UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES
M.Sc.

PAR
FRÉDÉRIC GAYTON

VERS UNE NOUVELLE INGÉNIERIE DE L'INFORMATION

JUIN 2004

CE MÉMOIRE A ÉTÉ ÉVALUÉ
PAR UN JURY COMPOSÉ DE :

M. Ismaïl Biskri, professeur et directeur de mémoire
Département de mathématiques et informatique à l'U.Q.T.R.

M. François Meunier, professeur
Département de mathématiques et informatique à l'U.Q.T.R.

M. Mhamed Mesfioui, professeur
Département de mathématiques et informatique à l'U.Q.T.R.

À mes enfants, Élisabeth, Corentin et Amandine.

REMERCIEMENTS

Je tiens à remercier sincèrement :

- Mon directeur de mémoire, Ismaïl Biskri, professeur d'informatique au département de mathématiques et d'informatique à l'Université du Québec à Trois-Rivières. Je le remercie tout spécialement pour son soutient indéfectible, sa sympathie et sa grande disponibilité.
- François Meunier, professeur d'informatique au département de mathématiques et d'informatique à l'Université du Québec à Trois-Rivières, pour avoir accepté d'évaluer ce mémoire et pour m'avoir toujours soutenu et encouragé dans mes démarches.
- Mhamed Mesfioui, professeur de mathématiques au département de mathématiques et d'informatique à l'Université du Québec à Trois-Rivières, pour avoir accepté d'évaluer ce mémoire.
- Chantal Lessard, secrétaire au département de mathématiques et d'informatique à l'Université du Québec à Trois-Rivières, pour sa gentillesse, sa bonne humeur et sa disponibilité.

Je tiens également à remercier :

- Dominique, pour la place qu'elle occupe à mes côtés et dans mon cœur. Je lui suis très reconnaissant pour l'écoute et la patience dont elle a fait preuve tout au long de mes travaux.
- Ma famille et mes amis, pour le soutient qu'ils représentent de près ou de loin.

TABLE DES MATIÈRES

Page

REMERCIEMENTS I

TABLE DES MATIÈRES II

LISTE DES FIGURES X

INTRODUCTION 1

CHAPITRE 1 5

ÉTAT DE L'ART 5

1.1 Introduction 5

1.2 Programmation orientée objet 8

1.2.1 Origine de l'orienté objet 8

1.2.2 Concepts de l'orienté objet 10

1.2.3 Caractérisation de l'approche orientée objet 12

1.2.3.1 Encapsulation 12

1.2.3.2 Héritage 13

1.2.3.3 Polymorphisme 14

1.2.4 Langages à objets 14

1.2.5 Apports des concepts 18

1.2.5.1 Encapsulation 18

1.2.5.2 Héritage 19

1.2.5.3 Messages 20

1.2.6 Avantages de l'orienté objet 21

1.2.6.1 Hausse de compréhensibilité 21
1.2.6.2 Réutilisation et évolution 22
1.2.6.3 Hausse de la qualité 23
1.2.6.4 Réduction des coûts 23
1.2.6.5 Réduction du temps de développement 24
1.2.7 Limites de l'orienté objet 24
1.2.7.1 Rapidité d'exécution 24
1.2.7.2 Dimension du code exécutable 24
1.2.7.3 Appel de méthode 25
1.2.7.4 Complexité du programme 25
1.2.7.5 Réutilisation 25
1.2.8 Conclusion 26

1.3 Programmation orientée agent 27
1.3.1 Origine de l'orienté agent 27
1.3.2 Définition d'un agent 29
1.3.3 Systèmes multi-agent 34
1.3.4 Conclusion 35

1.4 Programmation fonctionnelle 36
1.4.1 Origine de la programmation fonctionnelle 36
1.4.2 L'essence de la programmation fonctionnelle pure 38
1.4.2.1 Structure d'un programme fonctionnel 38
1.4.2.2 Des citoyens de première classe : les fonctions 39
1.4.2.3 Ordre normal d'évaluation 40
1.4.2.4 Définition de fonctions 42
1.4.2.5 Transparence référentielle 42
1.4.3 Types 44
1.4.3.1 Type fonction 45
1.4.3.2 Types algébriques (types structurés ou concrets) 46

1.4.3.3 Inférence de types 46
1.4.3.4 Polymorphisme des fonctions (polymorphisme paramétrique sémantique) 47
1.4.3.5 Surcharge des fonctions (polymorphisme ad-hoc) 47
1.4.4 Définitions de fonctions 48
1.4.4.1 Curryfication 49
1.4.4.2 Fonction d'ordre supérieur (forme fonctionnelle) 49
1.4.5 Stratégie d'évaluation 50
1.4.5.1 Évaluation paresseuse 50
1.4.5.2 Structures de données infinies 52
1.4.5.3 Entrées-sorties 52
1.4.5.3.1 Génération des sorties 53
1.4.5.3.2 Lecture des entrées 53
1.4.6 Conclusion 54

1.5 Conclusion 55

CHAPITRE 2 57

CONCEPTS ET OUTILS FONCTIONNELS 57

2.1 Introduction 57

2.2 Le lambda-calcul 58
2.2.1 La notion de fonction « classique » 59
2.2.2 La fonction chez Frege 59
2.2.3 La fonction dans la lambda-notation 62
2.2.4 La bêta-réduction 64
2.2.5 Le lambda-calcul 65
2.2.5.1 Syntaxe 65
2.2.5.2 Variable libre, variable liée 66

2.2.5.3 Règles de substitution 67
2.2.5.4 Règles de réduction 69
2.2.5.4.1 λ-β-réduction 69
2.2.5.4.2 Forme normale 70
2.2.5.4.3 Stratégies de réduction 70

2.3 Structure opérateur-opérande 70
2.3.1 Système applicatif 70
2.3.2 Mécanismes d'évaluation 72
2.3.3 La notion de type 75

2.4 Logique combinatoire 75
2.4.1 Principe applicatif 76
2.4.2 Expressions combinatoires 77
2.4.2.1 Expressions combinatoires de différents types 79
2.4.2.2 Arbres applicatifs 80
2.4.2.3 Évaluation des expressions combinatoires 80
2.4.3 Combinateurs et réductions 81
2.4.3.1 Les combinateurs élémentaires 82
2.4.3.1.1 Le combinateur "I" d'identité 82
2.4.3.1.2 Le combinateur "B" de composition 83
2.4.3.1.3 Le combinateur "S" de substitution 83
2.4.3.1.4 Le combinateur "Φ" de coordination 83
2.4.3.1.5 Le combinateur "Ψ" de distribution 84
2.4.3.1.6 Le combinateur "C*" de changement de type 84
2.4.3.1.7 Le Combinateur "C" de permutation 84
2.4.3.1.8 Le combinateur "W" de duplication 85
2.4.3.1.9 Le combinateur "K" d'effacement 85
2.4.3.2 Les combinateurs complexes 85
2.4.3.3 Puissance d'un combinateur 86

2.4.3.4 Combinateurs à distance 86
2.4.3.5 Théorèmes sur les combinateurs 87
2.4.3.6 Les types des combinateurs 87
2.4.3.7 Formes normales 89
2.4.3.8 Théorème de Church-Rosser 90

2.5 Conclusion 91

CHAPITRE 3 92

RECHERCHE THÉORIQUE 92

3.1 Introduction 92

3.2 Modèle utilisé 93
3.2.1 Notations 93
3.2.2 Restrictions 93
3.2.2.1 Contrainte d'intégrité 94
3.2.2.2 Contrainte d'utilisation 98
3.2.3 Conséquences 99

3.3 Rappels théoriques 101
3.3.1 Modules et principe applicatif 101
3.3.2 Récapitulatif sur les combinateurs élémentaires 102

3.4 Cas généraux d'enchaînement de plusieurs modules 103
3.4.1 Traitement en série 103
3.4.1.1 Deux modules en série 104
3.4.1.2 Trois modules en série 105
3.4.1.3 N modules en série 106
3.4.2 Traitement en parallèle 107
3.4.2.1 Parallélisme indépendant 108

3.4.2.1.1 Parallélisme indépendant de deux modules 108
3.4.2.1.2 Parallélisme indépendant de n modules 109
3.4.2.2 Parallélisme à entrées dépendantes 110
3.4.2.2.1 Parallélisme à entrées dépendantes de deux modules 110
3.4.2.2.2 Parallélisme à entrées dépendantes de n modules 111
3.4.2.3 Parallélisme à sorties dépendantes 112
3.4.2.3.1 Parallélisme à sorties dépendantes de deux modules 112
3.4.2.3.2 Parallélisme à sorties dépendantes de n modules 113
3.4.2.4 Parallélisme dépendant 114
3.4.2.4.1 Parallélisme dépendant de deux modules 114
3.4.2.4.2 Parallélisme dépendant de n modules 115
3.4.2.5 Parallélisme avec récupération 116
3.4.2.5.1 Parallélisme de deux modules avec récupération 117
3.4.2.5.2 Parallélisme de trois modules avec récupération 121
3.4.2.5.3 Parallélisme de n modules avec récupération 123
3.4.3 Combinaison série-parallèle 125
3.4.3.1 Traitement parallèle inclus dans le traitement sériel 125
3.4.3.2 Traitement sériel inclus dans le traitement parallèle 129
3.4.3.3 Cas général regroupant les approches précédentes 131

3.5 Cas particuliers d'enchaînement de plusieurs modules 135
3.5.1 Branches d'injection 135
3.5.2 Branches d'éjection 138
3.5.3 Autres cas particuliers 141
3.5.3.1 Parallélisme spécial 141
3.5.3.2 Parallélisme imbriqué 143
3.5.3.3 Sériel imbriqué 145
3.5.3.4 Modules non commutatifs 147

3.6 Étude de cas 148

3.7 Conclusion 152

CHAPITRE 4 153

IMPLÉMENTATION 153

4.1 Introduction 153

4.2 Considérations générales 153

4.3 Implémentation 157

4.3.1 Les modules 157

4.3.2 La base de données 158

4.3.3 L'arbre d'exécution 159

4.3.4 L'interface 163

4.4 Exemple de construction 166

4.5 Cas de test 171

4.5.1 Modules simples utilisés 172

4.5.2 Série de 2 modules 173

4.5.3 Série de 3 modules 173

4.5.4 Série de 6 modules 174

4.5.5 Parallélisme indépendant de 3 modules 174

4.5.6 Parallélisme à entrées dépendantes de 3 modules 175

4.5.7 Parallélisme à sorties dépendantes de 3 modules 176

4.5.8 Parallélisme dépendant de 3 modules 177

4.5.9 Parallélisme de 2 modules avec récupération 177

4.5.10 Parallélisme de 3 modules avec récupération 178

4.5.11 Parallélisme de 6 modules avec récupération 179

4.5.12 Traitement parallèle inclus dans le traitement sériel 180

4.5.13 Traitement sériel inclus dans le traitement parallèle 181

4.5.14 Cas général de mélange des traitements sériels et parallèles 181
4.5.15 Branches d'injection 182
4.5.16 Branches d'éjection 183
4.5.17 Parallélisme spécial 184
4.5.18 Parallélisme imbriqué 185
4.5.19 Sériel imbriqué 185
4.5.20 Ordre des connexions 186
4.5.21 Étude de cas générale 187

4.6 Conclusion 189

CONCLUSION 190

ANNEXE A 194

DÉTAILS DE L'IMPLÉMENTATION 194

BIBLIOGRAPHIE 219

LISTE DES FIGURES

Page

Figure 1.1 – Organisation des langages de programmation. 16
Figure 1.2 - Architecture d'agent. 33
Figure 3.1 - Représentation d'un module M. 93
Figure 3.2 - Représentation de 2 modules M1 et M2 reliés. 95
Figure 3.3 - Représentation d'un module M avec deux liens en entrée. 95
Figure 3.4 - Représentation d'un module M avec deux liens en sortie. 95
Figure 3.5 - Représentation de deux modules M1 et M2 liés à une entrée. 96
Figure 3.6 - Représentation de deux modules M1 et M2 liés à une sortie. 96
Figure 3.7 - Représentation d'un module M avec deux sorties. 97
Figure 3.8 - Représentation d'un module M avec deux entrées. 98
Figure 3.9 - Représentation d'un module complexe Mc avec deux sorties, dont une est reliée à un module simple M1 98
Figure 3.10 - Représentation d'un module complexe Mc dont la chaîne de traitement est (M1I1)(M2(M1I2)). 99
Figure 3.11 - Représentation d'un module complexe Mc dont la chaîne de traitement est (M1I1)(M2(M1I1)). 100
Figure 3.12 - Représentation de 2 modules en série. 104
Figure 3.13 - Représentation de 3 modules en série. 105
Figure 3.14 - Représentation de n modules en série. 107
Figure 3.15 – Parallélisme indépendant de 2 modules. 108
Figure 3.16 – Parallélisme indépendant de n modules. 109
Figure 3.17 – Parallélisme à entrées dépendantes de 2 modules. 110
Figure 3.18 – Parallélisme à entrées dépendantes de n modules. 111
Figure 3.19 – Parallélisme à sorties dépendantes de 2 modules. 112
Figure 3.20 – Parallélisme à sorties dépendantes de n modules. 113

Figure 3.21 – Parallélisme dépendant de 2 modules. 115
Figure 3.22 – Parallélisme dépendant de n modules. 116
Figure 3.23 – Parallélisme de 2 modules avec récupération..... 117
Figure 3.24 – Parallélisme de 3 modules avec récupération..... 121
Figure 3.25 – Parallélisme de n modules avec récupération..... 123
Figure 3.26 – Exemple de parallélisme inclus dans le traitement sériel..... 126
Figure 3.27 – Exemple de traitement sériel inclus dans le parallélisme..... 130
Figure 3.28 – Combinaison série-parallèle de 12 modules..... 132
Figure 3.29 – Branche d'injection i dans un traitement série. 135
Figure 3.30 – Branche d'injection i dans un traitement parallèle..... 135
Figure 3.31 – N branches d'injection i1, …, in sur un module M..... 136
Figure 3.32 – Exemple avec branches d'injection..... 136
Figure 3.33 – Branche d'éjection e dans un traitement série..... 138
Figure 3.34 – Branche d'éjection e dans un traitement parallèle..... 138
Figure 3.35 – N branches d'éjection e1, …, en à partir d'un module M..... 139
Figure 3.36 – Exemple avec branches d'éjection. 139
Figure 3.37 – Exemple de parallélisme spécial. 141
Figure 3.38 – Exemple de parallélisme imbriqué..... 143
Figure 3.39 – Exemple de sériel imbriqué..... 145
Figure 3.40 – Module complexe avec comme entrées : (a) I1 et I2, (b) I2 et I1. 147
Figure 3.41 – Représentation du module complexe Mc de l'étude de cas. 148
Figure 3.42 – Exemple général d'une chaîne de traitement.. 149
Figure 4.1 – Structure de notre base de donnée..... 158
Figure 4.2 – Représentation d'un nœud avec deux enfants. 160
Figure 4.3 – Interface principale de l'application..... 164
Figure 4.4 – Interface de saisie des types en entrée d'un module. 166
Figure 4.5 – Arbre d'exécution d'un module complexe Mc dont la chaîne de traitement est (BM2M1E1)((M1E1)(M2E2)). 171
Figure 4.6 - Représentation des modules simples utilisés. 172

Figure 4.7 - M9, module illustrant la série de 2 modules. 173
Figure 4.8 - M10, module illustrant la série de 3 modules. 173
Figure 4.9 - M11, module illustrant la série de 6 modules. 174
Figure 4.10 - M12, module illustrant le parallélisme indépendant de 3 modules..... 175
Figure 4.11 - M13, module illustrant le parallélisme à entrées dépendantes de 3 modules. 175
Figure 4.12 - M14, module illustrant le parallélisme à sorties dépendantes de 3 modules. 176
Figure 4.13 - M15, module illustrant le parallélisme dépendant de 3 modules..... 177
Figure 4.14 - M16, module illustrant le parallélisme de 2 modules avec récupération. 178
Figure 4.15 - M17, module illustrant le parallélisme de 3 modules avec récupération. 178
Figure 4.16 - M18, module illustrant le parallélisme de 6 modules avec récupération. 179
Figure 4.17 - M19, module illustrant le traitement parallèle inclus dans le traitement sériel. 180
Figure 4.18 - M20, module illustrant le traitement sériel inclus dans le traitement parallèle. 181
Figure 4.19 - M21, module illustrant le cas général de mélange des traitements sériels et parallèles. 182
Figure 4.20 - M22, module illustrant les branches d'injection. 183
Figure 4.21 - M23, module illustrant les branches d'éjection. 184
Figure 4.22 - M24, module illustrant le parallélisme spécial. 184
Figure 4.23 - M25, module illustrant le parallélisme imbriqué. 185
Figure 4.24 - M26, module illustrant le sériel imbriqué. 186
Figure 4.25 - M27, module illustrant l'importance de l'ordre des connexions. 186
Figure 4.26 - M28, module illustrant l'importance de l'ordre des connexions. 187
Figure 4.27 - M29, module illustrant l'étude de cas générale. 188

INTRODUCTION

L'ingénierie de la langue se retrouve à l'intersection de plusieurs disciplines comme l'informatique, l'intelligence artificielle, la linguistique, la psychologie, la sémiologie, la logique, la philosophie, la terminologie, etc.

Malgré le succès des technologies développées et mises à la disposition des ingénieurs de la langue, ces derniers émettent des insatisfactions dues aux limites importantes suivantes :

- elles offrent un ensemble limité de fonctionnalités,
- elles sont souvent conçues dans une architecture qui limite les possibilités de communication avec d'autres logiciels,
- il est difficile, voire impossible, d'intégrer de nouvelles fonctionnalités sans rebâtir l'ensemble de l'application,
- elles rendent difficile la collaboration tant recherchée par les experts et leurs médiateurs.

Il existe aussi une autre insuffisance au niveau des outils de développement mis à la disposition des chercheurs. Nous pouvons nous apercevoir aisément qu'il y a une sorte de paradoxe entre les outils existants et la finalité de ces outils appliqués au secteur de la recherche. En effet, le chercheur a pour données des hypothèses, mais ne peut connaître les résultats finaux de ses expériences. Il peut tout au plus les supposer ou les espérer, ce qui est contraire aux outils de développement classiques qui nécessitent que le développeur connaisse les tenants et les aboutissants de ses tentatives.

C'est pourquoi, le projet de recherche, mené sous la direction de mon professeur M. Ismaïl Biskri, a pour objectif de mettre au point un outil d'aide au développement pour les chercheurs non informaticiens. Cette mise en œuvre se fera par la conception d'une

plate-forme informatique dotée d'une boîte à outils et d'une interface usager qui servira de laboratoire aux ingénieurs de la langue pour concevoir leurs chaînes de traitement, en se servant des différentes fonctionnalités mises à leur disposition. Il faudra notamment leurs permettre d'enchaîner, via une interface graphique, des modules de traitement informatique qui soient exécutables, suite à une validation des liens créés entre les modules.

Dans une telle perspective, une chaîne de traitement est vue comme un assemblage ou une combinaison de modules pouvant être réalisée de manière sérielle ou parallèle. La chaîne, une fois construite et validée, représente à son tour un module dit complexe. Un module est simple s'il est seul, et complexe s'il est composé de plusieurs modules simples. Ainsi, l'exécution d'un module simple fera appel à l'exécutable de ce dernier, alors que l'exécution d'un module complexe nécessitera l'exécution des modules qui le composent dans l'ordre spécifié à la construction du module complexe. L'interprétation d'une chaîne de traitement sera ainsi l'aboutissement de l'interprétation des fonctions primitives qui la forment et de la façon dont ces fonctions sont agencées.

Nous pouvons donc déjà identifier deux parties importantes de cette plate-forme. Une première partie qui concerne la construction de chaînes de traitement valides, et une deuxième qui concerne leur exécution. Il va alors falloir définir un modèle théorique de construction systématique de chaînes de traitement. Nous adressant à des chercheurs, nous opterons pour une approche semi-automatique, interactive et constamment sous le contrôle de l'utilisateur, pour qu'ils puissent progressivement et facilement affiner une solution en cours de développement. De plus, notre choix se portera sur l'utilisation des exécutables des modules, et non sur leur code source, afin d'être totalement indépendant de tout langage de programmation, et d'ainsi permettre aux non informaticiens de construire facilement des chaînes de traitement.

Tout cela laisse entrevoir une nouvelle technologie « pièces détachées » où un expert en ingénierie de l'information pourra choisir des modules et construire une chaîne de traitement tel un enfant faisant une construction avec son jeu de LEGO. Une multitude de chaînes de traitement sera possible. Il sera permis de leur donner un nom et de les stocker en mémoire pour une réutilisation. Pour ce faire nous aurons besoin d'imprimer dans une base de données, par exemple, la combinaison des modules dans la chaîne de traitement. Un moyen élégant pour représenter cette combinaison pourrait être la logique combinatoire [BISK 1997].

Dans le cadre de ma maîtrise, l'objectif de la recherche se limitera à la partie qui concerne l'exécution d'une chaîne de traitement. Il faudra néanmoins développer un formalisme capable de représenter les enchaînements de modules possibles. La littérature offre des moyens grammaticaux et des outils logiques qui peuvent répondre à un tel défi, en particulier les grammaires catégorielles, la logique combinatoire et le lambda-calcul.

Nous commencerons par faire une étude comparative de différents paradigmes de programmation [APPL 1997, TUCK 2001, SEBE 2002, WATT 2004] afin de déterminer une approche qui réponde aux besoins reliés à notre recherche. Le chapitre 1 s'intéressera donc à l'étude de trois paradigmes qui ont retenu notre attention, à savoir les paradigmes objet [BOOC 1994, BOUZ 1998, BUDD 1997, LUCK 2004, MEYE 2000], agent [CHAI 2002, DELI 2002, JENN 1998, WOOL 2000] et fonctionnel [COUT 2003, ECKE 1996, EISE 1987, HEND 1980, MICH 1989]. Nous verrons que les éléments de la programmation fonctionnelle répondent au mieux à nos exigences. En effet, la capacité de tout voir comme des expressions et des fonctions (à une ou zéro variable), rejoint notre idée de chaînes de traitement. Nous pouvons aisément voir chaque module d'une chaîne de traitement comme une fonction, et l'enchaînement des modules comme un nouveau module, dit complexe, qui serait équivalent à une fonction complexe. De plus, d'après Sebesta [SEBE 2002] , « Il n'y a pas de variables dans le

sens des langages impératifs, si bien qu'il ne peut pas y avoir d'effets de bord ». L'approche fonctionnel va donc nous permettre de facilement enchaîner les modules, sans soucis de l'effet de bord.
Nous consacrerons alors le chapitre 2 à une analyse de la programmation fonctionnelle. Dans ce domaine, le lambda-calcul et la logique combinatoire sont les moyens utilisés actuellement par les informaticiens pour analyser les propriétés sémantiques des langages de programmation de haut-niveau. L'étude de ces deux théories nous permettra d'affirmer que la logique combinatoire de Curry [CURR 1958] est à privilégier, plutôt que le lambda-calcul et la lambda-abstraction de Church [CHUR 1941]. En effet, d'après Desclés [DESC 1990], la théorie combinatoire nous permet d'en arriver à une logique dite sans variable, ce qui résout le problème majeur lié à l'utilisation du lambda-calcul, à savoir celui du télescopage des variables.

Nous retiendrons donc les combinateurs comme outils pour sauvegarder, d'une façon efficace et élégante, l'enchaînement des modules qui composent une chaîne de traitement. Le chapitre 3 sera consacré au développement d'une théorie capable de formaliser les chaînes de traitement, afin de pouvoir exécuter tous les modules d'une chaîne dans l'ordre attendu. Pour cela, nous ferons appel à la théorie des combinateurs qui ne sont rien d'autres que des opérateurs abstraits, dont le but est de composer les opérateurs d'un système applicatif pour former des opérateurs plus complexes. Ceci rejoint notre problématique, puisqu'une chaîne de traitement est l'arrangement de plusieurs modules, chacun étant vu comme un opérateur. De par leurs caractéristiques comportementales, certains combinateurs vont nous permettre de mémoriser l'ordre d'exécution des modules dans une chaîne de traitement.

Avant de conclure sur les travaux effectués, le chapitre 4 présentera l'implémentation de la théorie mise au point au chapitre précédent. Nous étudierons également un ensemble de cas de test, illustrant tous les cas généraux ou particuliers d'enchaînement de modules identifiés.

CHAPITRE 1

ÉTAT DE L'ART

1.1 Introduction

Aujourd'hui, il existe de nombreux langages de programmation, chacun ayant ses caractéristiques. Cependant, la plupart de ces langages sont associés à un paradigme de programmation, c'est-à-dire une approche dans la résolution de problèmes ou une manière de penser. Il existe plusieurs paradigmes de programmation : impératif, déclaratif, orienté objet, fonctionnel, logique, orienté agent, distribué, orienté règle, orienté contrainte, etc. Nous pouvons également citer la programmation générique, qui donnera plus tard (2000) la programmation par aspect. Les paradigmes impératif et déclaratif constituent deux familles principales dont les autres paradigmes sont une spécialisation.

Voici une brève description des paradigmes principaux :

- La programmation impérative, ou procédurale, est supportée par la majorité des langages de programmation. Elle est orientée traitement puisque conceptuellement dominée par le modèle de la machine de Von Neumann. Ainsi, la notion d'affectation, ou d'effet de bord (le programme se déroule en écrasant les valeurs d'une zone mémoire par une nouvelle valeur), et la présence de riches structures de contrôle de l'exécution (pour spécifier précisément l'ordre dans lequel les instructions doivent être exécutées) sont des caractéristiques essentielles des langages impératifs. Le type d'abstraction utilisée est l'algorithme. La programmation impérative est la meilleure pour, par exemple, résoudre un système d'équations linéaires.

- Au contraire, l'idéal du paradigme déclaratif est qu'un programme ne fait qu'énoncer les propriétés de la solution de manière à la contraindre, sans décrire comment la calculer. Les langages déclaratifs sont des langages plus centrés données. Les instructions de ces langages permettent au programmeur de spécifier ce qui va être fait avec les données. Ces langages sont généralement d'un plus haut niveau que les langages orientés traitement (programmation plus naturelle).

- Le paradigme par objets est généralement une simple variante du paradigme impératif dont il n'a jamais été clairement séparé. Les types d'abstractions utilisées sont les classes et les objets. Les données ainsi que les traitements qui portent sur ces données sont regroupés dans une même entité : l'objet. La classe décrit l'ensemble des propriétés que partage un ensemble d'objets. Les objets communiquent entre eux par des messages. Ce modèle de programmation est plus proche de la représentation de la réalité de par son principe de regroupement des éléments en fonction de critères communs pour simplifier la complexité. La programmation orientée objet est la meilleure pour les logiciels industriels où la complexité est dominante.

- Le paradigme fonctionnel est une des variantes dans la famille des langages déclaratifs. Il se fonde sur le modèle mathématique de l'application de fonctions, c'est pourquoi il est également appelé paradigme applicatif. Le résultat de l'évaluation d'une fonction sert d'entrée à une autre fonction, et ainsi de suite jusqu'à l'obtention du résultat final désiré. Le calcul, et donc la programmation, est entièrement réalisé au moyen de l'évaluation d'expressions. Il n'y a pas de notion d'emplacement mémoire (variable) qui puisse être affecté ou modifié. Il n'y a que des valeurs qui résultent de fonctions et peuvent être utilisées par d'autres fonctions. Il n'y a pas non plus d'instructions (ou commandes). Alors que la notion de flot de contrôles a marqué des générations de langages, cette

notion disparaît dans le paradigme fonctionnel. Cela conduit à des langages qui ne sont pas contaminés par les effets de bord, les affectations, le séquencement explicite et d'autres caractéristiques vues comme repoussantes. Ces langages sont donc mathématiquement plus propres, sans élément référentiellement opaque. En général, ils sont moins efficaces du point de vue de l'exécution, mais permettent un niveau d'abstraction plus élevé. Ainsi, ce modèle de programmation est plus proche de l'utilisateur.

- La programmation logique fait aussi partie des langages déclaratifs. Les types d'abstractions utilisées sont les buts, souvent exprimés comme des prédicats de calcul (logique formelle). Ce paradigme est le moins utilisé et reste représenté essentiellement par le langage Prolog (PROgrammation LOGique), créé en 1972 pour le traitement de la langue naturelle. Tout comme le style fonctionnel, c'est un modèle de programmation plus proche de l'utilisateur qui repose sur des modèles mathématiques.

- La programmation orientée agent est basée sur le concept d'agent qui étend celui d'objet. En effet, un agent possède les caractéristiques des objets, mais est en plus capable d'indépendance et d'une certaine intelligence.

- La programmation orientée règle est la meilleure pour la conception de base de connaissances. Les types d'abstractions utilisées sont les règles IF-THEN.

Il est à noter qu'un langage peut offrir plusieurs styles de programmation, ils ne sont pas mutuellement exclusifs. Comme, par exemple, le langage ML qui est de style fonctionnel et impératif.

Nous voyons que le concept de paradigme de programmation donne naissance à différents langages spécifiques qui mettent en évidence des applications typiques de ces

langages. Chaque style est basé sur son propre cadre conceptuel. Ainsi, il n'y a pas un paradigme unique bon pour toutes les applications! Notre objectif est alors de trouver quelle façon de penser serait la mieux adaptée aux besoins de notre recherche.

Dans ce chapitre, nous allons donc nous intéresser à l'étude de trois paradigmes qui ont retenu notre attention, à savoir les paradigmes objet, agent et fonctionnel. Nous allons tout d'abord porter notre analyse vers le paradigme objet puisqu'il fait partie des incontournables d'aujourd'hui, bien que, nous le verrons, cette approche présente certaines limites par rapport au problème qui est le notre. Ensuite, nous nous pencherons sur le paradigme agent, puisque cette approche plutôt récente semble avoir un bon avenir. Enfin, nous verrons le paradigme fonctionnel qui par la notion de fonction appliquée à des arguments paraît le mieux répondre à nos exigences.

1.2 Programmation orientée objet

La programmation orientée objet repose sur le concept d'objet comme représentation du monde réel. Pour en savoir plus, nous allons commencer par parler de l'origine de l'orienté objet, puis nous verrons les concepts et les caractéristiques de ce style de programmation. Ceci va nous permettre de distinguer les différents langages à objet. Ensuite, nous soulignerons les avantages et les limites reliés à l'orienté objet, avant de conclure.

1.2.1 Origine de l'orienté objet

Les principes de la programmation orientée objet, sont nés en fait de deux tendances relativement anciennes :

- D'une part, la difficulté d'exploiter des programmes de taille importante a imposé la notion de programmation structurée. Dans cette approche, l'analyse du problème privilégie les traitements (décomposition de l'application en sous-programmes) et néglige l'aspect des données.
- D'autre part la manipulation de volumes d'informations importants a mis en évidence la nécessité de structurer et de regrouper en entités les ensembles de données partageant les mêmes caractéristiques. Ceci a donné naissance à la programmation dirigée par les données (les données orientent l'analyse et influencent la structure des programmes).

Aujourd'hui, les applications informatiques sont de plus en plus complexes (taille et structure) et les domaines d'applications de plus en plus diversifiés. Ainsi, l'usage de l'une ou de l'autre de ces approches s'est vite révélé problématique dans la conception de grands systèmes car ceux-ci nécessitent une décomposition à la fois selon les données et les sous-programmes. Le premier langage ayant proposé une technique mariant les deux approches (données et procédures) fût Simula.

Le langage Simula (O.J Dahl et K. Nygaard) a été développé comme une extension du langage Algol 60 à partir de 1963 au Norvegian Computer Center (Oslo). Il a été conçu pour la description des systèmes et la programmation de simulations. Il fut le premier langage à introduire les notions de classe et d'objet. Une classe encapsule une structure de données et un ensemble de procédures permettant de les manipuler. C'est une entité générique dont les représentants, les objets, sont créés dynamiquement. Cette philosophie remettait en cause la séparation qui existait entre données et programmes. Elle fût exploitée plus tard dans la conception des langages à types abstraits d'où sont issues Clu et Ada.

Depuis, la vogue des langages de manipulation d'objet s'est répandue. L'objet constitue une représentation cohérente de la réalité, ainsi la programmation orientée objet, en tant

que paradigme, constitue une nouvelle manière de réfléchir sur le sens du traitement informatique et sur la façon de structurer les informations à l'intérieur d'un ordinateur.

1.2.2 Concepts de l'orienté objet

L'approche orientée objet est un concept nouveau qui a rapidement envahi le milieu du développement des logiciels. Elle est présentée comme une technologie qui mène à une meilleure intégration des logiciels et, de ce fait, à une augmentation de la productivité dans ce domaine. Les techniques de programmation orientée objet permettent au programmeur, d'une part, d'utiliser des éléments de code préalablement développés. La réutilisation de tels composants réduit la complexité du code développé, ainsi que le temps et les efforts nécessaires à son développement. D'autre part, la programmation orientée objet permet à l'utilisateur et au programmeur de visualiser les concepts comme une variété d'unités ou d'objets, une hiérarchie de composants ou de structures d'organisation diverses. Cela permet aux programmeurs de représenter facilement les relations entre les composants, les objets, les tâches à accomplir et les conditions à remplir.

Avec l'introduction de langages de plus en plus de haut niveau, la complexité des problèmes à traiter a augmenté progressivement. Une méthode très puissante pour gérer la complexité a donc été inventée : l'*abstraction*. Une abstraction décrit les caractéristiques essentielles d'une entité, sans s'occuper des détails. Ces caractéristiques doivent permettre de la distinguer de toutes les autres entités et de préciser ainsi ses limites conceptuelles. Une abstraction comporte une partie statique qui décrit la structure de l'entité à modéliser (les éléments qui la composent), et une partie dynamique qui décrit son comportement.

Le concept clé de cette approche est l'*objet*. Un objet est une « abstraction d'une donnée caractérisée par un identifiant unique et invariant, une classe d'appartenance et un état

représenté par une valeur simple ou structurée » [BOUZ 98], il définit ensuite la *classe* comme une « abstraction d'un type de donnée caractérisée par des propriétés (attributs et opérations) communes à des objets, et permettant de créer des objets ayant ces propriétés ». Ainsi, la classe décrit l'ensemble des propriétés que partage un ensemble d'objets. Les données ainsi que les traitements qui portent sur ces données sont regroupés dans une même entité : l'objet.

Le mécanisme par lequel l'exécution d'un programme produit un objet à partir d'une classe constitue l'*instanciation*. Par cette dernière, la définition d'une classe sert de modèle à la construction de ses représentants physiques, appelés instances, occurrences ou objets. Une instance constitue un objet particulier, créé en respectant les plans de construction de sa classe.

Attention, il ne faut pas confondre objet et classe! En effet, il existe une différence fondamentale entre classe et objet : les objets existent seulement pendant l'exécution d'un programme, alors que les classes sont des descriptions purement statiques d'ensembles possibles d'objets. Autrement dit, à l'exécution, il n'existe que des objets, mais, dans le programme, nous manipulons des classes.

Conceptuellement, le contrôle dans les langages à objets est assuré par des communications entre les objets, et non par des appels de procédures. La *communication par envoi de messages* assure l'interaction entre les objets. L'adressage d'une requête (message) à un objet provoque l'activation d'une de ses méthodes. Un envoi de message mentionne toujours le récepteur du message, le sélecteur (nom) de la méthode à exécuter, et les arguments de la méthode à exécuter. Ainsi un programme, vu sous l'angle de la programmation orientée objet, est considéré comme un ensemble d'objets qui interagissent entre eux par envois de messages.

1.2.3 Caractérisation de l'approche orientée objet

Le paradigme objet est une nouvelle façon de concevoir le logiciel. L'accent est mis sur les données et les actions qu'elles supportent plutôt que sur une vision totalement procédurale. Il est essentiellement caractérisé par l'*encapsulation* des données, par l'*héritage* des attributs et méthodes, et par le *polymorphisme*.

1.2.3.1 Encapsulation

L'encapsulation est le mécanisme par lequel le programmeur cache une partie de l'information pour préserver l'intégrité de l'objet. Elle a pour principal objectif de masquer l'implémentation des structures au programmeur qui les utilise en fonction de leurs spécifications. Elle est généralement mise en œuvre par une séparation physique entre l'interface qui décrit les spécifications des fonctionnalités et le corps qui en réalise l'implémentation. Il n'est alors pas possible d'agir directement sur les données d'un objet, il est nécessaire de passer par l'intermédiaire de ses opérations (méthodes) qui jouent le rôle d'interface obligatoire.

Ainsi, l'encapsulation facilite la représentation et l'organisation des connaissances et des données, et influe sur l'usage ultérieur de l'objet. Elle est aussi appelée *dissimulation*, en ce sens que l'accès à un ensemble défini de structures de données est limité à une liste de fonctions déclarées par le concepteur. Toute fonction qui n'est pas explicitement autorisée à accéder à la structure interne de l'objet provoquera une erreur.

La modélisation par objets dispose donc d'un schéma de représentation simple : l'objet en tant qu'encapsulation de données et de traitement, la classe en tant que type abstrait. Plus qu'une simple programmation dirigée par les données, la programmation par objets est une véritable programmation par abstraction de données.

1.2.3.2 Héritage

Le mécanisme d'héritage permet de définir de nouvelles classes à partir de classes existantes. C'est en fait le procédé par lequel une classe, dite sous-classe ou classe dérivée, reçoit une partie de sa définition d'une autre classe, dite classe de base ou super classe [DESF 97], [BOUZ 98], [BUDD 97]. Il s'agit d'un problème de partage efficace des connaissances. La classe est considérée comme un réservoir de connaissances à partir duquel il est possible de définir d'autres classes plus spécifiques, complétant les connaissances de leur classe de base. Les connaissances les plus générales sont ainsi mises en commun dans des classes qui sont spécialisées par définition de sous-classes contenant des connaissances de plus en plus spécifiques. Ainsi, une sous-classe constitue une spécialisation de la description de sa *super classe*. C'est pourquoi nous parlons aussi de *généralisation-spécialisation*, puisque c'est elle qui permet une classification des objets en fonction de leurs points communs (généraux) et de leur spécificité par un même graphe.

La relation d'héritage organise donc hiérarchiquement les classes en graphes d'héritage, ce qui traduit le principe de généralisation spécialisation. L'héritage est dit simple si une classe ne peut directement hériter que d'une super classe, ou multiple si une classe peut directement hériter de plusieurs classes de base [SEBE 02].

De façon générale, dire qu'une classe B hérite d'une classe A, c'est dire que les propriétés (structurelles et comportementales) de A sont aussi celles de B. Plus particulièrement, la sous-classe B peut redéfinir certaines propriétés de sa super classe A, nous parlons alors de *substitution*, ou elle peut rajouter des propriétés qui lui sont propres, et nous parlons dans ce cas d'*enrichissement (surcharge, overloading)*.

Dans plusieurs langages, il existe une classe appelée *Object* qui représente l'ensemble de toutes les instances possibles d'une application. Par définition, toutes les classes héritent

de la classe *Object* qui constitue alors la racine du graphe d'héritage. Ceci permet d'introduire la notion de *classe abstraite* : une classe abstraite est une classe utilisée pour factoriser des propriétés communes à plusieurs classes, mais qui est trop générique pour instancier des objets. Une classe abstraite n'a donc, en principe, aucune instance, car elle n'a pas assez de propriétés pour que l'on puisse caractériser avec suffisamment de précision ses instances.

1.2.3.3 Polymorphisme

Dans [DESF 97], le polymorphisme est défini comme la « faculté de considérer sous une même forme des éléments de nature différente ». C'est la capacité d'obtenir plusieurs réactions à une sollicitation donnée. Autrement dit, c'est la possibilité pour un même message de déclencher des traitements différents suivant les objets auxquels il est adressé. Il permet donc de donner le même nom à des traitements différents. Le nom d'une méthode peut être *surchargé* pour implanter plusieurs fonctions du même nom laissant le compilateur choisir la bonne.

En programmation par objets, le polymorphisme caractérise une entité qui fait référence, au moment de l'exécution, à des occurrences de différentes classes. Par exemple, les opérateurs arithmétiques sont polymorphes par le fait que l'addition, la soustraction peuvent s'appliquer aux entiers, aux réels, voire aux ensembles. Cette caractéristique est la conséquence directe de la communication par envoi de messages.

1.2.4 Langages à objets

La programmation orientée objet permet de structurer l'univers des applications en fonction d'objets plutôt qu'en fonction de procédures. Dans ce contexte, les langages à objets, regroupant les langages de programmation qui intègrent la notion d'héritage à

l'abstraction de données, sont largement utilisés. Il existe plusieurs catégories de langages à objets : les *langages basés objets*, les *langages de classes*, les *langages acteurs* et les langages dits *orientés objets*.

Un langage de programmation est dit *basé objets* s'il autorise la modélisation directe d'objets du monde réel, tels des pièces mécaniques, des automobiles, des personnes ou des comptes en banque, l'objet étant défini comme un ensemble d'opérations et un état qui mémorise les derniers effets de ces opérations. Les objets de tels langages ne contiennent pas nécessairement de classes d'appartenance, ni d'héritage : ils sont appelés *packages* et n'appartiennent à aucune classe. Le langage Ada constitue un exemple de langage basé objets.

Un langage de classes est un langage qui supporte la notion de classes, laquelle permet de réunir des objets et de les assimiler à un modèle. Les objets de ces classes peuvent être transmis en paramètres, affectés à des variables et organisés en structures. Toutefois, il n'existe pas de relation hiérarchique entre les données des différents niveaux. Autrement dit, les relations d'héritage n'existent pas dans ces langages, qui constituent en fait des langages de classes.
Un langage est orienté objet si, à la fois, il est basé *objets*, supporte la notion de *classes* et autorise l'*héritage*.

Quant aux langages de programmation orientée objet concurrente, ils mettent en jeu des processus qui peuvent s'apparenter à des acteurs. Dans cette famille de langages, chaque objet, appelé *acteur*, constitue un processus qui s'exécute de façon autonome, émettant et recevant des messages d'autres acteurs. Un acteur peut en créer un autre par simple reproduction ou copie de lui-même. La copie engendrée est alors soit rigoureusement identique, soit différente (spécialisée par adjonction de nouvelles caractéristiques). Un tel mécanisme s'apparente à l'instanciation des langages orientés objets.

Nous pouvons retrouver un récapitulatif sur l'organisation des langages de programmation, à la figure 1.1 suivante :

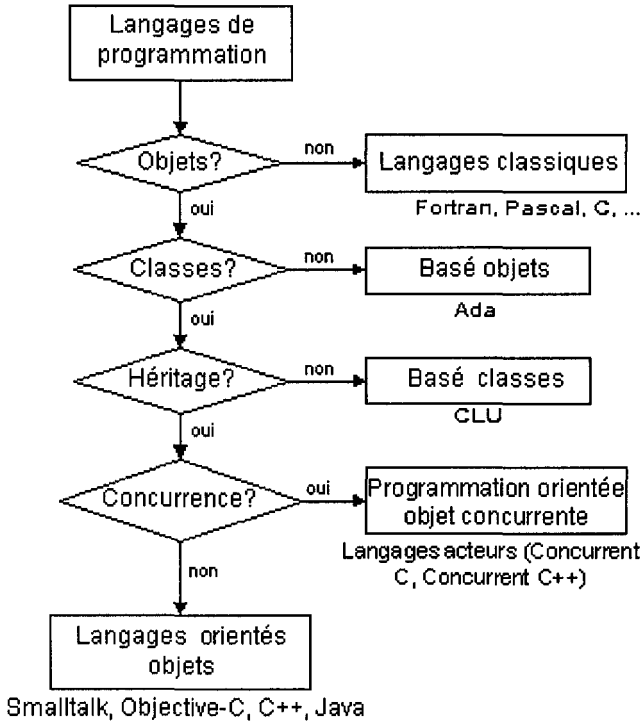


Figure 1.1 – Organisation des langages de programmation.

Nous allons maintenant présenter trois langages orientés objets :

- SMALLTALK : C'est seulement en 1972 qu'est apparue la première version du langage SmallTalk (Alan Kay) qui est le père spirituel des langages orientés objets modernes et le digne héritier de Simula et Lisp. SmallTalk a généralisé la notion d'objet qui devient plus tard l'entité unique de son univers. SmallTalk 72 introduit l'interaction via des messages. La notion d'héritage par hiérarchie de classes apparaît dans SmallTalk 76. SmallTalk 80 a définit une norme et a uniformisé le modèle en perfectionnant la définition de la classe par

l'introduction de la notion de méta classe. SmallTalk est plus qu'un langage de programmation, il est également synonyme d'environnement de programmation. C'est un langage objet pur puisque tout est objet et qu'il respecte la philosophie orientée objet. SmallTalk n'est pas typé (pas de contrôle de type à la compilation : typage dynamique). Il possède une hiérarchie de classes, basée sur l'héritage simple, qui comporte une racine unique *Object* de qui toute classe hérite. La hiérarchie comprend tous les éléments du système.

- C++ : Le langage C++ (B. Stroustrup) ajoute au langage C les notions de classes hiérarchisées et d'héritage. Presque entièrement compatible avec C, il est par ailleurs très influencé par Simula 67 et Algol 68. En C++, l'entité fondamentale est la classe plutôt que l'objet. La possibilité d'organiser les classes hiérarchiquement en bénéficiant d'un mécanisme d'héritage permet de programmer de manière orientée objet dans une très large mesure. C'est un langage riche et puissant qui facilite une approche structurée de la programmation, notamment en permettant de développer des composants logiciels réutilisables. Avec C++, il est possible de faire de la programmation classique ou de la programmation orientée objet, c'est pourquoi nous parlons d'un langage hybride.

- JAVA : Java est un nouveau langage orienté objet et dynamique, avec des éléments de C, C++ et d'autres langages. Il est considéré comme une version améliorée de C++ et comporte des librairies adaptées à l'environnement Internet, mais il renonce à quelques notions telles que les pointeurs et la surcharge des opérateurs. Selon Sun Microsystems, Java est conçu pour être un langage simple, orienté objet, distribué, interprété, fiable, robuste, sûr, indépendant de l'architecture matérielle, portable, performant et dynamique. Dans le même ordre d'idée, Java est aussi conçu pour distribuer du contenu exécutable sur les réseaux et, dans ce contexte, offre plusieurs véritables possibilités : comme par exemple

animer les pages Web et enrichir la présentation des informations à l'écran sous la forme de séquences animées et d'applications interactives.

1.2.5 Apports des concepts

Aujourd'hui, il existe un double enjeu dans le développement de logiciel. D'une part, un enjeu économique qui consiste à maîtriser les coûts de production et de maintenance, ainsi que les délais, etc. D'autre part, un enjeu technique dont l'objectif est la maîtrise du taux d'erreurs (maintenance, fiabilité, etc.) et le respect de normes (qualité, documentation, etc.) afin d'augmenter la qualité du produit. Pour relever ces défis, un logiciel doit respecter plusieurs critères de qualité tels que la modularité, la réutilisation, la facilité d'entretien, l'extensibilité, etc. Voyons ce qu'apportent les concepts de l'orienté objet.

1.2.5.1 Encapsulation

Elle constitue un excellent moyen pour la protection de l'information. De plus, vu de l'extérieur, un objet se caractérise uniquement par les spécifications de ses méthodes, ce qui donne un code plus compréhensif.

L'abstraction couplée avec l'encapsulation permet de réduire les interdépendances entre classes. Ainsi, l'implémentation d'un objet peut être modifiée sans affecter les applications qui emploient cet objet.

Elle réduit la complexité des applications, améliore leur modularité, facilite leur maintenance et augmente la réutilisation des composants logiciels.

1.2.5.2 Héritage

La plupart des modèles de conception ignorent cet aspect du développement de systèmes qui constitue, en revanche, l'un des points forts de l'approche par objets. En effet, pour réaliser des progrès en matière de réutilisation des composants logiciels et d'extensibilité, il est indispensable d'utiliser des relations entre les classes : une classe peut être une extension, une spécialisation ou une combinaison d'autres classes. C'est une technique efficace pour réutiliser et enrichir.

L'héritage induit une organisation hiérarchique des classes rendant plus aisée l'exploration et la maintenance d'une bibliothèque de classes (ensemble de classes fournissant un service particulier). Il facilite donc l'organisation des composants logiciels pour une équipe de programmeurs.

La technologie objet propose deux techniques liées à l'héritage visant à accroître la réutilisation : les classes abstraites (factorisation des propriétés communes de certaines classes) et l'héritage multiple.

L'héritage amène des avantages tels que la réutilisation de logiciels, le partage de code, l'uniformité des interfaces, des composants logiciels, un prototypage rapide, le masquage de l'information, etc. [BUDD 97] :

- **La réutilisation :** Lorsqu'un comportement est hérité d'une autre classe, le code n'a pas à être réécrit. Ceci réduit le temps passé à continuellement réécrire du code qui a déjà été écrit quelque part. Ainsi, la réutilisation augmente la fiabilité du code et réduit les coûts de maintenance.
- **Partage de code :** Plusieurs programmeurs ou projets peuvent utiliser les mêmes classes. De plus, plusieurs classes développées dans un projet peuvent hériter d'une même classe. Ainsi, plusieurs types d'objets partagent le code qu'ils héritent; ce code doit être écrit qu'une seule fois. Il y a donc économie du texte

source par factorisation de la partie commune de la définition de plusieurs classes.

- **Consistance des interfaces :** Lorsque deux ou plusieurs classes héritent de la même super classe, nous sommes assurés qu'elles auront les mêmes comportements. L'héritage permet donc d'unifier différents types d'objets sur la base de leurs ressemblances.
- **Composantes logicielles :** L'héritage permet au programmeur de fournir des composants réutilisables. Cela permet alors le développement d'applications nouvelles qui ne demandent presque pas de code. Il existe sur le marché de nombreuses librairies de ce type: «frameworks»...
- **Prototypage rapide :** Lors de la construction d'un système à partir de composantes réutilisables, le temps de développement est investit dans les nouvelles zones moins connues du logiciel. Il est alors possible de générer des prototypes rapidement permettant de mieux comprendre les besoins et les buts du système itérativement. Ceci apporte aussi une économie dans la conception et le développement de systèmes.
- **Masquage d'information :** Il permet de se concentrer sur la compréhension de la nature des abstractions disponibles et de leur interface. La compréhension des mécanismes d'implantation n'est pas nécessaire à la bonne utilisation de ces composantes. Ceci permet une modélisation plus naturelle qui accroît la modularité des programmes, ce qui facilite la mise au point et la maintenance.

1.2.5.3 Messages

Dans la technologie objet, le comportement de l'objet est de la responsabilité de l'objet lui-même, ce qui facilite la détection des erreurs.

1.2.6 Avantages de l'orienté objet

Nous retrouvons dans [BOUZ 98] que « l'approche objet apporte des avantages décisifs comme la modélisation des objets de l'application, la modularité, la réutilisabilité et l'extensibilité du code qui conduisent à une meilleure productivité des développeurs et à une plus grande qualité des applications ». En effet, les trois principes fondamentaux de l'orienté objet offrent des avantages directs tels qu'une hausse de compréhensibilité des designs et du code, un plus grand potentiel de réutilisation, et une hausse de la capacité d'évolution des designs et du code. Ils amènent aussi des avantages indirects comme une meilleure qualité du logiciel, une réduction des coûts liés au logiciel, et une réduction du temps de développement du produit. Voici les avantages, reliés à l'utilisation de l'orientée objet, plus en détail.

1.2.6.1 Hausse de compréhensibilité

Une hausse au niveau de la compréhensibilité est liée à différents facteurs. Tout d'abord, l'utilisation d'abstractions près de la réalité du domaine permet de réduire le fossé sémantique (écart entre le modèle et le monde réel). Dans le même ordre d'idée, la décomposition orientée objet produit des entités plus proches de notre façon de penser, et l'utilisation de patrons de design réduit les efforts de compréhension. La baisse de couplage, réalisée par l'encapsulation, réduit encore les efforts de compréhension d'une partie d'un système.

Ensuite, la décomposition orientée objet regroupe les données avec les services associés, ce qui évite de considérer séparément les modèles de données et les modèles fonctionnels. Ainsi, avec l'utilisation des mécanismes des objets (constructeur/destructeur, contrôle des accès, encapsulation, etc.) et les éléments de design directement spécifiés dans le code, la maintenance est plus facile.

Enfin, l'héritage et le polymorphisme offrent les avantages de compréhension suivants :

- Comprendre l'abstraction de base permet de comprendre 90% du design ainsi que le rôle de toutes les classes dérivées.
- Permet d'éliminer la plupart des traitements particuliers typiques des langages fonctionnels : traitement générique maintenant en orientée objet.
- Réduit le nombre d'abstractions à considérer dans la plupart des traitements.

1.2.6.2 Réutilisation et évolution

Commençons par définir ces deux termes :

- La réutilisation est la capacité de prendre un élément de design ou de code existant afin de l'utiliser dans un nouveau contexte ou projet.
- L'évolution est la capacité de modifier un élément de design ou de code existant afin de lui faire représenter un concept différent ou implémenter une fonctionnalité nouvelle ou différente.

En développement logiciel le problème à résoudre est souvent instable : le client change ses besoins ou veut plus de fonctionnalités ou veut un autre système du même domaine d'application. Mais l'encapsulation permet de plus facilement modifier une abstraction. En effet, la modification des structures de données internes est facile à réaliser pourvu que l'interface de la classe ne change pas. De plus, les abstractions obtenues ont la propriété d'être plus stables puisqu'elles représentent le domaine d'application. Et finalement, la baisse de couplage limite fortement les contraintes d'utilisation dans un autre contexte.

La capacité de réutilisation ou d'évolution est aussi améliorée grâce à l'héritage. Effectivement, il favorise naturellement la réutilisation par factorisation des propriétés et comportements communs, ce qui évite les répétitions d'attributs et de méthodes par

partage de données et de code. De plus, il permet la réutilisation et l'évolution par de petites adaptations au logiciel pour être utilisé dans un contexte autre que celui pour lequel il avait été originalement créé.

Une autre forme de réutilisation existe à travers des composants tels que :

- les classes génériques de C++ (templates) et la STL (Standard Template Library) qui permettent le partage de structures de données et d'algorithmes,
- les patrons de design qui permettent la réutilisation d'architecture logicielle,
- les frameworks qui permettent aussi de fournir un environnement de réutilisation pour une gamme de projets.

1.2.6.3 Hausse de la qualité

Nous pouvons augmenter la compréhensibilité, grâce à l'encapsulation et la cohésion de l'abstraction, ce qui a un effet direct sur la qualité du code produit et testé. Il est aussi possible d'avoir une meilleure qualité par réduction du volume des tests à développer, grâce à un couplage réduit inter-composants.

1.2.6.4 Réduction des coûts

La réutilisation diminue les coûts de développement. De plus, la réutilisation diminue les coûts de maintenance puisque nous pouvons utiliser des composants déjà testés, utilisés, ayant fait leurs preuves dans d'autres projets. Il faut toutefois considérer que la réutilisation n'est pas obtenue sans effort et sans planification, réutiliser a un coût.

1.2.6.5 Réduction du temps de développement

Le découpage orienté objet permet, en principe, une baisse du couplage et donc une optimisation du travail efficace sur plusieurs équipes indépendantes. La réduction du temps de développement ne se produit pas non plus facilement, il faut développer de nouvelles façons de planifier, de modéliser, de développer, etc. Changer les pratiques a un coût.

1.2.7 Limites de l'orienté objet

Les limitations principales du modèle objet se situent au niveau du mécanisme d'héritage. Mais voyons les obstacles majeurs apportés par l'orienté objet.

1.2.7.1 Rapidité d'exécution

Nous ne pouvons espérer obtenir autant de performance avec des outils génériques qu'avec une implantation spécifique. Par contre, la perte de performance est mineure par rapport à l'augmentation de la rapidité du développement. De plus, au début de la conception, nous ignorons souvent où la performance sera importante, il vaut alors mieux développer un système compréhensible et optimiser ensuite.

1.2.7.2 Dimension du code exécutable

L'utilisation de librairies réutilisables produit souvent des exécutables plus gros que les systèmes conçus spécifiquement pour un projet. Mais contenir les coûts, produire de la qualité et du code exempt d'erreur est plus important. D'autant plus que le prix de la mémoire a considérablement réduit.

1.2.7.3 Appel de méthode

Il y a un coût à payer pour un appel de méthode par rapport à un appel de procédure (~10%) mais il y a un gain à utiliser la technologie orientée objet. En C++, nous pouvons combattre en partie ce coût en résolvant statiquement certains appels et en utilisant le mécanisme des méthodes *inline*.

1.2.7.4 Complexité du programme

L'encapsulation et l'héritage permettent de modéliser des solutions de façon plus intuitive et naturelle, mais pas de développer des modèles clairs, simples, faciles à comprendre. En effet, bien que la programmation orientée objet soit reconnue comme une solution à la complexité du logiciel, la sur-utilisation de l'héritage crée de la complexité liée aux dépendances entre classes [SEBE 02]. Il y a donc perte de simplicité conceptuelle ainsi que perte de clarté dans l'implémentation. Pour comprendre un algorithme, il faut alors parfois faire le «yo-yo» dans la hiérarchie de classes [BUDD 97]. En effet, lorsque le graphe d'héritage devient complexe, son exploitation requiert beaucoup d'expérience. De plus, d'après [GAUT 96], « l'héritage multiple est une notion complexe et difficile à (bien) utiliser. Il pose encore de nombreux problèmes théoriques et pratiques, notamment au sujet des conflits de noms, qui sont inévitables ». Une autre complexité apparaît alors lorsqu'il y a héritage des mêmes attributs en plusieurs copies (deadly diamond), ce qui peut survenir sournoisement au cours de la maintenance et de l'évolution.

1.2.7.5 Réutilisation

Concernant la réutilisation potentielle apportée par les principes de l'orienté objet, il est à noter que le code qui nous intéresse peut être difficile à trouver et sans garantie de bon

fonctionnement. De plus, il est généralement problématique de détacher un bout de code d'un programme à cause des nombreuses dépendances qui y sont implantées, et le bout de code extrait peut demander d'importantes modifications pour fonctionner dans le nouveau programme. Il s'avère donc que c'est souvent plus facile de tout réécrire, d'autant plus qu'écrire un code réutilisable demande de la connaissance, de l'expérience et du travail.

1.2.8 Conclusion

La programmation orientée objet est une *nouvelle façon de penser* la décomposition de problèmes et l'élaboration de solutions. Les programmes deviennent un ensemble d'objets faiblement couplés qui communiquent par envoi de messages. Les concepts fondamentaux, de la programmation par objets se résument aux principes de décomposition hiérarchique, d'abstraction de données et à la relation d'héritage.

La décomposition hiérarchique permet le découpage d'une application en un ensemble de parties fonctionnellement indépendantes. Les langages à objet présentent un net progrès par rapport aux langages modulaires qui permettent la définition des modules mais pas leur création dynamique.

L'encapsulation ou l'abstraction de données spécifie que les objets sont simplement des données encapsulées avec les méthodes correspondantes à leurs manipulations. Ceci procure donc une protection des données de l'objet en interdisant tout accès «sauvage». L'accès est coordonné par l'emploi de messages de l'interface. De plus, l'encapsulation procure aux langages à objets une souplesse supérieure à celle des langages modulaires grâce à la possibilité d'association dynamique «message-méthode» et permet la co-existence de réalisations multiples d'une même interface.

La relation d'héritage permet la définition d'une classe d'objets par une spécialisation d'une autre classe (affinement). Ceci favorise la réutilisation, mais introduit de la

complexité. En fait, La programmation orientée objet *favorise* ou *facilite* la réutilisation en limitant les interdépendances entre les composants logiciels, en favorisant la construction de composants dont le rôle et le comportement sont clairement établis, et en mettant en place des mécanismes puissants d'abstraction comme l'héritage, l'encapsulation, et le polymorphisme qui permettent l'extension de composants sans les modifier.

Nous avons vu que les concepts orientés objet apportent des éléments intéressants pour favoriser et accélérer le processus de développement de logiciel, ainsi que pour augmenter la qualité des systèmes mis au point. Cependant, le paradigme objet est loin d'offrir une solution idéale puisqu'il engendre des coûts liés à l'héritage, qu'il n'assure pas une réutilisation facile des composants logiciels (bien qu'il la facilite), et qu'il ne permet pas de résoudre le problème de complexité lié aux grosses applications.

1.3 Programmation orientée agent

La programmation orientée agent est un nouveau paradigme de programmation basé sur le concept d'agent qui étend celui d'objet. Par la suite, nous sommes progressivement passés aux systèmes multi-agents (SMA) qui est un domaine de recherche en pleine effervescence.
Nous allons donc commencer par donner les origines de l'orienté agent. Puis, nous définirons ce qu'est un agent, et le situerons par rapport aux objets. Nous ferons ensuite une présentation des SMA. Enfin, nous conclurons avec les limites actuelles de la programmation orientée agent.

1.3.1 Origine de l'orienté agent

Depuis une dizaine d'années, la programmation orientée objet est présentée comme une technologie importante pour aider à la construction de systèmes complexes comme les

architectures réparties. Cependant, force est de constater que la technologie objet, malgré ses nombreux atouts — tels que l'encapsulation, le polymorphisme, la modularité — n'apporte pas toujours une réponse aux problèmes de conception et de développement. Ainsi, comme nous venons de le voir, le concept de réutilisabilité, présenté comme le fer de lance de la programmation orientée objet, est souvent mis en défaut.

Les systèmes d'aujourd'hui sont souvent distribués sur plusieurs machines. « Ils interagissent et communiquent entre eux et doivent aussi s'exécuter indépendamment les uns des autres. Ils sont souvent divisés en sous-systèmes (indépendants) qui exécutent chacun une partie du travail dans un but commun » [DELI 02]. Compte tenu des limites de l'orienté objet pour ces systèmes, la programmation orientée-agent fut proposée par Yoav Shoham [SHOH 93] comme un nouveau paradigme de programmation basé sur une « vue sociétale de la programmation ».

Le concept d'agent a été l'objet d'études pour plusieurs décennies dans différentes disciplines. Il a été non seulement utilisé dans les systèmes à base de connaissances, la robotique, le langage naturel et d'autres domaines de l'intelligence artificielle, mais aussi dans des disciplines comme la philosophie et la psychologie. Aujourd'hui, avec l'avènement de nouvelles technologies et l'expansion de l'Internet, ce concept est encore associé à plusieurs nouvelles applications comme *l'agent ressource, l'agent courtier, l'assistant personnel, l'agent interface, l'agent ontologique,* etc.

Le domaine de l'intelligence artificielle est à l'origine des systèmes multi-agents (SMA). L'expression intelligence artificielle (IA) est employée pour la première fois (1955-1970) par John McCarthy. Il fonde l'intelligence artificielle sur le postulat mécaniste qui veut que toute activité intelligente soit modélisable et reproductible par une machine. L'intelligence artificielle distribuée (IAD) est un sous-domaine de l'intelligence

artificielle qui s'occupe des situations où plusieurs systèmes interagissent pour résoudre un problème commun [MOUL 96]. L'IAD se divise en deux branches principales :

- La résolution distribuée de problèmes (RDP) qui étudie comment distribuer des compétences au niveau de chaque partie du système, de façon à ce qu'il soit globalement plus compétent que chacune de ses parties.
- La simulation des systèmes complexes (SSC), qui concerne plus particulièrement les systèmes multi-agents. Les SMA traitent le comportement d'un ensemble d'agents autonomes qui essaient de résoudre un problème commun.

La différence notable entre la RDP et les SMA est que la RDP possède une approche descendante («top-down») et les SMA une approche ascendante («bottom-up»).

1.3.2 Définition d'un agent

Le concept d'agent vient étendre celui d'objet. En effet, un agent possède les caractéristiques des objets, mais est en plus capable d'indépendance et d'une certaine intelligence. Comparativement a un objet, un agent peut refuser (ou accepter) d'exécuter le message. Le refus peut être dû :

- à un manque de compétence,
- à une surcharge de travail,
- au non respect de ces objectifs.

L'agent peut éventuellement proposer de déléguer le travail à d'autres agents. Un agent est dirigé par des buts à satisfaire. Donc on peut en conclure que les objets ne sont pas des agents car ils n'ont aucun but et le mécanisme d'envoi de message se résume à un appel de procédures.

De plus, bien qu'il existe une similarité superficielle entre objet et agent, la terminologie objet n'est pas adaptée aux systèmes agents :

- les objets sont généralement passifs alors que les agents sont permanents actifs,

- les agents sont autonomes et responsables de leurs actions alors que les objets n'en sont toujours pas,
- on ne peut prévoir tous les comportements des agents dans les systèmes,
- l'approche orientée-objet ne fournit pas un ensemble adéquat de concepts et de mécanismes pour modéliser les systèmes complexes dans lesquels les rapports évoluent dynamiquement,
- certains chercheurs définissent un agent comme un objet actif ayant une autonomie et un objectif.

Les agents se divisent en deux catégories. Tout d'abord, les agents cognitifs qui sont des agents augmentés d'un état mental : croyances, désirs, intention. Ils ont également une entité intelligente c'est-à-dire qu'ils ont une reproduction d'un certain savoir-faire humain. Ensuite, les agents mobiles qui sont des agents capables de se déplacer sur le réseau pour s'exécuter sur un site et de coopérer avec d'autres agents sur d'autres sites.

Dans la littérature, on trouve une multitude de définitions d'agents. Elles se ressemblent toutes, mais diffèrent selon le type d'application pour laquelle est conçu l'agent. Nous trouvons dans [FIKE 71] une discussion sur les différentes définitions attribuées aux agents ainsi que la différence entre un agent et un programme classique. À titre d'exemple, voici l'une des premières définitions de l'agent due à Ferber [FERB 95] : « Un agent est une entité autonome, réelle ou abstraite, qui est capable d'agir sur elle-même et sur son environnement, qui, dans un univers multi-agent, peut communiquer avec d'autres agents, et dont le comportement est la conséquence de ses observations, de ses connaissances et des interactions avec les autres agents ». Il ressort de cette définition des propriétés clés comme l'*autonomie*, l'*action*, la *perception* et la *communication*. D'autres propriétés peuvent être attribuées aux agents. Nous citons en particulier la *réactivité*, la *rationalité*, l'*engagement* et l'*intention*.

Dans la communauté agent, certains chercheurs tentent de définir les caractéristiques essentielles d'un agent. Nous retrouvons dans un article de Stan Franklin et Art Graesser [FRAN 97] une comparaison de plusieurs définitions connues d'un agent. Nous retrouvons les définitions de Wooldridge & Jennings, de Pattie Maes, de Barbara Hayes-Roth, de Lenny Foner, de Brustoloni, ainsi que celles d'organismes ou compagnies comme IBM. Ils en arrivent à donner leur propre définition : « An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future ». Leur définition décrit un agent autonome, distinct d'un programme. Mais pour savoir si un système est bien un agent autonome, ils préconisent de décrire son environnement, ses capacités de perception, ses actions, ses objectifs, et l'architecture de sélection des actions.

Dans [MULL 96] nous retrouvons une série d'articles qui cherchent à encadrer la notion d'agent grâce à une taxonomie partagée par la plupart des chercheurs et nous permettent de mieux cerner la notion d'agent. L'article de Michael Wooldridge (1996), « Agents as a Rorschach test : a response to Franklin and Graesser », souligne la notion d'autonomie, de réactivité, et d'habilité sociale qui sont pour lui des caractéristiques essentielles d'un agent que nous ne retrouvons pas dans la définition de Franklin et Graesser.

Récemment, Jennings, Sycara et Wooldridge [JENN 98] ont proposé la définition suivante pour un agent : « Un agent est un système informatique, *situé* dans un environnement, et qui agit d'une façon *autonome* et *flexible* pour atteindre les objectifs pour lesquels il a été conçu ».

Les notions "situé", "autonome" et "flexible" sont définies comme suit:

- *situé* : l'agent est capable d'agir sur son environnement à partir des entrées sensorielles qu'il reçoit de ce même environnement. Exemples : systèmes de contrôle de processus, systèmes embarqués, etc.

- *autonome* : l'agent est capable d'agir sans l'intervention d'un tiers (humain ou agent) et contrôle ses propres actions ainsi que son état interne.
- *flexible* : l'agent dans ce cas est *capable de répondre à temps* (l'agent doit être capable de percevoir son environnement et élaborer une réponse dans les temps requis), *proactif* (l'agent doit exhiber un comportement proactif et opportuniste, tout en étant capable de prendre l'initiative au "bon" moment), et *social* (l'agent doit être capable d'interagir avec les autres agents, logiciels et humains, quand la situation l'exige afin de compléter ses tâches ou aider ces agents à accomplir les leurs).

D'après [WOO 00], un agent est un système informatique encapsulé situé dans un environnement dans lequel il est capable d'effectuer une action flexible et autonome, compatible avec les objectifs de la conception. Les agents sont :

- des entités clairement identifiables de résolution de problèmes avec des bornes et des interfaces bien définies,
- situés dans un environnement particulier ; ils reçoivent des entrées liées aux états de cet environnement par des capteurs et agissent sur cet environnement par des émetteurs,
- destinés à atteindre un objectif spécifique,
- autonomes et responsables de leur comportement,
- capables d'adopter un comportement flexible pour résoudre des problèmes selon les objectifs de la conception; ils sont réactifs (capables de s'adapter aux changements d'état de leur environnement) et proactifs (capables d'adopter un nouvel objectif),
- capables dans un univers multi-agents, de communiquer, coopérer, se coordonner, négocier les uns avec les autres.

La figure 1.2 suivante donne, de façon générale, l'architecture interne d'un agent.

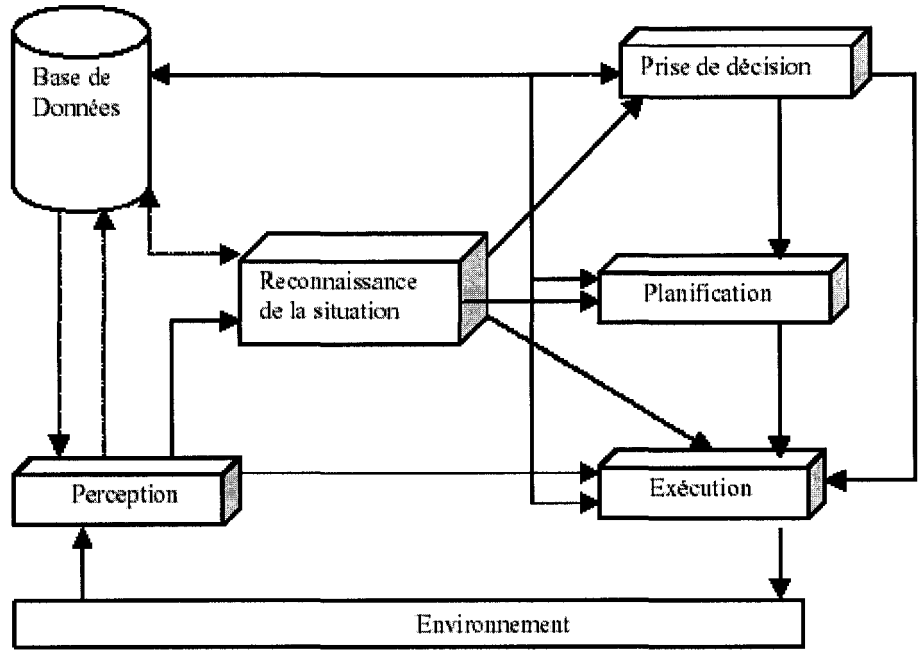


Figure 1.2 - Architecture d'agent.

Lorsqu'un agent perçoit une situation dans l'environnement, il essaie de la reconnaître. Si la situation lui est familière, il peut enclencher un processus de planification afin de résoudre le problème. Il peut aussi reconnaître la situation en terme d'action et donc, passe à l'exécution de la tâche (Reconnaissance- Exécution). Lorsque l'agent perçoit des situations qu'il connaît très bien, il peut faire intervenir son comportement réactif en passant directement à l'action (Perception-Exécution). S'il ne peut pas résoudre un problème (situation non-familière), il engage un processus de coopération pour demander de l'aide aux autres agents (Reconnaissance-Prise de décisions).

Bien entendu, suivant les applications certaines propriétés sont plus importantes que d'autres, il peut même s'avérer que pour certains types d'applications, des propriétés additionnelles soient requises. Il convient cependant de souligner que la présence des propriétés qu'on vient de voir comme l'autonomie, la flexibilité, la sociabilité, etc.,

donne naissance au paradigme agent tout en le distinguant des systèmes conventionnels comme les systèmes distribués, les systèmes orientés objets et les systèmes experts.

Le débat sur la définition d'un agent et les moyens de caractériser un système d'agent semble ouvert. Nous voyons que ce milieu de recherche est encore jeune et cherche à bien délimiter la problématique reliée aux agents.

1.3.3 Systèmes multi-agent

Nous retrouvons dans [CHAI 02], qu'un *système multi-agent (SMA)* est un système distribué composé d'un ensemble d'agents. Contrairement aux systèmes d'IA, qui simulent dans une certaine mesure les capacités du raisonnement humain, les SMA sont conçus et implantés idéalement comme un ensemble d'agents interagissants, le plus souvent, selon des modes de *coopération*, de *concurrence* ou de *coexistence* [CHAI 94], [CHAI 96], [MOUL 96].

Un SMA est généralement caractérisé par le fait :

- que chaque agent a des informations ou des capacités de résolution de problèmes limitées, ainsi chaque agent a un point de vue partiel,
- qu'il n'y a aucun contrôle global du système multi-agent,
- que les donnés sont décentralisées,
- que le calcul est asynchrone.

Les SMA sont actuellement très largement utilisés, particulièrement pour les applications complexes nécessitant l'interaction entre plusieurs entités. Dès lors, il est devenu impératif de s'investir dans les méthodologies orientées-agent et autres techniques de modélisation adéquates. Plusieurs méthodologies ont été proposées pour le développement des SMA. Ces méthodologies constituant soit une extension des méthodologies orientées-objet, soit une extension des méthodologies à base de

connaissances, demeurent incomplètes. Par ailleurs, peu d'efforts ont été faits en matière de standardisation des plates-formes SMA. Il apparaît donc évident que le développement des SMA reste encore un domaine ouvert.

1.3.4 Conclusion

Au regard des caractéristiques d'agent, il apparaît que l'approche orientée-agent dans le développement de logiciel consiste en une décomposition du problème en agents ayant des interactions, une autonomie, et un objectif spécifique à atteindre. Les concepts clés d'abstraction liés à un système orienté-agent sont : agent, interaction, organisation.

L'orienté agent est une discipline qui s'intéresse aux comportements collectifs produits par les interactions de plusieurs entités autonomes et flexibles appelées agents, que ces interactions tournent autour de la coopération, de la concurrence ou de la coexistence entre ces agents.

Comme nous venons de le voir, les SMA sont à l'intersection de plusieurs domaines tels que l'intelligence artificielle, les systèmes distribués, le génie logiciel, ... C'est une discipline qui s'intéresse aux interactions entre plusieurs agents dont l'objectif est de résoudre un problème commun.

Cependant, il n'existe pas encore d'environnement de développement complet pour la mise au point de SMA. En effet, d'après [DELI 02], « les outils existants ne permettent pas encore le développement complet, et ce de façon relativement simple, de SMA appliqués à des systèmes réels d'envergure intéressante ». La programmation orientée agent est un domaine encore récent qui nécessite la mise au point d'outils de développement adaptés aux SMA, et qui exige des méthodologies systématiques pour la spécification et la conception des applications SMA.

1.4 Programmation fonctionnelle

Depuis les critiques, par J. Backus [BACK 1978], comme quoi les langages de programmation classiques (impératifs) sont trop liés aux architectures des machines de type Von Neumann, un nouveau style de programmation est né : la programmation applicative ou fonctionnelle.

Un programme fonctionnel est constitué d'une séquence de déclarations de fonctions. Les langages fonctionnels font donc appel aux fonctions comme composant de base. On parle parfois de langages applicatifs pour souligner que l'on applique une fonction à une expression. Ces langages ont connu une remarquable évolution en quelques années.

Après avoir présenté d'où vient la programmation fonctionnelle et quels sont les langages qui lui sont associés, les éléments qui font l'essence de la programmation fonctionnelle seront discutés : la notion de fonction, l'ordre normal d'évaluation et la transparence référentielle. Ensuite, nous verrons ce qui caractérise cette approche, avec ce qui tourne autour de la notion de type et de fonction. Enfin, nous regarderons la stratégie d'évaluation utilisée dans ces langages, avant de conclure.

1.4.1 Origine de la programmation fonctionnelle

La programmation fonctionnelle ou applicative constitue un style de programmation aussi ancien que la programmation impérative mais dont l'impact ne s'est considérablement accru que depuis le début des années 1980. Les langages fonctionnels trouvent leur origine dans les travaux de G. Frege [FREG 1893], et voient le jour avec l'article historique de M. Schönfinkel [SCHO 1924]. Ils sont basés sur le lambda-calcul inventé par Alonzo Church [CHUR 1941] pour étudier la calculabilité.

La programmation avec ces langages est très proche des mathématiques traditionnelles. Ces langages réduisent donc le fossé conceptuel entre la programmation et les mathématiques. Ils utilisent une notation qui ressemble à celle des mathématiques et est donc concise et de haut niveau, permettant l'écriture rapide de programmes complexes. De surcroît, il est aisé de raisonner formellement sur ces programmes qui par ailleurs peuvent être plus facilement exécutés sur des architectures parallèles.

Parmi les ancêtres des langages fonctionnels actuels, on peut citer APL ainsi que FP, qui fut introduit par Backus (1978). Ce dernier a popularisé FP à la suite de son célèbre Turing Award lecture qui, ironiquement, lui avait été attribué pour son travail sur Fortran.

Les seuls exemples de langages déclaratifs impurs à avoir obtenu une assez large diffusion sont Lisp (pour la programmation fonctionnelle) et Prolog (pour la programmation logique). En plus d'être impur, il faut mentionner que Lisp (1959) n'est pas un langage fonctionnel moderne. Il a cependant eu une influence marquée et continue d'être utilisé et enseigné dans diverses variantes plus ou moins répandues, dont Scheme (1975).

Le langage ML (1987) est un représentant relativement moderne du courant des langages essentiellement fonctionnels, ce qui signifie qu'il n'est pas pur mais inclut certaines constructions impératives. Il limite cependant les affectations à une certaine classe d'identificateurs, mais utilise en outre des primitives d'entrée-sortie non référentiellement transparentes. Seul langage fonctionnel répandu à inclure les exceptions et une notion de modules génériques considérés comme des valeurs de (presque) première classe, ML a introduit des éléments marquants du typage moderne et de l'inférence de type. Il en existe aujourd'hui plusieurs variantes, dont la plus connue est probablement Standard ML.

La famille des langages fonctionnels comporte en fait 2 groupes, d'inégales importances :

- Les langages fonctionnels purs, sans variables, fondés sur la logique combinatoire [CUR 58], définie par Schönfinkel dès les années 1920. Il n'y a que peu de langages fonctionnels purs, les plus connus étant probablement Miranda et Haskell (1990).
- Les langages avec variables, les plus nombreux, fondés sur le lambda-calcul de Church [CHU 41].

Nous allons débuter par ce qui fait l'essence de la programmation purement fonctionnelle. Ce courant, qui s'est renforcé depuis l'apparition de langages appropriés, s'oppose à ceux qui préconisent un simple style fonctionnel, même entaché de quelques aspects impératifs.

1.4.2 L'essence de la programmation fonctionnelle pure

La première caractéristique essentielle d'un langage fonctionnel pur est bien sûr d'offrir la notion de fonction. La transparence référentielle et un ordre normal d'évaluation sont également comptés dans les caractéristiques essentielles.

1.4.2.1 Structure d'un programme fonctionnel

Un programme fonctionnel consiste en un ensemble de définitions et en une expression destinée à être évaluée. La notion d'exécution d'un programme est remplacée par celle d'évaluation d'une expression. Une expression est utilisée uniquement pour décrire (dénoter) une valeur. Autrement dit, la signification d'une expression est sa valeur, et il n'y a aucun autre effet obtenu durant le processus accompli pour parvenir à cette valeur.

L'évaluation d'une expression consiste en sa simplification ou réduction, jusqu'à obtenir la forme la plus simple qui est alors considérée comme la représentation de la valeur qui est le résultat. En effet, l'évaluation d'une expression est simplement un jeu de substitutions et de simplifications qui se fonde sur les définitions fournies par le programmeur et certaines définitions primitives. Seul le résultat de la réduction est important, le chemin suivi importe peu et ne peut d'ailleurs pas être précisé par le programmeur. L'évaluateur du langage choisit lui-même l'ordre des réductions à accomplir et l'absence d'effet de bord implique que ces réductions ne produiront aucun effet observable tant qu'elles ne fournissent pas la valeur résultante.

D'une manière générale, les programmes sont peu verbeux et proche d'une notation mathématique. En particulier, il n'y a généralement pas d'indication de type et peu de séparateurs ou de mots réservés.

1.4.2.2 Des citoyens de première classe : les fonctions

Une fonction associe une valeur unique à chaque valeur à laquelle elle est appliquée. Nous disons qu'elle prend un argument (reçoit une entrée) et retourne un résultat (produit une sortie) de telle manière que chaque entrée détermine entièrement la sortie produite. Une fonction n'a pas d'état, pas de mémoire qui pourrait modifier son résultat suivant l'histoire passée. Le résultat de l'application ne peut dépendre que de la définition statique de la fonction et de la valeur à laquelle elle est appliquée.

Nous définissons une fonction de façon abstraite en donnant ses propriétés. La définition d'une fonction d'un langage fonctionnel est donc proche de la définition d'une fonction mathématique. Le lambda-calcul en est la théorie sous-jacente.

Un avantage de cette approche est qu'elle traite les fonctions comme des citoyens de première classe car elle leur confère tous les droits et privilèges des objets mathématiques. Les fonctions sont traitées comme des données, elles peuvent être passées comme arguments, construites et retournées comme valeur, et composées avec d'autres formes de données. Une définition de fonction est donc évaluable. Les fonctions peuvent être récursives et polymorphes et une série de constructions syntaxiques est disponible pour faciliter leur définition et leur utilisation. Comme n'importe quelle valeur, une fonction peut notamment être manipulée de manière anonyme (sans qu'un nom lui soit attribué) et être stockée dans une structure de données.

1.4.2.3 Ordre normal d'évaluation

Grâce à la théorie du lambda-calcul, il est possible de décrire précisément le comportement d'une fonction au moyen d'une lambda-expression. Le lambda-calcul est une théorie formelle introduite par Church vers 1930, et dont la syntaxe et la sémantique sont bien définies.

Le lambda-calcul définit une série de conversions possibles pour des lambda-expressions de manière à pouvoir les transformer et les simplifier. Ces règles permettent de simplifier une lambda-expression pour aboutir à une forme normale, ne contenant plus aucune expression réductible. Une expression sous forme normale possède une interprétation mathématique directe qui décrit le sens de l'expression originale. L'idée est donc de définir la sémantique du lambda-calcul en réduisant d'abord toute expression à sa forme normale pour laquelle une dénotation mathématique simple existe. Cependant, il y a plusieurs stratégies possibles de réduction (dès que l'on a plusieurs radicaux, candidats simultanés à la réduction) et il faut savoir que certaines expressions ne se laissent pas réduire.

Les théorèmes de Church-Rosser nous apprennent que :

- La forme normale d'une lambda-expression est unique. En conséquence, l'ordre d'évaluation n'a pas d'importance quant au résultat obtenu (la séquence de réduction n'influence pas la forme normale obtenue, si celle-ci est trouvée).
- S'il existe une forme normale, on peut toujours la trouver au moyen d'une stratégie appelée réduction en ordre normal. Cette stratégie consiste essentiellement en une réduction qui se fait d'abord sur le membre le plus extérieur d'une expression, contrairement à l'intuition habituelle.

Ceci exprime donc un fondement théorique important : il y a au plus un résultat possible, et l'ordre normal de réduction y aboutira, si ce résultat existe. Aucune suite de réductions ne peut aboutir à un résultat incorrect, le pire qui puisse arriver est la non-terminaison, qui correspond au cas d'un programme impératif à boucle sans fin.

Ces deux résultats importants sont complétés par la démonstration que le lambda-calcul est suffisamment expressif pour exprimer toutes les fonctions calculables, et qu'il est donc universel. En d'autres termes, n'importe quelle structure traditionnelle de donnée ou de contrôle peut être simulée dans le lambda-calcul.

Le lambda-calcul est donc le prototype des langages fonctionnels purs, et les langages modernes en restent très proches. Les adjonctions sont davantage le fait de commodités syntaxiques que de bouleversements sémantiques. Il a d'ailleurs souvent été remarqué que la plupart des langages fonctionnels se ressemblent et diffèrent plus par leur syntaxe que par leur sémantique.

1.4.2.4 Définition de fonctions

Il est important de considérer les équations de définition d'une fonction comme étant d'abord des équations mathématiques plutôt que des algorithmes de programmation. Mais nous ne pouvons pas pour autant oublier que ces équations représentent aussi une règle de réduction dans une expression et présentent donc également un aspect opérationnel. En particulier, le signe égal utilisé dans cette définition a une direction : il n'est pas possible d'intervertir les membres droit et gauche d'une équation de définition, alors que ceci serait possible dans une équation mathématique. Il s'agit en fait de règles de réécriture selon lesquelles la partie gauche doit être remplacée par la partie droite après la substitution appropriée des arguments. De telles règles sont algorithmiques, mais préservent le sens des expressions manipulées grâce à la transparence référentielle.

1.4.2.5 Transparence référentielle

La transparence référentielle se trouve au cœur des différences entre les notations mathématiques usuelles et les programmes impératifs conventionnels. C'est la propriété, possédée par la notation mathématique, de substitution d'égal à égal. Cela signifie que si A = B, toute propriété obtenue à partir d'une propriété vraie en remplaçant A par B sera encore vraie. Cette propriété joue bien sûr un rôle fondamental dans le raisonnement mathématique. Les manipulations de formules en algèbre, en arithmétique et en logique reposent constamment sur elle. Cependant, un langage de programmation viole la transparence référentielle s'il autorise des effets de bord, ce qui est le cas des langages impératifs ainsi que des langages fonctionnels impurs.

La transparence référentielle est une exigence essentielle du paradigme fonctionnel. La sauvegarde rigoureuse de cette propriété est avant tout un garde-fou : elle permet d'éviter les pièges de la programmation impérative. Ce qui, dans la terminologie Pascal,

Ada ou C est appelé fonction n'est pas une fonction au sens mathématique du terme. Dans ces langages une fonction peut reposer sur des effets de bord et peut donc livrer des résultats différents lors de plusieurs invocations, malgré des arguments identiques. Par ailleurs la synonymie (création d'alias) peut aussi transgresser la transparence référentielle, par exemple s'il existe des alias entre variables globales et arguments.

Nous pouvons certes considérer que les effets de bord sur des variables globales ne sont que le résultat d'une mauvaise programmation. Mais en fait, le problème surgit dès qu'un langage comporte des variables et des affectations. Si plus d'une affectation est autorisée sur la même variable X, nous ne pouvons pas utiliser le fait que X = A à un endroit du programme pour en conclure une propriété de X à partir d'une propriété de A en un autre point. L'affectation à une variable de multiples valeurs dans le temps est donc incompatible avec la transparence référentielle et empêche d'appliquer aux données le raisonnement mathématique habituel. Il a ainsi pu être dit que l'affectation est aux données ce que le « goto » est à la structure de contrôle.

L'affectation, telle x := x + 1, est une instruction : elle n'affirme pas une propriété mais ordonne à la machine de changer quelque chose dans son fonctionnement interne, la valeur associée à la variable x. L'affectation est la plus simple des instructions fonctionnant par effet de bord sur le déroulement du programme. Ce rôle central est spécifique à la programmation impérative. Il n'existe pas d'effet de bord en mathématiques classiques : x = 1 affirme une égalité mais ne tente pas de changer quoi que ce soit. La véritable notion de variable au sens de la programmation traditionnelle (un objet dont la valeur peut être changée à volonté) est étrangère aux mathématiques usuelles : x = x +1 n'a pas de sens, sauf pour des informaticiens qui se sont tordus l'esprit suffisamment longtemps avec des langages impératifs.

Par opposition à cet état de fait, les pures fonctions sont référentiellement transparentes. Cette propriété se traduit dans la pratique par le raisonnement équationnel : des quantités

égales sont interchangeables, c'est-à-dire qu'à des quantités quelconques nous pouvons toujours substituer des quantités égales. Il s'ensuit que les langages fonctionnels purs interdisent toute affectation, la notion même d'affectation n'existant pas. Cette contrainte n'est cependant pas suffisante car il faut également veiller à ce que les opérations d'entrée-sortie soient référentiellement transparentes.

En conséquence, il n'y a pas de variables, ou pour être plus précis, les variables ne sont pas modifiables! Il s'agit donc bien de variables au sens mathématique et non au sens informatique. La valeur d'une variable ne peut être définie qu'une seule fois dans sa portée et cette variable conservera cette valeur immuable jusqu'à ce qu'elle cesse d'exister. Une variable est donc simplement un moyen de nommer des valeurs intermédiaires.

La transparence référentielle est une idée simple mais extrêmement puissante : elle n'est pas seulement décisive lors du raisonnement formel mais également d'une grande aide pour le raisonnement informel. Il en résulte de grands bénéfices lors de l'écriture et du test de programmes car la transparence référentielle permet de considérer un texte de programme d'une manière statique, non pas en effectuant mentalement une séquence d'opérations mais en l'interprétant comme un énoncé statique de propriétés vraies. À noter que, dans le paradigme impératif, l'introduction de la programmation structurée a permis un premier pas vers ce même but. La transparence référentielle rend la sémantique des langages fonctionnels purs beaucoup plus simple que la sémantique des langages impératifs.

1.4.3 Types

Malgré l'absence d'indication de type, les langages fonctionnels purs utilisent un typage statique fort. Toute expression a un type unique, qui peut être déterminé statiquement à partir des composants de l'expression, et aucune erreur de type ne peut survenir durant

son évaluation. Cependant, ce système de type est très riche car il inclut plusieurs polymorphismes.

Nous retrouvons bien sûr des types de base (prédéfinis) tels que des types pour valeurs numériques, booléennes ou de caractères. De plus, nous disposons de constructeurs de types permettant par exemple de définir des structures qui regroupent un ensemble de valeurs et les manipulent comme une valeur unique.

L'utilisateur peut définir de nouveaux types au moyen de ces constructeurs de types, en utilisant une syntaxe qui est commune aux expressions de valeurs et aux expressions de types. En fait, il existe un véritable langage d'expressions pour dénoter des types. Ce langage contient des constantes prédéfinies (par exemple « num », « char », « bool »), des variables désignant n'importe quel type (a, b, c) ainsi que des opérateurs permettant de les combiner (→, (), []). Une expression contenant une variable de type est dite polymorphe. Beaucoup de types sont anonymes mais peuvent optionnellement recevoir un nom.

Finalement, un type peut aussi être défini comme abstrait, cachant ainsi sa structure.

1.4.3.1 Type fonction

Les valeurs les plus importantes en programmation fonctionnelle sont les valeurs de type fonction. Les types fonctions doivent être considérés comme abstraits, avec une unique opération prédéfinie qui est l'application, c'est-à-dire – en termes impératifs – l'appel de cette fonction. L'évaluation d'une valeur fonction ne conduit à aucune simplification car une fonction n'a pas de forme normale vers laquelle elle pourrait être réduite.

1.4.3.2 Types algébriques (types structurés ou concrets)

Le programmeur peut introduire de nouveaux types au moyen de définitions algébriques. Il existe différentes catégories de construction de types dont les types énumératifs, les unions de types, les types polymorphes, les types récursifs, les listes, les tableaux, etc.

Le système de type des langages purement fonctionnels ne requiert pas de pointeurs. L'heureuse conséquence pour l'utilisateur d'un langage fonctionnel est l'uniformité conceptuelle. Il n'y a pas à se soucier de sémantique par valeur ou par référence, il n'y a qu'une sémantique uniforme par valeur (même si le compilateur utilise sûrement une implémentation par référence). Cela n'empêche pas la déclaration de structures de données complexes. Au contraire, les types utilisés sont d'un plus haut niveau d'abstraction et se rapprochent des définitions mathématiques correspondantes.

1.4.3.3 Inférence de types

Les langages fonctionnels modernes font usage d'une autre particularité dans le domaine du typage : l'inférence de type. Chaque valeur a un type, mais le programmeur n'est pas obligé de le spécifier car le compilateur peut le déterminer lui-même.

Les seuls endroits dans un programme où le programmeur doit faire explicitement référence aux types sont dans les définitions de types elles-mêmes. Le compilateur détermine ensuite le type de chaque valeur en déduisant le type des expressions du programme d'après leur contexte d'utilisation. Le vérificateur de types est capable de déterminer si le programme est correctement typé, et, si c'est le cas, de déterminer le type de toutes les expressions du programme.

L'algorithme d'inférence de type détermine pour chaque fonction le type le plus général possible, et tente donc de rendre chaque fonction polymorphe. S'il le souhaite, le programmeur peut annoter ou contraindre une définition à un type plus restrictif. L'inférence de type inclut bien sûr l'ensemble des types, y compris ceux qui sont anonymes ou définis par le programmeur.

Par opposition à ce que nous trouvons en Ada avec la généricité, ou en C++ avec les « templates », cela signifie que le compilateur cherche lui-même la plus grande généralité (et donc réutilisabilité) des fonctions, et met à disposition l'ensemble (généralement infini) des instanciations possibles. Par opposition au polymorphisme paramétrique syntaxique d'Ada et de C++, c'est ici un polymorphisme paramétrique sémantique, discuté dans la prochaine section.

1.4.3.4 Polymorphisme des fonctions (polymorphisme paramétrique sémantique)

Beaucoup de fonctions sont complètement indifférentes au type de leurs arguments, comme par exemple la fonction d'identité, id x = x, qui prend en argument n'importe quel type et retourne une valeur de ce même type.

L'inférence de type permet donc de déterminer l'usage le plus général qu'il est possible de faire d'une fonction, non pas sur la base de déclarations que devrait explicitement faire le programmeur, mais sur la base de la définition qu'il a effectivement retenue pour la fonction.

1.4.3.5 Surcharge des fonctions (polymorphisme ad-hoc)

Un traitement systématique de la surcharge est également disponible, compatible avec l'inférence de type. L'idée est d'introduire une notion de classe de type qui désigne un

ensemble de fonctions surchargées. Le but est essentiellement de rendre le polymorphisme par inclusion (retrouvé en particulier dans les langages par objets) compatible avec le typage statique. Une classe de type définit un ensemble d'opérations qui doivent être disponibles pour chaque type de cette classe, et fournit optionnellement une implémentation pour ces opérations. Cette implémentation est héritée par tous les types de la classe, mais peut être remplacée par une implémentation spécifique. Un type peut faire partie de plusieurs classes (héritage multiple). Le mécanisme de l'inférence de type va exprimer les types obtenus en mentionnant les contraintes de classes qui pèsent sur eux, c'est-à-dire en indiquant de quelles classes (les plus générales possibles) ils doivent être instances.

1.4.4 Définitions de fonctions

Précisons encore que les langages fonctionnels modernes adhèrent à deux principes de complétude :

- Complétude syntaxique : il n'y a pas de cas particuliers dans la syntaxe du langage. Toute expression syntaxiquement valide peut remplacer une expression donnée. Il n'y a pas de restrictions qui imposent des constantes ou des expressions statiques en des endroits donnés.
- Complétude sémantique : tous les objets manipulés sont des citoyens de première classe, ils ont tous les mêmes droits :
    - être nommés,
    - être la valeur d'une expression,
    - être argument d'une fonction,
    - être résultat d'une fonction,
    - être élément d'un quelconque type.

  Les citoyens de seconde classe sont les types et les modules.

1.4.4.1 Curryfication

La curryfication est un mécanisme dont le nom vient du mathématicien Haskell B. Curry et qui nous permet de voir toutes les fonctions comme prenant un seul argument. L'application de fonction est l'opération la plus commune d'un programme fonctionnel et est associative à gauche de sorte que f x y soit identique à (f x) y, ce qui signifie que le résultat de l'application f x est une fonction que l'on applique à y.

Le mécanisme de curryfication joue un rôle crucial car il est toujours utilisé, même si le programmeur ne le réalise pas et se représente les fonctions qu'il définit comme prenant de multiples arguments. Conceptuellement, nous ne considérons que des fonctions n'ayant qu'un seul argument, et toute fonction ayant de multiples arguments est décomposée en fonctions à argument unique. Nous devinons là un des mécanismes de base de la composition des programmes fonctionnels.

1.4.4.2 Fonction d'ordre supérieur (forme fonctionnelle)

Une fonction d'ordre supérieur, ou forme fonctionnelle, est une fonction qui admet des fonctions parmi ses arguments, ses résultats ou les deux. Comme nous l'avons déjà vu dans le cadre de la curryfication, les fonctions d'ordre supérieur forment une caractéristique très importante du style fonctionnel, permettant souvent une grande concision et favorisant la modularité par la possibilité de composer des fonctions et d'abstraire un comportement fonctionnel donné. Un exemple classique de forme fonctionnelle est la fonction de composition qui prend deux fonctions comme paramètres.

1.4.5 Stratégie d'évaluation

Dans la programmation fonctionnelle, l'évaluation d'une expression – qui correspond à l'exécution d'un programme impératif – consiste en l'application de règles de réduction pour aboutir à une forme normale, soit une forme canonique représentant la valeur finale. Cette forme normale est unique et la séquence de réduction particulière choisie n'influence pas le résultat, si celui-ci existe. De plus, si une forme normale existe, elle peut toujours être trouvée au moyen de la réduction en ordre normal, qui est une stratégie particulière de réduction.

1.4.5.1 Évaluation paresseuse

Cependant l'implémentation naïve de cette stratégie est inefficace. C'est pourquoi la réduction en ordre normal est réalisée au moyen d'une technique d'implémentation particulière appelée l'évaluation paresseuse (lazy evaluation). L'évaluation paresseuse évite beaucoup de calculs inutiles tout en préservant les avantages théoriques de la réduction en ordre normal, qui est seule à même de rendre universel un langage purement fonctionnel. En effet, il n'est pas possible de programmer des entrées-sorties purement fonctionnelles sans adopter la réduction en ordre normal.

Dans un langage impératif ordinaire, les arguments d'une fonction sont évalués avant l'appel de la fonction (appel par valeur ou évaluation stricte). Il est cependant possible que cet argument ne soit pas utilisé dans le corps de la fonction, de sorte que ce travail a pu être inutile. Pour le moins, ce travail a été accompli trop tôt, c'est-à-dire avant qu'il ne soit véritablement requis. Cela suggère un meilleur schéma consistant à retarder l'évaluation de l'argument jusqu'à ce que sa valeur soit réellement nécessaire, et cette considération est justement le fondement de l'évaluation paresseuse. Cette stratégie

retarde l'évaluation de sous-expressions aussi longtemps que possible, avec l'espoir que la valeur de la sous-expression ne sera jamais – ou seulement partiellement – requise.

Considérons le passage d'arguments dans les appels de fonctions. Un paramètre de fonction ne sera pas évalué lorsque cette fonction est appliquée ou appelée. Si cette fonction ne fait que retourner cette valeur ou la placer dans une structure de donnée, son évaluation sera donc reportée. Nous pouvons dire qu'en principe une évaluation est uniquement requise lorsqu'une expression doit être affichée ou imprimée.

Par opposition, le passage de paramètre « par valeur » des langages impératifs est la forme la plus stricte possible car la fonction échoue si un seul des arguments effectifs n'est pas défini. Cependant, certains calculs peuvent s'exécuter même en présence d'un argument indéfini (ou d'un argument non évalué, ce qui est la même chose), tant que celui-ci n'est pas utilisé. La forme qui assure la plus grande chance de terminer l'exécution est donc la technique qui évalue les arguments non pas au moment de l'appel, mais le plus tard possible pendant l'exécution de la fonction, c'est-à-dire lorsque nous avons réellement besoin de la valeur de l'argument.

La technique d'évaluation paresseuse évalue les arguments au plus une fois et, si possible, pas du tout! La plupart des auteurs pensent désormais que l'évaluation paresseuse est cruciale en programmation fonctionnelle. Cette forme d'évaluation est également extrêmement utile dans la pratique du programmeur car elle le libère de soucis secondaires liés à l'ordre d'évaluation. Une conséquence de cette technique est qu'il y a incertitude sur l'instant de l'évaluation (pas de séquence garantie d'évaluation), mais ce n'est pas gênant dans un langage sans effets de bord.

Un autre argument en faveur de l'évaluation paresseuse est l'augmentation de la puissance d'expression du langage, en permettant en particulier la construction et la manipulation de structures de données infinies.

1.4.5.2 Structures de données infinies

L'évaluation non-stricte permet de traiter des structures de données infinies, car elles ne posent pas de problèmes tant que nous n'en évaluons que des parties finies. Les structures de données infinies sont loin de n'être qu'une simple curiosité. Comme les fonctions d'ordre supérieur, elles ont une grande influence sur le style de programmation et fournissent un large éventail de possibilités nouvelles non disponibles en programmation impérative. Leur puissance provient de leur utilisation pour séparer données et contrôle : un programmeur peut décrire des structures de données sans avoir à se soucier de la manière dont elles seront évaluées.

Ces listes infinies ne sont possibles que grâce à l'évaluation paresseuse qui garantit qu'elles ne seront évaluées qu'à la demande, au fur et à mesure de l'usage qui en sera véritablement fait. Or nous n'avons jamais besoin de la totalité d'une liste infinie, nous avons seulement besoin d'un élément à la fois.

Finalement, l'évaluation paresseuse est également essentielle pour la réalisation purement fonctionnelle des entrées-sorties.

1.4.5.3 Entrées-sorties

Les entrées-sorties représentent le point faible de beaucoup de langages pseudo-fonctionnels car elles ne sont pas effectuées de manière référentiellement transparente. Cependant, il est possible de faire toutes les entrées-sorties de manière purement fonctionnelle, pour autant que nous disposions d'un mécanisme d'évaluation en ordre normal tel que celui de l'évaluation paresseuse. Ainsi, la transparence référentielle est entièrement maintenue à l'intérieur du programme, ce qui n'exclut pas du tout que le monde extérieur ne soit pas fonctionnel.

1.4.5.3.1 Génération des sorties

La valeur retournée par l'expression évaluée représente les sorties d'un programme fonctionnel. Les valeurs intermédiaires – résultats des expressions et fonctions locales – ne font par contre pas partie des sorties générées. L'évaluation de ces valeurs de sortie ne produit aucun effet de bord. Par contre, l'interprétation de ces valeurs par l'environnement peut produire des effets de bord, par exemple leur impression sur un terminal ou leur écriture dans un fichier. En d'autres termes, le langage reste purement fonctionnel, mais l'environnement ne l'est généralement pas.

1.4.5.3.2 Lecture des entrées

Nous ne pouvons pas utiliser une fonction de lecture des entrées qui retournerait la prochaine entrée saisie. En effet, une telle fonction repose sur un effet de bord car elle modifie la position courante de lecture dans le fichier d'entrée. La solution fonctionnelle consiste à considérer les entrées comme une liste potentiellement infinie de valeurs, liste qui débute avec l'évaluation de la fonction principale et dure tant que celle-ci se poursuit. Ces entrées peuvent aussi bien provenir du clavier, de fichiers ou de diverses unités périphériques. Cette liste infinie est référentiellement transparente et peut être utilisée dans toute fonction, comme n'importe quelle valeur visible au moyen d'un nom. Nous pouvons donc librement la parcourir et la traiter, ce qui correspondra à la consommation des valeurs d'entrées. Le mécanisme d'évaluation paresseuse va conduire à attendre les entrées requises, c'est-à-dire celles qui sont nécessaires à la génération des sorties. Par contre, l'évaluateur n'attend pas les entrées non utilisées. C'est donc uniquement la génération des sorties – l'évaluation du résultat – qui détermine la consommation des entrées.

1.4.6 Conclusion

D'après [SEBE 02], dans les langages impératifs, il est relativement difficile de raisonner au sujet du sens (sémantique complexe). Par contre, il est habituellement relativement facile de raisonner à propos des ressources requises car le programmeur possède généralement un modèle mental réaliste, proche de la machine, du déroulement de l'exécution. Si la simplicité sémantique représente un avantage majeur des langages fonctionnels, qui les rend aptes à être le sujet de raisonnements concernant leur sens, une faiblesse est par contre constituée par la difficulté de raisonner à propos de leur comportement en espace, et dans une moindre mesure, en temps. Des changements à première vue innocents dans les programmes (et préservant leur sens) peuvent changer considérablement leur complexité espace-temps.

La programmation fonctionnelle est-elle l'antithèse de la programmation impérative? Nous pouvons davantage la voir comme une évolution logique vers des langages de plus haut niveau d'abstraction, en particulier en ce qui concerne l'usage des expressions. D'une façon plus générale, la programmation fonctionnelle contribue à l'heureux mouvement qui éloigne les langages de programmation de la machine sous-jacente qui permet de les exécuter. Si nous considérons qu'un facteur clé de l'évolution des langages est l'usage d'expressions de plus en plus complexes pour décrire un résultat, plutôt que l'emploi d'une suite d'instructions pour y parvenir, la programmation fonctionnelle est une évolution logique qui nous amène à une conclusion : tout est expression.

Dans [APPL 97], nous retrouvons que les langages fonctionnels purs ne permettent pas d'effet de bord, puisque les valeurs des paramètres ne sont jamais changées durant un appel de fonction. De ce fait, les paramètres ne sont jamais passés par référence, nom ou valeur-retour, seulement par valeur. Nous pouvons alors dire que sur un autre plan, la programmation fonctionnelle (sans affectation) peut être comparée à la programmation

structurée (sans instruction « goto »). À la discipline dans le flot de contrôle correspond la discipline dans le flot de données. Il ajoute que les langages fonctionnels forment une bonne base pour l'exécution en parallèle, puisqu'un programme n'est rien de plus qu'une fonction p(a1,a2,…,an), où chaque paramètre ai est aussi une fonction retournant une valeur à p. Chacun des ai peut être assigné à un processeur différent et évalué indépendamment des autres aj. Il dit aussi que le style fonctionnel produit des programmes plus courts qui sont plus faciles à vérifier et corriger qu'avec les langages procéduraux.

Les langages fonctionnels ne sont-ils que des jouets de théoriciens? Ce sont au minimum d'excellents langages de prototypage, grâce à leur très haut niveau obtenu sans renoncer pour autant au typage statique fort. Ils sont particulièrement adaptés à l'animation et à la transformation de spécifications formelles, mais se révèlent également d'excellents langages d'enseignement de par leur simplicité et leur rigueur. Les langages fonctionnels purs représentent un pont entre mathématique et informatique. Les mathématiques et ses méthodes de preuve sont étudiés depuis des siècles, les langages fonctionnels peuvent tirer un grand avantage de ce large corpus de recherche.

1.5 Conclusion

Nous avons vu les caractéristiques de la programmation orientée objet et pouvons constater les limites qui y sont rattachées, notamment au niveau du mécanisme d'héritage qui génère une complexité conceptuelle. La réutilisabilité, argument majeur du développement en orienté objet, a aussi été remise en cause. Cependant, par sa capacité à modéliser le monde réel en offrant un mécanisme de regroupement de tâches reliées sous une même entité, ce paradigme est puissant pour la conception de systèmes complexes du domaine industriel. Mais, là aussi, des limites ont été soulignées dès que des architectures reparties sont mises en cause. C'est d'ailleurs ce qui a poussé au

développement du paradigme agent. En effet, ce dernier répond très bien aux exigences des systèmes d'aujourd'hui qui utilisent beaucoup le découpage en sous-systèmes et les applications en réseau. Malheureusement, l'orienté agent est une discipline plutôt récente qui manque encore de standardisation tant au niveau de la définition même d'un agent, qu'au niveau des outils de développement disponibles. De plus, les agents sont surtout intéressants pour les systèmes distribués, ce qui ne nous concerne pas. Enfin, dans la dernière partie, nous avons vu que par opposition au paradigme traditionnel de la programmation impérative, la programmation fonctionnelle offre des avantages appréciables : style déclaratif, modèle sémantique simple et bien connu, parallélisme inhérent. Par sa nature fonctionnelle très proche des mathématiques, ce style de programmation offre une plus grande capacité d'abstraction par rapport à la résolution de problèmes.

Comptes tenus des besoins reliés à notre recherche, les éléments de la programmation fonctionnelle répondent au mieux à nos exigences. En effet, la capacité de tout voir comme des expressions et des fonctions, à une ou zéro variable, rejoint notre idée de chaînes de traitement. Nous pouvons aisément voir chaque module d'une chaîne de traitement comme une fonction, et l'enchaînement des modules comme un nouveau module dit complexe qui serait équivalent à une fonction complexe. De plus, « Il n'y a pas de variables dans le sens des langages impératifs, si bien qu'il ne peut pas y avoir d'effets de bord. » [SEBE 02], le fonctionnel va donc nous permettre de facilement enchaîner les modules, sans soucis de l'effet de bord.

Mais avant d'aller au cœur de notre projet, il nous faut voir les concepts du paradigme fonctionnel plus en détail. C'est ce qui fait l'objet du prochain chapitre.

CHAPITRE 2

CONCEPTS ET OUTILS FONCTIONNELS

2.1 Introduction

Les langages applicatifs, ou fonctionnels, sont basés sur le lambda-calcul de A. Church (1941) et sur la notion de structure opérateur-opérande. En effet, d'après [DESC 1990], « ils s'enracinent sur l'opération fondamentale d'application d'un opérateur sur un opérande afin de produire dynamiquement un résultat ». C'est pourquoi nous parlons de langages applicatifs. Ils sont aussi fondés sur la logique combinatoire de H.B. Curry (1929, 1958, 1972) et de F. Fitch (1972), qui étudie les processus opératoires (organisations d'opérateurs plus ou moins complexes) dans des systèmes applicatifs typés. Un processus opératoire est capable de construire des opérateurs, par des opérations explicites, à partir d'opérateurs déjà construits ou donnés initialement. Chaque processus pouvant lui-même être construit à partir de processus plus élémentaires. C'est cette notion qui nous paraît très intéressante pour notre recherche.

Ces langages offrent une grande capacité d'abstraction, ce qui explique que nous retrouvons des applications concrètes des formalismes logiques applicatifs aussi bien dans le domaine informatique avec LISP ou l'intelligence artificielle, que dans les sciences cognitives pour la représentation des connaissances, pour décrire les opérations du langage, ou pour les catégorisations cognitives.

L'objectif de ce chapitre est de mieux comprendre ce qui caractérise les langages fonctionnels, et d'introduire les combinateurs qui nous semblent être des outils intéressants. Dans un premier temps, nous allons donc voir les concepts du lambda-calcul en partant des notions de fonction. Ensuite, nous allons présenter la structure

opérateur-opérande propre aux systèmes applicatifs, puis la notion de type qui nous amènera vers les systèmes applicatifs typés. Enfin, nous aborderons les concepts de la logique combinatoire qui vont nous permettre d'en arriver aux combinateurs avant de conclure.

La théorie présentée ainsi que plusieurs exemples s'inspirent de [DESC 1990]. La partie sur les combinateurs reprend largement une présentation de mon directeur de maîtrise, Monsieur Ismaïl Biskri, faites dans le cadre d'un de mes cours à la maîtrise.

2.2 Le lambda-calcul

Le λ-calcul (lambda-calcul) est un langage inventé en 1931, par le logicien Alonzo Church, dans le but de résoudre des problèmes de logique. Au fil des années, le λ-calcul est devenu un langage de programmation universel, qui est à la base des langages fonctionnels. Il est aujourd'hui un outil central de l'informatique. En effet, nous le retrouvons au niveau des « procédures d'abstraction du langage LISP, défini par McCarthy en 1960 » [DESC 1990], ainsi qu'en intelligence artificielle.

De plus, en 1997, le logicien français Jean-Louis Krivine arrive à démontrer que le λ-calcul peut aussi définir tous les raisonnements et toutes les structures mathématiques. C'est donc aussi un langage mathématique universel.

Le λ-calcul permet de fournir un fondement aux mathématiques plus simple que la théorie des ensembles, et fondé sur la notion de fonction. Nous allons donc voir les notions de fonction dans un premier temps : la notion de fonction « classique » pour commencer, puis la fonction chez Frege qui permettra de présenter la notion de substitution et d'extensionalité, et enfin la notion de fonction dans la λ-notation qui introduira la notion de λ-expression. Dans un deuxième temps, nous verrons la β-

réduction qui permettra de donner un sens aux λ-expressions. Puis finalement, nous présenterons le λ-calcul, c'est-à-dire sa syntaxe, le concept de variable libre ou liée, le problème du télescopage des variables, la λ-β-réduction, et la notion de forme normale.

2.2.1 La notion de fonction « classique »

Les travaux de Dirichlet (1829) aboutissent à la fonction définie par son graphe. Nous parlons de la conception ensembliste puisqu'une application est considérée comme une correspondance fonctionnelle, entre deux ensembles A et B, caractérisée par un ensemble de couples (x,y) :

$$f : \quad A \rightarrow B$$
$$x \rightarrow y = f(x)$$

Ceci permet essentiellement de définir les fonctions numériques. Cependant, cette approche ensembliste n'est pas suffisante pour formaliser adéquatement certains processus fonctionnels qui sont modélisés en informatique notamment.

2.2.2 La fonction chez Frege

G.Frege (1893) a généralisé la conception ensembliste de Dirichlet en considérant des fonctions qui relèvent de la logique. Un connecteur propositionnel (et, ou, implique, équivaut à, etc.), ou opérateur logique, est bien une fonction d'un ensemble de couples de valeurs de vérité dans un ensemble {vrai, faux}. Mais, cette conception algébrique des opérations logiques (algèbre de Boole) ne suffit pas. Alors Frege oppose les *objets* aux *fonctions*. Les premiers représentent une seule chose ou valeur, et sont dits « saturés », les fonctions non.

Ceci introduit la notion de substitution. En effet, prenons par exemple la fonction '2 + x', le symbole 'x' représente de façon indéterminée des objets qui lui seraient

substituables. Nous parlons alors d'objet indéterminé ou de place d'argument (non saturé). La substitution remplie la place d'argument par un argument (objet), et après substitution nous obtenons une valeur.

Voici maintenant deux définitions avant de présenter la notion d'extension :

- Concept (ou prédicat logique) : Fonction à un argument dont les valeurs sont toujours des valeurs de vérité (vrai, faux).
- Relation (binaire, ternaire,...) : Fonction à deux (à trois, etc.) arguments dont les valeurs (après substitution de tous les arguments) sont toujours des valeurs de vérité.

Si un objet est argument d'un concept donnant pour résultat la valeur vrai, nous disons alors que « le concept s'applique à l'objet » ou que « l'objet tombe sous le concept ».

Pour un concept f, c'est-à-dire une fonction à valeur dans {vrai, faux}, l'extension de f, désignée par Ext(f), est la classe des objets x qui tombent sous le concept f :

$$Ext(f) = \{ x / f(x) = vrai \}$$

À une fonction quelconque, Frege associe un objet particulier qu'il nomme Wertverlauf, qui n'est autre que l'extension de la relation exprimée par 'y = f(x)' :

$$Ext(\text{'}y = f(x)\text{'}) = \{ (x,y) / (y = f(x)) = vrai \}$$

Le Wertverlauf n'est rien d'autre que le graphe de la fonction f.

Exemples

- f = 'x > 2' alors Ext(f) = {3, 4, ...}
- Ext('$x^2 = 9$') = {-3, 3}
- Ext('y = 2 + x') = {(0,2), (1,3), (2,4), ...}

Avec f et g qui désignent deux concepts, Frege défini l'égalité extensionnelle entre concepts (f = g) à partir de l'égalité entre les extensions (Ext(f) = Ext(g)), ou bien à partir de la généralité de l'égalité entre les valeurs des deux concepts ($\forall x$, f(x)=g(x)). Ceci donne : $f = g \Leftrightarrow Ext(f) = Ext(g) \Leftrightarrow \forall x, f(x) = g(x)$

Mais attention, les concepts de même classes extensionnelles peuvent être discriminables. En effet, prenons les concepts '$x^2 = 9$' et '$4x^2 = 36$', nous voyons qu'ils ne signifient pas la même chose, bien que leurs extensions soient identiques :

$Ext(\text{'}x^2 = 9\text{'}) = Ext(\text{'}4x^2 = 36\text{'}) = \{-3, 3\}$

C'est alors B. Russel qui ramène l'équivalence à une implication :

$\forall x, f(x) = g(x) \Rightarrow Ext(f) = Ext(g)$

Lorsque deux concepts ne seront pas plus discriminés que leurs extensions, nous dirons qu'ils vérifient le principe d'extensionalité [Ext] suivant:

[Ext] (f = g) $\Leftrightarrow$ Ext(f) = Ext(g)

Lorsque nous n'admettons pas le principe d'extensionalité [Ext], nous distinguons alors une égalité non extensionnelle entre concepts ($f \equiv g$) d'une égalité extensionnelle entre concepts (f = g), la première impliquant la seconde, d'où :

$(f \equiv g) \Rightarrow (f = g) \Leftrightarrow Ext(f) = Ext(g) \Leftrightarrow \forall x, f(x) = g(x)$

Il est maintenant possible d'avoir des concepts qui seraient discriminables (il est faux que $f \equiv g$) alors qu'extensionnellement ils sont égaux (f = g).

Exemple

Les concepts '$x^2 = 1$' et '$(x + 1)^2 = 2(x + 1)$' sont discriminables bien que leurs extensions soient égales (ils prennent la valeur vrai pour les seuls arguments –1 et +1, et le faux pour tous les autres arguments). Ainsi ils ne "signifient" pas la même chose :

"racine carrée de l'unité" est différent de "nombre incrémenté de l'unité tel que son carré soit égal à son double".

2.2.3 La fonction dans la lambda-notation

Remarquons tout d'abord que la notation usuelle f(x) est ambiguë puisqu'elle désigne, selon le contexte, tantôt la valeur de la fonction pour l'argument x, tantôt la fonction elle-même. De plus, 'x – y' peut être considérée comme définissant une fonction f de x, ou une fonction g de y.

La lambda-notation de A. Church (1941) enlève ces ambiguïtés. En effet, il introduit le symbole "λ", appelé abstracteur ou opérateur d'abstraction, qui à partir de l'expression "f(x)" construit la nouvelle expression fonctionnelle "(λx . f(x))", et nous avons ainsi :

(1) (λx . f(x)) qui désigne la fonction en soi ;

(2) f(x) qui désigne la valeur de la fonction pour l'argument x ;

(3) en particulier : (λx . x – y) qui désigne la fonction f de x ;

(4) en particulier : (λy . x – y) qui désigne une fonction de y .

Nous pouvons alors avoir des fonctions qui se ramènent à une variable seulement. Regardons l'exemple suivant pour s'en convaincre.

Exemple

Désignons par ⊕ une fonction d'un seul argument qui, lorsqu'elle est appliquée à b seul, produit une nouvelle fonction, notée (⊕ b); et cette dernière étant appliquée à a seul, produit la valeur (a + b). Nous avons donc :

$$((\oplus b) a) = (a + b)$$

En utilisant la lambda-notation de Church et les deux opérateurs d'abstraction λx et λy, nous posons :

$$\oplus \equiv (\lambda y . (\lambda x . (x + y)))$$

L'expression précédente est appelée une lambda-expression. Elle décrit parfaitement la nature applicative d'une fonction à un seul argument :

$$(\oplus b) = ((\lambda y . (\lambda x . (x + y))) b) = (\lambda x.(x+b))$$

et $$((\oplus b) a) = ((\lambda x . (x + b)) a) = (a + b)$$

D'une façon générale, une lambda-expression représente une fonction donnée par l'expression suivante : λ<variables>.<corps>. Les variables correspondent aux paramètres et le corps correspond à ce que fait la fonction avec ces paramètres.

Soit (λx . M), où M est le corps de la lambda-expression. Quand nous appliquons cette lambda-expression à un argument N, nous obtenons une nouvelle lambda-expression, en procédant à la substitution de N à chaque occurrence de la variable x dans le corps M. Désignons par " M [x : = N] " le résultat de cette substitution. Nous obtenons alors :

$$(\lambda x . M) N = M [x : = N]$$

Exemple

La fonction "carré" est exprimée par l'expression ($\lambda x . x^2$). En appliquant la fonction "carré" au nombre 5, nous obtenons :

$$(\lambda x . x^2) 5 = x^2 [x : = 5]$$

Commentaire

La lambda-expression ($\lambda x . x^2$) est indépendante du symbole x. Ainsi, nous avons :

$$(\lambda x . x^2) \equiv (\lambda y . y^2) \equiv (\lambda z . z^2)$$

Nous disons que le symbole λ "lie" entre elles les occurrences de la variable : toute substitution d'une valeur à la *variable liée* devra être effectuée pour toutes les occurrences de la variable dans le corps (c'est-à-dire sous la portée de l'opérateur d'abstraction).

2.2.4 La bêta-réduction

Il s'agit maintenant de donner un sens à ces λ-expressions. Soient les deux expressions suivantes :

(a) (λx . M) N

(b) M [x := N]

L'expression (a) décrit l'application de la lambda-expression λx . M à l'expression N, mais cette opération n'a pas été effectuée. L'expression (b) dénote le résultat de la substitution (de N aux occurrences de x dans M), après l'exécution du programme d'opérations décrit par (a). Les deux expressions (a) et (b) ne sont pas situées dans un seul et même temps. Nous relierons (a) et (b) par le relateur, noté ▶, qui est non symétrique puisque le résultat (b) vient après la déclaration du programme d'opérations (a). Ainsi :

(λx . M) N ▶ M [x := N]

Ceci est la règle importante de β-réduction : $(\lambda x . u)\ v$ ▶ $u\ [x := v]$

Elle exprime que si nous appliquons la lambda-expression $\lambda x . u$ à l'argument v, nous obtenons la même chose qu'en calculant directement u avec x remplacé par v.

Exemple

(λx . x^2) 5 ▶ 25

Nous emploierons le signe = lorsqu'il sera toujours possible de remonter du résultat (b) à l'opération (a) qui lui a donné naissance.

L'étude des relations engendrées par ▶ conduit à la λ-β-réduction de Curry ; l'étude des relations engendrées par = conduit à la λ-β-égalité de Curry. La prochaine partie va

permettre de donner le cadre formel pour étudier les relations engendrées par la λ-β-réduction et la λ-β-égalité.

2.2.5 Le lambda-calcul

Dans cette partie, nous allons commencer par aborder la syntaxe du lambda-calcul, puis préciser les notions de variable libre et de variable liée. Ensuite, nous présenterons avec plus de précisions les règles de substitution, qui vont nous amener au problème du télescopage des variables, et les règles de réduction. Enfin, nous introduirons la notion de forme normale.

2.2.5.1 Syntaxe

Le lambda-calcul, dans son aspect syntaxique, a pour tâche de préciser les règles formelles qui permettent de :

- Construire toutes les lambda-expressions correctes ;
- Appliquer une lambda-expression à une autre en procédant aux substitutions ;
- Définir d'éventuelles classes d'expressions réductibles les unes aux autres.

La définition d'un lambda-calcul consiste à se donner des *atomes* et des *règles* pour construire toutes les lambda-expressions licites à partir des atomes. Les atomes représentent des fonctions à une place d'argument. Les atomes se composent d'un ensemble (fini ou infini) de constantes et un ensemble infini de variables.

L'ensemble des lambda-expressions est défini récursivement par les règles suivantes :

- Chaque atome est une lambda-expression ;
- Si X et Y sont des lambda-expressions, alors (XY) est une lambda-expression ;

- Si Y est une lambda-expression et x une variable, alors (λx . Y) est une lambda-expression.

Par la suite, nous adopterons les conventions d'écriture suivantes :

- L'identité entre lambda-expressions est désignée par '≡'.
- Les variables seront désignées par des lettres minuscules.
- Les lambda-expressions seront désignées par des lettres majuscules.

2.2.5.2 Variable libre, variable liée

X présente une occurrence dans Y (ou Y contient X) est défini par récurrence comme suit :

- X présente une occurrence dans X;
- Si X présente une occurrence dans Y ou dans Z, alors X présente une occurrence dans (YZ);
- Si X présente une occurrence dans Y, alors, pour chaque z, X présente une occurrence dans (λz.Y).

Exemples

- (xy) présente 2 occurrences dans (xy)(λx.xy);
- x présente une occurrence dans (λx.xy) mais pas dans (λx.y);
- dans l'expression (λx.xy) (xy), x présente 2 occurrences mais ces occurrences sont différentes, la première sera liée, la seconde libre.

Définition

Une occurrence d'une variable x dans Y est dite liée si et seulement si Y est de la forme (λx.X), autrement, elle est dite libre.

Une occurrence d'une variable x dans Y est dite libre (respectivement liée) dans Y lorsqu'elle a au moins une occurrence libre (respectivement liée) dans Y.

Exemple

La variable x est libre dans (xy), mais liée dans (λx.xy). Une même variable peut être à la fois libre et liée dans une même expression : x est libre et liée dans ((λx.xy)(xy)).

2.2.5.3 Règles de substitution

L'expression (XY) représente l'application de la fonction X à l'argument Y. Supposons que Y soit une expression où x présente des occurrences libres, (λx.Y) représente alors une fonction construite par abstraction : pour un argument donné a, la valeur de cette fonction est obtenue en substituant a aux occurrences de x dans Y.

Les difficultés d'interprétation du lambda-calcul viennent avec des expressions comme (XX) qui représente le résultat d'une fonction X appliquée à elle-même, ou avec des expressions comme (λx.Y), où x ne présente aucune occurrence libre dans Y. C'est pourquoi, certaines restrictions peuvent être apportées au lambda-calcul. Par exemple, imposer des restrictions sur les types, pour ne pas autoriser l'auto-application d'une fonction, ou encore interdire des lambda-expressions de la forme (λx.Y), où x n'a pas d'occurrence dans Y.

Ceci nous amène au problème du télescopage des variables lors d'une substitution. La clause 5 de la définition suivante évite de tels télescopages.

Définition

Le résultat de la substitution de N à chaque occurrence libre de x dans M, M [x := N], est définie par :

1) $x [x := N] \equiv N$
2) $a [x := N] \equiv a$, pour chaque atome $a \neq x$
3) $M1M2 [x := N] \equiv (M1 [x := N]) (M2 [x := N])$
4) $(\lambda x . Y) [x := N] \equiv \lambda x . Y$
5) $(\lambda y . Y) [x := N] \equiv$
   - $\lambda y . (Y [x := N])$ si $y \neq x$ et y n'a pas d'occurrence libre dans N, ou x n'a pas d'occurrence libre dans Y.
   - $\lambda z . ((Y [z := y]) [x := N])$ si $y \neq x$ et y a une occurrence libre dans N, et x a une occurrence libre dans Y.

Exemple

Sans la clause 5 nous avons :

$(\lambda y . x) [x := u] \equiv$
- $\lambda y . u$ pour $y \neq u$ (fonction constante)
- $\lambda u . u$ pour $y = u$ (fonction identité)

Ainsi, le résultat de la substitution dépend des relations entre les variables, il n'est pas déterministe. Avec la clause 5, l'interprétation de la substitution est invariante, nous avons juste la fonction constante :

$(\lambda y . x) [x := u] \equiv$
- $\lambda y . u$ pour $y \neq u$ (fonction constante)
- $\lambda z . u$ pour $y = u$ (fonction constante)

En effet, pour $y = u$, avec la clause 5 nous avons :

$(\lambda y . x) [x := u] \equiv \lambda z . ((x [z := y]) [x := u]) \equiv \lambda z . (u [z := y]) \equiv \lambda z . u$

Remarque

Il ne faut pas confondre les expressions :

$(\lambda x . Y) N$ et $(\lambda x . Y) [x := N]$ (clause 4).

Nous avons : $(\lambda x . Y) N \blacktriangleright Y [x := N]$

Alors que : $(\lambda x . Y) [x := N] \equiv \lambda x . Y$

En effet, dans la deuxième expression, nous ne pouvons pas substituer N à x dans (λx . Y) puisque x est liée!

2.2.5.4 Règles de réduction

2.2.5.4.1 λ-β-réduction

La λ-β-réduction est la relation de préordre (réflexive et transitive) engendrée par la λ-β-réduction directe (▶) et les changements de variables (≡).
La notation est la suivante :

λ-β-réduction directe : $(\lambda x . M) N \blacktriangleright M[x := N]$

λ-β-réduction : $X \rightarrow Y$

Exemple

(λx (λy. xy) b) a	▶	(λy. xy) b [x := a]
	≡	(λy. ay) b
	▶	(ay) [y := b]
	≡	ab
(λx (λy. xy) b) a	→	ab

Dans la λ-β-réduction directe, le membre de gauche représente l'opération à effectuer (ou le redex), tandis que le membre de droite représente l'opération effectuée (ou le contractum). La λ-β-réduction est donc un processus opératoire généralement non symétrique. Lorsque qu'il y a symétrie, nous parlons de λ-β-égalité, et écrivons : $(\lambda x . M) N = M[x := N]$.

2.2.5.4.2 Forme normale

Une lambda-expression est une forme normale si et seulement si elle ne peut être réduite par une λ-β-réduction. L'expression irréductible est appelée forme normale : elle représente la valeur prise par la fonction. Une lambda-expression est donc dans une forme normale si elle ne contient aucun redex. Certaines expressions n'ont pas de forme normale.

Exemple

La forme normale de (λx (λy. xy) b) a est ab.

2.2.5.4.3 Stratégies de réduction

Il existe deux stratégies de réduction : l'ordre normal et l'ordre applicatif.

Ordre normal (on part de l'extérieur de l'expression) :

(λx (λy. xy) b) a → (λy. ay) b → ab

Ordre applicatif (on part de l'intérieur de l'expression) :

(λx (λy. xy) b) a → (λx. xb) a → ab

2.3 Structure opérateur-opérande

Dans cette partie, nous allons tout d'abord définir les systèmes applicatifs. Nous présenterons ensuite les deux modes d'évaluation des expressions applicatives. Enfin, nous introduirons la notion de type qui caractérise les systèmes applicatifs typés.

2.3.1 Système applicatif

Les systèmes applicatifs sont organisés autour de la notion d'opérateur et d'opérande :

Un opérateur f s'applique à un opérande a.

Nous parlons alors d'une opération d'application ou d'une application, et la noterons, d'après la notation de J. von Neumann (1925) : [f , a]

Quand l'opération d'application est exécutée, un résultat b apparaît. Le relateur ▶ est introduit pour donner l'expression applicative suivante : [f , a] ▶ b

Exemple

Lambda-calcul	$(\lambda x . x^2)\ 3$	▶	$(3)^2$
Expression applicative	$[(\lambda x . x^2), 3]$	▶	$(3)^2$

Le résultat b, obtenu après l'application de l'opérateur f à l'opérande a, est soit un opérateur, soit un opérande. Ainsi, l'opération d'application est réitérable.

Exemple

$[[f, a_2], a_1] \blacktriangleright [b_2, a_1] \blacktriangleright b1$

Avec dans ce cas b_2 opérateur et b_1 opérande.

L'opérande d'un opérateur peut-être simple ou structuré. Dans ce dernier cas, nous parlons d'opérateur à deux, ... , ou n places, dont l'opérande sera un couple d'objets, ... , ou un n-uplet d'objets.

Exemple

$[+, <13, 34>] \blacktriangleright 47$

Avec + opérateur à deux places, et $<13, 34>$ un couple de nombres.

Définition

Un système applicatif est la donnée :

- d'un ensemble (éventuellement vide) d'objets atomiques (constantes ou variables) et d'opérateurs atomiques (constants ou variables),

- de l'opération d'application, considérée comme une opération primitive, qui permet de construire à partir des atomes (objets et opérateurs) des expressions dérivées (objets ou opérateurs dérivés).

2.3.2 Mécanismes d'évaluation

L'évaluation d'une expression applicative peut-être vue comme une application. Notons *eval* l'opérateur d'évaluation. Nous avons :

[eval , [f , a]] ▶ [eval , b] ▶ eval (b)

Exemples

- [eval , [($\lambda x . x^2$) , 3]] ▶ [eval , $(3)^2$] ▶ 9
- [eval , [+ , < 13 , 34 >]]
  [[eval , +] , [eval , < 13 , 34 >]]
  [eval (+) , < eval (13) , eval (34) >]
  [(+) , < 13 , 34 >]
  47
  avec (+) la procédure d'addition qui résulte de l'évaluation de l'opérateur +.

Il existe deux modes différents d'évaluation : l'évaluation applicative et l'évaluation normale. Dans l'évaluation applicative, il faut évaluer tous les éléments de l'opérande avant d'appliquer l'évaluation de l'opérateur. Un interprète LISP utilise généralement ce mode d'évaluation. Dans l'évaluation normale, il faut évaluer l'opérateur le plus à gauche, puis appliquer l'opérateur évalué au premier opérande, puis recommencer. Ce mode d'évaluation est utilisé dans la logique combinatoire et dans le lambda-calcul.

Exemple

Nous avons deux interprétations possibles de l'addition.

Soit l'expression applicative [+ , (1, 2, 3)], où + a pour opérande la liste (1, 2, 3).

Soit l'expression applicative [[[+ , 1] , 2] , 3], où + a pour opérande 1, et où le résultat est un opérateur qui a pour opérande 2, ...

- Évaluation applicative :

[eval , [+ , (1, 2, 3)]]
[eval (+) , eval (1, 2, 3)]
[eval (+) , < eval (1), eval (2, 3) >]
[eval (+) , < eval (1), < eval (2), eval (3) > >]
[eval (+) , < eval (1), < 2 , 3 > >]
[eval (+) , < 1 , < 2 , 3 > >]
[eval (+1) , < 2 , 3 >]
[eval (+3) , 3]
[eval (+6)]
eval (6)
6

- Évaluation normale :

[eval , [[[+ , 1] , 2] , 3]]
[[[eval (+) , eval (1)] , 2] , 3]
[eval , [[(+1) , 2] , 3]]
[[eval (+1) , eval (2)] , 3]
[eval , [(+3) , 3]]
[eval (+3) , eval (3)]
[eval , 6]
eval (6)
6

Théorème de Church-Rosser

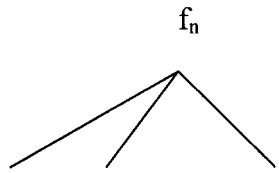
Le mode d'évaluation applicative et le mode d'évaluation normale conduisent toujours au même résultat.

Représentations arborescentes

Dans un système applicatif, nous distinguons les deux expressions applicatives suivantes : 1) $[f_n , < a_n , \dots , a_1 >]$

2) $[\dots [f_n , a_n] , \dots , a_1]$

- La première expression est représentable par la structure arborescente suivante :



- L'évaluation de l'expression applicative (1) est représentable par l'arbre d'évaluation :

f_n $< a_n , \dots , a_1 >$

---

b

où b représente le résultat de l'évaluation de l'application de f_n à son opérande structuré.

- L'évaluation de l'expression applicative (2) est représentable par l'arbre d'évaluation :

f_n a_n

---

b_{n-1} a_{n-1}

... b_1 a_1

---

b

2.3.3 La notion de type

Nous désirons souvent manipuler différents types d'expressions. L'évaluation d'un opérateur arithmétique sera différente selon que ses opérandes sont des nombres entiers, des nombres réels, des vecteurs, etc. Certains opérateurs ne s'appliquent qu'à des expressions logiques donnant pour résultat des expressions logiques (ce sont les opérateurs booléens). D'autres opérateurs s'appliquent à des nombres donnant pour résultat une valeur logique (ce sont les opérateurs relationnels).

La notion de type est introduite pour considérer différents types d'expressions, c'est-à-dire des classes hétérogènes d'expressions. Donnons-nous plusieurs sortes (d'expressions) de base, par exemple, dans certains langages de programmation, les sortes des entiers naturels, des réels, des booléens, des chaînes de caractères, etc. À partir de ces sortes d'expression, considérées comme des types élémentaires, nous construisons, au moyen d'opérations explicites, tous les types (dérivés).

Un système applicatif dont les expressions se voient assigner des types sera dit système applicatif typé.

2.4 Logique combinatoire

La partie précédente nous permet d'en arriver à la logique combinatoire. Nous présenterons tout d'abord le principe applicatif, puis les expressions combinatoires, et enfin les combinateurs.

2.4.1 Principe applicatif

En logique classique, un prédicat à n places est appliqué à ses arguments en une seule étape. Le système fondationnel de la logique introduit par M. Schönfinkel (1924) et les systèmes fondationnels de la logique combinatoire de H.B. Curry (1958) font opérer les prédicats sur leurs opérandes en plusieurs étapes.

Les systèmes que nous allons considérer maintenant comprendront uniquement des opérateurs unaires (de différents types), grâce au principe applicatif de Schönfinkel. Ce dernier fait appel à une opération, appelée opération de curriage ou de curryfication qui associera un opérateur unaire Curry(fn) à chaque opérateur fn (à n places).

Rappelons qu'un opérateur n-aire est un opérateur qui nécessite n arguments pour construire un résultat dont l'évaluation est possible. Il faut donc considérer n applications à n opérandes successives pour construire le résultat. Dans un système applicatif reposant sur le principe applicatif, les opérandes ne sont pas structurés sous forme d'uple d'objets, chaque opérateur s'applique à un seul opérande à la fois.

Énonçons le principe applicatif :
Un opérateur n-aire fn est représenté par un opérateur unaire Curry(fn) équivalent, qui donne pour résultat, lorsque ce dernier est appliqué à un opérande, la représentation d'un opérateur (n-1)-aire.

Pour une fonction binaire f2, l'opération de curriage 'Curry' est définie à l'aide de la lambda-notation définie par :

$$\text{Curry}(f2) \equiv \lambda y.(\lambda x.f2(x,y))$$

D'une façon générale, soit f2 un opérateur à 2 places, l'opérateur Curry(f2) est un opérateur unaire défini par :

$$Curry(f2) \equiv \lambda y . (\lambda x . f2(x,y))$$

Appliqué à un opérande constant a, Curry(f2) lui associe un résultat défini par :

$$(Curry(f2))\ a \blacktriangleright \lambda x . f2(x,a).$$

Ce résultat est à son tour un opérateur à une place.

Plus généralement, la transformée par curriage, d'un opérateur fn à n places, sera définie par :

$$Curry(fn) = \lambda xn . (\ldots (\lambda x1 . fn(x1,\ldots,xn)) \ldots)$$

Où Curry(fn) peut être considéré comme un opérateur unaire donnant pour résultat, après application à l'opérande constant a, un opérateur (n-1)-aire défini par :

$$(Curry(fn))\ a \blacktriangleright \lambda xn\text{-}1 . \ \ldots \ . \ \lambda x1 . fn(x1, \ldots , xn\text{-}1, a)$$

En logique, tout prédicat n-aire Pn sera représenté canoniquement, au moyen de l'opération Curry(.) par un prédicat unaire P'1 défini par : $P'1 \equiv Curry(Pn)$

Exemple

Considérons le prédicat binaire « ... est plus grand que ... », le prédicat équivalent construit par curriage est : $P'1 \equiv \lambda y . (\lambda x . (x \text{ est plus grand que } y))$.

P'1 est un opérateur unaire qui, appliqué à 3, donne pour résultat le prédicat unaire P''1 suivant : $P''1 \equiv \lambda x . (x \text{ est plus grand que } 3)$.

Appliqué à 5, P''1 donne pour résultat la proposition '5 est plus grand que 3' qui dénote le vrai en arithmétique.

2.4.2 Expressions combinatoires

Nous allons désormais considérer des systèmes applicatifs organisés à partir du principe applicatif. Les expressions combinatoires sont des expressions applicatives représentées par des agencements linéaires d'opérateurs et d'opérandes, c'est-à-dire par des suites de

symboles désignant des opérateurs et des opérandes organisés par des parenthèses métalinguistiques.
Nous considérons des atomes (opérateurs et opérandes absolus) constants ou variables.
Nous engendrons toutes les expressions combinatoires au moyen des règles suivantes :

1) les atomes sont des expressions combinatoires;
2) si X et Y sont des expressions combinatoires, alors (XY) est une expression combinatoire (ce qui signifie que X est un opérateur qui s'applique à l'opérande Y).

Nous adoptons, pour présenter les expressions combinatoires, les conventions applicatives suivantes :

- l'application est représentée par la juxtaposition de l'opérateur préfixé X à l'opérande Y, placée entre des parenthèses, soit : (XY);
- nous admettons l'associativité à gauche, ce qui permet de supprimer les parenthèses métalinguistiques superflues;
- si (XY) = (UV), alors : X ≡ U et Y ≡ V.

Ainsi, l'expression combinatoire (XY) a la même signification que l'expression applicative :
[X,Y] où X désigne l'opérateur appliqué à l'opérande Y.

L'expression 'X Y1 Y2 ... Yn' est une expression combinatoire qui représente l'expression applicative suivante : [... [[X,Y1],Y2], ...], Yn].
X est un opérateur qui est appliqué à Y1, d'où l'opérateur (XY1), qui est appliqué à Y2, d'où l'opérateur ((XY1)Y2), et ainsi de suite.

En adoptant la convention 2, nous pouvons simplifier l'expression combinatoire en supprimant toutes les parenthèses externes (associativité à gauche).

Ainsi, les 2 expressions suivantes XY1Y2Y3 et X(Y1Y2)Y3 sont distinctes. En restaurant les parenthèses, la première devient : (((XY1)Y2)Y3) et la seconde devient : ((X(Y1Y2))Y3). Dans la première, X est appliqué à Y1, le résultat à Y2 et ainsi de suite. Dans la seconde, X est appliqué à l'opérande (Y1Y2), le résultat est appliqué à Y3.

2.4.2.1 Expressions combinatoires de différents types

Lorsque nous considérons des expressions combinatoires de différents types, nous assignons des types engendrés à partir de l'ensemble S des sortes au moyen des seules règles suivantes :

3) les sortes de S sont des types;
4) si x et y sont des types, alors Oxy est un type.

L'introduction des types permet de limiter la 'libre combinatoire' entre opérateurs et opérandes. Un opérateur X de type Oxy ne peut s'appliquer à un opérande Y de type z à condition que z soit identique à x. Nous en déduisons la règle applicative suivante :

5) si Oxy : X et x : Y, alors y : XY.

Nous en déduisons aussi les règles suivantes :
5') si Oxy : X et y : XY, alors x : Y;
5'') si x : Y et y : XY, alors Oxy : X.

Les règles applicatives 5, 5', 5'' sont des schémas de règle représentés par :

$$5)\quad \frac{Oxy : X \qquad x : Y}{y : XY}$$

5') Oxy : Y y : XY

---

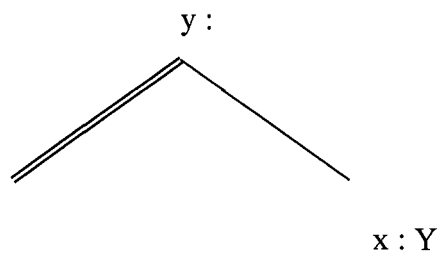
y : Y

5'') x :Y y: XY

Oxy : X

2.4.2.2 Arbres applicatifs

Une expression combinatoire a une 'histoire constitutive' qui est représentée par un arbre, dit 'arbre applicatif'. La contribution de l'opérateur est représentée par une branche doublée, la contribution de l'opérande est représentée par une branche simple. L'application de l'opérateur X de type Oxy, appliqué à l'opérande Y de type x est représentée par l'arbre applicatif suivant :



2.4.2.3 Évaluation des expressions combinatoires

Lorsque X est appliquée à son opérande Y, donnant pour résultat Z, nous avons la relation (d'évaluation) suivante :

(XY) > Z

En utilisant l'opérateur d'évaluation externe eval, nous avons :

(XY) > Z si et seulement si eval(XY) = Z

Les expressions (XY) et Z doivent être de type identique, d'où la règle suivante (règle d'isotypicalité) :

6) si Oxy : X et x : Y alors y : Z.

2.4.3 Combinateurs et réductions

La logique combinatoire a eu comme précurseur Frege (1879). La notion de combinateur est introduite pour la première fois par Schönfinkel (1924), elle est reprise par Curry et Feys en 1958. Le point de départ des recherches de H.B. Curry fut double : l'analyse de l'émergence du paradoxe de B. Russell, et la formalisation complète de la notion de substitution. Cependant, la variable est très souvent plurivoque (qui a plusieurs valeurs) et est donc difficile à gérer automatiquement lors d'un processus de substitution. Alors Curry introduit des opérateurs généraux abstraits, appelés combinateurs, qui sont comme des programmes de construction pour donner des opérateurs complexes, la construction restant indépendante de toute interprétation des unités composées dans un domaine externe. En fait, l'objectif est de simplifier et régulariser la substitution en se passant du concept de variable liée afin de simplifier l'automatisation et le contrôle des opérations de substitution. On parle de logique « sans variables » (expression de J. Rosser). Ceci peut ouvrir une nouvelle conception de la programmation puisque la variable informatique dépend d'un état de l'environnement qui, lui, évolue dynamiquement au cours de l'exécution d'un programme par une machine.

Les combinateurs sont des opérateurs abstraits qui nous permettent de construire à partir d'opérateurs (d'un système applicatif), des opérateurs de plus en plus complexes. Les capacités abstractives des combinateurs en font des outils puissants qui trouvent des applications jusque dans la linguistique lorsqu'en 1965, pour répondre au modèle de la

grammaire générative de N. Chomski, S.K. Shaumyan propose un modèle linguistique du langage et des organisations dans les langues, appelé « grammaire applicative universelle », qui se base explicitement sur la logique combinatoire de H.B. Curry.

Mais voyons comment fonctionnent les combinateurs : L'action d'un combinateur sur un argument est définie par une règle spécifique appelée β-réduction (au sens de Curry). Cette règle établit une relation entre une expression avec un combinateur et une expression équivalente sans combinateur. Nous allons alors commencer cette partie en présentant les combinateurs élémentaires et leurs actions, puis nous introduirons les notions de combinateurs complexes, de puissance d'un combinateur, et de combinateurs à distance. Nous suivrons avec les théorèmes de ces opérateurs abstraits puis avec les types qui leurs sont attribués. Enfin, nous revisiterons la notion de forme normale en la positionnant par rapport aux combinateurs, ce qui nous amènera au théorème de Church-Rosser et ses corollaires.

2.4.3.1 Les combinateurs élémentaires

2.4.3.1.1 Le combinateur "I" d'identité

Nous associons à un opérateur f quelconque, l'opérateur complexe I f. L'action du combinateur I est caractérisée par la règle de réduction suivante :

$$I\ f \rightarrow f$$

La λ-expression qui représente I est la suivante : $\lambda x\ .x$

2.4.3.1.2 Le combinateur "B" de composition

Soient deux opérateurs complexes f et g. Le combinateur B leur associe un opérateur complexe B f g tel que pour un argument x on ait la règle de réduction suivante:

$$B\ f\ g\ x \rightarrow f\ (g\ x)$$

La λ-expression qui représente B est la suivante : $\lambda\ x\ y\ z\ .x(yz)$

2.4.3.1.3 Le combinateur "S" de substitution

Combinateur de distribution (Curry), ou combinateur de substitution (Szabolcsi, Steedman).
Soient f et g deux opérateurs, f est binaire et g est unaire. Le combinateur S a pour effet d'associer à ces deux opérateurs l'opérateur complexe S f g. L'action de cet opérateur à un opérande x est spécifiée par la règle de réduction suivante :

$$S\ f\ g\ x \rightarrow f\ x\ (g\ x)$$

La λ-expression qui représente S est la suivante : $\lambda\ x\ y\ z\ .xz\ (yz)$

2.4.3.1.4 Le combinateur "Φ" de coordination

Combinateur de distribution (Curry), ou combinateur de coordination (Biskri, Desclés).
Soient f, g, h des opérateurs. Le combinateur Φ a pour effet d'associer à ces trois opérateurs l'opérateur complexe Φ f g h. L'action de cet opérateur à un opérande x est spécifiée par la règle de réduction suivante :

$$\Phi\ f\ g\ h\ x \rightarrow f\ (g\ x)\ (h\ x)$$

La λ-expression qui représente Φ est la suivante : $\lambda\ x\ y\ z\ u\ .\ x\ (y\ u)\ (z\ u)$

2.4.3.1.5 Le combinateur "Ψ" de distribution

L'action du combinateur Ψ de distribution (Curry) est définie par la règle de réduction suivante :

$$\Psi\ f\ g\ x\ y \rightarrow f\ (g\ x)\ (g\ y)$$

La λ-expression qui représente Ψ est la suivante : $\lambda\ x\ y\ u\ v\ .\ x\ (y\ u)\ (y\ v)$

2.4.3.1.6 Le combinateur "C*" de changement de type

L'action du combinateur C_* est définie par la règle de réduction suivante :

$$C_*\ X\ Y \rightarrow Y\ X$$

Le combinateur C_* a pour but de changer le statut de l'opérateur et d'en faire un opérande. Ainsi si X est opérateur ayant pour opérande Y alors X devient opérande de l'opérateur (C_* Y).

La λ-expression qui représente C_* est la suivante : $\lambda\ x\ y\ .\ y\ x$

2.4.3.1.7 Le Combinateur "C" de permutation

Ce combinateur associe à un opérateur f un opérateur complexe C f tel que si f agit sur les opérandes x et y pris dans cet ordre, C f agira sur les opérandes y et x pris dans cet autre ordre. L'action du combinateur C est définie par la réduction suivante :

$$C\ f\ y\ x \rightarrow f\ x\ y$$

La λ-expression qui représente C est la suivante : $\lambda\ x\ y\ z\ .\ x\ z\ y$

2.4.3.1.8 Le combinateur "W" de duplication

Soit f un opérateur quelconque et soit x son opérande. Le combinateur W leur associe un opérateur complexe W f tel que :

W f x ➔ f x x

En d'autres termes le combinateur W construit un nouvel opérateur W f, qui s'il a pour opérande x alors son action est réductible à l'action de f sur l'opérande x dupliqué : f agit sur x et le résultat est de nouveau appliqué à x.

La λ-expression qui représente W est la suivante : λ x y . x y y

2.4.3.1.9 Le combinateur "K" d'effacement

Ce combinateur nous permet d'effacer un opérande que nous pouvons appeler "opérande fictif". Ainsi l'action de K se traduit dans la réduction suivante :

K f x ➔ f

La λ-expression qui représente K est la suivante : λ x y . x

2.4.3.2 Les combinateurs complexes

Outre les combinateurs élémentaires que nous venons de voir, il existe des combinateurs complexes construits à partir des combinateurs élémentaires. Nous avons par exemple : B C C , W B , B C*.

L'action de ces combinateurs est déterminée par l'application enchaînée des combinateurs élémentaires à partir du combinateur élémentaire le plus à gauche.

Exemple

Prenons l'expression B C C x y z, on peut considérer que la réduction du combinateur B C C est exprimée à travers la réduction de B puis de C puis de C :

1 B C C x y z

2 C (C x) y z

3 (C x) z y

4 x y z

2.4.3.3 Puissance d'un combinateur

Il existe un cas particulier de combinateurs complexes :

B^2, B^3, ..., B^n, C^2, C^3, ..., C^n, W^2, W^3, ..., W^n, ...

Nous résumons ce genre de combinateurs dans la définition suivante :

Si χ est un combinateur alors χ^n itère n fois l'action du combinateur χ tel que :

$\chi^1 = \chi$ et $\chi^n = B\ \chi\ \chi^{n-1}$

Exemples

B^2 a b c d ➔ a (b c d)

B^3 a b c d e ➔ a (b c d e)

C^3 a b c d e ➔ B C C^2 a b c d e

2.4.3.4 Combinateurs à distance

Un autre cas particulier de combinateurs complexes se distingue : les combinateurs dont l'action se fait à distance.

Ces combinateurs sont définis par :

Si χ est un combinateur alors χ_n diffère son action de n pas, avec $\chi_n = B^n \chi$.

Exemples

C_2 a b c d e ➔ B^2 C a b c d e ➔ a b c e d

B_3 a b c d e ➔ B^3 B a b c d e ➔ a b c (d e)

2.4.3.5 Théorèmes sur les combinateurs

Tous les combinateurs peuvent être exprimés en fonction des combinateurs S et K (Curry, 1958).

Ainsi nous avons les exemples suivants :

I = S K K

B = S (KS) K

W = S S (K (S K K))

C = S (B B S) (K K)

Φ = B (B S) B

Ψ = Φ (Φ (Φ B)) B (K K)

2.4.3.6 Les types des combinateurs

Le fait que nous fassions évoluer les combinateurs dans un cadre applicatif typé nous engage à attribuer des schémas de types applicatifs aux combinateurs. Ces schémas de types sont fonctions de types variables que nous déterminons par l'opération d'unification. Voici les types reliés à chaque combinateur :

- Le schéma du type applicatif du combinateur I est : Faa.
- Le schéma du type applicatif du combinateur B est : FFacFFbaFbc
- Le schéma du type applicatif du combinateur S est : F(Fa(Fbc))(F(Fab)(Fac)).
- Le schéma du type applicatif du combinateur C_* est : FaFFabb.
- Le schéma du type applicatif du combinateur C est : F(Fa(Fbc))(Fb(Fac)).
- Le schéma du type applicatif du combinateur W est : F(Fa(Fab))(Fab).
- Le schéma du type applicatif du combinateur K est : FaFba.

Remarque

Les types a, b, c sont des types variables, ils sont identifiés par unification.

Exemple

Prenons le cas du combinateur B. Le type de B est FFacFFbaFbc. Dans un contexte bien défini, selon les types des opérandes de B il sera possible de calculer le type applicatif de B à partir de son schéma.

Soit l'expression applicative typée B X Y Z. X est de type Fyx. Y est de type Fzy. Z est de type z. La règle applicative présentée plus haut nous permet de calculer son type par la construction suivante :

FFacFFbaFbc : B Fyx : X Fzy : Y z : Z

FFbyFbx : B X (application de la règle applicative avec unification : a = y, c = x)

Fzx : B X Y (application de la règle applicative avec unification : b = z)

---

x : B X Y Z

2.4.3.7 Formes normales

Prenons les trois expressions suivantes :

1. B C a b c d

2. a (W b c) d

3. a (b c)

Les expressions 1 et 2 peuvent être réduites car elles contiennent des combinateurs. L'expression 3 ne peut pas être réduite car elle ne contient pas de combinateurs.

Définition 1

Une expression est dite sous forme normale lorsqu'elle ne peut plus être réduite.

L'expression 3 est sous forme normale.

Les expressions 1 et 2 ne sont pas sous forme normale.

Définition 2

Soit E' une expression sous forme normale. Soit E une expression qui n'est pas sous forme normale. Si E est réduite à E' alors nous dirons que E' est la forme normale de E.

Pour les expressions 1 et 2 les formes normales sont respectivement :

a b d c et a (b c c) d

Remarque

Certaines expressions combinatoires n'ont pas de formes normales.

Par exemple, l'expression W W W :

W W W ==> W W W ==> W W W ==> W W W ==> ...

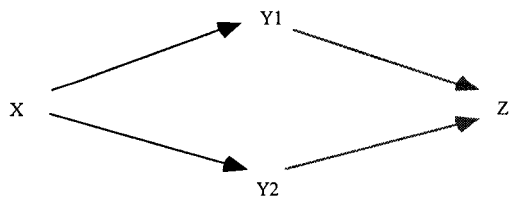
2.4.3.8 Théorème de Church-Rosser

La question que nous sommes en droit de nous poser est la suivante : Une expression combinatoire peut-elle avoir plusieurs formes normales ?

La réponse à cette question est donnée par le théorème de Church-Rosser :

> Si une expression combinatoire X se réduit en une expression combinatoire Y_1 et si X se réduit en une autre expression combinatoire Y_2 alors il existe une expression combinatoire Z tel que Y_1 et Y_2 se réduisent en Z.

Ce théorème révèle le fait que la logique combinatoire possède la propriété de Church-Rosser pour les relations transitives. Nous résumons cette propriété par le diagramme suivant :



L'intérêt de ce théorème est dans ses corollaires :

Corollaire 1

Une expression combinatoire a au plus une seule forme normale.

Corollaire 2

Si X=Y alors il existe un Z tel que X —> Z et Y—> Z.

Corollaire 3

Si X = Y et Y est une forme normale, alors X —> Y.

Corollaire 4

Si X = Y alors soit X et Y n'ont pas de forme normale, soit X et Y ont la même forme normale.

Corollaire 5

Si X et Y sont deux expressions applicatives qui ont des formes normales différentes alors $X \neq Y$.

2.5 Conclusion

La logique combinatoire et le λ-calcul sont les moyens utilisés actuellement par les informaticiens pour analyser les propriétés sémantiques des langages de programmation de haut-niveau. Nous venons d'étudier ces deux théories, et en arrivons à la conclusion que la logique combinatoire de Curry avec les combinateurs est à privilégier, plutôt que le lambda-calcul et la lambda-abstraction de Church. En effet, la théorie combinatoire nous permet d'en arriver à une logique dite sans variable, ce qui résout le problème majeur lié à l'utilisation du lambda-calcul, à savoir celui du télescopage des variables.

Nous retenons donc les combinateurs comme outils pour notre recherche. Le prochain chapitre, consacré à notre recherche théorique, va mettre en évidence tout l'intérêt et la puissance de ces combinateurs.

CHAPITRE 3

RECHERCHE THÉORIQUE

3.1 Introduction

Une chaîne de traitement est l'enchaînement de plusieurs modules. La chaîne, une fois construite et validée, représente un module complexe mémorisé dans une base de données. À cet effet, le logiciel possède une base de données des modules existants, qu'ils soient simples ou complexes. Un module est simple s'il est seul, et complexe si composé de plusieurs modules. Ainsi, l'exécution d'un module simple fera appel à l'exécutable de ce dernier, alors que l'exécution d'un module complexe nécessitera l'exécution des modules qui le composent dans l'ordre spécifié à la construction du module complexe. C'est à ce niveau que se situe notre travail, à savoir, trouver une façon efficace et élégante de sauvegarder l'enchaînement des modules qui composent une chaîne de traitement (ou module complexe).

Notre objectif dans ce chapitre est de montrer que nous pouvons utiliser les combinateurs afin de modéliser les enchaînements de modules. De par leurs caractéristiques comportementales, certains combinateurs vont en effet nous permettre de mémoriser l'ordre d'exécution des modules dans une chaîne de traitement.

Nous allons commencer par présenter le modèle utilisé avec ses notations, ses restrictions et les conséquences qu'elles impliquent. Puis, nous ferons des rappels théoriques pour situer le principe applicatif par rapport aux modules, ainsi que pour récapituler les propriétés applicatives des combinateurs. Ensuite, nous allons considérer les différents schémas d'enchaînement possible des modules, en partant des cas généraux pour aller jusqu'aux cas particuliers, et nous finirons par une étude de cas qui

illustrera la théorie mise en place. Enfin, nous ferons une brève conclusion pour récapituler nos résultats.

3.2 Modèle utilisé

3.2.1 Notations

Soient M un module, I l'ensemble des entrées de M et O l'ensemble de ses sorties, nous noterons M(I) = O ou MI = O, l'application du module M sur I pour produire le résultat O. Nous adopterons la représentation de la figure 3.1 suivante :

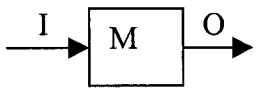


Figure 3.1 - Représentation d'un module M.

Par exemple, I = {int, float} signifie que le module M prend un entier et un réel en entrée. Ou encore, O = {char, bool} signifie que le module M donne un caractère et un booléen en sortie.

Nous noterons I1I2 = I qui signifie que l'ensemble I1 uni avec l'ensemble I2 permet de satisfaire en cardinalité et en types l'ensemble I, que ces ensembles soient ceux d'entrée ou de sortie du module.

3.2.2 Restrictions

Lors de la création de chaînes de traitement, le logiciel permet d'assister l'utilisateur. En effet, l'enchaînement de modules nécessite certaines vérifications. Il faut notamment valider le respect du nombre et du type des entrées de chaque module de la chaîne, et

nous parlerons de contrainte d'intégrité. Si nous voulons pouvoir utiliser les modules complexes construits, il faut de plus s'assurer que les sorties d'un module sont bien reliées à ses entrées, et nous parlerons de contrainte d'utilisation. Ces étapes de contrôle s'effectuent lors de la validation de la chaîne construite par l'utilisateur.

3.2.2.1 Contrainte d'intégrité

La contrainte d'intégrité concerne la cardinalité et les types des entrées. Nous avons alors émis les deux hypothèses restrictives suivantes :

1. L'une concerne la cardinalité des entrées-sorties, c'est-à-dire que nous considérons les entrées et sorties dans leur totalité, sans faire de partition (prendre seulement une partie de l'entrée fournie ou une partie de la sortie produite) ou de sélection (prendre une entrée parmi deux proposées). En effet, pour effectuer une partition ou une sélection, il suffira d'utiliser des modules pour cette tâche. Par exemple, pour une partition, il suffira d'avoir ou de définir un module Mp tel que Mp(I) = O, avec cardinalité de I strictement supérieure à 1 et O qui représente une partition de l'entrée fournie, la partition s'effectuant par le code du module Mp.
2. L'autre suppose que les types connectés sur l'entrée d'un module correspondent à ceux attendus par ce dernier, c'est-à-dire qu'ils sont identiques ou compatibles, sans faire de casting (transformation d'un type en un autre).

Lors de la construction d'une chaîne de traitement, la contrainte d'intégrité a plusieurs conséquences :

- Si deux modules M1 et M2 sont reliés comme sur la figure 3.2 qui suit, avec M1(I1) = O1 et M2(I2) = O2, nous imposons O1 = I2, en cardinalité et types

compatibles, pour ne pas avoir de partition à gérer et ainsi respecter la contrainte d'intégrité des types.

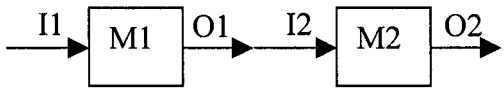


Figure 3.2 - Représentation de 2 modules M1 et M2 reliés.

- Si plusieurs liens sont connectés sur l'entrée d'un module M, avec M(I) = O, comme sur la figure 3.3 qui suit, nous aurons donc IaIb = I pour ne pas avoir de sélection ou partition à gérer. Un module peut donc recevoir plusieurs liens sur son entrée si l'ensemble de ces liens satisfait, en cardinalité et en types, l'entrée attendue du module.

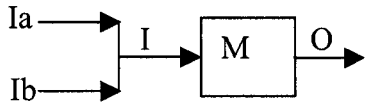


Figure 3.3 - Représentation d'un module M avec deux liens en entrée.

- Si un module M, avec M(I) = O, a sa sortie connectée sur plusieurs liens, comme sur la figure 3.4 qui suit, nous aurons donc Oa = Ob = O pour ne pas avoir de partition à gérer. Un module ne peut donc pas être relié à plusieurs liens différents. Cela revient alors à avoir la sortie O de M avec plusieurs liens identiques, ce qui permet, par exemple, à une même sortie d'être reliée à deux modules distincts qui attendent la même entrée.

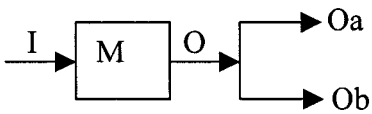


Figure 3.4 - Représentation d'un module M avec deux liens en sortie.

- Si plusieurs modules ont leurs entrées connectées au même lien, par exemple M1 et M2 tels que M1(I1) = O1 et M2(I2) = O2, comme sur la figure 3.5 suivante, nous aurons donc I1 = I2 = I pour ne pas avoir de partition à gérer. Cela revient alors à permettre que plusieurs modules aient leurs entrées connectées sur un même lien.

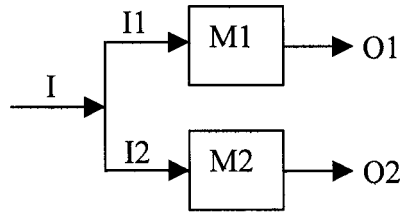


Figure 3.5 - Représentation de deux modules M1 et M2 liés à une entrée.

- Si plusieurs modules ont leurs sorties connectées au même lien, par exemple M1 et M2 tels que M1(I1) = O1 et M2(I2) = O2, comme sur la figure 3.6 suivante, nous aurons donc O1O2 = O pour ne pas avoir de sélection ou partition à gérer. Cela revient alors à permettre que plusieurs modules aient leurs sorties connectées sur un même lien.

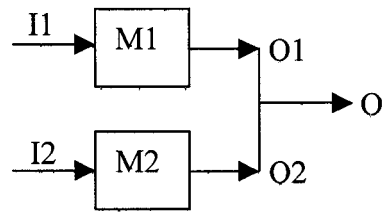


Figure 3.6 - Représentation de deux modules M1 et M2 liés à une sortie.

De plus, la contrainte d'intégrité fait en sorte qu'un module simple peut avoir plusieurs entrées distinctes mais une seule sortie, et qu'un module complexe peut avoir plusieurs entrées distinctes et plusieurs sorties distinctes. En effet :

- Par rapport aux sorties, si nous regardons la figure 3.7 suivante, nous constatons deux cas différents. Premièrement, si Oa, ou Ob, n'est pas explicitement relié à l'entrée I, nous sommes alors dans l'impossibilité d'utiliser M puisque le seul lien connu est OaOb = M(I) et que nous ne pouvons rien déduire sur ce qui uni Oa ou Ob, avec I. Nous rejetons donc ce cas qui correspond à celui d'un module simple avec plusieurs sorties distinctes. Deuxièmement, si Oa et Ob sont explicitement reliés à l'entrée I, nous sommes alors dans le cas d'un module complexe avec plusieurs sorties telles que Oa = f(I) et Ob = g(I), avec f et g des opérateurs. M étant alors utilisable, ce dernier cas est accepté et il concerne bien les modules complexes puisque ceux-ci sont le résultat d'un assemblage de modules simples auxquels nous venons juste d'imposer le maintient d'un lien entre leur sortie et leur entrée.

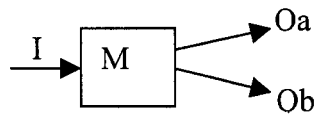


Figure 3.7 - Représentation d'un module M avec deux sorties.

- Par rapport aux entrées, si nous regardons la figure 3.8 suivante, nous avons un seul cas qui est que la ou les sorties sont obligatoirement reliées aux entrées, que ce soit pour un module simple ou complexe. Le lien est O = M(IaIb), dans cet exemple.

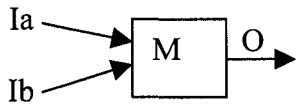


Figure 3.8 - Représentation d'un module M avec deux entrées.

Cependant, pour des raisons d'analogie entre un module simple et une fonction, nous limiterons le nombre d'entrées des modules simples à un seul ensemble. Nous aurons donc un seul ensemble d'entrée et un seul de sortie pour les modules simples, et plusieurs ensembles possibles en entrée et en sortie pour les modules complexes.

3.2.2.2 Contrainte d'utilisation

Une dernière restriction est reliée à l'utilisation des modules complexes, nous l'appellerons la contrainte d'utilisation. Dans le cas d'un module complexe avec plusieurs ensembles en sortie, il nous faut bien conserver la relation de chaque ensemble avec les éléments en entrée si nous voulons réussir à utiliser une sortie indépendamment des autres lors de sa connexion sur un module quelconque. Soit l'exemple présenté à la figure 3.9 suivante, nous avons Mc(I) = OaOb, M1(I1) = O1, et Oa = I1 d'après nos restrictions précédentes. Le module Mr, résultant de cette chaîne de traitement, sera tel que : Mr(I) = O1Ob = (M1I1)Ob = (M1Oa)Ob. Nous constatons qu'il est nécessaire de connaître la relation précise de Oa et Ob avec I, car nous ne pouvons pas nous contenter de la relation OaOb = Mc(I).

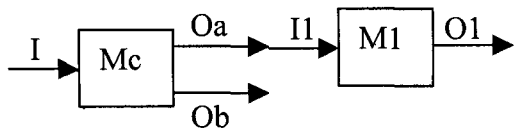


Figure 3.9 - Représentation d'un module complexe Mc avec deux sorties, dont une est reliée à un module simple M1.

Nous imposerons alors de conserver entre parenthèses la relation de chaque sortie d'un module avec les éléments en entrée, et nous noterons :

- L'entrée de Mc est I,
- Les sorties de Mc sont Oa et Ob, avec Oa = f(I) et Ob = g(I), f et g étant l'expression d'un agencement des modules qui constituent Mc,
- L'exécution de Mc sera (Oa) (Ob) = (f(I)) (g(I)).

Dans le cas d'un module avec une seule sortie, les parenthèses sont aussi imposées. En effet, leur omission risquerait de prêter à confusion si nous nous référons au cas de la partie 3.4.2.3.1. Nous aurons donc systématiquement des parenthèses pour nos sorties.

3.2.3 Conséquences

Un module simple aura un seul ensemble d'entrée et un seul autre de sortie, tandis qu'un module complexe pourra en avoir plusieurs en entrée et plusieurs en sortie. Chaque ensemble de sortie sera entre parenthèses, et chaque ensemble d'entrée sera clairement identifié. Ainsi, si nous considérons le module complexe Mc représenté à la figure 3.10 suivante, sa chaîne de traitement sera (M1I1)(M2(M1I2)). Cette représentation nous permet de savoir immédiatement que Mc possède deux sorties (d'après les parenthèses) et deux entrées (d'après I1 et I2).

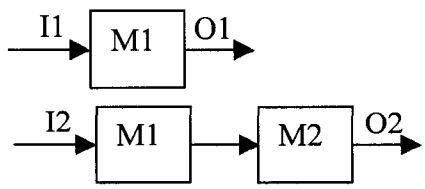


Figure 3.10 - Représentation d'un module complexe Mc dont la chaîne de traitement est (M1I1)(M2(M1I2)).

De plus, cette notation nous permet de distinguer le module complexe de la figure 3.10 de celui de la figure 3.11 qui suit.

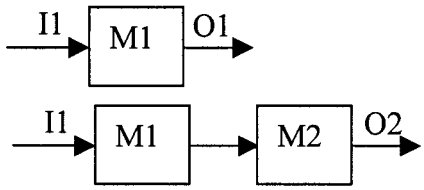


Figure 3.11 - Représentation d'un module complexe Mc dont la chaîne de traitement est (M1I1)(M2(M1I1)).

Dans le cas de la figure 3.10, les ensembles d'entrée I1 et I2 peuvent être différents. Dans le cas de la figure 3.11, il y a un unique ensemble d'entrée I1. Ainsi, ces deux modules complexes seront différents.

Nous constatons alors que le modèle est adapté pour distinguer les différents cas de figure possibles. Il nous reste cependant une dernière considération à faire. En effet, l'objectif actuel de la recherche étant de trouver un moyen de modéliser l'enchaînement des modules dans une chaîne de traitement, nous devons aussi considérer que les modules sont bien connectés et que les entrées des modules de la chaîne sont déjà vérifiées et validées. Ainsi, grâce aux contraintes d'intégrité et d'utilisation, nous aurons à faire à des chaînes de traitement valides. Cette étape de validation s'effectue lors de la construction d'une chaîne. Vu que notre travail consiste à modéliser l'enchaînement des modules d'une chaîne, afin que ces derniers soient exécutés dans l'ordre désiré, nous posons comme hypothèse que les chaînes sont correctement construites avant d'être modélisées.

Après avoir présenté le modèle utilisé et les hypothèses de départ, continuons par reprendre quelques éléments de la théorie du chapitre précédent qui vont nous être utiles, à savoir, le principe applicatif et les combinateurs.

3.3 Rappels théoriques

3.3.1 Modules et principe applicatif

Dans la théorie des combinateurs, nous évoluons dans un système applicatif, et à ce titre, nous pouvons voir nos modules comme des opérateurs dont les entrées seraient leurs arguments et dont les sorties seraient le résultat de l'application de l'opérateur à ses arguments.

Le principe applicatif permet de passer à une logique avec une variable. En effet, un module qui recevrait plusieurs entrées ou arguments sera vu comme un opérateur prenant le premier argument pour former un opérateur complexe qui prendra le deuxième argument pour former un autre opérateur complexe qui prendra ... pour former un opérateur complexe qui appliqué au dernier argument donnera le résultat.

Voici un exemple pour illustrer ce principe. Soit un module M qui prend deux entiers en entrée et dont le résultat en sortie est un entier qui est la somme des deux arguments. Ce module permet donc de faire l'addition de deux entiers, et nous aurons ainsi $M(x,y) = z$, avec $z = x + y$. Ainsi, si nous appliquons M à deux arguments en même temps, nous aurons $M(2,3) = 2 + 3 = 5$, par exemple. D'après le principe applicatif, nous pouvons appliqué M à un argument seulement et obtenir un opérateur complexe M' qui s'appliquera au deuxième argument pour donner le résultat. Prenons les deux cas possibles, à savoir :

- M que nous appliquons tout d'abord à $x = 2$ pour former l'opérateur complexe M' qui s'appliquera à y. Avec $x = 2$, nous avons donc $M'(y) = 2 + y$. Ainsi, avec $y = 3$, nous obtenons bien le bon résultat $M'(3) = 2 + 3 = 5$.
- M que nous appliquons tout d'abord à $y = 3$ pour former l'opérateur complexe M' qui s'appliquera à x. Avec $y = 3$, nous avons donc $M'(x) = x + 3$. Ainsi, avec $x = 2$, nous obtenons bien le bon résultat $M'(2) = 2 + 3 = 5$.

Nous voyons donc que l'ordre d'application aux arguments n'a pas d'importance. Cette logique à une variable va nous être très utile dans la mesure où nous devons enchaîner des modules et qu'elle permet d'évaluer provisoirement un module qui appliqué à un argument donne un module complexe applicable au prochain argument, etc.

3.3.2 Récapitulatif sur les combinateurs élémentaires

Soient f, g, h des opérateurs et x, y des arguments. L'application des combinateurs est définie de la façon suivante :

Identité	I f	→	f
Composition	B f g x	→	f (g x)
Substitution	S f g x	→	f x (g x)
Coordination	Φ f g h x	→	f (g x) (h x)
Distribution	Ψ f g x y	→	f (g x) (g y)
Changement de type	C* X Y	→	Y X
Permutation	C f y x	→	f x y
Duplication	W f x	→	f x x
Effacement	K f x	→	f

Compte tenu de ces définitions, nous pouvons tout de suite présumer que le combinateur d'identité I est adapté pour un module simple. En effet, l'exécution d'un module simple nécessite uniquement d'exécuter le programme propre au module lui-même. Il suffira alors de mémoriser I devant le nom de l'exécutable des modules simples. Soit M un tel module et E son exécutable, nous aurons alors le champ exécutable de M qui sera IE. La règle applicative de I (I f → f) permet de passer de IE à E lors de l'exécution. Cette notation offre l'avantage de pouvoir tout de suite reconnaître un module simple.

En ce qui concerne les modules complexes, le combinateur de composition B paraît adapté au traitement en série et le combinateur de coordination Φ semble bon pour le traitement en parallèle. En effet, dans [DESC 1990, page 157], nous retrouvons que « le combinateur Φ exprime des situations de calculs parallèles ».

Dans la prochaine partie, nous allons étudier les différents cas généraux d'enchaînements qui peuvent caractériser les modules complexes.

3.4 Cas généraux d'enchaînement de plusieurs modules

Dans cette partie, nous allons voir le cas de modules qui s'enchaînent en série, le cas de modules qui s'enchaînent en parallèle, et le cas de modules qui combinent ces deux approches.

3.4.1 Traitement en série

Dans ce cas, le module complexe est caractérisé par l'enchaînement en série de plusieurs modules existants. Le combinateur B, de par sa définition, est intéressant pour représenter un enchaînement en série de deux modules. En effet, la règle applicative de

B, B f g x → f (g x), montre que f va être appliquée au résultat de g appliquée à x, ce qui représente bien l'enchaînement en série de g puis f.

Mais il faut valider cette hypothèse et voir si B peut aussi être utilisé dans le cas de n modules mis en série. Nous commencerons donc par voir le cas de deux modules en série, puis de trois modules, et enfin de n modules.

3.4.1.1 Deux modules en série

Soient deux modules M1 et M2, avec M1(I1) = O1 et M2(I2) = O2, si nous mettons ces modules en série, nous obtenons la représentation donnée à la figure 3.12 suivante :

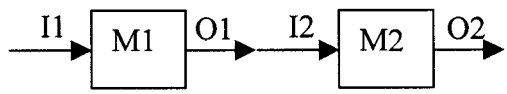


Figure 3.12 - Représentation de 2 modules en série.

D'après nos restrictions de départ, nous avons O1 = I2, en cardinalité et en types compatibles. Nous aurons donc :

O2 = M2(I2) = M2(M1(I1)) (3.1)

En utilisant B, nous obtenons :

O2 = B M2 M1 I1

Effectivement, en appliquant B, nous aurons :

B M2 M1 I1 = M2(M1I1) (application de B)
= M2(M1(I1)) (restauration des parenthèses)
= O2 (d'après 3.1)

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I1) = O2

Avec I1 les entrées de Mr et O2 ses sorties. Pour exécuter Mr, il faudra exécuter M1 suivi de M2. Nous mémorisons donc que l'exécutable de Mr sera :

(B M2 M1 I1)

L'exécutable est mémorisé avec des parenthèses d'après notre contrainte d'utilisation. Ceci ne sera pas répété dans chaque partie afin d'en alléger le contenu. De plus, nous incluons I1 pour savoir exactement où cette entrée est utilisée. Cela peut sembler inutile dans cet exemple, mais sera très intéressant dès qu'il s'agira de gérer des modules plus complexes ayant plusieurs entrées distinctes ou plusieurs fois la même entrée.

Jusque là, le combinateur B répond à nos besoins. Voyons ce qu'il en est avec trois modules qui se suivent.

3.4.1.2 Trois modules en série

Soient trois modules, M1, M2 et M3, avec M1(I1) = O1, M2(I2) = O2 et M3(I3) = O3, si nous mettons ces modules en série, nous obtenons la représentation de la figure 3.13 suivante :

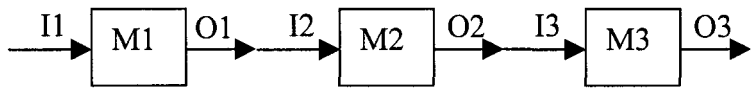


Figure 3.13 - Représentation de 3 modules en série.

D'après nos restrictions de départ, nous avons O1 = I2 et O2 = I3, en cardinalité et en types compatibles. Nous aurons donc :

O3 = M3(I3) = M3(M2(I2)) = M3(M2(M1(I1))) (3.2)

En utilisant B, nous obtenons :

O3 = M3 (B M2 M1 I1) = B M3 (B M2 M1) I1

Effectivement, en appliquant B, nous aurons :

B M3 (B M2 M1) I1 = M3 (B M2 M1 I1)	(application de B)
= M3(M2(M1I1))	(application de B)
= M3(M2(M1(I1)))	(restauration des parenthèses)
= O3	(d'après 3.2)

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I1) = O3

Avec I1 les entrées de Mr et O3 ses sorties. Pour exécuter Mr, il faudra exécuter M1, puis M2, puis M3. Nous mémorisons donc que l'exécutable de Mr sera :

(B M3 (B M2 M1) I1)

Jusque là, le combinateur B répond toujours à nos besoins. Voyons ce qu'il en est si nous généralisons à n modules qui se suivent.

3.4.1.3 N modules en série

Soient n modules, M1, M2, ..., et Mn, avec M1(I1) = O1, M2(I2) = O2, ..., et Mn(In) = On, si nous mettons ces modules en série, nous obtenons la représentation de la figure 3.14 suivante :

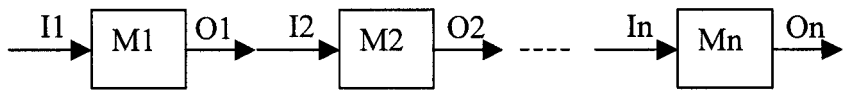


Figure 3.14 - Représentation de n modules en série.

D'après nos restrictions de départ, nous avons O1 = I2, O2 = I3, ..., et On-1 = In, en cardinalité et en types compatibles. Nous aurons donc :

On = Mn(In) = Mn(Mn-1(On-1)) = ... = Mn(Mn-1(...(M3(M2(M1(I1))))...))

En appliquant B progressivement aux deux premiers en série, puis aux trois, etc., nous aurons :

On = Mn(Mn-1(...(M3(B M2 M1 I1))...)) (2 en série)

= Mn(Mn-1(...(B M3 (B M2 M1) I1))...)) (3 en série)

= ...

= B Mn (B Mn-1 (...(B M3 (B M2 M1)...))I1) (n en série)

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I1) = On

Avec I1 les entrées de Mr et On ses sorties. Pour exécuter Mr, il faudra exécuter M1, puis M2, ..., puis Mn. Nous mémorisons donc que l'exécutable de Mr sera :

(B Mn (B Mn-1 (...(B M3 (B M2 M1))...) I1)

Le combinateur B répond donc bien au traitement en série. Voyons maintenant ce qu'il en est du traitement en parallèle.

3.4.2 Traitement en parallèle

Dans ce cas, le module complexe est caractérisé par l'enchaînement en parallèle de plusieurs modules existants. Nous allons distinguer 5 parallélismes différents : le

parallélisme indépendant, le parallélisme à entrées dépendantes, le parallélisme à sorties dépendantes, le parallélisme dépendant, et le parallélisme avec récupération.

3.4.2.1 Parallélisme indépendant

Le parallélisme indépendant signifie que les entrées du module complexe résultant n'auront aucun lien entre elles, et que les sorties du module complexe résultant n'auront aussi aucun lien entre elles.

3.4.2.1.1 Parallélisme indépendant de deux modules

Soient M1 et M2, deux modules avec M1(I1) = O1 et M2(I2) = O2, si nous mettons ces modules en parallèle, nous obtenons la représentation de la figure 3.15 suivante :

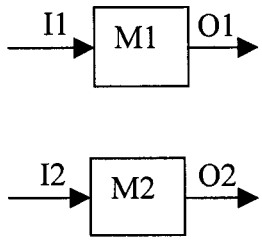


Figure 3.15 – Parallélisme indépendant de 2 modules.

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I1I2) = O1O2

Avec I1 et I2 les entrées de Mr, et O1 et O2 ses sorties. Pour exécuter Mr, il faudra exécuter M1 et M2. Nous mémorisons donc que l'exécutable de Mr sera :

(M1I1)(M2I2)

Comptes tenus de nos hypothèses de départ, puisque O1 est relié à I1 par O1 = M1(I1) et que O2 est relié à I2 par O2 = M2(I2), nous acceptons que le module complexe final ait deux sorties distinctes. Mais il faut les parenthèses pour respecter notre restriction d'utilisation, ce qui permet de tout de suite voir que Mr a deux sorties séparées.

Nous pouvons facilement déduire le cas de n modules en parallélisme indépendant.

3.4.2.1.2 Parallélisme indépendant de n modules

Soient M1, M2, M3, ..., et Mn, n modules en parallèle avec M1(I1) = O1, M2(I2) = O2, M3(I3) = O3, ..., et Mn(In) = On, si nous mettons ces modules en parallèle, nous obtenons la représentation de la figure 3.16 suivante :

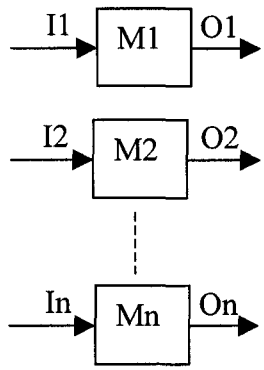


Figure 3.16 – Parallélisme indépendant de n modules.

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I1I2...In) = O1O2...On

Avec I1, I2, ..., In les entrées de Mr, et O1, O2, ..., On ses sorties. Pour exécuter Mr, il faudra exécuter M1, M2, ..., Mn. D'après les définitions de O1, O2, ..., et On, nous mémorisons donc que l'exécutable de Mr sera :

(M1I1)(M2I2)...(MnIn)

Nous constatons qu'aucun combinateur n'est utilisé dans ce cas. Passons maintenant au cas suivant.

3.4.2.2 Parallélisme à entrées dépendantes

Le parallélisme à entrées dépendantes signifie que les entrées du module complexe résultant sont liées entre elles, et que les sorties du module complexe résultant n'ont aucun lien entre elles.

3.4.2.2.1 Parallélisme à entrées dépendantes de deux modules

Soient M1 et M2, deux modules avec M1(I1) = O1 et M2(I2) = O2, si nous mettons ces modules en parallèle, nous obtenons la représentation de la figure 3.17 suivante :

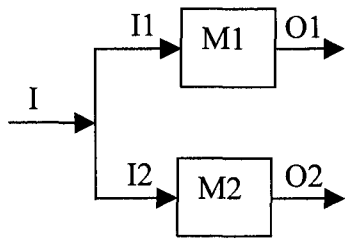


Figure 3.17 – Parallélisme à entrées dépendantes de 2 modules.

D'après nos hypothèses de départ, nous avons I1 = I2 = I. Ainsi, si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I) = O1O2

Avec I l'entrée de Mr, et O1 et O2 ses sorties. Pour exécuter Mr, il faudra exécuter M1 et M2. Comme I1 = I2 = I, nous mémorisons que l'exécutable de Mr sera :

(M1I)(M2I)

Ce cas de parallélisme est identique au parallélisme indépendant étudié précédemment, avec pour seule différence que toutes les entrées sont égales. Il nous suffit alors de prendre les résultats de la partie 3.4.2.1 et de mettre toutes les entrées égales. Nous pouvons alors facilement déduire le cas de n modules en parallélisme à entrées dépendantes.

3.4.2.2.2 Parallélisme à entrées dépendantes de n modules

Soient M1, M2, M3, ..., et Mn, n modules en parallèle avec M1(I1) = O1, M2(I2) = O2, M3(I3) = O3, ..., et Mn(In) = On. Si nous mettons ces modules en parallèle, nous obtenons la représentation de la figure 3.18 suivante :

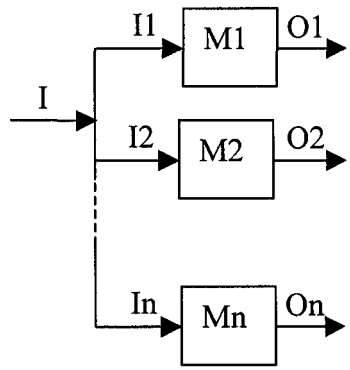


Figure 3.18 – Parallélisme à entrées dépendantes de n modules.

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I) = O1O2...On

Avec I l'entrée de Mr, et O1, O2, ..., On ses sorties. Pour exécuter Mr, il faudra exécuter M1, M2, ..., Mn. Dans la partie 3.4.2.1.2 nous avions le résultat (M1I1)(M2I2)...(MnIn), avec I = I1 = I2 = ... = In, nous mémorisons donc que l'exécutable de Mr sera :

(M1I)(M2I)...(MnI)

Nous constatons qu'ici aussi aucun combinateur n'est utilisé. Passons au cas suivant.

3.4.2.3 Parallélisme à sorties dépendantes

Le parallélisme à sorties dépendantes signifie que les entrées du module complexe résultant n'ont aucun lien entre elles, et que les sorties du module complexe résultant sont liées entre elles.

3.4.2.3.1 Parallélisme à sorties dépendantes de deux modules

Soient M1 et M2, deux modules avec M1(I1) = O1 et M2(I2) = O2, si nous mettons ces modules en parallèle, nous obtenons la représentation de la figure 3.19 suivante :

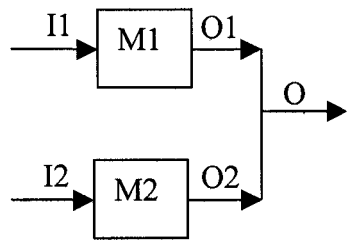


Figure 3.19 – Parallélisme à sorties dépendantes de 2 modules.

D'après nos hypothèses de départ, nous avons O1O2 = O. Ainsi, si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I1I2) = O1O2 = O

Avec I1 et I2 les entrées de Mr, et O sa sortie. Pour exécuter Mr, il faudra exécuter M1 et M2. D'après la définition de O1, O2, et O, nous mémorisons donc que l'exécutable de Mr sera :

((M1I1)(M2I2))

Attention aux parenthèses pour respecter notre restriction d'utilisation. En effet, nous avons une seule sortie pour la chaîne de traitement résultante, et nous englobant alors O1O2 de parenthèses pour enlever toute confusion. Effectivement, sans ses dernières nous aurions (M1I1)(M2I2) l'exécutable de Mr et pourrions croire à deux sorties distinctes, ce qui n'est pas le cas ici. Voilà donc pourquoi nous mettons des parenthèses autour d'un ensemble de sorties même s'il est unique.

Ce cas de parallélisme est identique au parallélisme indépendant étudié précédemment, avec pour seule différence que toutes les sorties sont regroupées en une seule. C'est pourquoi il suffit de prendre nos résultats de la partie 3.4.2.1, et de regrouper les ensembles de sorties en un seul par le simple ajout des parenthèses autour des résultats. Nous pouvons alors facilement déduire le cas de n modules en parallélisme à sorties dépendantes.

3.4.2.3.2 Parallélisme à sorties dépendantes de n modules

Soient M1, M2, M3, ..., et Mn, n modules en parallèle avec M1(I1) = O1, M2(I2) = O2, M3(I3) = O3, ..., et Mn(In) = On. Si nous mettons ces modules en parallèle, nous obtenons la représentation de la figure 3.20 suivante :

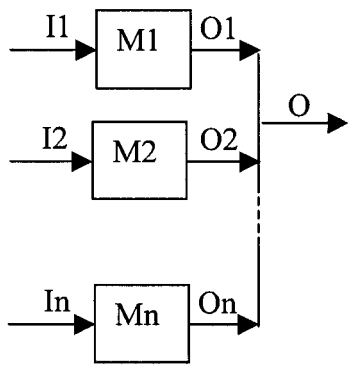


Figure 3.20 – Parallélisme à sorties dépendantes de n modules.

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I1I2…In) = O = O1O2…On

Avec I1, I2, …, In les entrées de Mr, et O sa sortie. Pour exécuter Mr, il faudra exécuter M1, M2, …, Mn. Dans la partie 3.4.2.1.2 nous avions le résultat (M1I1)(M2I2)…(MnIn), avec l'ajout des parenthèses autour de ce résultat, nous mémorisons que l'exécutable de Mr sera :

((M1I1)(M2I2)…(MnIn))

Nous constatons encore qu'aucun combinateur n'est utilisé. Passons maintenant au quatrième cas de parallélisme.

3.4.2.4 Parallélisme dépendant

Le parallélisme dépendant signifie que les entrées du module complexe résultant sont liées entre elles, et que les sorties du module complexe résultant sont aussi liées entre elles.

3.4.2.4.1 Parallélisme dépendant de deux modules

Soient M1 et M2, deux modules avec M1(I1) = O1 et M2(I2) = O2, si nous mettons ces modules en parallèle, nous obtenons la représentation de la figure 3.21 suivante :

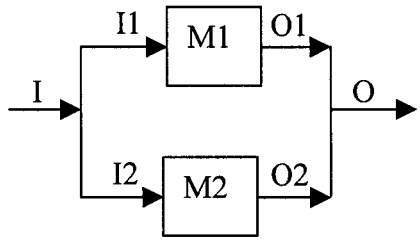


Figure 3.21 – Parallélisme dépendant de 2 modules.

D'après nos hypothèses de départ, nous avons I1 = I2 = I et O1O2 = O. Ainsi, si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I) = O1O2 = O

Avec I l'entrée de Mr, et O sa sortie. Pour exécuter Mr, il faudra exécuter M1 et M2. D'après les définitions de O, O1, O2, I1, et I2, nous mémorisons l'exécutable de Mr :

((M1I)(M2I))

Ce cas de parallélisme est un mélange des deux formes de parallélisme étudiées précédemment. Pour obtenir le résultat, il suffit donc de prendre celui correspondant au parallélisme indépendant de la partie 3.4.2.1, de lui ajouter des parenthèses autour et de poser toutes les entrées égales. Nous pouvons alors facilement déduire le cas de n modules en parallélisme dépendant.

3.4.2.4.2 Parallélisme dépendant de n modules

Soient M1, M2, M3, ..., et Mn, n modules en parallèle avec M1(I1) = O1, M2(I2) = O2, M3(I3) = O3, ..., et Mn(In) = On. Si nous mettons ces modules en parallèle, nous obtenons la représentation de la figure 3.22 suivante :

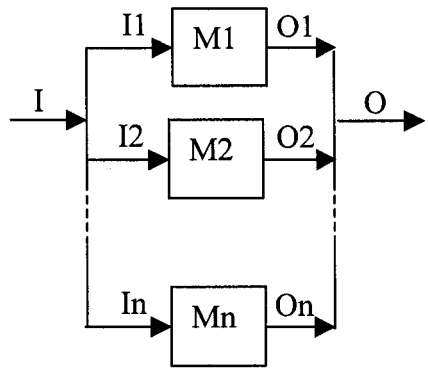


Figure 3.22 – Parallélisme dépendant de n modules.

D'après nos hypothèses de départ, nous avons I1 = I2 = … = In = I et O1O2…On = O. Ainsi, si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I) = O = O1O2…On

Avec I l'entrée de Mr, et O sa sortie. Pour exécuter Mr, il faudra exécuter M1, M2, …, Mn. D'après le résultat de la partie 3.4.2.1.2, nous mémorisons que l'exécutable de Mr sera :

((M1I)(M2I)…(MnI))

Nous constatons qu'ici aussi aucun combinateur n'est encore utilisé. Passons maintenant au dernier cas de parallélisme.

3.4.2.5 Parallélisme avec récupération

Le parallélisme avec récupération est identique au parallélisme dépendant avec pour seule différence que le résultat est récupéré dans un module. L'intérêt de définir ce type de parallélisme est qu'il représente des cas courants d'agencements de modules.

3.4.2.5.1 Parallélisme de deux modules avec récupération

Soient M1 et M2, deux modules avec M1(I1) = O1 et M2(I2) = O2, si nous mettons ces modules en parallèle, et transmettons le résultat dans un module M' tel que M'(I') = O', d'après nos hypothèses de départ, nous avons I1 = I2 = I et O1O2 = I'. Nous obtenons alors la représentation de la figure 3.23 suivante :

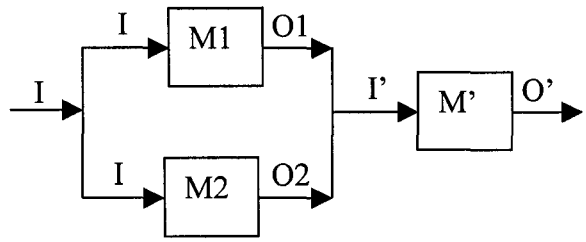


Figure 3.23 – Parallélisme de 2 modules avec récupération.

Nous aurons donc :

$$O' = M'(I') = M'(O1O2) = M'(M1(I)M2(I)) = M'((M1I)(M2I)) \quad (3.3)$$

Nous aurions pu déduire directement ce résultat en remarquant que I' = ((M1I)(M2I)), puisque M1 et M2 sont en parallélisme dépendant comme dans la partie 3.4.2.4.

Le combinateur Φ, de par sa définition, est intéressant pour représenter ce parallélisme des modules. En effet, la règle applicative de Φ, Φ f g h x → f (g x) (h x), montre que f va être appliquée au résultat de g appliquée à x et au résultat de h appliquée à x, puisque si nous remettons des parenthèses, f (g x) (h x) signifie en fait, f ((g x) (h x)). Cela représente bien le parallélisme de g et h dont les résultats sont récupérés dans f.
En utilisant Φ, nous obtenons donc :

O' = Φ M' M1 M2 I

Effectivement, en appliquant Φ, nous aurons :

Φ M' M1 M2 I = M'(M1I)(M2I)	(application de Φ)
= M'((M1I)(M2I))	(restauration des parenthèses)
= O'	(d'après 3.3)

Dans cette façon de faire, nous avons appliqué M' à O1 et O2. Mais avec le principe applicatif, nous pouvons raisonner autrement en appliquant M' à O1 pour former M'' qui appliqué à O2 donnera le résultat O'. Ainsi M'' = M'O1 et :

O' = (M'O1)O2 = M''O2

Si nous développons, nous avons :

M'' = M'O1 = M'(M1I) = B M' M1 I

Et alors :

O' = M''O2 = B M' M1 I (M2 I)

Dans ce cas, nous ne pouvons utiliser Φ pour représenter O', mais le combinateur S est intéressant ici. En effet, la nature applicative de S, S f g x → f x (g x), signifie que f est appliqué à x pour donner fx, lui-même appliqué à gx. Ceci rejoint notre cas puisque nous avons M'' = B M' M1 I appliqué à O2 = M2 I, alors :

O' = S (B M' M1) M2 I

L'utilisation du combinateur Φ est donc équivalente à celle de S et B avec :

Φ M' M1 M2 I = S (B M' M1) M2 I

Ceci nous amène à voir un lien entre le combinateur Φ et les combinateurs S et B. En effet, d'après [DESC 1990, page 178], il existe la relation suivante entre Φ, S et B :

Φ = B(BS)B

Démonstration de Φ = B (B S) B

Soient f, g et h des opérateurs et x un opérande.

(1) B (B S) B f g h x = B S (B f) g h x (application de B à (B S), B et f)
= S (B f g) h x (application de B à S, (B f) et h)
= B f g x (h x) (application de S à (B f g), h et x)
= f (g x) (h x) (application de B à f, g et x)
= Φ f g h x (définition : f (g x) (h x) = Φ f g h x)

Nous avons donc B (B S) B implique Φ.

(2) Φ f g h x = f (g x) (h x) (application de Φ à f, g, h et x)
= B f g x (h x) (définition : f (g x) = B f g x, avec f = f et g = g)
= S (B f g) h x (définition : f x (g x) = S f g x, avec f = (B f g) et g = h)
= B S (B f) g h x (définition : f (g x) = B f g x, avec f = S, g = (B f) et x = g)
= B (B S) B f g h x (définition : f(g x) = B f g x, avec f = (B S), g = B et x = f)

Nous avons donc Φ implique B (B S) B.

D'après (1) et (2) nous déduisons l'égalité : Φ = B (B S) B.

Nous pouvons remarquer les égalités suivantes :

Φ f g h x = S (B f g) h x = B (B S) B f g h x

Pour des questions d'implémentation et d'optimisation, nous pouvons nous demander s'il est préférable d'utiliser le combinateur Φ ou son équivalent B(BS)B?

- Du point de vu de l'espace mémoire, nous pourrions faire des économies avec l'emploie de Φ puisque nous utilisons alors un seul symbole au lieu de quatre (ou six avec les parenthèses) lors de la modélisation des chaînes de traitement. Mais de nos jours, avec des ordinateurs aux capacités de stockage énormes, cet avantage ne fait pas vraiment de différence.
- Au niveau du temps d'exécution, nous pourrions faire des économies en utilisant B(BS)B à la place de Φ. En effet, lors de l'étape de modélisation, la phase 2 de recherche des schémas des combinateurs serait alors plus rapide puisqu'il n'y aurait que deux schémas (ceux de B et S) au lieu de trois à retrouver. Néanmoins, d'après la partie (2) de la démonstration précédente, il apparaît plus facile de repérer Φ que B(BS)B en ayant la forme de départ sans combinateurs.

D'après ce qui vient d'être exposé, nous utiliserons dorénavant le combinateur Φ, et non son équivalent B(BS)B.

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I) = O', avec I les entrées de Mr et O' ses sorties.

Pour exécuter Mr, il faudra exécuter M1 en parallèle avec M2, le tout suivi de M'. Nous mémorisons donc que l'exécutable de Mr sera :

(Φ M' M1 M2 I)

Jusque là, le combinateur Φ répond à nos besoins. Mais il faut valider cette hypothèse dans le cas de n modules mis en parallèle. Continuons alors tout d'abord par voir le cas de trois modules en parallèle.

3.4.2.5.2 Parallélisme de trois modules avec récupération

Soient M1, M2 et M3, trois modules en parallèle avec M1(I1) = O1, M2(I2) = O2 et M3(I3) = O3. Si nous mettons ces modules en parallèle, et transmettons le résultat dans un module M' tel que M'(I') = O', d'après nos hypothèses de départ, nous avons O1O2O3 = I' et I1 = I2 = I3 = I. Nous obtenons alors la représentation de la figure 3.24 suivante :

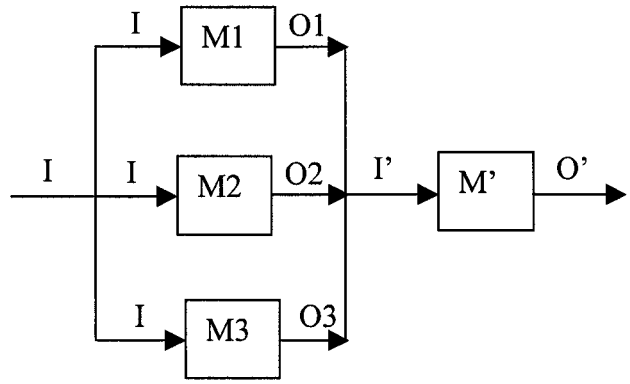


Figure 3.24 – Parallélisme de 3 modules avec récupération.

Nous aurons donc :

$$\begin{aligned} O' = M'(I') &= M'(O1O2O3) \\ &= M'(M1(I)M2(I)M3(I)) \\ &= M'((M1I)(M2I)(M3I)) \end{aligned} \quad (3.4)$$

En utilisant le principe applicatif, l'exécution du module résultant donnera :

$$O' = M'O1O2O3 \quad (3.5)$$
$$= (M'O1)O2O3$$
$$= M''O2O3 \quad \text{(avec M' qui appliqué à O1 donne un opérateur complexe M'')}$$
$$= (M''O2)O3$$
$$= M'''O3 \quad \text{(avec M'' qui appliqué à O2 donne un opérateur complexe M''')}$$
$$= O'$$

Nous voyons, grâce à la notation applicative, qu'il est alors équivalent de considérer M' qui s'applique à M1 et M2 en parallèle pour former un opérateur complexe M'' qui sera appliqué à O3, avec M'' = Φ M' M1 M2 I. Cela nous permet de retomber sur le cas de deux modules en parallèle, et nous avons :

O' = M'((M1I)(M2I)(M3I)) (d'après 3.4)
= (M'(M1I)(M2I)) (M3I) (M' appliqué à M1 et M2 en parallèle)
= (Φ M' M1 M2 I) (M3I) (utilisation de Φ)

Cependant, l'application de M'' à O3 ne peut pas se modéliser avec le combinateur Φ. Dans ce cas, comme dans la partie 3.4.2.5.1 précédente, le combinateur S est intéressant. Effectivement, avec S nous obtenons :

O' = (Φ M' M1 M2 I) (M3I) = S (Φ M' M1 M2) M3 I

En effet :

S (Φ M' M1 M2) M3 I = Φ M' M1 M2 I (M3I) (application de S)
= M' (M1I) (M2I) (M3I) (application de Φ)
= M' O1 O2 O3 (hypothèses)
= O' (d'après 3.5)

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I) = O'

Avec I les entrées de Mr et O' ses sorties. Pour exécuter Mr, il faudra exécuter M1, M2 et M3 en parallèle, le tout suivi de M'. Nous mémorisons donc que l'exécutable de Mr sera :

(S (Φ M' M1 M2) M3 I)

Les combinateurs S et Φ sont satisfaisants pour l'instant. Voyons ce qu'il en est si nous généralisons à n modules en parallèle.

3.4.2.5.3 Parallélisme de n modules avec récupération

Soient M1, M2, ..., et Mn, n modules en parallèle avec M1(I1) = O1, M2(I2) = O2, ..., et Mn(In) = On. Si nous mettons ces modules en parallèle, et transmettons le résultat dans un module M' tel que M'(I') = O', d'après nos hypothèses de départ, nous avons O1O2...On = I' et I1 = I2 = ... = In = I. Nous obtenons alors la représentation de la figure 3.25 suivante :

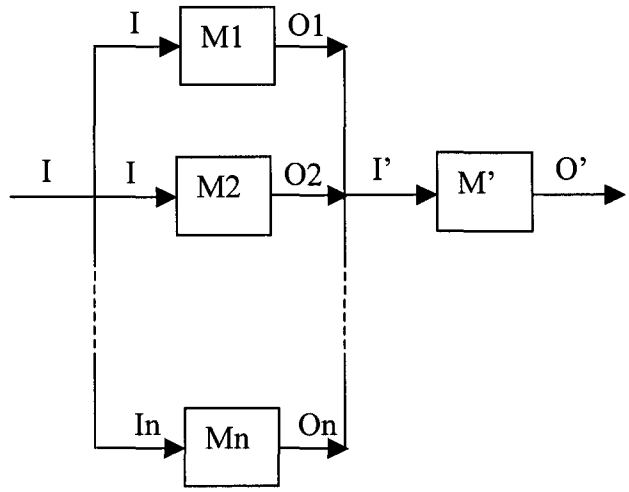


Figure 3.25 – Parallélisme de n modules avec récupération.

Nous aurons donc :

$$O' = M'(I') = M'(O1O2...On)$$
$$= M'(M1(I)M2(I)...Mn(I))$$
$$= M'((M1I)(M2I)...(MnI))$$

Nous voyons alors tout de suite, grâce à la notation applicative, le lien avec les cas précédents. En effet, d'après la partie 3.4.2.5.2 précédente, nous voyons qu'il est possible, pour un cas de n modules en parallèle, de se ramener à un cas de n-1 modules en parallèle. Grâce à la récursivité, nous pouvons donc aboutir à un cas de deux modules en parallèle dont nous connaissons la solution.

À savoir que l'obtention de O' sera le résultat de M' appliqué à M1 et M2 en parallèle, pour former l'opérateur complexe M'' qui sera appliqué à O3, pour former l'opérateur complexe M''' qui sera appliqué à O4, ... , pour former l'opérateur complexe M^{n-1} qui sera appliqué à On.

Pour ce faire, il suffit de poser :

- M'' l'opérateur complexe résultant de M1 et M2 en parallèle,
- M''' l'opérateur complexe résultant de M'' appliqué à O3,
- M'''' l'opérateur complexe résultant de M''' appliqué à O4,
- ...,
- M^{n-1} l'opérateur complexe résultant de M^{n-2} appliqué à On-1,
- O' le résultat de M^{n-1} appliqué à On.

Nous avons donc :

- M'' = Φ M' M1 M2 I,
- M''' = M'' O3 = Φ M' M1 M2 I (M3I) = S (Φ M' M1 M2) M3 I,
- M'''' = M''' O4 = S (S (Φ M' M1 M2) M3) M4 I,
- ...,
- M^{n-1} = M^{n-2} On-1 = S (...(S (S (Φ M' M1 M2) M3) M4)...) Mn-1 I,
- O' = M^{n-1} On = S (S (...(S (S (Φ M' M1 M2) M3) M4)...) Mn-1) Mn I.

Nous en déduisons immédiatement que si Mr représente le module complexe résultant de la chaîne de traitement précédente, nous avons :

Mr(I) = O', avec I les entrées de Mr et O' ses sorties.

Pour exécuter Mr, il faudra exécuter M1, M2, ..., et Mn en parallèle, le tout suivi de M'.

Nous mémorisons donc que l'exécutable de Mr sera :

(S (S (...(S (Φ M' M1 M2) M3)...) Mn-1) Mn I)

Nous avons donc vu que le combinateur B répond bien au traitement en série et que les combinateurs Φ et S répondent bien au traitement en parallèle avec récupération. Nous allons voir maintenant, à travers quelques exemples, ce qu'il en est lorsque nous combinons le traitement en série avec le traitement en parallèle.

3.4.3 Combinaison série-parallèle

Dans ce cas, le module complexe est caractérisé par l'enchaînement de modules existants en combinant l'approche sérielle et parallèle. Ce type de module complexe vient compléter l'ensemble des enchaînements généraux possibles. Effectivement, une chaîne de traitement peut avoir les modules qui la composent juste en série, ou juste en parallèle, ou bien en série et parallèle mélangés.

Étant donné que nous avons pu établir des lois générales concernant l'enchaînement de n modules, qu'ils soient en série ou en parallèle, nous devrions pouvoir facilement traiter les cas qui unissent ces deux approches. Mais pour bien voir la façon de mémoriser l'ordre d'exécution des modules, nous allons traiter plusieurs exemples. Tout d'abord, un exemple qui inclus le traitement parallèle dans le traitement série. Ensuite, un deuxième exemple qui inclus le traitement série dans le traitement parallèle. Enfin, un dernier exemple qui représente un cas d'enchaînement de modules plus général puisqu'il regroupe ensemble les deux cas précédents.

3.4.3.1 Traitement parallèle inclus dans le traitement sériel

Nous allons étudier l'exemple présenté à la figure 3.26 suivante :

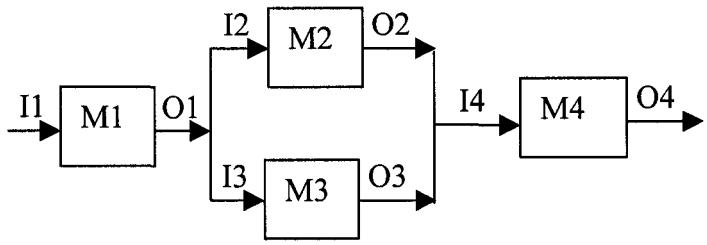


Figure 3.26 – Exemple de parallélisme inclus dans le traitement sériel.

Nous avons M1, M2, M3, M4 qui représentent un module complexe, avec M1(I1) = O1, M2(I2) = O2, M3(I3) = O3, et M4(I4) = O4.

D'après nos hypothèses restrictives de départ, nous avons I2 = I3 = O1, I4 = O2O3. De plus, si nous considérons M' comme le module résultant de M2 et M3 en parallèle, nous aurons M1, M', M4 en série, et cet exemple est donc bien un cas où le traitement parallèle est inséré dans un traitement série.

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I1) = O4, avec I1 les entrées de Mr et O4 ses sorties.

Pour exécuter Mr, il faudra exécuter M1, M', M4 en série.

À travers cet exemple, nous allons en profiter pour montrer l'équivalence des résultats obtenus que M' soit considéré comme le module complexe résultant de M2 et M3 en parallélisme indépendant, parallélisme à entrées dépendantes, parallélisme à sorties dépendantes, parallélisme dépendant, ou parallélisme avec récupération :

- Soit M' le module complexe résultant de M2 et M3 en parallélisme indépendant. D'après la partie 3.4.2.1.1, nous avons :

  M'(I2I3) = O2O3 = (M2I2)(M3I3)

Or I2 = I3 = O1 et O2O3 = I4, alors :

M'(I2I3) = M'(O1O1) = O2O3 = I4 = (M2O1)(M3O1)

Nous avons donc :

Mr(I1) = O4 = M4(I4) = M4(M2O1M3O1)

D'après la définition de Φ, nous déduisons :

Mr(I1) = Φ M4 M2 M3 O1 = Φ M4 M2 M3 (M1I1)

Avec la définition de B, cela donne :

B (Φ M4 M2 M3) M1 I1

Nous mémorisons donc que l'exécutable de Mr sera :

(B (Φ M4 M2 M3) M1 I1)

- Soit M' le module complexe résultant de M2 et M3 en parallélisme à entrées dépendantes. D'après la partie 3.4.2.2.1, nous avons :

M'(O1) = O2O3 = (M2I2)(M3I3)

Or I2 = I3 = O1 et O2O3 = I4, alors :

M'(O1) = O2O3 = I4 = (M2O1)(M3O1)

Nous avons donc :

Mr(I1) = O4 = M4(I4) = M4(M2O1M3O1)

D'après la définition de Φ, nous déduisons :

Mr(I1) = Φ M4 M2 M3 O1 = Φ M4 M2 M3 (M1I1)

Avec la définition de B, cela donne :

B (Φ M4 M2 M3) M1 I1

Nous mémorisons donc que l'exécutable de Mr sera :

(B (Φ M4 M2 M3) M1 I1)

- Soit M' le module complexe résultant de M2 et M3 en parallélisme à sorties dépendantes. D'après la partie 3.4.2.3.1, nous avons :

M'(I2I3) = I4

Or I4 = O2O3 et I2 = I3 = O1, alors :

I4 = O2O3 = (M2I2)(M3I3) = (M2O1)(M3O1).

Nous avons donc :

Mr(I1) = O4 = M4(I4) = M4(M2O1M3O1)

D'après la définition de Φ, nous déduisons :

Mr(I1) = Φ M4 M2 M3 O1 = Φ M4 M2 M3 (M1I1)

Avec la définition de B, cela donne :

B (Φ M4 M2 M3) M1 I1

Nous mémorisons donc que l'exécutable de Mr sera :

(B (Φ M4 M2 M3) M1 I1)

- Soit M' le module complexe résultant de M2 et M3 en parallèle avec récupération dans M4. D'après la partie 3.4.2.5.1, nous avons :

  O4 = Φ M4 M2 M3 O1

  Nous avons donc :

  Mr(I1) = Φ M4 M2 M3 O1 = Φ M4 M2 M3 (M1I1)

  Avec la définition de B, cela donne :

  B (Φ M4 M2 M3) M1 I1

  Nous mémorisons donc que l'exécutable de Mr sera :

  (B (Φ M4 M2 M3) M1 I1)

Dans tous les cas, nous obtenons B (Φ M4 M2 M3) M1 I1 qui modélise bien l'enchaînement de Mr. Effectivement, si nous développons ce résultat, nous allons exécuter les modules qui composent Mr dans l'ordre attendu pour obtenir O4 :

B (Φ M4 M2 M3) M1 I1 = Φ M4 M2 M3 (M1 I1)	(application de B)
= Φ M4 M2 M3 O1	(M1I1=O1)
= M4 (M2O1) (M3O1)	(application de Φ)
= M4 (M2I2)(M3I3)	(O1=I2=I3)
= M4 O2 O3	(M2I2=O2, M3I3=O3)
= M4 I4	(O2O3=I4)
= O4	(M4I4=O4)

Nous pouvons faire deux constatations. Premièrement, notre résultat, B (Φ M4 M2 M3) M1 I1, est le même quelque soit le parallélisme considéré. Deuxièmement, ce résultat exprime bien le parallélisme inclus dans le sériel puisque B signifie que nous effectuons M1 suivi de Φ M4 M2 M3, qui est M2 et M3 en parallèle suivi de M4.

Étant donné ces résultats, nous voyons que le parallélisme considéré importe peu pour résoudre nos enchaînements de modules. Pour éviter de nous répéter en étudiant les différentes possibilités du parallélisme, nous nous limiterons désormais au parallélisme avec récupération. Passons alors maintenant au cas d'un traitement sériel inclus dans un traitement parallèle.

3.4.3.2 Traitement sériel inclus dans le traitement parallèle

Pour illustrer ce cas où le traitement en série est inclus dans un traitement en parallèle, nous allons étudier l'exemple présenté à la figure 3.27 suivante :

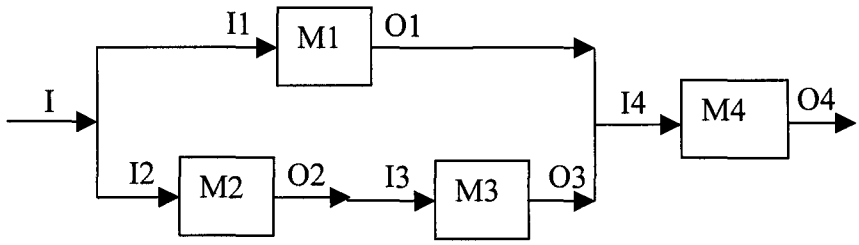


Figure 3.27 – Exemple de traitement sériel inclus dans le parallélisme.

Nous avons M1, M2, M3, M4 qui représentent un module complexe, avec M1(I1) = O1, M2(I2) = O2, M3(I3) = O3, et M4(I4) = O4.

D'après nos hypothèses restrictives de départ, nous avons I1 = I2 = I, I4 = O1O3, et I3 = O2. De plus, si nous considérons M' comme le module résultant de M2 et M3 en série, nous aurons M1 et M' en parallèle avec récupération dans M4, et cet exemple est donc bien un cas où le traitement sériel est inséré dans un traitement parallèle.

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I) = O4, avec I les entrées de Mr et O4 ses sorties.

Soit M' le module complexe résultant de M2 et M3 en série. D'après la partie 3.4.1.1, et I2 = I, nous avons :

M'(I2) = O3 = B M3 M2 I2 = B M3 M2 I

Pour exécuter Mr, il faudra exécuter M1 et M' en parallèle. D'après la partie 3.4.2.5.1, nous aurons alors :

Mr(I) = O4 = Φ M4 M1 M' I

Nous mémorisons donc que l'exécutable de Mr sera :

(Φ M4 M1 (B M3 M2) I)

Nous obtenons donc Φ M4 M1 (B M3 M2) I qui modélise bien l'enchaînement de Mr. Effectivement, si nous développons ce résultat, nous allons exécuter les modules qui composent Mr dans l'ordre attendu pour obtenir O4 :

Φ M4 M1 (B M3 M2) I	= M4 (M1I) (BM3M2I)	(application de Φ)
	= M4 (M1I1) (BM3M2I2)	(I=I1=I2)
	= M4 O1 (BM3M2I2)	(M1I1=O1)
	= M4 O1 (M3(M2I2))	(application de B)
	= M4 O1 (M3I3)	(M2I2=O2=I3)
	= M4 O1O3	(M3I3=O3)
	= M4 I4	(O1O3=I4)
	= O4	(M4I4=O4)

Nous pouvons alors constater que notre résultat, Φ M4 M1 (B M3 M2) I, exprime bien le traitement série inclus dans le traitement parallèle puisque Φ signifie que nous effectuons en parallèle M1 et B M3 M2, qui est M2 et M3 en série.

Notre théorie s'utilise bien jusqu'à présent, étudions alors un exemple plus général qui va mélanger plusieurs possibilités d'enchaînement des modules.

3.4.3.3 Cas général regroupant les approches précédentes

Pour illustrer les cas généraux que sont le traitement en série et en parallèle, nous allons étudier une chaîne de traitement composée de 12 modules simples, à travers l'exemple de la figure 3.28 suivante :

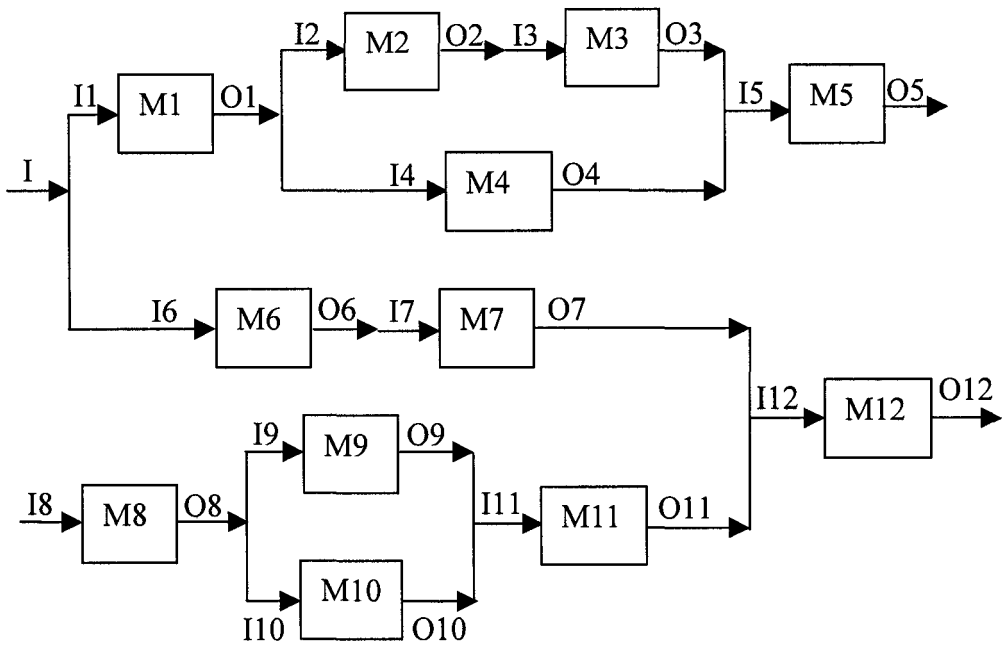


Figure 3.28 – Combinaison série-parallèle de 12 modules.

Nous avons M1, M2, M3, M4, M5, M6, M7, M8, M9, M10, M11, M12 qui représentent un module complexe, avec M1(I1) = O1, M2(I2) = O2, M3(I3) = O3, M4(I4) = O4, M5(I5) = O5, M6(I6) = O6, M7(I7) = O7, M8(I8) = O8, M9(I9) = O9, M10(I10) = O10, M11(I11) = O11, M12(I12) = O12.

D'après nos hypothèses restrictives de départ, nous avons I = I1 = I6, O1 = I2 = I4, O2 = I3, O3O4 = I5, O6 = I7, O8 = I9 = I10, O9O10 = I11, et O7O11 = I12.

Nous commençons par rechercher des cas plus simples de quelques modules pour construire des modules complexes intermédiaires :

- Soit M23 le module complexe résultant de M2 et M3 en série. D'après la partie 3.4.1.1, et I2 = O1, nous avons :

  M23(I2) = M23(O1) = O3 = B M3 M2 I2 = B M3 M2 O1

- Soit M'1 le module complexe résultant de M1, M23, M4 et M5. D'après la partie 3.4.3.1, et I1 = I, nous avons :

  M'1(I1) = O5 = B (Φ M5 M23 M4) M1 I1

  Étant donné M23 et I1 = I, nous avons :

  M'1(I) = O5 = B (Φ M5 (B M3 M2) M4) M1 I
- Soit M67 le module complexe résultant de M6 et M7 en série. D'après la partie 3.4.1.1, et I6 = I, nous avons :

  M67(I6) = M67(I) = O7 = B M7 M6 I6 = B M7 M6 I
- Soit M'2 le module complexe résultant de M8, M9, M10 et M11. D'après la partie 3.4.3.1, nous avons :

  M'2(I8) = O11 = B (Φ M11 M9 M10) M8 I8
- Soit M'3 le module complexe résultant de M67 et M'2 en parallélisme à sorties dépendantes. D'après la partie 3.4.2.3.1, nous avons :

  M'3(II8) = I12 = (M67I) (M'2I8)

  Étant donné M'2, et M67, nous avons :

  M'3(II8) = (BM7M6I)(B(ΦM11M9M10) M8 I8)
- Soit M'4 le module complexe résultant de M'3 et M12 en série, nous avons :

  M'4(II8) = O12 = M12(M'3(II8))

  Étant donné M'3, nous avons :

  M'4(II8) = M12((BM7M6I)(B(ΦM11M9M10) M8 I8))

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(II8) = O5O12, avec II8 les entrées de Mr et O5O12 ses sorties.

D'après ce qui précède, nous avons O5 = M'1(I) et O12 = M'4(II8). Nous aurons alors :

Mr(II8) = B(ΦM5(BM3M2)M4)M1I M12((BM7M6I)(B(ΦM11M9M10)M8I8))

Nous mémorisons donc que l'exécutable de Mr sera :

(B(ΦM5(BM3M2)M4)M1I) (M12((BM7M6I)(B(ΦM11M9M10)M8I8)))

Ceci modélise bien l'enchaînement des modules de Mr. Effectivement, si nous développons ce résultat, nous allons exécuter les modules qui composent Mr dans l'ordre attendu pour obtenir O5O12 :

Mr(II8) = B(ΦM5(BM3M2)M4)M1I M12((BM7M6I)(B(ΦM11M9M10)M8I8))

L'application de B donne :

Mr(II8) = ΦM5(BM3M2)M4(M1I) M12((M7(M6I))(ΦM11M9M10(M8I8)))

Avec I=I1=I6, M1I1=O1, M6I6=O6, M8I8=O8, nous avons :

Mr(II8) = ΦM5(BM3M2)M4O1 M12((M7O6)(ΦM11M9M10O8))

L'application de Φ donne :

Mr(II8) = M5((BM3M2O1)(M4O1)) M12((M7O6)(M11(M9O8)(M10O8)))

Avec O1=I4=I2, M4I4=O4, O6=I7, M7I7=O7, O8=I9=I10, M9I9=O9, M10I10=O10, nous avons :

Mr(II8) = M5((BM3M2I2)O4) M12(O7(M11(O9O10)))

L'application de B donne :

Mr(II8) = M5((M3(M2I2))O4) M12(O7(M11(O9O10)))

Avec M2I2=O2, O9O10=I11, nous avons :

Mr(II8) = M5((M3O2)O4) M12(O7(M11I11))

Avec O2=I3, M3I3=O3, M11I11=O11, nous avons :

Mr(II8) = M5(O3O4) M12(O7O11)

Avec O3O4=I5, M5I5=O5, O7O11=I12, M12I12=O12, nous avons :

Mr(II8) = O5O12

C'est le résultat que nous voulions, et nous pouvons alors constater que notre théorie s'applique bien.

Dans la prochaine partie, nous allons aborder les agencements de modules qui correspondent à des cas particuliers.

3.5 Cas particuliers d'enchaînement de plusieurs modules

Dans cette partie, nous allons voir différents agencements particuliers de modules. Nous traiterons tout d'abord le cas des branches d'injection, ensuite celui des branches d'éjection, et enfin d'autres cas particuliers.

3.5.1 Branches d'injection

Nous parlons de branches d'injection lorsque des liens en entrée de module sont ajoutés directement du départ. Mais pour mieux saisir la notion de branches d'injection, voyons les figures 3.29, 3.30, et 3.31 suivantes :

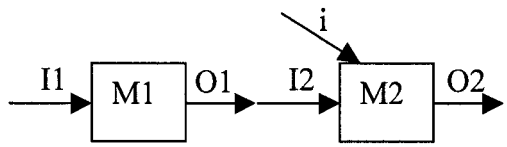


Figure 3.29 – Branche d'injection i dans un traitement série.

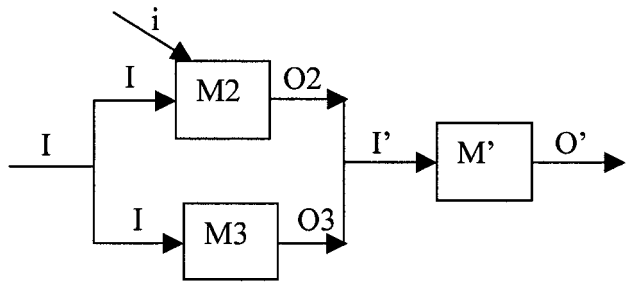


Figure 3.30 – Branche d'injection i dans un traitement parallèle.

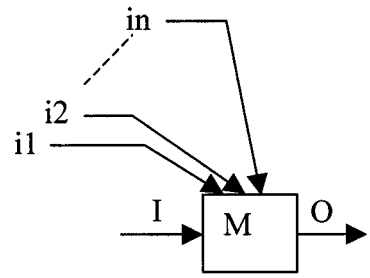


Figure 3.31 – N branches d'injection i1, ..., in sur un module M.

Nous pouvons constater assez facilement que, dans un cas en série, en parallèle ou mixte, le traitement des branches d'injection sur un module M sera toujours le même.

En effet, lors de la construction d'une chaîne de traitement, il faut s'assurer que l'ensemble des liens en entrée du module satisfait, en cardinalité et en types compatibles, l'ensemble des entrées de M. Dans l'affirmative, il suffit ensuite, tout simplement, d'ajouter les branches d'injection comme entrées attendues pour le module complexe qui représentera la chaîne de traitement.

Pour voir à quel point le traitement des injections de branche est aisé, nous allons étudier l'exemple présenté à la figure 3.32 suivante :

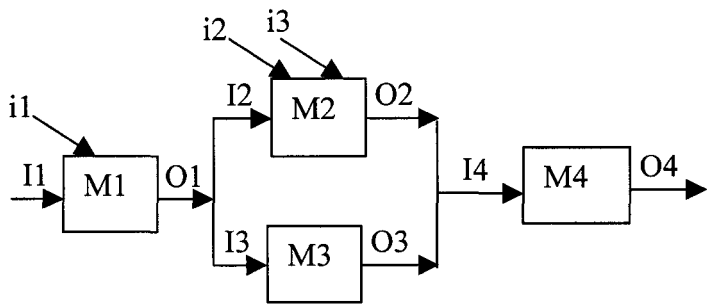


Figure 3.32 – Exemple avec branches d'injection.

Lors de la construction de cette chaîne de traitement, nous nous assurons que I1i1 satisfait l'entrée de M1, et que I2i2i3 satisfait l'entrée de M2.

Nous avons alors M1, M2, M3 et M4, qui représentent un module complexe, avec M1(I1i1) = O1, M2(I2i2i3) = O2, M3(I3) = O3 et M4(I4) = O4.

D'après nos hypothèses restrictives de départ, nous avons O1 = I3 = I2 et O2O3 = I4.

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, en ajoutant les branches d'injection en entrée de Mr, nous avons :

Mr(I1i1i2i3) = O4, avec I1i1i2i3 les entrées de Mr et O4 ses sorties.

D'après ce que nous venons de poser, nous aurons :

Mr(I1i1i2i3) = O4
= M4I4
= M4(O2O3)
= M4(M2(I2i2i3)M3(I3))
= M4(M2(O1i2i3)M3(O1))
= M4(M2((M1(I1i1))i2i3)M3(M1(I1i1)))

Nous mémorisons donc que l'exécutable de Mr sera :

(M4(M2((M1(I1i1))i2i3)M3(M1(I1i1))))

Ceci modélise bien l'enchaînement des modules de Mr. Effectivement, si nous développons ce résultat, nous allons exécuter les modules qui composent Mr dans l'ordre attendu pour obtenir O4.

Nous voyons donc qu'avec les branches d'injection, au moment de la construction d'une chaîne de traitement, il faut s'assurer que chaque module recevant une ou plusieurs

branches a ces dernières qui satisfont les conditions d'entrée du module. Il suffit ensuite d'ajouter les branches d'injection comme entrées attendues du module complexe qui représentera la chaîne de traitement.

Dans la prochaine partie, nous allons traiter les branches d'éjection.

3.5.2 Branches d'éjection

Nous parlons de branches d'éjection lorsque des liens, en sortie de module, sont ajoutés directement sur la fin. Mais pour mieux saisir la notion de branches d'éjection, voyons les figures 3.33, 3.34, et 3.35 suivantes :

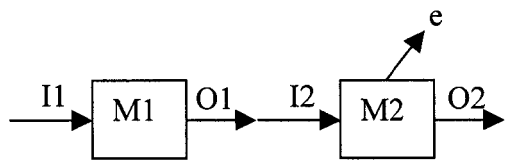


Figure 3.33 – Branche d'éjection e dans un traitement série.

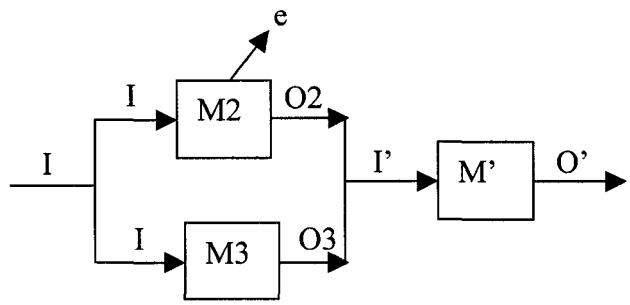


Figure 3.34 – Branche d'éjection e dans un traitement parallèle.

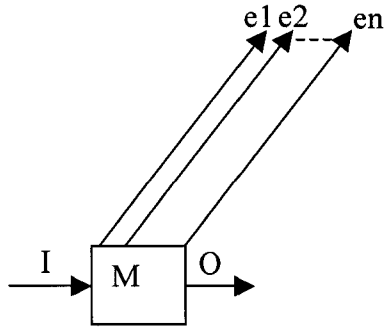


Figure 3.35 – N branches d'éjection e1, ..., en à partir d'un module M.

Nous pouvons constater assez facilement que, dans un cas en série, en parallèle ou mixte, le traitement des branches d'éjection à partir d'un module M sera toujours le même.

En effet, si nous regardons la figure 3.35 par exemple, nos restrictions de départ imposent O = e1 = e2 = ... = en. Il suffit ensuite, tout simplement, d'ajouter les branches d'éjection comme sorties du module complexe qui représentera la chaîne de traitement.

Pour voir à quel point le traitement des éjections de branche est aisé, nous allons étudier l'exemple présenté à la figure 3.36 suivante :

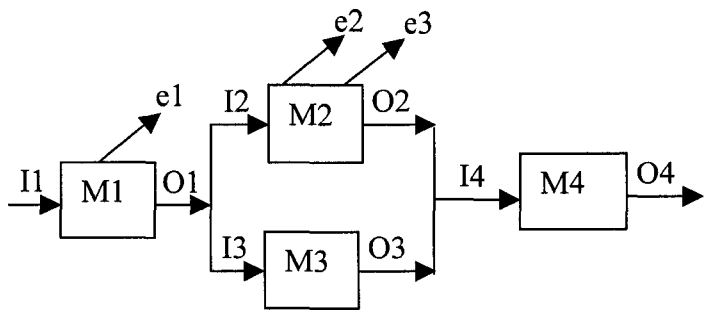


Figure 3.36 – Exemple avec branches d'éjection.

Nous avons alors M1, M2, M3 et M4, qui représentent un module complexe, avec M1(I1) = O1, M2(I2) = O2, M3(I3) = O3 et M4(I4) = O4.

D'après nos hypothèses restrictives de départ, nous avons O1 = I3 = I2 et O2O3 = I4. De plus, lors de la construction de cette chaîne, nous imposons O1 = e1 et O2 = e2 = e3.

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, avec les branches d'éjection en sortie de Mr, nous avons :

Mr(I1) = O4e1e2e3 = O4O1O2O2, avec I1 ses entrées et O4O1O2O2 ses sorties.

Nous aurons donc :

Mr(I1) = O4O1O2O2	(e1=O1, e2=e3=O2)
= B(ΦM4M2M3)M1I1 O1O2O2	(d'après 3.4.3.1)
= B(ΦM4M2M3)M1I1 (M1I1)(M2I2)(M2I2)	(O1=M1I1, O2=M2I2)
= B(ΦM4M2M3)M1I1 (M1I1)(M2(M1I1))(M2(M1I1))	(I2=O1=M1I1)
= B(ΦM4M2M3)M1I1 (M1I1)(BM2M1I1)(BM2M1I1)	(d'après B)

Nous mémorisons donc que l'exécutable de Mr sera :

(B(ΦM4M2M3)M1I1) (M1I1) (BM2M1I1) (BM2M1I1)

Ceci modélise bien l'enchaînement des modules de Mr. Effectivement, si nous développons ce résultat, nous allons exécuter les modules qui composent Mr dans l'ordre attendu pour obtenir O4O1O2O2.

Nous voyons donc qu'avec les branches d'éjection, il suffit de les ajouter comme sorties attendues du module complexe qui représentera la chaîne de traitement.

Dans la prochaine partie, nous allons traiter d'autres cas particuliers.

3.5.3 Autres cas particuliers

Dans cette partie, nous allons étudier des cas spéciaux pour voir si notre théorie s'applique encore.

3.5.3.1 Parallélisme spécial

Certaines formes de parallélisme peuvent être spéciales comme dans l'exemple de la figure 3.37 suivante :

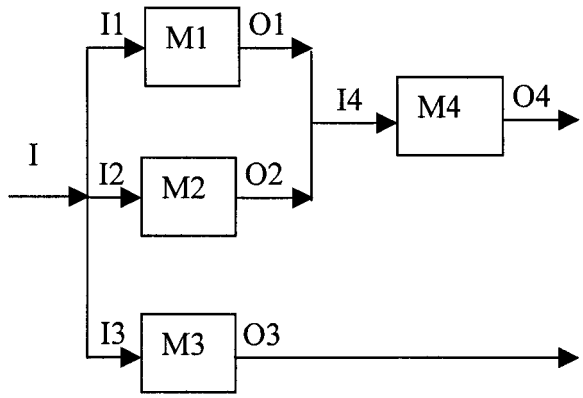


Figure 3.37 – Exemple de parallélisme spécial.

Nous avons M1, M2, M3, M4, qui représentent un module complexe, avec M1(I1) = O1, M2(I2) = O2, M3(I3) = O3, M4(I4) = O4.

D'après nos hypothèses restrictives de départ, nous avons I = I1 = I2 = I3, O1O2 = I4.

Soit M'1 le module complexe résultant de M1, M2, M4. D'après la partie 3.4.2.5.1, nous avons :

M'1(I) = O4 = Φ M4 M1 M2 I

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I) = O4O3, avec I les entrées de Mr et O4O3 ses sorties.

Pour exécuter Mr, il faut exécuter M'1 et M3 en parallélisme à entrées dépendantes. D'après la partie 3.4.2.2.1, nous avons alors :

Mr(I) = O4O3 = (M'1I)(M3I)

Avec la définition de M'1, nous obtenons :

Mr(I) = (ΦM4M1M2I)(M3I)

Nous mémorisons donc que l'exécutable de Mr sera :

(ΦM4M1M2I) (M3I)

Ceci modélise bien l'enchaînement des modules de Mr. Effectivement, si nous développons ce résultat, nous allons exécuter les modules qui composent Mr dans l'ordre attendu pour obtenir O4O3.

C'est le résultat que nous voulions, et nous pouvons alors constater que notre théorie de la partie 3.4.2 sur les différentes formes de parallélismes, d'une part s'applique bien, et d'autre part permet d'enlever l'aspect spécial du parallélisme de cet exemple. Du coup, les formes de parallélismes qui peuvent sembler originales à première vue peuvent toutes se ramener à nos cas de base du traitement en parallèle.

Nous allons passer à un autre cas particulier qui est le parallélisme imbriqué.

3.5.3.2 Parallélisme imbriqué

Un exemple de parallélisme imbriqué est présenté à la figure 3.38 suivante :

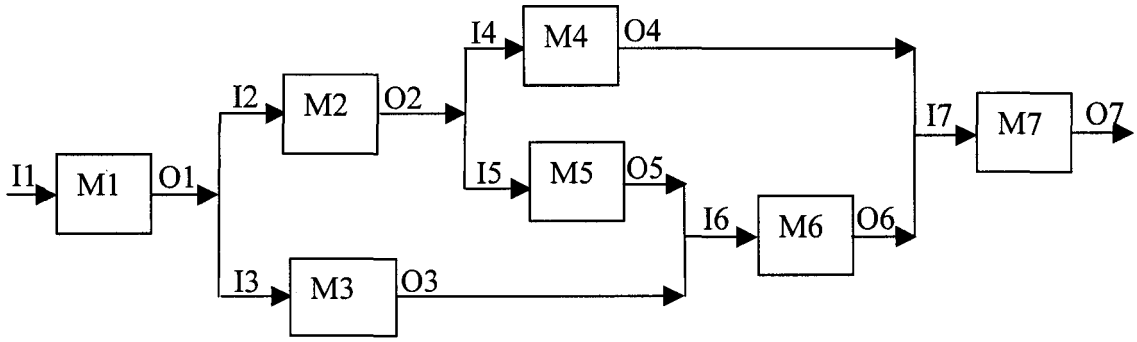


Figure 3.38 – Exemple de parallélisme imbriqué.

Nous avons M1, M2, M3, M4, M5, M6, M7, qui représentent un module complexe, avec M1(I1) = O1, M2(I2) = O2, M3(I3) = O3, M4(I4) = O4, M5(I5) = O5, M6(I6) = O6, M7(I7) = O7.

D'après nos hypothèses restrictives de départ, nous avons O1 = I2 = I3, O2 = I4 = I5, O5O3 = I6, O4O6 = I7.

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I1) = O7, avec I1 les entrées de Mr et O7 ses sorties.

En développant Mr, nous obtenons :

Mr(I1) = M7I7
= M7(O4O6)
= M7((M4I4) (M6I6))
= M7((M4O2) (M6(O5O3)))

= M7((M4(M2I2)) (M6((M5I5) (M3I3))))
= M7((M4(M2O1)) (M6((M5O2) (M3O1))))
= M7((M4(M2(M1I1))) (M6((M5(M2I2)) (M3(M1I1)))))
= M7((M4(M2(M1I1))) (M6((M5(M2O1)) (M3(M1I1)))))
= M7((M4(M2(M1I1))) (M6((M5(M2(M1I1))) (M3(M1I1)))))

Nous ne pouvons plus développer davantage. Nous regardons alors si des schémas des combinateurs S, B ou Φ sont présents dans Mr. Nous trouvons des schémas de B :

- M4(M2(M1I1)) = BM4(BM2M1)I1, d'après la partie 3.4.1.2
- M5(M2(M1I1)) = BM5(BM2M1)I1, d'après la partie 3.4.1.2
- M3(M1I1) = BM3M1I1, d'après la partie 3.4.1.1

Nous en déduisons alors :

Mr(I1) = M7((BM4(BM2M1)I1)(M6((BM5(BM2M1)I1)(BM3M1I1))))

Nous mémorisons donc que l'exécutable de Mr sera :

(M7((BM4(BM2M1)I1)(M6((BM5(BM2M1)I1)(BM3M1I1)))))

Ceci modélise bien l'enchaînement des modules de Mr. Effectivement, si nous développons ce résultat, nous allons exécuter les modules qui composent Mr dans l'ordre attendu pour obtenir O7.

Nous pouvons alors constater que le traitement d'un cas spécial, dans lequel nous ne reconnaissons pas d'enchaînement général de la partie 3.4, se fait tout simplement en développant les sorties du module complexe final qui représentera la chaîne de traitement décrite. Ce développement se fait en utilisant les définitions des modules et les liens imposés par nos hypothèses de départ. Une fois le développement terminé, nous regardons si des schémas de B, S ou Φ apparaissent. Dans l'affirmative, nous faisons les remplacements nécessaires, et c'est terminé.

Nous allons passer à un autre cas particulier qui est le sériel imbriqué.

3.5.3.3 Sériel imbriqué

Un exemple de traitement série imbriqué est présenté à la figure 3.39 suivante :

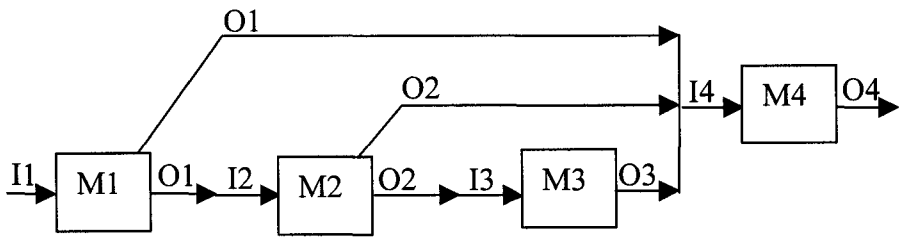


Figure 3.39 – Exemple de sériel imbriqué.

Nous avons M1, M2, M3, M4, qui représentent un module complexe, avec M1(I1) = O1, M2(I2) = O2, M3(I3) = O3, M4(I4) = O4. Nous voyons que M1, M2, M3, M4 sont en série et qu'une éjection de O1 et O2 sur M4 fait en sorte que plusieurs séries semblent imbriquées les unes dans les autres : la série M1M4, la série M1M2M4, et la série M1M2M3M4.

D'après nos hypothèses restrictives de départ, sur la cardinalité et la compatibilité de type des entrées, nous avons O1 = I2, O2 = I3, et O1O2O3 = I4.

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(I1) = O4, avec I1 les entrées de Mr et O4 ses sorties.

En développant Mr, nous obtenons :

Mr(I1) = M4I4
= M4(O1O2O3)
= M4((M1I1) (M2I2) (M3I3))
= M4((M1I1) (M2(M1I1)) (M3(M2(M1I1))))

Nous ne pouvons plus développer davantage. Nous regardons alors si des schémas des combinateurs S, B ou Φ sont présents dans Mr. Nous trouvons des schémas de B :

- M2(M1I1) = BM2M1I1, d'après la partie 3.4.1.1
- M3(M2(M1I1)) = BM3(BM2M1)I1, d'après la partie 3.4.1.2

Nous en déduisons alors :

Mr(I1) = M4((M1I1) (BM2M1I1) (BM3(BM2M1)I1))

Nous mémorisons donc que l'exécutable de Mr sera :

(M4((M1I1) (BM2M1I1) (BM3(BM2M1)I1)))

Ceci modélise bien l'enchaînement des modules de Mr. Effectivement, si nous développons ce résultat, nous allons exécuter les modules qui composent Mr dans l'ordre attendu pour obtenir O4.

Nous pouvons alors faire le même constat qu'avec le parallélisme imbriqué : le traitement d'un cas spécial, dans lequel nous ne reconnaissons pas d'enchaînement général de la partie 3.4, se fait tout simplement en développant les sorties du module complexe final qui représentera la chaîne de traitement décrite. Ce développement se fait en utilisant les définitions des modules et les liens imposés par nos hypothèses de départ. Une fois le développement terminé, nous regardons si des schémas de B, S ou Φ apparaissent. Dans l'affirmative, nous faisons les remplacements nécessaires, et c'est terminé.

Nous allons maintenant voir le cas des modules non commutatifs.

3.5.3.4 Modules non commutatifs

Voyons l'exemple de la figure 3.40 suivante :

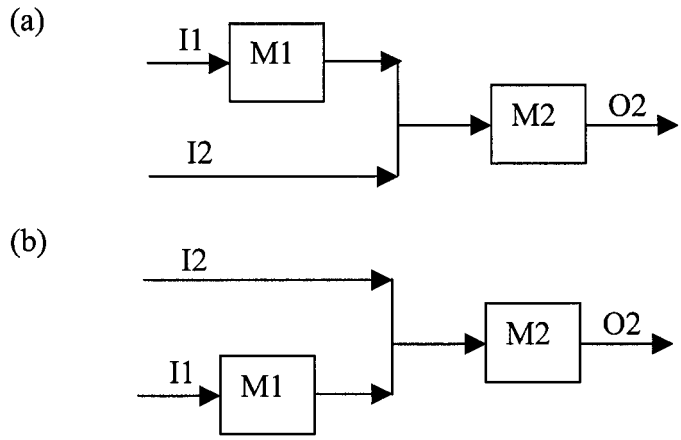


Figure 3.40 – Module complexe avec comme entrées : (a) I1 et I2, (b) I2 et I1.

La seule différence entre (a) et (b) est l'ordre des entrées I1 et I2. En (a), nous aurons O2=M2((M1I1)I2), et en (b), nous aurons O2=M2(I2(M1I1)).

Soient i1 et i2 deux ensembles en entrée, M2 sera un module non commutatif si M2(i1i2) est différent de M2(i2i1), et sera un module commutatif dans le cas contraire. Par exemple, M2 sera commutatif s'il est un module d'addition de deux nombres, et il sera non commutatif s'il est un module de soustraction entre deux nombres.

Ainsi, nous devons faire attention à l'ordre des connexions sur les modules non commutatifs. Nous devrions donc avoir deux représentations différentes pour distinguer (a) de (b), ce qui est bien le cas avec notre façon de coder O2. La nature commutative

d'un module n'étant pas toujours facile à déterminer, nous considèrerons toujours différentes les représentations (a) et (b), et devrons donc faire attention à l'ordre des entrées qui arrivent sur un module.

Nous remarquons que la théorie développée dans ce chapitre tient bien la route. Pour mieux s'en convaincre, la prochaine partie va faire l'objet d'une étude de cas générale.

3.6 Étude de cas

Notre étude de cas va consister à étudier un exemple général regroupant un grand nombre de cas particuliers étudiés précédemment. De plus, nous utiliserons un module complexe au sein de la chaîne de traitement, pour mieux comprendre l'utilisation de ces derniers. À cette fin, nous avons retenu le module complexe Mc de la partie 3.5.3.1, représenté à la figure 3.41 suivante :

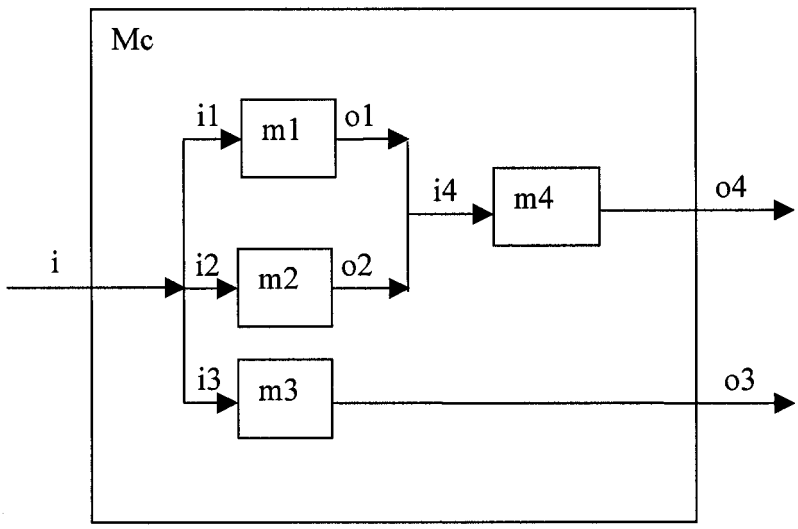


Figure 3.41 – Représentation du module complexe Mc de l'étude de cas.

Et voici notre exemple général présenté à la figure 3.42 suivante :

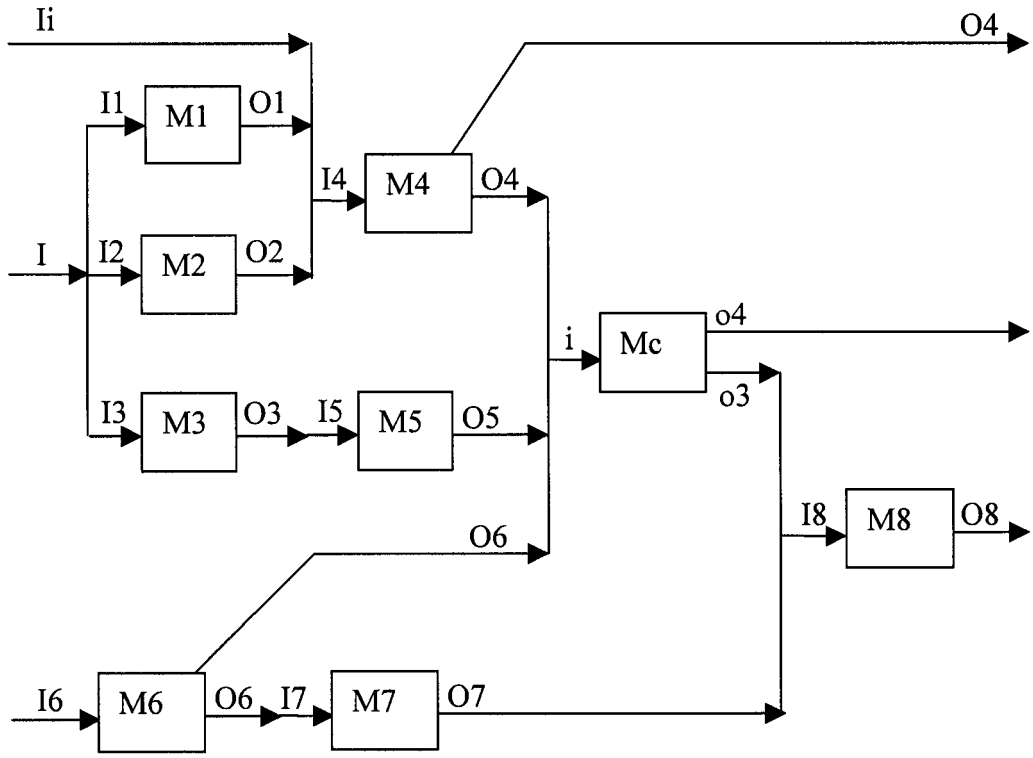


Figure 3.42 – Exemple général d'une chaîne de traitement..

Dans cet exemple, nous retrouvons une branche d'injection Ii sur M4. Il y a aussi deux branches d'éjection : O4 à partir de M4 et o4 à partir du module complexe Mc.

Nous avons M1, M2, M3, M4, M5, Mc, M6, M7, M8 qui représentent un module complexe, avec M1(I1) = O1, M2(I2) = O2, M3(I3) = O3, M4(I4) = O4, M5(I5) = O5, Mc(i) = o4o3, M6(I6) = O6, M7(I7) = O7, M8(I8) = O8.

D'après nos hypothèses restrictives de départ, nous avons I = I1 = I2 = I3, IiO1O2 = I4, O3 = I5, O6 = I7, O4O5O6 = i, o3O7 = I8. De plus, d'après la partie 3.5.3.1, nous avons Mc(i) = o4 o3 = (Φm4m1m2i)(m3i).

Si nous considérons Mr comme le module complexe résultant qui représente la chaîne de traitement précédente, nous avons :

Mr(Ii I I6) = O4 o4 O8, avec Ii I I6 les entrées de Mr et O4 o4 O8 ses sorties.

En développant O4, nous obtenons :

O4	= M4I4	(définition : O4=M4I4)
	= M4(IiO1O2)	(hypothèse : I4=IiO1O2)
	= M4(Ii(M1I1)(M2I2))	(définitions : O1=M1I1 et O2=M2I2)
	= M4(Ii(M1I)(M2I))	(hypothèse : I1=I2=I)

Nous arrêtons alors puisque tout est exprimé en fonction des entrées de Mr. Nous terminerons à chaque fois notre développement pour cette raison.

En développant i, nous obtenons :

i	= O4 O5 O6	(hypothèse : i=O4O5O6)
	= O4 (M5I5) (M6I6)	(définitions de O5 et O6)
	= O4 (M5O3) (M6I6)	(hypothèses : I5=O3)
	= O4 (M5(M3I3)) (M6I6)	(définitions de O3)
	= O4 (M5(M3I)) (M6I6)	(hypothèse : I3=I)

En remplaçant O4 par ce que nous avons trouvé auparavant, nous avons :

i = (M4(Ii(M1I)(M2I))) (M5(M3I)) (M6I6)

En développant o4, nous obtenons :

o4 = Φ m4 m1 m2 i (définition de o4)

En remplaçant i par ce que nous avons trouvé auparavant, nous avons :

o4 = Φ m4 m1 m2 ((M4(Ii(M1I)(M2I))) (M5(M3I)) (M6I6))

En développant O8, nous obtenons :

O8	= M8I8	(définition de O8)
	= M8(o3O7)	(hypothèse : I8=o3O7)

= M8((m3i) (M7I7)) (définitions de o3 et O7)

= M8((m3i) (M7O6)) (hypothèse : I7=O6)

= M8((m3i) (M7(M6I6))) (définition de O6)

En remplaçant i par ce que nous avons trouvé auparavant, nous avons :

O8 = M8((m3((M4(Ii(M1I)(M2I))) (M5(M3I)) (M6I6))) (M7(M6I6)))

En appliquant ce que nous venons de développer, nous déduisons :

Mr(Ii I I6) = O4 o4 O8

= M4(Ii(M1I)(M2I))

Φm4m1m2((M4(Ii(M1I)(M2I))) (M5(M3I)) (M6I6))

M8((m3((M4(Ii(M1I)(M2I))) (M5(M3I)) (M6I6))) (M7(M6I6)))

Nous ne pouvons plus développer davantage. Nous regardons alors si des schémas des combinateurs S, B ou Φ sont présents dans Mr. Nous trouvons les schémas suivants :

- M5(M3I) = BM5M3I, d'après la partie 3.4.1.1
- M7(M6I6) = BM7M6I6, d'après la partie 3.4.1.1

Nous en déduisons alors :

Mr(Ii I I6) = M4(Ii(M1I)(M2I))

Φm4m1m2((M4(Ii(M1I)(M2I))) (BM5M3I) (M6I6))

M8((m3((M4(Ii(M1I)(M2I))) (BM5M3I) (M6I6))) (BM7M6I6))

Nous mémorisons donc que l'exécutable de Mr sera :

(M4(Ii(M1I)(M2I)))

(Φm4m1m2((M4(Ii(M1I)(M2I))) (BM5M3I) (M6I6)))

(M8((m3((M4(Ii(M1I)(M2I))) (BM5M3I) (M6I6))) (BM7M6I6)))

Ceci modélise bien l'enchaînement des modules de Mr. Effectivement, si nous développons ce résultat, nous allons exécuter les modules qui composent Mr dans l'ordre attendu pour obtenir O4 o4 O8.

Cet exemple illustre l'application de la théorie mise en place dans ce chapitre et montre à quel point celle-ci s'utilise facilement. Nous pouvons donc maintenant passer à la conclusion pour récapituler notre démarche.

3.7 Conclusion

Dans ce chapitre, l'étude de divers cas d'enchaînement de modules a permis de montrer à quel point il est aisé de mémoriser l'ordre d'exécution d'un module complexe en utilisant la théorie des combinateurs. Les module simples utilisent le combinateur I, et les chaînes de traitements font appels aux combinateurs B, S et Φ, pour être modélisées. B est utilisé pour modéliser des enchaînements en série, tandis que S et Φ servent pour les enchaînements en parallèle.

Comme nous l'avons vu, l'étape de modélisation se déroule de la façon suivante :

1) Développement des sorties du module complexe final qui représentera la chaîne de traitement décrite. Ce développement se fait en utilisant les définitions des modules et les liens imposés par nos hypothèses de départ, sur les ensembles d'entrée et de sortie.
2) Une fois le développement terminé, nous regardons si des schémas de B, S ou Φ, apparaissent. Dans l'affirmative, nous faisons les remplacements nécessaires, et c'est terminé.

La théorie développée s'avère efficace au regard des exemples abordés puisqu'elle nous a permis de modéliser toutes les chaînes présentées. Pour mieux s'en convaincre, voyons maintenant le prochain chapitre qui va présenter l'implémentation de cette théorie et les résultats obtenus.

CHAPITRE 4

IMPLÉMENTATION

4.1 Introduction

Nous voici maintenant à l'étape de l'implémentation de la théorie développée au chapitre précédent. L'objectif est de pouvoir exécuter les modules simples d'une chaîne de traitement dans l'ordre spécifié par cette dernière. À titre de rappel, nous pouvons dire que nous avons déterminé deux types de modules, soient les modules simples et les modules complexes. Les modules complexes sont un agencement de modules simples dont l'ordre d'exécution est spécifié par leur chaîne de traitement. Et nous avons vu qu'il était possible de modéliser ces chaînes de traitement en utilisant la théorie des combinateurs, notamment en utilisant le combinateur I pour les modules simples, et les combinateurs B, Φ et S pour les modules complexes. L'exécution de telles chaînes de traitement dépendra donc de leur modélisation.

Nous débuterons ce chapitre par des considérations générales d'analyse du problème. Ensuite, nous expliquerons comment a été réalisée l'implémentation, notamment à travers l'utilisation d'un arbre d'exécution. Puis, nous traiterons un exemple qui montre en détail comment est construit l'arbre d'exécution. Enfin, nous verrons l'ensemble des tests effectués, ce qui nous permettra de conclure.

4.2 Considérations générales

D'après ce qui a été vu, un module simple possède un seul ensemble en entrée et un seul en sortie. Un tel module peut être exécuté et possède donc un exécutable. Nous convenons que les modules simples s'exécuteront par l'intermédiaire de fichiers textes.

C'est-à-dire qu'un fichier texte contiendra l'ensemble des entrées nécessaires au module, et qu'un autre fichier texte récupèrera l'ensemble des sorties produites par l'exécution. Ainsi, pour lancer l'exécution d'un module simple, il faudra spécifier son exécutable, mais aussi son fichier texte des entrées et celui des sorties.

Nous savons aussi qu'un module complexe peut avoir plusieurs ensembles d'entrée ou de sortie, ce qui est lié au fait qu'il est l'enchaînement de plusieurs modules simples. En effet, lors de la construction d'un module complexe, nous connectons ensemble plusieurs modules simples et nous déduisons la chaîne de traitement qui va permettre de savoir quelles sont ses entrées et ses sorties, quels sont les modules simples qui le composent et dans quel ordre ils sont reliés. Ainsi, l'exécution de tels modules se fera en exécutant les modules simples de leurs chaînes de traitement dans l'ordre spécifié par ces dernières. Nous fonctionnerons toujours par l'intermédiaire de fichiers textes. C'est-à-dire qu'un fichier texte servira à récupérer les ensembles des sorties produites par son exécution.

Les chaînes de traitement sont donc un agencement particulier de symboles. La gestion de ces chaînes nécessite de définir les symboles possibles. Nous avons convenu des symboles suivants :

- Pour les modules simples nous avons utilisé la lettre 'I' comme code d'exécution. Il représente le combinateur d'identité I et signifie que le module en question est simple.
- Pour les modules complexes, la chaîne de traitement est constituée de la lettre 'E' avec un numéro pour stipuler chacune des différentes entrées, de la lettre 'M' avec un numéro pour stipuler chacun des modules simples impliqués, des lettres 'B' pour le combinateur de composition B, 'P' pour le combinateur de coordination Φ et 'S' pour le combinateur de substitution S, et de parenthèses ouvrantes et fermantes pour stipuler comment s'appliquent les modules et les combinateurs et comment sont organisées les sorties.

Notre objectif se limite ici à l'exécution des modules, ceux-ci une fois construits. Nous considérons donc, comme hypothèse de départ, que la chaîne de traitement associée à un module complexe est valide. Une chaîne sera dite valide si :

- Il n'y a pas d'espaces.
- Tout est en majuscules.
- Il y a seulement les symboles cités ci-dessus.
- Les parenthèses sont équilibrées : égalité entre le nombre de fermantes et d'ouvrantes.
- Chaque sortie est entre parenthèses.
- Tous les symboles sont placés correctement pour pouvoir être appliqués.
- Les modules impliqués existent sous forme d'exécutable.

Cette notation nous permet de connaître les modules à exécuter avec les différents M, le nombre d'entrées avec les différents E, et le nombre de sorties avec les différents blocs disjoints de parenthèses. Voici quelques exemples :

- (M1E1)(M2E2) signifie qu'il y a deux sorties M1E1 et M2E2, qu'il y a deux entrées E1 et E2, et qu'il faudra exécuter M1 et M2.
- (M2(E1(BM1M3E1))) signifie qu'il y a une sortie M2(E1(BM1M3E1)), qu'il y a une entrée E1, et qu'il faudra exécuter M1, M2 et M3. Les combinateurs et les parenthèses restantes permettent de préciser l'ordre d'exécution des modules.
- (M2(E2(M3E1)))(M2E3)(M3E4) signifie qu'il y a trois sorties M2(E2(M3E1)), M2E3 et M3E4, qu'il y a quatre entrées E1, E2, E3 et E4, et qu'il faudra exécuter M2 et M3. Les entrées sont ordonnées suivant leur numérotation dans le fichier texte des entrées, soit avec E1 en premier, E2 en deuxième, etc.

Après l'étape d'exécution, si un module complexe possède le comportement recherché et que nous sommes intéressés à l'utiliser pour construire d'autres chaînes de traitement, une étape de sauvegarde sera nécessaire. La sauvegarde d'un module complexe devra permettre de déduire un exécutable à partir du code de sa chaîne de traitement. Ainsi, le

module complexe au départ sera devenu un module simple, et pourra alors être utilisé. Cette transformation nécessitera de regrouper les entrées et sorties pour que le module puisse être simple.

Pour revenir au problème de l'exécution des modules, nous devons nous interroger sur ce que sera l'information pertinente à conserver pour chaque module. Nous retiendrons les caractéristiques suivantes pour les modules :

- Un numéro unique d'identification.
- Un nom commun donné au module, plus significatif que le code.
- Une description avec toute l'information nécessaire à la bonne compréhension de ce que fait le module.
- Un domaine et un sous-domaine pour permettre de classer les modules.
- La chaîne de traitement du module.
- L'endroit de l'exécutable pour les modules simples.
- Le nombre d'ensembles en entrée et le nombre d'ensembles en sortie du module.
- Les types des entrées attendues et des sorties fournies.

Tous les modules auront ainsi une représentation uniforme qui nous permettra de les conserver dans une base de données. Au lancement de notre programme nous commencerons alors par l'affichage à l'écran de tous les modules de la base de données. Ensuite, pour chaque module que l'utilisateur voudra exécuter jusqu'à ce qu'il quitte l'application, nous aurons l'algorithme d'exécution suivant :

1) Sélection d'un module par l'utilisateur.
2) Saisie des entrées par l'utilisateur.
3) Exécution de la chaîne de traitement.

Voyons maintenant comment a été réalisée l'implémentation.

4.3 Implémentation

Pour la réalisation de notre programme, nous avons utilisé Visual Studio .NET. Nous avons développé une solution composée de plusieurs éléments, chacun étant un programme en C++ avec son fichier d'en-tête (.h) et son fichier d'implémentation (.cpp).

Nous commencerons par présenter l'implémentation des modules, suivie de celle de la base de données. Nous verrons ensuite le fonctionnement de l'arbre d'exécution qui permet d'exécuter les modules d'une chaîne de traitement dans le bon ordre. Enfin, nous terminerons avec les éléments d'interface.

4.3.1 Les modules

Nos modules sont tous dans une base de données au départ. Nous avons alors défini une structure qui puisse récupérer l'information d'un module contenu dans la base de données : la classe Cmodule. Ainsi, les attributs de cette classe sont les caractéristiques identifiées précédemment pour un module.

Nous constatons que cette classe sert de structure commune pour regrouper l'information d'un module quelconque. Elle permet donc de récupérer l'information des modules contenue dans la base de données et d'y accéder facilement à tout moment dans notre programme. Pour ce faire, nous travaillerons avec un tableau d'objets Cmodule qui contiendra tous les modules présents dans la base de données.

Veuillez vous référer à l'annexe A pour les détails de l'implémentation.

4.3.2 La base de données

Nous avons regroupé tous nos modules dans une base de données Microsoft Access « BDmodules.mdb ». Nous avons utilisé trois tables : « Modules » pour l'information des modules, « Inputs » pour les types en entrées et « Outputs » pour les types en sorties. Ces tables sont présentées avec leurs attributs à la figure 4.1.

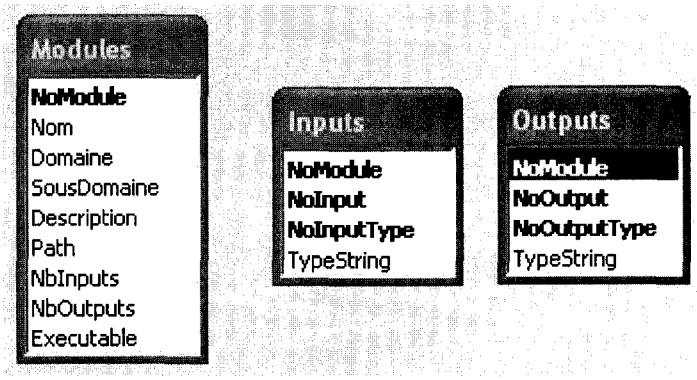


Figure 4.1 – Structure de notre base de donnée.

Toutes les clés NoModule, NoInput, NoOutput, NoInputType et NoOutputType sont des entiers qui commencent à 1. Pour la table Modules, les champs définis sont ceux qui caractérisent un module, à part les types des ensembles d'entrée et de sortie qui ont été mis respectivement dans la table Inputs et la table Outputs. La seule particularité réside dans l'utilisation de ces deux dernières tables. Effectivement, elles possèdent une clé composée qui identifie de manière unique chaque type d'entrée ou de sortie que nous appelons TypeString, puisque les types sont représentés par des chaînes de caractères. NoModule précise le module concerné. NoInput, ou NoOutput, précisent l'ensemble d'entrée, ou de sortie, concerné. NoInputType, ou NoOutputType, précisent la place du type dans l'ensemble d'entrée, ou de sortie. Mais voyons un exemple pour mieux comprendre. Soit l'enregistrement 3 – 1 – 2 – int de la table Inputs, cela signifiera que le module numéro 3 attend un entier comme deuxième élément de son premier ensemble

en entrée. Ainsi, si nous reprenons l'exemple d'un module complexe avec deux entrées {int, int} et {float}, si son numéro de module est 3, nous aurons alors la table Inputs remplie avec les enregistrements suivants :

3 – 1 – 1 – int

3 – 1 – 2 – int

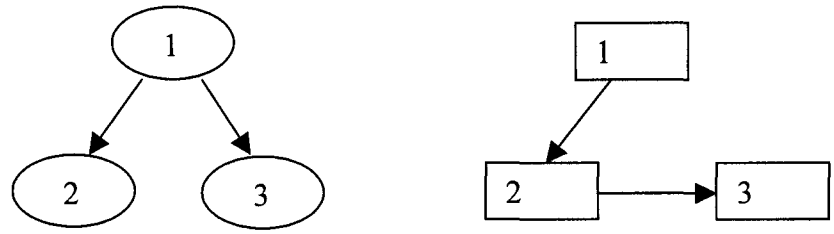
3 – 2 – 1 - float

Dans l'idée de mieux supporter tout développement futur, nous avons défini une classe CBd. En effet, cela nous permet d'avoir plusieurs bases de données Access, chacune pouvant ainsi être spécialisée dans un domaine ou attitrée à une personne, par exemples. Dans le cadre de notre application, nous n'avons cependant eu besoin d'utiliser qu'une seule base de données. De plus, la classe possède une structure pour les tables et un ensemble de fonctions qui permettent de facilement prendre en compte tout changement dans la base de données, tels que l'ajout ou la suppression de tables ou la modification du nombre d'attributs d'une table.

Veuillez vous référer à l'annexe A pour les détails de l'implémentation.

4.3.3 L'arbre d'exécution

Nous exécutons un module particulier par l'intermédiaire d'un arbre n-aire d'exécution. Un nœud de l'arbre est défini par un code, un argument, un fichier texte des entrées, un fichier texte des sorties, un nœud parent, un nœud enfant et un nœud frère. Un nœud peut avoir théoriquement plusieurs enfants (cf. figure 4.2.a), ce qui se traduit pratiquement par le fait qu'il aura alors un enfant avec les autres enfants comme frères successifs de ce dernier (cf. figure 4.2.b).



a) représentation théorique b) représentation pratique

Figure 4.2 – Représentation d'un nœud avec deux enfants.

L'arbre est structuré de la façon suivante : un nœud racine au niveau 0, un ou plusieurs nœuds de sortie au niveau 1, des nœuds de module aux autres niveaux et des nœuds d'entrée en guise de feuilles. Avec une telle structure, nous pouvons représenter le module à exécuter.

Au niveau 0, nous retrouvons la racine qui représente le module à exécuter. Ainsi, le code de la racine est celui du module à exécuter et son argument est la chaîne de traitement correspondante. La racine n'a pas de fichier texte des entrées, puisque celles-ci sont dans les feuilles de l'arbre, mais possède un fichier texte qui contiendra toutes les sorties générées lors de l'exécution de ce module. Par exemple, si le module Mx est exécuté, son fichier texte sera « Mx_output.txt ». La racine n'a jamais de nœud parent, ni de nœud frère, elle a seulement un nœud enfant qui est un nœud de sortie.

Au niveau 1, nous retrouvons les nœuds de sortie qui représentent les différentes sorties du module à exécuter. Ainsi, le code d'un nœud de sortie est la lettre S et son argument est la chaîne de traitement correspondante. Un nœud de sortie a un fichier texte des entrées et un autre des sorties. Par exemple, si l'argument de la sortie est ARG, elle aura « S_ARG_input.txt » comme fichier des entrées et « S_ARG_output.txt » comme fichier des sorties. Le fichier texte des entrées servira à récupérer les résultats d'exécution des modules impliqués dans la sortie. Le fichier texte des sorties reste vide jusqu'à la fin de

l'exécution de tous ces modules. Une fois la sortie complètement exécutée, son fichier des entrées est simplement copié dans celui des sorties. Nous aurions pu nous contenter d'avoir un seul fichier texte pour les nœuds de sortie, mais cette façon de faire évite d'exécuter plusieurs fois des sorties qui ont la même représentation. Toutes les sorties ont la racine comme nœud parent. Une sortie est enfant de la racine, et les autres, si elles existent, sont des frères successifs de cette dernière. Ainsi, une sortie a juste d'autres sorties comme frères. Les enfants d'une sortie dépendent de son argument.

Aux autres niveaux, nous retrouvons des nœuds de module et des nœuds d'entrée. En effet, pour simplifier la compréhension et l'utilisation de l'arbre, nous limitons les types de nœuds possibles en appliquant les combinateurs de la chaîne de traitement du module à exécuter dès le début de sa construction. Ainsi, il ne reste que des modules, des entrées et des parenthèses dans la chaîne de traitement. Nous avons donc uniquement des nœuds de module et d'entrée, dont les positions dans l'arbre reflètent l'ordre d'application des modules, stipulé par l'agencement des parenthèses.

Un nœud de module a pour code la lettre M suivie d'un numéro, et son argument est la chaîne de traitement sur laquelle s'applique le module. De plus, son fichier texte des entrées sert à rassembler l'ensemble des entrées nécessaires à l'exécution du module, tandis que son fichier texte des sorties sert à regrouper l'ensemble des sorties produites par cette exécution. Par exemple, si nous avons un module Mi, d'argument ARG, il aura « Mi_ARG_input.txt » comme fichier des entrées et « Mi_ARG_output.txt » comme fichier des sorties. Grâce à cette notation, tout comme pour les sorties, nous savons si un module a déjà été exécuté grâce à son fichier de sortie qui doit alors être rempli. Cela évite aussi d'exécuter plusieurs fois des modules qui ont le même code et le même argument. Pour un nœud de module, son parent peut être une sortie ou un module, son frère peut être un module ou une entrée, et son enfant peut être un module ou une entrée. Un tel nœud représente toujours un module simple.

Un nœud d'entrée a pour code la lettre E suivie d'un numéro, et son argument est la chaîne nulle. De plus, les fichiers textes d'entrée et de sortie sont les mêmes, ce qui revient à n'utiliser qu'un seul fichier texte. Par exemple, si nous avons une entrée Ei, elle aura « Ei.txt » comme fichier texte. Ce dernier sert à contenir les entrées stipulées par l'utilisateur. Pour un nœud d'entrée, son parent peut être un module ou une sortie, et son frère peut être un module ou une entrée. Lors de la construction, un tel nœud est toujours une feuille de l'arbre et n'a donc jamais d'enfants.

L'utilisation d'un tel arbre se fait en trois étapes. Tout d'abord, il faut construire l'arbre à partir des informations disponibles sur le module à exécuter. Ensuite, il faut remplir les fichiers textes des entrées par l'intermédiaire de l'utilisateur. Enfin, il reste à exécuter l'arbre.

L'arbre va se construire en deux temps, à partir de la chaîne de traitement d'un module particulier. Dans un premier temps, nous préparons la chaîne de traitement pour limiter les types de nœuds à manipuler. Ainsi, dans un cas de module simple Mx nous transformons son exécutable 'I' en '(MxE1)', ce qui correspond bien à un module simple puisque nous avons une entrée et une sortie. Dans un cas de module complexe nous appliquons les combinateurs éventuellement présents, c'est-à-dire que les combinateurs sont appliqués de façon à ne laisser que des modules, des entrées et des parenthèses dans la chaîne de traitement. Dans un deuxième temps, nous construisons l'arbre en commençant par l'initialisation de la racine, suivi de l'ajout des sorties, suivi du développement récursif de chaque sortie en arbre de nœuds de module et d'entrée. Il faut néanmoins noter que l'ajout d'un frère se fait par insertion sur la tête de la liste des frères.

Au moment d'exécuter l'arbre, nous avons un arbre déjà construit et les fichiers des entrées remplies. L'exécution s'effectue de bas en haut et de droite à gauche. Nous

partons de la racine pour trouver la feuille la plus à droite. L'algorithme qui permet d'atteindre la feuille est :

- Si le nœud a un frère, aller dessus.
- Sinon si le nœud a un enfant, aller dessus.
- Sinon, nous sommes sur la feuille.

Nous commençons donc par nous positionner sur la feuille la plus à droite de l'arbre. Une fois dessus, nous exécutons la feuille, ensuite nous supprimons le nœud en prenant bien soin de mettre à jour son ex-frère ou son ex-parent, puis nous terminons par nous positionner sur la nouvelle feuille la plus à droite. L'exécution est récursive et nous nous arrêtons quand il ne reste plus que la racine. En cours d'exécution, suite aux suppressions effectuées, une feuille n'est plus forcément un nœud d'entrée. Nous retrouvons donc trois types de nœuds à exécuter : les nœuds d'entrée, de module ou de sortie.

Veuillez vous référer à l'annexe A pour les détails de l'implémentation. De plus, pour mieux comprendre le comportement de l'arbre lors de sa construction, veuillez vous référer à la partie 4.4 qui montre en détail comment l'arbre est construit.

4.3.4 L'interface

Dans l'interface principale, nous retrouvons trois parties distinctes. La première permet d'effectuer toutes les initialisations nécessaires, lors du démarrage de l'application. La deuxième permet de naviguer parmi les modules de la base de données, grâce à des boutons de navigation. La dernière permet d'exécuter le module en cours d'affichage, grâce à un bouton d'exécution. L'interface est présentée à la figure 4.3 suivante.

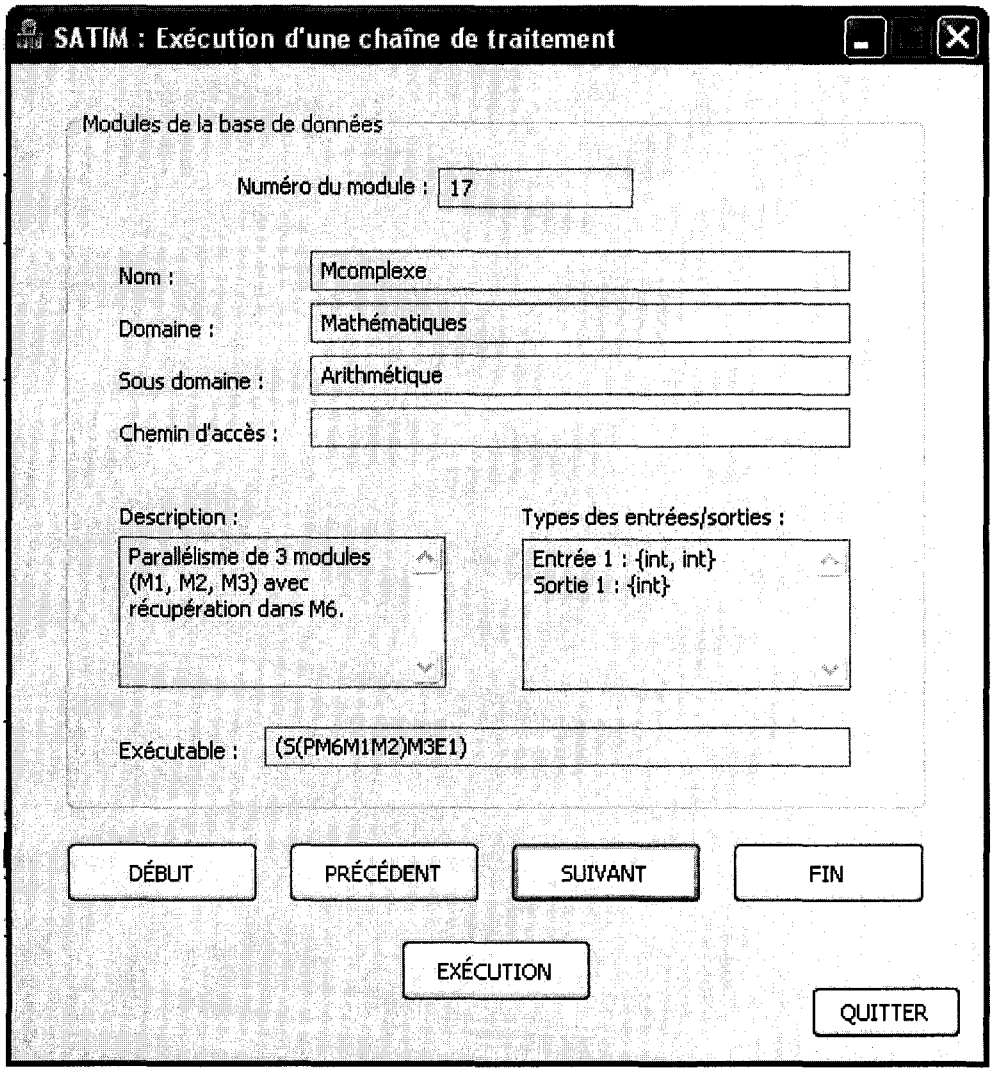


Figure 4.3 – Interface principale de l'application.

À l'initialisation de la fenêtre nous initialisons la base de données, puis nous regroupons tous les modules dans un tableau d'objets Cmodule, puis nous affichons le premier module à l'écran, puis nous initialisons les boutons de navigation. Il faut néanmoins noter que les éléments qui caractérisent la base de données sont spécifiés lors de l'initialisation de cette dernière. Nous y retrouvons le nom, le chemin d'accès, et le nom des tables de la base de données.

Pour naviguer parmi les modules, nous avons défini les boutons :

- DÉBUT : permet de se placer sur le premier module.
- PRÉCÉDENT : permet de se placer sur le module précédent.
- SUIVANT : permet de se placer sur le module suivant.
- FIN : permet de se placer sur le dernier module.

Ainsi, l'utilisateur devra choisir le module qu'il veut exécuter avec les boutons.

L'exécution du module en cours d'affichage se fait avec le bouton EXÉCUTION. Nous commençons par vider le répertoire « Résultats » qui correspond au chemin d'accès du répertoire contenant les fichiers textes qui sont produits lors de l'exécution. Ceci évite d'avoir des conflits de noms (si un fichier texte d'entrée ou de sortie d'une exécution passée porte le même nom qu'un fichier nécessaire à l'exécution en cours) et permet d'avoir des résultats plus clairs à lire. Puis, nous construisons l'arbre d'exécution avec le code et la chaîne de traitement du module sélectionné. Ensuite, nous faisons la saisie des entrées par l'utilisateur. Enfin, nous exécutons l'arbre.

En effet, nous avons vu que l'utilisateur devait effectuer la saisie des entrées avant de pouvoir lancer l'exécution. Ainsi, pour chaque type attendu en entrée d'un module, nous utilisons une interface de saisie. À l'initialisation de la fenêtre, un message stipule quel type doit être saisi, comme sur la figure 4.4 ci-dessous. Ensuite, l'utilisateur effectue la saisie de l'information demandée. Enfin, l'utilisateur appuie sur le bouton OK, ce qui écrit le type fournit dans le fichier texte des entrées et ferme la fenêtre de saisie. Cette opération est renouvelée pour chaque entrée.

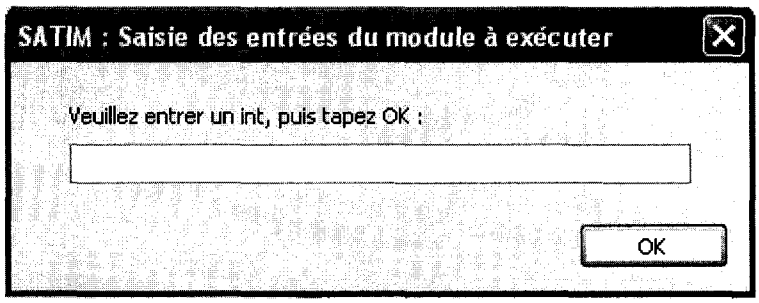


Figure 4.4 – Interface de saisie des types en entrée d'un module.

Nous sommes à un stade expérimental du développement, et n'effectuons donc pas de contrôle de type lors de la saisie. Normalement, il faudrait avoir un contrôle strict sur les types. Ceci pouvant, par exemple, être effectué en tenant à jour une base de données des types simples ou des types définis par l'utilisateur. Il faudrait alors avoir une interface pour pouvoir définir et enregistrer nos propres types.

4.4 Exemple de construction

Nous allons prendre l'exemple d'un module complexe Mc dont la chaîne de traitement est (BM2M1E1)((M1E1)(M2E2)). Ce module a deux entrées et deux sorties. Nous considérons que les modules Mc, M1 et M2 sont bien représentés dans la base de données et que les modules simples M1 et M2 possèdent un exécutable. Au début, nous vidons le dossier des résultats. Ensuite, nous construisons l'arbre en plusieurs étapes. Puis, nous faisons la saisie des entrées nécessaires. Enfin, nous exécutons l'arbre. À travers cet exemple, l'objectif est d'illustrer la construction de l'arbre. Nous allons donc voir en détail cette phase de construction.

Un nœud sera représenté de la façon suivante :

code	argument	entrées	sorties	parent	enfant	frère

Pour alléger les représentations qui vont suivre, nous donnerons les détails d'un nœud lors de sa première apparition mais ne préciserons plus ses fichiers textes par la suite, ceux-ci ne changeant pas. De plus, dans le même ordre d'idée, nous ne noterons pas le parent, l'enfant et le frère, qui seront alors symbolisés par une flèche vers le haut pour préciser un parent, une flèche vers le bas pour préciser un enfant et une flèche vers la droite pour préciser un frère.

La première étape de construction est l'application des combinateurs. La chaîne de traitement devient alors : (M2(M1(E1)))((M1E1)(M2E2)).

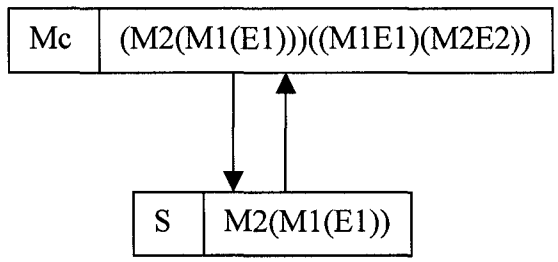
La deuxième étape est l'initialisation de la racine. Nous aurons donc comme racine :

Mc	(M2(M1(E1)))((M1E1)(M2E2))		Mc_output.txt

La troisième étape est l'ajout des sorties.
Nous ajoutons la première sortie qui est :

S	M2(M1(E1))	S_M2(M1(E1))_input.txt	S_M2(M1(E1))_output.txt

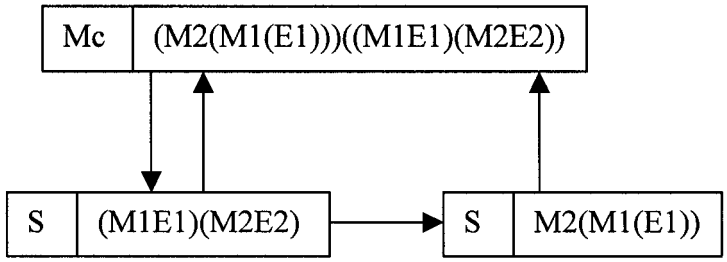
Et nous obtenons :



Nous ajoutons la deuxième sortie qui est :

S	(M1E1)(M2E2)	S_ (M1E1)(M2E2)_input.txt	S_ (M1E1)(M2E2)_output.txt

L'insertion d'un nouveau nœud se faisant en tête de liste, nous obtenons :



Nous constatons que l'enfant de la racine a été mis à jour.

La quatrième étape est le développement de chaque sortie. Celui-ci s'effectue de haut en bas et de gauche à droite. Le développement va le plus bas possible avant d'aller vers la droite.

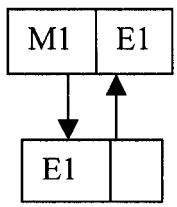
Nous développons la première sortie (M1E1)(M2E2).
Nous avons donc tout d'abord M1E1 qui donne :

M1	E1	M1_E1_input.txt	M1_E1_output.txt

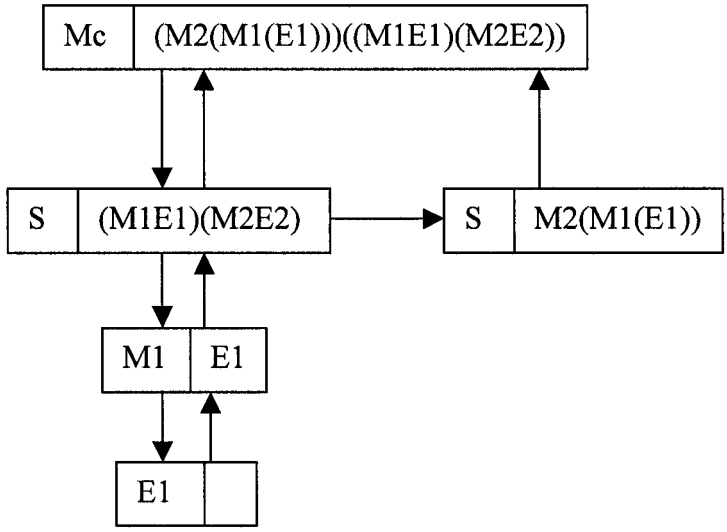
Il reste E1 qui donne :

E1		E1.txt	E1.txt

Nous l'ajoutons à M1, ce qui donne :



Nous ne pouvons plus descendre et ajoutons donc M1 dans l'arbre, ce qui donne :



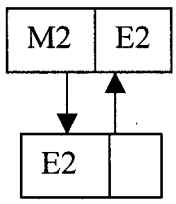
Nous avons ensuite M2E2 qui donne :

M2	E2	M2_ E2_input.txt	M2_ E2_output.txt

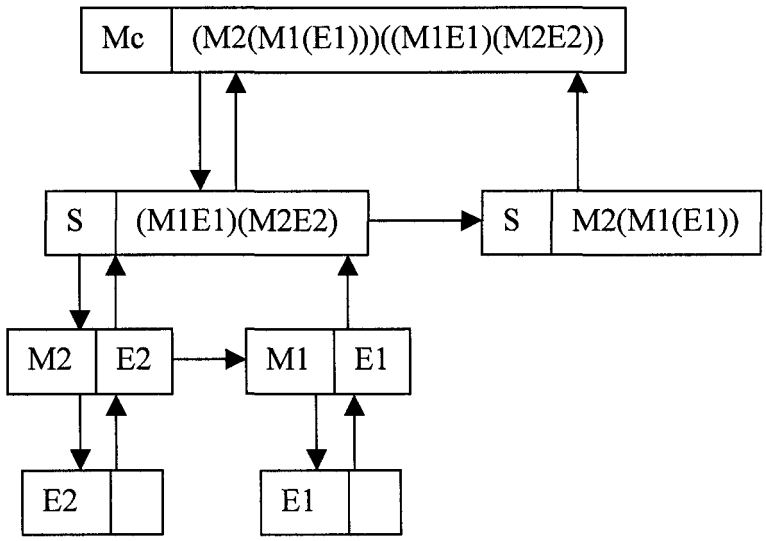
Il reste E2 qui donne :

E2		E2.txt	E2.txt

Nous l'ajoutons à M2, ce qui donne :



Nous ne pouvons plus descendre et ajoutons donc M2 dans l'arbre. L'insertion se faisant en tête de liste, nous obtenons :



Nous développons maintenant la deuxième sortie M2(M1(E1)).

Nous avons tout d'abord M2 qui donne :

M2	M1(E1)	M2_ M1(E1)_input.txt	M2_ M1(E1)_output.txt

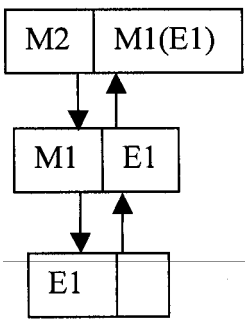
Il reste M1 qui donne :

M1	E1	M1_ E1_input.txt	M1_ E1_output.txt

Il reste E1 qui donne :

E1		E1.txt	E1.txt

Nous ajoutons E1 à M1, et le tout à M2, ce qui donne :



Nous ajoutons M2 dans l'arbre, ce qui donne l'arbre d'exécution final, présenté à la figure 4.5 suivante :

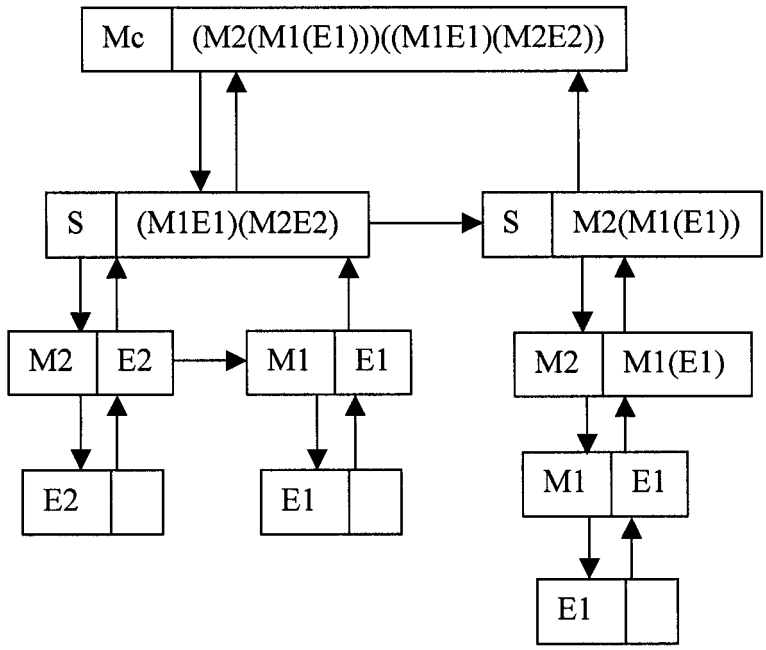


Figure 4.5 – Arbre d'exécution d'un module complexe Mc dont la chaîne de traitement est (BM2M1E1)((M1E1)(M2E2)).

4.5 Cas de test

Nous voici à l'étape des tests. L'objectif de cette partie est de montrer le bon fonctionnement du programme développé. Pour ce faire, nous allons reprendre tous les cas d'exécution possible identifiés au chapitre précédent, et vérifier si leur exécution donne bien les résultats attendus. Pour illustrer ces différents cas de modules complexes, nous avons créé les exécutables de huit modules simples. Tous les modules, qu'ils soient simples ou complexes, sont enregistrés dans la base de données pour pouvoir les exécuter avec notre programme. Afin d'alléger le document, un seul exemplaire d'entrée

sera présenté pour chaque test. De plus, dans le même ordre d'idée, nous ne présenterons pas les résultats intermédiaires de l'exécution, mais seulement la sortie finale du module exécuté.

4.5.1 Modules simples utilisés

Nous avons défini huit modules simples pour construire nos différents cas de test :

Le module M1 effectue l'addition de deux entiers.

Le module M2 effectue la soustraction de deux entiers.

Le module M3 effectue la multiplication de deux entiers.

Le module M4 effectue le carré d'un entier.

Le module M5 effectue la duplication d'un entier.

Le module M6 calcul le plus petit entier parmi trois entiers.

Le module M7 calcul le plus grand entier parmi trois entiers.

Le module M8 calcul la moyenne de six entiers.

Tous ces modules sont simples et ont donc 'I' comme chaîne de traitement, ainsi qu'une seule entrée et une seule sortie. Voici leur représentation donnée à la figure 4.6 qui suit.

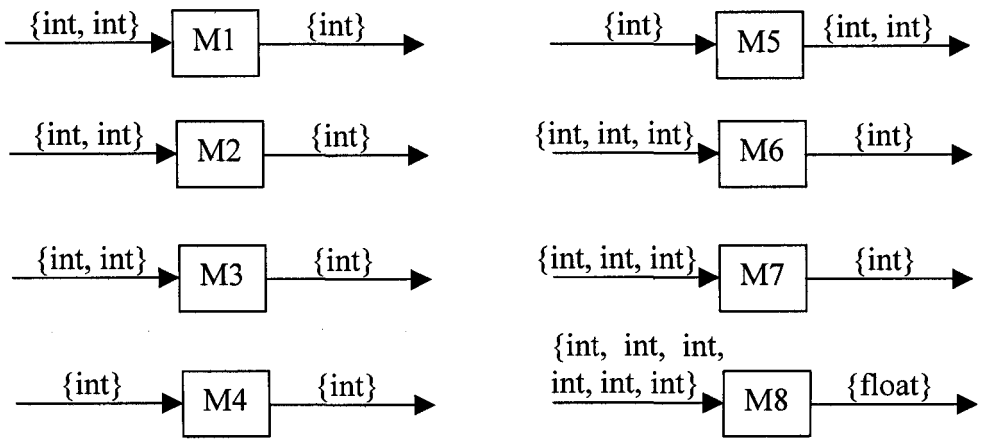


Figure 4.6 - Représentation des modules simples utilisés.

4.5.2 Série de 2 modules

M9 est un module complexe qui représente deux modules M1 et M4 en série. Voici sa représentation donnée à la figure 4.7.

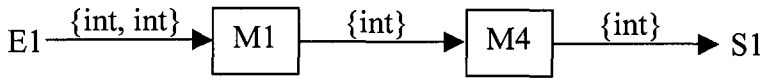


Figure 4.7 - M9, module illustrant la série de 2 modules.

Ce module possède une entrée E1={int, int} et une sortie S1={int}. D'après la partie 3.4.1.1, sa chaîne de traitement est :

(BM4M1E1)

À l'exécution du programme, nous obtenons le résultat suivant :

M9({1, 2}) = {9}

Ceci est bien le résultat attendu puisque $(1+2)^2 = 9$. La démonstration de la validité des résultats étant triviale, nous laisserons cela de côté dans la suite du document.

4.5.3 Série de 3 modules

M10 est un module complexe qui représente trois modules M1, M4 et M5 en série. Sa représentation est donnée à la figure 4.8.

Figure 4.8 - M10, module illustrant la série de 3 modules.

Ce module possède une entrée E1={int, int} et une sortie S1={int, int}. D'après la partie 3.4.1.2, sa chaîne de traitement est :

(BM5(BM4M1)E1)

À l'exécution du programme, nous obtenons le résultat suivant :

M10({1, 2}) = {9, 9}

4.5.4 Série de 6 modules

M11 est un module complexe qui représente six modules M5, M1, M5, M3, M4 et M4 en série. Sa représentation est donnée à la figure 4.9.

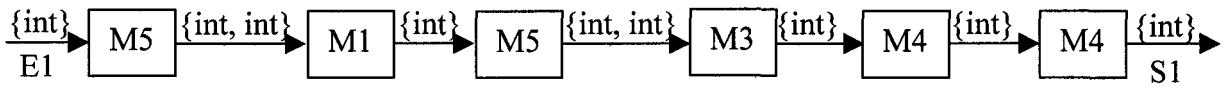


Figure 4.9 - M11, module illustrant la série de 6 modules.

Ce module possède une entrée E1={int} et une sortie S1={int}. D'après la partie 3.4.1.3, sa chaîne de traitement est :

(BM4(BM4(BM3(BM5(BM1M5))))E1)

À l'exécution du programme, nous obtenons le résultat suivant :

M11({1}) = {256}

4.5.5 Parallélisme indépendant de 3 modules

M12 est un module complexe qui représente le parallélisme indépendant de trois modules M1, M2 et M5. Sa représentation est donnée à la figure 4.10.

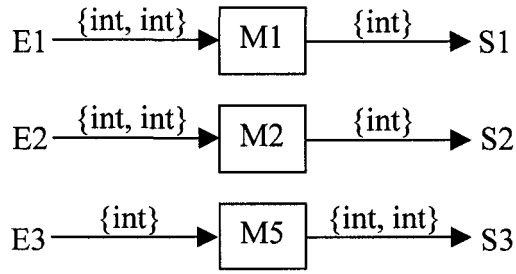


Figure 4.10 - M12, module illustrant le parallélisme indépendant de 3 modules.

Ce module possède trois entrées E1={int, int}, E2={int, int}, E3={int} et trois sorties S1={int}, S2={int}, S3={int, int}. D'après la partie 3.4.2.1.2, sa chaîne de traitement est :

(M1E1)(M2E2)(M5E3)

À l'exécution du programme, nous obtenons le résultat suivant :

M12({1, 2} {3, 4} {5}) = {3} {-1} {5, 5}

4.5.6 Parallélisme à entrées dépendantes de 3 modules

M13 est un module complexe qui représente le parallélisme à entrées dépendantes de trois modules M1, M2 et M3. Sa représentation est donnée à la figure 4.11.

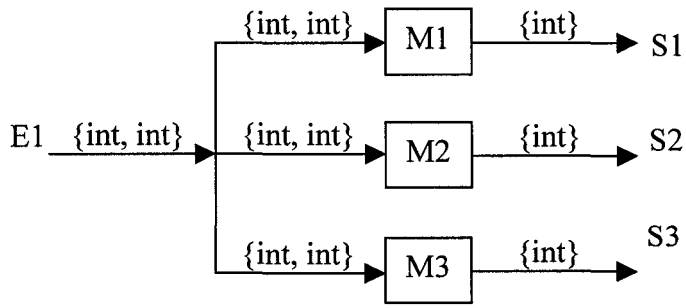


Figure 4.11 - M13, module illustrant le parallélisme à entrées dépendantes de 3 modules.

Ce module possède une entrée E1={int, int} et trois sorties S1={int}, S2={int}, S3={int}. D'après la partie 3.4.2.2.2, sa chaîne de traitement est :

(M1E1)(M2E1)(M3E1)

À l'exécution du programme, nous obtenons le résultat suivant :

M13({1, 2}) = {3} {-1} {2}

4.5.7 Parallélisme à sorties dépendantes de 3 modules

M14 est un module complexe qui représente le parallélisme à sorties dépendantes de trois modules M1, M2 et M5. Sa représentation est donnée à la figure 4.12.

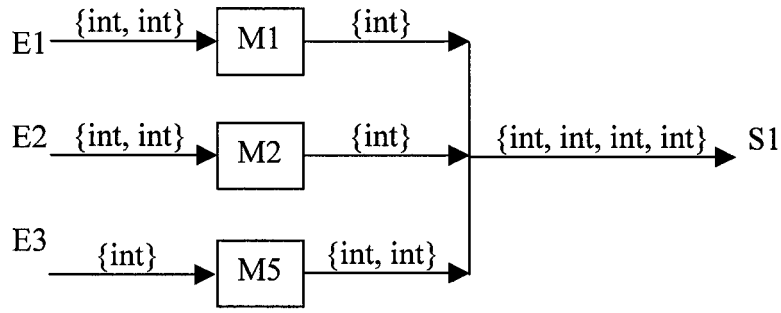


Figure 4.12 - M14, module illustrant le parallélisme à sorties dépendantes de 3 modules.

Ce module possède trois entrées E1={int, int}, E2={int, int}, E3={int} et une sortie S1={int, int, int, int}. D'après la partie 3.4.2.3.2, sa chaîne de traitement est :

((M1E1)(M2E2)(M5E3))

À l'exécution du programme, nous obtenons le résultat suivant :

M14({1, 2} {3, 4} {5}) = {3, -1, 5, 5}

4.5.8 Parallélisme dépendant de 3 modules

M15 est un module complexe qui représente le parallélisme dépendant de trois modules M1, M2 et M3. Sa représentation est donnée à la figure 4.13.

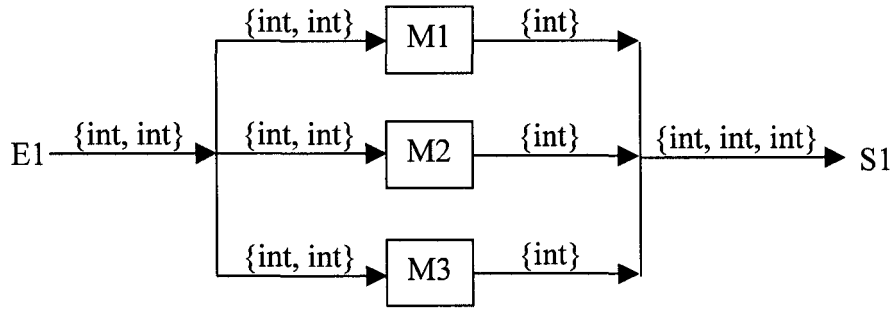


Figure 4.13 - M15, module illustrant le parallélisme dépendant de 3 modules.

Ce module possède une entrée E1={int, int} et une sortie S1={int, int, int}. D'après la partie 3.4.2.4.2, sa chaîne de traitement est :

((M1E1)(M2E1)(M3E1))

À l'exécution du programme, nous obtenons le résultat suivant :

M15({1, 2}) = {3, -1, 2}

4.5.9 Parallélisme de 2 modules avec récupération

M16 est un module complexe qui représente le parallélisme de deux modules M1 et M2 avec récupération dans M3. Sa représentation est donnée à la figure 4.14.

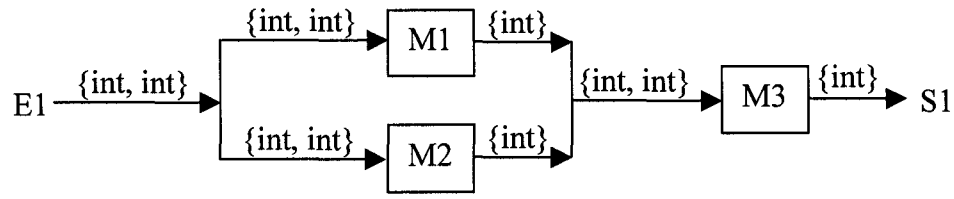


Figure 4.14 - M16, module illustrant le parallélisme de 2 modules avec récupération.

Ce module possède une entrée E1={int, int} et une sortie S1={int}. D'après la partie 3.4.2.5.1, sa chaîne de traitement est :

(PM3M1M2E1)

À l'exécution du programme, nous obtenons le résultat suivant :

M16({1, 2}) = {-3}

4.5.10 Parallélisme de 3 modules avec récupération

M17 est un module complexe qui représente le parallélisme de trois modules M1, M2 et M3 avec récupération dans M6. Sa représentation est donnée à la figure 4.15.

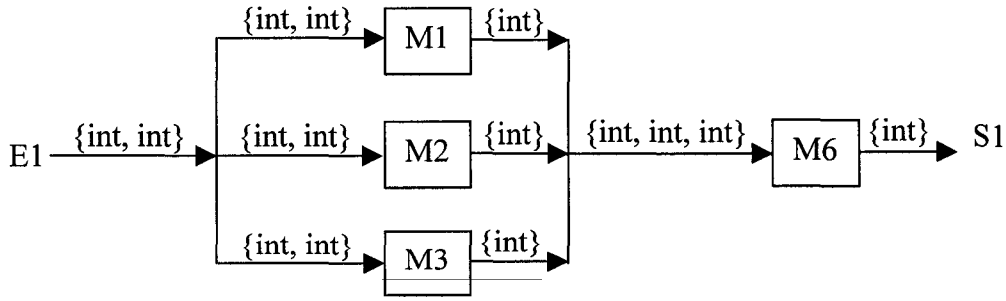


Figure 4.15 - M17, module illustrant le parallélisme de 3 modules avec récupération.

Ce module possède une entrée E1={int, int} et une sortie S1={int}. D'après la partie 3.4.2.5.2, sa chaîne de traitement est :

(S(PM6M1M2)M3E1)

À l'exécution du programme, nous obtenons le résultat suivant :

M17({1, 2}) = {-1}

4.5.11 Parallélisme de 6 modules avec récupération

M18 est un module complexe qui représente le parallélisme de six modules M1, M2, M3, M1, M2 et M3 avec récupération dans M8. Sa représentation est donnée à la figure 4.16.

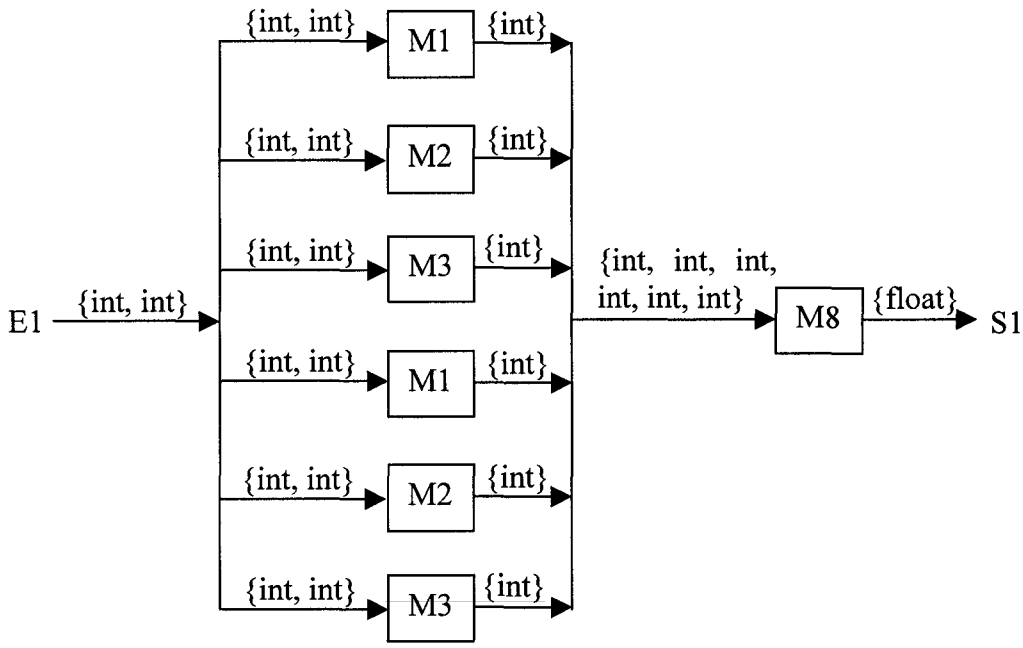


Figure 4.16 - M18, module illustrant le parallélisme de 6 modules avec récupération.

Ce module possède une entrée E1={int, int} et une sortie S1={float}. D'après la partie 3.4.2.5.3, sa chaîne de traitement est :

(S(S(S(S(PM8M1M2)M3)M1)M2)M3E1)

À l'exécution du programme, nous obtenons le résultat suivant :

M18({1, 2}) = {1.333333}

4.5.12 Traitement parallèle inclus dans le traitement sériel

M19 est un module complexe qui représente le traitement parallèle inclus dans le traitement sériel. Sa représentation est donnée à la figure 4.17.

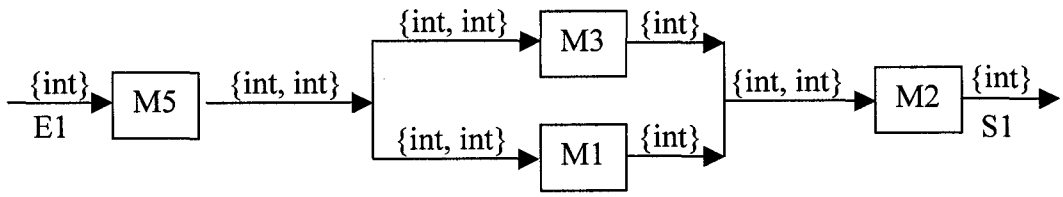


Figure 4.17 - M19, module illustrant le traitement parallèle inclus dans le traitement sériel.

Ce module possède une entrée E1={int} et une sortie S1={int}. D'après la partie 3.4.3.1, sa chaîne de traitement est :

(B(PM2M3M1)M5E1)

À l'exécution du programme, nous obtenons le résultat suivant :

M19({1}) = {-1}

4.5.13 Traitement sériel inclus dans le traitement parallèle

M20 est un module complexe qui représente le traitement sériel inclus dans le traitement parallèle. Sa représentation est donnée à la figure 4.18.

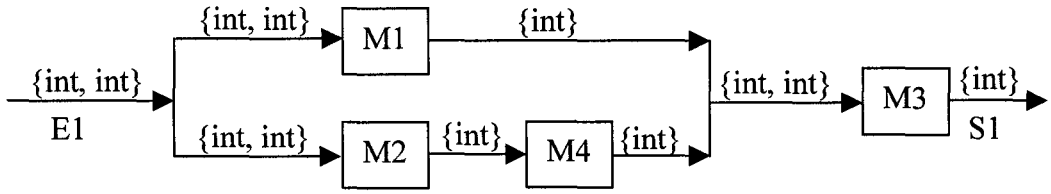


Figure 4.18 - M20, module illustrant le traitement sériel inclus dans le traitement parallèle.

Ce module possède une entrée E1={int, int} et une sortie S1={int}. D'après la partie 3.4.3.2, sa chaîne de traitement est :

(PM3M1(BM4M2)E1)

À l'exécution du programme, nous obtenons le résultat suivant :

M20({1, 2}) = {3}

4.5.14 Cas général de mélange des traitements sériels et parallèles

M21 est un module complexe qui représente le cas général de mélange des traitements sériels et parallèles. Sa représentation est donnée à la figure 4.19.

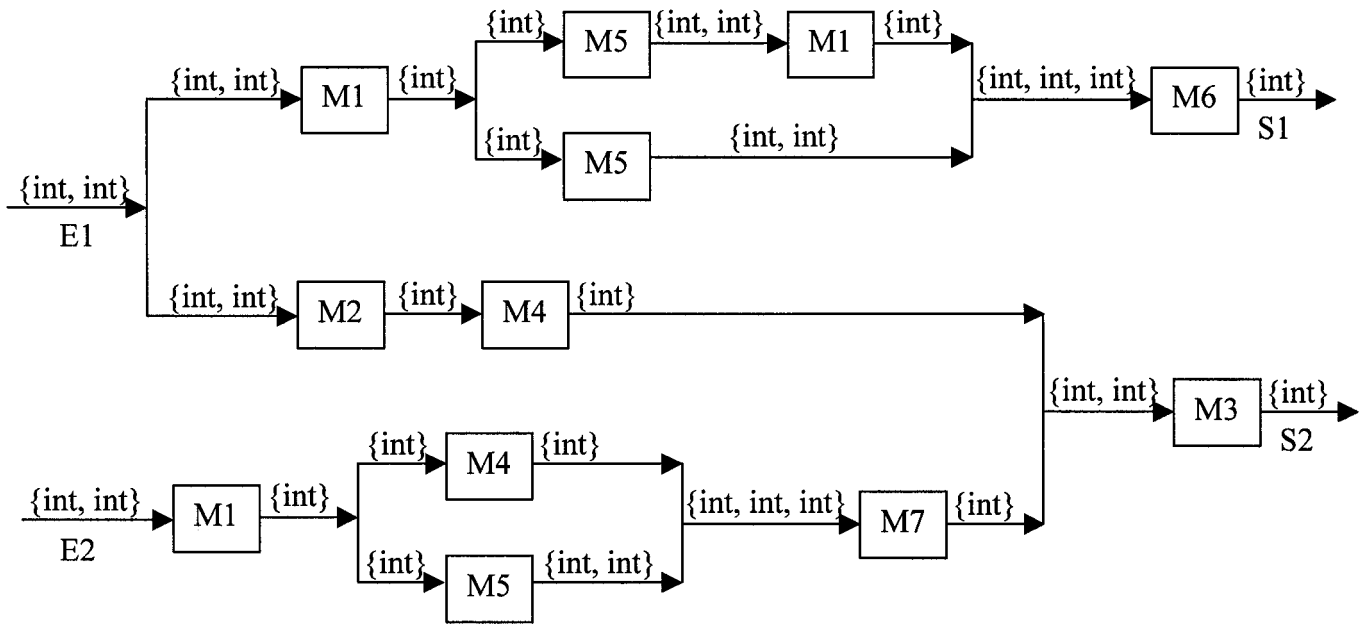


Figure 4.19 - M21, module illustrant le cas général de mélange des traitements sériels et parallèles.

Ce module possède deux entrées E1={int, int}, E2={int, int} et deux sorties S1={int}, S2={int}. D'après la partie 3.4.3.3, sa chaîne de traitement est :

(B(PM6(BM1M5)M5)M1E1)(M3((BM4M2E1)(B(PM7M4M5)M1E2)))

À l'exécution du programme, nous obtenons le résultat suivant :

M21({1, 2} {3, 4}) = {3} {49}

4.5.15 Branches d'injection

M22 est un module complexe qui représente les branches d'injection. Sa représentation est donnée à la figure 4.20.

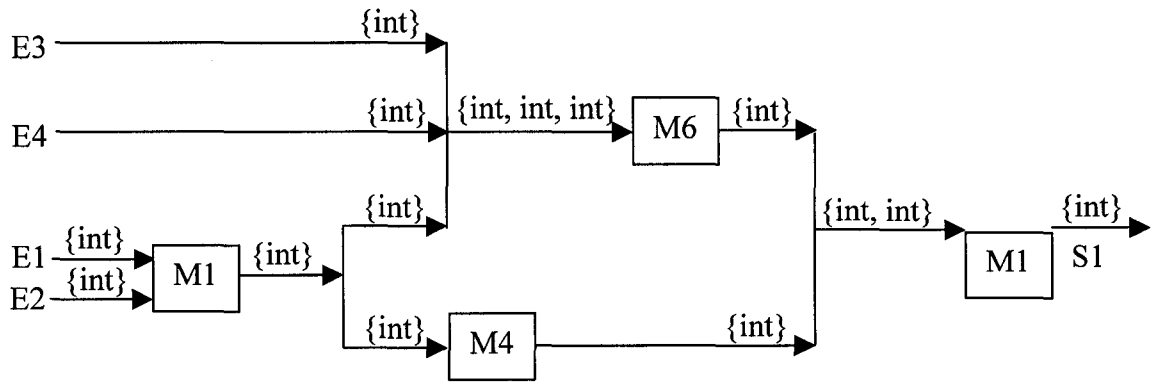


Figure 4.20 - M22, module illustrant les branches d'injection.

Ce module possède quatre entrées E1={int}, E2={int}, E3={int}, E4={int} et une sortie S1={int}. D'après la partie 3.5.1, sa chaîne de traitement est :

(M1(M6((M1(E1E2))E3E4)M4(M1(E1E2))))

À l'exécution du programme, nous obtenons le résultat suivant :

M22({1} {2} {3} {4}) = {12}

4.5.16 Branches d'éjection

M23 est un module complexe qui représente les branches d'éjection. Sa représentation est donnée à la figure 4.21.

Ce module possède une entrée E1={int, int} et quatre sorties S1={int}, S2={int}, S3={int}, S4={int}. D'après la partie 3.5.2, sa chaîne de traitement est :

(B(PM7M4M5)M1E1)(M1E1)(BM4M1E1)(BM4M1E1)

À l'exécution du programme, nous obtenons le résultat suivant :

M23({1, 2}) = {9} {3} {9} {9}

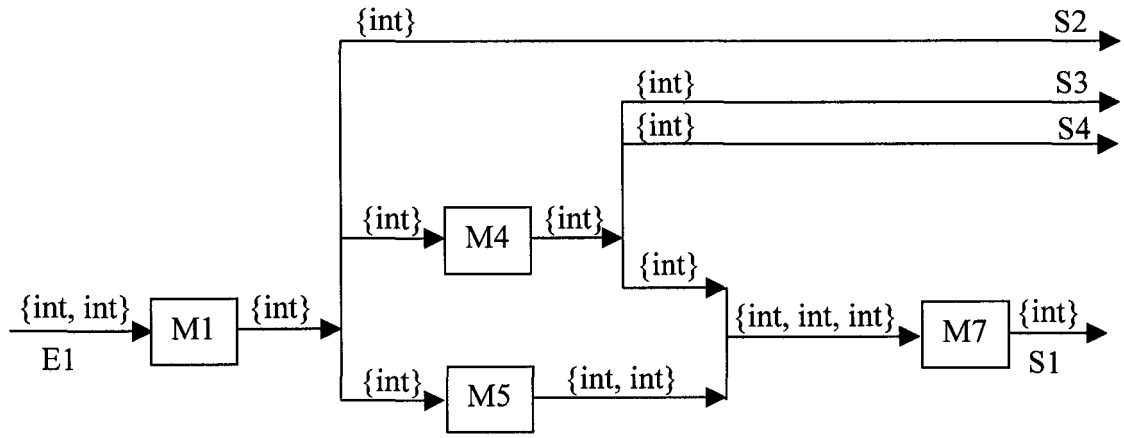


Figure 4.21 - M23, module illustrant les branches d'éjection.

4.5.17 Parallélisme spécial

M24 est un module complexe qui représente le parallélisme spécial. Sa représentation est donnée à la figure 4.22.

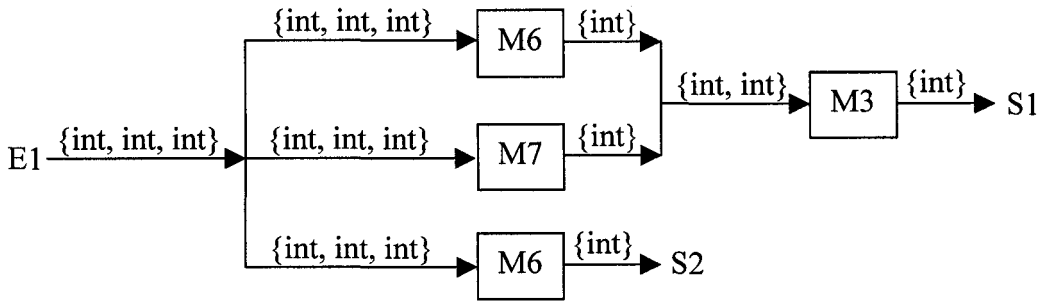


Figure 4.22 - M24, module illustrant le parallélisme spécial.

Ce module possède une entrée E1={int, int, int} et deux sorties S1={int}, S2={int}. D'après la partie 3.5.3.1, sa chaîne de traitement est :

(PM3M6M7E1)(M6E1)

À l'exécution du programme, nous obtenons le résultat suivant :

M24({1, 2, 3}) = {3} {1}

4.5.18 Parallélisme imbriqué

M25 est un module complexe qui représente le parallélisme imbriqué. Sa représentation est donnée à la figure 4.23.

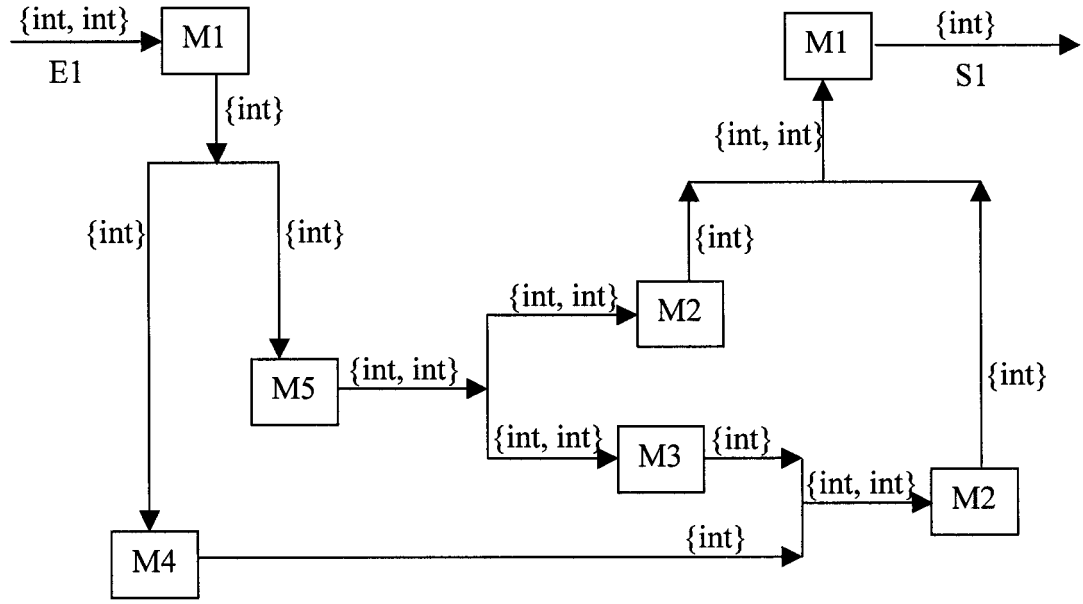


Figure 4.23 - M25, module illustrant le parallélisme imbriqué.

Ce module possède une entrée E1={int, int} et une sortie S1={int}. D'après la partie 3.5.3.2, sa chaîne de traitement est :

(M1((BM2(BM5M1)E1)(M2((BM3(BM5M1)E1)(BM4M1E1)))))

À l'exécution du programme, nous obtenons le résultat suivant :

M25({1, 2}) = {0}

4.5.19 Sériel imbriqué

M26 est un module complexe qui représente le sériel imbriqué. Sa représentation est donnée à la figure 4.24.

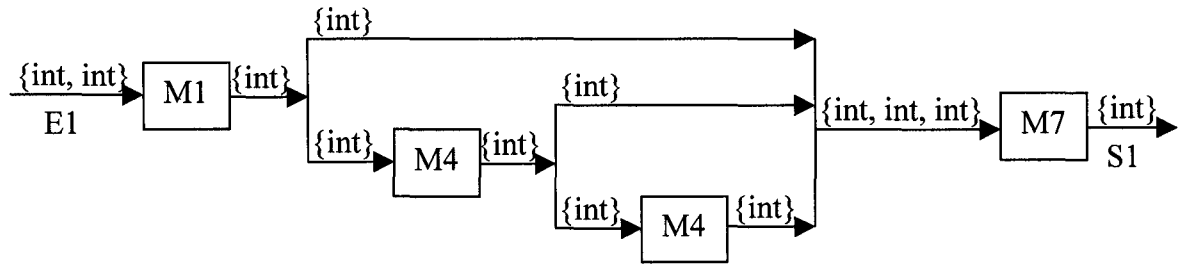


Figure 4.24 - M26, module illustrant le sériel imbriqué.

Ce module possède une entrée E1={int, int} et une sortie S1={int}. D'après la partie 3.5.3.3, sa chaîne de traitement est :

(M7((M1E1)(BM4M1E1)(BM4(BM4M1)E1)))

À l'exécution du programme, nous obtenons le résultat suivant :

M26({1, 2}) = {81}

4.5.20 Ordre des connexions

M27 et M28 sont des modules complexes qui illustrent l'importance de l'ordre des connexions pour les modules non commutatifs. Leurs représentations sont données aux figures 4.25 et 4.26.

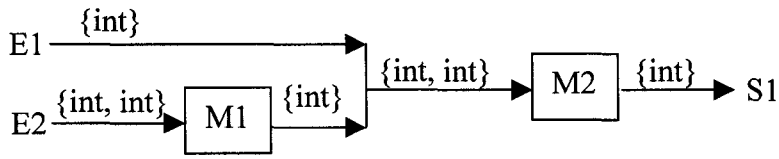


Figure 4.25 - M27, module illustrant l'importance de l'ordre des connexions.

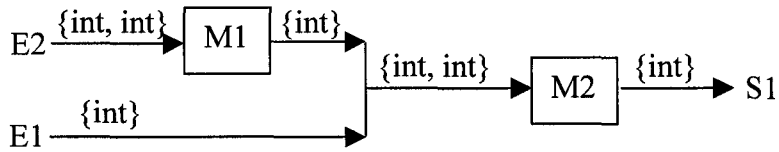


Figure 4.26 - M28, module illustrant l'importance de l'ordre des connexions.

Ces modules possèdent chacun deux entrées E1={int}, E2={int, int} et une sortie S1={int}. D'après la partie 3.5.3.4, leurs chaînes de traitement sont :

(M2(E1(M1E2))) pour M27,

(M2((M1E2)E1)) pour M28.

À l'exécution du programme, nous obtenons les résultats suivants :

M27({1} {2, 3}) = {-4}

M28({1} {2, 3}) = {4}

Les résultats sont bien différents puisque M2 est un module de soustraction et n'est donc pas commutatif. Si nous avions pris M3, le module de multiplication, à la place de M2, nous aurions obtenu les mêmes résultats avec M27 et M28. En effet, la multiplication est commutative.

4.5.21 Étude de cas générale

M29 est un module complexe qui représente l'étude de cas générale. Sa représentation est donnée à la figure 4.27.

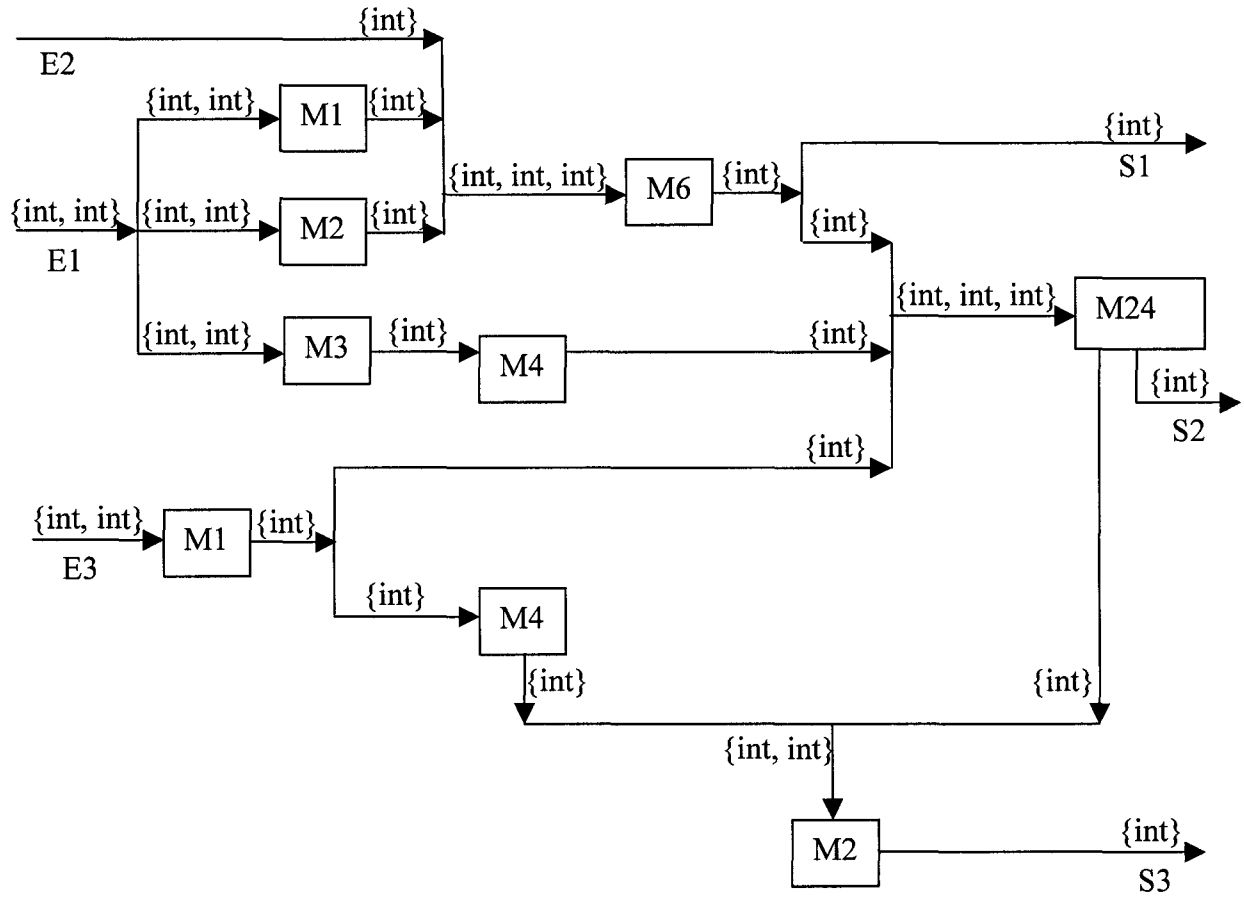


Figure 4.27 - M29, module illustrant l'étude de cas générale.

Ce module possède trois entrées E1={int, int}, E2={int}, E3={int, int} et trois sorties S1={int}, S2={int}, S3={int}. D'après la partie 3.6, sa chaîne de traitement est :

S1S2S3, avec

$$\begin{cases} S1 = (M6(E2(M1E1)(M2E1))) \\ S2 = (PM3M6M7((M6(E2(M1E1)(M2E1)))(BM4M3E1)(M1E3))) \\ S3 = (M2((M6((M6(E2(M1E1)(M2E1)))(BM4M3E1)(M1E3)))(BM4M1E3))) \end{cases}$$

À l'exécution du programme, nous obtenons le résultat suivant :

M29({1, 2} {3} {4, 5}) = {-1} {-9} {-82}

4.6 Conclusion

L'ensemble des résultats obtenus est satisfaisant. En effet, pour chaque cas illustré au chapitre 3, nous avons eu de bons résultats lors de différentes exécutions. Ceci nous permet d'affirmer que la théorie développée, au regard des cas identifiés, permet d'exécuter correctement un module complexe.

Nous pouvons remarquer qu'un module complexe est théoriquement un enchaînement particulier de modules simples, et que pratiquement cela se traduit par un enchaînement d'exécutables. En effet, chaque module simple est finalement un exécutable. Ainsi, la programmation d'un module complexe est complètement indépendante d'un langage de programmation particulier puisqu'elle ne consiste qu'en un enchaînement d'exécutables. Cette indépendance rejoint bien notre objectif d'offrir une plate-forme de développement destinée à des non informaticiens. Effectivement, ces derniers n'auront pas à faire de la programmation, au sens classique du terme, mais auront juste à enchaîner des exécutables pour obtenir le traitement qu'ils recherchent. Cette façon de faire permet aussi d'avoir beaucoup de souplesse lors du développement des chaînes de traitement. Il est assurément aisé d'ajouter ou de supprimer des modules simples pour modifier un module complexe, d'exécuter ce dernier en cours de construction pour vérifier son comportement, etc. En effet, tout changement n'implique pas de recompilation, mais juste une modification de la chaîne de traitement. Ceci satisfait donc aussi l'objectif d'offrir une plate-forme suffisamment souple pour des chercheurs afin qu'ils puissent progressivement et facilement affiner une solution en cours de développement. Cela les détache de la contrainte des langages de programmation usuels qui imposent de connaître l'ensemble des tenants et des aboutissants d'une solution.

Nous avons donc bien réussi à offrir un moyen d'exécuter des chaînes de traitement qui soit intéressant pour des chercheurs non informaticiens.

CONCLUSION

Certaines insuffisances ont été constatées au niveau des outils de développement dont disposent les ingénieurs de la langue. C'est pourquoi nous avons voulu mettre en œuvre une « nouvelle ingénierie de l'information ». L'objectif était de définir un modèle théorique de constructions systématiques de chaînes de traitement qui permette à des chercheurs non informaticiens d'enchaîner différents modules informatiques préexistants. Nous adressant à des chercheurs, il fallait leurs permettre d'exécuter une chaîne de traitement à tout moment pour qu'ils puissent progressivement et facilement affiner une solution en cours de développement. Il fallait aussi permettre de sauvegarder une chaîne jugée satisfaisante afin qu'ils puissent la réutiliser. De plus, nous adressant à des non informaticiens, notre choix s'est porté sur l'utilisation des exécutables des modules, et non sur leur code source, afin d'être totalement indépendant de tout langage de programmation, et d'ainsi leurs permettre de construire facilement des chaînes de traitement. En effet, dans une telle perspective, une chaîne de traitement est vue comme un assemblage ou une combinaison de modules. Cela laisse entrevoir une nouvelle façon de programmer qui consiste simplement à relier des modules exécutables de manière sérielle ou parallèle.

Dans un tel cadre, une chaîne sera dite valide si les modules qui la composent sont correctement reliés entre-eux, c'est-à-dire si chaque module reçoit le bon nombre et le bon type de ses arguments en entrée. De plus, la chaîne, une fois construite et validée, représentera à son tour un module dit complexe. Un module est simple s'il est seul, et complexe s'il est composé de plusieurs modules simples. Ainsi, l'exécution d'un module simple fera appel à l'exécutable de ce dernier, alors que celle d'un module complexe nécessitera l'exécution des modules qui le composent dans l'ordre spécifié à la construction du module complexe. L'interprétation d'une chaîne de traitement sera ainsi

l'aboutissement de l'interprétation des fonctions primitives qui la forment et de la façon dont ces fonctions sont agencées.

Ma recherche concerne le problème de l'exécution d'une chaîne de traitement, et non celui de sa construction. Ainsi, nous ne nous sommes pas occupé de la validité des enchaînements, mais il a néanmoins fallu développer un formalisme capable de représenter les enchaînements de modules possibles. Nous partions donc avec l'hypothèse que les chaînes à exécuter sont valides et stockées en mémoire dans une base de données.

Avant toute chose, nous avons étudié les paradigmes objet, agent et fonctionnel, pour déterminer l'approche qui réponde le mieux à nos besoins. Malgré la puissance de modélisation du monde réel qu'offre l'orienté objet, notamment grâce aux concepts d'abstraction et d'encapsulation, nous avons constaté des limites importantes reliées aux autres concepts, tels que l'héritage ou le polymorphisme. En effet, ces derniers induisent une complexité conceptuelle et affectent la réutilisabilité. Ensuite, l'étude du paradigme agent a mis en évidence le manque de standardisation tant au niveau de la définition même d'un agent, qu'au niveau des outils de développement disponibles. Effectivement, l'orienté agent est une discipline plutôt récente. De plus, les agents sont surtout intéressants pour les systèmes distribués, ce qui ne nous concerne pas. Nous avons alors terminé notre étude avec les éléments de la programmation fonctionnelle, qui répondent au mieux à nos exigences. En effet, la capacité de tout voir comme des expressions et des fonctions (à une ou zéro variable), rejoint notre idée de chaînes de traitement. Nous pouvons aisément voir chaque module d'une chaîne de traitement comme une fonction, et l'enchaînement des modules comme un nouveau module, dit complexe, qui serait équivalent à une fonction complexe.

Par la suite, nous nous sommes alors consacrés à une analyse de la programmation fonctionnelle. Dans ce domaine, le lambda-calcul et la logique combinatoire sont les

moyens utilisés actuellement par les informaticiens pour analyser les propriétés sémantiques des langages de programmation de haut-niveau. L'étude de ces deux théories nous a permis d'affirmer que la logique combinatoire de Curry est à privilégier, plutôt que le lambda-calcul et la lambda-abstraction de Church. En effet, d'après Desclés [DESC 1990], la théorie combinatoire nous permet d'en arriver à une logique dite sans variables, ce qui résout le problème majeur lié à l'utilisation du lambda-calcul, à savoir celui du télescopage des variables. De plus, les combinateurs ne sont rien d'autres que des opérateurs abstraits, dont le but est de composer les opérateurs d'un système applicatif pour former des opérateurs plus complexes. Ceci rejoint notre problématique, puisqu'une chaîne de traitement est l'arrangement de plusieurs modules, chacun étant vu comme un opérateur.

Nous avons donc fait appel aux combinateurs pour développer une théorie capable de formaliser les chaînes de traitement, afin de pouvoir exécuter tous les modules d'une chaîne dans l'ordre attendu. L'étude de divers cas d'enchaînement de modules a permis de montrer à quel point il était aisé de mémoriser l'ordre d'exécution d'un module complexe en utilisant la théorie des combinateurs. Les modules simples utilisent le combinateur I, et les chaînes de traitements font appels aux combinateurs B, S et Φ, pour être modélisées. B est utilisé pour modéliser des enchaînements en série, tandis que S et Φ servent pour les enchaînements en parallèle. Nous avons ainsi pu modéliser tous les cas d'enchaînements de modules identifiés.

Finalement, nous avons implémenté la partie qui concerne l'exécution des chaînes de traitement modélisées. L'exécution des cas généraux ou particuliers d'enchaînement de modules identifiés s'est très bien déroulée, puisque tous les modules simples impliqués dans une chaîne quelconque se sont exécutés dans le bon ordre. Nous avons donc développé une théorie qui s'avère efficace pour modéliser et exécuter des chaînes de traitement. Nous avons pu remarquer qu'un module complexe est théoriquement un enchaînement particulier de modules simples, et que pratiquement cela se traduit par un

enchaînement d'exécutables. Ceci nous dégage de tout langage de programmation et permet ainsi de s'adresser aux non informaticiens. Effectivement, ces derniers n'auront pas à faire de la programmation, au sens classique du terme, mais auront juste à enchaîner des exécutables pour obtenir le traitement qu'ils recherchent. Nous avons aussi vu que cette façon de construire une chaîne de traitement, en enchaînant des modules existants, ainsi que la possibilité de l'exécuter à tout moment, rejoint les besoins des chercheurs en les détachant de la contrainte des langages de programmation usuels, qui imposent de connaître l'ensemble des tenants et des aboutissants d'une solution. Nous avons donc bien réussi à offrir un moyen d'exécuter des chaînes de traitement qui soit intéressant pour des chercheurs non informaticiens.

Pour offrir une plate-forme opérationnelle aux ingénieurs de la langue, il faudra mettre au point une interface de construction des chaînes de traitement. Celle-ci aurait tout avantage à représenter visuellement les modules simples et les liens qui les unissent, en utilisant une boîte à outils, par exemple. Il faudra définir des règles et des contraintes sur l'enchaînement de ces modules, notamment au niveau du nombre et des types des arguments qui transitent entre chacun d'entre eux. Une vérification dynamique permettrait d'avoir des chaînes de traitement valides à tout moment de leur construction. Ainsi, les chaînes pourraient être sauvegardées et exécutées n'importe quand. La transformation d'un module complexe en module simple, pour pouvoir l'utiliser à son tour comme élément d'une nouvelle chaîne, nécessitera d'en obtenir une forme exécutable. Le projet d'une telle plate-forme laisse entrevoir une nouvelle façon de programmer qui réponde aux besoins actuels des chercheurs.

ANNEXE A

DÉTAILS DE L'IMPLÉMENTATION

Nous retrouvons ici les classes principales du projet avec des explications supplémentaires et leur code source.

La classe CModule

Nous considérons que la clef unique d'identification des modules sera un entier qui commence à 1. De plus, un module pouvant avoir plusieurs ensembles d'entrée ou de sortie, chacun d'eux étant un ensemble de types, nous avons défini des tableaux (tabInputs et tabOutputs) de tableaux de chaînes de caractères. Ainsi, chaque élément de tabInputs ou tabOutputs sera un ensemble de types. Par exemple, si nous considérons un module complexe avec deux entrées {int, int} et {float}, nous aurons tabInputs[0] = {int, int} et tabInputs[1] = {float} qui seront des tableaux de chaînes de caractères. Le traitement des sorties est équivalent à celui des entrées.

```
////////////////////////////////////////////////////////////////
//                                                            //
//    PROGRAMME :       Module.h                              //
//                                                            //
//    DESCRIPTION :     Un "module" est une fonction ou un    //
//                      programme, utilisé comme composant    //
//                      d'une chaîne de traitement.           //
//                                                            //
//                                                            //
//    AUTEUR :    Frédéric Gayton                             //
//    DATE :      Février 2004                                //
//                                                            //
////////////////////////////////////////////////////////////////

// Pour éviter les inclusions multiples
#pragma once

////////////////////////////////////////////////////////////////
//    CLASSE :          CModule                               //
//                                                            //
//    DESCRIPTION :     Déclaration d'un object "module"      //
////////////////////////////////////////////////////////////////
class CModule
{
private:
  // MAX_IOS : nombre maximum d'entrées/sorties autorisées
  enum { MAX_IOS = 100 };
```

```
//-----------------------------
//    Variables
//-----------------------------
  // Clef d'identification pour un module
  int code;
  // Nom significatif pour un module
  CString nom;
  // Domaine pour classer les modules
  CString domaine;
  // Sous-domaine pour préciser la classification
  CString sous_domaine;
  // Description du module
  CString description;
  // Chemin d'accès à l'exécutable du module
  CString chemin;
  // Nombre d'entrées du module
  int nbInput;
  // Nombre de sorties du module
  int nbOutput;
  // Exécutable du module
  CString executable;
  // Tableaux des entrées et sorties du module. Chaque entrée ou
  // sortie est un tableau de CString avec les types
  CStringArray tabInputs[MAX_IOS];
  CStringArray tabOutputs[MAX_IOS];

public:
//-----------------------------
//    Constructeurs & Destructeurs
//-----------------------------
  // Constructeur par défaut
  CModule(void);
  // Destructeur
  ~CModule(void);

//-----------------------------
//    Fonctions d'accès aux données privées
//-----------------------------
  // Fixe la valeur des variables
  void SetCode(int clef)                  { code = clef; }
  void SetName(CString name)              { nom = name; }
  void SetDomaine(CString dom)            { domaine = dom; }
  void SetSousDomaine(CString sous_dom)   { sous_domaine = sous_dom; }
  void SetDescription(CString descr)      { description = descr; }
  void SetPath(CString path)              { chemin = path; }
  void SetNbInput(int nbIn)               { nbInput = nbIn; }
  void SetNbOutput(int nbOut)             { nbOutput = nbOut; }
  void SetExecutable(CString exe)         { executable = exe; }
```

```
  void SetTabInputs(CStringArray *tab)
  {
    for(int i=0; i<nbInput; i++)
    {
      tabInputs[i].Copy(tab[i]);
    }
  }
  void SetTabOutputs(CStringArray *tab)
  {
    for(int i=0; i<nbOutput; i++)
    {
      tabOutputs[i].Copy(tab[i]);
    }
  }

  // Récupère la valeur des variables
  int GetCode(void)                 { return code; }
  CString GetName(void)             { return nom; }
  CString GetDomaine(void)          { return domaine; }
  CString GetSousDomaine(void)      { return sous_domaine; }
  CString GetDescription(void)      { return description; }
  CString GetPath(void)             { return chemin; }
  int GetNbInput(void)              { return nbInput; }
  int GetNbOutput(void)             { return nbOutput; }
  CString GetExecutable(void)       { return executable; }
  CStringArray *GetTabInputs(void)  { return tabInputs; }
  CStringArray *GetTabOutputs(void) { return tabOutputs; }

//-----------------------------
//    Fonctions
//-----------------------------
  // Copie un module source dans le notre
  void CopyMod(CModule &modSource);
};

////////////////////////////////////////////////////////////
//                                                        //
//    PROGRAMME :       Module.cpp                        //
//                                                        //
//    DESCRIPTION :     Fichier d'implémentation.         //
//                                                        //
//                                                        //
//    AUTEUR :    Frédéric Gayton                         //
//    DATE :      Février 2004                            //
//                                                        //
////////////////////////////////////////////////////////////

#include "StdAfx.h"
#include "Module.h"
```

```
// Constructeur
CModule::CModule(void)
{
}

// Destructeur
CModule::~CModule(void)
{
}

// Copie un module source dans le notre
void CModule::CopyMod(CModule &modSource)
{
  SetCode(modSource.GetCode());
  SetName(modSource.GetName());
  SetDomaine(modSource.GetDomaine());
  SetSousDomaine(modSource.GetSousDomaine());
  SetDescription(modSource.GetDescription());
  SetPath(modSource.GetPath());
  SetNbInput(modSource.GetNbInput());
  SetNbOutput(modSource.GetNbOutput());
  SetExecutable(modSource.GetExecutable());
  SetTabInputs(modSource.GetTabInputs());
  SetTabOutputs(modSource.GetTabOutputs());
}
```

1..1.1.1.1.1 La classe CBD

Hormis les éléments nécessaires pour l'accès à la base de données tels que le driver, la DNS, etc., une base de données sera caractérisée par son nom, l'endroit où elle se trouve et son nombre de tables. Une table sera définie par son nom, son nombre d'attributs, son nombre d'enregistrements et les valeurs des enregistrements (toutes sous forme de chaîne de caractères) dans un tableau à 1 dimension. La classe est munie des fonctions Init, SetTable et FillTables. Après avoir instancié un objet CBd, la fonction Init permet d'initialiser l'objet en spécifiant le nom et le chemin d'accès d'une base de données Access. Nous pouvons donc avoir plusieurs objets CBd, chacun correspondant à une base de données particulière.

Une fois l'objet initialisé, la fonction SetTable permet de lui attribuer une table en spécifiant le nom et le nombre d'attributs de la table. Il faut appeler cette fonction pour

chaque table, ce qui permet de facilement prendre en compte l'ajout ou la suppression d'une table. Le fait de devoir fournir le nombre d'attributs d'une table nous permet tout aussi facilement de prendre en compte l'ajout ou la suppression d'attributs. Finalement, la fonction FillTables permet de remplir les tables avec les valeurs contenues dans la base de données.

```
////////////////////////////////////////////////////////////
//                                                        //
//    PROGRAMME :       BD.h                              //
//                                                        //
//    DESCRIPTION :     Contient le nécessaire pour       //
//                      manipuler une base de données     //
//                      Microsoft Access.                 //
//                                                        //
//                                                        //
//    AUTEUR :    Frédéric Gayton                         //
//    DATE :      Mars 2004                               //
//                                                        //
////////////////////////////////////////////////////////////

// Pour éviter les inclusions multiples
#pragma once
// Pour avoir accés aux objets CDatabase, CRecordset
#include <afxdb.h>
// Pour utiliser la classe CStringArray
#include <afxcoll.h>

////////////////////////////////////////////////////////////
//    CLASSE :          CBd                               //
//                                                        //
//    DESCRIPTION :     Déclaration d'un object "BD"      //
////////////////////////////////////////////////////////////
class CBd
{
private:
  // MAX_TAB : nombre maximum de tables d'une BD
  enum { MAX_TAB = 100 };

  // Nom de la BD (ex: "bd.mdb")
  CString name;
  // Chemin de la BD (ex: "c:\\documents and settings\\bureau\\")
  CString path;
  // Nombre de tables de la BD
  int nbTables;
  // Driver de la BD
  CString driver;
  // DNS de la BD
  CString dns;
```

```
  // La BD
  CDatabase bd;
  // Un enregistrement
  CRecordset rs;

public:
  // Une table est définie par son nom, son nombre d'attributs, son
  // nombre d'enregistrements, et les valeurs des enregistrements dans
  // un tableau.
  // Les valeurs sont toutes en CString dans un tableau à 1 dimension.
  struct table
  {
    CString tableName;
    int nbChamps;
    int nbEnr;
    CStringArray tableau;
  };
  // Tableau des tables
  table *tabTables;

//-----------------------------
//    Constructeurs & Destructeurs
//-----------------------------
  // Constructeur par défaut
  CBd(void);
  // Destructeur
  ~CBd(void);

//-----------------------------
//    Fonctions d'accès aux données privées
//-----------------------------
  void SetName(CString BDname)      { name = BDname; }
  void SetPath(CString BDpath)      { path = BDpath; }
  CString GetName(void)       { return name; }
  CString GetPath(void)       { return path; }

//-----------------------------
//    Fonctions
//-----------------------------
  // Initialise la BD avec son nom et son chemin d'accès.
  // Si BDpath = "c:\\documents and settings\\username\\bureau\\"
  // la BD est sur le bureau.
  // Si BDpath = "" la BD est dans le dossier du projet.
  // Si BDpath = "BD\\" la BD est dans le dossier BD du dossier du
  // projet.
  void Init(CString BDname, CString BDpath);
  // Permet de définir les tables de la BD.
  // Il faut spécifier le nom et le nombre d'attributs de la table.
  void SetTable(CString tableName, int nbChamps);
  // Rempli les tables avec les valeurs contenue dans la BD.
  // Retourne FAUX si une erreur s'est produite.
  bool FillTables(void);
};
```

```
////////////////////////////////////////////////////////
//                                                    //
//    PROGRAMME :       BD.cpp                        //
//                                                    //
//    DESCRIPTION :     Fichier d'implémentation.     //
//                                                    //
//                                                    //
//    AUTEUR :    Frédéric Gayton                     //
//    DATE :      Mars 2004                           //
//                                                    //
////////////////////////////////////////////////////////

#include "StdAfx.h"
#include "BD.h"

// Constructeur
CBd::CBd(void)
{
  tabTables = new table[MAX_TAB];
}

// Destructeur
CBd::~CBd(void)
{
  delete [] tabTables;
}

// Initialise la BD
void CBd::Init(CString BDname, CString BDpath)
{
  SetName(BDname);
  SetPath(BDpath);
  driver = "MICROSOFT ACCESS DRIVER (*.mdb)";
  dns.Format("ODBC;DRIVER={%s};DSN='';DBQ=%s",driver,path+name);
  nbTables = 0;
}

// Permet de définir les tables de la BD
// Il faut spécifier le nom et le nombre d'attributs de la table
void CBd::SetTable(CString tableName, int nbChamps)
{
  // Met le nom et le nombre de champs de la table
  // dans le tableau des tables
  tabTables[nbTables].tableName = tableName;
  tabTables[nbTables].nbChamps = nbChamps;
  // Initialise le nombre d'enregistrements à zéro
  tabTables[nbTables].nbEnr = 0;
  // Incrémente de 1 le nombre de tables de la BD
  nbTables ++;
}
```

```
// Rempli les tables avec les valeurs contenue dans la BD
bool CBd::FillTables(void)
{
  // Numéro d'un enregistrement
  int noEnr;
  // Chaîne tampon
  CString tmpStr;

  // Ouvre la BD
  if(!bd.Open(NULL, FALSE, FALSE, dns, FALSE))
  {
    // Retourne FAUX si erreur d'ouverture de la BD
    return false;
  }

  // Fait pointer l'objet Recordset sur la BD
  rs.m_pDatabase = &bd;

  // Ouvre, lit, et ferme les tables
  for(int i=0; i<nbTables; i++)
  {
    // Ouvre la table i
    if(!rs.Open(CRecordset::snapshot,"SELECT * FROM " +
            tabTables[i].tableName, CRecordset::none))
    {
      // Ferme la BD et retourne FAUX si erreur
      bd.Close();
      return false;
    }
    // Se place sur le premier enregistrement, le numéro 0
    rs.MoveFirst();
    noEnr = 0;
    // Lit les enregistrements tant qu'il y en a
    while(!rs.IsEOF())
    {
      // Balaye les champs de la table i
      for(int noChamps=0; noChamps<tabTables[i].nbChamps; noChamps++)
      {
        // Récupère la valeur de la BD et la met dans tableau
        rs.GetFieldValue(noChamps, tmpStr);
        tabTables[i].tableau.Add(tmpStr);
        tmpStr.Empty();
      }
      // Incrémente de 1 le nombre d'enregistrements
      tabTables[i].nbEnr ++;
      // Se place sur le prochain enregistrement
      rs.MoveNext();
      noEnr ++;
    }
    // Ferme l'objet Recordset
    rs.Close();
  }
```

```
  // Ferme la BD
  bd.Close();

  return true;
}
```

L'arbre d'exécution

Nous y retrouvons deux variables globales : exeRep et txtRep. Nous avons exeRep = "Modules\\Exécutables\\" qui correspond au chemin d'accès du répertoire contenant les exécutables des modules simples, et txtRep = "Résultats\\" qui correspond au chemin d'accès du répertoire contenant les fichiers textes qui sont produits lors de l'exécution. Ceci permet à l'utilisateur de modifier facilement l'emplacement désiré pour ces répertoires. Ici, nous avons un seul répertoire qui contient les exécutables, mais il pourrait y en avoir plusieurs. Nous aurions pu utiliser le champ Path des modules de la base de données, mais ce dernier est variable suivant l'utilisateur ou la machine. C'est pourquoi, une norme devra être définie pour gérer facilement l'emplacement des exécutables, ce que nous laissons au soin des futurs développements.

La saisie des entrées est laissée au soin de celui qui utilise l'arbre, c'est pourquoi nous retrouvons uniquement deux fonctions publiques, l'une pour la construction et l'autre pour l'exécution. L'exécution d'un module simple utilise la fonction « spawnl » offerte par l'intermédiaire du fichier « process.h ». Cette fonction permet de lancer un exécutable automatiquement à partir de notre programme. Il faut notamment lui passer le chemin d'accès à l'exécutable, ainsi que les arguments nécessaires à ce dernier. Nous devons donc aussi lui passer les noms du fichier texte des entrées et du fichier texte des sorties puisque ce sont les arguments que nous avions définis pour les exécutables des modules simples.

L'arbre utilise ManipText qui regroupe un ensemble de fonctions servant à manipuler des fichiers textes. Nous avons défini les fonctions TextFileExist, TextFileEmpty, TextCreateSoft, TextCreateHard et TextCopy, dont voici une brève description :

- TextFileExist permet de savoir si un fichier texte particulier existe d'après le chemin d'accès absolu ou relatif spécifié.
- TextFileEmpty permet de savoir si un fichier texte particulier est vide ou bien s'il contient déjà de l'information.
- TextCreateSoft permet de créer un fichier texte, sauf s'il existe déjà.
- TextCreateHard permet de créer un fichier texte, même s'il existe déjà (il y a alors écrasement du fichier existant).
- TextCopy permet de copier un fichier texte source dans un fichier texte destination avec les options suivantes :
    - OVERWRITE_CREATE :
        - Si destination existe, l'écraser avec la source.
        - Si destination n'existe pas, le créer et le remplir avec la source.
    - OVERWRITE_NOTHING :
        - Si destination existe, l'écraser avec la source.
        - Si destination n'existe pas, ne rien faire.
    - APPENDTOEND_CREATE :
        - Si destination existe, lui ajouter la source à la fin.
        - Si destination n'existe pas, le créer et le remplir avec la source.
    - APPENDTOEND_NOTHING :
        - Si destination existe, lui ajouter la source à la fin.
        - Si destination n'existe pas, ne rien faire.
    - APPENDTOSTART_CREATE :
        - Si destination existe, lui ajouter la source au début.
        - Si destination n'existe pas, le créer et le remplir avec la source.
    - APPENDTOSTART_NOTHING :
        - Si destination existe, lui ajouter la source au début.

        - Si destination n'existe pas, ne rien faire.
    - NOTHING_CREATE :
        - Si destination existe, ne rien faire.
        - Si destination n'existe pas, le créer et le remplir avec la source.

L'arbre utilise aussi ManipString qui regroupe un ensemble de fonctions servant à manipuler la chaîne de traitement d'un module. Nous avons défini les fonctions IsCharNumeric, OutOfString, ExtractNumbers, ExtractFromParenthese, GetOperator, GetOperande, ApplyB, ApplyP, ApplyS et DevelopString, dont voici une brève description :

- IsCharNumeric permet de savoir si un caractère donné correspond à un chiffre.
- OutOfString permet de savoir si nous sommes hors ou dans une chaîne de caractères en passant la longueur de la chaîne et la position à vérifier.
- ExtractNumbers permet d'extraire les chiffres consécutifs qui se trouvent dans une chaîne de caractères à partir d'une position donnée.
- ExtractFromParenthese permet d'extraire la suite de caractères qui se trouve entre parenthèses à partir d'une position donnée de la parenthèse ouvrante.
- GetOperator permet d'extraire la partie opérateur après un combinateur qui se trouve à une position donnée. Un opérateur est toujours un module M quelconque ou bien contenu entre parenthèses.
- GetOperande permet d'extraire la partie opérande après un combinateur qui se trouve à une position donnée. Un opérande est toujours une entrée E quelconque ou bien contenu entre parenthèses.
- ApplyB permet d'appliquer le combinateur B en spécifiant sa position. Bfgx donnera f(g(x)), avec f et g opérateurs et x opérande.
- ApplyP permet d'appliquer le combinateur Φ en spécifiant sa position. Pfghx donnera f((g(x))(h(x))), avec f, g et h opérateurs et x opérande.
- ApplyS permet d'appliquer le combinateur S en spécifiant sa position. En théorie, Sfgx donnerait fx(g(x)), avec f et g opérateurs et x opérande. Mais dans

notre cas, f est toujours un "bloc" entre parenthèses, soit S(bloc)gx, avec bloc étant Pabc ou S(Pabc)d ou S(S(Pabc)d)e ou etc., suivant le nombre d'opérateurs en parallèle, avec a, b, c, d et e opérateurs. Alors, S sera traité récursivement sur lui-même en conservant les opérateurs impliqués, jusqu'à trouver P, et en traitant enfin le tout.

- DevelopString permet de développer une chaîne de traitement. C'est-à-dire que les combinateurs sont appliqués de façon à ne laisser que des modules, des entrées et des parenthèses dans la chaîne de traitement.

```
////////////////////////////////////////////////////////////
//                                                        //
//    PROGRAMME :       Tree.h                            //
//                                                        //
//    DESCRIPTION :     Un "tree" est un arbre            //
//                      d'exécution pour un module.       //
//                                                        //
//                                                        //
//    AUTEUR :    Frédéric Gayton                         //
//    DATE :      Mai 2004                                //
//                                                        //
////////////////////////////////////////////////////////////

// Pour éviter les inclusions multiples
#pragma once

// Chemin d'accès au répertoire des exécutables
const CString exeRep = "Modules\\Exécutables\\";
// Chemin d'accès aux fichiers textes de l'exécution
const CString txtRep = "Résultats\\";

////////////////////////////////////////////////////////////
//    CLASSE :          CTree                             //
//                                                        //
//    DESCRIPTION :     Déclaration d'un object "tree"    //
////////////////////////////////////////////////////////////
class CTree
{
private:
//-----------------------------
//    Variables qui caractérisent un noeud
//-----------------------------
  // Code d'un noeud :
  //  - M s'il représente un module
  //  - S s'il représente une sortie
  //  - E s'il représente une entrée
  CString code;
```

```
  // Argument d'un noeud :
  //  - Chaîne de traitement sur laquelle s'applique un module
  //  - Chaîne de traitement qui représente une sortie
  //  - Chaîne nulle si c'est une entrée
  CString arg;
  // Nom du fichier texte des entrées d'un noeud
  CString inputFileName;
  // Nom du fichier texte des sorties d'un noeud
  CString outputFileName;
  // Noeud parent
  CTree *parent;
  // Noeud enfant
  CTree *enfant1;
  // Noeud frère
  CTree *frere;

//-----------------------------
//    Fonctions de construction
//-----------------------------
  // Initialisation de la racine de l'arbre (niveau 0)
  // avec le code et la chaîne de traitement du module à traiter.
  void InitRacine(CString codeString, CString exeString);
  // Construction des sorties (niveau 1)
  void ConstructOutputs(void);
  // Développement récursif de l'arbre d'exécution (autres niveaux)
  void Develop(void);

//-----------------------------
//    Fonctions d'exécution
//-----------------------------
  // Suppression d'un noeud
  CTree* Destruction(CTree *Node);
  // Exécution d'un noeud d'entrée
  void ExecuteInput(void);
  // Exécution d'un noeud de module
  void ExecuteModule(void);
  // Exécution d'un noeud de sortie
  void ExecuteOutput(void);

public:
//-----------------------------
//    Constructeurs & Destructeurs
//-----------------------------
  // Constructeur par défaut
  CTree(void);
  // Constructeur de copie
  CTree(CTree *T);
  // Destructeur
  ~CTree(void);
```

```
//-----------------------------
//    Fonctions publiques
//-----------------------------
  // Construction de l'arbre d'exécution d'un module
  // défini par son code et sa chaîne de traitement
  void CTree::Construction(CString codeString, CString exeString);
  // Exécution de l'arbre d'exécution d'un module
  void Execution(void);
};


////////////////////////////////////////////////////////////
//                                                        //
//    PROGRAMME :       Tree.cpp                          //
//                                                        //
//    DESCRIPTION :     Fichier d'implémentation.         //
//                                                        //
//                                                        //
//    AUTEUR :    Frédéric Gayton                         //
//    DATE :      Mai 2004                                //
//                                                        //
////////////////////////////////////////////////////////////

#include "StdAfx.h"
#include "Tree.h"

#include "ManipString.h"
#include "ManipText.h"
// Pour utiliser la fonction "spawnl"
#include <process.h>

// Constructeur par défaut
CTree::CTree(void)
{
  enfant1 = NULL;
  frere = NULL;
  parent = NULL;
}

// Constructeur de copie
CTree::CTree(CTree *T)
{
  code = T->code;
  arg = T->arg;
  inputFileName = T->inputFileName;
  outputFileName = T->outputFileName;
  enfant1 = T->enfant1;
  frere = T->frere;
  parent = T->parent;
}
```

```
// Destructeur
CTree::~CTree(void)
{
}

// Initialisation de la racine de l'arbre (niveau 0)
// avec le code et la chaîne de traitement du module à traiter.
// La racine n'a pas de fichier texte des entrées puisqu'elles
// vont être dans les feuilles de l'arbre.
void CTree::InitRacine(CString codeString, CString exeString)
{
  this->code = codeString;
  this->arg = exeString;
  this->outputFileName = txtRep + this->code + "_output.txt";
  // Création du fichier texte des sorties
  TextCreateHard(this->outputFileName);
}

// Construction des sorties (niveau 1)
void CTree::ConstructOutputs(void)
{
  // Chaîne tampon
  CString tmpString;
  // Position dans la chaîne de traitement
  int pos = 0;
  // Caractère de parcours d'une chaîne
  char c = this->arg.GetAt(pos);

  // Si la sortie commence par une ouvrante
  if(c == '(')
  {
    // Première sortie récupérée des parenthèses
    tmpString = ExtractFromParenthese(this->arg, pos);
    // Création et initialisation d'un noeud pour la sortie
    CTree *OutNode = new CTree();
    OutNode->code = "S";
    OutNode->arg = tmpString;
    OutNode->inputFileName = txtRep + OutNode->code + "_"
      + OutNode->arg + "_input.txt";
    OutNode->outputFileName = txtRep + OutNode->code + "_"
      + OutNode->arg + "_output.txt";
    OutNode->parent = this;
    // Création des fichiers textes d'entrée et sortie
    TextCreateHard(OutNode->inputFileName);
    TextCreateHard(OutNode->outputFileName);
    // Sortie ajoutée comme enfant de la racine
    this->enfant1 = OutNode;
    // Positionnement sur le caractère après "(sortie)..." :
    // sortie.lenght() + 2 (pour la fermante et le prochain caractère)
    pos += tmpString.GetLength();
    pos += 2;
```

```
    // Si la position existe, il y a d'autres sorties
    while(pos < this->arg.GetLength())
    {
      // Récupération de la sortie
      tmpString = ExtractFromParenthese(this->arg, pos);
      // Création et initialisation d'un noeud pour la sortie
      CTree *OutNode = new CTree();
      OutNode->code = "S";
      OutNode->arg = tmpString;
      OutNode->inputFileName = txtRep + OutNode->code + "_"
        + OutNode->arg + "_input.txt";
      OutNode->outputFileName = txtRep + OutNode->code + "_"
        + OutNode->arg + "_output.txt";
      OutNode->parent = this;
      // Création des fichiers textes d'entrée et sortie
      TextCreateHard(OutNode->inputFileName);
      TextCreateHard(OutNode->outputFileName);
      // Noeud tampon qui récupère l'enfant de la racine
      CTree *TmpNode = this->enfant1;
      // Sortie mise à la place de l'enfant de la racine
      this->enfant1 = OutNode;
      // Ancien enfant (le noeud tampon) mis comme frère
      // de l'enfant (le noeud de sortie)
      OutNode->frere = TmpNode;
      // Positionnement sur la prochaine sortie
      pos += tmpString.GetLength();
      pos += 2;
    }
  }

  // Sinon la sortie ne commence pas par une ouvrante
  else
  {
    // Message d'erreur
    MessageBox(NULL, "Une sortie doit être entre parenthèses!",
      this->arg, MB_OK);
  }
}

// Développement récursif de l'arbre d'exécution (autres niveaux)
void CTree::Develop(void)
{
  // Chaîne tampon
  CString tmpString;
  // Chaîne du numéro d'un module M ou d'une entrée E
  CString numString;
  // Position dans la chaîne de traitement
  int pos = 0;
  // Caractère de parcours d'une chaîne
  char c = this->arg.GetAt(pos);
```

```
// Si l'argument du noeud commence par 'M'
if(c == 'M')
{
  // Création d'un nouveau noeud
  CTree *NewNode = new CTree();
  // Récupération du numéro de module
  pos += 1;
  numString = ExtractNumbers(this->arg, pos);
  // Initialisation du code
  NewNode->code = "M" + numString;
  // Positionnement sur l'argument de M
  pos += numString.GetLength();
  c = this->arg.GetAt(pos);
  // Si l'argument commence par 'E'
  if(c == 'E')
  {
    // Récupération du numéro d'entrée
    pos += 1;
    numString = ExtractNumbers(this->arg, pos);
    // Initialisation de l'argument de M à l'entrée E
    NewNode->arg = "E" + numString;
    // Positionnement sur le caractère qui suit le numéro d'entrée
    pos += numString.GetLength();
  }
  // Sinon si l'argument commence par '('
  else if(c == '(')
  {
    // Récupération du bloc
    tmpString = ExtractFromParenthese(this->arg, pos);
    // Initialisation de l'argument de M au bloc entre parenthèses
    NewNode->arg = tmpString;
    // Positionnement sur la suite du bloc
    pos += tmpString.GetLength();
    pos += 2;
  }
  // Assignation des noms aux fichiers textes d'entrée et sortie
  NewNode->inputFileName = txtRep + NewNode->code + "_"
    + NewNode->arg + "_input.txt";
  NewNode->outputFileName = txtRep + NewNode->code + "_"
    + NewNode->arg + "_output.txt";
  // Initialisation du parent
  NewNode->parent = this;
  // Création des fichiers textes d'entrée et sortie
  TextCreateHard(NewNode->inputFileName);
  TextCreateHard(NewNode->outputFileName);
  // Ajout du nouveau noeud comme enfant
  this->enfant1 = NewNode;
  // Si la position suivante existe, il y a d'autres arguments
  if(pos < this->arg.GetLength())
  {
    // Création, initialisation et développement d'un tampon,
    // qui n'est ni un noeud, ni une feuille.
    // C'est pourquoi on s'intéresse juste à son enfant
```

```
      CTree *TMP = new CTree();
      TMP->arg = this->arg.Right(this->arg.GetLength() - pos);
      TMP->Develop();
      CTree *NewNode = TMP->enfant1;
      // Un nouveau parent pour le nouveau noeud et tous ces frères
      NewNode->parent = this;
      while(NewNode->frere != NULL)
      {
        // Nouveau noeud déplacé sur ces frères
        NewNode = NewNode->frere;
        NewNode->parent = this;
      }
      // Repositionne le nouveau noeud sur le premier enfant
      NewNode = TMP->enfant1;
      delete TMP;
      // Noeud tampon qui récupère l'enfant de la racine
      CTree *TmpNode = this->enfant1;
      // Sortie mise à la place de l'enfant de la racine
      this->enfant1 = NewNode;
      // Ancien enfant (le noeud tampon) mis comme dernier
      // frère de l'enfant (le noeud de sortie)
      while(NewNode->frere != NULL)
      {
        NewNode = NewNode->frere;
      }
      NewNode->frere = TmpNode;
    }
    // Développement du nouveau noeud puisque ce n'est pas une feuille
    NewNode->Develop();
  }

  // Sinon si l'argument du noeud commence par 'E'
  else if(c == 'E')
  {
    // Récupération du numéro d'entrée
    pos += 1;
    numString = ExtractNumbers(this->arg, pos);
    // Création et initialisation d'une nouvelle feuille
    CTree *NewNode = new CTree();
    NewNode->code = "E" + numString;
    NewNode->arg = "";
    NewNode->inputFileName = NewNode->outputFileName = txtRep
      + NewNode->code + ".txt";
    NewNode->parent = this;
    // Création des fichiers textes d'entrée et sortie
    TextCreateHard(NewNode->inputFileName);
    TextCreateHard(NewNode->outputFileName);
    // Ajout du nouveau noeud comme enfant
    this->enfant1 = NewNode;
    // Positionnement sur le caractère qui suit le numéro d'entrée
    pos += numString.GetLength();
```

```
    // Si la position existe, il y a d'autres arguments
    if(pos < this->arg.GetLength())
    {
      // Création, initialisation et développement d'un tampon,
      // qui n'est ni un noeud, ni une feuille.
      // C'est pourquoi on s'intéresse juste à son enfant
      CTree *TMP = new CTree();
      TMP->arg = this->arg.Right(this->arg.GetLength() - pos);
      TMP->Develop();
      CTree *NewNode = TMP->enfant1;
      // Un nouveau parent pour le nouveau noeud et tous ces frères
      NewNode->parent = this;
      while(NewNode->frere != NULL)
      {
        // Nouveau noeud déplacé sur ces frères
        NewNode = NewNode->frere;
        NewNode->parent = this;
      }
      // Repositionne le nouveau noeud sur le premier enfant
      NewNode = TMP->enfant1;
      delete TMP;
      // Noeud tampon qui récupère l'enfant de la racine
      CTree *TmpNode = this->enfant1;
      // Sortie mise à la place de l'enfant de la racine
      this->enfant1 = NewNode;
      // Ancien enfant (le noeud tampon) mis comme dernier
      // frère de l'enfant (le noeud de sortie)
      while(NewNode->frere != NULL)
      {
        NewNode = NewNode->frere;
      }
      NewNode->frere = TmpNode;
    }
  }

  // Sinon l'argument du noeud commence par '('
  else
  {
    // Récupération du bloc entre parenthèses
    tmpString = ExtractFromParenthese(this->arg, pos);
    // Création, initialisation et développement d'un tampon,
    // qui n'est ni un noeud(M), ni une feuille(E).
    // C'est pourquoi on s'intéresse juste à son enfant
    CTree *TMP = new CTree();
    TMP->arg = tmpString;
    TMP->Develop();
    CTree *NewNode = TMP->enfant1;
    delete TMP;
    NewNode->parent = this;
    // Ajout du nouveau noeud comme enfant
    this->enfant1 = NewNode;
    // Positionnement sur la suite du bloc
    pos += tmpString.GetLength();
```

```
    pos += 2;
    // Si la position existe, il y a d'autres arguments
    if(pos < this->arg.GetLength())
    {
      // Création, initialisation et développement d'un tampon,
      // qui n'est ni un noeud(M), ni une feuille(E).
      // C'est pourquoi on s'intéresse juste à son enfant
      CTree *TMP = new CTree();
      TMP->arg = this->arg.Right(this->arg.GetLength() - pos);
      TMP->Develop();
      CTree *NewNode = TMP->enfant1;
      // Un nouveau parent pour le nouveau noeud et tous ces frères
      NewNode->parent = this;
      while(NewNode->frere != NULL)
      {
        NewNode = NewNode->frere;
        NewNode->parent = this;
      }
      // Repositionne le nouveau noeud sur le premier enfant
      NewNode = TMP->enfant1;
      delete TMP;
      // Noeud tampon qui récupère l'enfant de la racine
      CTree *TmpNode = this->enfant1;
      // Sortie mise à la place de l'enfant de la racine
      this->enfant1 = NewNode;
      // Ancien enfant (le noeud tampon) mis comme dernier
      // frère de l'enfant (le noeud de sortie)
      while(NewNode->frere != NULL)
      {
        NewNode = NewNode->frere;
      }
      NewNode->frere = TmpNode;
    }
  }
}

// Suppression d'un noeud
CTree* CTree::Destruction(CTree *Node)
{
  // Si ce n'est pas la racine, soit s'il a un parent
  if(Node->parent != NULL)
  {
    // Si l'enfant de son parent est lui-même,
    // alors le noeud précédent est son parent.
    if(Node->parent->enfant1 == Node)
    {
      // Positionnement du noeud courant sur son parent
      Node = Node->parent;
      // Suppression de son enfant
      Node->enfant1 = NULL;
    }
```

```
    // Sinon le noeud précédent est un frère
    else
    {
      // Positionnement du noeud courant sur l'enfant de son parent
      Node = Node->parent->enfant1;
      // Déplacement de frère en frère (jusqu'à l'avant dernier)
      while(Node->frere->frere != NULL)
      {
        // Positionnement sur le prochain frère
        Node = Node->frere;
      }
      // Suppression de son frère (le dernier de la liste)
      Node->frere = NULL;
    }
  }
  // Sinon c'est la racine
  else
  {
    Node = NULL;
  }

  return Node;
}

// Exécution d'un noeud d'entrée
void CTree::ExecuteInput(void)
{
  // Si l'exécution n'a pas déjà eu lieu, le fichier de sortie
  // du parent est vide. Sinon on ne fait rien.
  if(TextFileEmpty(this->parent->outputFileName) == 0)
  {
    // Copie l'entrée dans le fichier d'entrée du noeud parent
    TextCopy(this->outputFileName, this->parent->inputFileName,
      "APPENDTOEND_CREATE");
  }
}

// Exécution d'un noeud de module
void CTree::ExecuteModule(void)
{
  // Status de l'exécution
  int status;
  // Si l'exécution n'a pas déjà eu lieu, son fichier de sortie
  // est vide
  if(TextFileEmpty(this->outputFileName) == 0)
  {
    // Initialise le chemin d'accès à l'exécutable du module
    CString exePath = exeRep + this->code + ".exe";
    // Exécution du module
    status = spawnl(P_WAIT, exePath, this->inputFileName,
      this->outputFileName, NULL);
```

```
    // Si erreur, lors de l'exécution du module simple (avec un .exe)
    if(status != 0)
    {
      MessageBox(NULL, "Erreur d'exécution!", "Module simple.", MB_OK);
    }
  }
  // Si nous ne sommes pas à la racine, le noeud a un parent
  if(this->parent != NULL)
  {
    // Copie la sortie dans le fichier d'entrée du noeud parent
    TextCopy(this->outputFileName, this->parent->inputFileName,
      "APPENDTOEND_CREATE");
  }
}

// Exécution d'un noeud de sortie
void CTree::ExecuteOutput(void)
{
  // Si l'exécution n'a pas déjà eu lieu, son fichier de sortie
  // est vide
  if(TextFileEmpty(this->outputFileName) == 0)
  {
    // Copie le fichier d'entrée de la sortie dans son fichier
    // de sortie
    TextCopy(this->inputFileName, this->outputFileName,
      "APPENDTOEND_CREATE");
  }
  // Copie le fichier de sortie dans le fichier de sortie de la racine
  TextCopy(this->outputFileName, this->parent->outputFileName,
    "APPENDTOEND_CREATE");
}

// Construction de l'arbre d'exécution d'un module
// défini par son code et sa chaîne de traitement
void CTree::Construction(CString codeString, CString exeString)
{
  // PRÉPARATION DE LA CHAÎNE
  // Si c'est un module simple
  if(exeString == "I")
  {
    exeString = "(" + codeString + "E1)";
  }
  // Sinon il est complexe
  else
  {
    // Développement de la chaîne de traitement
    exeString = DevelopString(exeString);
  }

  // CONSTRUCTION DE L'ARBRE
  // Initialise la racine (niveau 0)
  this->InitRacine(codeString, exeString);
```

```
  // Ajoute les sorties (niveau 1)
  this->ConstructOutputs();
  // Développement de chaque sortie (autres niveaux)
  CTree *TmpNode = this->enfant1;
  TmpNode->Develop();
  while(TmpNode->frere != NULL)
  {
    TmpNode = TmpNode->frere;
    TmpNode->Develop();
  }
}

// Exécution de l'arbre d'exécution d'un module
void CTree::Execution(void)
{
  // Si le noeud existe
  if(this != NULL)
  {
    // Création du noeud courant qui sert à se promener dans l'arbre.
    // Il est initialisé à la racine au départ.
    CTree *CurrentNode = this;

    // Si le noeud courant a un frère
    if(CurrentNode->frere != NULL)
    {
      // Déplacement du noeud courant sur le frère
      CurrentNode = CurrentNode->frere;
      // Exécution du module courant
      CurrentNode->Execution();
    }
    // Sinon si il a un enfant
    else if(CurrentNode->enfant1 != NULL)
    {
      // Déplacement du noeud courant sur l'enfant
      CurrentNode = CurrentNode->enfant1;
      // Exécution du module courant
      CurrentNode->Execution();
    }
    // Sinon le noeud courant a ni frere, ni enfant : on l'exécute
    else
    {
      // Récupération du premier caractère du code
      char c = CurrentNode->code.GetAt(0);
      // Si c'est une entrée à exécuter
      if(c == 'E')
      {
        // Exécution de l'entrée
        CurrentNode->ExecuteInput();
        // Suppression du noeud
        CurrentNode = CurrentNode->Destruction(CurrentNode);
        // Exécution du module courant
        CurrentNode->Execution();
      }
```

```
      // Sinon si c'est un module à exécuter
      else if(c == 'M')
      {
        // Exécution du module
        CurrentNode->ExecuteModule();
        // Suppression du noeud
        CurrentNode = CurrentNode->Destruction(CurrentNode);
        // Exécution du module courant
        CurrentNode->Execution();
      }
      // Sinon si c'est une sortie à exécuter
      else if(c == 'S')
      {
        // Exécution de la sortie
        CurrentNode->ExecuteOutput();
        // Suppression du noeud
        CurrentNode = CurrentNode->Destruction(CurrentNode);
        // Exécution du module courant
        CurrentNode->Execution();
      }
      // Sinon erreur
      else
      {
        MessageBox(NULL, "Problème lors de l'exécution!",
          "Exécution de l'arbre.", MB_OK);
      }
    }
  }
  // Sinon terminé
  else
  {
    MessageBox(NULL, "Exécution terminée!",
      "Exécution de l'arbre.", MB_OK);
  }
}
```

BIBLIOGRAPHIE

[AJDU 1935] AJDUKIEWICZ, K., (1935), "Die syntaktishe Konnexität", *Studia Philosophica*, Vol. 1, p. 1-27, repris dans *Polish Logic 1920-1939*, sous le titre "Syntactic connexion", trad. De H. Weber, 1967, Oxford, Clarendon (S.McCalled.).

[AMBL 1992] AMBLER, A. L., BURNETT, M. M., ZIMMERMAN, B. A., (1992), *Operationnal versus definitional : a perspective on programming paradigms*, IEEE Computer.

[APPL 1997] APPLEBY, D., VANDEKOPPLE, J. J., (1997), *Programming languages : paradigm and practice*, 2nd Ed., McGraw-Hill.

[BACK 1978] BACKUS, J. W., (1978), "Can programming be liberated from the Von Neumann's style? A functional style and its algebra of programs", Commun ACM, Vol. 21, No. 8, p. 613-641.

[BARE 1984] BARENDREGT, H. P., (1984), *The lambda calculus, its syntax and semantics*, 2nd Ed., North-Holland.

[BARH 1953] BAR-HILLEL, Y., (1953), "A quasi-arithmetical notation for syntactic description", *Language* 29, p. 47-58; traduit dans *Languages* 9, 1968, p. 9-22.

[BERA 1993] BERARD, E. V., (1993), *Essays on Object-Oriented Software Engineering*, Addison-Wesley.

[BISK 1997] BISKRI, I., DESCLÉS, J.P., (1997), "Applicative and Combinatory Categorial Grammar (from syntax to functional semantics)", dans *Recent Advances in Natural Language Processing (selected Papers of RANLP 95)* Ed. Ruslan Mitkov & Nicolas Nicolov. John Benjamins Publishing Company, Numéro 136.

[BISK 2002] BISKRI, I., MEUNIER, J.G., (2002), "SATIM : Système d'Analyse et de Traitement de l'Information Multidimensionnelle", JADT 2002, St-Malo, France.

[BOOC 1991] BOOCH, G., (1991), *Object-Oriented Design*, Benjamin Cummings.

[BOOC 1994] BOOCH, G., (1994), *Object-oriented analysis and design with applications*, 2nd Ed., Benjamin/Cummings, California.

[BOUC 1994] BOUCHÉ, M., (1994), *La démarche objet : concepts et outils*, Afnor, Paris.

[BOUZ 1998] BOUZEGHOUB, M., GARDARIN, G., VALDURIEZ, P., (1998), *Les objets*, 2ème Ed., Eyrolles, Paris.
[BUDD 1987] BUDD, T. A., (1987), *A little Smalltalk*, Addison-Wesley, Reading, MA.

[BUDD 1997] BUDD, T. A., (1997), *An introduction to object-oriented programming*, 2nd Ed., Addison-Wesley.

[CHAI 1994] CHAIB-DRAA, B., (1994), "Distributed Artificial Intelligence: An overview", in A. Ken, J. G. Williams, C. M. Hall, and R. Kent, editors, Encyclopedia Of Computer Science And Technology, volume 31, pages 215-243, Marcel Dekker, Inc.

[CHAI 1996] CHAIB-DRAA, B., (1996), "Interaction between agents in routine, familiar and unfamiliar situations", International Journal of Intelligent and Cooperative Information Systems, 1(5):7-20.

[CHAI 2002] CHAIB-DRAA, B., JARRAS, I., (2002), "Aperçu sur les systèmes multiagents", dans Les cahiers de la série scientifique.

[CHUR 1941] CHURCH, A., (1941), Annals of mathematics studies, Vol. 6 : The *calculi of lambda conversion*, Princeton Univ. Press, Princeton, NJ. Reprinted by Klaus Reprint Corporation, New York, 1965.

[COAD 1991] COAD, P., YOURDON, E., (1991), *Object-Oriented Analysis*, 2nd Ed., Prentice-Hall.

[COUT 2003] COUTURIER, A., JEAN-BAPTISTE, G., (2003), *Programmation fonctionnelle : specifications & applications*, Cépaduès Editions.

[COX 1984] COX, B. J., (1984), Message / object programming : An evolutionary change in programming technology, *IEEE software*, January, 1984: 50-61. Also in *Tutorial : Object oriented programming*, Vol. 1, Concepts edited by G.E. Peterson (1987), p. 150-161. Washington, DC: Computer Society Press.

[COX 1986] COX, B. J., (1986), *Object-Oriented Programming*, Addison-Wesley.

[CURR 1958] CURRY, H. B., FEYS, R., (1958), *Combinatory logic*, Vol. 1, North-Holland.

[CURR 1967] CURRY, H. B., (1967), "Logic, Combinatory", *The Encyclopedy of Philosophy*, 4, New-York, p. 504-509.

[CURR 1972] CURRY, H. B., HINDLEY, J. R., SELDIN, J. P., (1972), *Combinatory logic*, Vol. 2, North-Holland.

[DAHL 1966] DAHL, O. J., NYGAARD, K., (1966), "Simula – An Algol-based simulation language", Comm. of the ACM, N9, V9, p. 671-678.

[DELI 2002] DELISLE, S., GARNEAU, T., (2002),"Programmation orientée-agent : évaluation comparative d'outils et environnements", dans Journées Francophones sur l' Intelligence Artificielle Distribuée et les Systèmes Multi-Agents (JFIADSMA'2002), Hermès, Lille, France, 28-30 octobre 2002.

[DESC 1990] DESCLÉS, J. P., (1990), *Langages applicatifs, langues naturelles et cognition*, Hermès, Paris.

[DESF 1997] DESFRAY, P., (1997), *Modélisation par objets : la fin de la programmation*, Masson, Paris.

[ECKE 1996] ECKERT, G., (1996), "Langages purement fonctionnels modernes", dans *Génie logiciel : principes, méthodes et techniques*, de STROHMEIER, A., BUCHS, D., 1996, 1ère Ed., Presses polytechniques et universitaires romandes, p. 103-135.

[EISE 1987] EISENBACH, S., (1987), *Functional programming : languages, tools and architectures*, New York : Wiley.

[ELLI 1990] ELLIS, M. A., STROUSTRUP, B., (1990), *The annotated C++ reference manual*, Reading, MA : Addison-Wesley.

[FERB 1995] FERBER, J., (1995), *Les systèmes multi-agents, vers une intelligence collective.*
InterEditions.

[FIKE 1971] FIKES, R., NILSSON, N., (1971), "A new approach to the application of theorem proving to problem solving". *Artificial Intelligence*, 2(3/4):189-208.

[FITC 1974] FITCH, F. B., (1974), *Elements of combinatory logic*, Yale Univ. Press.

[FRAN 1997] FRANKLIN, S., GRAESSER, A., (1997), "Is it an agent, or just a program?: A taxonomy for autonomous agents", in J. P. Mueller, M. Wooldridge, and N. R. Jennings, editors, *Intelligent Agents III: Theories, Architectures, and Languages (LNAI Volume 1193),* pages 21-35. Springer-Verlag: Heidelberg, Germany.

[FREG 1892] FREGE, G., (1892), "Über sinn und bedeutung", Zeitschrift für philosophie und philosophisches kritik, Vol. 100, p. 25-50.

[FREG 1893] FREGE, G., (1893), *Grundgesetze der arithmetik, begriffsschriftlich abgeleitet*, Band. 1 Jena (1893); Band 2 Jena (1903); traduction en anglais par M. Furth : *The basic laws of arithmetic, exposition of the system*, 1964, Univ. of California Press.

[GAUT 1996] GAUTIER, M., MASINI, G., PROCH, K., (1996), *Cours de programmation par objets : principes et applications avec Eiffel et C++*, Masson, Paris.

[GOSL 1996] GOSLING, J., JOY, B., STEELE, G., (1996), *The Java language specification*, Addison-Wesley, Reading, MA.

[HEIJ 1967] HEIJENOORT (van), J., (1967), *From Frege to Gödel, a source book in mathematical logic, 1879-1931*, Harvard Univ. Press.

[HEND 1980] HENDERSEN, P., (1980), *Functional programming : application and implementation*, Prentice-Hall, Englewood Cliffs, NJ.

[HIND 1986] HINDLEY, J. R., SELDIN, J. P., (1986), *Introduction to Combinators and Lambda-Calculus*, Cambridge Univ. Press.

[HUSS 1913] HUSSERL, E., (1913), *Logische Untersuchungen*, Max Niemeyer, Halle.

[JACO 1992] JACOBSON, I., (1992), *Object-Oriented Software Engineering*, Addison-Wesley.

[JENN 1998] JENNINGS, N. R., WOOLDRIDGE, M., SYCARA, K., (1998), "A roadmap of agent research and development", Int Journal of Autonomous Agents and Multi-AgentSystems, 1(1):7- 38.

[KAY 1976] KAY, A., GOLDBERG, A., (1976), *Smalltalk-72 instruction manual*, Technical report SSL-76-6, Xerox Parc, Palo Alto, California.

[LAMB 1958] LAMBEK, J., (1958), "The mathematics of sentence structure", *American Mathematical Monthly*, 65, p. 154-165.

[LAMB 1961] LAMBEK, J., (1961), "On the calculus syntactic types", *Proceeding of symposia in applied mathematics*, Vol. XII, America Mathematical Society, Providence, Rhode Island, p. 166-178.

[LAMB 1986] LAMBEK, J., SCOTT, P. J., (1986), *Introduction to higher order categorical logic*, Cambridge Univ. Press.

[LESN 1989] LESNIEWSKI, S., (1989), *Sur les fondements de la mathématique. Fragments (Discussions préalables, méréologie, ontologie)*. Traduit du polonais par Georges Kalinowski, Hermès.

[LISK 1975] LISKOV, B. H., ZILLES, S. N., (1975), "Specification techniques for data abstractions", IEEE *Transactions on software engineering*, SE-1, N 1, p. 7-18.

[LONG 1989] LONCHAMP, J., (1989), *Les langages de programmation : concepts essentiels, évolution et classification*, Masson, Paris.

[LOUD 1993] LOUDEN, K. C., (1993), *Programming languages : Principles and practice*, Boston : PWS.

[LUCK 2004] LUCK, M., ASHRI, R., (2004), *Agent-based software development*, Artech House.

[MEYE 2000] MEYER, B., (2000), *Conception et programmation orientées objet*, Eyrolles.

[MICH 1989] MICHAELSON, G., (1989), *An introduction to functional programming through lambda calculus*, Wokingham, UK : Addison-Wesley.

[MILL 2004] MILLER, R., (2004), *C++ for artists : the art, philosophy, and science of object-oriented programming*, Biblio Distribution.

[MOUL 1996] MOULIN, B., CHAIB-DRAA, B., (1996), "An overview of distributed artificial intelligence", in G. M. P. O'Hare and N. R. Jennings, editors, Foundations of Distributed AI, pages 3-54, John Wiley & Sons: Chichester, England.

[MULL 1996] MÜLLER, J. P., WOOLDRIDGE, M., JENNINGS, N. R., (1996), *Intelligent Agents III, Agent Theories, Architectures, and Langages (ATAL),* in ECAI'96 Workshop (ATAL), Budapest, Hungary, August 12-13, Ed Springer Berlin.

[NEUM 1925] NEUMANN (von), J., (1925), "An axiomatisation of set theory", *Heijenoort*, 1967, p. 393-413.

[PEYT 1990] PEYTON-JONES, S. L., (1990), *Mise en œuvre des langages fonctionnels de programmation*, Masson et Prentice-Hall. Traduit de *The implementation of functional programming languages*, Prentice-Hall, Englewood Cliffs, NJ, (1987).

[PINS 1988] PINSON, L. J., WIENER, R. S., (1988), *An introduction to object-oriented programming and Smalltalk*, Addison-Wesley.

[PRAT 1995] PRATT, T., ZELKOWWITZ, M. V., (1995), *Programming languages : design and implementation*, 3rd Ed., Englewood Cliffs, NJ : Prentice-Hall.

[PRES 1997] PRESSMAN, R. S., (1997), *Software engineering : a practitioner's approach*, 4th Ed., McGraw-Hill.

[ROSS 1935] ROSSER, J. B., (1935), "A mathematical logic without variables"; part 1 *Annals of maths.* (2) 36, p. 127-150; part 2 *Duke Math.* J. 1, p. 328-355.

[RUSS 1995] RUSSELL, S. J., NORVIG, P., (1995), *Artificial Intelligence : A Modern Approach*, Prentice Hall.

[SCHO 1924] SCHÖNFINKEL, M., (1924), "Über die Bausteine der Mathematischen Logik", *Math. Annalen* 92, p. 305-15; traduction en anglais dans Heijenoort 1967 : "On the building blocks of mathematical logic", p. 355-366.

[SEBE 2002] SEBESTA, R. W., (2002), *Concepts of programming languages*, 5th Ed., Addison-Wesley.

[SHOH 1993] SHOHAM, Y., (1993), *Agent-Oriented Programming*, Artificial Intelligence, Vol. 60, No. 1, p. 51-92, Mars 1993.

[SOMM 1992] SOMMERVILLE, I., (1992), *Software engineering*, 4th Ed., Addison-Wesley, Reading, MA.

[STEE 1988] STEEDMAN, M., (1988), "Combinators and grammars", in Oehrle et alii, 1988, p. 417-442.

[STRO 1997] STROUSTRUP, B., (1997), *The C++ programming language*, 3rd Ed., Reading, MA : Addison-Wesley.

[SUN 1995] SUN, (1995), *About Java*, Mountain view, CA : Sun Microsystems.

[TAYL 1999] TAYLOR, D., (1999), *The keys to object technologie*, in Handbook of Object Technologie, Editor-in-chief : Saba ZAMIR, CRC Press.

[TUCK 2001] TUCKER, A., NOONAN, R., (2001), *Programming languages : principles and paradigms*, McGraw-Hill.

[WATT 2004] WATT, D., (2004), *Programming language design concepts*, John Wiley & Sons.

[WEXE 1981] WEXELBLAT, R. L. (ed.), (1981), *History of programming languages*, Academic Press, New York.

[WOOL 1995] WOOLDRIDGE, M., JENNINGS, N. R., (1995), "Agent Theories, Architectures, and Langages: a survey", in Wooldridge and Jennings eds. *Intelligent Agents*. Springer-Verlag, 1-22.

[WOOL 2000] WOOLDRIDGE, M., JENNINGS, N. R., (2000), "Agent-Oriented Software Engineering", in Handbook of Technology (ed. J. Bradshaw) AAAI/MIT Press.