opérations sur les valeurs composites se produisent également lors de la manipulation des ensembles. Modula et Pascal, par exemple, supportent l'union (+), l'intersection (*) et la différence (-) sur des ensembles de valeurs discrètes. On dit que les opérandes d'ensembles ont des types compatibles si leurs éléments ont le même type de base T.

❖ **Sous-gammes**

Pour les opérateurs arithmétiques simples, la subtilité du système de type principal apparaît lorsqu'un ou plusieurs opérandes ont des types de sous-gamme (ce qu'ADA appelle des sous-types avec des contraintes de gamme).

EXEMPLE : Inférence des types de sous-gammes

```
type Atype = 0..20 ;
Btype = 10..20 ;
var a : Atype ;
b : Btype ;
```

Quel est le type de a + b ? Il ne s'agit certainement ni de Atype ni de Btype, puisque les valeurs possibles vont de 10 à 40. On pourrait imaginer qu'il s'agit d'un nouveau type de sous-gamme anonyme avec 10 et 40 comme limites. La réponse habituelle en Pascal et ses descendants est de dire que le résultat de toute opération arithmétique sur une sous-gamme a le type de base de la sous-gamme, dans ce cas un entier.

Si le résultat d'une opération arithmétique est affecté à une variable d'un type de sous-gamme, une vérification sémantique dynamique peut être nécessaire. Pour éviter les dépenses liées à des contrôles inutiles, un compilateur peut garder une trace, au moment de la compilation, des valeurs les plus grandes et les plus petites possibles de chaque expression, en calculant essentiellement la valeur anonyme de 10... 40 anonyme.

3.5 Flux de Contrôle

Le flux de contrôle dans les langages d'assemblage est réalisé au moyen de sauts conditionnels et inconditionnels. Les premières versions de Fortran imitaient l'approche de bas niveau en s'appuyant fortement sur les déclarations goto pour la plupart des flux de contrôles non procéduraux:

```
if (A .lt. B) goto 10 ! ".lt." means "<"
...
10
```

Le chiffre 10 sur la ligne du bas est une *étiquette de déclaration*. Les déclarations Goto ont également occupé une place importante dans d'autres langages à caractère impératif.

3.5.1 Flux structuré et non structuré

Les concepteurs de langages ont débattu avec passion des mérites et des inconvénients des gotos. Il semble juste de dire que les détracteurs ont gagné. ADAet C# n'autorisent les gotos que dans des contextes limités. Modula, Java et la plupart des

langages de script ne les autorisent pas du tout. C++ et Fortranles autorisent principalement pour des raisons de compatibilité avec leurs prédécesseurs. L'abandon des gotos faisait partie d'une "révolution" plus vaste dans le domaine de l'ingénierie logicielle, connue sous le nom de *programmation structurée*.

3.5.2 Évaluation de l'expression

Une expression consiste généralement soit en un simple objet (par exemple, une constante littérale, ou une variable ou constante nommée), soit en un *opérateur ou une* fonction appliquée à un ensemble d'opérandes ou d'arguments, chacun étant à son tour une expression. Il est classique d'utiliser le terme opérateur pour < les fonctions intégrées qui utilisent une syntaxe spéciale et simple, et d'utiliser le terme *opérande* pour un argument d'un opérateur. Dans la plupart des langages impératifs, les appels de fonction consistent en un nom de fonction suivi d'une liste d'arguments entre parenthèses et séparés par des virgules, comme dans

```
my_func(A, B, C)
```

Les opérateurs sont généralement plus simples, ne prenant qu'un ou deux arguments, et se passant des parenthèses et des virgules :

```
a + b

- c
```

Certains langages définit leurs opérateurs comme du sucre syntaxique pour des fonctions plus "normales". En ADA, par exemple :

```
a + b est l'abréviation de "+"(a, b) ;  en C++, a + b est l'abréviation de a.operator+(b)
```

En général, un langage peut spécifier que les appels de fonction (invocations d'opérateurs) utilisent la notation prefix, infix ou postfix. Ces termes indiquent, respectivement, si le nom de la fonction apparaît avant, parmi ou après ses différents arguments :

```
prefix:    op a b    or  op (a,  b)   or   (op a b)

infix:     a  op b

postfix:   a b  op
```

3.5.3 Séquencement

Le séquencement est au cœur de la programmation impérative. C'est le principal moyen de contrôler l'ordre dans lequel les effets secondaires (par exemple, les affectations) se produisent : lorsqu'une instruction en suit une autre dans le texte du programme, la première instruction s'exécute avant la seconde. Dans la plupart des langages impératifs, les listes d'instructions peuvent être accompagnées de début... fin ou {... } et être ensuite utilisées dans tout contexte où une seule déclaration est attendue. Une telle liste délimitée est généralement appelée déclaration composée.

La figure 3.3 montre quatre organigrammes : une commande simple, une sous-commande séquentielle, une commande si et une commande en cours. Chacun de ces organigrammes a une entrée et une sortie unique. Il en va de même pour les autres commandes conditionnelles, telles que les commandes de cas, et les autres commandes itératives, telles que les commandes for. Il s'ensuit que *toute* commande formée en composant des commandes simples, séquentielles, conditionnelles et itératives a un flux de contrôle à entrée unique et sortie unique. Il est parfois nécessaire d'implémenter des flux de contrôle plus généraux. En particulier, des flux de contrôle à entrée unique et à sorties multiples sont souvent souhaitables.

Un *séquenceur* est une construction qui transfère le contrôle à un autre point du programme, appelé *destination du* séquenceur. Les séquenceurs permettent de mettre en œuvre divers flux de contrôle, avec des entrées multiples et/ou des sorties multiples.
La simple existence de séquenceurs dans un langage de programmation affecte radicalement sa sémantique. Par exemple, nous avons jusqu'à présent affirmé que la commande séquentielle "C1 ; C2" est exécutée en exécutant d'abord C1 puis C2. Cela n'est vrai que si C1 se *termine normalement*. Mais si *C1 exécute* un séquenceur, C1 se *termine brusquement,* ce qui peut entraîner le saut de C2 (selon la destination du séquenceur).

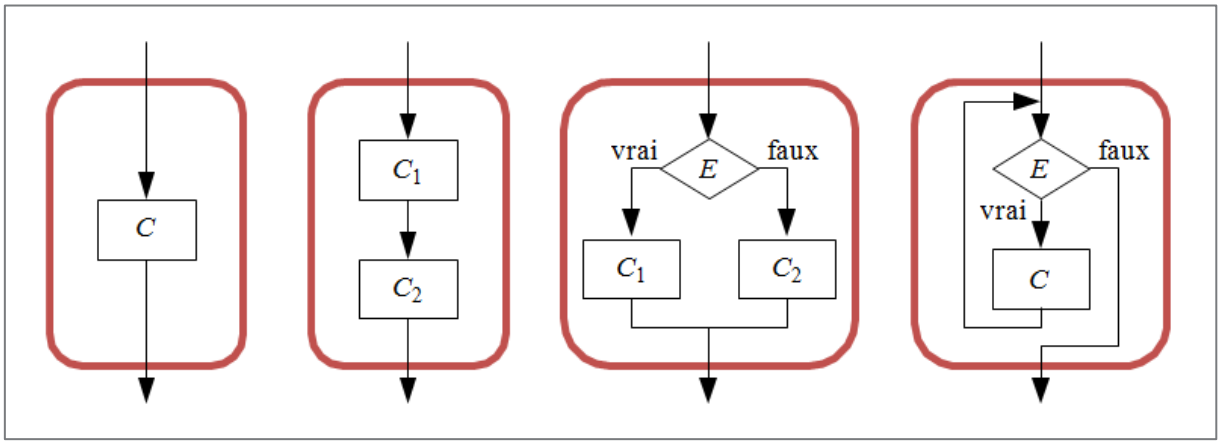

Figure 3.3 : *Organigrammes des commandes primitives, séquentielles, si et pendant.*

Certains types de séquenceurs sont capables de *véhiculer des* valeurs. Ces valeurs sont calculées à l'endroit où le séquenceur est exécuté, et sont disponibles pour être utilisées à la destination du séquenceur.

3.5.4 Sélection

Les énoncés de sélection dans la plupart des langages impératifs utilisent une variante de la notation **if**... **then** ... **else** introduite dans l'Algol 60 :

```
si condition, alors déclaration

sinon si condition, alors déclaration

sinon si condition, alors déclaration
```

Les langages diffèrent dans les détails de la syntaxe. En Pascal, la clause then et la clause else sont toutes deux defined pour contenir une seule déclaration (il peut bien sûr s'agir d'une déclaration composée de début... fin).

3.5.5 Sauts (*jump)*

Un *saut (jump)* est un séquenceur qui transfère le contrôle à un point de programme spécifique. Un saut a généralement la forme " ***goto*** *L* ; ", et cela transfère le contrôle directement au point de programme désigné par *L,* qui est une *étiquette*.
EXEMPLE : Saut en C:

```
si (E1) C1

else {
C2
goto X ;
}
C3
tand que (E2) { C4

X : C5
}
```

Ici, l'étiquette X indique un point de programme particulier, à savoir le début de la commande C5. Ainsi, le saut "**goto** X;" transfère le contrôle au début de C5. La figure 3.4 montre l'organigramme correspondant à ce fragment de programme. Notez que la commande si et la commande tand que ont des sous-chiffres identifiables, qui sont mis

en évidence. Le fragment de programme dans son ensemble a une seule entrée et une seule sortie, mais la commande si a deux sorties, et la commande while a deux entrées.

Les sauts non restreints permettent à toute commande d'avoir plusieurs entrées et plusieurs sorties. Ils ont tendance à donner lieu à un code "spaghetti", appelé ainsi parce que son organigramme est enchevêtré.

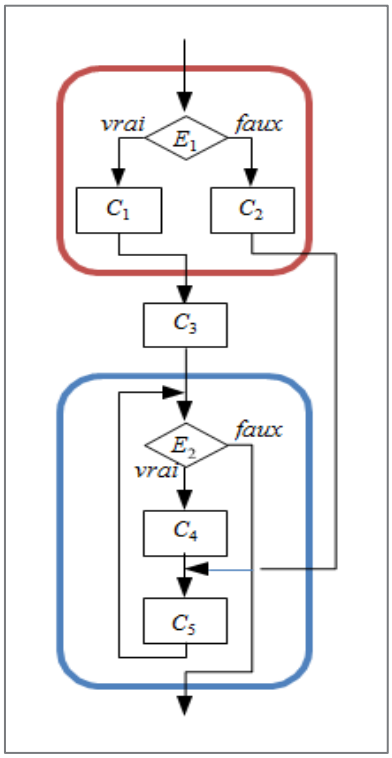

Figure 3.4 : *Organigramme d'un fragment de programme avec un saut*

Même dans les langages de programmation qui prennent en charge les sauts, leur utilisation est limitée par le simple fait de restreindre la portée de chaque étiquette : le saut "'**goto** *L* ;" n'est légal que dans la limite de la portée de *L*. En C, par exemple, la portée de chaque étiquette est la plus petite commande de bloc englobant ("'{ . . . }"). Il est donc possible de sauter à l'intérieur d'une commande de bloc, ou d'une commande de bloc à une commande de bloc englobant commande de bloc ; mais il *n*'est *pas* possible de passer à une commande de bloc de l'extérieur, ni d'un corps de fonction à un autre.

3.5.6 Itération

L'itération est un mécanisme qui permet à un ordinateur d'effectuer des opérations similaires de manière répétée. Sans au moins un de ces mécanismes, la durée d'exécution d'un programme (et donc la quantité de travail qu'il peut effectuer et la quantité d'espace qu'il peut utiliser) serait une fonction linéaire de la taille du texte du

programme. Dans la plupart des langages, l'itération prend la forme de *boucles*. Comme les énoncés dans une séquence, les itérations d'une boucle sont généralement exécutées pour leurs effets secondaires et leur modifications de variables.
Les boucles se présentent sous deux formes principales, qui diffèrent par les mécanismes utilisés pour déterminer combien de fois il faut itérer. Une boucle à *commande par énumération* est exécutée une fois pour chaque valeur d'un ensemble fini donné ; le nombre d'itérations est connu avant le premier début de l'itération. Une boucle à commande *logique* est exécutée jusqu'à ce qu'une condition booléenne (qui doit généralement dépendre des valeurs modifiées dans la boucle) change de valeur.

3.5.7 Échappement

Une *échappement* est un séquenceur qui met fin à l'exécution d'une commande ou d'une procédure contenant du texte. En termes d'organigramme, la destination d'une fuite est toujours le point de sortie d'un sous-organigramme englobant. Avec les échappements, nous pouvons programmer des flux de contrôle à entrée unique et à sorties multiples.

Le *séquenceur de sortie de* ADA met fin à une boucle de clôture, qui peut être une boucle de commande, une boucle de commande forcée ou une boucle de base ("**boucle de fin** *C* ; " rovoque simplement l'exécution répétée du corps de boucle *C)*. Un séquenceur de sortie à l'intérieur d'un corps de boucle termine la boucle.

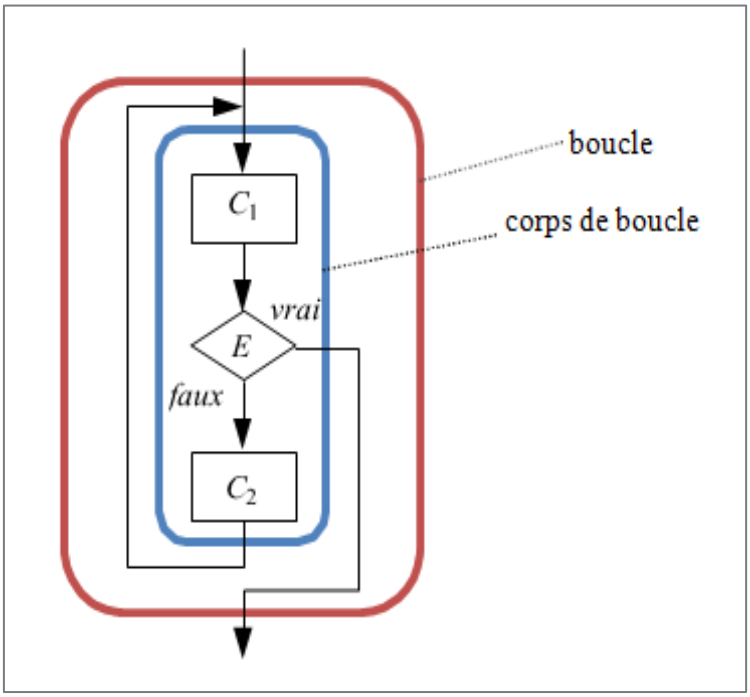

Figure 3.5 : *Organigramme d'une boucle avec une échappée.*

Dans la boucle de base suivante, il y a un seul séquenceur de sortie (conditionnel) :

```
loop
  C1
  exit when E;
  C2
end loop;
```

L'organigramme correspondant est présenté à la figure 3.5, dans lequel le corps de la boucle et la boucle de base elle-même sont mis en évidence. Notez que la boucle a une seule sortie, mais que le corps de la boucle a deux sorties. La sortie normale du corps de la boucle suit l'exécution de C2, après quoi le corps de la boucle est répété. L'autre sortie du corps de la boucle est provoquée par le séquenceur de sortie, qui met immédiatement fin à la boucle. Il est également possible pour un séquenceur de sortie de terminer une boucle *extérieure.* Un séquenceur de sortie ADA peut se référer à n'importe quelle boucle d'encerclement nommée. L'exemple suivant illustre cette possibilité.

EXEMPLE : Séquenceur de sortie en ADA

Le type suivant représente un agenda pour une année complète :

```
type Year_Diary is array (Month_Number, Day_Number)
    of Diary_Entry;
```

Considérez la fonction suivante :

```
function search (diary: Year_Diary;
                 this_year: Year_Number;
                 key_word: String)
return Date is
-- Recherchez dans le journal la première date dont l'entrée
 correspond à mot-clé.
-- Renvoie la date correspondante, ou le 1er janvier s'il n'y a pas de
 correspondance.
match_date: Date := (this_year, 1, 1);

begin
   search:
(1)    for m in Month_Number loop
(2)          for d in Day_Number loop
                 if matches(diary(m,d), key_word) then
                     match_date := (this_year, m, d);
```

```
(3)                        exit search;
                         end if;
                      end loop;
                   end loop;
             return match_date;
          end;
```

Ici, la boucle extérieure commençant à (1) est appelée recherche. Ainsi, le séquenceur "recherche de **sortie" en** (3) termine cette boucle extérieure. En revanche, une simple "**sortie" en** (3) ne terminerait que la boucle intérieure (2).

Le ***séquenceur de retour*** est un type d'évasion particulièrement important**. Le séquenceur de retour dans** le corps d'une fonction doit également porter le résultat de la fonction. Les séquenceurs de retour sont supportés par JAVA, C, C++, et ADA.

3.5.8 Exceptions

Une *exception* est une entité qui représente une situation anormale (ou une famille de situations anormales). Tout code qui détecte une situation anormale peut *lancer* (ou *lever*) une exception appropriée. Cette exception peut ensuite être *reprise* dans une autre partie du programme, où un construct appelé *gestionnaire d'exception* (ou simplement *gestionnaire) se* remet de la situation anormale. Chaque exception peut être détectée et traitée, et le programmeur a un contrôle total sur l'endroit et la manière dont chaque exception est traitée. Les exceptions ne peuvent pas être ignorées : si une exception est lancée, le programme s'arrêtera à moins qu'il ne capture l'exception.
Notez les propriétés importantes suivantes des exceptions :

- Si une sous-commande lance une exception, la commande d'accompagnement lance également cette exception, à moins qu'il ne s'agisse d'une commande de traitement des exceptions capable de prendre en compte cette exception particulière. Si le corps d'une procédure lance une exception, l'appel de procédure correspondant lance également cette exception.
- La commande qui lance une exception est interrompue brusquement (et ne sera jamais reprise).
- Certaines exceptions sont intégrées, et peuvent être lancées par des opérations intégrées. Il s'agit par exemple du débordement arithmétique et de l'indexation de tableaux hors gamme.
- D'autres exceptions peuvent être déclarées par le programmeur, et peuvent être lancées explicitement lorsque le programme lui-même détecte une situation anormale.

Les langages orientés objet traitent les exceptions comme des objets. JAVA, par exemple, possède une classe d'exception intégrée, et chaque exception est un objet d'une

sous-classe d'exception. Chaque sous-classe d'Exception représente une situation anormale différente. Un objet d'exception contient un message explicatif (et éventuellement d'autres valeurs), qui sera transmis au gestionnaire. Étant des valeurs de première classe, les exceptions peuvent être stockées et transmises en tant que paramètres, mais aussi être lancées et capturées. Le séquenceur JAVA "**throw** *E* ; " lance l'exception produite par l'expression *E*.

La commande de traitement des exceptions JAVA a la forme :

```
try
  C0
catch (T1 I1) C1
. . .
catch (Tn In) Cn
finally Cf
```

Il est capable de saisir toute exception de la classe *T1* ou ... ou *Tn*. Si la sous-commande *C0* lance une exception de type *Ti*, alors le gestionnaire d'exception *Ci* est exécuté avec l'identifiant *Ii* lié à cette exception. Juste avant que la commande de traitement des exceptions ne se termine (que ce soit normalement ou brusquement), la sous-commande *Cf* est exécutée sans condition. (La *dernière* clause est facultative).

Une caractéristique importante de JAVA est que chaque méthode doit spécifier les exceptions qu'elle peut lancer, le titre d'une méthode peut inclure une *spécification* d'*exception de la* forme " *lance T1, ... Tn*", où *T1, ... T1, ..., Tn* sont les classes d'exceptions qui peuvent être lancées. En l'absence d'une spécification d'exception, la méthode est supposée ne jamais lancer d'exception.

EXEMPLE : Les exceptions en JAVA

Considérez un programme JAVA qui consiste à lire et à traiter les données relatives aux précipitations pour chaque mois de l'année.

La méthode suivante tente de lire un nombre littéral :

```
static float readFloat (BufferedReader input)
      throws IOException, NumberFormatException {
   . . . // sautent les caractères d'espace
   if (. . .) // fin de la saisie atteinte
(1)  throw new IOException("end of input");
      String literal = . . .; // lire les caractères non spatiaux
(2) float f = Float.parseFloat(literal);
     return f;
}
```

Cette méthode précise qu'elle peut lancer une exception de classe IOException ou NumberFormatException (mais aucune autre). Si la fin de l'entrée est atteinte (de sorte qu'aucun littéral ne peut être lu), le séquenceur à (1) construit une exception de classe IOException (contenant un message explicatif) et lance cette exception. Si un littéral est lu mais qu'il devient pour ne pas être un littéral numérique bien formé, la méthode de bibliothèque Float.parseFloat appelée à (2) construit une exception de la classe NumberFormatException (contenant le littéral mal formé) et lance cette exception.

3.6 Concurrence

La concurrence n'est pas une idée nouvelle. L'intérêt généralisé pour la concurrence est un phénomène relativement récent, cependant ; il provient en partie de la disponibilité de machines multicœurs et multiprocesseurs à faible coût, et en partie de la prolifération des applications graphiques, multimédia et web, qui sont toutes naturellement représentées par des fils de contrôle concurrents.

3.6.1 Programmes et processus

Un *processus séquentiel* est un ensemble d'étapes totalement ordonnées, chaque étape étant un *changement d'état* dans un composant quelconque d'un système informatique. Leur ordre total est définés dans le sens où, étant donné deux étapes quelconques, il est toujours possible de dire laquelle est la plus précoce. Un *programme séquentiel* spécifie les changements d'état possibles d'un processus séquentiel, qui ont lieu dans un ordre déterminé par les structures de contrôle du programme. Un *programme concurrent* spécifie les changements d'état possibles de deux ou plusieurs processus séquentiels. Aucun ordre n'est naturellement défini entre les changements d'état de l'un de ces processus et les changements d'état de l'un des autres. Par conséquent, on dit qu'ils s'exécutent *simultanément*, et ils peuvent même s'exécuter *simultanément*.

3.6.2 Les primitives de la concurrence

Un processus conventionnel, *lourd*, est l'exécution d'un programme. Pour soutenir cette exécution, un système d'exploitation fournit normalement un espace d'adressage, une allocation de mémoire principale et une part du temps CPU, ainsi qu'un accès à files, aux périphériques d'entrée/sortie, aux réseaux, etc. Cela représente des frais généraux importants, de sorte que la création d'un processus nécessite une quantité non négligeable de ressources de l'ordinateur, et que *le passage* d'un processus à l'autre prend un temps de significant. L'implémentation d'un système concurrent au moyen de nombreux processus entraîne donc une pénalité : une mauvaise exécution efficiency.

Un *fil* est une alternative légère. Comme un processus lourd, un fil est un flow de contrôle par un programme, mais il ne possède pas de ressources de calcul indépendantes. Au contraire, un fil existe au sein d'un processus et dépend des ressources du processus. Pour faire passer l'ordinateur d'un fil à l'autre, il suffit d'échanger le contenu de ses registres de travail. Le reste de l'environnement d'exécution reste le même, de sorte que le changement de contexte de fil à fil peut être très rapide. (Le concept de thread est né dans le monde en temps réel, où le terme de ***tâche est*** plus souvent utilisé. Fidèle à ses origines, ADA utilise la terminologie "tâche" tand que JAVA va de pair avec "thread").

3.6.2.2 Interrompt

La fin d'une opération d'entrée/sortie simultanée est une condition relativement peu fréquente à laquelle l'unité centrale devrait réagir rapidement. Normalement, il ne serait pas possible de le vérifier à plusieurs reprises sur un site efficace. Ainsi, lorsque des transferts d'entrée/sortie parallèles ont été introduits, la fin de chaque opération d'entrée/sortie a été faite pour provoquer une *interruption*. C'est-à-dire qu'un signal provenant du matériel d'entrée/sortie force un appel asynchrone à une routine qui s'occupera de la nouvelle situation. Une interruption est donc, en fait, un appel de procédure invisible inséré à un point aléatoire du programme !

3.6.2.3 Événements

Un *événement* est une entité qui représente une catégorie de changements d'état. Nous pouvons considérer les événements *e* comme des valeurs d'un type abstrait qui est équipé des opérations **event_wait(*e*)** et **event_signal(*e*)**, où :
event_wait(*e*) bloque *toujours,* et ne débloque que lorsque la prochaine occurrence signalée d'un événement de catégorie *e se* produit ; **event_signal(*e*)** débloque *tous les* processus qui attendent *e*.
Les opérations event_wait et event_signal fournissent des implémentations des primitives de transmission et de réception, où nous faisons en sorte qu'un événement unique représente chaque condition.

3.6.2.4 Messages

Le passage des messages nécessite un *canal* capable de transmettre un message d'un processus à l'autre. Un tel canal peut être explicitement identifié, ou implicite par les identités des processus émetteur et récepteur. Un canal explicitement identifié peut être connu comme une *file d'attente de messages*, un

tampon ou autre ; un tel canal peut permettre la communication entre un nombre arbitraire d'expéditeurs et de récepteurs. Un canal identifié implicitement ne peut prendre en charge que la communication individuelle. La question se pose également de savoir si le canal prend en charge la communication dans un seul sens (*simplex*) ou dans les deux sens (*duplex*).

3.6.2.5 Appels de procédure à distance

En exploitant l'*appel de procédure à distance*. L'environnement d'exécution détermine le site où se trouve une procédure et communique avec ce site pour l'invoquer.

Le site qui fournit la procédure, à la réception d'un appel à distance, peut créer un processus pour mettre en œuvre l'opération. Un processus serveur sur le site distant peut également recevoir tous les appels pour une procédure et fournir ce service à chaque appelant à tour de rôle. Si le site bénéfique est plus conciliant, le serveur peut bifurquer des fils de discussion pour desservir des appelants simultanés. Le choix est déterminé par les coûts relatifs de la création du processus ou du fil par rapport à la communication, et par le degré de simultanéité souhaité. Il s'agit là de questions pragmatiques plutôt que de questions de principe.

3.6.3 Abstractions de contrôle simultanées

Les concepteurs de langages de programmation ont largement réussi à créer des abstractions qui facilitent une programmation séquentielle de cette qualité. L'attention s'est de plus en plus portée sur les problèmes soulevés par la concurrence et la distribution.

3.6.3.3 Régions critiques conditionnelles

La *région critique conditionnelle* est une commande composite qui assure à la fois l'exclusion mutuelle et la communication. L'idée clé est que chaque variable partagée entre les processus doit être déclarée comme telle. Une variable non déclarée comme telle est locale à un processus, et un compilateur peut facilement vérifier qu'aucun autre processus n'a accès à une telle variable. D'un seul coup, cela permet d'éliminer l'une des principales sources d'erreur dans la programmation concurrente.

Dans la région critique conditionnelle, la *commande await* "await *E*'' *se* bloque jusqu'à ce que l'expression *E* (qui accède à la variable partagée *v*) donne la valeur vraie. Pendant l'attente, un processus renonce à son utilisation exclusive de la variable partagée. Lorsqu'il reprend, son exclusivité est réacquise comme nous allons voir dans la section 4.4.7.

3.7 Conclusion

Dans ce chapitre, nous avons étudié les concepts de types abstraits et de classes, qui sont des unités de programme bien adaptées pour être les éléments constitutifs de grands programmes, ensuite nous avons étudié les unités génériques, et vu comment elles peuvent être instanciées pour générer des unités de programme ordinaire, puis nous avons étudié le polymorphisme, qui permet à un sous-type ou une sous-classe d'hériter des opérations de son type ou de sa classe parent, après nous avons étudié les conversions de types et les sauts(jumps), enfin nous avons étudié les exceptions, qui sont une technique efficace pour gérer de manière robuste les situations anormales et nous avons vu comment la concurrence conduit à des problèmes de déterminisme qui n'existent pas dans les programmes séquentiels.

3.8 Exercices

1. Concevez une unité générique qui met en œuvre des ensembles (sets). Un ensemble est une collection non ordonnée de valeurs dans laquelle aucune valeur n'est dupliquée. Paramétrez votre unité générique par rapport au type de valeurs de l'ensemble. Fournissez une opération qui construit un ensemble vide, une opération qui ajoute une valeur donnée à un ensemble, une opération qui unit deux ensembles, et une opération qui teste si une valeur donnée est contenue dans un ensemble. Implémentez votre unité générique en, C++ ou JAVA.
2. JAVA, C et C++ disposent d'un séquenceur continue, qui met fin à l'itération en cours de la boucle d'enclenchement, provoquant ainsi le démarrage immédiat de la prochaine itération. Montrer que le séquenceur de poursuite peut être compris en termes d'échappement tout usage.
3. Considérons le fragment de code suivant (en JAVA, C ou C++) :

```
p = 1 ; m = n ; a = b ;
tant que (m > 0) {

si (m % 2 != 0) p *= a ; m /= 2 ;
a *= a ;
}
```

3.1 Réécrire le fragment de code entièrement en termes de sauts conditionnels et inconditionnels. Expliquez pourquoi cela rend le fragment de code plus difficile à lire.

3.2 Dessinez l'organigramme correspondant. Tracez les sous-diagrammes correspondant à la commande en cours, à la commande si et à la commande séquentielle entre crochets..

4. Le programmeur, inquiet de l'inefficacité des opérations de sémaphore, tente d'initialiser ces variables en toute sécurité, *sans exclusion mutuelle,* comme suit :

```
procedure initialiser (v : in out T) is
begin
if not v. initialisé then
. . . ; -- donner v sa valeur initiale
v.initialisé : = vrai ;
end if;
end initialiser;
```

4.1 Expliquez la condition de concurrence qui l'empêche de fonctionner de manière fiable.

4.2 Reconnaissant l'erreur, le programmeur fait quelques recherches et apprend le *verrouillage à double vérification,* une technique qui n'utilise l'exclusion mutuelle que lorsque c'est nécessaire, et réécrit l'initialisation comme suit :

```
sema_initialize(mutex, 1);
. . .
procedure initialize (v : in out T) is
begin
if not v.initialized then
sema_wait(mutex);
if not v.initialized then
. . . ; -- give v its initial value
v.initialized := true;
end if;
sema_signal(mutex);
end if;
end initialize;
```

- Expliquez le raisonnement du programmeur en pensant que cela résout efficacement le problème de l'exclusion mutuelle.

CHAPITRE 4

LES PARADIGMES

4.1 Introduction

Les paradigmes de programmation sont basés sur des sélections différentes de concepts fondamentaux et donnent lieu à des styles de langage et de programmation très contrastés. Dans ce chapitre, nous allons identifier les concepts fondamentaux de ces différents paradigmes (le paradigme impératif, le paradigme orienté objet, le paradigme concurrent, le paradigme fonctionnel, le paradigme logique et le scrip),en montrant comment les concepts ont été sélectionnés et combinés lors de la conception du langage.

4.2 La programmation impérative

4.2.1 Concepts fondamentaux

Les variables et les *commandes* sont des concepts clés de la programmation impérative pour la raison suivante. De nombreux programmes sont écrits pour modéliser des processus du monde réel affectant des entités du monde réel, et une entité du monde réel possède souvent un état qui varie avec le temps. Ainsi, les entités du monde réel peuvent être modélisées naturellement par des variables, et les processus du monde réel par des commandes qui inspectent et mettent à jour ces variables.

Les procédures sont un concept clé de la programmation impérative car elles sont abstraites par rapport aux commandes. Nous pouvons distinguer le comportement observable d'une procédure et l'algorithme (commandes) par lequel la procédure réalise son comportement, une séparation utile des préoccupations entre les utilisateurs de la procédure et son exécutant.

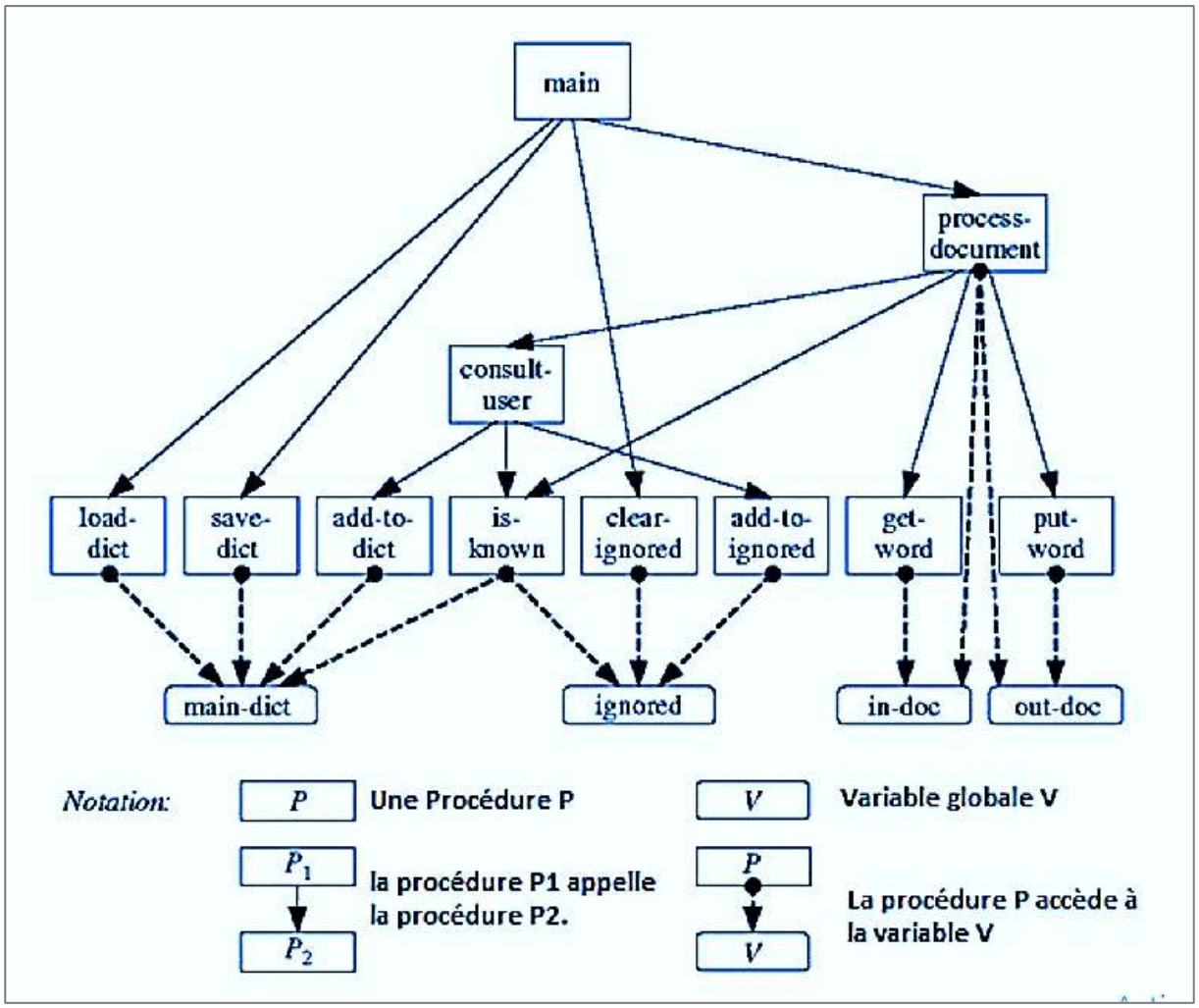

Figure 4.1 : *Architecture d'un programme impératif avec des variables globales.*

Un programme impératif traditionnel est constitué de procédures et de variables globales. La figure 4.1 montre l'architecture d'un tel programme, qui se compose de plusieurs procédures et de plusieurs variables globales auxquelles ces procédures accèdent. Les procédures s'appellent les unes les autres, mais cela ne crée qu'un couplage lâche : un modification à l'implémentation d'une procédure n'affectera probablement pas ses appelants. Cependant, l'accès aux variables globales crée un couplage serré : tout modification à la représentation d'une variable globale est susceptible de forcer modifications à toutes les procédures qui y accèdent ; de plus, chaque procédure est sensible à la façon dont la variable globale est initialisée, inspectée et mise à jour par d'autres procédures.

La figure 4.2 illustre l'architecture d'un programme impératif. Il n'y a pas des variables globales, mais seulement des variables et des paramètres locaux (non représentés). Deux types abstraits ont été conçus, chacun étant équipé d'opérations suffisant pour cette application. Maintenant, toutes les unités

de programme sont couplées entre elles de manière souple. Le programmeur de maintenance pourra comprendre chaque unité individuellement, et pourra la modifier sans craindre une cascade de modifications vers d'autres unités. En particulier, puisque la représentation de chaque type d'abstraction est privée, le programmeur de maintenance peut modifier la représentation sans craindre une cascade de modifications vers d'autres unités.

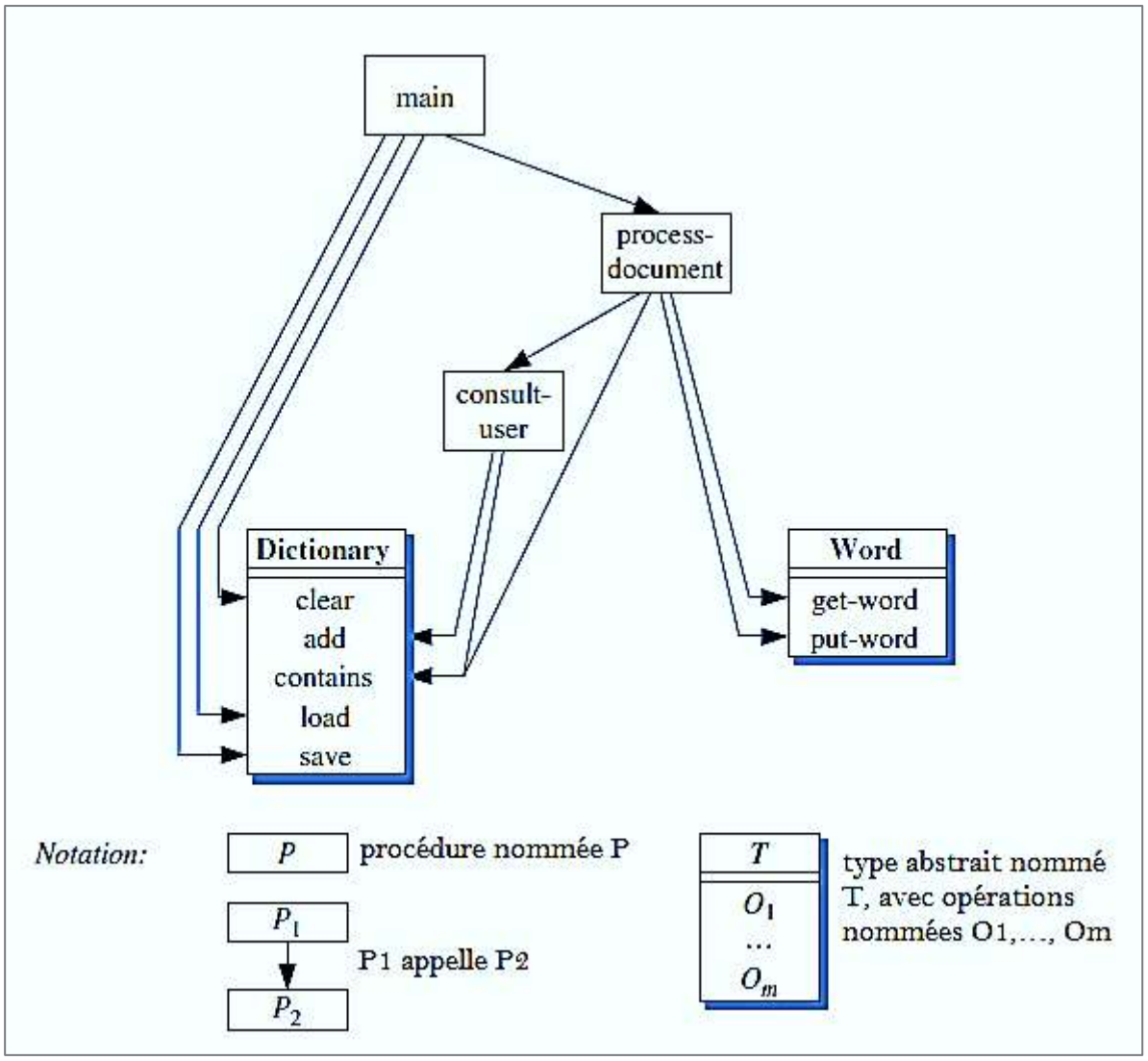

Figure 4.2 : *Architecture d'un programme impératif avec abstraction de données.*

4.2.2 Langage C

C se caractérise par la présence d'opérations de bas niveau et de faibles contrôles de type. Il manque également de concepts modernes tels que l'abstraction de données, l'abstraction générique ou les exceptions. Néanmoins, les programmeurs experts en C peuvent l'utiliser pour écrire des logiciels système très efficace.

4.2.2.1 Variables, stockage et contrôle

C prend en charge les variables globales et locales. Il prend également en charge les variables de tas à un niveau inférieur. Son allocateur est une fonction malloc dont l'argument est la *taille* de la variable de tas à allouer :

```
IntNode* ptr;
ptr = malloc(sizeof IntNode);
```

L'expression " **sizeof** *T*" donne la taille (en octets) d'une valeur de type *T*.
C présente certaines des caractéristiques d'un langage d'expression. Comme nous l'avons vu, une tâche est une expression. Un appel de fonction est également une expression (même si le type de résultat est **nul**). Le fait que les assignations en C sont des expressions, et que toutes les procédures en C sont des fonctions, oblige en fait les programmeurs à écrire des expressions avec des effets secondaires. Les programmeurs C doivent faire preuve d'une grande autodiscipline pour éviter d'écrire du code illisible.

4.2.2.2 Valeurs et types

C dispose d'un répertoire limité de types primitifs : types d'énumération, types d'entiers (avec diverses plages, signées ou non), et types de points flottants (avec une précision simple ou double). Il n'y a pas de type booléen spécifique : *faux* et *vrai* sont conventionnellement représentés par des entiers nuls et non nuls. Il n'y a pas non plus de type de caractère spécifique: les caractères sont conventionnellement représentés par des valeurs de type *char* mais, malgré le nom du type, ces valeurs ne sont que de petits entiers.
C ne prend pas directement en charge les types récursifs. Les programmeurs doivent plutôt déclarer les types récursifs à l'aide de pointeurs.

EXEMPL : Liste liée en C

Examinant les définitions suivantes du type C :

```
struct IntNode;
typedef IntNode* IntPtr;
struct IntNode {
        int elem;
        IntPtr succ;
};
```

Les valeurs de type IntPtr sont des pointeurs vers les nœuds d'une liste liée. Chaque nœud contiendra un élément entier et un pointeur succédant au nœud suivant, sauf le dernier nœud, qui contiendra un pointeur nul (représenté conventionnellement par 0).
Une affectation C "*V* = *E*" est en fait une expression, qui donne la valeur de *E et* affecte cette valeur à la variable *V*. C permet également de combiner des opérateurs binaires ordinaires avec l'assignation : si ⊗ est un opérateur binaire, alors "V⊗=E" est une

alternative concise à "$V = V \otimes E$''. Les opérateurs pré et post-incrémentation de C sont encore plus concis, tous deux nommés "++" : si V est un entier ou une variable de pointeur, "++V" incrémente V *et* donne sa *nouvelle* valeur, tandis que "V++" incrémente V et donne sa valeur *initiale.* C fournit également des opérateurs de pré-décroissance et de post-décroissance nommés " -- ".

EXEMPLE : Programmation concise en C

Considérons la fonction C suivante :

```
int palindromique (char s[], int n) {
/* Retourne non nul si et seulement si la séquence de caractères de s[0], . . . ,
* s[n-1] est un palindrome.
*/    int l, r;
   l = 0; r = n-1;
   while (l < r) {
         if (s[l++] != s[r--])
           return 0;
   }
   return 1;
}
```

Notez que le type de résultat est **int**, bien que le résultat soit logiquement un booléen. Notez également l'utilisation d'opérateurs post-incrément et post-décrément pour rendre le corps de la boucle concis et efficace.
Considérons maintenant la version alternative suivante :

```
int palindromique (char s[], int n) {
char* lp, rp;
while (lp < rp) {
          if (*lp++ != *rp--)
            return 0;
    }
    return 1;
}
```

Il remplace les variables entières l et r (qui ont été utilisées pour indexer le tableau s) par des variables de pointage lp et rp (qui pointent sur les composantes du tableau s). L'expression "*lp++" donne le caractère vers lequel pointe lp, puis incrémente lp (le faisant pointer vers la composante suivante de s). L'expression "*rp--" donne également le caractère vers lequel rp pointe, puis décrémente rp (le faisant pointer vers la composante précédente de s). L'expression "lp < rp" vérifie si la composante que lp désigne précède (est à gauche de) la composante que rp désigne.

Cette deuxième version est parfaitement légale, mais beaucoup plus difficile à comprendre que la première. Cependant, elle illustre un idiome de programmation que les programmeurs C expérimentés apprennent à reconnaître.

4.2.2.3 Liaisons et portée

Un programme C se compose d'un type de global, de déclarations de variables globales et de la définitions de fonctions. Une de ces fonctions doit être nommée main, et c'est le programme principal.
À l'intérieur d'une fonction ou d'une commande de bloc, nous pouvons déclarer des types locaux, des variables locales et des variables statiques. Les variables locales ont une durée de vie plus courte que les variables globales, et contrairement à ces dernières, elles ne sont pas la cause d'un couplage étroit entre les fonctions.

Une *variable statique* est globale dans le sens où sa durée de vie est la durée d'exécution totale du programme, mais locale dans le sens où sa portée est la fonction ou la commande de bloc dans laquelle elle est déclarée.

4.2.3 Abstraction procédurale

C ne prend en charge que les fonctions. Cependant, on peut obtenir l'effet d'une procédure correcte en écrivant une fonction dont le type de résultat est *nul*.

C ne supporte qu'un seul mécanisme de paramètre, le paramètre de copie. C'est- à-dire que chaque paramètre formel se comporte comme une variable locale qui est initialisée à la valeur de l'argument correspondant. Cependant, nous pouvons obtenir l'effet d'un paramètre de référence en passant un *pointeur sur* une variable en tant qu'argument.

EXEMPLE : Fonction en C avec des arguments de pointeur

Considérons la fonction C suivante :

```
void minimax (int[] a, int n, int* min, int* max) {
/* Définir *min au minimum, et *max au maximum, de les
* nombres entiers dans a [0], . . . ,a[n-1].
*/
   *min = *max = a[0];
   int i;
   for (i = 1; i < n; i++) {
        int elem = a[i];
        if (elem < *min) *min = elem;
        else if (elem > *max) *max = elem;
     }
}
```

Voici un appel possible à la fonction :

```
int[] temps = {13, 15, 20, 18, 21, 14, 12};
int low, high;
. . .
minimax(temps, 7, &low, &high);
```

Les paramètres formels min et max sont en fait des paramètres de référence. Les arguments corrélatifs sont des *pointeurs de* variables.

4.3 La programmation orientée objet

4.3.1 Concepts fondamentaux

Un programme *orienté objet* consiste en un ou plusieurs *objets* qui interagissent les uns avec les autres pour résoudre un problème. Un objet contient des informations d'état (données, représentées par d'autres objets) et des opérations (code). Les objets interagissent en s'envoyant des *messages* entre eux. Ces messages sont comme des appels de procédure ; les procédures sont appelées *méthodes*. Chaque objet est une instance d'une *classe,* qui détermine les données que l'objet conserve comme informations d'état et les messages que l'objet comprend. Le *protocole de* la classe est l'ensemble des messages que ses instances comprennent.

4.3.1.1 Objets

La principale construction des langages orientés objet est clairement celle de l'objet. Une capsule contenant à la fois des données et les opérations qui les manipulent et qui fournissent une interface avec le monde extérieur par laquelle l'objet est accessible.

4.3.1.2 Classes

Une classe est un modèle pour un ensemble d'objets. Elle établit ce que seront ses données (type à obtenir avec leur visibilité) et fixe le nom, la signature, la visibilité et l'implémentation pour chacune de ses méthodes. Dans un langage avec classes, chaque objet appartient à (au moins) une classe. Par exemple, un objet pourrait être une instance de la classe suivante :

```
class Counter{
private int x;
public void reset(){
        x = 0;
}
public int get(){
        return x;
}
public void inc(){
        x = x+1;}}
```

On peut voir que la classe contient l'implémentation de trois méthodes qui sont déclarées publiques (visibles par tous), tandis que la variable d'instance x est privée (inaccessible de l'extérieur de l'objet lui-même mais accessible dans l'implémentation des méthodes de cette classe).

4.3.1.3 Encapsulation

L'encapsulation et la dissimulation d'informations représentent deux des points cardinaux de l'abstraction de données. Chaque langage permet de définir des objets en cachant une partie (soit des données, soit des méthodes). Dans chaque classe, il y a donc au moins deux *vues* : la partie privée et la partie publique. Dans la vue privée, tout est visible : c'est le niveau d'accès possible à l'intérieur de la classe elle-même (par ses méthodes). En revanche, dans la vue publique, seules les informations explicitement exportées sont visibles.

4.3.1.4 Sous-types

On peut faire correspondre une classe, de façon naturelle, à l'ensemble des objets qui sont des instances de cette classe. Cet ensemble d'objets est le type associé à cette classe. Dans les langages typés, cette relation est explicite. La définition d'une classe introduit également la définition d'un type dont les valeurs sont les instances de cette classe. Dans les langages sans type (comme Smalltalk), la correspondance n'est que conventionnelle et implicite.

Certains langages (comme Java et C++) utilisent une équivalence basée sur le nom pour les types qui ne s'étend pas correctement à une relation de compatibilité complètement structurelle. Dans ces langages, ce n'est donc pas la seule propriété structurelle entre les interfaces qui définit la relation de sous-type, mais elle doit être explicitement introduite par le programmeur. C'est le rôle de la définition des sous-classes, ou classes dérivées, qui dans notre exemple seront désignées en utilisant la construction d'extension neutre:

```
  class NamedCounter extending Counter{
private String name;
public void set_name(String n){
        name = n;
}
public String get_name(){
        return name;
}
}
```

La classe NamedCounter est une sous-classe de Counter (qui, à son tour, est une superclasse de NamedCounter), c'est-à-dire que le type NamedCounter est un sous-type de Counter. Les instances de NamedCounter contiennent tous les champs de Counter

(même ses champs privés, mais ils sont inaccessibles dans la sous-classe), en plus d'avoir de nouveaux champs introduits par la définition. De cette façon, la compatibilité structurelle peut être garantie (une sous-classe est explicitement dérivée de sa superclasse), mais cela est explicitement indiqué dans le programme.

Redéfinition d'une méthode. Dans l'exemple simple de NamedCounter, les sous-classes se limitent à étendre l'interface de la superclasse. Une caractéristique fondamentale du mécanisme des sous-classes est la capacité d'une sous-classe à modifier la définition (l'implémentation) d'une méthode présente dans sa superclasse. Ce mécanisme est appelé "*méthode dérogatoire*".

```
class NewCounter extending Counter{
 private int num_reset = 0; public void reset(){
        x = 0;
       num_reset = num_reset + 1;
}
public int howmany_resets(){
        return num_reset;
}
}
```

La classe NewCounter étend simultanément l'interface de Counter avec de nouveaux champs et redéfinit la méthode de réinitialisation. Une méthode de réinitialisation envoyée à une instance de NewCounter entraînera l'invocation de la nouvelle implémentation.

Classes abstraites . Pour simplifier l'exposition, nous avons introduit le type assorti d'une classe comme ensemble de ses instances. Cependant, de nombreux langages permettent de définir des classes qui ne peuvent pas avoir d'instances parce que la classe n'a pas la mise en oeuvre d'une méthode. Dans ces classes, il n'y a que le nom et le type (c'est-à-dire la signature) d'une ou plusieurs méthodes - leur implémentation est omise. Les classes de ce type sont appelées classes *abstraites. Les* classes abstraites servent à fournir des interfaces et peuvent être implémentées dans des sous-classes qui redéfinissent.

Shadowing. En plus de modifier l'implémentation d'une méthode, une sous-classe peut également redéfinir une variable d'instance (ou champ) définie dans une superclasse. Ce mécanisme est appelé "*shadowing*". Pour des raisons d'implémentation (voir ci-dessous), le *Shadowing* est significativement différent de l'overriding. Pour l'instant, nous nous contentons de noter que, dans une sous-classe, une variable d'instance peut être redéfinie avec le même nom et le même type que dans la superclasse. Nous pourrions, par exemple, modifier nos Counters étendus en utilisant la sous-classe suivante de NewCounter où, pour une raison (étrange), la valeur initiale de num_reset est initialisée à 3 et est incrémentée de 3 à chaque fois :

```
class EvenNewCounter extends NewCounter{
private int num_reset = 2;
public void reset(){
        x = 0;
        num_reset = num_reset + 3;
}
public int howmany_resets(){
         return num_reset;
}
}
```

En utilisant la notion habituelle de visibilité dans les langages structurés en blocs, chaque référence à num_reset dans EvenNewCounter renvoie à la variable locale (initialisée à 3) et non à celui déclaré dans NewCounter. Un message de réinitialisation envoyé à une position de EvenNewCounter provoquera l'appel de la nouvelle implémentation pour la réinitialisation qui utilisera le nouveau champ num_reset. Cependant, comme nous le verrons plus loin, il y a une grande différence, tant au niveau sémantique qu'au niveau de l'implémentation, entre l'overriding et le shadowing.

4.3.1.5 Héritage

L'héritage permet la réutilisation du code dans un contexte extensible. En modifiant l'implémentation d'une méthode, une classe met automatiquement la modification à la disposition de toutes ses sous-classes, sans aucune intervention de la part du programmeur.

Héritage et visibilité. Nous avons déjà vu qu'il existe deux points de vue sur chaque classe : le privé et le public, ce dernier étant partagé entre tous les clients de la classe. Une sous-classe est un client particulier de la superclasse. Elle utilise les méthodes de la superclasse pour étendre les fonctionnalités de la superclasse, mais doit parfois accéder à certaines données non publiques. De nombreux langages introduisent donc une troisième vue d'une classe : une pour les sous-classes. En reprenant le terme du C++, nous pouvons l'appeler la vue *protégée d*'une classe.

Héritage simple et multiple. Dans certains langages, une classe peut hériter d'une seule superclasse immédiate. La hiérarchie d'héritage est donc un arbre et on dit que le langage a un héritage unique (ou simple). D'autres langages, en revanche, permettent à une classe d'hériter des méthodes de plusieurs superclasses immédiates ; la hiérarchie d'héritage dans ce cas est un graphe orienté acyclique et le langage a un héritage multiple.

Seuls quelques langages supportent l'héritage multiple (parmi lesquels Eiffel et C++), car il présente des problèmes qui n'ont pas de solution élégante, que ce soit au niveau conceptuel ou au niveau de l'implémentation. Les problèmes fondamentaux sont liés aux conflits de noms. Nous avons un conflit de noms lorsqu'une classe C hérite simultanément de A et B, qui fournissent tous deux une implémentation pour des méthodes ayant la même signature.

Voici un exemple simple :

```
class A{
    int x;
    int f(){
        return x;
    }
}
class B{
    int y;
    int f(){
        return y;
    }
}
class C extending A,B{
    int h(){
        return f();
    }
}
```

4.3.2 Langage JAVA

JAVA est un langage orienté objet assez pur : toutes les valeurs composites sont des objets, et un programme est entièrement constitué de classes. JAVA prend également en charge la programmation simultanée et distribuée, bien que même celle-ci soit basée sur des concepts orientés objet.

4.3.2.1 Variables, stockage et contrôle

JAVA prend en charge les variables globales, locales et de tas. Cependant, les seules variables globales d'un programme JAVA sont les variables de classe. Les variables de tas sont des objets créés par le **nouvel** allocateur ; elles sont détruites *automatiquement* lorsqu'elles ne sont plus accessibles. La désallocation automatique (implémentation par un éboueur) simplifie la tâche du programmeur et élimine une source commune d'erreurs.

JAVA hérite de toutes les commandes et séquenceurs de C++, à l'exception des sauts. Les exceptions JAVA sont des objets d'une sous-classe d'Exception. Une méthode JAVA peut spécifier les classes d'exceptions qu'elle peut lancer ; une méthode sans spécification d'exception ne peut lancer aucune exception.

4.3.2.2 Valeurs et types

Les types primitifs de JAVA sont similaires à ceux du C et du C++. Une différence importante est que leurs représentations et leurs opérations sont précisément stipulées par JAVA. Par exemple, JAVA stipule que les valeurs **int** sont des entiers de 32 bits de complément à deux. (C et C++ permettent

tous deux à leurs compilateurs d'implémenter **int** en termes de longueur de mot et d'arithmétique natives de l'ordinateur, en insistant seulement sur le fait que **int** a une plage plus large que **int court** mais une plage plus étroite que int **long**). De même, la JAVA stipule que les valeurs **flottantes** sont des nombres à 32 bits de la norme *IEEE floating-point*.

4.3.2.3 Liaisons et portée

Un programme JAVA se compose d'un certain nombre de déclarations de classe, qui peuvent être regroupées en packages. L'utilisateur doit désigner une classe équipée d'une méthode appelée main, et c'est le programme principal.
Dans chaque déclaration de classe, nous déclarons les composantes variables, les constructeurs, les méthodes et les classes internes de cette classe. Au sein d'un constructeur ou d'une méthode, nous pouvons déclarer des variables locales.

Toute variable JAVA peut être déclarée avec le spécificateur *final*, ce qui permet de l'inspecter mais pas de la mettre à jour. En effet, il s'agit d'une constante plutôt que d'une variable. Le spécificateur *final* peut également être utilisé pour déclarer une méthode, indiquant que la méthode n'est pas prioritaire, ou pour déclarer une classe, indiquant que la classe ne peut pas avoir de sous-classes.

4.3.2.4 Extraction des données

JAVA permet l'abstraction de données au moyen de classes. Une déclaration de classe nomme la classe elle-même et sa superclasse (Objet par défaut), déclare les composants variables de la classe, et définit ses constructeurs et méthodes.

JAVA fait la distinction entre les variables d'instance et les variables de classe, et aussi entre les méthodes d'instance et les méthodes de classe. Les variables de classe et les méthodes de classe se distinguent par un spécificateur. *statique*.

JAVA soutient le polymorphisme d'inclusion sans restriction : un objet d'une sous-classe peut être traité comme un objet de sa superclasse en toutes circonstances.

EXEMPLE : Polymorphisme d'inclusion en JAVA

Considérez la classe suivante :

```
class Person {
   private String surname, forename;
   private bool female;
   private int birthYear;
   public Person (String sname, String fname,
                  char gender, int birth) {
      surname = sname;
```

```
                    forename = fname;
                    female = (gender == 'F' || gender == 'f');
                                        birthYear = birth;

                }
                public String getSurname () {
                        return surname;
                }
                public void changeSurname (String sname) {
                         surname = sname;
                }
                public void print () {
                        . . . // écrire le nom de cette personne
                }
            }
```

et la sous-classe suivante :

```
            class Student extends Person {
                private int student_id;
                private String degree;
                public Student (String sname, String fname,
                            char gender, int birth, int id,
                                        String deg) {
                     . . .
                }
                void changeDegree (String deg) {
                       degree = deg;
                }
                void print () {
          . . . // écrire le nom et l'identifiant de cet étudiant
                }
            }
```

Le code suivant déclare et utilise un ensemble d'objets Personne :

```
            Person p = new Person[10];
            Person dw = new Person("Watt", "David", 'M', 1946);
            Student jw = new Student("Watt", "Jeff", 'M', 1983,
                           0100296, "BSc");
            p[0] = dw;
            p[1] = jw; // legal
                for (int i = 0; i < 2; i++)
                      p[i].print(); // safe
```

L'affectation à p[1] est légale car les types Personne et Étudiant sont compatibles. De plus, la méthode appelée "p[i].print()" est sûre, et appellera soit la méthode d'impression de la classe Person, soit la méthode d'impression de la classe Student, selon la classe de l'objet donné par p[i]. Il s'agit d'une répartition dynamique.

4.3.2.5 Abstraction générique

JAVA n'a pas soutenu l'abstraction générique. Il s'agissait là d'une faiblesse notable, qui a obligé les programmeurs à trop s'appuyer sur le polymorphisme d'inclusion, comme l'illustre l'exemple suivant.

EXEMPLE : Classe de collection en JAVA

Considérez la classe suivante :

```
class Stack {
 // Un objet Stack représente une pile dont les éléments
 // sont des objets.
  private Object[] elems;
  private int depth;
  public Stack () { depth = 0; }
  public void push (Object x) { elems[depth++] = x; }
  public Object pop () { return elems[--depth]; }
  public boolean isEmpty () { return (depth == 0); }
}
```

Comme chaque classe JAVA est une sous-classe d'objet, un objet Stack peut contenir des objets de n'importe quelle classe. Il peut même être hétérogène, contenant des objets de différentes classes :

```
Stack stack = new Stack();
String s1 = . . .; Date d1 = . . .;
stack.push(s1);
stack.push(d1);
. . .
Date d2 = (Date)stack.pop();
String s2 = (String)stack.pop();
But now consider the following code:
Stack stack = new Stack();
for (. . .) {
    String s = . . .;
    stack.push(s);
}
while (! stack.isEmpty()) {
    String s = (String)stack.pop();
     . . .
}
```

Un lecteur humain peut facilement voir que seuls les objets de type "String" sont poussés sur la pile, donc seuls les objets de type "String" peuvent être poussés. Mais le compilateur ne peut pas voir cela ; il peut seulement déduire que le type de la méthode appelée "stack.pop()" est Object, d'où la nécessité de lancer le résultat pour taper String.

4.3.2.6 Abstraction procédurale

JAVA soutient les constructeurs et les méthodes, qui sont toujours des composantes d'une classe par ticulière. JAVA ne prend pas en charge les procédures qui sont indépendantes de toute classe et ne prend en charge que les paramètres de copie. En d'autres termes, le paramètre formel agit comme une variable locale qui est initialisée avec la valeur de l'argument. Lorsque

l'argument est une valeur primitive, il peut être inspecté mais pas mis à jour. En revanche, lorsque l'argument est un objet, il est accessible par un pointeur ; le pointeur ne peut pas être mis à jour, mais l'objet lui-même le peut. Cela nous donne l'effet d'un mécanisme de paramètres de référence pour les objets uniquement.

4.3.2.7 Liaison dynamique

L'implémentation de JAVA fait appel à des *liaisons dynamiques*. Considérons une applet composée de plusieurs classes compilées, situées sur un serveur Web. Au départ, seul le code objet d'une classe est téléchargé depuis le serveur, à savoir la classe contenant le programme principal, qui commence immédiatement à s'exécuter. Le code objet de chaque autre classe est téléchargé et lié au programme en cours d'exécution uniquement lorsque (et si) nécessaire. La liaison dynamique permet d'éviter de télécharger les classes qui constituent le programme. (Elle *réduira* potentiellement l'overhead, si certaines des classes ne sont pas nécessaires dans une exécution particulière du programme).

EXEMPLE : Liaison dynamique en JAVA

Supposons que la classe suivante soit donnée :

```
class Widget {
    private . . . secret;
    public Widget () {. . .} // constructeur accédant au secret
    public . . . // méthodes accédant au secret
    }
```

Supposons qu'un programmeur malveillant souhaite écrire un programme principal qui accède à w.secret directement :

```
class Malicious {
    public static void main (. . .) {
        Widget w = new Widget();
        . . . w.secret . . .
    }
}
```

Heureusement, le compilateur JAVA rejettera cette tentative illégale d'accès à une variable d'instance privée.

Cependant, si le programmeur parvient à obtenir le code source de la classe Widget, il compile une version modifiée :

```
class Widget {
    public . . . secret; // Maintenant, secret est public!
    public Widget () {. . .}
    public . . . // méthodes

}
```

et compile ensuite Malicious en important la version modifiée de la classe Widget. Enfin, il s'assure que la version *originale de la classe* Widget se trouve dans l'environnement d'exécution du programme. Le programme malveillant accède alors avec succès à w.secret.

4.4 La programmation concurrente

4.4.1 Concepts fondamentaux

Dans un programme concurrent, nous utiliserons le terme "*thread*" pour désigner l'entité active que le programmeur considère comme fonctionnant en même temps que d'autres threads. Dans la plupart des systèmes, les threads d'un programme donné sont mis en œuvre en plus d'un ou plusieurs *processus* fournis par le système d'exploitation. Les concepteurs de systèmes d'exploitation font souvent la distinction entre un processus *lourd,* qui possède son propre espace d'adressage, et un ensemble de processus *légers,* qui peuvent partager un espace d'adressage. Nous utiliserons parfois le mot *tâche* pour désigner une unité de travail bien définie qui doit être réalisée par un fil. Dans un langage de programmation commun, une collection de threads partage un "sac de tâches" commun - une liste de travail à effectuer. Chaque fil retire à plusieurs reprises une tâche, l'exécute et en reprend une autre. Parfois, le travail d'une tâche implique l'ajout de nouvelles tâches dans le sac.

4.4.2 Démarrage de plusieurs threads

La syntaxe pour lancer des calculs multiples a tendance à être simple. Dans Modula, un thread est lancé en invoquant un objet de type procédure ; lorsque la "procédure" revient, le thread disparaît. Pendant ce temps, le calcul qui a lancé le thread continue de s'exécuter. Le nouveau thread peut lui-même créer d'autres threads en les invoquant. L'exemple ci-dessous montre un programme qui fusionne-trie un tableau en créant récursivement des threads.

```
Type                                                    1
    DataArray = array whatever of integer;              2
thread MergeSort(                                       3
    reference Tangled : DataArray;                      4
    value LowIndex, HighIndex : integer);               5
variable                                                6
    MidPoint : integer := (LowIndex + HighIndex) div 2; 7
begin                                                   8
    if LowIndex + 1 < HighIndex then -- worth sorting   9
    MergeSort(Tangled, LowIndex, MidPoint);             10
    MergeSort(Tangled, MidPoint+1, HighIndex);          11
    Merge(Tangled, 1, MidPoint, MidPoint+1,             12
          HighIndex);                                   13
    end; -- worth sorting                               14
  end; -- MergeSort                                     15
```

MergeSort est déclaré à la ligne 3 comme un thread, *et* non comme une *procédure*. Toutes les invocations de MergeSort, y compris les récursives des lignes 10 et 11, créent de nouveaux threads exécutant des instances de MergeSort qui fonctionnent indépendamment du programme principal. Malheureusement, MergeSort n'attend pas que ses enfants finissent et se précipite à la ligne 12, fusionnant les deux moitiés du tableau avant qu'elles ne soient correctement triées. Vous verrez bientôt des mécanismes de synchronisation qui me permettront de corriger ce bogue.

Chaque thread reçoit sa propre pile. Les variables déclarées localement à un thread sont comme des variables locales dans une procédure ; chaque thread reçoit ses propres variables locales. De même, toute procédure appelée depuis un thread (telle que Merge, appelée à la ligne 12) obtient de nouveaux enregistrements d'activation sur la pile du thread. Cependant, les variables qui sont en dehors de la portée du thread sont partagées entre tous les threads dans lesquels elles sont visibles par les règles normales de portée. En d'autres termes, la chaîne statique de la pile d'un thread finit par pointer vers l'extérieur de la pile privée du thread dans la pile partagée. (Cet arrangement est parfois appelé pile de **cactus**, car les piles ressemblent aux branches d'un cactus saguaro ou cholla).

Certains langages permettent de démarrer les threads par une déclaration de *cobegin*. Toutes les instructions du *cobegin* sont lancées comme des threads séparés. Cette construction comprend une étape de synchronisation implicite : Le *cobegin* ne s'achève pas avant que chacun de ses fils n'ait terminé.

4.4.3 Syntaxe de création de thread

Presque tous les systèmes concurrents permettent de créer (et de détruire) des threads de manière dynamique. Les détails syntaxiques et sémantiques varient considérablement d'un langage ou d'une bibliothèque à l'autre, mais la plupart se conforment à l'une des six options principales : co-début, fork/join, boucles parallèles, lancement à l'élaboration, réception implicite et réponse rapide. Les deux premières options délimitent des threads avec des constructions spéciales de flux de contrôle. Les autres utilisent une syntaxe ressemblante (ou identique) à celle des sous-programmes.
La plupart des bibliothèques utilisent des fork/join, tout comme Java et C#. ADA utilise à la fois launch-at-elaboration et fork. OpenMP utilise des boucles de co-début et des boucles parallèles. Les systèmes RPC sont généralement basés sur la réception implicite.

4.4.3.1 Boucles parallèles

De nombreux systèmes concurrents, dont OpenMP, plusieurs dialectes du Fortran et la bibliothèque Parallel FX pour .NET, récemment annoncée, fournissent une boucle dont les itérations.

En C# avec Parallel FX, le code équivalent ressemble à ceci

```
Une boucle parallèle en C#
Parallel.For(0, 3, i => {
Console.WriteLine("Thread " + i + " ici ");
});
```

Le troisième argument de Parallel. For est un délégué, dans ce cas une expression lambda. Une méthode Foreach similaire prévoit deux arguments - un itérateur et un délégué.

4.4.3.2 Fork/Join

Le co-début, les boucles parallèles et le lancement à l'élaboration conduisent tous à un schéma de contrôle-flux simultané co-début vs fork/join dans lequel les exécutions de threads sont correctement imbriquées (voir figure 4.3a). L'opération fork est plus générale : elle fait de la création de threads une opération explicite et exécutable. L'opération de jointure compagnon, lorsqu'elle est fournie, permet à un thread d'attendre l'achèvement d'un thread précédemment fourché.
Comme les fork and join ne sont pas liées à des constructions imbriquées, elles peuvent conduire à des modèles arbitraires de flux de contrôle simultané (figure 4.3b).

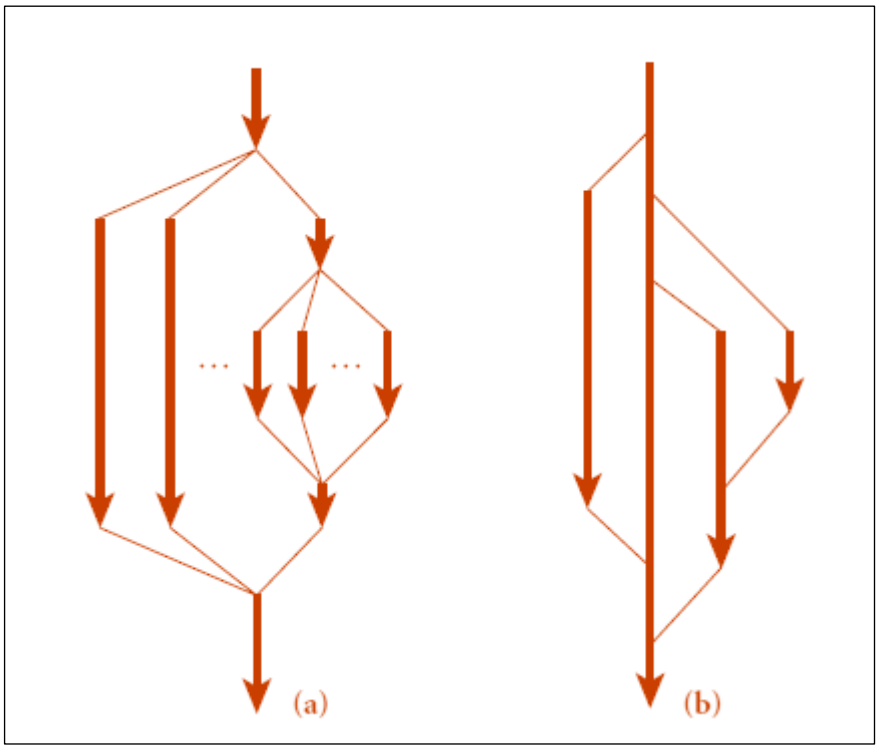

Figure 4.3 : ***Durée de vie des threads concurrents***.

Avec le co-début, les boucles parallèles ou l'élaboration du lancement (a), les threads sont toujours correctement imbriqués. Avec le fork/join (b), des modèles plus généraux sont possibles.

4.4.3.3 Co-début

La sémantique habituelle d'un énoncé composé (parfois délimité par la forme générale de début de co-début. . . fin) appelle une exécution séquentielle des énoncés constitutifs. Une construction de co-début appelle plutôt une exécution simultanée :

EXEMPLE

```
co-begin  – – toutes les n instructions s'exécutent simultanément
stmt 1
stmt 2
. . .
stmt n
end
```

Chaque énoncé peut lui-même être un composé séquentiel ou parallèle, ou (communément) un appel de sous-programme.
Le co-début était le principal moyen de créer des threads en Algol-68. Il apparaît dans Co-débuter dans OpenMP une variété d'autres systèmes également, y compris OpenMP :

EXEMPLE

```
#pragma omp sections
{
# pragma omp section
{ printf("thread 1 here\n"); }
# pragma omp section
{ printf("thread 2 here\n"); }
}
```

En C, les directives OpenMP commencent toutes par #pragma omp. (Le signe # doit apparaître dans la colonne 1). La plupart des directives, comme celles présentées ici, doivent apparaître immédiatement avant une construction en boucle ou un énoncé composé délimité par des accolades.

4.4.3.4 Lancement lors de l'élaboration

Dans plusieurs langages, dont ADA, le code d'un thread peut être déclaré avec une syntaxe ressemblant à celle d'un sous-routine sans paramètres. Dans ADA (qui appelle ses threads Tâches élaborées dans les tâches ADA), on peut écrire

```
procedure P is
task T is
...
end T;
begin -- P
...
end P;
```

La tâche T a son propre début. bloc de fin, qu'elle commence à exécuter dès que le contrôle entre dans la procédure P. Si P est récursive, il peut y avoir plusieurs instances de T en même temps, qui s'exécutent toutes simultanément les unes avec les autres et avec la tâche en cours d'exécution (l'instance actuelle de) P. Le programme principal se comporte comme une tâche initiale par défaut. Lorsque le contrôle atteint la fin de la procédure P, il attend que l'instance appropriée de T (celle qui a été créée au début de cette instance de P) se termine avant de revenir. Cette règle garantit que les variables locales de P (qui sont visibles par T selon les règles statiques habituelles de portée) ne sont jamais désallouées avant que T n'en ait fini avec elles.

4.4.4 Mécanismes de niveau de langage

Bien que largement utilisés, les sémaphores sont aussi largement considérés comme étant de trop "bas niveau" pour un code bien structuré et maintenable. Ils souffrent de deux problèmes principaux. Premièrement, comme leurs opérations ne sont que des appels de sous-programmes, il est facile d'en laisser un de côté (par exemple, sur un chemin de contrôle avec plusieurs instructions if imbriquées). Deuxièmement, à moins qu'elles ne soient cachées dans une abstraction, les utilisations d'un sémaphore donné ont tendance à se disperser dans un programme, ce qui rend difficile leur suivi à des fins de maintenance logicielle.

4.4.4.1 Sémaphores

Le sémaphore est au cœur de la plupart des méthodes de synchronisation. Son implémentation (souvent cachée au programmeur) est illustrée dans l'exemple suivant

```
type
    Semaphore =
        record -- champs initialisés comme indiqué
        Value : integer := 1;
        Waiters : queue of thread := empty;
        end;
```

Les sémaphores ont deux opérations, qui sont invoquées par des déclarations. (Les opérations peuvent être présentés comme des appels de procédure). J'appelle le premier *vers le bas* (parfois les gens l'appellent *P*, *attendre* ou *acquérir*). La deuxième opération est en *haut* (aussi appelée *V*, *signal* ou *libération*) :

- ✓ *baisse de la* valeur S.Value . Il bloque ensuite l'appelant, en sauvegardant son identité dans S.Waiters, si la valeur est maintenant négative.
- ✓ *up* S increments S.Value. Il débloque le premier threadd'attente dans S. Waiters si Value est maintenant non positif.

Ces deux opérations sont *indivisibles, c'est-à-dire qu'*elles se complètent instantanément dans un thread pour ce qui est des autres threads. Par conséquent, un seul thread à la fois peut *monter* ou *descendre* un sémaphore particulier à la fois.

Les sémaphores peuvent être utilisés pour mettre en œuvre l'exclusion mutuelle. Toutes les régions qui utilisent les mêmes variables partagées sont associées à un sémaphore particulier, initialisé avec la valeur = 1. Un thread qui souhaite entrer dans une région *abaisse* le sémaphore associé. Il a maintenant atteint l'exclusion mutuelle en acquérant un verrou exclusif. Lorsque le thread quitte la région, il *remonte le* même sémaphore, libérant ainsi le verrouillage. Le premier thread à essayer d'entrer dans sa région réussit. Un autre thread qui tente d'entrer alors que le premier est encore dans sa région sera bloqué. Lorsque le premier thread quitte la région, le deuxième thread est débloqué. La valeur est toujours égale à 0 ou 1 (si seulement deux threads sont en compétition). C'est pourquoi les sémaphores utilisés pour l'exclusion mutuelle sont souvent appelés *sémaphores binaires*.

4.4.5 Moniteur

Un moniteur est un module ou un objet avec des opérations, un état interne et un certain nombre de *variables d'état*. Une seule opération d'un moniteur donné est autorisée à être active à un moment donné. Un thread qui appelle un moniteur occupé est automatiquement retardé jusqu'à ce que le moniteur soit libre. Au nom de son thread d'appel, toute opération peut se suspendre en *attendant* une variable d'état. Une opération peut également *signaler* une variable de condition, auquel cas l'un des threads en attente est repris, généralement celui qui a attendu le premier.

Comme les opérations (*entrées*) d'un moniteur s'excluent automatiquement les unes les autres à temps, le programmeur est déchargé de la responsabilité d'utiliser correctement les opérations P et V. De plus, comme le moniteur est une abstraction, toutes les opérations sur les données encapsulées, y compris la synchronisation, sont rassemblées en un seul endroit. La figure 4.4 présente une solution basée sur le moniteur pour résoudre le problème du tampon borné. Il convient de souligner que les variables d'état du moniteur ne sont pas les mêmes que celles des sémaphores. Plus précisément, elles n'ont pas de "mémoire" : si aucun thread n'attend une condition au moment où un signal se produit, alors le signal n'a aucun effet. En revanche, une opération V sur un sémaphore incrémente le compteur du sémaphore, permettant à une future opération P de réussir, même si aucune n'attend maintenant.

```
monitor bounded_buf
imports bdata, SIZE
exports insert, remove

    buf : array [1..SIZE] of bdata
    next_full, next_empty : integer := 1, 1
    full_slots : integer := 0
    full_slot, empty_slot : condition

    entry insert(d : bdata)
        if full_slots = SIZE
            wait(empty_slot)
        buf[next_empty] := d
        next_empty := next_empty mod SIZE + 1
        full_slots +:= 1
        signal(full_slot)

    entry remove : bdata
        if full_slots = 0
            wait(full_slot)
        d : bdata := buf[next_full]
        next_full := next_full mod SIZE + 1
        full_slots –:= 1
        signal(empty_slot)
        return d
```

Figure 4.4 *: Code basé sur le moniteur pour un tampon délimité.*

L'insertion et la suppression sont des sous-programmes *d'entrée* : ils nécessitent un accès exclusif aux données du moniteur. Comme les conditions sont sans mémoire, l'insertion et la suppression peuvent toutes deux se terminer en toute sécurité par un signal.

4.4.6 Mutexes

Certains langages, comme Modula-3, prédisent un type de *mutex* qui est implémenté par des sémaphores binaires. La déclaration de *verrouillage* entoure toutes les déclarations qui doivent exclure d'autres threads, comme le montre l'exemple suivant.

```
variable                                    1
    A, B : integer;                         2
    AMutex, BMutex : mutex;                 3
procedure Modify();                         4
begin                                       5
    lock AMutex do                          6
        A := A + 1;                         7
    end;                                    8
```

```
            lock BMutex do                          9
                B := B + 1;                         10
            end;                                    11
            lock AMutex, BMutex do                  12
                A := A + B;                         13
                B := A;                             14
            end;                                    15
        end; -- Modify                              16
```

La variable A est protégée par le mutex AMutex, et la variable B est protégée par le BMutex. La déclaration de **verrouillage** (comme à la ligne 6) équivaut à une opération de **descente au** début et de **montée à la** fin. Plusieurs threads peuvent s'exécuter simultanément dans Modify. Cependant, un thread qui exécute la ligne 7 empêche tout autre thread d'exécuter l'une des lignes 7, 13 et 14. Il est possible qu'un thread se trouve à la ligne 7 et un autre à la ligne 10. Je verrouille les lignes 7 et 10 parce que sur de nombreuses machines, l'incrémentation nécessite plusieurs instructions, et si deux threads exécutent ces instructions à peu près en même temps, la variable peut être incrémentée une seule fois au lieu de deux. Je verrouille les lignes 13 et 14 ensemble pour m'assurer qu'aucun thread ne peut intervenir après la ligne 13 et avant la ligne 14 pour modifier A. Le verrouillage multiple de la ligne 12 verrouille d'abord AMutex, puis BMutex. L'ordre est important pour éviter les blocages.

4.4.7 Régions critiques conditionnelles

Les régions critiques conditionnelles (CCR) sont une autre alternative aux sémaphores, proposée par Brinch Hansen à peu près en même temps que les moniteurs. Une région critique est une section critique délimitée syntaxiquement dans laquelle un code est autorisé à accéder à une variable *protégée.* Une région critique *conditionnelle* spécifie également une condition booléenne, qui doit être vraie avant que le contrôle n'entre dans la région :

```
region protected variable, when Boolean condition do
. . .
end region
```

Aucun thread ne peut accéder à une variable protégée sauf dans une déclaration de région pour cette variable, et tout thread qui atteint une déclaration de région attend que la condition soit vraie et qu'aucun autre thread ne soit actuellement dans une région pour la même variable. Les régions peuvent s'imbriquer, mais comme pour les appels de surveillance imbriqués, le programmeur doit s'inquiéter d'une éventuelle impasse.

Le langage Edison permet de programmer une synchronisation plus expressive que les mutex, mais moins sujette aux erreurs que les sémaphores nus. Un exemple standard qui montre la nécessité de la synchronisation est le **tampon délimité**, qui est un tableau

rempli par les threads de production et vidé par les threads de consommation. Tous les producteurs et les consommateurs doivent s'exclure mutuellement pendant qu'ils inspectent et modifient les variables qui composent le tampon délimité. En outre, lorsque le tampon est plein, les producteurs doivent bloquer au lieu d'**attendre,** ce qui revient à tester à plusieurs reprises si le tampon a de la place. De même, lorsque le tampon est vide, les consommateurs doivent bloquer. Les exemples suivants montrent comment coder cette application avec des régions critiques conditionnelles.

```
constant                                                    1
    Size = 10; -- capacité du tampon (Buffer)               2
Type                                                        3
    Datum = ... -- contenu de la mémoire tampon (Buffer)    4
variable                                                    5
    Buffer : array 0..Size-1 of Datum;                      6
    InCount, OutCount : integer := 0;                       7
procedure PutBuffer(value What : Datum);                    8
begin                                                       9
    region Buffer, InCount, OutCount                        10
    await InCount - OutCount < Size do                      11
        Buffer[InCount mod Size] := What;                   12
        InCount := InCount + 1;                             13
    end; -- region                                          14
end -- PutBuffer;                                          15
procedure GetBuffer(result Answer : Datum);                 16
begin                                                       17
    region Buffer, InCount, OutCount                        18
    await InCount - OutCount > 0 do                         19
        Answer := Buffer[OutCount mod Size];                20
        OutCount := OutCount + 1;                           21
    end; -- region                                          22
end GetBuffer;                                              23
```

Les déclarations de **région qui** commencent aux lignes 10 et 18 sont comme des déclarations de **verrouillage,** sauf qu'elles nomment des variables à protéger, et non des muets, et qu'elles ont une composante d'**attente.** Le compilateur peut vérifier que les variables partagées ne sont accessibles que dans les déclarations de **région,** et il peut inventer les mutex appropriés.

La condition attendue est vérifiée pendant que les mutex correspondants sont tenus. Si la condition est fausse, les mutex sont libérés et le thread est bloqué (sur un sémaphore implicite). Chaque fois qu'un thread sort d'une région, tous les threads des régions en conflit (ceux qui utilisent certaines des mêmes variables partagées) qui sont bloqués pour des conditions sont débloquées, récupèrent leurs mutex et testent à

nouveau leurs conditions. Cette revérification répétée des conditions peut constituer un problème de performance majeur.

4.4.8 Synchronisation en Java

En Java, chaque objet accessible à plus d'un thread a un verrou d'exclusion mutuelle implicite, acquis et libéré au moyen de déclarations synchronisées :

```
synchronisé (my_shared_obj) {
... // section critique
}
```

Toutes les exécutions de déclarations synchronisées qui se réfèrent à un même objet partagé s'excluent mutuellement dans le temps. Les instructions synchronisées qui se réfèrent à des objets différents peuvent être exécutées simultanément. En tant que forme de sucre syntaxique, une méthode d'une classe peut être préfixée par le mot-clé synchronisé, auquel cas le corps de la méthode est considéré comme ayant été entouré par une instruction synchronisée implicite (cette). Les invocations de méthodes non synchronisées d'un objet partagé et les accès directs à des champs publics peuvent se dérouler simultanément, ou avec une déclaration ou une méthode synchronisée.

Dans une déclaration ou une méthode synchronisée, un thread peut se suspendre en appelant la méthode prédéfinie "wait". Wait n'a pas d'arguments en Java : le langage de base ne fait pas la distinction entre les différentes raisons pour lesquelles les threads peuvent être suspendus sur un objet donné (la bibliothèque java.util.concurrent, devenue standard avec Java 5, fournit un mécanisme formulant des conditions ; plus d'informations à ce sujet ci-dessous). Java permet de réveiller un thread pour de fausses raisons ; les programmes doivent donc intégrer l'utilisation de l'attente dans une boucle de test des conditions :

```
while (!condition) {
wait();
}
```

Un thread qui appelle la méthode d'attente d'un objet libère le verrou de l'objet. Cependant, dans le cas d'instructions synchronisées imbriquées ou d'appels imbriqués à des méthodes synchronisées, le thread *ne libère pas de* verrou sur les autres objets.

4.4.9 Coopération par appel de procédure

Une autre forme de coopération est réalisée par les appels de procédure. Lorsqu'un thread (le *client) en appelle un* autre (le *serveur)*, les informations peuvent être transmises dans les deux sens par le biais de paramètres. En général, les paramètres sont limités aux modes valeur et résultat. Un seul thread peut agir comme un client pour certains appels et comme un serveur pour d'autres.

4.4.9.1 Appel de procédure à distance (RPC)

On entend par appel de *procédure à distance (RPC)* une invocation qui est traitée non pas par des déclarations *d'acceptation*, mais par une procédure d'exportation ordinaire. De tels appels peuvent traverser des unités de compilation, des processus, des ordinateurs et même des programmes qui sont écrites à des moments différents dans des langages différentes.

4.4.9.2 Évaluation à distance (REV)

L'évaluation à distance (REV) est une technique qui permet aux clients d'envoyer non seulement des paramètres, mais aussi des procédures, au serveur. Les procédures peuvent faire référence à d'autres procédures exportées par le serveur. Par exemple, un serveur de distribution de courrier peut exporter une procédure DeliverMail. Un client qui souhaite envoyer une centaine de messages identiques pourrait utiliser RPC, en invoquant la procédure DeliverMail une centaine de fois, à chaque fois que le message passe. Il pourrait aussi utiliser REV, en envoyant une petite procédure au serveur qui invoque DeliverMail une centaine de fois. La méthode REV est probablement beaucoup plus efficace. Elle libère également le concepteur du serveur de courrier des soucis liés au fait que l'ensemble des procédures exportées par le serveur n'est pas exactement adapté à chaque client.

4.4.9.3 Rendez-vous

Un *rendez-vous* est un moyen explicite pour le serveur d'accepter les appels de procédure d'un autre thread. Un thread s'exécute au sein d'un module. Ce module exporte des *entrées*, qui sont les identificateurs de type procédure pouvant être invoqués par d'autres threads. La déclaration d'une entrée comprend une déclaration de ses paramètres formels.

ADA a été le premier grand langage de programmation à intégrer des fonctions de programmation simultanée de haut niveau. Un module de tâche est en quelque sorte similaire à un package, mais son corps est exécuté comme un nouveau processus. Les variables globales sont accessibles par un corps de tâche, conséquence normale des règles de portée d'ADA, de sorte que les tâches peuvent interagir par leur effet sur les variables partagées. Toutefois, l'ADA ne fournit que peu de soutien à cet égard, et le programmeur est chargé de veiller à ce que le partage des variables globales n'ait pas d'effets néfastes.

Un module de tâche ADA a à la fois une spécification et un corps. La spécification déclare les entrées qu'elle rend publiques. Le corps contient les déclarations privées, et l'implémentation des entrées publiques en termes de commandes d'acceptation. Contrairement à un package, un module de tâche peut être soit une ***tâche unique, soit*** un ***type de tâche*** qui peut être utilisé pour créer plusieurs tâches avec la même interface.

EXEMPLE : Rendez-vous en ADA

Le type de tâche ADA suivant met en œuvre un type de tampon abstrait à un seul emplacement :

```
task type Message_Buffer is
   entry send_message (item : in Message);
   entry receive_message (item : out Message);
end Message_Buffer;
task body Message_Buffer is
   buffer : Message;
begin
   loop
      accept send_message (item : in Message) do
         buffer := item;
      end;
      accept receive_message (item : out Message) do
         item := buffer;
      end;
   end loop;
end Message_Buffer;
```

Notez comment la structure de contrôle de Message_Buffer garantit que les messages sont alternativement stockés et récupérés, de sorte que toute tentative de récupération par un appelant alors que le tampon est vide est automatiquement forcée d'attendre que le tampon soit rempli à nouveau. Notez également l'absence de signalisation explicite et de constructions d'exclusion mutuelle.

Ces derniers ne sont pas nécessaires, car les variables locales d'un corps de tâche (comme le tampon (Buffer)) ne sont accessibles que par cette tâche, qui le fait donc en toute sécurité.

4.4.10 Passage des messages

La transmission de messages domine encore l'informatique distribuée et haut de gamme. Les supercalculateurs et les clusters à grande échelle sont programmés principalement en Fortran ou en C/C++ avec le progiciel de bibliothèque MPI. L'informatique distribuée repose de plus en plus sur des abstractions client-serveur superposées à des bibliothèques qui mettent en œuvre la norme Internet TCP/IP. Comme pour l'informatique à mémoire partagée, de nombreux langages de transmission de messages ont également été développés pour des domaines d'application particuliers, ou à des fins de recherche ou de pédagogie.

4.4.11 CSP

CSP (Communicating Sequential Processes) est une proposition faite par C. A. R. Hoare pour le passage de messages entre des threads qui ne partagent pas de variables. La communication se fait par l'*envoi* et la *réception de* déclarations. Bien que la déclaration d'envoi ressemble à une invocation de procédure, il s'agit en fait d'une spécification de modèle. Le modèle est construit à partir d'un identifiant et de

paramètres réels. Il est mis en correspondance avec un modèle dans une instruction de *réception dans* le thread de destination. Les variables du modèle de *réception* sont comme des paramètres formels ; elles acquièrent les valeurs des paramètres réels dans le modèle d'*envoi* correspondant. Les modèles correspondent si le nom du modèle et le nombre de paramètres sont les mêmes et si tous les modèles de paramètres formels correspondent aux modèles de paramètres réels. La correspondance est même utilisée pour les déclarations d'affectation, comme le montrent les exemples suivants

```
left := 3;  1
right := 4; 2
x := cons(left, right); -- assigns pattern "cons(3,4)" 3
form(right) := form(right+1); -- right := right+1 4
factor(cons(left,right)) := factor(cons(5,6)); 5
      -- left := 5; right := 6 6
right = imply(); -- pattern with no parameters 7
muckle(left) := mickle(left+1); -- match error 8
```

Les variables peuvent contenir des valeurs de modèle, comme à la ligne 3. Ici, "contre" n'est pas un appel de procédure, mais un constructeur de modèle. La ligne 4 montre que la correspondance entre le réel et le formel est comme une affectation ordinaire. La ligne 5 montre que l'appariement fonctionne de manière récursive. Il n'est pas nécessaire que les modèles aient des paramètres (ligne 7). Si le nom du modèle n'est pas conforme, la correspondance échoue (ligne 8). Dans chacun de ces cas (sauf le dernier), une *réception* dans un thread avec le modèle à gauche correspondrait à un *envoi dans un* autre thread avec le modèle à droite.

4.5 La programmation fonctionnelle

4.5.1 Concepts fondamentaux

La *programmation fonctionnelle* définit les sorties d'un programme comme une fonction mathématique des entrées, sans notion d'état interne, et donc sans effets secondaires.. Pour rendre la programmation fonctionnelle pratique, les langages fonctionnels fournissent un certain nombre de caractéristiques qui sont souvent absentes des langages impératifs, notamment

- Valeurs de fonctions de premier ordre et fonctions d'ordre supérieur
- Polymorphisme étendu
- Types de listes et opérateurs
- Retours de fonctions structurés
- Constructeurs (agrégats) d'objets structurés

Un langage fonctionnel pur doit fournir des agrégats complètement généraux : comme il n'y a aucun moyen de mettre à jour les objets existants, les nouveaux objets créés doivent être initialisés "d'un seul coup".

Comme Lisp était le langage fonctionnel d'origine, et qu'il est probablement encore le plus utilisé, plusieurs caractéristiques de Lisp sont couramment, bien qu'elles soient décrites de manière inexacte comme si elles concernaient la programmation fonctionnelle en général. Il s'agit notamment des caractéristiques suivantes Homogénéité des programmes et des données.

4.5.2 Language LISP

LISP (List Processing language) représente une famille de langages apparentés, qui partagent toutes le noyau commun d'idées épousées pour la première fois dans LISP 1.5. La plupart des dialectes de LISP ne sont pas purement fonctionnels (des variables sont parfois utilisées, et certaines fonctions ont des effets secondaires).

Les valeurs fondamentales manipulées par LISP sont appelées des **atomes**. Un atome est soit un nombre (entier ou réel), soit un symbole qui ressemble à un identifiant typique (comme ABC ou L10). Les atomes peuvent être structurés en **S-expressions**, qui sont définis de manière récursive comme étant soit

1. Un atome, ou
2. (S1.S2), où S1 et S2 sont des expressions S. l'exemple suivant montre quelques expressions S.

100	1
(A.B)	2
((10.AB).(XYZ.SSS))	3

Toutes les expressions S qui ne sont pas des atomes ont deux composantes : la tête (appelée, pour des raisons historiques, la voiture), et la queue (appelée, pour des raisons historiques, le cdr1). Cette définition conduit à une organisation simple de la mémoire d'exécution : les atomes numériques sont placés dans un mot informatique, les atomes symboliques sont représentés par un pointeur vers une entrée de table de symboles, et les expressions S sont représentées par une paire de pointeurs vers des atomes ou des sous-expressions.

4.5.2.1 Formulaires

L'appel (contre A B) doit-il signifier réunir les atomes A et B, ou faut-il rechercher A et B dans le tableau des symboles au cas où ils seraient des paramètres formels dans le contexte actuel ? LISP évalue tous les paramètres réels, donc A et B sont évalués en les recherchant dans le tableau des symboles. Si on veut que A et B soient traités comme des atomes plutôt que comme des identificateurs, on doit les **citer,** c'est-à-dire empêcher leur évaluation. Le programmeur peut utiliser la **citation**, appelée (**quote** arg), pour empêcher l'évaluation. La **citation** est appelée une **forme, et** non une fonction2, car elle est comprise comme un cas particulier par l'interprète LISP. Si c'était une fonction, son

paramètre serait évalué, ce qui est exactement ce que la **citation** est censée empêcher. Le code dans l'exemple suivant construit l'expression S (A.B).

```
(cons (quote A) (quote B))
```

Comme les programmeurs ont souvent besoin de citer des paramètres, LISP permet une forme abrégée de **citation** : A signifie la même chose que (**citation** A), donc (contre A 'B) construira également l'expression S comme montré dans l'exemple ci-dessus.

Pour être efficace, tout langage de programmation a besoin d'une forme de mécanisme d'évaluation conditionnelle. LISP utilise la forme **cond.** (Certains dialectes de LISP fournissent également une forme **if**.) **Cond** prend une séquence d'une ou plusieurs paires (listes de deux éléments) comme paramètres. Chaque paire est considérée à tour de rôle. Si le premier élément d'une paire est évalué à t, alors le second élément est évalué et renvoyé comme valeur de **cond** (et toutes les autres paires sont ignorées). Si la première composante est évaluée à nil (c'est-à-dire false), alors la deuxième composante est ignorée et la paire suivante est considérée. Si toutes les paires sont prises en compte, et que tous les premiers composants sont évalués à nil, alors **cond** renvoie la valeur nil.

À titre d'exemple, supposons qu'on veuille créer un prédicat qui vérifie si une liste liée à l'identifiant L contient deux éléments ou plus. L'exemple suivant montre le code.

```
(cond                          1
      ((atom L) nil)           2
      ((atom (cdr L)) nil)     3
      (t t)                    4
)                              5
```

D'abord, la ligne 2 teste si L est un atome. Si c'est le cas, c'est la liste vide (égale à zéro), qui ne contient certainement pas deux éléments ou plus. Ensuite, la ligne 3 vérifie si cdr(L) est un atome. Cdr donne la liste qui reste après avoir enlevé son premier élément. Si cdr(L) est un atome, alors la liste n'avait qu'un seul élément, et le prédicat retourne à nouveau faux. Dans tous les autres cas, la liste doit avoir au moins deux éléments, donc le prédicat renvoie vrai. Dans la plupart des cas, la dernière paire donnée à **cond** a t comme premier élément. Une telle paire représente une sorte de clause "**else",** couvrant tous les cas non inclus dans les paires précédentes.

4.5.2.2 Syntaxe de la fonction

Les programmes ainsi que les données sont représentés sous forme de listes. Autrement dit, LISP est **homonyme :** Les programmes et les données ont la même représentation. Cette propriété, que l'on trouve rarement dans les langages de programmation, permet à un programme LISP de créer ou de modifier d'autres fonctions LISP. Pour permettre aux programmes d'être représentés sous forme de listes, les invocations de fonctions LISP ne sont pas représentées sous la forme habituelle de NomFonction(arg1, arg2, ...), mais plutôt sous la forme (NomFonction arg1 arg2 ...). Par

exemple, l'expression S (10.20) peut être construite en évaluant (contre 10 20). Lorsqu'une liste est évaluée, le premier élément de la liste est recherché (dans la table des symboles d'exécution) pour trouver quelle fonction doit être exécutée. Sauf dans des cas particuliers (formes telles que **cond**), les autres éléments de la liste sont évalués et transmis à la fonction en tant que paramètres réels. La valeur calculée par le corps de la fonction est ensuite renvoyée comme valeur de la liste.

4.5.2.3 Aperçu du langage Scheme

Le langage Scheme est un des représentants du paradigme de la programmation fonctionnelle, dont le langage le plus connu est Lisp. Scheme est un « descendant » de LISp (LISt Processing).

Il se fonde sur le formalisme du lambda-calcul. En Scheme, on va représenter les termes du lambda-calcul (notamment par des listes), et l'interpréteur Scheme aura pour fonction d'évaluer ces termes. Une des particularité du langage, liée à la nature du lambda-calcul, est de ne pas différencier les données et les programmes. En d'autres termes, une liste en Scheme va pouvoir représenter à la fois une donnée (e.g. la suite des 4 premiers nombres premiers: (2 3 5 7)), ou une fonction (e.g. la fonction ajouter2: (define (ajouter2 x) (+ x 2))). En Scheme, (presque) tout est représenté sous forme de listes: ce sont les structures fondamentales du langage. Elle s'écrivent à l'aide de parenthèses, comme dans (2 3 5 7).

La plupart des implémentations Scheme utilisent un interprète qui exécute une boucle "lecture-évaluation-impression". L'interpréteur lit de manière répétée une expression d'une entrée standard (généralement tapée par l'utilisateur), évalue cette expression et imprime la valeur résultante. Si l'utilisateur tape

```
(+ 5 3)
l'interprète imprimera 8
Si l'utilisateur tape 8
l'interprète imprimera également 8
```

(Le chiffre 8 est déjà entièrement évalué.) Pour éviter au programmeur de devoir taper un programme entier mot à mot au clavier, la plupart des implémentations de Scheme fournissent une fonction de chargement qui lit (et évalue) les données d'un fichier :

```
(charger "mon_programme_de_ Scheme ")
```

Scheme (comme tous les dialectes Lisp) utilise la notation *polonaise de Cambridge* pour les expressions. Les parenthèses indiquent une application de fonction (ou dans certains cas l'utilisation d'une macro). La première expression à l'intérieur de la parenthèse gauche indique la fonction ; les autres expressions sont ses arguments. Supposons que l'utilisateur tape

```
((+ 5 3))
```

Lorsqu'il voit l'ensemble intérieur des parenthèses, l'interprète appelle la fonction +, en passant 5 et 3 comme arguments. En raison de l'ensemble extérieur des parenthèses,

il tentera alors d'appeler 8 comme une fonction à argument zéro une erreur d'exécution :

eval : 8 n'est pas une procédure.

Contrairement à la situation dans presque tous les autres langages de programmation, les parenthèses supplémentaires modifient la sémantique des programmes Lisp/Scheme.

(+ 5 3) ⇒ 7

((+ 5 3)) ⇒ *erreur*

Ici, le ⇒ signifie "évalue à". Ce symbole ne fait pas partie de la syntaxe de Scheme lui-même.

Le lambda-calcul fournit un cadre formel pour exprimer les fonctions et les calculs. Par exemple, la fonction qui calcule le carré d'un nombre réel s'écrira λx.x2. En Scheme, l'écriture reste très proche de celle du lambdacalcul: cette fonction s'écrira (lambda (x) (* x x)) (notez la position infixée de l'opérateur). Cette liste comprend trois termes: (i) la forme spéciale lambda, (ii) la liste de paramètres (x) et (iii) le corps de la fonction (* x x),qui permet de définir le « résultat » du calcul.

Pour créer une fonction dans Scheme, on évalue une *expression lambda* :

(lambda (x) (* x x)) =⇒ *fonction*

Le premier "argument" de lambda est une liste de paramètres formels pour la fonction (dans ce cas, le paramètre unique x). Les autres "arguments" (ici encore un seul) constituent le corps de la fonction. Scheme fait la distinction entre les fonctions et les *formes* dites *spéciales* (dont lambda), qui ressemblent à des fonctions mais qui ont des règles d'évaluation spéciales.

À proprement parler, seules les fonctions ont des arguments, mais nous utiliserons également la terminologie pour désigner les sous-expressions qui ressemblent à des arguments sous une forme spéciale.

Une expression lambda ne donne pas de nom à sa fonction ; cela peut être fait en utilisant let ou define. En ce sens, une expression lambda est comme les agrégats qui permettent de spécifier des tableaux ou d'enregistrer des valeurs.

Lorsqu'une fonction est appelée, l'implémentation du langage rétablit l'environnement de référencement qui était en vigueur lorsque l'expression lambda a été évaluée (comme tous les langages à portée statique et les sous-programmes imbriqués de première classe, Scheme utilise le deep binding). Il ajoute ensuite à cet environnement des liaisons pour les paramètres formels et évalue les expressions du corps de la fonction dans l'ordre. La valeur de la dernière de ces expressions (le plus souvent il n'y en *a* qu'une) devient la valeur retournée par la fonction :

```
((lambda (x) (* x x)) 3) ⇒ 9
```

Des expressions conditionnelles simples peuvent être écrites en utilisant if :

```
(si (< 2 3) 4 5) ⇒ 4
(si #f 2 3) ⇒ 3
```

4.5.2.4 Listes et numéros

Comme tous les dialectes Lisp, Scheme offre une multitude de fonctions permettant de manipuler les listes. Les trois plus importantes sont car, qui renvoie la tête d'une liste, cdr ("coulder"), qui renvoie le reste de la liste (tout ce qui suit la tête), et cons, qui joint une tête au reste d'une liste :

```
(car '(1 2 3)) ⇒ 1
(cdr '(1 2 3)) ⇒ (2 3)
(cons 1 '(2 3)) ⇒ (1 2 3)
(car '(1 2 3)) ⇒ 2
(cdr '(1 2 3)) ⇒ (3 4)
(contre 1 '(2 3)) ⇒ (1 2 3)
```

Le prédicat null? est également utile, car il détermine si son argument est la liste vide. Rappelons que la notation "(2 3 4) indique une liste *correcte*, dans laquelle le dernier élément est la liste vide :

```
(cdr '(1)) ⇒ ()
(contre 1 2) ⇒ (1 . 2) ; une liste inappropriée
```

Pour un accès rapide aux éléments arbitraires d'une séquence, Scheme fournit un type de vecteur indexé par des entiers, comme un tableau, et peut avoir des éléments de types hétérogènes, comme un enregistrement.

Scheme fournit également une multitude de fonctions numériques et logiques (booléennes) et de formes spéciales. Le manuel de langage décrit une hiérarchie de cinq types numériques : entier, rationnel, réel, complexe et nombre. Les deux derniers niveaux sont facultatifs : les implémentations peuvent choisir de ne pas fournir de nombres qui ne sont pas réels. La plupart des implémentations, mais pas toutes, utilisent des représentations de précision arbitraire à la fois des entiers et des rationnels, ces derniers étant stockés en interne sous forme de paires (numérateur, dénominateur).

4.5.2.5 Liaisons

Les noms peuvent être liés à des valeurs en introduisant une portée imbriquée :

```
(let ((a 5)
(b 6)
(square (lambda (x) (* x x)))
(plus +))
(sqrt (plus (square a) (square b)))) ⇒ 7.0
```

La forme spéciale let prend deux ou plusieurs arguments. Le premier d'entre eux est une liste de paires. Dans chaque paire, le premier élément est un nom et le second est la valeur que le nom doit représenter dans les arguments restants à laisser. Les arguments restants sont ensuite évalués dans l'ordre ; la valeur de l'ensemble de la construction est la valeur de l'argument final.

La portée des liaisons produites par let n'est que le deuxième argument :

```
(let ((a 5))
(let ((a 6)
(b a))

(+ a b))) ⇒ 11
```

Ici, b prend la valeur de l'*extérieur* a. La façon dont les noms deviennent visibles "d'un seul coup" à la fin de la liste de déclaration exclut la définition de fonctions récursives.

Pour celles-ci, on utilise letrec :

```
(letrec ((fact
(lambda (n)
(if (= n 1) 1
(* n (fact (- n 1)))))))
(fact 5)) =⇒ 120
```

Il existe également une construction let* dans laquelle les noms deviennent visibles "un à la fois" afin que les plus récents puissent utiliser les plus anciens, mais pas l'inverse. La portée du Scheme est statique. (Le Common Lisp est aussi à portée statique. La plupart des autres dialectes Lisp sont à portée dynamique). Si let et letrec permettent à l'utilisateur de créer des portées imbriquées, elles n'affectent pas la signification des noms globaux (noms connus au niveau le plus extérieur de l'interpréteur Scheme). Pour ces régimes, il existe un formulaire spécial appelé "define" qui a l'effet secondaire suivant créer un lien mondial pour un nom :

```
(define hypot
(lambda (a b)
(sqrt (+ (* a a) (* b b)))))

(hypot 5 6) ⇒ 7
```

4.5.2.6 Tests d'égalité et recherche

Le programme prévoit plusieurs fonctions de contrôle de l'égalité. Pour les comparaisons numériques, = effectue des conversions de type lorsque cela est nécessaire (par exemple, pour comparer un nombre entier et un nombre à virgule flottante). Pour une utilisation générale, eqv? Effectue une comparaison *peu profonde*, tandis que equal? effectue une comparaison *profonde* (récursive), en utilisant eqv? aux feuilles. La fonction eq? effectue également une comparaison peu profonde, et peut être moins chère que eqv? dans certaines circonstances (en particulier, eq ? n'est pas nécessaire pour détecter l'égalité de valeurs discrètes stockées dans différents endroits, bien qu'elle puisse l'être dans certaines applications).

Pour rechercher des éléments dans les listes, Scheme fournit deux ensembles de fonctions, dont chacun comporte des variantes correspondant aux trois prédicats d'égalité d'usage général. Les fonctions memq, memv et member prennent un élément et une liste comme argument, et renvoient le suffixe le plus long de la liste (le cas échéant) commençant par l'élément :

```
(memq 'z '(x y z w)) ⇒ (z w)
(memv '(z) '(x y (z) w)) ⇒ #f ; (eq? '(z) '(z)) ⇒ #f
(member '(z) '(x y (z) w)) ⇒ ((z) w) ; (equal? '(z) '(z)) ⇒ #t
```

Les fonctions memq, memv et member effectuent leurs comparaisons en utilisant respectivement eq ?, eqv ? et equal ? Elles renvoient #f si l'élément souhaité n'est pas trouvé. Il s'avère que les expressions conditionnelles de Scheme (par exemple, if) traitent tout ce qui n'est pas #f comme vrai.

4.5.2.7 Flux de contrôle et affectation

Nous avons déjà vu le formulaire spécial else. Il a un cousin nommé cond qui resembles un else. . . plus général . elsif. . . autre :

```
(cond
((< 3 2) 1)
((< 4 3) 2)
(else 3)) ⇒ 3
```

Les arguments pour cond sont des paires. Ils sont considérés dans l'ordre du premier au dernier. La valeur de l'expression globale est la valeur du deuxième élément de la première paire dans laquelle le premier élément évalue à #t. Si aucun des premiers éléments n'est évalué à #t, alors la valeur globale est #f. Le symbole else n'est autorisé que comme premier élément de la dernière paire de la construction, où il sert de sucre syntaxique pour #t. La récursivité, bien sûr, est le principal moyen de faire les choses de façon répétitive dans Scheme.

Pour les programmeurs qui souhaitent utiliser les effets secondaires, Scheme propose des constructions d'affectation, de séquencement et d'itération. L'assignation utilise la forme spéciale set ! et les fonctions set-car ! et set-cdr!

```
(let ((x 2) ; initialiser x à 2
(l '(a b))) ; initialiser l à (a b)
(set! x 3) ; attribuer à x la valeur 3
(set-car! l '(c d)) ; attribuer à la tête de l la valeur (c d)
(set-cdr! l '(e)) ; attribuer au reste de l la valeur (e)
... x ⇒ 3
... l ⇒ ((c d) e)
```

Les valeurs de retour des différentes variétés de set! dépendent de l'implémentation. Le séquençage utilise la forme spéciale begin:

```
(begin
(display "hi ")
(display "mom"))
```

4.5.3 Évaluation paresseuse

L'évaluation paresseuse nous donne l'avantage d'une évaluation d'ordre normal (ne pas évaluer les sous-expressions inutiles) tout en fonctionnant avec un facteur constant de la vitesse d'évaluation d'ordre applicatif pour les expressions dans lesquelles tout est nécessaire. L'astuce consiste à marquer chaque argument en interne avec un "mémo" qui indique sa valeur, si elle est connue. Toute tentative d'évaluation de l'argument fixe la valeur dans le mémo comme un effet secondaire, ou renvoie la valeur (sans la recalculer) si elle est déjà fixée.

L'ordre d'évaluation des expressions (et sous-expressions) n'est pas pertinent. Cette liberté d'évaluer dans n'importe quel ordre rend les langages fonctionnels particulièrement fertiles pour généraliser l'idée d'une sémantique de court-circuit. (La forme **cond** exige de veiller à ce que la première branche textuelle réussie soit prise en compte. Bien que les branches puissent être évaluées dans n'importe quel ordre, les erreurs d'exécution rencontrées lors de l'évaluation des conditions textuellement plus tard que la première branche réussie doivent être supprimées).

L'évaluation paresseuse est intéressante principalement parce qu'une évaluation stricte peut évaluer plus que ce qui est réellement nécessaire. Par exemple, si on veut calculer quel élève a obtenu la meilleure note à un examen, on peut évaluer (voiture (trier les élèves)). Les évaluateurs stricts trieront la liste complète des étudiants, puis jetteront tout sauf le premier élément. Un évaluateur paresseux ne fera que le tri nécessaire pour produire la voiture de la liste, puis s'arrêtera (car il n'y a pas de référence au cdr de la liste triée). Or, le tri visant à trouver l'élément maximum est une approche inefficace au départ, et nous ne pouvons pas reprocher aux évaluateurs stricts

leur inefficacité lorsque l'algorithme lui-même est si mauvais. Cependant, une évaluation paresseuse parvient à sauver cette approche inefficace (mais très claire) et à la rendre plus efficace.

Une évaluation paresseuse est plus appropriée pour un tel problème. Elle suit unu évaluation scheme avant-première, en reportant l'évaluation des paramètres jusqu'à ce que cela soit nécessaire. C'est-à-dire, dans une invocation imbriquée, comme celle de l'exemple suivant :

```
(foo (bar L) (baz (rag L)))
```

foo est invoqué devant bar ou baz, et en fait ils peuvent ne jamais être invoqués du tout, si, par exemple, foo ignore ses paramètres. Si foo a besoin d'évaluer son deuxième paramètre, baz est invoqué, mais pas rag, à moins que baz lui-même n'en ait besoin. De plus, une fois qu'une fonction a été invoquée, le résultat qu'elle renvoie peut ne pas être entièrement calculé. Par exemple, le corps de bar peut indiquer qu'elle renvoie (contre 1 (frob L)). Il retournera une cellule cons (allouée à partir du tas) avec 1 dans la voiture et une suspension dans le cdr ; la suspension indique qu'une frob doit être invoquée sur L pour obtenir une valeur. Cette suspension ne peut jamais être activée.

L'algorithme d'évaluation paresseuse est le suivant :

1. Pour évaluer une liste, il faut en faire une suspension (en combinant le nom de la fonction, les paramètres, qui ne sont pas encore à évaluer, et l'environnement de référencement).
2. Pour évaluer une suspension, faites une suspension à partir de chacun de ses paramètres et invoquez sa fonction dans son environnement de référencement.
3. Pour évaluer l'invocation d'une contre-attaque, créez une nouvelle cellule de contre-attaque dans le tas et initialisez sa voiture et son cdr aux paramètres, qui sont laissés en suspens.
4. Pour évaluer une fonction booléenne primitive telle que null ou eq, n'évaluez le(s) paramètre(s) que dans la mesure où cela est nécessaire. Chaque fonction primitive a sa propre méthode d'évaluation paresseuse.

4.6 La programmation logique

Les systèmes de programmation logique permettent au programmeur d'énoncer une série d'*axiomes à* partir desquels les théorèmes peuvent être prouvés. L'utilisateur d'un programme logique énonce un théorème, ou *objectif,* et l'implémentation du langage tente de trouver une collection d'axiomes et d'étapes d'inférence (y compris le choix des valeurs des variables) qui, ensemble, impliquent l'objectif. Parmi les différents langages logiques existants, Prolog est de loin le plus utilisé.

Dans presque tous les langages logiques, les axiomes sont écrits sous une forme standard appelée *clause Horn*. Une clause Horn se compose d'une *tête*, ou terme *consécutif H,* et d'un *corps composé des* termes *Bi* :

$$H \leftarrow B1, B2, \ldots, Bn$$

La sémantique de cette affirmation est que lorsque les *Bi* sont tous vrais, on peut en déduire que *H est vrai* aussi. En lisant à voix haute, on dit "*H*, si B1, B2, ... et *Bn*". Les clauses de Horn peuvent être utilisées pour saisir la plupart des énoncés logiques, mais pas tous.

Afin d'obtenir de nouvelles déclarations, un système de programmation logique combine les déclarations existantes, en annulant les termes similaires, par un processus appelé *résolution*. Si nous savons que *A* et *B* impliquent *C,* par exemple, et que *C implique D,* nous pouvons en déduire que *A* et *B* impliquent *D* :

$$C \leftarrow A, B$$

$$D \leftarrow C$$

$$D \leftarrow A, B$$

En général, des termes tels que *A*, *B*, *C* et *D* peuvent être constitués non seulement de constantes, mais aussi de *prédicats* appliqués à des *atomes* ou à des *variables* :
Pendant la résolution, les variables libres peuvent acquérir des valeurs par *unification* avec des expressions en termes correspondants, tout comme les variables acquièrent des types en ML :

flowery(*X*) ← rainy(*X*)
rainy(Rochester)
flowery(Rochester)

fleuri(*X*) ← pluvieux(*X*) pluvieux(Rochester) fleuri(Rochester)

4.6.1 Syntaxe

Les programmes logiques sont des ensembles de formules logiques d'une forme particulière. Nous commençons donc par quelques notions de base pour définir la syntaxe.

La logique qui nous intéresse ici est la logique du premier ordre ; elle est également appelée *calcul des prédicats*. La terminologie fait référence au fait que les symboles sont utilisés pour exprimer (ou, dans une terminologie plus ancienne, pour "prédicat") les propriétés des éléments d'un domaine de discours fixe, *D* . Des logiques plus expressives (deuxième, troisième, etc., ordre) permettent également de prédicier des

objets plus complexes tels que des ensembles et des fonctions sur D (deuxième ordre), des ensembles de fonctions (troisième ordre), etc., en plus des éléments de D .

4.6.1.1 Les programmes logiques

Une formule de logique du premier ordre peut avoir une structure très complexe qui détermine souvent l'effort nécessaire pour trouver une preuve. Dans la démonstration automatique du théorème, et aussi dans la programmation logique, des classes particulières de formules, appelées *clauses,* ont été identifiées qui se prêtent à une manipulation plus efficace, notamment en utilisant une règle d'inférence spéciale appelée *résolution*. Une version restreinte du concept de clause, appelée clauses *définies, ainsi que des* formes restreintes de résolution (*résolution SLD*). Grâce à elles, la procédure de recherche d'une preuve est non seulement particulièrement simple mais permet également de calculer explicitement les valeurs des variables nécessaires à la preuve. Ces valeurs peuvent être considérées comme le résultat du calcul, ce qui donne lieu à un modèle de calcul intéressant basé sur la déduction logique. Nous verrons ce modèle plus en détail ci-dessous, pour l'instant nous allons nous concentrer sur les aspects syntaxiques.

Soit H, A1, . . . , An *étant des* formules atomiques. Une clause définie (pour nous, simplement une "clause") est une formule de la forme :

$$H : -\text{A1}, \ldots, An.$$

Si $n = 0$, on dit que la clause est une *unité,* ou un fait, et le symbole : - est omis (mais pas le point final). Un programme logique est un ensemble de clauses, tandis qu'un programme PROLOG pur est une séquence de clauses. Une requête (ou objectif) est une séquence d'atomes A1, , An.

Clarifions certains points concernant cette définition. Tout d'abord, le symbole : - que nous n'avons pas inclus dans notre alphabet, est simplement un symbole indiquant une implication (inversée) (←) et est le même que celui souvent rencontré dans les vrais langages de logique. Du point de vue logique, les virgules d'une clause ou d'une requête doivent être interprétées comme une conjonction logique. La notation "H : −A1, An." est donc une abréviation de "H ←A1∧ - -∧ An. "Notez que le point fait partie de la notation et qu'il est important car il indique à un interprète ou à un compilateur potentiel que la clause sur laquelle il travaille est terminée.

La partie à gauche de : - s'appelle la *tête de* la clause, tandis que celle de droite s'appelle le *corps*. Un fait est donc une clause dont le corps est vide. Un programme est un ensemble de clauses dans le cas du formalisme théorique. Dans le cas de PROLOG, en revanche, un programme est considéré comme une séquence car, comme nous le verrons, l'ordre des clauses est pertinent.

4.6.2 Langage Prolog

Un interprète Prolog fonctionne dans le contexte d'une *base de données* de *clauses* (clauses de Horn) qui sont supposées être vraies. Chaque clause est composée de *termes*, qui peuvent être des constantes, des variables ou des *structures*. Une constante est soit un atome, soit un nombre. Une structure peut être considérée soit comme un prédicat logique, soit comme une structure de données.

Les atomes dans Prolog sont similaires aux symboles dans Lisp. Lexicalement, un atome ressemble à un identificateur commençant par une lettre minuscule, une séquence de caractères de "ponctuation" ou une chaîne de caractères entre guillemets :

```
foo my_Const + "Hi, Mom
```

Les nombres ressemblent aux nombres entiers et aux constantes à virgule flottante des autres langages de programmation. Une variable ressemble à un identificateur commençant par une lettre majuscule :

```
Foo My_var X
```

Les variables peuvent être *instanciées à des* valeurs arbitraires (c'est-à-dire qu'elles peuvent prendre des valeurs arbitraires) au moment de l'exécution, à la suite de l'unification. La portée de chaque variable est limitée à la clause dans laquelle elle apparaît. Il n'y a pas de déclaration, la vérification de type ne se produit que lorsqu'un programme tente d'utiliser une valeur d'une manière particulière au moment de l'exécution.

Les structures sont constituées d'un atome appelé le *foncteur* et d'une liste d'arguments :

```
rainy(rochester)
teaches(scott, cs254)
bin_tree(foo, bin_tree(bar, glarch))
```

Prologue exige que la parenthèse d'ouverture vienne immédiatement après le foncteur, sans espace intermédiaire. Les arguments peuvent être des termes arbitraires : constantes, variables ou structures (imbriquées). En interne, une implémentation de Prolog peut représenter une structure en utilisant des cellules de contre-réflexion de type Lisp. Conceptuellement, le programmeur peut préférer considérer certaines structures (par exemple, Rainy) comme des prédicats logiques. Nous utilisons le terme "prédicat" pour désigner la combinaison d'un foncteur et d'une "arité" (nombre d'arguments).

Les clauses contenues dans une base de données Prolog peuvent être classées comme des *faits* ou des *règles*, dont chacune se termine par un délai. Un fait est une clause "Horn" sans côté droit. Il semble que un seul terme (le symbole d'implication est implicite) :

```
rainy (rochester).
```

Une règle a un côté droit :

```
enneigé(X) :- pluvieux(X), froid(X).
```

Les variables qui apparaissent dans la tête d'une clause de Horn sont universellement quantifiées : pour tout X, X est enneigé si X est pluvieux et X est froid.
Il est également possible d'écrire une clause avec un côté gauche vide. Une telle clause est appelée une *requête,* ou un *objectif*. Les requêtes n'apparaissent pas dans les programmes Prolog. On construit plutôt une base de données de faits et de règles, puis on lance l'exécution en donnant au Interpréteur Prolog (ou le programme Prolog compilé) une requête à laquelle il faut répondre (c'est-à-dire un objectif à prouver).

Dans la plupart des implémentations de Prolog, les requêtes sont saisies avec une version spéciale ?- du symbole d'implication. Si nous devions taper ce qui suit :

```
rainy(seattle).
rainy(rochester).
?- rainy(C).
```

L'interprète Prolog répondrait par

```
C = seattle
```

Bien sûr, C = rochester serait également une réponse valable, mais Prolog trouvera d'abord Seattle, car il est le premier dans la base de données. (La dépendance de la commande est l'une des façons dont Prolog s'écarte de la logique pure ; nous abordons cette question plus loin). Si nous voulons trouver toutes les solutions possibles, nous pouvons demander à l'interprète de continuer en tapant un point-virgule :

```
C = Seattle ;
C = rochester
```

Si nous tapons un autre point-virgule, l'interprète indiquera qu'aucune autre solution n'est possible :

```
C = Seattle ;
C = rochester ;
Non
```

De même, étant donné

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X) :- rainy(X), cold(X).
```

la requête

```
?- snowy(C).
```

n'apportera qu'une seule solution.

4.6.2.1 Unification

L'unification des structures dans Prolog s'apparente beaucoup à l'unification par ML des types de paramètres formels et réels. Un paramètre formel de type liste int * 'b' , par exemple, s'unifiera avec un paramètre réel de type liste a * real dans ML en instanciant 'a' à int et 'b à real.

Dans Prolog, l'égalité est définie en termes d'"unifiabilité". L'objectif =(A, B) réussit si et seulement si A et B peuvent être unifiés. Pour des raisons de commodité, l'objectif peut être écrit comme A= B ; la notation infixe est simplement le sucre syntaxique. En respectant les règles ci-dessus pour l'égalité et l'unification, nous avons

```
?- a = a.
Yes              % constante s'unifie avec elle-même
?- a = b.
No               % mais pas avec une autre constante
?- foo(a, b) = foo(a, b).
Yes              % les structures sont récursivement identiques
?- X = a.
X = a ;          % variable unifie avec constante
No               % une seule fois
?- foo(a, b) = foo(X, b).
X = a ;          % arguments doivent s'unifier
No               % seulement une possibilité
```

Il est possible d'unifier deux variables sans les instancier. Si l'on tape

```
?- A = B.
```

l'interprète répondra simplement

```
A = B
```

Si, toutefois, nous tapons

```
?- A = B, A = a, B = Y.
```

(en unifiant A et B avant de lier a à A), l'interprète répondra

```
A = a
B = a
Y = a
```

Dans le même ordre d'idées, supposons que l'on nous donne les règles suivantes :

```
takes_lab(S) :- takes(S, C), has_lab(C).
has_lab(D) :- meets_in(D, R), is_lab(R).
```

(S prend une classe de lab si S prend C et C est une classe de lab. En outre, D est une classe de lab si D se réunit dans la salle R et R est un lab). Une tentative de résolution de

ces règles unifieront la tête du second avec le second terme dans le corps du premier, ce qui entraînera l'unification de C et D, même si aucun des deux n'est instancié.

4.6.2.2 Arithmétique

Les opérateurs arithmétiques habituels sont disponibles dans Prolog, mais ils jouent le rôle de prédicats, et non de fonctions. Ainsi, +(2, 3), qui peut aussi s'écrire 2 + 3, est une structure à deux arguments, et non un appel de fonction. En particulier, il ne s'unifie pas avec 5 :

```
?- (2 + 3) = 5.
Non
```

Pour traiter l'arithmétique, Prolog fournit un prédicat intégré, c'est-à-dire, qui unifie son premier argument avec la valeur arithmétique de son deuxième argument :

```
?- est (X, 1+2).
X = 3
?- X est 1+2.
X = 3              % infix est également correct
?- 1+2 est 4-1.
Non                % premier argument (1+2) n'est déjà instancié
?- X est Y.
ERREUR             % deuxième argument (Y) doit déjà être instancié
?- Y est 1+2, X est Y.
Y = 3
X = 3              % Y est instancié au moment où il est nécessaire
```

4.6.2.3 Listes

Tout comme la vérification de l'égalité, la manipulation des listes est une opération suffisamment courante dans Prolog pour justifier sa propre notation. La construction [a, b, c] est un sucre syntaxique pour la structure .(a, .(b, .(c, [])), où [] est la liste vide et . est un prédicat intégré de type contre.

Prolog ajoute cependant une commodité supplémentaire : une barre verticale optionnelle qui délimite la "queue" de la liste. En utilisant cette notation, [a, b, c] pourrait être exprimé comme [a | [b, c]], [a, b | [c]], ou [a, b, c | []]. La notation à barres verticales est particulièrement pratique lorsque la queue de la liste est une variable :

```
member(X, [X | _]).
member(X, [_ | T]) :- member(X, T).
sorted([]). % liste vide est triée
sorted([_]). % singleton est trié
sorted([A, B | T]) :- A =< B, sorted([B | T]).
    % la liste composée est triée si les deux premiers éléments sont dans
    % l'ordre et le reste de la liste (après le premier élément) est trié
```

Ici =< est un prédicat intégré qui fonctionne sur les nombres. Le soulignement est un espace réservé pour une variable qui n'est nécessaire nulle part ailleurs dans la clause. Notez que [a, b | c] est la liste *impropre* .(a, .(b, c)). La séquence de jetons [a | b, c] est syntaxiquement incorrecte.

L'un des aspects intéressants de la résolution Prolog est qu'elle ne fait pas de distinction générale entre les arguments d'"entrée" et de "sortie" dans les fonctions, les prédicats et les règles bidirectionnelles (il existe certaines exceptions, comme le prédicat is décrit dans la sous-section suivante). Ainsi, étant donné que

```
append([], A, A).
append([H | T], A, [H | L]) :- append(T, A, L).
```

nous pouvons taper

```
?- append([a, b, c], [d, e], L).
L = [a, b, c, d, e]
?- append(X, [d, e], [a, b, c, d, e]).
X = [a, b, c]
?- append([a, b, c], Y, [a, b, c, d, e]).
Y = [d, e]
```

Cet exemple met en évidence la différence entre les fonctions et les prédicats de Prolog. Les premières ont une notion claire des intrants (arguments) et des extrants (résultats) ; les seconds n'en ont pas. Dans un langage impératif ou fonctionnel, nous appliquons les fonctions aux arguments pour générer des résultats. Dans un langage logique, nous recherchons les valeurs pour lesquelles un prédicat est vrai (tous les langages logiques ne sont pas aussi flexibles).

4.6.2.4 Exemple étendu : Tic-Tac-Toe

Le tic-tac-toe dans Prolog et l'ordre des termes dans la partie droite peuvent affecter à la fois l'efficacité d'un programme Prolog et sa capacité à se terminer. L'ordre des termes permet également au programmeur Prolog d'indiquer que certaines résolutions sont *préférables* et doivent être envisagées avant d'autres options de "repli". Prenons, par exemple, le problème du passage au tic-tac-toe. (Le tic-tac-toe est un jeu qui se joue sur une grille de 3 × 3 cases. Deux joueurs, X et O, placent à tour de rôle des marqueurs dans les cases vides.

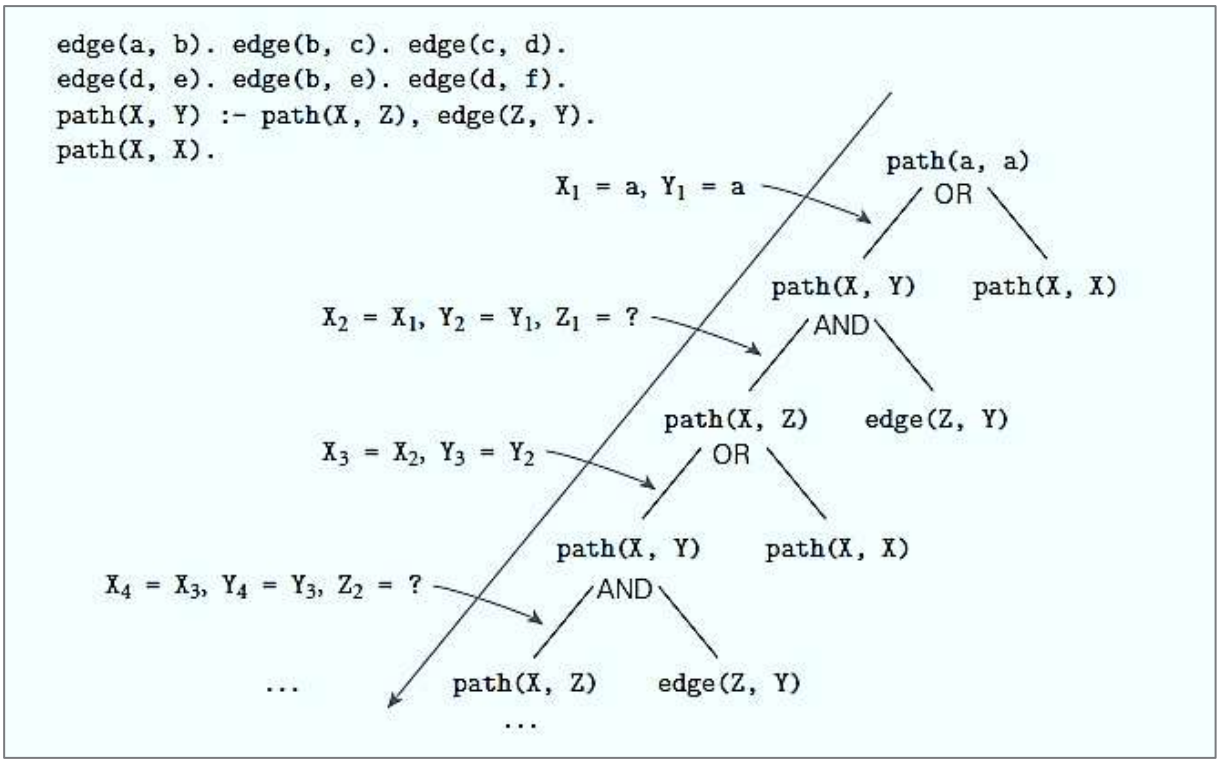

Figure 4.5 : *Régression infinie en Prolog.*

Dans cette figure, même une simple requête comme ?- path(a, a) ne se terminera jamais : l'interprète ne trouvera jamais la branche triviale.

Un joueur gagne s'il place trois marqueurs à la suite, horizontalement, verticalement ou en diagonale). Numérotons les carrés de 1 à 9 dans l'ordre des rangées principales. De plus, utilisons le fait x(n) du Prolog pour indiquer que le joueur X a placé un marqueur dans la case *n*, et o(m) pour indiquer que le joueur O a placé un marqueur dans la case *m*. Pour simplifier, supposons que l'ordinateur est le joueur X, et que c'est le tour de X de jouer. Nous aimerions pouvoir émettre une requête ?- coup(A) qui amènera l'interprète du Prolog à choisir une bonne case A que l'ordinateur occupera ensuite.

Il est clair que nous devons être capables de dire si trois carrés donnés se trouvent dans une rangée. Un de l'exprimer est :

```
ordered_line(1, 2, 3). ordered_line(4, 5, 6).
ordered_line(7, 8, 9). ordered_line(1, 4, 7).
ordered_line(2, 5, 8). ordered_line(3, 6, 9).
ordered_line(1, 5, 9). ordered_line(3, 5, 7).
line(A, B, C) :- ordered_line(A, B, C).
line(A, B, C) :- ordered_line(A, C, B).
line(A, B, C) :- ordered_line(B, A, C).
line(A, B, C) :- ordered_line(B, C, A).
line(A, B, C) :- ordered_line(C, A, B).
line(A, B, C) :- ordered_line(C, B, A).
```

Il est facile de prouver qu'il n'y a pas de stratégie gagnante pour le morpion : chaque joueur peut forcer un match nul. Supposons cependant que notre programme joue contre un adversaire moins que parfait. Notre tâche consiste alors à ne jamais perdre et à maximiser nos chances de gagner si notre adversaire commet une erreur. Les règles suivantes fonctionnent bien.

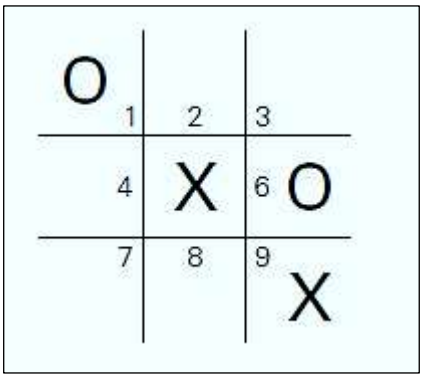

Figure 4.6 : *Une "scission" en tac-tac-toe.*

Si X prend la case centrale inférieure (case 8), aucun mouvement futur de O ne pourra empêcher X de gagner la partie - O ne peut pas bloquer à la fois la ligne 2-5-8 et la ligne 7-8-9.

```
move(A) :- good(A), empty(A).
full(A) :- x(A).
full(A) :- o(A).
empty(A) :- \+(full(A)).
% strategy:
good(A) :- win(A). good(A) :- block_win(A).
good(A) :- split(A). good(A) :- strong_build(A).
good(A) :- weak_build(A).
```

La règle initiale indique que l'on peut satisfaire le coup de but(A) en choisissant une bonne case vide. Le \+ est un prédicat intégré qui réussit si son argument (un but) ne peut être prouvé. La case n est vide si nous ne pouvons pas prouver qu'elle est pleine, c'est-à-dire si ni x(n) ni o(n) ne figurent dans la base de données.
La clé de la stratégie réside dans l'ordre des cinq dernières règles. Notre premier choix est de gagner :

```
win(A) :- x(B), x(C), ligne(A, B, C).
```

Notre deuxième choix est d'empêcher notre adversaire de gagner :

```
block_win(A) :- o(B), o(C), ligne(A, B, C).
```

Notre troisième choix est de créer une "scission" - une situation dans laquelle notre adversaire ne peut pas nous empêcher de gagner au prochain coup (voir figure 4.6) :

```
split(A) :- x(B), x(C), different(B, C),
line(A, B, D), line(A, C, E), empty(D), empty(E).
same(A, A).
different(A, B) :- \+(same(A, B)).
```

4.6.2.5 Flux de contrôle impératif

Prolog fournit au programmeur plusieurs fonctions de contrôle-flux explicites. La plus importante de ces fonctionnalités est connue sous le nom de "*cut*".
La coupure est un prédicat à argument zéro écrit comme un point d'exclamation : ! En tant que sous-objectif, il réussit toujours, mais avec un effet secondaire crucial : il engage l'interprète à faire tous les choix qui ont été faits depuis l'unification de l'objectif parent avec le côté gauche de la règle actuelle, y compris le choix de cette unification elle-même. Rappelons par exemple notre définition de l'appartenance à une liste :

```
membre(X, [X | _]).
member(X, [_ | T]) :- membre(X, T).
```

Si un atome donné a apparaît *n* fois dans la liste L, alors l'objectif ?- membre(a, L) peut réussir *n* fois. Ces succès "supplémentaires" ne sont pas toujours appropriés. Ils peuvent conduire à des calculs inutiles, en particulier pour les longues listes, lorsque le membre est suivi d'un objectif qui peut échouer :

```
prime_candidate(X) :- membre(X, candidats), prime(X).
```

Supposons que le calcul du nombre premier (X) soit coûteux. Pour déterminer si a est un candidat de premier ordre, nous vérifions d'abord s'il fait partie de la liste des candidats, puis nous vérifions s'il est de premier ordre. Si prime(a) échoue, Prolog fera marche arrière et tentera de satisfaire à nouveau le membre(a, les candidats). Si a figure plus d'une fois sur la liste des candidats, le sous-objectif réussira à nouveau, ce qui entraînera le réexamen du sous-objectif prime(a), même si ce sous-objectif est voué à l'échec. Nous pouvons gagner un temps considérable en interrompant toutes les autres recherches de a après que le premier ait été trouvé :

```
member(X, [X |  ]) :- !.
member(X, [_ | T]) :- membre(X, T).
```

La coupure sur le côté droit de la première règle dit que si X est la tête de L, nous ne devons pas essayer d'unifier le membre (X, L) avec le côté gauche de la deuxième règle ; la coupure nous engage à la première règle.

En principe, il est possible de remplacer toutes les utilisations de la coupe par des utilisations de \+ - pour limiter la coupe à l'implémentation de \+. Cela permet souvent de rendre un programme plus lisible. Cependant, comme nous l'avons vu, cela le rend souvent moins efficace.

4.7 Les scriptes

Les langages de scriptes modernes ont deux principaux groupes d'ancêtres. Dans l'un de ces ensembles se trouvent les interpréteurs de commandes ou "shells" de l'informatique traditionnelle par lots et "terminaux" (ligne de commande). Dans l'autre, on trouve divers outils de traitement de texte et de génération de rapports. Les exemples du premier ensemble comprennent le JCL d'IBM, l'interpréteur de commandes MS-DOS et les familles de shells sh et csh d'Unix. Les exemples du second ensemble comprennent le RPG d'IBM et les familles sed et awk d'Unix.

Certains auteurs réservent le terme de "scripting" aux langages de colle utilisés pour coordonner plusieurs programmes. Dans l'usage courant, cependant, le scripting est un concept plus large et plus vague. Il inclut clairement les scripts web. Pour la plupart des auteurs, il inclut également les *langages d'extension*.

De nombreux lecteurs seront familiarisés avec les "macros" de base visuelles de Microsoft Office et des applications connexes. D'autres peuvent être familiers avec le langage d'extension basé sur Lisp de l'éditeur de texte emacs. Plusieurs langages, dont Tcl, Rexx, Python et les dialectes Guile et Elk de Scheme, ont des implémentations conçues de manière à pouvoir être intégrées dans un programme plus vaste et utilisées pour étendre ses fonctionnalités.

4.7.1 Caractéristiques communes

Bien qu'il soit difficile de définir précisément les langages de script, ils ont en général plusieurs caractéristiques communes.

Utilisation par lots et interactive. Quelques langages de script (notamment Perl) ont un compilateur qui insiste sur la lecture de l'ensemble du programme source avant qu'il ne produise une sortie. La plupart des autres langages, cependant, sont prêts à compiler ou à interpréter leur entrée ligne par ligne. Python, Rexx, Tcl, Guile et (avec un court script d'aide) Ruby acceptent toutes les commandes du clavier.

Économie d'expression. Pour permettre à la fois un développement rapide et une utilisation interactive, les langages de script ont tendance à exiger un minimum de "passe-partout". Certains font un usage intensif de la ponctuation et d'identificateurs très courts (Perl est connu pour cela), tandis que d'autres (par exemple, Tcl et AppleScript) ont tendance à être plus "à l'anglaise", avec beaucoup de mots et peu de ponctuation. Tous tentent d'éviter les déclarations extensives et la structure de haut niveau communes aux langages conventionnelles. Où un programme simple ressemble à ceci en Java, et comme ça en ADA,

```
class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

et comme ça en Ada,,

```
with ada.text_IO; use ada.text_IO;
procedure hello is
begin
      put_line("Hello, world!");
end hello;
```

en Perl, Python ou Ruby c'est simplement

```
print "Hello, world!\n"
```

Absence de déclarations ; règles de cadrage simples. La plupart des langages de script renoncent aux déclarations et prévoient des règles simples pour régir la portée des noms. Dans certains langages (par exemple Perl), tout est global par défaut ; des déclarations optionnelles peuvent être utilisées pour limiter une variable à une portée imbriquée. Dans d'autres langages (par exemple, Tcl et PHP), tout est local par défaut ; les globals doivent être explicitement importés. Python adopte la règle intéressante selon laquelle toute variable à laquelle une valeur est attribuée est locale au bloc dans lequel l'attribution apparaît. Une syntaxe spéciale est nécessaire pour assigner à une variable un périmètre environnant.

Typage dynamique flexible. En raison de l'absence de déclarations, la plupart des langages de script sont typés dynamiquement. Dans certains (par exemple, PHP, Ruby et Python), le type d'une variable est vérifié immédiatement avant son utilisation. Dans d'autres (par exemple Perl et Tcl), une variable sera interprétée différemment selon le contexte. En Perl, par exemple, le programme

```
$a = "4";
print $a . 3 . "\n"; # '.' Est une concaténation
print $a + 3 . "\n"; # '+' Est un ajout
```

ecrira

```
43
7
```

Cette interprétation contextuelle est similaire à la coercition, sauf qu'il n'y a pas nécessairement une notion de type "naturel" à partir de laquelle un objet doit être converti ; les diverses interprétations possibles peuvent toutes être également "naturelles".

Types de données de haut niveau. Les types de données de haut niveau comme les ensembles, les sacs, les dictionnaires, les listes et les tuples sont de plus en plus courants dans les bibliothèques standard des langages de programmation conventionnels. Quelques langages (notamment le C++) permettent aux utilisateurs de redéfinir les opérateurs d'infixation standard pour rendre ces types aussi faciles à utiliser que les types plus primitifs, centrés sur le matériel. Les langages de script vont plus loin en intégrant des types de haut niveau dans la syntaxe et la sémantique du langage lui-

même. Dans la plupart des langages de script, par exemple, il est courant d'avoir un "tableau" qui est indexé par des chaînes de caractères, avec une implémentation sous-jacente basée sur des tables de hachage. Le stockage est invariablement un ramassage d'ordures.

Accès facile aux installations du système. La plupart des langages de programmation permettent de demander au système d'exploitation sous-jacent d'exécuter un autre programme, ou d'effectuer directement certaines opérations. Dans les langages de script, cependant, ces demandes sont beaucoup plus fondamentales et bénéficient d'un support beaucoup plus direct. Perl, par exemple, fournit bien plus de 100 commandes intégrées qui accèdent aux fonctions du système d'exploitation pour l'entrée et la sortie, la manipulation de fichiers et de répertoires, la gestion des processus, l'accès aux bases de données, les sockets, la communication et la synchronisation interprocessus, la protection et l'autorisation, l'horloge et la communication réseau. Ces commandes intégrées sont généralement un peu plus faciles à utiliser que les appels de bibliothèque correspondants dans des langages comme le C.

La mise en correspondance sophistiquée des modèles et la manipulation des cordes. Conformément à leur ascendance de traitement de texte et de génération de rapports, et pour faciliter la manipulation des entrées et sorties de texte pour les programmes externes, les langages de script ont tendance à avoir des facilités extraordinairement riches pour la correspondance de modèles, la recherche et la manipulation de chaînes. Généralement, ils sont basés sur des *expressions régulières étendues*.

4.7.2 Création de scripts pour le World Wide Web

Une grande partie du contenu du World Wide Web - en particulier le contenu visible par les moteurs de recherche est statique : des pages qui changent rarement, voire jamais. Du point de vue des langages de programmation, la simple lecture d'un enregistrement audio ou vidéo n'est pas particulièrement intéressante. Nous nous concentrons ici sur le contenu généré à la volée par un programme d'un script associé à un URI (uniform resource identifier) Internet. Supposons que nous tapions un URI dans un navigateur sur une machine cliente, et que le navigateur envoie une requête au serveur web approprié. Si le contenu est créé dynamiquement, une première question évidente se pose : le script qui le crée s'exécute-t-il sur le serveur ou la machine cliente ? Ces options sont respectivement connues sous le nom de "script web *côté serveur*" et "script web *côté client*". *Les* scripts côté serveur sont généralement utilisés lorsque le fournisseur de services veut garder un contrôle total sur le contenu de la page, mais ne peut (ou ne veut) pas créer le contenu à l'avance.

4.7.2.1 Scripts CGI

Le mécanisme original pour les scripts web côté serveur est l'interface de passerelle commune (Common Gateway Interface - CGI). Un script CGI est un programme exécutable résidant dans un répertoire spécial connu du programme du serveur web.

Lorsqu'un client demande l'URI correspondant à un tel programme, le serveur exécute le programme et renvoie sa sortie au client. Naturellement, cette sortie doit être quelque chose que le navigateur comprendra, typiquement du HTML.

Les scripts CGI peuvent être écrits dans n'importe quel langage disponible sur la machine du serveur, bien que Perl soit particulièrement populaire : ses mécanismes de gestion des chaînes et de "colle" sont parfaitement adaptés à la génération de HTML, et il était déjà largement disponible pendant les premières années du Web. Pour prendre un exemple simple, bien qu'un peu artificiel, supposons que nous aimerions pouvoir surveiller l'état d'une machine serveur partagée par une certaine communauté d'utilisateurs.

Les scripts CGI sont couramment utilisés pour traiter les formulaires en ligne. Un exemple simple est présenté à la *figure 4.7*. L'élément FORM du fichier HTML spécifie l'URI du script CGI, qui est invoqué lorsque l'utilisateur appuie sur le bouton Submit. Les valeurs précédemment saisies dans les champs INPUT sont transmises au script soit en tant que partie arrière de l'URI (pour un formulaire de type get), soit dans le flux de saisie standard (pour un formulaire de type post, illustré ici). Avec l'une ou l'autre méthode, nous pouvons accéder aux valeurs en utilisant la routine param de la bibliothèque Perl CGI standard, chargée au début de notre script comme il est présenté dans la figure 4.8

```
<HTML>
<HEAD>
<TITLE>Adder</TITLE>
</HEAD>
<BODY>
<FORM action="/cgi-bin/add.perl" method="post">
<P><INPUT name="argA" size=3>First addend<BR>
   <INPUT name="argB" size=3>Second addend
<P><INPUT type="submit">
</FORM>
</BODY>
</HTML>
```

Adder

12 First addend

34 Second addend

Submit

```
#!/usr/bin/perl

use CGI qw(:standard);      # provides access to CGI input fields
$argA = param("argA");  $argB = param("argB");  $sum = $argA + $argB;

print "Content-type: text/html\n\n";

print "<HTML>\n<HEAD>\n<TITLE>Sum</TITLE>\n</HEAD>\n<BODY>\n";
print "<P>$argA plus $argB is $sum";
print "</BODY>\n</HTML>\n";
```

```
<HTML>
<HEAD>
<TITLE>Sum</TITLE>
</HEAD>
<BODY>
<P>12 plus 34 is 46</BODY>
</HTML>
```

Sum

12 plus 34 is 46

Figure 4.7 : Un formulaire CGI interactif

La source de la page web originale est indiquée en haut à gauche, avec la page rendue à droite. L'utilisateur a saisi 12 et 34 dans les champs de texte. Lorsque le bouton "Submit" est enfoncé, le navigateur du client envoie une requête au serveur pour l'URI /cgi-bin/add.perl. Les valeurs 12 et 13 sont contenues dans la requête. Le script Perl, présenté au milieu, utilise ces valeurs pour générer une nouvelle page web, présentée en HTML en bas à gauche, avec la page rendue à droite.

```
#!/usr/bin/perl

print "Content-type: text/html\n\n";

$host = `hostname`; chop $host;
print "<HTML>\n<HEAD>\n<TITLE>Status of ", $host,
      "</TITLE>\n</HEAD>\n<BODY>\n";
print "<H1>", $host, "</H1>\n";
print "<PRE>\n", `uptime`, "\n", `who`;
print "</PRE>\n</BODY>\n</HTML>\n";
```

Figure 4.8 : Un simple script CGI en Perl.

4.7.2.2 Scripts intégrés côté client

Si les scripts intégrés côté serveur sont généralement plus rapides que les scripts CGI(Common Gateway Interface) au moins lorsque les coûts de démarrage prédominent, la communication sur Internet est encore trop lente pour des pages véritablement interactives. Si nous voulons que le comportement ou l'apparence de la page change lorsque l'utilisateur déplace la souris, clique, tape ou cache ou expose des fenêtres, nous devons vraiment exécuter une sorte de script sur la machine du client.
Parce qu'ils s'exécutent sur le site du concepteur du site web, les scripts CGI et, dans une moindre mesure, les scripts côté serveur intégrables peuvent être écrits dans de nombreux langages différents. Tout ce que le client voit est du HTML standard. Les scripts côté client, en revanche, nécessitent un interpréteur sur la machine du client. Par conséquent, il existe une forte incitation à la convergence des langages de script côté client : la plupart des concepteurs veulent que leurs pages soient visibles par un public aussi large que possible. Alors que Visual Basic est largement utilisé dans des organisations spécifiques, où tous les clients concernés sont connus pour utiliser Internet Explorer, les pages destinées au grand public utilisent presque toujours JavaScript pour les fonctions interactives.

4.7.2.3 Scripts intégrés côté serveur

La plupart des serveurs web offrent désormais un mécanisme de "chargement de modules" qui permet d'intégrer des interprètes pour un ou plusieurs langages de script dans le serveur lui-même. Les scripts dans la ou les langages pris en charge peuvent alors être intégrés dans des pages web "ordinaires". Le serveur web interprète ces

scripts directement, sans lancer un programme externe. Il remplace ensuite les scripts par le résultat qu'ils produisent, avant d'envoyer la page au client. Les clients n'ont aucun moyen de savoir que ces scripts existent.

L'équivalent PHP de la figure 4.8 apparaît dans la figure 4.9. La plupart du texte de cette figure est du HTML standard. Le code PHP est intégré entre les délimiteurs <? php et ?>. Ces délimiteurs ne sont pas eux-mêmes du HTML ; ils indiquent plutôt une *instruction de traitement* qui doit être exécutée par l'interpréteur PHP pour générer du texte de remplacement. Les parties "passe-partout" de la page peuvent ainsi apparaître textuellement ; elles n'ont pas besoin d'être générées par des commandes d'impression (Perl) ou d'écho (PHP). Notez que les différents fragments de script font partie d'un seul et même programme. La variable $host, par exemple, est définie dans le premier fragment et utilisée à nouveau dans le second.

```
<HTML>
<HEAD>
<TITLE>Status of <?php echo $host = chop(‘hostname‘) ?></TITLE>
</HEAD>
<BODY>
<H1><?php echo $host ?></H1>
<PRE>
<?php echo ‘uptime‘, "\n", ‘who‘ ?>
</PRE>
</BODY>
</HTML>
```

Figure 4.9 : Un simple script PHP intégré dans une page web.

Lorsqu'elle est servie par un hôte compatible PHP, cette page exécute l'équivalent du script CGI de la figure 4.8.

```
<HTML><BODY><P>
<?php
        for ($i = 0; $i < 20; $i++) {
            if ($i % 2) { ?>
<B><?php
                echo " $i"; ?>
</B><?php
            } else echo " $i";
        }
 ?>
</BODY></HTML>
```

Figure 4.10 : Un script PHP fragmenté.

Les instructions if et for fonctionnent comme on peut s'y attendre, malgré l'intervention du HTML brut. Lorsqu'un navigateur le demande, cette page affiche les chiffres de 0 à 19, les nombres impairs étant écrits en gras.

Les scripts PHP peuvent même être fragmentés au milieu de déclarations structurées. La figure 4.10 contient un script dans lequel les instructions if et for recouvrent des fragments. En effet, le texte HTML entre la fin d'un fragment de script et le début du suivant se comporte comme s'il avait été produit par une commande echo. Les concepteurs de sites web sont libres d'utiliser l'approche (écho ou échappement au HTML brut) qui leur semble la plus adaptée à la tâche à accomplir.

4.7.2.4 XSLT

La plupart des lecteurs auront sans doute eu l'occasion d'écrire, ou du moins de lire, le HTML (langage de balisage hypertexte) utilisé pour composer les pages web. Le HTML a, pour la plupart, une structure imbriquée dans laquelle des fragments de documents (*éléments*) sont délimités par des *balises* qui indiquent leur but ou leur apparence, par exemple, que les titres de haut niveau sont délimités par <H1> et </H1>. Malheureusement, en raison de la manière chaotique et informelle dont le web a évolué, le HTML s'est retrouvé avec de nombreuses incohérences dans sa conception, et des incompatibilités entre les versions mises en œuvre par les différents fournisseurs.

XML (Extensible Markup Language) est un langage plus récent et plus général pour la saisie de données structurées. Par rapport au HTML, sa syntaxe et sa sémantique sont plus régulières et plus cohérentes, et sont mises en œuvre de manière plus uniforme d'une plate- forme à l'autre. Il est *extensible*, ce qui signifie que les utilisateurs peuvent définir leurs propres balises. Il établit également une distinction claire entre le *contenu d'*un document (les données qu'il capture) et la *présentation* de ces données. La présentation, en fait, est reportée à une norme complémentaire connue sous le nom de XSL (langage de feuille de style extensible). Le XSLT est une partie du XSL consacrée à la *transformation du* XML : sélection, réorganisation et modification des balises et des éléments qu'elles délimitent.

4.7.2.5 Applets Java

Un applet est un programme conçu pour fonctionner à l'intérieur d'un autre programme. Le terme est le plus souvent utilisé pour les programmes Java qui affichent leur résultat dans (une partie de) une page web. Pour permettre l'exécution des applets, la plupart des navigateurs modernes contiennent une machine virtuelle Java.

Comme JavaScript, les applets Java peuvent être utilisés pour créer des pages animées ou interactives. Outre la similitude des noms de langage, le fait que de nombreuses tâches peuvent être accomplies avec l'un ou l'autre de ces mécanismes a créé une grande confusion entre les deux. En fait, ils sont cependant très différents.

```
<HTML>
<HEAD>
<TITLE>Adder</TITLE>
<SCRIPT type="text/javascript">
function doAdd() {
    argA = parseInt(document.adder.argA.value)
    argB = parseInt(document.adder.argB.value)
    x = document.getElementById('sum')
    while (x.hasChildNodes())
        x.removeChild(x.lastChild)  // delete old content
    t = document.createTextNode(argA + " plus "
        + argB + " is " + (argA + argB))
    x.appendChild(t)
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="adder" onsubmit="return false">
<P><INPUT name="argA" size=3> First addend<BR>
   <INPUT name="argB" size=3> Second addend
<P><INPUT type="button" onclick="doAdd()" value="Calculate">
</FORM>
<P><SPAN id="sum"></SPAN>
</BODY>
</HTML>
```

Adder

12 First addend

34 Second addend

Calculate

12 plus 34 is 46

Figure 4.11 : *Une page web interactive en JavaScript.*

La source apparaît à gauche. La version rendue sur la droite montre l'apparence de la page après que l'utilisateur ait entré deux valeurs et cliqué sur le bouton Calculer, ce qui fait apparaître le message de sortie. En entrant de nouvelles valeurs et en cliquant à nouveau, l'utilisateur peut calculer autant de sommes qu'il le souhaite. Chaque nouveau calcul remplacera le message de sortie.

Pour intégrer une applet dans une page web, on utiliserait traditionnellement une balise APPLET :

```
<APPLET width=150 height=150 code="Clock.class">
```

En voyant cet élément intégré dans la page, le navigateur du client demanderait l'URI *Clock.class au* serveur. En supposant que le serveur renvoie un applet, il exécute cet applet et affiche la sortie sur la page.

Contrairement à un script JavaScript, une applet ne produit pas de sortie HTML pour le navigateur. Elle contrôle plutôt directement une partie du contenu de la page, dans laquelle elle utilise des routines provenant de l'une des bibliothèques d'interface utilisateur graphique (GUI) de Java (généralement AWT ou Swing) pour afficher ce qu'elle veut. Les attributs width et height de l'élément APPLET indiquent au navigateur la taille de la partie de la page que doit avoir l'applet.

En effet, les applets permettent au concepteur de sites web d'échapper complètement au HTML et de créer un "look and feel" très précis, indépendant de tout choix de design incarné par le navigateur. Les images, bien sûr, offrent un autre moyen d'échapper au

HTML , avec un contenu statique ou simplement animé, tout comme les objets intégrés d'autres types (les films au format Flash ou QuickTime sont des exemples courants). La plupart des navigateurs modernes fournissent un mécanisme de "plugin" qui permet d'installer des interprètes pour des formats arbitraires. Pour les prendre en charge, la norme HTML fournit un élément OBJECT générique qui est destiné à être utilisé pour tout contenu intégré non rendu par le navigateur lui-même. L'élément APPLET est maintenant officiellement déprécié : on est censé utiliser ce qui suit à la place :

```
<P><OBJET codetype="application/java" classid="java:Clock.class" largeur=150
hauteur=150>
```

Les applets sont soumis à certaines restrictions destinées à éviter qu'ils n'endommagent la machine du client. Cependant, la plupart d'entre elles peuvent utiliser l'intégralité du langage Java, et il est généralement simple de dissimuler une applet à un programme autonome ou vice versa. L'applet typique n'a pas d'interaction significative avec le navigateur ou tout autre programme. Pour cette raison, les applets ne sont généralement pas considérés comme un mécanisme de script.

4.8 Conclusion

Dans ce chapitre, nous avons étudié les principaux paradigmes de programmation (la programmation impérative, la programmation orientée objet, la programmation concurrente, la programmation fonctionnelle, la programmation logique et les scripts) en insistant sur les concepts fondamentaux, ses pragmatiques et ses pratiques de programmation.

D'abord, Nous avons identifié les concepts clés de la programmation impérative : valeurs et types, liaisons et portée, variables et abstraction de procédurale, ensuite nous avons identifié les concepts clés de la programmation orientée objet : objets, classes et les sous-classes, le polymorphisme d'héritage et d'inclusion, puis nous avons présenté le paradigme de la programmation fonctionnelle. Dans sa nature il s'agit d'un modèle de calcul qui ne contient pas le concept de variable. Le calcul se fait par la réécriture des termes qui désignent des fonctions, après nous avons présenté les principales caractéristiques du paradigme de la programmation logique et enfin nous avons étudié les concepts communs à la plupart des langages de script :

4.9 Exercices

LA PROGRAMMATION IMPERATIVE

1 Considérez le code C suivant :

```
p = 1; m = n; a = b;
while (m > 0) {
if (m % 2 != 0) p = p * a;
m = m / 2;
a = a * a;
}
```

Rédigez une version plus concise de ce code. La version concise est-elle plus ou moins lisible?

2 C ne prend pas en charge l'abstraction de données ou l'abstraction générique. Néanmoins, il est possible de construire une bibliothèque d'unités de programme C qui permet d'obtenir certains des avantages de l'abstraction de données et de l'abstraction générique.

2.1 Montrez comment vous écrivez une unité de programme C qui produit l'effet d'un type abstrait. L'unité de programme doit fournir un type nommé (tel que Date), ainsi que certaines opérations sur ce type, sans révéler comment ce type est défini.

2.2 Suggérez maintenant comment vous pourriez obtenir l'effet d'un type *générique abstrait*. Vous devez permettre à une unité de programme qui implémente un type d'abstraction générique (tel que List) d'être instanciée selon les besoins. Les programmeurs d'application doivent pouvoir relier ces unités de programme à leurs programmes. Quels outils de gestion de logiciels seraient nécessaires, en plus du compilateur C?

LA PROGRAMMATION ORIENTÉ OBJET

1 Étant donné des définitions Java suivantes :

```
interface A {
int val=1;
int foo (int x);
}
interface B {
int z=1;
int fie (int y);
}
class C implements A, B {
int val = 2;
```

```
int z =2;
int n = 0;
public int foo (int x){ return x+val+n;}
public int fie (int y){ return z+val+n;}
}
class D extends C {
int val=3;
int z=3;
int n=3;
public int foo (int x){return x+val+n;}
public int fie (int y){ return z+val+n;}
}
```

Considérons maintenant le fragment de programme suivant :

```
int u, v, w, z;
A a;
B b;
D d = new D();
a = d;
b = d;
System.out.println(u = a.foo(1));
System.out.println(v = b.fie(1));
System.out.println(w = d.foo(1));
System.out.println(z = d.fie(1));
```

- Donnez les valeurs de u, v, w et z à la fin de l'exécution.

2 Le fragment de Java suivant est-il correct? Dans le cas positif, la méthode fie est redéfinie (overridden) ? Qu'est-ce qu'elle imprime ?

```
class A {
int x = 4;
int fie (A p) {return p.x;}
}
class B extends A{
int y = 6;
int fie (B p) {return p.x+p.y;}
}
public class binmeth {
public static void main (String [] args) {
B b = new B();
A a = new A();
int zz = a.fie(a)+ b.fie(a) ;
System.out.print(zz);
}
}
```

3 Chaque objet JAVA est une variable de tas (accessible par un pointeur), tandis qu'un objet C++ peut être soit une variable globale ou locale (accessible directement), soit une variable de tas (accessible par un pointeur). Quels sont les avantages et les inconvénients de l'inflexibilité de JAVA?

LA PROGRAMMATION CONCURRENTE

1 Quelle est la valeur initiale habituelle du champ valeur dans un sémaphore ?
2 Montrer un fragment de code qui, s'il est exécuté par deux threads, peut laisser la valeur x soit 0 soit 15, selon l'ordre dans lequel les deux threads entrelacent leur exécution. N'utilisez pas de synchronisation.
3 Montrer comment utiliser chacune des méthodes suivantes pour restreindre un fragment de code C afin qu'il ne puisse être exécuté que par un seul thread à la fois : sémaphores et régions critiques conditionnelles.
4 Créer une situation de blocage avec un seul thread, en utilisant chacune des méthodes suivantes : sémaphores et régions critiques conditionnelles.

LA PROGRAMMATION FONCTIONNELLE

1 Il peut être utile d'associer un nom à une valeur (terme) de façon à pouvoir la réutiliser par la suite. Ceci s'effectue à l'aide de la forme spéciale define. On peut ainsi définir des variables simples, comme (define annee 2020), mais aussi des fonctions, comme (define carre (lambda (x) (* x x))).

2 Cette dernière écriture peut être replacée par (define (carre x) (* x x)).

3 Donnez un nom et redéfinissez de cette manière les fonctions précedemment définies.

4 Définissez les fonctions dont les correspondants en lambda-calcul sont:

$\lambda x.x - 9.$

$\lambda xy.x \times (y + 2).$

$\lambda xyz.(x - z) \times ((y - x) \times z).$

5 Appliquer une fonction f à un paramètre x s'écrit (f x). Appliquez les fonctions précédemment définies à quelques paramètres et vérifiez qu'elles donnent le bon résultat.

6 La récursivité est au cœur des processus de calcul. En Scheme, une fonction que l'on définit avec define peut s'appeler elle-même et la récursivité est alors gérée de façon transparente.

7 Définissez les fonctions récursives suivantes.

- La fonction factoriel.
- La fonction itération (composition d'une même fonction n fois).
- Les fonctions qui déterminent la parité d'un entier.

8 En Scheme, une fonction est un terme comme un autre. Cela veut dire en particulier qu'une fonction peut être donnée en argument à une autre fonction et que le résultat d'un calcul peut être une fonction.

5.1 Définissez les fonctions suivantes.

- La fonction qui prend en argument deux fonctions et qui retourne la fonction qui est la composition des deux fonctions.
- La fonction qui retourne la somme de deux fonctions. ($\forall x$, f(x) = f1(x) + f2(x).
- La fonction qui effectue l'opération inverse.

9 Définir les fonctions suivantes :

- (der l) retourne le dernier élément de la liste x.
- (nieme l n) retourne le n-ième élément de la liste l.
- (elem l e) retourne vrai si e est un élément de la liste l.
- (concat x y) retourne la concaténation des listes x et y.
- (long l) retourne la longueur de la liste l.

LA PROGRAMMATION LOGIQUE

1 Citez les principales différences entre un programme logique et un programme PROLOG.

2 Ecrire en PROLOG un programme qui calcule la longueur (entendue comme le nombre d'éléments) d'une liste et renvoie cette valeur sous forme numérique. (Conseil : considérez une définition inductive de la longueur et utilisez l'opérateur is pour incrémenter la valeur dans le cas inductif).

3 Si, dans un programme logique, l'ordre des atomes dans le corps d'une clause est modifié, la sémantique du programme est-elle modifiée ? Justifiez votre réponse.

5. Étant donné le programme logique suivant :

```
membre(X, [X | Xs]).
member(X, [Y | Xs]):- membre(X, Xs).
```

- Quel est le résultat de l'évaluation de l'objectif :

(f(X), [1, f(2), 3]).

4 Étant donné le programme PROLOG suivant (rappelons que X et Y sont des variables, tandis que a et b sont des constantes) :

```
p(b):- p(b).
p(X):- r(b).
p(a):- p(a). r(Y).
```

- Indiquez si l'objectif p(a) prend fin ou non ; justifiez votre réponse.

6 Considérons le programme logique suivant :

```
p(X):- q(a), r(Y).
q(b).
q(X):- p(X).
r(b).
```

- Indiquez si l'objectif p(b) prend fin ou non ; justifiez votre réponse.

7 Donnez un exemple d'un programme logique, *P*, et d'un objectif, *G*, tel que l'évaluation de *G* dans *P* produit un effet différent lorsque deux règles de sélection différentes sont utilisées. (Suggestion : étant donné que nous avons vu que pour les réponses calculées, il n'y a pas de différence dans l'utilisation de règles de sélection différentes, examinez ce qui arrive à les calculs qui ne se terminent pas et qui échouent).

LES SCRIPTES

1 Le code HTML suivant comprend un exemple de lien vers un autre document : Voir le site de l'Acme

<A HREF="http://www.acme.com/mission.html"> déclaration de mission</A>.
La forme la plus simple de lien HTML est <A HREF="URL">lien-texte</A>. Notez une expression régulière qui correspond à cette forme de lien.

2 Expliquez les circonstances dans lesquelles il est logique de réaliser une tâche interactive sur le Web sous la forme d'un script CGI, d'un script intégré côté serveur ou d'un script côté client. Pour chacun de ces choix de l'implémentation, donnez trois exemples de tâches pour lesquelles c'est clairement l'approche privilégiée.

2.1 Écrivez une page web avec PHP intégré pour imprimer les 10 premières lignes du triangle de Pascal. Une fois rendue, votre sortie devrait ressembler à la Figure 4.12 ci-dessous.

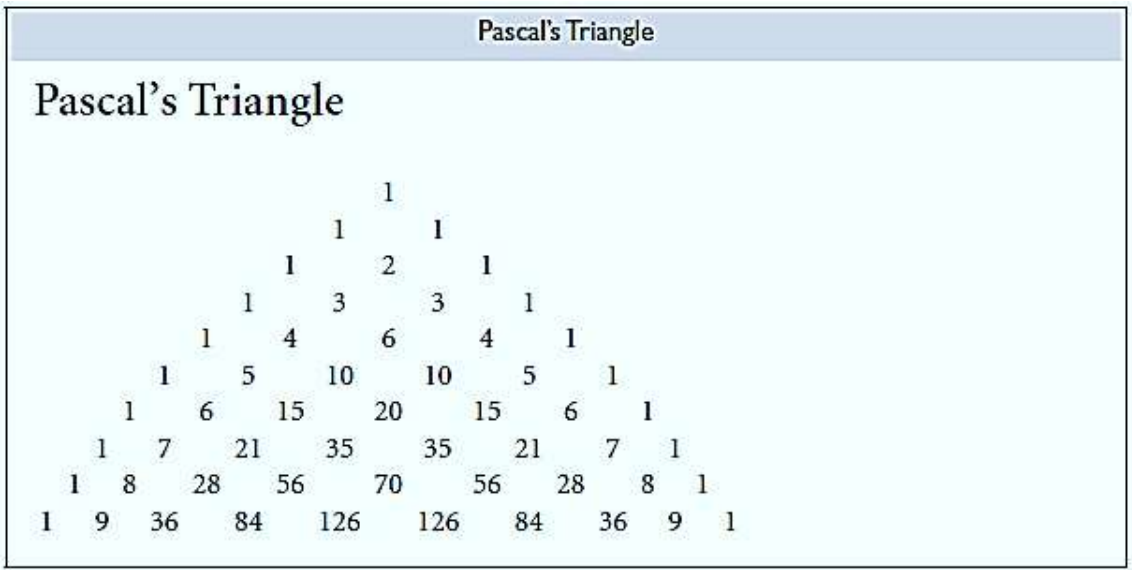

Figure 4.12 Le triangle de Pascal rendu dans une page web

2.2 Modifiez votre page pour créer un formulaire à affichage automatique qui accepte le nombre de lignes désirées dans un champ de saisie.

2.3 Réécrivez votre page en JavaScript.

3 Modifier le code de la figure 4.11 de manière à ce qu'il remplace le formulaire par sa sortie, comme le font les versions CGI et PHP de la figure 4.7.

4 Modifiez les scripts CGI et PHP de la figure 4.8 de manière à ce qu'ils apparaissent en bas du formulaire, comme le fait la version JavaScript de la figure 4.11.

Bibliographie

- Gabbrielli, Maurizio et Simone Martini. *Langages de programmation : principes et paradigmes*. Springer Science & Business Media, 2010.
- David A Watt, *Programming language design concepts,* John wiley& sons, 2004
- Daniel P. Friedman, Mitchell Wand, Christopher T. Haynes, Essentials of Programming Languages, MIT Press, 2001
- Van-Roy, Peter, et Seif Haridi. *Concepts, techniques et modèles de programmation informatique*. MIT press, 2004.
- Finkel, Raphael A., et Raphael A. Finkel. *Conception de langage de programmation avancée.* Lecture : Addison-Wesley, 1996.
- Scott, Michael Lee. *Pratique des langages de programmation*. Morgan Kaufmann, 2000.