UNIVERSITÉ DU QUÉBEC À MONTRÉAL

# CONCEPTION ET IMPLÉMENTATION D'UN LANGAGE DE DESCRIPTION DE DIAGRAMMES DE MATHÉMATIQUES DISCRÈTES ET D'INFORMATIQUE THÉORIQUE

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN MATHÉMATIQUES

PAR

MATHIEU BOURGEOIS

DÉCEMBRE 2011

UNIVERSITÉ DU QUÉBEC À MONTRÉAL
Service des bibliothèques

*Avertissement*

La diffusion de ce mémoire se fait dans le respect des droits de son auteur, qui a signé le formulaire *Autorisation de reproduire et de diffuser un travail de recherche de cycles supérieurs* (SDU-522 – Rév.01-2006). Cette autorisation stipule que «conformément à l'article 11 du Règlement no 8 des études de cycles supérieurs, [l'auteur] concède à l'Université du Québec à Montréal une licence non exclusive d'utilisation et de publication de la totalité ou d'une partie importante de [son] travail de recherche pour des fins pédagogiques et non commerciales. Plus précisément, [l'auteur] autorise l'Université du Québec à Montréal à reproduire, diffuser, prêter, distribuer ou vendre des copies de [son] travail de recherche à des fins non commerciales sur quelque support que ce soit, y compris l'Internet. Cette licence et cette autorisation n'entraînent pas une renonciation de [la] part [de l'auteur] à [ses] droits moraux ni à [ses] droits de propriété intellectuelle. Sauf entente contraire, [l'auteur] conserve la liberté de diffuser et de commercialiser ou non ce travail dont [il] possède un exemplaire.»

# REMERCIEMENTS

Un projet d'une telle envergure n'aurait pas été possible sans un soutien constant. Je ne peux donc suffisamment remercier mon directeur, M. Roger Villemaire, pour le support qu'il m'a fourni pendant ces trois années. Ce mémoire ne serait pas entre vos mains sans son aide.

Je voudrais aussi remercier M. Étienne Gagnon pour ses nombreuses idées qui ont façonné SDDL de manière significative, ainsi que pour les nombreuses gaffes conceptuelles qu'il m'a empêché de commettre.

Je ne peux évidemment passer sous silence les membres de la communauté LaTeX, particulièrement ceux de la conférence TUG 2010 qui m'ont accueilli malgré mon inexpérience et qui m'ont fourni de précieux commentaires et suggestions en rapport avec mon projet. Je voudrais remercier John Bowman pour son travail sur Asymptote ainsi que Karl Berry pour l'intérêt démontré envers mon projet et pour ses suggestions pour mon article. Évidemment, il est impossible de remercier les membres de LaTeX sans remercier au passage le grand gourou Donald Knuth, sans qui ce mémoire serait difficilement présentable.

Je remercie vivement la fondation de l'Université du Québec à Montréal ainsi que le Fonds Québecois de la Recherche sur la Nature et les Technologies pour leur support financier durant les différentes phases de mon projet. Merci d'avoir cru en mon projet ainsi qu'en moi-même.

Finalement, je voudrais remercier mes parents Suzie et Yves ainsi que ma conjointe Annie pour leur support inconditionnel ainsi que pour l'aide incroyable qu'ils m'ont fournie sans nécessairement s'en rendre compte.

TABLE DES MATIÈRES

LISTE DES FIGURES . . . . . . . . . . ix

RÉSUMÉ . . . . . . . . . . xi

INTRODUCTION . . . . . . . . . . 1

CHAPITRE I
PROBLÉMATIQUE ET OBJECTIFS . . . . . . . . . . 7

1.1 Description de la problématique . . . . . . . . . . 7

1.1.1 Les notes de cours : une mise en contexte . . . . . . . . . . 7

1.1.2 Domaine d'application élargi . . . . . . . . . . 8

1.2 Description des objectifs recherchés . . . . . . . . . . 9

1.2.1 Intégration avec LaTeX . . . . . . . . . . 9

1.2.2 Description de structures . . . . . . . . . . 10

1.2.3 Hiérarchie solide et extensible . . . . . . . . . . 10

1.2.4 Objectifs moindres . . . . . . . . . . 10

1.3 Analyse des solutions existantes . . . . . . . . . . 11

1.3.1 Paquets LaTeX standards . . . . . . . . . . 12

1.3.2 XY-pic . . . . . . . . . . 13

1.3.3 Metapost . . . . . . . . . . 14

1.3.4 Asymptote . . . . . . . . . . 15

1.3.5 Conclusion de l'analyse . . . . . . . . . . 17

1.4 Diagrammes représentatifs du domaine . . . . . . . . . . 18

1.4.1 L'approche de Knuth . . . . . . . . . . 19

1.4.2 Description des exemples et des concepts présents dans les diagrammes 20

CHAPITRE II
STRUCTURE ET SOLUTION INFORMATIQUE . . . . . . . . . . 21

2.1 Objectifs d'une structure dans une solution informatique . . . . . . . . . . 21

2.1.1 Les différentes composantes d'une solution informatique . . . . . . . 21

2.1.2 Considérations pour une méthodologie de développement . . . . . . 23

2.2 La programmation orientée-objet . . . . . . . . . . . . . . . . . . . . . . . . . 24

2.2.1 Concepts de bases . . . . . . . . . . . . . . . . . . . . . . . . . 24

2.2.2 Différence avec la programmation basée-objet . . . . . . . . . . . . 25

2.3 Méthode pour la conceptualisation d'une hiérarchie . . . . . . . . . . . . . . 25

2.3.1 Origine de la méthode . . . . . . . . . . . . . . . . . . . . . . . 26

2.3.2 Les différentes étapes de conceptualisation . . . . . . . . . . . . . . 26

2.4 Développement de la hiérarchie . . . . . . . . . . . . . . . . . . . . . . . . 27

2.4.1 Extraction des différents concepts . . . . . . . . . . . . . . . . . . 27

2.4.2 Éléments particuliers . . . . . . . . . . . . . . . . . . . . . . . . 27

CHAPITRE III
ÉLÉMENTS GRAPHIQUES . . . . . . . . . . . . . . . . . . . . . . . . . . 31

3.1 Courbes et segments en informatique . . . . . . . . . . . . . . . . . . . . . 31

3.2 Courbes de Bézier . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

3.2.1 Définition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

3.2.2 Définition alternative . . . . . . . . . . . . . . . . . . . . . . . . 33

3.2.3 Propriétés d'une courbe de Bézier . . . . . . . . . . . . . . . . . . 34

3.2.4 Désavantages des courbes de Bézier . . . . . . . . . . . . . . . . . 36

3.3 Splines cubiques . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

3.4 Le cas d'un cercle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

3.5 Structures informatiques dans les solutions déjà existantes . . . . . . . . . . 41

3.5.1 Postscript et PDF . . . . . . . . . . . . . . . . . . . . . . . . . . 42

3.5.2 Asymptote . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

CHAPITRE IV
THÉORIE DES LANGAGES INFORMATIQUES . . . . . . . . . . . . . . . . . 45

4.1 Théorie des langages et automates . . . . . . . . . . . . . . . . . . . . . . 45

4.1.1 Les éléments de base d'un langage (Aho, Sethi et Ullman, 1986) . . 46

4.1.2 Expressions régulières . . . . . . . . . . . . . . . . . . . . . . . . 46

4.1.3 Automates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

4.1.4 Lien entre langage régulier et automate fini . . . . . . . . . . . . . 51

4.2 Grammaires . . . . . 52
4.2.1 Définition d'une grammaire formelle . . . . . 52
4.2.2 Hiérarchie de Chomsky . . . . . 54
4.2.3 Utilité dans les langages de programmation . . . . . 55
4.3 SableCC . . . . . 56
4.3.1 Qu'est-ce que SableCC . . . . . 56
4.3.2 Avantages de SableCC . . . . . 56
4.4 Langage réflexif . . . . . 57
4.4.1 Définition d'un langage réflexif . . . . . 57
4.4.2 Avantages . . . . . 57
4.4.3 Désavantages . . . . . 58
CHAPITRE V
MISE EN OEUVRE DE LA SOLUTION . . . . . 59
5.1 Éléments de langage en SDDL . . . . . 59
5.1.1 Implémentation de la hiérarchie . . . . . 59
5.1.2 Grammaire . . . . . 60
5.1.3 Implémentation des différentes propriétés du langage . . . . . 61
5.2 Réalisation en Java . . . . . 63
5.2.1 Traversée de l'arbre grammatical et exécution . . . . . 63
5.2.2 Définition d'objets quelconques . . . . . 64
5.2.3 Gestion des erreurs . . . . . 65
5.2.4 Structures de support . . . . . 66
5.3 Accès à Postscript, PDF et LaTeX . . . . . 66
5.3.1 Postscript . . . . . 66
5.3.2 PDF . . . . . 67
5.3.3 LaTeX . . . . . 68
5.3.4 Génération du code Asymptote . . . . . 68
CHAPITRE VI
RÉALISATION DES DIAGRAMMES EN SDDL . . . . . 71
6.1 Analyse du code des diagrammes existants . . . . . 71

6.1.1 Rappel des objectifs cruciaux à observer . . . . . . . . . . . . . . . . 72

6.1.2 Diagramme de compilation . . . . . . . . . . . . . . . . . . . . . . . . . . 73

6.1.3 Diagramme de tableau . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

6.1.4 Diagramme de Venn . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

6.2 Présentation du produit final . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

6.2.1 Statistiques et documentation . . . . . . . . . . . . . . . . . . . . . . . 80

6.2.2 Présentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

6.2.3 Site web . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

6.3 Conclusion de notre analyse . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

CONCLUSION . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83

APPENDICE A
PRÉSENTATION DU PROJET POUR LE TEX USERS GROUP . . . . . . . . . 85

APPENDICE B
PRÉSENTATION DE LA GRAMMAIRE DE SDDL . . . . . . . . . . . . . . . . . 91

APPENDICE C
PRÉSENTATION DE LA HIÉRARCHIE . . . . . . . . . . . . . . . . . . . . . . 97

APPENDICE D
PRÉSENTATION DES DIAGRAMMES CHOISIS . . . . . . . . . . . . . . . . . 105

BIBLIOGRAPHIE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127

# LISTE DES FIGURES


| Figure | | Page |
|---|---|---|
| 1.1 | Code XY-pic pour un diagramme | 14 |
| 1.2 | Diagramme résultant du code XY-pic | 14 |
| 1.3 | Code Metapost pour dessiner un triangle | 15 |
| 1.4 | Diagramme résultant du code Metapost | 15 |
| 1.5 | Code Asymptote pour dessiner un carré avec un texte à chaque coin | 16 |
| 1.6 | Diagramme résultant du code Asymptote | 16 |
| 3.1 | Courbes de Bézier linéaire, quadratique et cubique | 34 |
| 3.2 | Cercle complet approximé par une b-spline composé de quatre courbes de Bézier | 38 |
| 3.3 | Quart de cercle approximé par une courbe de Bézier | 39 |
| 3.4 | Variation de la différence entre un arc de cercle standard et la courbe de Bézier dans son domaine de référence | 41 |
| 4.1 | Automate acceptant les mots formés d'une série de a suivi d'un b | 49 |
| 5.1 | Extrait de la grammaire de SDDL pour une instruction | 61 |
| 5.2 | Code Java et SDDL pour calculer une combinaison linéaire | 63 |
| 5.3 | Code réflexif pour créer une forme quelconque | 65 |
| 5.4 | Code pour générer du code Asymptote pour un cercle | 69 |
| 6.1 | Diagramme d'automate de compilation | 73 |
| 6.2 | Code pour le diagramme d'automate | 74 |
| 6.3 | Diagramme de tableau | 76 |
| 6.4 | Code pour le diagramme de tableau | 77 |
| 6.5 | Diagramme de Venn | 78 |

6.6 Code du diagramme de Venn . . . . . . . . . . . . . . . . . . . . . . . . . . 79

# RÉSUMÉ

Dans ce mémoire, nous analysons la description de diagrammes structurés dans le domaine des mathématiques discrètes et de l'informatique théorique. Cette analyse s'effectue à partir de méthodes établies du génie logiciel. Nous trouvons un ensemble de propriétés que nous recherchons dans une solution informatique nous permettant de créer ces diagrammes. Nous analysons aussi les outils classiques à partir des propriétés précédentes qui nous permettent de réaliser ces diagrammes dans un environnement LaTeX. Nous observons que ces outils ne sont pas aussi puissants qu'ils le pourraient en fonction des propriétés établies par notre analyse.

Par la suite, en nous basant sur les méthodes de représentation graphique des courbes et sur la théorie des langages informatiques, nous jetons les bases d'un langage de description. Nous présentons aussi un outil, développé en Java qui nous permet de décrire aisément nos diagrammes d'informatique théorique et de mathématiques discrètes à partir d'une hiérarchie d'objets claire et extensible. Nous avons implémenté les éléments essentiels de notre langage en `SDDL` (Structured Diagram Description Language). Nous l'avons aussi validé à l'aide d'exemples représentatifs tirés de sources d'autorité. Nous avons finalement présenté notre solution dans le cadre de la conférence `TUG` 2010. Nous l'avons aussi présenté avec un article dans la revue du `TeX Users Groups`, `TUG`.

Mots clés : diagrammes, informatique théorique, mathématiques discrètes, langage

# INTRODUCTION

Le logiciel de mise en page de document LaTeX est devenu une référence pour la création de documents mathématiques (American Mathematical Society, 2008). Plusieurs raisons suscitent cet intérêt : une structure naturelle permettant de représenter clairement ses idées, le formatage automatique de qualité et un résultat final de qualité professionnelle à un coût moindre.

Un traité mathématique peut contenir plusieurs types d'objets : textes, formules, graphiques, diagrammes, etc. Dans le cas présent, nous nous intéressons principalement à la représentation de diagrammes. Par diagramme, on entend un dessin qui possède une structure discrète sous-jacente (souvent combinatoire, mais pas obligatoirement). Un exemple simple serait un graphe ou un arbre.

À partir du moment où nous nous intéressons à ces diagrammes et à leur représentation, il devient intéressant de voir ce que LaTeX nous fournit afin de générer des structures de qualité. En effet, les diagrammes sont essentiels à la transmission effective des idées et des résultats en informatique et en mathématiques discrètes. De plus, la consistance nous oblige à chercher à faire nos diagrammes à partir d'un modèle cohérent. Entre autres, il nous faut conserver la même police dans les diagrammes et dans le texte, en plus d'avoir accès aux symboles mathématiques. Nous pourrions aussi importer des diagrammes extérieurs à LaTeX mais les résultats obtenus ne serait pas consistant avec le texte. Nous devons donc chercher à obtenir une méthode de représentation adéquate pour nos diagrammes.

Bien sûr, il existe déjà plusieurs solutions à ce problème. Il nous faut donc les analyser avec soin, avec des critères formels. Cela implique de répertorier les différents paquets LaTeX permettant de faire du dessin avec leurs différentes propriétés. Il faut ensuite

ajouter les logiciels qui utilisent LaTeX indirectement, donc qui n'en font pas partie, mais s'en servent tout de même. Cela nous donne une vue d'ensemble complète des solutions déjà existantes à notre problème.

Ensuite, avant de penser à analyser ces solutions, nous devons déterminer ce que nous voulons exactement. À cette fin, nous avons analysé le processus qu'a utilisé Donald Knuth, créateur de TeX à la base de LaTeX, afin de matérialiser son logiciel de formatage. Nous devons aussi utiliser un ensemble de contraintes à soumettre à nos différentes solutions. Il était essentiel d'utiliser LaTeX pour le formatage du texte, mais aussi pour la clarté d'écriture, la qualité de la documentation et la capacité de créer aisément des structures. Par la suite, il devient aisé et très formel d'analyser chacunes des différentes solutions proposées.

Malheureusement, cette analyse nous a permis de constater qu'aucune solution n'était suffisamment bonne pour satisfaire nos besoins. La plupart des paquets étaient trop simples, alors que plusieurs étaieentiernt composés de macros, rendant la vérification de code quasi impossible. Quelques solutions (telles que XY-Pic (Rose, 2008), Metapost (TeX Users Group, 2009) et Asymptote (Hammerlindl, Bowman et Prince, 2010a)), fournissaient de nombreuses idées intéressantes, mais aucune ne convenait parfaitement à nos besoins. Nous en sommes donc venus à la conclusion que la meilleure solution serait de créer notre propre solution, tout en réutilisant les idées développées par les solutions précédentes.

Afin de créer une solution robuste, nous avons décidé de créer une hiérarchie orientée objet afin de représenter les différentes structures du domaine des mathématiques discrètes et de l'informatique théorique. Cela se justifie par le fait que les hiérarchies objets, au-delà de leurs avantages, possèdent des méthodes de conceptualisation formelles et bien définies. Nous avons utilisé la méthode favorisée par James Rumbaugh (Rumbaugh et al., 1991), un des créateurs de UML (Object Management Group, 2009).

Ce processus demande en tout premier lieu une analyse du domaine afin d'obtenir les structures principales. Par la suite, les liens entre les différentes classes sont analysés en

fonction des besoins respectifs. Finalement, après plusieurs itérations et simplifications, cette méthode fournit une hiérarchie complète. Par contre, elle doit par la suite être adaptée aux spécificités du langage d'implémentation.

Notre hiérarchie a été développée autour de plusieurs concepts importants. Certains sont naturellement liés au domaine d'utilisation. Par exemple, un point est une structure naturelle pour la description de positions dans un diagramme. Le même raisonnement s'applique pour des formes géométriques de base comme le cercle, le carré et l'ellipse. Certains objets sont des simplifications de concepts communs. Par exemple, un point abstrait est un point qui peut être déterminé de manière absolue ou relative. Une forme est une classe de base représentant l'ensemble des objets qui peuvent être dessinés. Il existe aussi un dessin, qui représente une forme à un endroit particulier.

Bien que cela pourrait déterminer complètement notre domaine, afin d'avoir une hiérarchie complète, il faut inclure certains éléments de structure. Par exemple, bien que nous puissions déterminer assez aisément ce qu'est un cercle, la manière de le décrire pour être dessiné n'est pas aussi simple. Il en est de même pour des formes courbes plus complexes ainsi que des courbes de manière générale car un ordinateur ne peut pas représenter aisément toutes formes de chemin. Nous pouvons créer un concept général de chemin mais celui-ci doit se ramener à la réalité d'un ordinateur.

Cela est déterminé en analysant deux choses : le format de document PDF et la description de chemin d'Asymptote. Le premier utilise des splines cubiques comme élément de base pour décrire les courbes dans un document. C'est intéressant, mais limitatif. Asymptote, de son côté, prend une description complexe mais précise d'une courbe et en déduit un ensemble de splines cubiques qu'il est possible de placer dans un document PDF ensuite. Nous en concluons que le bon chemin à prendre est probablement d'implémenter comme structure de base des chemins tels que décrits par Asymptote avec une structure de splines cubiques.

Par contre, certaines questions théoriques surviennent lors de cette analyse. Le grand problème relève de la puissance d'expression des splines cubiques. En effet, nous verrons

que les splines cubiques sont puissantes, mais pas suffisamment puissantes pour décrire un cercle, une forme géométrique pourtant extrêmement simple. Nous verrons par contre par la suite qu'il est possible de décrire ce même cercle avec des splines cubiques avec une précision acceptable pour la création de documents papier, malgré son inexactitude.

Afin d'adapter la hiérarchie, un langage d'implémentation a dû être choisi. Étant donné les contraintes de temps, il fût décidé de réutiliser un langage de description de diagrammes déjà existant : Asymptote. L'utilisation de ce langage nous permet d'obtenir des primitives simples et un accès à LaTeX déjà déterminé, ce qui diminue de beaucoup la charge de travail. De plus, Asymptote nous permet de créer nos images résultantes dans les formats Postscript (Adobe Systems Incorporated, 2010a) et PDF (Adobe Systems Incorporated, 2010b) sans avoir à réaliser nous-mêmes du code de très bas niveau.

Évidemment, si Asymptote fournissait tout ce dont nous avions besoin, nous n'aurions pas besoin d'utiliser autre chose. Par contre, c'est un langage basé sur les structures (Hammerlindl, Bowman et Prince, 2010b) au lieu des objets. Cela implique que notre hiérarchie ne peut être construite à l'aide de l'héritage objet. Étant donné la complexité que cela engendrerait, un langage supplémentaire doit-être utilisé pour la création de la hiérarchie. Cela nous amène donc à choisir Java (Oracle, 2010). Une recherche de la compagnie TIOBE considère Java comme étant le langage le plus utilisé au moment de la présente rédaction (TIOBE Software, 2008).

Avant de considérer le développement d'un nouveau langage, la première chose à faire consiste à considérer ce à quoi nous voulons qu'il ressemble. Afin de réaliser cette étape, nous avons analysé notre base de diagrammes en donnant une description la plus proche possible d'un langage de programmation. En simplifiant chaque description autant que possible, nous avons pu extraire quelques concepts de base importants pour la description de nos diagrammes. De plus, certaines contraintes supplémentaires peuvent être ajoutées qui dépendent plus de l'implémentation que de la théorie. En quelque sorte, il faut, pour réaliser notre projet, nous baser sur des technologies qui ont fait leurs preuves.

Afin de développer notre langage, il nous faut des concepts pour le décrire. Cela s'effectue à l'aide de la théorie des langages et des automates. Plus particulièrement, nous utiliserons les grammaires afin de façonner un langage qui convient à nos besoins, tout en étant simple à analyser par un programme informatique à l'aide d'un logiciel spécialisé, SableCC (Gagnon, 1998). Nous verrons que les grammaires nous permettent de décrire complètement notre langage afin d'être analysé par ordinateur.

Une fois que nous avons notre langage final, il ne reste plus qu'à vérifier les résultats obtenus. Afin de boucler la boucle, nous avons réalisé la programmation de chacun des diagrammes que nous avions choisis au départ pour notre base de tests. Une fois ceux-ci réalisés, il ne restait plus qu'à analyser les différences entre les deux et voir si les résultats obtenus étaient satisfaisants. Cela nous permet de démontrer qu'il s'agit d'une solution naturelle et efficace étant donné que nous pouvons créer les objets de notre domaine pour produire des diagrammes de grande qualité.

Le reste de ce mémoire est structuré comme suit. Le chapitre 1 expose la problématique et les objectifs recherchés. Le chapitre 2 introduit la création de notre solution informatique et, plus spécifiquement, de la hiérarchie objet. Le chapitre 3 décrit la théorie mathématique derrière la description de courbes et segments par ordinateur. Le chapitre 4 expose les dessous de la théorie des langages, que nous utilisons afin de créer `SDDL`. Le chapitre 5 expose la création du langage au-dessus de la hiérarchie. Finalement, le chapitre 6 analyse le résultat final afin de voir si le tout répond aux objectifs définis au départ.

[Cette page a été laissée intentionnellement blanche]

# CHAPITRE I

# PROBLÉMATIQUE ET OBJECTIFS

Dans ce chapitre, nous nous étendrons sur une définition correcte et complète de la problématique ainsi que sur les objectifs que nous souhaitons atteindre. Nous verrons aussi une analyse des solutions qui existent déjà et les raisons qui nous poussent à développer notre propre solution. Finalement, nous définirons une base de diagrammes aptes à représenter adéquatement nos objectifs.

## 1.1 Description de la problématique

### 1.1.1 Les notes de cours : une mise en contexte

Ce projet fut motivé principalement par la rédaction de notes de cours. En effet, dans un milieu universitaire (particulièrement pour les étudiants au baccalauréat, mais également pour les élèves de cycles supérieurs), il est extrêmement utile de pouvoir utiliser des notes de cours de qualité.

La création de notes de cours adaptées vient tout de même avec quelques problèmes. Premièrement, surtout en informatique, la plupart des livres sont écrits en anglais. Cela implique souvent de devoir écrire nos propres versions sous la forme de notes de cours. De plus, l'informatique est un domaine qui évolue à un rythme fulgurant. Cela implique que des notes adéquates et adaptées doivent être à jour. Comme certains éléments en informatique peuvent évoluer à chaque année, voire plus vite, il faut être capable d'écrire extrêmement rapidement des textes si nous souhaitons qu'ils soient utiles un tant soit

peu.

Cela nous amène à notre problématique principale : plusieurs cours nécessitent un support plus complexe que simplement des mots, à savoir des dessins et des diagrammes, voire des animations. Cela est requis dans plusieurs cours en informatique théorique, particulièrement dans l'étude des structures de données et des algorithmes. Il nous faut donc une méthode rapide et efficace pour créer des diagrammes à placer dans nos notes de cours.

### 1.1.2 Domaine d'application élargi

Bien évidemment, cibler purement les notes de cours et les structures de données n'est qu'un point de départ. Nous pouvons élargir notre domaine d'application à des diagrammes structurés, c'est-à-dire un dessin qui possède une structure sous-jacente cruciale, au point de pouvoir n'être décrit qu'à partir de cette structure. Par exemple, un graphe est un ensemble de sommets étiquetés reliés ensemble par des flèches. Tout dessin de graphe est donc un diagramme selon notre définition. D'un autre côté, une peinture possède parfois une structure, mais celle-ci est largement insuffisante pour décrire l'oeuvre. Par exemple, personne n'oserait décrire un Picasso comme étant un amas de cercles et de carrés de différentes couleurs.

L'ensemble des structures que nous visons se rapporte donc essentiellement à deux domaines : l'informatique théorique et les mathématiques discrètes. L'informatique théorique est surtout définie par les structures de données et les algorithmes (et tout sujet s'y reliant). Les mathématiques discrètes sont surtout définies par la théorie des graphes et la combinatoire. Nous négligeons ici volontairement les mathématiques continues pour deux raisons. La première est qu'une structure de taille infinie peut souvent être représentée par une structure finie (donc discrète). La seconde est que les besoins en mathématiques continues sont largement différents des besoins en mathématiques discrètes. En effet, dessiner le graphe d'une fonction de deux variables est un sujet complexe, mais totalement différent de créer un graphe. Afin de rendre nos besoins plus

spécifiques, nous éviterons ce genre de dessins, qui possèdent déjà de nombreux logiciels pour les réaliser (entre autres, Maple) (Maplesoft, 2010).

## 1.2 Description des objectifs recherchés

La première partie de notre travail consiste à décrire les objectifs que nous souhaitons atteindre dans notre création de diagrammes. En effet, plusieurs paramètres importants peuvent être utilisés. Il est donc important de ne pas perdre de vue l'ensemble tout en donnant la priorité à certains éléments.

### 1.2.1 Intégration avec LaTeX

LaTeX est un langage de description de documents texte. Créé à l'origine sous la forme de TeX par Donald Knuth (Knuth, 1984), puis modifié par Leslie Lamport (Lamport et Bibby, 1986), celui-ci est rapidement devenu un standard dans la rédaction d'articles scientifiques dans plusieurs domaines. Bien qu'utilisé surtout pour la rédaction d'articles, il a été prévu au départ pour la rédaction de livres. Il se distingue par sa capacité à créer aisément du contenu mathématique et le fait de s'occuper du contenu en laissant le logiciel s'occuper de la mise en page du texte. Il est si populaire que de nombreux mémoires, incluant celui-ci, sont écrits à partir de LaTeX.

Étant donné que le monde des mathématiques et de l'informatique théorique utilise abondamment LaTeX, il est important qu'un logiciel de création de diagrammes permette de l'utiliser. En effet, si notre objectif est de créer du contenu de qualité professionnelle, il faut que les polices soient constantes au travers de notre article (ou de nos notes de cours). Naturellement, plusieurs diagrammes vont contenir du texte. Il faut donc, pour être consistant, générer ce texte avec LaTeX afin d'avoir une forme standard pour l'ensemble du document. De plus, il est tout à fait possible de vouloir mettre des symboles mathématiques dans un diagramme (par exemple, un symbole d'intégrale). Si le texte est généré avec LaTeX, cela vient automatiquement. Sinon, il devient très difficile d'obtenir un texte de belle qualité. Il s'agit donc d'un besoin primordial.

### 1.2.2 Description de structures

Un autre élément extrêmement important que nous recherchons est de pouvoir faire adéquatement la description de différentes structures. Étant donné que les diagrammes sont composés de structures, il faut pouvoir en créer aisément et les relier de manière aisée. Cela implique quelques petites contraintes, entre autres de pouvoir utiliser des variables pour conserver les structures en question. Idéalement, la solution la plus solide en terme de description de structures utiliserait l'orienté-objet. Une des composantes les plus importantes d'une solution orientée objet est la présence d'une hiérarchie de classes d'objets.

### 1.2.3 Hiérarchie solide et extensible

Un objectif que nous recherchons est que le langage nous fournisse une hiérarchie d'objets (ou de structures) de qualité afin de pouvoir créer une grande quantité de diagrammes à partir d'éléments de cette hiérarchie. Une hiérarchie peut être décrite comme étant une structure permettant d'intégrer facilement de nouveaux aspects sans avoir à dupliquer inutilement du code. Il serait aussi important qu'elle puisse être étendue si certains éléments supplémentaires sont nécessaires. Par exemple, il pourrait être intéressant pour des gens en génie électrique de pouvoir faire des diagrammes électriques, malgré le fait que cela ne fait pas partie de nos contraintes de départ. Cela pourrait être ajouté par extension du système.

### 1.2.4 Objectifs moindres

#### Gestion des erreurs

Dans des notes de cours, il est fréquent d'avoir des diagrammes. Par contre, écrire un diagramme peut s'avérer plus compliqué que d'écrire du texte. Comme plusieurs utilisateurs potentiels ne seront pas des experts du système, il faut que les erreurs soient détectées et traitées de manière adéquate. Cela assure à l'utilisateur final une expérience

plaisante en spécifiant les erreurs qu'il commet et des méthodes pour les réparer. S'il faut que l'utilisateur cherche pendant des heures pour trouver la petite erreur qu'il a commise, il va tout simplement cesser d'utiliser le logiciel.

Langage fluide et clair

Bien que moins important, le fait d'avoir un langage clair est excellent pour attirer les utilisateurs vers ce système. En effet, si le langage est un gribouillis incompréhensible, il sera fort probablement dur d'apprentissage et difficile à aborder. Par fluide et clair, nous entendons que le langage soit, jusqu'à un certain point, compréhensible pour quelqu'un qui ne connaît pas le système.

Qualité de la documentation et courbe d'apprentissage

Finalement, il faut qu'une documentation existe et qu'elle soit complète. De préférence, une bonne documentation contiendra un répertoire de toutes les fonctionnalités, mais aussi des tutoriels avec des exemples codés de cas standards. Cela incite les utilisateurs à essayer le logiciel en prenant un exemple et en l'adaptant à leurs besoins. Cela amène (avec les autres objectifs) à un langage simple d'apprentissage, mais tout de même puissant. Comme notre objectif est de trouver une solution simple, il faut que n'importe qui puisse commencer à s'en servir rapidement, sans avoir besoin de lire l'ensemble de la documentation.

## 1.3 Analyse des solutions existantes

Naturellement, une fois que nous savons ce que nous recherchons, il nous faut analyser toutes les solutions qui existent déjà. Cela nous permettra d'identifier une solution qui est déjà fonctionnelle ou, au mieux, une qui pourra être étendue ou utilisée afin de simplifier le travail de création, si besoin est. Au minimum, nous pourrons extraire les concepts intéressants de chaque solution et voir s'il est possible de les intégrer à notre solution.

### 1.3.1 Paquets LaTeX standards

Étant donné l'importance de l'utilisation de LaTeX, les premières solutions sont celles disponibles à l'intérieur même de LaTeX. Celles-ci sont appelées des paquets. Plusieurs paquets différents existent, et bon nombre d'entre eux sont des solutions bricolées, et donc incomplètes.

Dans les solutions intéressantes, on note premièrement la présence de la commande \picture. Celle-ci a été ajoutée dans TeX par Donald Knuth, comme une solution à tout oubli potentiel dans son système. Cette commande prend du texte quelconque et le passe par la suite au pilote qui compile le document. Cela permet entre autres de transférer des informations au pilote qui compile en Postscript et permet donc l'utilisation de toute la force de Postscript dans la rédaction de dessin. Un gros problème de cette solution est que le texte à écrire dépend du pilote utilisé, limitant de beaucoup son utilisation. Il est tout de même devenu la base de nombreux paquets qui se servent de Postscript.

Plusieurs solutions ont aussi été développées pour toutes sortes de domaines. On retrouve entre autres des diagrammes de Feynmann et de Lewis (`feyn` et `lewis`), des diagrammes de flots (`nassflow`) et des arbres linguistiques (`xyling`, `rst`, `qtree` et `lingtrees`). Naturellement, ces solutions ne conviennent pas car elles sont pour des domaines trop spécifiques par rapport à notre objectif.

Une solution d'un domaine plus général qui se démarque est le paquet `PSTricks` (Van Zandt, 1993a). Créé en 1993 par le professeur Timothy Van Zandt, celui-ci est devenu très populaire au fil des années étant donné les nombreuses possibilités qui existent en Postscript. Un des grands avantages est sa documentation qui est très prolifique et très claire (Van Zandt, 1993b).

Cela est bien intéressant mais, malheureusement, `PSTricks` souffre du même problème que chaque paquet interne à LaTeX. Il est fait à partir de macros. En effet, TeX a été écrit complètement à partir de macros. Il s'agit d'un mécanisme de remplacement textuel, où l'on définit une entrée et toutes les occurrences de cette entrée sont remplacées par une

sortie bien spécifique. Comme une macro est toujours remplacée par son contenu avant d'être analysée plus en détail, toute forme d'erreur se retrouve à un niveau extrêmement bas. Cela fait que n'importe quelle erreur est quasi impossible à cerner correctement, rendant le travail de l'utilisateur qui souhaite apprendre très complexe. De plus, le fait de fournir le Postscript directement à l'utilisateur rend celui-ci très puissant, mais très dangereux. Finalement, bien que de nombreuses formes soient disponibles, il n'existe pas de hiérarchie en tant que telle dans `PSTricks`, ce qui le rend impossible à étendre.

### 1.3.2 XY-pic

Une autre solution intéressante peut être retrouvée à l'intérieur de LaTeX. Il s'agit de `XY-pic`, un paquet écrit par Kristoffer H. Rose (Rose et al., 2010). Ce paquet a été créé spécifiquement pour la description de diagrammes à l'intérieur de LaTeX et possède une documentation exhaustive. Malheureusement, cette documentation est extrêmement complexe, même pour un utilisateur avancé. `XY-pic` utilise principalement des boîtes LaTeX positionnées à différents endroits afin de créer les diagrammes. Ceux-ci sont par la suite reliés par différents objets, puis décorés. Cela fournit un résultat final de qualité. `XY-pic` possède entre autres un mode pour créer des diagrammes en positionnant les éléments dans une matrice. Cela rend les dessins plus simples conceptuellement, mais pas toujours positionnés adéquatement. `XY-pic` permet aussi de faire du positionnement relatif à d'autres objets, ce qui peut être très pratique. Comme nous pouvons le voir dans l'exemple suivant, `XY-pic` est très pratique dans les résultats qu'il nous permet d'obtenir, mais sa syntaxe laisse de beaucoup à désirer.

Par contre, les défauts de `XY-pic` sont extrêmement visibles. En premier lieu, la syntaxe du langage est extrêmement difficile à comprendre, car elle est basée sur la composition de nombreux caractères spéciaux. De plus, comme il s'agit d'un paquet LaTeX, le tout est implémenté à l'aide de macros et rend donc la recherche d'erreur laborieuse (surtout lorsqu'on prend en compte la syntaxe déjà difficile à utiliser). Cela rend `XY-pic` inapte à l'utilisation par quelqu'un qui ne souhaite pas investir trop de temps dans son apprentissage. Finalement, les nombreux modes de dessins sont souvent incompatibles

```
\xymatrix{
 U \ar@/_/[ddr]_y \ar@/^/[drr]^x
   \ar@{.>}[dr]|-{(x,y)}
 & X \times_Z Y \ar[d]^q \ar[r]_p
 & X \ar[d]_f
 & Y \ar[r]^g
 & Z
\\
\\
}
```

**Figure 1.1** Code XY-pic pour un diagramme

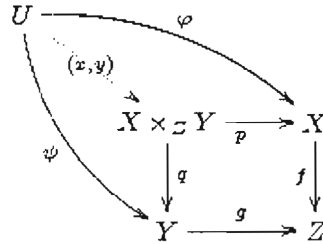



**Figure 1.2** Diagramme résultant du code XY-pic

entre eux et ne peuvent souvent pas être utilisés dans un même dessin.

### 1.3.3 Metapost

Étant donné le problème commun à tous les paquets à l'intérieur de LaTeX, la solution la plus logique serait d'essayer d'en sortir. Cela a donné le logiciel `Metapost` (Hobby, 1992). Celui-ci a été inspiré de `Metafont` (Knuth, 1986), que Knuth a utilisé pour dessiner des polices pour TeX. Une des grandes avancées du langage est la description de courbes. Celui-ci réutilise un algorithme de Knuth qui permet de décrire des courbes tout en en contrôlant les différents paramètres. De plus, comme le logiciel n'est pas rédigé en macros, il permet de détecter de manière adéquate les erreurs de l'usager. Nous pouvons déjà voir dans le code fourni en exemple qu'il s'agit d'un langage beaucoup plus clair que `XY-pic`.

```
beginfig(1)
    pair A, B, C;
    A:=(0,0); B:=(1cm,0); C:=(0,1cm);
    draw A--B--C--cycle;
endfig;
```

**Figure 1.3** Code Metapost pour dessiner un triangle

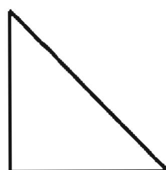


**Figure 1.4** Diagramme résultant du code Metapost

Par contre, quelques problèmes se dégagent de `Metapost`. Premièrement, le langage ne permet pas l'implémentation d'une hiérarchie solide. Cela ne serait pas un problème s'il en existait déjà une, mais ce n'est pas le cas. En effet, les diagrammes ont fait l'objet d'une extension du langage, mais rien de très développé. Il est donc faisable de faire des diagrammes, mais il ne s'agit pas nécessairement d'une solution à toute épreuve. De plus, comme toutes les coordonnées du langage sont données en valeur absolue, cela limite les possibilités du langage (entre autres, en ne pouvant utiliser des points définis par des objets).

### 1.3.4 Asymptote

Une autre possibilité se retrouve dans le logiciel `Asymptote`, qui se définit comme étant un langage inspiré de `Metapost` (Hammerlindl, Bowman et Prince, 2010b). Il est donc aussi externe à LaTeX, mais aussi inspiré du langage de programmation `C++`. Cela lui confère la création de structures avancées et la création de fonctions complexes. De plus, une documentation exhaustive et un traitement des erreurs de qualité en font une solution robuste.

Malheureusement, comme `Metapost`, `Asymptote` pose quelques problèmes. En effet, ce

```
size(3cm);
draw(unitsquare);
label("$A$",(0,0),SW);
label("$B$",(1,0),SE);
label("$C$",(1,1),NE);
label("$D$",(0,1),NW);
```

**Figure 1.5** Code Asymptote pour dessiner un carré avec un texte à chaque coin

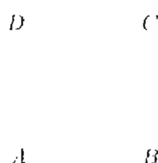


**Figure 1.6** Diagramme résultant du code Asymptote

logiciel fut écrit avec les mathématiques continues et la géométrie comme domaine d'utilisation. Ce n'est pas un problème en tant que tel (`Asymptote` le fait très bien), mais cela fait que la création de diagrammes discrets n'est pas vraiment développée. Il faudrait donc être capable de créer une hiérarchie de structures pour la création de diagrammes. Étant donné le support des structures possédant des fonctions internes, il devrait être possible d'ajouter cette possibilité à Asymptote.

Malheureusement, ce n'est pas présentement réalisable à l'intérieur du langage. En effet, `Asymptote` est un langage basé objet. Cela signifie qu'il possède des structures qui ressemblent à une description typique d'un objet, mais pas toutes les propriétés importantes qui se rattachent à un langage à objets standard, comme `Java` ou `C++`. Plus particulièrement, c'est l'absence de polymorphisme qui cause problème. En effet, la création d'une hiérarchie implique de créer des liens entre les différentes structures. Par exemple, un cercle, un triangle et un nuage sont toutes des formes avec certaines propriétés similaires. Il nous faut donc avoir des fonctions qui prennent des formes comme paramètres. Si nous avons une fonction $f$ qui prend une forme, nous souhaiterions écrire une seule fonction :

```
f(forme){...};
```

Malheureusement, en `Asymptote`, il faudrait écrire une fonction pour chaque forme distincte, comme cela :

```
f(cercle){...};
f(carre){...};
f(nuage){...};
```

Cela rend la création d'une hiérarchie extrêmement difficile, car pour chaque fonction utilisant un objet plus général, il nous faut écrire plusieurs fois cette fonction. De plus, cela rend l'expansion par l'usager de la hiérarchie essentiellement impossible car il doit rajouter des cas pour chaque fonction définie précédemment en plus d'ajouter son nouvel objet dans la hiérarchie. Cela est difficilement souhaitable.

### 1.3.5 Conclusion de l'analyse

Afin d'en venir à une conclusion adéquate, il faut se ramener aux objectifs recherchés. Cela nous permet de rapidement éliminer les solutions internes à LaTeX, étant donné la complexité de la gestion des erreurs (et, indirectement, la courbe d'apprentissage du langage). Cela nous laisse comme solution `Metapost` et `Asymptote`. Naturellement, le langage `Asymptote` est plus évolué que celui de `Metapost` et fournit des éléments de programmation plus intéressants. Cela implique donc qu'il nous permet plus aisément de construire nos structures tant désirées.

Par contre, `Asymptote` ne fournit pas une solution suffisante étant donné qu'il ne possède pas une hiérarchie d'objets adaptée aux diagrammes. De plus, étant donné que le langage est basé objet, il sera beaucoup trop compliqué d'y implanter une hiérarchie. Nous en venons donc à la conclusion que le mieux serait d'utiliser `Asymptote`, qui nous fournit un ensemble de services complexes (implémentation des chemins, liens avec LaTeX, etc...) et de l'implémenter avec un langage de plus haut niveau.

`Java` devient un choix intéressant pour cette tâche. En effet, c'est un langage connu dans l'industrie et simple d'apprentissage. Il s'agit d'un langage de programmation objet qui nous permettra à coup sûr d'implémenter une hiérarchie solide. Par contre, ce langage possède quelques petits problèmes, comme un très haut niveau de verbosité et beaucoup d'éléments non requis pour la description de diagrammes.

Nous en sommes donc venus à la conclusion que la meilleure solution serait d'utiliser `Java`, mais d'implémenter par-dessus celui-ci un langage dédié, qui éliminerait tout ce qui n'est pas requis pour la description des diagrammes. Nous appellerons ce langage `SDDL`, pour `Structured Diagram Description Language`.

#### Concepts intéressants à conserver

Avant de penser à concevoir notre logiciel, il serait bon de voir quels éléments nous souhaitons conserver des solutions existantes. À cette fin, l'élément le plus intéressant provient de `XY-Pic`. En effet, il permet de relier des objets entre eux par des flèches d'une manière relativement simple. Il serait donc intéressant de pouvoir permettre de décrire des liens entre différents objets par des points spécifiques à chacun d'entre eux.

De plus, l'algorithme de description de chemins utilisé par `Metapost` et `Asymptote` est extrêmement efficace et utile et pourrait donc être utilisé directement par notre langage. Cela sera simple à implémenter étant donné que nous souhaitons utiliser `Asymptote` pour compiler vers `Postscript`.

### 1.4 Diagrammes représentatifs du domaine

Finalement, avant de pouvoir commencer à définir les besoins de notre hiérarchie, il nous faut obtenir une définition valide et complète de notre domaine. Naturellement, il n'est pas évident de trouver une définition concrète et objective d'un diagramme en informatique théorique. Nous devons donc nous rabattre sur une autre solution, qui doit tout de même être la plus formelle possible.

### 1.4.1 L'approche de Knuth

Pour trouver une idée, nous nous ramènerons à la méthode que Donald Knuth a utilisée lorsqu'il a créé TEX. En effet, il souhaitait créer un logiciel qui ferait de « beaux » documents. Cela étant un terme très subjectif, Knuth décida de se définir un ensemble de journaux scientifiques dont les publications étaient du plus haut standard de qualité pour l'époque (Knuth, 1999). Après avoir obtenu ces différents exemples, il put s'affairer à essayer de reproduire une solution de cette qualité. Il devient donc évident, à la fin du cycle de développement du logiciel, que sa solution était cohérente et valide car elle permettait d'obtenir un résultat de qualité, en comparaison à ce que nous pourrions déjà définir comme étant de qualité.

Nous pouvons donc chercher à suivre la même méthode. Afin de la respecter, nous devons trouver quelques livres d'un haut niveau de qualité et prendre certains diagrammes que l'on retrouve à l'intérieur. Cela nous permettra de retrouver les éléments les plus cruciaux de notre domaine.

Plusieurs livres intéressants s'offrent à nous : nous utiliserons entre autres « The Art of Computer Programming » de Donald Knuth (Knuth, 1968). Il est naturel qu'un des livres les plus connus en informatique, écrit par Knuth de surcroît, possède un haut standard de qualité. Un autre livre important est le « livre du dragon » (Aho, Sethi et Ullman, 1986), qui étudie la théorie de la compilation et est extrêmement connu dans son domaine. De plus, celui-ci renferme l'étude de nombreuses structures de données, ce qui nous fournit de nombreux diagrammes de qualité. Finalement, il existe aussi un livre très connu dans l'étude des algorithmes (ce qui est aussi extrêmement lié à l'étude des structures de données) qui s'appelle « Introduction to Algorithms » (Cormen et al., 2001). Ces trois livres forment une base extrêmement connue et de qualité pour nos diagrammes. Nous avons extrait quinze (15) différents diagrammes de ces livres.

### 1.4.2 Description des exemples et des concepts présents dans les diagrammes

Les différents diagrammes que nous avons sélectionnés représentent différentes structures de données. On y retrouve plusieurs arbres et graphes. Les graphes sont de plusieurs types, allant d'un graphe standard à des graphes d'exécution. De plus, certaines structures présentes plus rarement se retrouvent dans notre ensemble. Entre autres, on retrouve un diagramme de Venn qui, bien que conceptuellement simple, est toujours relativement complexe lorsque nous souhaitons le réaliser avec un logiciel. On retrouve aussi des arbres linguistiques qui ne sont pas nécessairement complexes, mais surtout longs à réaliser.

Cet ensemble de diagrammes nous permet de couvrir l'ensemble des structures de données importantes en informatique théorique et en mathématiques discrètes. Il existe d'autres structures de données, mais celles-ci peuvent toujours être implémentées séparément étant donné que nous souhaitons faire une hiérarchie aisément extensible. Nous remarquons surtout dans nos exemples qu'un des éléments les plus importants est de pouvoir relier convenablement les différents éléments (dans un graphe, par exemple). Comme il s'agit de quelque chose qui est toujours présent, il s'agira d'un des points les plus importants à développer pour notre hiérarchie.

# CHAPITRE II

# STRUCTURE ET SOLUTION INFORMATIQUE

Afin de développer un produit qui convient à nos besoins, nous aurons besoin qu'il soit structuré adéquatement. Cela implique principalement la construction de la hiérarchie d'objets du domaine. Il faut déterminer quels sont les éléments recherchés dans notre solution, trouver une méthode de développement formelle et adéquate, puis la réaliser correctement. Cela assurera une solution de qualité.

## 2.1 Objectifs d'une structure dans une solution informatique

En premier lieu, il nous faut justifier le besoin d'une méthode de développement formelle et solide. En effet, un logiciel en informatique est très différent d'un problème mathématique standard. Une démarche mathématique se ramène à un concept bien connu : la preuve. Cela rend la résolution du problème crédible, à condition que la preuve soit bonne. Par contre, nous verrons qu'un problème informatique ne peut être résolu à l'aide d'une simple preuve.

### 2.1.1 Les différentes composantes d'une solution informatique

Une solution à un problème informatique possède différents éléments. Chacun d'entre eux possèdent un certain degré de formalisme qui affecte l'ensemble de la solution.

## Algorithmes

Les algorithmes sont les éléments les plus formels d'une solution informatique. On peut dire qu'ils se ramènent à la partie mathématique d'un problème informatique, en quelque sorte. Composé d'une série d'opérations aux comportements bien définis, le résultat d'un algorithme est prévisible. Étant donné la possibilité de prévoir le comportement d'un algorithme à partir de ses entrées, il est possible de démontrer que l'algorithme fournit exactement les résultats recherchés. Il suffit en effet de prouver que, à chaque entrée possible, on obtient la sortie qui serait logiquement attendue de l'algorithme. On peut penser par exemple à un algorithme de tri qui, pour chaque ensemble de nombres, retourne les nombres en ordre croissant.

Malgré tout, un algorithme en informatique n'est pas parfait. En effet, si le temps d'exécution d'un algorithme pour des entrées standards est beaucoup trop élevé, il ne sera pas utilisé, peu importe qu'il donne le bon résultat. Le temps d'opération et la quantité d'espace mémoire sont les deux principales contraintes qui entrent en ligne de compte pour les différents algorithmes d'une solution informatique.

## Solutions technologiques

L'autre partie extrêmement importante de tout logiciel informatique est le choix des technologies utilisées. En effet, les programmes informatiques ne sont pas directement créés à partir de circuit électriques. Chaque couche d'abstraction supplémentaire (circuits logiques, code assembleur, programmation de haut niveau) limite les possibilités du programmeur et vient avec certains avantages et certains inconvénients. Le fait de ne pas utiliser une solution déjà existante peut permettre une solution plus puissante, mais à un coût de développement potentiellement beaucoup plus élevé. Il est aussi possible, avec un certain ensemble de choix technologiques, de ne pas pouvoir obtenir un effet recherché dans une solution. Cela explique pourquoi nous avons décidé d'utiliser `Asymptote`. Il nous fournit un accès à LaTeX déjà codé ainsi qu'une traduction complexe vers un fichier image `Postscript`. La réalisation de ces différents éléments de

code aurait significativement augmenté la complexité de notre tâche.

Imperfection de toute solution informatique

Étant donné les limitations des outils technologiques à notre disposition, nous sommes obligés de constater que toute solution informatique est imparfaite. De plus, plusieurs contraintes données à des systèmes informatiques sont généralement vagues et peuvent être interprétées différemment en fonction de l'utilisateur. Par exemple, nous souhaitons réaliser un outil simple d'utilisation, mais puissant et extensible. Naturellement, plus un langage est puissant, plus il devient complexe. Un utilisateur peut trouver un outil très simple alors qu'un autre le trouvera incompréhensible. D'autres points comme la qualité de documentation, la courbe d'apprentissage et la gestion aisée des erreurs sont des critères qui possèdent une bonne part de subjectivité. Ce ne sont toutefois pas nécessairement de mauvais critères. Il faut simplement chercher, selon une définition la plus proche possible d'un utilisateur moyen, à maximiser les différents paramètres.

Ce genre de contraintes relève généralement du génie logiciel. Bien que ce mémoire ne constitue pas un travail de génie logiciel, nous sommes obligés d'emprunter quelques éléments à ce domaine afin d'obtenir un résultat adéquat.

### 2.1.2 Considérations pour une méthodologie de développement

Avant de choisir notre méthodologie, il est bon de vérifier les objectifs recherchés dans l'utilisation d'une telle méthode de développement. Cela nous permettra de dégager les éléments importants à garder en tête tout au long du développement de la hiérarchie.

Un des éléments les plus importants est d'obtenir une structure efficace pour décrire nos diagrammes. L'autre élément crucial est d'obtenir une hiérarchie qui représente adéquatement le domaine. De préférence, elle doit être complète et d'un usage naturel. On entend par là que les éléments sont suffisamment précis et proches d'éléments standards et connus du domaine, que toutes les opérations soient « naturelles » pour une personne qui ne connaît pas le langage. Il n'existe malheureusement pas de mesure pour

déterminer si une hiérarchie est naturelle ou non. Toutefois, si chaque section est réfléchie de manière formelle, le résultat final devrait être naturel. Certaines considérations mineures peuvent être plus ou moins mesurées, comme la longueur du code et les noms adéquats de fonctions.

## 2.2 La programmation orientée-objet

La programmation orientée-objet est un des phénomènes très importants en informatique. Depuis sa création, elle est devenue de plus en plus populaire étant donné la facilité de développement dans une telle méthode. Le fait d'utiliser une solution orientée-objet (au contraire de `Metapost` ou `Asymptote`) ne s'est pas fait de façon arbitraire. Il est donc de mise d'exposer les différents avantages qui nous ont poussés à choisir un tel paradigme pour notre logiciel.

### 2.2.1 Concepts de bases

La structure de base en programmation orientée-objet est, naturellement, l'objet. Celui-ci est défini comme étant une structure de données qui regroupe des données et des méthodes (Saunders, 1989). Bien qu'il ne s'agit pas d'un critère obligatoire, la plupart des langages de programmation créent les objets à partir d'une classe, qui définit les différents champs et méthodes présents et leurs effets. Les objets forment donc le minimum requis pour avoir un langage objet.

En tant que tel, cet ensemble de concepts est assez minimaliste. Il existe plusieurs autres éléments qui sont présents, à un degré plus ou moins grand, dans les langages à objets. Ceux-ci forment donc, en quelque sorte, la base de la programmation orientée-objet (Pierce, 2002).

- L'encapsulation des données, c'est-à-dire camoufler les données dans une structure spécifique (l'objet).
- L'héritage, qui permet de définir un objet comme étant une instance spécifique d'un autre objet. On peut appliquer le même raisonnement à deux classes. On dit alors

qu'une classe hérite d'une classe de base. Celle-ci possède exactement le même corps que sa classe de base, mais peut rajouter ou modifier certaines propriétés. Il s'agit d'un des aspects les plus importants dans le développement d'une hiérarchie.

- Le polymorphisme, qui nous permet de créer des propriétés qui varient en fonction des sous-classes. Par exemple, une classe `Nombre` pourrait avoir une méthode d'addition, alors que ses sous-classes `Entières` et `Réelles` redéfiniraient cette méthode en fonction de leur implémentation spécifique. Cela nous permet par la suite de prendre deux nombres et de les additionner, peu importe qu'il s'agisse de réels ou d'entiers.

### 2.2.2 Différence avec la programmation basée-objet

Les différentes priorités énumérées ci-haut sont les éléments de base qui nous permettent de créer une hiérarchie de qualité. Il s'agit de la raison principale justifiant de ne pas utiliser `Asymptote` pour créer la hiérarchie. En effet, celui-ci est un langage basé-objet. La différence principale est qu'un langage basé-objet ne nécessite pas l'implémentation de l'héritage ou du polymorphisme (Saunders, 1989). `Asymptote`, de son côté, possède un héritage de structure minimal, mais pas de polymorphisme. Cette différence est cruciale car cet élément est nécessaire pour la création d'une hiérarchie dans des contraintes acceptables.

En effet, une chose importante est d'ajouter des objets dans les diagrammes. Par exemple, supposons que nous avons une classe `Conteneur`. Celui-ci doit pouvoir contenir des `Cercles`, des `Carrés`, des `Triangles`, etc. S'il fallait écrire une fonction pour ajouter chacune des formes, le travail deviendrait énormément complexe et impossible à étendre pour l'usager final. Avec le polymorphisme, il suffit d'écrire une fonction qui permet d'ajouter une `Forme` quelconque.

## 2.3 Méthode pour la conceptualisation d'une hiérarchie

Nous décrirons ici la méthode que nous avons utilisé. Il s'agit de `OMT` (pour `Object Modeling Technique`).

### 2.3.1 Origine de la méthode

Cette méthode fut développée principalement par James Rumbaugh (mais avec l'aide de plusieurs personnes) (Rumbaugh et al., 1991) en 1991. Celui-ci est principalement connu comme étant un des auteurs de la norme `UML`, qui est devenu le schéma de modélisation objet standard et utilisé par tous les programmeurs objets (Group, 1999). Il est à noter que cette méthode est une méthode d'analyse complète, pour toute la durée de vie du logiciel. Nous n'aurons par contre besoin que de la partie analyse de cette méthode.

### 2.3.2 Les différentes étapes de conceptualisation

La première partie de cette méthode a déjà été effectuée implicitement lorsque nous avons recherché nos diagrammes relatifs du domaine. Habituellement, il est suggéré d'écrire un texte résumant l'ensemble du domaine, mais dans notre cas, il est plus approprié de résumer notre domaine avec des diagrammes. Par la suite, il faut prendre le temps de faire ressortir les différentes classes d'objets qui font parties du domaine et prendre le temps de bien les définir. On éliminera les classes vagues ou redondantes par la suite.

La prochaine étape consiste à définir les différentes associations entre les classes. Par exemple, une classe `Employé` et une classe `Compagnie` seraient liées par l'association `travaille pour`. Par la suite, on identifie les attributs (ou les champs) des différentes classes, en évitant les attributs d'implémentation. Finalement, on raffine le modèle en ajoutant l'héritage des classes. Cela permet de simplifier significativement la hiérarchie.

Cela fournit une hiérarchie complète de notre domaine. Par contre, rien ne démontre qu'elle est correcte. Afin de la vérifier, il suffit d'analyser les différents cas typiques de notre problème, regarder le résultat obtenu et relever toute erreur. Dans notre cas, il s'agit de coder (seulement en terme de la hiérarchie) les différents diagrammes et vérifier si nous considérons le tout satisfaisant. Par la suite, nous pourrons réitérer le même procédé jusqu'à obtenir satisfaction. Nous serons ainsi assurés d'avoir une hiérarchie

adaptée à notre domaine et complète.

## 2.4 Développement de la hiérarchie

Maintenant que nous avons une méthodologie correcte, nous pouvons créer notre hiérarchie. Le fait de s'assurer de sa complétude nous permettra de ne pas rencontrer de problèmes complexes lors de la phase d'implémentation du logiciel.

### 2.4.1 Extraction des différents concepts

Les concepts du domaine (qui se ramènent à des classes) se divisent en plusieurs catégories. Une des principales est le domaine des formes. Une forme peut être un carré, un cercle, etc. Une autre catégorie importante est formée des points, qui peuvent être définis de plusieurs manières différentes. Finalement, une autre section importante est formée des structures de conteneurs, incluant les `Canvas` (pour contenir des objets) et des `Drawing` (pour référencer des objets).

Étant donné le nombre relativement faible de formes distinctes, il ne nous est pas apparu utile de faire plusieurs catégories de formes distinctes. En effet, une séparation entre les courbes fermées et ouvertes peut nous permettre de factoriser un ou deux éléments de code, sans toutefois être significatif ou réellement utile (étant donné qu'il n'existe pas vraiment de forme standard définie par une courbe ouverte).

### 2.4.2 Éléments particuliers

Plusieurs éléments particuliers se sont révélés complexes à modéliser dans notre hiérarchie. Nous abordons ici les principales difficultés rencontrées ainsi que les solutions pour les régler.

### Création et placement d'objets

L'élément le plus important de notre hiérarchie est de pouvoir structurer les objets que nous souhaitons positionner. Nous avons donc créé une classe `Canvas`. Le nom est inspiré du mot canevas en peinture étant donné que nous avons apposé des restrictions similaires à ce domaine. En effet, nous avons décidé que lorsqu'un objet est placé dans un canevas, il ne peut être enlevé ou modifié. Cela nous permettra plus tard de faire des calculs pour les points de référence plus aisément. De plus, un canevas doit pouvoir posséder des sous-canevas. Pour cela, il nous faut une classe générale qui décrit tout élément pouvant être ajouté dans un canevas.

Cela est résolu avec l'introduction d'une classe `Shape`. Celle-ci représente toute forme qui peut être dessinée, qu'elle soit simple ou complexe. Un `Canvas` est une forme, afin de s'assurer de pouvoir positionner des sous-canevas. Une implication particulière de notre choix est qu'une `Shape` ne possède pas de position. Il s'agit d'une forme abstraite qui peut être positionnée à un certain endroit. Cela implique aussi qu'il est possible de prendre une `Shape` et de la placer à plusieurs endroits distincts. Lorsqu'elle est ajoutée dans un `Canvas`, une `Shape` est copiée, ce qui empêche toute modification subséquente de la forme.

### Objets multiples

Lorsqu'il s'agit de créer des diagrammes fréquemment, il est très probable que plusieurs éléments soient similaires. Par exemple, si nous créons un arbre avec des cercles comme sommets, il s'agit toujours, en quelque sorte, du même cercle. De plus, si nous créons un sous-canevas et que nous le positionnons à plusieurs endroits, certaines formes deviennent automatiquement dédoublés. Il devient donc primordial pour nous de pouvoir réutiliser un élément que nous avons déjà défini afin d'éviter la création à répétition d'objets.

Or, nous avons déjà spécifié qu'il était possible de positionner plusieurs fois le même

objet. Cela est vrai, mais il y a une limitation. En effet, ces objets ne peuvent pas être référencés. Si nous souhaitons connaître un objet, celui-ci doit pouvoir être distingué d'un autre. Il nous faut donc une classe qui représente une forme à un endroit précis. Pour cela, nous avons créé la classe `Drawing`. Celle-ci permet de représenter une `Shape` à un endroit bien précis dans un canevas. Cela fait toutefois que les éléments dans un sous-canevas multiple restent identiques. Nous verrons bientôt la solution à ce problème.

### Points de référence

Un des éléments les plus complexes relevait du fait de fournir des points de référence. En effet, faire de simples chemins entre des coordonnées en valeurs absolues n'était pas suffisant, même s'il est possible d'additionner ses points. La solution souhaitée était de permettre de définir des points de référence, ceux-ci pouvant être obtenus à partir de l'objet lui-même. Par exemple, un cercle pourrait retourner son point nord.

Un élément fondamental causait problème : la fonction pour accéder au point doit être définie dans la forme (par exemple, dans la classe `Circle`). Par contre, une forme en tant que telle ne possède pas de position spécifique. Par conséquent, son point n'est pas défini non plus. Pour cela, il faudrait obtenir un `Drawing`. Par contre, un `Drawing` ne peut pas posséder la définition de chaque fonction d'accès à un point pour chaque forme. Cela est contre-intuitif. Il faut donc demander à la forme, lorsqu'on appelle pour obtenir un point, de prendre le `Drawing` supplémentaire en paramètre. Notre intention est, lorsque nous écrirons un langage au-dessus de la hiérarchie, de pouvoir régler ce problème.

### Chemins de référence

Finalement, il nous reste un léger problème. Nous pouvons maintenant faire une référence à un objet, disons $a$, et obtenir un point spécifique de cet objet. Par contre, cela n'est pas suffisant dans un cas bien précis. Disons que $a$ est dans le canevas $b$. Ensuite, $b$ est placé à deux reprises dans le canevas $c$, obtenant deux `Drawing` différents, $c1$ et $c2$.

Nous ne pouvons pas accéder à $a$ car il est représenté seulement en fonction de $b$. Pour régler le problème, il faut faire un chemin de référence (disons `c1.b.a` dans le langage final). Pour cela, nous fournirons la possibilité de partir d'un `Drawing` et d'en obtenir un autre (à condition qu'il s'agisse d'un `Canvas`, naturellement). Cela va mettre à jour la position relative de l'objet par rapport à un niveau de profondeur supplémentaire. En accédant à cette fonction aussi souvent que nécessaire, il est possible de référencer un objet, peu importe son degré de profondeur réel. Cela nous permet donc de faire des liens complexes entre n'importe quel objet (voire même des objets de profondeur différentes).

# CHAPITRE III

# ÉLÉMENTS GRAPHIQUES

Dans ce chapitre, nous définissons les éléments graphiques requis pour faire des dessins informatiques, dont les courbes de Bézier ainsi que les splines cubiques. Les propriétés importantes de ces courbes sont démontrées. De plus, une preuve est fournie de la possibilité d'approximer adéquatement un cercle avec des splines cubiques. Finalement, nous discuterons du fonctionnement des différentes solutions informatiques en terme de structure, telles que PDF et Asymptote.

## 3.1 Courbes et segments en informatique

Naturellement, les éléments les plus importants dans des dessins informatiques sont les segments et les courbes. Pour des besoins de simplicité, nous essaierons de les intégrer dans la même définition, en traitant les segments comme un cas particulier des courbes. Nous donnons à cette courbe générale le nom de *chemin*.

Le premier problème lorsque nous voulons faire des courbes en informatique est la méthode pour les représenter. L'être humain peut tracer une infinité de courbes distinctes, sans avoir besoin de la décrire avec précision. Par contre, lorsqu'on entre dans le domaine de l'informatique, il faut non seulement être capable d'en faire une description, mais celle-ci doit être la plus petite possible.

Une des méthodes de représentation est sous la forme de pixels (Jr. et Kelley, 2007). Cette méthode implique de représenter un plan continu par un ensemble de points dis-

crets appelés *pixels*. Cela permet de représenter plusieurs formes de dessins extrêmement complexes et est utilisé dans certains logiciels d'images.

Une autre méthode est la méthode vectorielle. C'est la plus populaire. Au lieu de représenter directement le dessin tel qu'il apparaît, on fournit une description du dessin à partir de quelques concepts très simples (points, segments et cercles par exemple). Lorsque l'utilisateur souhaite voir le dessin, un logiciel traduit la description en pixels pour l'écran.

La dernière méthode possède plusieurs avantages. La description d'un dessin est souvent beaucoup plus compacte que si on décrit le dessin pixel par pixel. De plus, la description conserve la précision du dessin peu importe la résolution à laquelle il est présenté. Par exemple, si nous prenons un ensemble de pixels et qu'on l'agrandit plusieurs fois, il est très probable que nous ayons perdu beaucoup d'informations. Il nous faut par contre vérifier la puissance réelle d'une telle méthode. Nous devons donc introduire une des structures utilisées pour la description vectorielle.

## 3.2 Courbes de Bézier

Une courbe de Bézier est un concept relativement simple, développé indépendamment par Paul de Casteljau et Pierre Bézier entre 1959 et 1962. Ces deux personnes avaient pour objectif de créer des représentations par ordinateur de voitures pour les compagnies Citroen et Renault respectivement (Jr. et Kelley, 2007).

### 3.2.1 Définition

Soit $P_0$, $P_1$, $P_2$, ..., $P_n \in \mathbb{R}^2$ un ensemble de points dans un plan cartésien typique. Ceux-ci sont appelés des **points de contrôle**. Soit $t \in \mathbb{R}$ et $0 < t \leq 1$. La courbe de Bézier de degré n des points $P_0$ à $P_n$, est définie par la formule suivante :

$$B_{P_0,P_1,\ldots,P_n}(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i P_i \tag{3.1}$$

Le chemin est défini pour les valeurs entre 0 et 1 de $t$. Cette formule peut sembler légèrement arbitraire et complexe pour une forme supposément simple. Une seconde définition, récursive cette fois-ci, simplifie beaucoup la formule.

### 3.2.2 Définition alternative

$$
\begin{aligned}
B_P &= P && (3.2)\\
B_{P_0,P_1,\ldots,P_n}(t) &= (1-t)B_{P_0P_1\ldots P_{n-1}}(t) + tB_{P_1P_2\ldots P_n} && (3.3)
\end{aligned}
$$

Cette définition alternative se ramène à dire qu'une courbe de Bézier est une interpolation linéaire entre deux courbes de Bézier de degré inférieur. La démonstration se ramène à quelques manipulations bien simples et est laissée en exercice au lecteur.

Nous pouvons analyser les formules pour différents degrés. Par exemple, une formule de Bézier de degré 1 n'est qu'une droite :

$$B_{P_0,P_1}(t) = (1-t)P_0 + tP_1 \tag{3.4}$$

Alors qu'une courbe de degré 2 se ramène à une parabole :

$$B_{P_0,P_1,P_2}(t) = (1-t)^2P_0 + 2t(1-t)P_1 + t^2P_2 \tag{3.5}$$

Finalement, une courbe de Bézier cubique (de degré 3) fournit une courbe relativement complexe qui commence à $P_0$ et termine à $P_3$.

$$B_{P_0,P_1,P_2,P_3}(t) = (1-t)^3P_0 + 3t(1-t)^2P_1 + 3t^2(1-t)P_2 + t^3P^3 \tag{3.6}$$

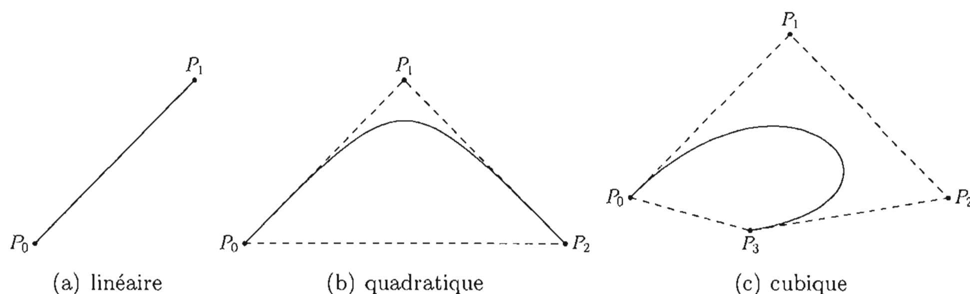



**Figure 3.1** Courbes de Bézier linéaire, quadratique et cubique

Évidemment, il n'y a pas d'avantages significatifs à utiliser une courbe de Bézier linéaire. Les courbes de Bézier quadratique sont, quant à elles, utilisées lors de la génération de police TrueType (Apple Computer, Inc., 1996). Finalement, les courbes de Bézier cubiques sont utilisées dans Postscript (Adobe Systems Incorporated, 2010a) et dans la plupart des produits LaTeX tel que Metafont (Knuth, 1986). Ces logiciels utilisent en fait une combinaison de courbes de Bézier appelées splines que nous verrons dans la section suivante.

### 3.2.3 Propriétés d'une courbe de Bézier

En regardant les différentes courbes, nous pouvons aisément voir quelques propriétés. Entre autres, il y a interpolation des points de départ et de fin. Cela peut se démontrer aisément en calculant $B(0)$ et $B(1)$ qui donne respectivement $P_0$ et $P_n$. De plus, les autres points ne semblent pas être interpolés. Il s'agit ici d'un cas général (les points peuvent être interpolés s'ils sont colinéaires).

Une autre propriété intéressante des courbes de Bézier est son invariance par transformation affine. Cela se ramène au théorème suivant. Dans le cadre de ce théorème, nous devrons transformer les coordonnées des points de contrôles des courbes de Bézier en coordonnées projectives homogènes. Ceci se ramène à rajouter un axe à chacun de nos points, dont la valeur sera toujours 1.

Théorème d'invariance affine

*Soit $B_{P_0,P_1,\ldots,P_n}(t)$ une courbe de Bézier. Soit $M$ une matrice 3 par 3 représentant une transformation affine. Alors, on a que la courbe est invariante par l'application de $M$, c'est-à-dire $MB_{P_0,P_1,\ldots,P_n}(t) = B_{MP_0,MP_1,\ldots,MP_n}(t)$.*

Pour le prouver, il suffit de développer la formule :

$$\begin{aligned} MB_{P_0,P_1,\ldots,P_n}(t) &= M\sum_{i=0}^{n}\binom{n}{i}(1-t)^{n-i}t^iP_i \\ &= \sum_{i=0}^{n}M\binom{n}{i}(1-t)^{n-i}t^iP_i \\ &= \sum_{i=0}^{n}\binom{n}{i}(1-t)^{n-i}t^i(MP_i) \\ &= B_{MP_0,MP_1,\ldots,MP_n}(t) \end{aligned}$$

Ce théorème implique que, pour transformer une courbe de Bézier, il suffit d'appliquer la transformation à chaque point de contrôle et de calculer une nouvelle courbe de Bézier. Donc, une courbe de Bézier est entièrement définie par ses points de contrôle. Cela fait qu'elle est très pratique en informatique car elle ne requiert que peu d'espace pour la définir (i.e. seulement les $n+1$ points de contrôle). De plus, il est très simple de générer une courbe de Bézier car il suffit d'utiliser la formule pour autant de valeurs de $t$ que nous avons besoin de précision dans la courbe. En conservant l'information avec cette méthode, nous conservons une courbe sans aucune perte d'information. Cela signifie que nous pouvons agrandir la résolution de la courbe autant que nous le souhaitons et il s'agira toujours d'une belle ligne courbe. Cette propriété est extrêmement intéressante en informatique.

### 3.2.4 Désavantages des courbes de Bézier

Les courbes de Bézier sont utilisées dans beaucoup de domaines. Par contre, elles possèdent quelques désavantages peu intéressants. Le plus important est probablement l'absence de contrôle local. Cela s'exprime par le fait que dès que l'on souhaite modifier une courbe de Bézier à un endroit particulier, tout le reste de la courbe est modifié au passage. Cela se démontre par le fait que si $0 < t < 1$, on a alors que

$$\forall i, \binom{n}{i}(1-t)^{n-i}t^i P_i \neq 0$$

Cela implique que chaque point de contrôle de la courbe de Bézier a une influence (aussi minime soit-elle) sur l'ensemble de la courbe. Le résultat est que les courbes de Bézier seules sont difficiles à utiliser et longues à évaluer si on utilise trop de points de contrôle.

## 3.3 Splines cubiques

La solution à ce problème est l'utilisation de splines cubiques. L'idée de base d'une spline est que, si une seule courbe cause trop de problèmes, il est probablement mieux d'utiliser plusieurs courbes, mais qui fonctionnent seulement dans un domaine plus petit. Plus formellement, une spline cubique (aussi appelé B-spline cubique) est composé de

- un ensemble de valeurs $t_i$ pour $0 \leq i \leq m-1$ où $m$ représente le nombre de noeuds. On appelle aussi ces valeurs le vecteur de noeud. Les $t$ forment une suite monotone croissante et sont tous des valeurs réelles entre 0 et 1.
- $L+1$ points de contrôle.
- Une valeur entière $n$ qui représente l'ordre de la fonction B-spline. Une spline doit respecter la condition $n \leq m-2$.

On définit une spline par la fonction suivante :

$$S(t) = \sum_{i=0}^{m-n-2} P_i b_{i,n}(t), t \in [0,1]$$

ou, dans le cas spécifique d'une spline de degré 3 :

$$S(t) = \sum_{i=0}^{3} P_i b_{i,3}(t), t \in [0,1]$$

Les $b_{i,n}$ représentent les bases de la B-spline (pour *Basis Spline*). Celles-ci sont définies par la fonction récursive suivante :

$$b_{j,0}(t) = \begin{cases} 1 & \text{si } t_j \leq t < t_{j+1} \\ 0 & \text{sinon} \end{cases}$$

$$b_{j,n}(t) = \frac{t - t_j}{t_{j+n} - t_j} b_{j,n-1}(t) + \frac{t_{j+n+1} - t}{t_{j+n-1} - t_{j+1}} b_{j+1,n-1}(t)$$

Il s'agit d'une équation complexe et nous ne nous y attarderons pas trop. Quelques détails sont intéressants à noter. Entre autre, il est intéressant de constater que, si l'on utilise $n+1$ points de contrôle et que les noeuds sont $n$ noeuds valant zéro et $n$ noeuds valant 1, le B-spline dégénère en une courbe de Bézier. Il s'agit donc d'une extension des capacités des courbes de Bézier.

Le problème principal évoqué par les courbes de Bézier est que l'utilisateur, lors de la création d'une courbe, manquait de contrôle local. Il serait donc intéressant que les B-splines nous permettent de régler ce problème. Lorsqu'on regarde la définition de la base $b_i$, on remarque que celle-ci se ramène toujours au cas de base qui peut être 0 ou 1 selon que nous sommes dans un domaine restreint ou non. Comme il s'agit du seul facteur de $P_i$, il est naturel d'affirmer que le contrôle est plus localisé qu'avant. D'autres propriétés intéressantes sont :

- Pour tout $t$, la somme de tout les $b_i(t)$ est 1.

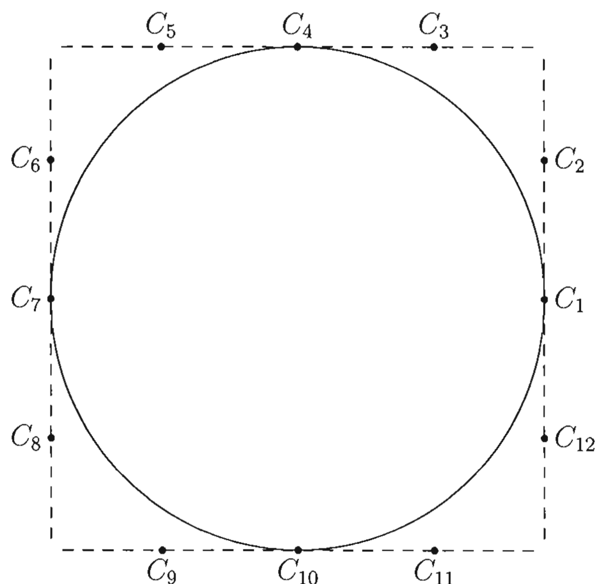



**Figure 3.2** Cercle complet approximé par une b-spline composé de quatre courbes de Bézier

- Un B-spline d'ordre $m$ sera nécessairement $C^{m-2}$, c'est-à-dire qu'elle possède 2 ordres de dérivées continues pour son domaine de définition.
- Tout comme les courbes de Bézier, les B-splines sont invariants par transformation affine. On peut donc appliquer une courbe en appliquant tout simplement la transformation sur les points de contrôle et en calculant le nouveau B-spline.

## 3.4 Le cas d'un cercle

Avec les outils que nous possédons, nous ne sommes pas en capacité de dessiner un cercle exact. Cela n'est pas grave étant donné que nous pouvons tout de même tenter de l'approximer. Pour ce faire, nous utiliserons une spline cubique possédant quatre points (est, nord, ouest et sud respectivement). Chaque courbe de Bézier sera donc identique aux autres à une rotation près. Il reste donc à déterminer les points de contrôle à utiliser.

Pour cela, prenons le premier quadrant du cercle. Nous passons du point $C_1(1,0)$ à $C_4(0,1)$. Il nous faut déterminer deux points de contrôle, $C_2$ et $C_3$. Pour commencer,

nous pouvons supposer, de manière tout à fait arbitraire, que $C_2$ se situe sur la tangente de $C_1$, et $C_3$ sur la tangente de $C_4$. Cela nous donne donc le schéma suivant :

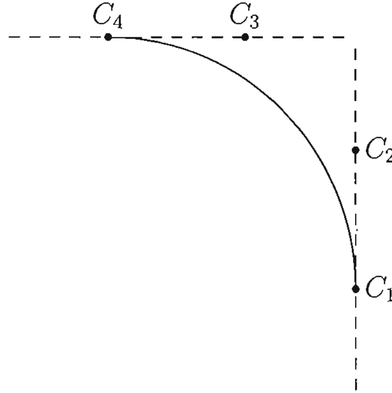



**Figure 3.3** Quart de cercle approximé par une courbe de Bézier

On a donc $C_2(1, k)$ et $C_3(k, 1)$ où $k$ est une variable à déterminer. Afin de la déterminer, nous devons donner une information supplémentaire. Par exemple, nous pouvons présupposer que le point à 45 degrés $C(0.5)$ doit être évalué exactement par la courbe de Bézier, c'est-à-dire $C(0.5) = (\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2})$. Nous obtenons donc la restriction

$$\begin{aligned} C(0.5) &= (1-t)^3 C_1 + 3(1-t)^2 t C_2 + 3(1-t)t^2 C_3 + t^3 C_4 \\ &= \frac{1}{8}C_1 + \frac{3}{8}C_2 + \frac{3}{8}C_3 + \frac{1}{8}C_4 \end{aligned}$$

ce qui nous permet d'obtenir l'égalité

$$(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}) = \frac{1}{8}C_1 + \frac{3}{8}C_2 + \frac{3}{8}C_3 + \frac{1}{8}C_4 \tag{3.7}$$

ce qui nous fournit, après substitution des valeurs des points, deux fois l'équation

$$\frac{\sqrt{2}}{2} = \frac{1}{2} + \frac{3k}{8} \tag{3.8}$$

ce qui nous fournit, après résolution

$$k = \frac{4}{3}(\sqrt{2} - 1) \approx 0.552285$$

Cela nous fournit donc une courbe de Bézier qui approxime notre cercle. Il reste à voir la précision de cette approximation.

Pour ce faire, nous devrons comparer l'arc de cercle que nous venons de créer à un cercle réel. Afin d'y arriver, nous considérerons chaque valeur de notre arc comme étant un vecteur dont nous devrons calculer la longueur. Plus cette longueur est proche de 1, plus nous avons une estimation réaliste. Nous calculons donc

$$f(t) = 1 - \|(1-t)^3 C_1 + 3(1-t)^2 t C_2 + 3(1-t) t^2 C_3 + t^3 C_4\|$$

Par la suite, une analyse des dérivées nous permet de conclure que cette fonction, dans l'intervalle $t \in [0,1]$ possède des maximums à $t = 0$ à trois reprises (au début et à la fin, étant donné la propriété naturelle des courbes de Bézier ; celle au milieu est due à notre contrainte). Finalement, elle possède deux (2) minimums à $t = 0,211$ et $t = 0,789$ et l'erreur avec un cercle standard est de 0.000273. Une telle variation, sur un cercle de rayon 1, est minime. Il s'agit d'une variation maximale inférieure à 0.03%. Cela signifie donc, pour un cercle possédant un rayon de 4000 pixels, il y aura une variation maximale de 1.2 pixels, ce qui est indétectable pour la taille du cercle. Cela en fait une bonne estimation pour cette section du cercle. Nous pouvons naturellement suivre le même raisonnement pour les sections restantes.

Il est aussi à noter que notre estimation n'est pas la meilleure. Notre choix de demander l'égalité à 45 degrés est arbitraire, et toute l'erreur se retrouve dans le côté négatif. De meilleures estimations ont été effectuées (Riskus, 2006), avec des résultats amenant une précision jusqu'à 25% plus grande (Lancaster, 2005).

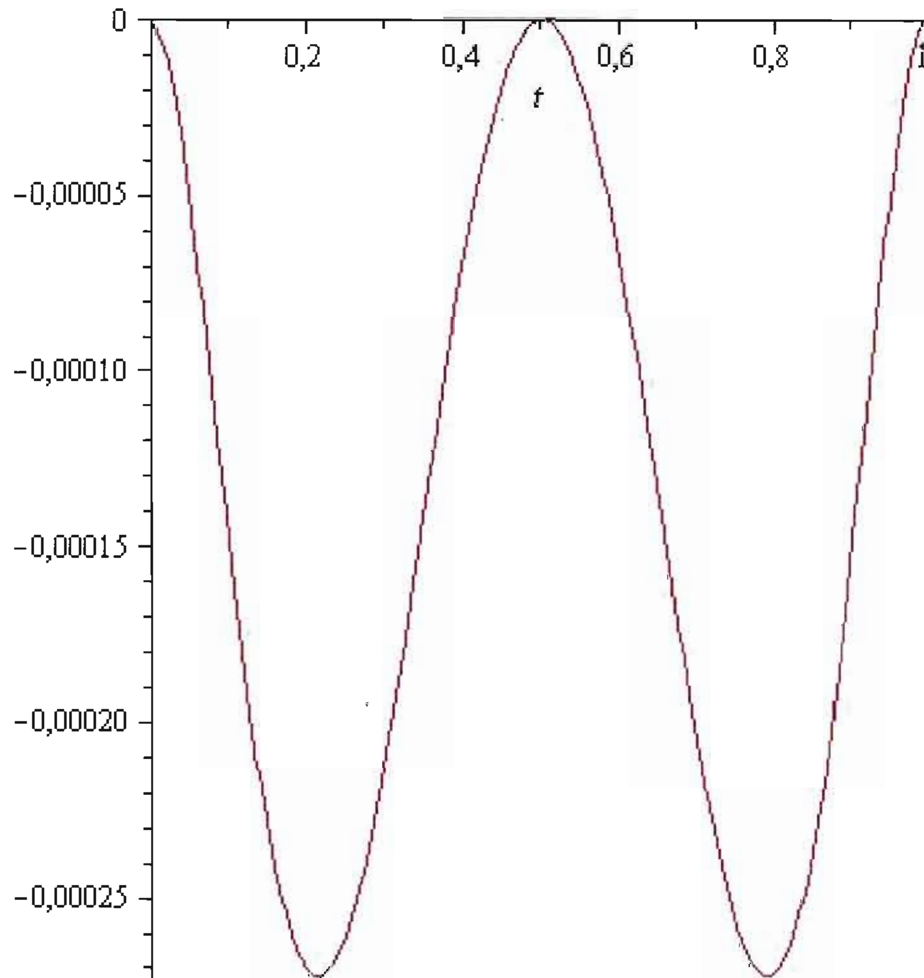



**Figure 3.4** Variation de la différence entre un arc de cercle standard et la courbe de Bézier dans son domaine de référence

## 3.5 Structures informatiques dans les solutions déjà existantes

Maintenant que nous savons comment fonctionne les différents modes de représentation des dessins en informatique, il faut analyser plus en détail comment fonctionne les logiciels dont nous allons nous servir. En effet, il nous faudra adapter nos structures de bas niveau à celles disponibles dans les langages dont nous nous servirons. Pour cela, nous allons analyser deux parties distinctes : un format de fichier (PDF) ainsi qu'un

langage intermédiaire (Asymptote).

### 3.5.1 Postscript et PDF

En premier lieu, il convient de comprendre comment fonctionne le format de fichier final que nous utiliserons. LaTeX permet de générer plusieurs formats de fichier différents. Pendant longtemps, le plus populaire aura été Postscript, créé par Adobe Systems. Cela était principalement lié aux imprimantes qui fonctionnaient directement avec ce format. De nos jours, le format PDF est beaucoup plus populaire, bien qu'ayant une base très proche de celle de Postscript.

Postscript a été inventé par John Warnock et Charles Geschke en 1982 (Adobe Systems Incorporated, 1994). Lorsqu'il a été créé, il servait à faire la description de pages pour impression. Il se distingue comme étant un langage de programmation complet, mais rarement utilisé directement. En effet, Postscript est difficile à utiliser car il utilise la notation polonaise inverse. Par exemple, on exprime l'appel de fonction $f(a, b, c)$ de la façon suivante.

```
a b c f
```

Cette méthode rend l'analyse d'un programme extrêmement rapide et convivial pour le logiciel qui l'interprète, car celui-ci est directement lié à la structure d'une pile d'exécution. Par contre, l'analyse d'un tel programme par un humain est complexe. La plupart du temps, Postscript était donc utilisé comme un langage de description final pour un fichier.

Il existait deux types de structures utilisés en Postscript pour mettre des dessins sur une page. Le premier était l'ajout de texte. Cela est intrinsèquement lié à la description des polices, qui dépasse le cadre de ce mémoire. Par contre, pour faire un dessin quelconque, le format utilisait les chemins, qui sont décrits soit par des segments, soit par des courbes de Bézier cubiques. On note donc le désavantage de l'utilisation d'une courbe de Bézier, soit l'absence de contrôle local.

Quant à lui, PDF a été créé en 1993 (Adobe Systems Incorporated, 2010b). À travers les gains de popularité, le format PDF est devenue en 2008 un standard ISO (International Organization for Standardization, 2008). Cela en fait donc un format de document standard extrêmement intéressant. Alors que Postscript a été créé pour des documents qui devaient être imprimés, PDF a été conçu pour la description de documents plus dynamiques, c'est-à-dire la possibilité d'ajouter des éléments multimédias au document. Aussi, il est possible de décrire chaque page indépendamment des suivantes. En effet, comme Postscript décrivait des documents d'impression, il est normal de connaître la page de départ avant d'imprimer la suivante. En PDF, chaque page est totalement indépendante du reste du document.

Pour la description d'éléments graphiques, les structures de base sont essentiellement les mêmes que Postscript. Quelques éléments comme la transparence, c'est-à-dire qu'un élément caché par un autre élément contribue au résultat final, ont été ajoutés, mais les éléments graphiques de bas niveau sont identiques.

### 3.5.2 Asymptote

Comme nous l'avons dit précédemment, tout ramener à des courbes de Bézier crée un problème de contrôle local des courbes. Il faut donc utiliser une méthode pour donner du contrôle à l'utilisateur. Pour cela, nous utiliserons Asymptote. Les structures de bas niveau de Asymptote sont identiques à celles de PDF, étant donné qu'il convertit directement les chemins Asymptote en chemins PDF. Par contre, Asymptote implémente un algorithme (Bowman, 2007), originellement développé par Donald Knuth qui permet de décrire un chemin en fonction des points par lesquels il passe. Asymptote utilise donc des splines cubiques (qui ne sont pas nécessairement des B-splines, toutefois).

Cet algorithme est complexe et en dehors du cadre de ce mémoire, mais peut être résumé comme étant un algorithme de choix des points de contrôle. En effet, une courbe de Bézier cubique est définie par 4 points. Le premier et le dernier sont interpolés de manière exacte. Ce que l'algorithme fait est qu'il prend un ensemble de points qui

seront tous interpolés, disons $P_1, P_2, P_3, P_4$. Ensuite, pour chaque paire consécutive de points (donc $(P_1, P_2), (P_2, P_3)$ et $(P_3, P_4)$), il générera une courbe de Bézier cubique. La partie la plus importante de l'algorithme consiste à prendre toutes les informations fournies par l'utilisateur (les points de contrôle ainsi que des options qui définissent la courbe comme les dérivées à certains endroits ou des informations comme la tension qui laisse l'algorithme calculer lui-même les dérivées à partir d'un indice réel simple) afin de calculer, pour chaque courbe de Bézier cubique, les deux points de contrôle manquants. En laissant à l'utilisateur le pouvoir de décider seulement les points où la courbe passe, le travail est simplifié, mais permet tout de même de terminer avec un ensemble de courbes de Bézier.

# CHAPITRE IV

# THÉORIE DES LANGAGES INFORMATIQUES

La création d'un langage de programmation n'est pas une mince tâche. Il faut donc s'assurer de le construire sur des bases théoriques solides. Nous présenterons ici les éléments théoriques importants de la théorie des langages ainsi que leurs applications dans le domaine de la compilation. Nous présenterons aussi l'outil `SableCC`, que nous avons utilisé dans le cadre de notre langage ainsi qu'un bref exposé sur les langages réflexifs.

La théorie des langages est un sujet vaste et extrêmement complexe. Un mémoire complet pourrait être rédigé seulement sur les aspects théoriques des automates réguliers. Afin de compresser tout ce contenu dans un seul chapitre, les éléments seront mentionnés sans preuve. Nous recommandons toutefois au lecteur curieux et assidu de consulter les références de ce chapitre pour des preuves et une couverture plus complète du sujet.

## 4.1 Théorie des langages et automates

Dans un premier temps, nous devrons déterminer ce qu'est, formellement, un langage. Par la suite, nous analyserons une structure de données qui est intimement liée à ce sujet, les automates.

### 4.1.1 Les éléments de base d'un langage (Aho, Sethi et Ullman, 1986)

#### Alphabet et lettre

Un alphabet est un ensemble (au sens mathématique du mot) de symboles appelés lettres. Un alphabet est fini et non-vide. Par exemple, l'alphabet latin standard ainsi que l'ensemble $\{0, 1\}$ constituent des alphabets.

#### Mot

Un mot d'un alphabet $A$ peut être un des trois cas suivants :

- le mot vide, $\epsilon$, formé d'aucune lettre,
- une lettre de l'alphabet $A$,
- Toute suite finie de lettres $a_0a_1 \ldots a_n$ où chaque $a_i$ fait partie de l'alphabet $A$.

Il est à noter que les deux premiers sont en fait des cas spécifiques du dernier, bien qu'ils sont souvent séparés pour raison de clarté. Dans le dernier cas, on dit que les caractères sont liés par l'opération de concaténation.

#### Langage

Un langage, au sens le plus général possible, est un ensemble de mots. Celui-ci peut être fini ou infini. La plupart des exemples intéressants (un langage de programmation par exemple) est un langage infini. En effet, il existe une infinité de programmes différents, tout comme il existe une infinité de nombres binaires.

### 4.1.2 Expressions régulières

Les expressions régulières forment un outil de travail extrêmement important en informatique. De nombreux outils ainsi que certains langages (grep, perl, ...) leurs sont consacrés.

On peut résumer les expressions régulières comme étant une méthode afin de décrire un langage infini par une expression fini. On utilisera dans la construction certaines opérations qui nous donneront des propriétés intéressantes pour le langage final. Tout mot qui peut être construit à partir de ces règles fait automatiquement partie du langage.

Constantes

L'élément de base d'une expression régulière est une constante. Celle-ci est définie à partir d'une lettre de l'alphabet et définit un seul mot : la constante elle-même. Par conséquent, l'expression régulière $a$ de l'alphabet $\{a, b\}$ représente le langage $\{a\}$. Il est à noter que nous pouvons aussi utiliser le mot vide comme une constante représentant l'ensemble vide.

Concaténation

L'opération de concaténation est une opération sur les mots qui permet de créer des expressions régulières plus longues à partir de deux expressions régulières valides. Le résultat de l'expression régulière $A$ concaténée à $B$ est donc n'importe quel mot de $A$ concaténé à n'importe quel mot de $B$. Par exemple, $ab$ est une expression régulière qui représente un langage à un mot : $\{ab\}$.

Union

L'union des expressions régulières est une opération qui correspond à l'union ensembliste. Lors de l'union de $A$ et de $B$ (défini comme $(A|B)$). nous pouvons choisir n'importe quel élément de $A$ ou de $B$ comme un mot valide. Par exemple, $(a|b)b$ représente le langage contenant deux mots : $\{ab, bb\}$.

Fermeture de Kleene

La fermeture de Kleene est une opération beaucoup plus intéressante. Celle-ci utilise une expression régulière $A$. Le résultat de la fermeture, noté $A^*$, est la concaténation

d'un nombre fini d'éléments de $A$, avec possibilité de répétitions. Par exemples, pour $a^*b$, toute chaîne de $a$ finie, possiblement de longueur zéro, suivie d'un $b$ fait partie du langage. L'expression régulière $(a|b)^*$ représente n'importe quelle chaîne finie de $a$ ou de $b$.

Les langages que nous pouvons générer avec des expressions régulières sont assez riches, bien qu'insuffisants pour générer un langage de programmation (Aho, Sethi et Ullman, 1986).

### 4.1.3 Automates

Avant de pouvoir expliquer à quoi servent les expressions régulières, nous devons discuter des automates, une structure d'une importance cruciale en informatique.

#### Automate fini

Un automate fini (Ullman, Hopcroft et Motwani, 1979) est un quintuplet composé de cinq éléments :

- Un ensemble fini d'états $E$
- Un alphabet fini $A$
- Un ou plusieurs états de départ $d$ (selon que l'automate est déterministe ou non)
- Un ensemble d'états dit finaux $F$.
- Une relation de transition $T \subseteq E \times A \times E$. Cela définit une manière de passer d'un état à un autre lors de l'apparition d'un symbole spécifique.

On représente en général un automate par un graphe avec des flèches nommées. Les flèches représentent la relation de transition alors que les sommets représentent les états dans $E$. On représente un état initial par une flèche qui ne part pas d'un état. Finalement, on représente un état final par un cercle double. Par exemple, l'automate suivant accepte l'ensemble des mots qui sont composés d'une série de a (possiblement nulle), suivi d'un b. Donc, les mots b, ab et aab serait des mots valides par rapport à

cet automate.

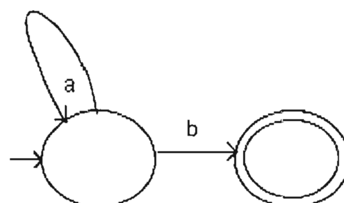



**Figure 4.1** Automate acceptant les mots formés d'une série de a suivi d'un b

L'ensemble $T$ peut prendre plusieurs formes en fonction du déterminisme de l'automate, comme nous allons maintenant le voir.

Automate non-déterministe

Un automate non-déterministe peut posséder plusieurs états de départ. De plus, son ensemble de transitions $T$ est quelconque. Il peut donc exister pour un état $e$ et un symbole $a$ plusieurs états $e'$ tels que $(e, a, e') \in T$.

Un automate non-déterministe peut être vu comme une forme de graphe dans lequel on peut se promener en utilisant certaines transitions. La suite des symboles des transitions que nous utilisons forment un mot. L'ensemble des mots obtenus par une suite de transitions allant d'un état initial à un état final constitue le langage de l'automate. Dans un automate non-déterministe, il peut néanmoins exister, pour le même mot, plusieurs chemins différents débutant dans un état initial. Il est important de noter qu'un mot est accepté dès qu'un de ces chemins se termine dans un état final.

Automate déterministe

Un automate non-déterministe nous permet de créer des langages, mais le fait d'avoir plus d'un choix pour le même symbole peut sembler inquiétant, surtout lorsqu'il s'agit d'informatique. Nous pouvons donc définir un automate déterministe plus simple. Celui-

ci possède un seul état de départ et son ensemble de transitions $T$ assure que pour chaque symbole, il n'y a qu'un seul état suivant possible. Il n'y a donc pour chaque mot qu'un seul chemin possible à partir de l'état initial. Plus formellement, l'automate sera déterministe si pour tout $e \in E$ et $a \in A$ il y a un seul $e' \in E$ tel que $(e, a, e') \in T$.

Le réflexe naturel serait de croire qu'en enlevant le non-déterminisme d'un automate, celui-ci devient beaucoup moins puissant, permettant de représenter beaucoup moins de langages. Or, il n'en est rien. Il a été prouvé (malheureusement, en trop de pages pour ce mémoire) que n'importe quel automate non-déterministe peut être ramené à un automate déterministe, quitte à voir le nombre d'états de l'automate augmenter exponentiellement. Nous pouvons donc toujours créer un automate non-déterministe et laisser la tâche à un ordinateur de créer un automate déterministe à partir de celui-ci (Ullman, Hopcroft et Motwani, 1979).

## Exécution d'un automate et contraintes temporelles

La méthode d'exécution d'un automate déterministe est assez simple. Nous lui fournissons un mot d'une certaine longueur, disons $n$. Nous démarrons de l'état initial et chaque lettre représente un choix à faire dans l'automate. Comme l'automate est déterministe, il n'y a qu'une possibilité à explorer. Si nous ne pouvons plus avancer avant la fin du mot, celui-ci ne fait pas partie du langage de l'automate. Si nous parvenons à avancer jusqu'au bout et que nous ne sommes pas dans un état final, il n'en fait pas partie non plus. Par contre, si l'état est final, nous avons trouvé un mot du langage. Comme le mot possède $n$ lettres, nous aurons au maximum $n$ choix à faire, rendant l'algorithme d'appartenance à un temps d'exécution linéaire. Un algorithme linéaire est naturellement extrêmement intéressant en informatique car nous voulons un programme qui s'exécute dans un temps raisonnable.

### 4.1.4 Lien entre langage régulier et automate fini

Nous connaissons donc maintenant deux méthodes pour créer un langage. Les expressions régulières ainsi que les automates. Un automate est difficile à concevoir pour un être humain, mais extrêmement rapide d'utilisation par un ordinateur. Une expression régulière est complexe pour un ordinateur, mais aisée à concevoir par un être humain. Il nous faut donc voir si nous pouvons obtenir des propriétés communes à ces deux outils.

#### Le lien

Il existe en effet un lien qui unit ces deux structures. Celui-ci est extrêmement fort : l'ensemble des langages qu'il est possible de générer avec des expressions régulières est exactement le même ensemble que ceux qui peuvent être générés par un automate fini déterministe. Cela implique que, pour chaque expression régulière, nous pouvons trouver un automate qui représente le même langage. Cela est vrai aussi en sens inverse. Par contre, il ne s'agit pas d'une bijection, car plusieurs expressions régulières peuvent correspondre au même automate (par exemple $a^*$ et $aa^*$). Il existe un algorithme permettant de transformer une expression régulière en automate, mais celui-ci est complexe et dépasse le cadre de ce mémoire. Nous pouvons toutefois mentionner que cet algorithme est relativement lent (s'exécute en $O(2^m)$ ou $m$ est la longueur de l'expression régulière), mais permet par la suite de trouver si un mot fait partie d'un langage en temps linéaire (McNaughton et Yamada, 1960).

#### Importance d'un tel lien

Ce lien est un élément crucial dans la capacité d'un ordinateur à travailler avec des langages de programmation. En effet, cela permet aux créateurs de langages de décrire des mots à partir d'expressions régulières (qu'ils comprennent) et de laisser un programme créer des automates qui peuvent reconnaître ces symboles (un `if` ou un `for`, mais aussi des variables) dans des programmes. Cela permet donc de créer une partie importante d'un langage, qu'on appelle l'analyse lexicale. Cela permet, à partir de l'utilisation de

l'automate créé par des expressions régulières, de transformer un texte en une suite de symboles plus spécifiques. Par exemple, le texte

```
if a == 1 then true else false;
```

pourrait être traduit par la suite de symboles suivants

```
IF VAR EQ NUM THEN BOOL ELSE BOOL FININSTR
```

Cette étape, l'analyse lexicale, est cruciale à la suite de l'analyse d'un programme. Les automates nous sont toutefois utiles, non seulement pour les mots clés, mais aussi pour des éléments plus complexes, comme des variables. Une variable est souvent identifiée comme étant une lettre suivie d'une suite de caractères quelconques (sauf exception, tel qu'un espace).

Malgré tout, il existe certains éléments qu'un automate ne peut reconnaître. Un exemple classique est le parenthésage équilibré (une suite de parenthèses telle que chaque parenthèse ouvrante est associé à une parenthèse fermante et, à n'importe quel moment, il n'existe pas plus de fermantes que d'ouvrantes). Il est en effet impossible aux automates de faire le lien entre les deux types de parenthèses (Ullman, Hopcroft et Motwani, 1979). Pour cela, il nous faudra utiliser des structures plus complexes.

## 4.2 Grammaires

Ces structures sont les grammaires. Il est possible de les voir comme étant une forme beaucoup plus générale des expressions régulières. Avec cette structure, nous pourrons nous assurer qu'un programme est bien formé, c'est-à-dire créer un arbre représentant la structure d'un programme, un peu comme la structure d'une phrase.

### 4.2.1 Définition d'une grammaire formelle

Une grammaire est constituée de quatre éléments :

- Un ensemble fini de terminaux. Ceux-ci sont des symboles représentant les lettres d'un langage. Par conséquent, on peut aussi appeler les terminaux l'alphabet d'une grammaire.
- Un ensemble fini de non-terminaux. Habituellement représenté par des majuscules (par des symboles différents des terminaux) pour des raisons de clarté, ceux-ci représentent des états intermédiaires de la grammaire.
- Un non-terminal de départ parmi l'ensemble décrit plus haut.
- Un ensemble de règles de production. Celles-ci vont varier selon la catégorie de langage choisie, mais, de manière générale, une règle de production est une paire ordonnée dont chaque élément est composé d'une suite de terminaux et de non-terminaux (par exemple $aB \rightarrow aAa$).

Le fait de représenter une règle de production avec une flèche permet d'en expliciter le rôle : transformer une suite de caractères en une autre. L'utilisation d'une grammaire consiste donc à démarrer avec le non-terminal de départ, puis d'appliquer une suite de règles de production pour modifier le non-terminal. Lorsqu'on obtient un mot formé uniquement de terminaux, il s'agit d'un mot du langage représenté par la grammaire. Par exemple,

```
S -> aA
A -> b
```

représente un langage contenant le seul mot *ab*. On applique doEn effet, unc un procédé de dérivation (en appliquant chaque règle). Par exemple, une dérivation possible du mot *ab* dans ce contexte (la seule d'ailleurs) est la suite suivante :

```
S (départ)
S -> aA (application de S -> aA)
aA -> ab (application de A -> b)
```

### 4.2.2 Hiérarchie de Chomsky

La définition que nous avons donnée précédemment d'un langage est beaucoup trop générale pour nous fournir des propriétés intéressantes. Afin de dégager des propriétés intéressantes, nous définissons plusieurs catégories. Plusieurs noms sont utilisés pour chaque catégorie, mais nous nous limiterons à utiliser ceux définis par le linguiste Noam Chomsky (Chomsky, 1956). Chacune de ces catégories sera décrite par les propriétés de sa grammaire. Il est à noter que nous avons négligé certaines catégories de langages qui ne sont pas liées au domaine.

#### Langage régulier

Un langage régulier est équivalent aux expressions régulières. Les règles de production de la grammaire doivent être de la forme

```
A -> Ba
A -> a
```

ou

```
A -> aB
A -> a
```

avec $A$ et $B$ des non-terminaux et $a$ un terminal. Il est à noter que toutes les règles d'une grammaire doivent être soit du premier cas, soit du deuxième cas. On les appelle récursives à gauche ou à droite respectivement.

#### Langage non-contextuel

Un langage non-contextuel est un peu plus complexe qu'un langage régulier. Les règles de production de la grammaire doivent être de la forme

```
A -> B
```

avec $A$ un non-terminal et $B$ une suite de terminaux et de non-terminaux. Le détail le plus important est de noter que les langages non-contextuels permettent de créer un système de parenthèses équilibrées. En effet, voici une grammaire qui accepte seulement un parenthésage équilibré :

```
P -> PP
P -> (P)
P ->
```

Il est évident qu'une grammaire régulière ne peut générer ce genre de langage. Par contre, un parenthésage équilibré est extrêmement important dans le domaine de la compilation afin d'exprimer clairement les idées d'un programmeur et, plus spécifiquement, permettre d'écrire des formules mathématiques complexes compréhensibles par l'ordinateur.

Langage contextuel

Les langages contextuels ne sont pas utilisés en général en compilation car ils sont beaucoup plus lents. Les règles de production de la grammaire doivent être de la forme

```
ABC -> ADC
```

où $B$ est un non-terminal, $A$ et $C$ sont des non-terminaux ou le mot vide et $D$ est un non-terminal (mais pas le mot vide). On voit clairement ici d'où vient le nom de ce type de langage : $B$ est transformé en fonction de ce qui est autour de lui (son contexte).

### 4.2.3 Utilité dans les langages de programmation

La prochaine tâche dans la création d'un langage est d'analyser la structure de notre programme. Pour effectuer cette tâche, nous pouvons utiliser une grammaire non-con-

textuelle. À cette fin, nous pouvons nous assurer que les équations sont correctement formées, que les instructions sont valides, etc.

Dans un autre ordre d'idée, la hiérarchie de Chomsky possède aussi des liens avec les machines de Turing, qui sont elles-mêmes intimement liées à la théorie de la complexité (Davis, Sigal et Weyuker, 1994). Ce domaine est extrêmement vaste et le lecteur pourra trouver dans les références plusieurs excellents ouvrages pour s'initier à ce domaine (Dhu et Ko, 2000).

## 4.3 SableCC

Bien qu'il est très intéressant d'étudier les algorithmes de compilation, l'implémentation de chacun de ces algorithmes dépasse largement le cadre de ce mémoire. Il nous faut donc trouver une méthode afin de créer notre langage de programmation. Plusieurs outils s'offrent à nous, mais notre choix s'arrêtera sur SableCC.

### 4.3.1 Qu'est-ce que SableCC

SableCC est ce qu'on appelle un compilateur de compilateurs (Gagnon, 1998). Un compilateur de compilateurs est un programme qui permet, à partir d'une définition abstraite d'un langage de programmation, d'obtenir un compilateur pour ce langage. Plus spécifiquement, le programme résultant nous permettra d'obtenir l'arbre syntaxique d'un programme, ce qui nous permettra de l'analyser récursivement et de l'exécuter (ou bien de le transformer dans un langage différent, selon le cas). Les formes abstraites que requiert un compilateur de compilateurs sont justement les expressions régulières (pour découvrir les symboles) et les grammaires non-contextuelles (pour l'analyse syntaxique des symboles obtenus précédemment).

### 4.3.2 Avantages de SableCC

Bien que SableCC n'est pas le compilateur de compilateurs le plus connu, il possède plusieurs aspects intéressants. Premièrement, comparativement à ses compétiteurs plus

âgés, SableCC est écrit dans un paradigme orienté-objet. Cela rend le compilateur beaucoup plus simple à utiliser. De plus, le logiciel génère automatiquement un ensemble de classes (dans un choix de plusieurs langages) qui permet de traiter aisément et clairement, avec des concepts modernes, les arbres syntaxiques résultant de l'analyse du programme.

## 4.4 Langage réflexif

Finalement, une fois que nous avons obtenu un arbre syntaxique et que nous savons que notre langage est bien formé, il nous reste la réflexivité à prendre en compte. En effet, nous ne voulons pas réécrire inutilement dans notre langage chaque fonction qui existe déjà en Java. Nous pourrons les utiliser avec un langage réflexif. Une autre possibilité d'un langage réflexif est de déléguer à `SDDL` la création de classes. Cela permet à l'utilisateur de référencer n'importe quelle classe `Java`, sans avoir à changer le code de `SDDL` lui-même. Cela serait impossible sans un langage réflexif.

### 4.4.1 Définition d'un langage réflexif

Un langage réflexif (Sobel et Friedman, 1996) est un langage qui est capable soit de

- Examiner son état, cette propriété est appelée introspection
- Modifier son état, cette propriété est appelée intercession

L'introspection permet à un langage de connaître, entre autres, les classes qu'il utilise, ainsi que les méthodes et les paramètres. L'intercession, beaucoup plus puissante, permettrait par exemple de rajouter, modifier ou supprimer des classes et/ou méthodes. Pour nos besoins, nous sommes naturellement intéressés par l'intercession.

### 4.4.2 Avantages

L'avantage le plus clair de l'utilisation de la réflexivité est la réutilisation du code. En effet, toutes les fonctionnalités du langage inférieur (Java, dans ce cas-ci) peuvent être

automatiquement passées au langage supérieur. Cela évite d'avoir à réécrire du code pour chaque nouvel élément à ajouter.

Un autre avantage assez explicite est de permettre à des utilisateurs un peu plus aguerri de lier du code des deux niveaux de langage ensemble. En effet, si un utilisateur crée une structure de données en Java, il peut toujours se servir exactement de la même classe dans le langage supérieur, sans avoir à faire la moindre modification au langage supérieur lui-même. Il est donc clair qu'un langage qui utilise la réflexivité peut devenir très puissant.

### 4.4.3 Désavantages

Certains désavantages accompagnent aussi la réflexivité. Un des plus importants est la contrainte de sécurité. En effet, si un utilisateur peut appeler n'importe quelle classe Java à partir du langage supérieur, il peut très aisément appeler des fonctions qui cassent le programme si un contrôle suffisamment important n'est pas effectué par le logiciel. Il peut aussi, sans causer des problèmes de sécurité, confondre un usager en provoquant des effets de bord sur l'état du logiciel, c'est-à-dire modifier des propriétés du reste du programme sans nécessairement que cela paraisse.

Un autre désavantage évident est qu'un utilisateur qui souhaite utiliser les fonctions du langage inférieur doit, en quelque sorte, connaître celui-ci. Cela implique donc l'apprentissage d'un deuxième langage, ce qui en soit peut paraître trop pour des utilisateurs plus traditionnels.

Lorsqu'on analyse froidement les avantages et les inconvénients, on voit que les avantages sont beaucoup plus importants. En effet, le projet étant limité dans le temps et dans le personnel, il est en quelque sorte impossible de fournir des librairies de fonctions suffisamment complètes pour les utilisateurs. Se brancher sur les librairies Java fournit automatiquement tout ce dont nous avons besoin et donne accès à une quantité innombrable de projets déjà créés dans ce langage.

# CHAPITRE V

# MISE EN OEUVRE DE LA SOLUTION

Maintenant que nous avons des bases théoriques solides ainsi qu'une hiérarchie bien adaptée à nos besoins, nous pouvons commencer à implémenter notre solution. Cela implique premièrement la création de notre hiérarchie objet en Java. Deuxièmement, nous devons implémenter une grammaire afin de décrire SDDL. Finalement, il faut ajouter du code afin d'implémenter certaines propriétés voulues pour notre langage. Il est à noter que nous avons inclus en annexe un diagramme de notre hiérarchie ainsi que la grammaire complète de SDDL.

## 5.1 Éléments de langage en SDDL

### 5.1.1 Implémentation de la hiérarchie

L'implémentation de la hiérarchie étant séparée du reste du langage, il est normal de commencer avec celle-ci étant donné qu'elle représente le module le plus important de notre projet. Cela nous permet de le tester au fur et à mesure avant de commencer à créer un langage de plus haut niveau qui l'utilise.

Un des problèmes les plus importants du développement de la hiérarchie fut de développer correctement la définition et l'accès à des points spécifiques de certains objets. Comme l'idée était de permettre d'obtenir une quantité essentiellement illimitée de points différents, il fallait une méthode robuste et efficace.

Notre méthode implique de fournir des informations par rapport au dessin englobant à chaque sous-dessin. Cela est requis car un objet présent à plusieurs endroits différents auquel on demande un point spécifique se bute à une ambiguïté. Même en utilisant seulement un `Drawing`, qui est techniquement une instance unique d'une forme, il faut faire attention. En effet, si on a un `Drawing` d'un cercle qui est à l'intérieur d'une ellipse, par exemple, nous pourrions décider de placer cette ellipse à plusieurs endroits. Nous aurions donc le même problème d'ambiguïté du positionnement des points. Afin de remédier à ce problème, nous avons fourni au `Drawing` sa position à l'intérieur du diagramme. Lorsqu'il est copié, nous n'avons qu'à changer l'information pour que le positionnement soit correct.

### 5.1.2 Grammaire

Le développement de la grammaire fut naturellement une section cruciale du développement de SDDL. Comme nous avions déjà développé des objectifs spécifiques pour la forme du langage, la création fut relativement directe. Quelques éléments importants ont toutefois dû être ajoutés en cours de route. En particulier, des éléments séparateurs ont été d'une importance cruciale. Il aurait donc pu être souhaitable de diminuer la verbosité en enlevant des parenthèses ou des virgules, mais cela cause (dans la plupart des cas) des problèmes au niveau de l'ambiguïté de la grammaire (tel que nous l'avons vu précédemment).

Nous avons aussi décidé de décréter qu'une expression générale est une instruction. Cela peut sembler contre-intuitif, car une expression (comme `2 + 2`) ne semble pas être une instruction. La raison derrière ce changement est le dessin de chemin. En effet, nous pouvons déterminer un chemin (`Path`) comme n'importe quel autre objet. Par contre, cela implique de l'ajouter par la suite séparément à un canevas. Cela semble contre-intuitif : nous voyons souvent les liens entre les objets comme étant des liens de niveau supérieur (bref, faits dans le canevas principal). Pour lier `a` et `b`, il semble plus logique d'écrire

```
a--b;
```

que

```
lien = a--b;
put lien in main;
```

Afin de régler ce problème, nous avons décidé de laisser les expressions être des instructions. Cela fait que définir une ligne (comme a--b) devient automatiquement une instruction valide. Celle-ci sera traitée comme un cas spécial afin d'être placée directement au centre du canevas principal. Cela implique aussi, comme nous le verrons plus tard, de rajouter du code afin de s'assurer que les expressions qui ne sont pas des instructions soient refusées par le programme.

```
stm =
  {exp} exp sc |
  {assign} id assign exp sc |
  {while} while_kwd exp do_kwd [stms]:stm* end_kwd sc? |
  {if} if_kwd exp then_kwd [stms]:stm* else_part? end_kwd sc? |
  {for} for_kwd id from_kwd [low_bound]:exp to_kwd [up_bound]:exp
                               do_kwd [stms]:stm* end_kwd sc? |
  {return} return_kwd exp sc;
```

**Figure 5.1** Extrait de la grammaire de SDDL pour une instruction

### 5.1.3 Implémentation des différentes propriétés du langage

Plusieurs propriétés de SDDL ne venaient pas directement avec la grammaire. Celles-ci ont nécessité un traitement bien particulier, en fonction de la propriété recherchée.

#### Typage

La grammaire de SDDL indique clairement que le langage est à typage dynamique étant donné que les créations de variables ne permettent pas de donner un type. Le même

phénomène est présent dans la définition des fonctions. Par contre, le fait de souhaiter un typage dynamique implique l'ajout d'une large section de code dédiée à la vérification de type. Chaque opération ne peut être appliquée qu'à certains types d'éléments et donc, avant d'effectuer l'opération, il faut vérifier si le type est valide.

Éléments objets

Implémenter tous les principes d'un langage orienté-objet est un travail de longue haleine. Par contre, étant donné que notre objectif était de simplifier le langage de description final, nous avons décidé de ne pas permettre la création de nouvelles classes à l'intérieur de SDDL. Cela implique que nous pouvons reléguer la création de classes uniquement au langage Java. Cela a donc grandement facilité la tâche car nous n'avions pas besoin d'implémenter la très grande majorité des concepts objets que nous « empruntons » à Java.

Élimination de la verbosité

Une bonne partie de l'élimination de la verbosité vient de la conception même du langage. En effet, le fait de reléguer la définition des classes à Java ainsi que de créer le typage dynamique ont, purement par choix, éliminé une bonne partie de la verbosité.

Quelques éléments supplémentaires ont permis d'éliminer un excès de verbosité. On note entre autres le fait de définir la description du diagramme sans le spécifier dans le cadre d'une fonction. Un autre phénomène important est d'avoir décidé de promouvoir les points et certaines fonctions reliées à l'état de primitive naturelle du langage. En effet, une addition entre un point `a` et `b` s'effectue

```
c = a + b
```

au lieu de, en Java,

```
AbstractPoint c = a.add(b)
```

Cela peut sembler comme un maigre gain, mais lorsqu'appliqué à un calcul complexe impliquant des points (par exemple, le calcul d'une combinaison linéaire), les gains combinés rendent le code extrêmement clair.

```
AbstractPoint c = a.multiply(1.0/3.0).add(b.multiply(2.0/3.0));

c = (a / 3.0 + 2.0 * b / 3.0);
```

**Figure 5.2** Code Java et SDDL pour calculer une combinaison linéaire

## 5.2 Réalisation en Java

Afin de réaliser SDDL, il a bien sûr fallu écrire une grammaire, mais beaucoup de code autour sert à intégrer la grammaire au restant des fonctionnalités de notre hiérarchie. Ce code fut complètement écrit en Java et remplit plusieurs fonctions.

### 5.2.1 Traversée de l'arbre grammatical et exécution

La grande majorité du code Java supplémentaire écrit a servi à permettre la traversée de l'arbre grammatical créé par notre grammaire. Cette traversée permet en quelque sorte l'exécution du code SDDL et est donc cruciale. Pour la plupart, il s'agit d'une traversée en profondeur relativement classique. Quelques petits détails retiennent toutefois l'attention.

- Certains morceaux de code ne doivent pas être exécutés et ont donc un comportement spécial. C'est le cas des définitions de fonctions, qui ne sont exécutées qu'à l'appel. On mémorise donc en quelque sorte une partie de l'arbre grammatical et, lors d'un appel, on transfère le contrôle d'exécution à cette partie. Évidemment, lors de la lecture de la définition de la fonction, il faut stocker quelques informations sur nos fonctions dans une structure définie à cet effet (un `MethodInfo`).
- Certaines opérations ont plusieurs sens distincts qui ne peuvent être spécifiés dans la grammaire. La sémantique d'une telle opération n'apparaît clairement qu'au moment de la lecture de l'opération (dans notre cas, à l'exécution). Un exemple serait le

symbole d'addition. Bien que sa grammaire soit très claire, il peut s'agir d'une addition de nombres, mais aussi d'une addition de points ou d'une concaténation de chaînes de caractères. Il est donc nécessaire à l'exécution de vérifier la classe de chaque objet requis par l'opération afin de trouver la bonne sémantique pour l'opération, si une telle sémantique existe. Si aucune n'est trouvée, on peut alors soulever une erreur selon le mécanisme standard (par exemple, additionner un point et une chaîne de caractères).

- Certaines expressions grammaticalement correctes ne sont valides que sous certaines conditions. Par exemple, n'importe quelle expression (`exp`) est acceptée comme étant une instruction (`stm`) grammaticalement valide. Par exemple, l'instruction `2 + 2;` va être acceptée par la grammaire. Par contre, elle n'a pas de sens car elle n'accomplit rien du tout. Il faut donc s'assurer de vérifier dans la traversée que l'exécution est légale. Dans ce cas-ci, il faut s'assurer que l'expression de niveau supérieur (une addition) est valide pour être une instruction. Dans un tel cas, une erreur serait soulevée encore selon le mécanisme standard.

### 5.2.2 Définition d'objets quelconques

Une partie moins grande du code est consacrée à l'intégration de la hiérarchie de SDDL dans la définition du langage. Comme nous souhaitions utiliser des classes quelconques, nous avons utilisé la réflexivité Java telle que nous le décrivions plus tôt. Cela implique de récupérer les noms des fonctions et des classes lors de leurs utilisations. Toutes les vérifications s'effectuent à l'exécution. Nous savons donc seulement si une classe A existe bel et bien lorsque nous essayons de créer un objet à partir de celle-ci. De plus, chaque fonction appelée sur un objet de classe A n'est vérifiée que lors de son utilisation. Habituellement, la solution aurait été de vérifier avant l'exécution l'existence de chaque classe ainsi que les fonctions de chaque classe pour s'assurer que l'exécution soit correcte. Dans notre cas, la vérification à l'exécution est suffisante car les programmes sont très courts.

De plus, quelques contraintes ont dû être ajoutées afin de s'assurer que seules les parties

```
@Override
public void caseADrawingTerm(ADrawingTerm node){

  String className = node.getClassName().getText();

  try{

    this.currentExpressionClass =
         Class.forName("sddl.hierarchy." + className);
    this.currentExpressionValue =
         this.currentExpressionClass.getConstructor().newInstance();

  } catch(NoSuchMethodException e){

    throw new InterpreterException("class " + className + " is
          ill-defined (no base constructor)", node.getClassName());

  } catch(Exception e){

    throw new InterpreterException("unable to load " + className +
          " class (maybe it does not exist?)", node.getClassName());

  }

}
```

**Figure 5.3** Code réflexif pour créer une forme quelconque

requises de Java sont exposées à l'utilisateur final. La restriction principale est de ne créer que des objets qui font partie du paquet `sddl.hierarchy.*`. Il est toutefois à noter que cela n'isole pas totalement le reste de Java car il est tout à fait possible de créer une classe qui permet d'accéder à Java complètement.

### 5.2.3 Gestion des erreurs

La gestion des erreurs s'effectue au travers des mécanismes déjà mis en place par Java, c'est-à-dire les exceptions. Une classe spécifique aux exceptions de l'interpréteur a été créée, qui permet de spécifier non seulement un message, mais aussi un noeud bien spécifique dans l'arbre grammatical afin d'identifier l'endroit où l'erreur a été découverte.

Cela est extrêmement utile afin de permettre à l'utilisateur de corriger la majorité de ses erreurs.

De plus, une partie du code de gestion d'erreurs sert à représenter la pile d'exécution. En effet, afin d'identifier la source d'un problème, il est important de pouvoir savoir quelles fonctions sont impliquées. Nous pouvons donc identifier la ligne de chaque appel au moment où l'erreur s'est produite et ainsi identifier des problèmes, entre autres dans le cas des fonctions récursives.

### 5.2.4 Structures de support

D'importances mineures, la plupart de ces structures servent à gérer l'exécution des méthodes.

- La classe `Frame` représente le cadre d'exécution d'une fonction. Cette classe permet d'obtenir des fonctions récursives car on peut imbriquer un `Frame` dans un autre.
- La classe `MethodInfo` représente une fonction définie à l'intérieur de SDDL. Elle permet de transférer l'exécution à la définition fournie par l'utilisateur et regroupe les informations généralement utiles à la vérification sémantique (nom, paramètres, etc.)
- La classe `PrimitiveMethodInfo` représente une fonction de Java accessible par la réflexivité. Elle fournit aussi quelques informations pour la vérification sémantique.
- La classe `MethodTable` fournit une table pour gérer l'ensemble des fonctions, afin d'accéder directement à une fonction à partir de son nom.

## 5.3 Accès à Postscript, PDF et LaTeX

### 5.3.1 Postscript

Afin de réaliser correctement notre langage, nous avions besoin d'accéder à différentes technologies. Étant donné que nous voulons terminer avec des dessins en format Postscript, il devient important de pouvoir fournir aisément du code dans ce format. Il

s'agit toutefois d'un format relativement complexe, et il devient donc long et complexe de fournir une traduction valide pour chacun de nos différents concepts.

Le problème principal de Postscript est qu'il possède un ensemble de primitives de très bas niveau. En effet, afin de dessiner une courbe, la seule méthode possible est l'utilisation de courbes de Bézier avec des points de contrôle qui doivent être fournis à l'avance. Comme nous l'avons vu précédemment, les courbes de Bézier sont très pratiques pour l'ordinateur, mais peu pratique pour l'utilisateur car la spécification de points de contrôles qui ne sont pas interpolés est peu intuitive. De plus, afin de faire des courbes quelconques, il faut les séparer en plusieurs courbes de Bézier de tailles inférieures.

Il existe des algorithmes qui permettent de prendre des courbes et de les séparer en sections adéquates. Un de ces algorithmes est implémenté en Asymptote (Bowman, 2007). Il est donc naturel, afin d'éviter d'avoir à réimplémenter un algorithme aussi complexe, d'utiliser Asymptote pour obtenir nos courbes de façon systématique.

### 5.3.2 PDF

Le format PDF possède l'avantage d'intégrer plusieurs fonctions multimédias, telles que les liens, les vidéos, les formulaires (Adobe Systems Incorporated, 2006), etc. Malgré tout, PDF, outre les quelques ajouts nommés ci-haut, est essentiellement similaire à Postscript. Évidemment, les nombreuses possibilités fournies par le format PDF sont désirables (intégrer des vidéos à des notes de cours, par exemple). Il existe toutefois de nombreux convertisseurs qui peuvent prendre un fichier Postscript et le mettre au format PDF. Par contre, cette conversion ne permet pas d'ajouter les fonctionnalités PDF qui nous intéressent. Il devient donc crucial d'obtenir une méthode pour fournir des images directement au format PDF. Encore là, le fait de passer par Asymptote nous donne une solution rapide et élégante car le fait de fournir du PDF est inclus dans le logiciel. En générant directement du Asymptote, nous pourrions donc obtenir automatiquement le format recherché.

### 5.3.3 LaTeX

L'élément le plus crucial du langage que nous recherchions au début était de pouvoir utiliser du texte tel que mis en forme par LaTeX dans nos diagrammes. Un tel besoin implique plusieurs problèmes.

Premièrement, un texte LaTeX est conservé sous la forme d'une chaîne de caractères standard. Par contre, après avoir été traité, le résultat obtenu est une boîte contenant le texte mis en forme. La taille de cette boîte est déterminée lors de la mise en page et ne peut donc pas être déterminée directement par notre programme. Or, si nous ne connaissons pas la taille exacte de notre boîte de texte, il est impossible de positionner un texte par une position autre que son coin supérieur gauche. Cela est un comportement peu intéressant étant donné que dans de nombreux graphes, on place le texte au centre d'un sommet (donc, par exemple, le centre de la boîte sera placé au centre d'une ellipse).

Afin de pallier à ce problème, il faudrait demander à LaTeX de créer les boîtes une première fois et de nous fournir les tailles des boîtes. Par la suite, il faudrait générer une page LaTeX de la bonne taille avec nos dessins, puis dire à LaTeX de générer les boîtes de textes aux endroits que nous aurons calculé entre temps. Bien que viable, cette méthode implique beaucoup de code et de cas différents (l'accès à LaTeX par un programme externe va varier d'un système d'exploitation à l'autre, entre autres). Étant donné qu'Asymptote fournit déjà un accès à LaTeX à l'intérieur du logiciel, le travail requis pour l'utilisation de LaTeX est de beaucoup diminué.

### 5.3.4 Génération du code Asymptote

Afin d'avoir une solution valide, il ne reste donc plus qu'à connecter notre solution à Asymptote. Pour le faire, nous devons premièrement générer un fichier de code Asymptote valide pour, par la suite, le faire valider en appelant le programme de manière externe. Étant donné que le projet a été écrit spécifiquement pour fournir du code Asymptote, une solution simple a été utilisée afin d'écrire le code : le canevas principal

```
public void draw(AbstractPoint location, PrintWriter out)
{

  AbstractPoint p1 = location.add(new Point(radius, 0.0));
  AbstractPoint p2 = location.add(new Point(0.0, radius));
  AbstractPoint p3 = location.add(new Point(-radius, 0.0));
  AbstractPoint p4 = location.add(new Point(0.0, -radius));

  out.println("draw(" + p1.solve() + ".." + p2.solve() +
             ".." + p3.solve() + ".." + p4.solve() +
             "..cycle);");

}
```

**Figure 5.4** Code pour générer du code Asymptote pour un cercle

sera appelé à être dessiné, qui appellera récursivement chacun de ses sous-canevas.

La génération du code Asymptote a été réservée dans une fonction `draw` spécifique à chaque forme (`Shape`) qui écrit dans un fichier ouvert fourni en paramètre. Un exemple d'une telle méthode pour la classe `Circle` est illustré à la figure 5.4.

Une telle méthode est simple, mais suffisante pour nos besoins. Bien sûr, à long terme, une solution plus robuste devrait s'imposer. En effet, comme l'écriture du fichier est faite directement lors de l'exécution, les objets doivent être appelés exactement dans l'ordre dont ils sont dépendants. Par exemple, un segment dessiné entre deux cercles doit être dessiné après les cercles car sinon les références seront absentes. Dans ce cas-ci, le segment va toujours être définie par la suite. Par contre, si nous voulions rajouter la possibilité d'utiliser des extensions d'Asymptote, cela impliquerait de les écrire au début du fichier. Il est impossible de le faire avec cette solution.

Une solution plus robuste serait d'utiliser un outil lié à SableCC qui est ObjectMacro (Gagnon, 2009). Cette extension nous permet de simplifier la génération de code et d'écrire à des endroits arbitraires dans un fichier à l'aide de macros et de programmation objet. Cela nous permettrait aussi d'étendre la génération de code à plusieurs formats différents, par exemple générer directement du code PDF au lieu de générer du code

Asymptote.

# CHAPITRE VI

# RÉALISATION DES DIAGRAMMES EN SDDL

Maintenant que nous avons notre programme complet et fonctionnel, il faut vérifier son efficacité. Nous procédons en deux temps. Premièrement, il faut revenir à nos dessins de base et les coder avec notre logiciel. Par la suite, il faut aussi les analyser en fonction des différents facteurs cruciaux que nous avons énumérés dans les chapitres précédents. Nous pourrons donc, à partir de cela, conclure si le logiciel s'avère efficace pour nos besoins ou si certains problèmes l'empêchent de fonctionner adéquatement.

Naturellement, nous avons pris le temps d'appliquer cette analyse à chacun des dessins dans notre base de diagrammes du domaine. Par manque d'espace, nous n'analyserons dans ce chapitre que trois dessins. Une attention particulière a été portée au choix de ces dessins afin de représenter un éventail assez large des capacités du logiciel et des différents éléments cruciaux à obtenir dans la description d'un diagramme. Tous les diagrammes sont disponibles en annexe, avec code, sans analyse détaillée toutefois.

## 6.1 Analyse du code des diagrammes existants

Pour faire notre analyse, nous devons premièrement obtenir chaque dessin. Dans ce cas, nous utiliserons un diagramme tiré du domaine de la compilation. Cela nous fournit un graphe relativement standard avec un niveau de complexité intéressant, sans toutefois être trop complexe. Par la suite, nous analyserons un diagramme représentant une structure de tableau. Cela montre la capacité du logiciel de travailler avec des structures de

données générales. Finalement, nous effectuerons l'analyse d'un diagramme de Venn qui n'est, a priori, pas supporté par le logiciel, afin de voir la complexité d'un diagramme totalement général et qui sort des conventions pré-établies par le logiciel.

### 6.1.1 Rappel des objectifs cruciaux à observer

Naturellement, afin de faire une analyse rationnelle de notre solution, il faut se remémorer les différents objectifs que nous nous étions fixés. Les plus importants étaient les suivants :

- Intégration avec LaTeX. Étant donné le design de notre solution, ce point est naturellement réalisé dans tous les diagrammes.
- Description de structures. Il nous faut pouvoir décrire une structure de manière aisée. Le fait d'être aisé est naturellement subjectif, mais nous pouvons ramener le tout au fait qu'il soit plus simple de décrire dans notre logiciel une structure que d'utiliser les primitives de base du langage. Par exemple, il est plus aisé de décrire une structure en parlant de boîtes et de liens qu'en parlant de lignes et de courbes.
- Hiérarchie solide et extensible. Nous ne testerons pas l'extensibilité de la solution car elle a déjà été démontrée. Toutefois, les différents codes nous permettront d'analyser à quel point nous utilisons la hiérarchie que nous avons construite et à quel point elle est réellement pratique.

Nous avions aussi d'autres objectifs de moindre importance :

- Gestion des erreurs. Il n'est pas possible avec du code complété de tester la gestion des erreurs. L'auteur peut toutefois garantir que, sans la gestion des erreurs, une grande quantité de temps aurait été perdu dans la rédaction du code des diagrammes et ce, malgré la connaissance profonde du langage sous-jacent.
- Langage fluide et clair. Cela peut être testé par le lecteur en regardant le code. Si une analyse rapide lui permet de comprendre le sens du code, alors celui-ci est clair. Naturellement, l'auteur ne peut effectuer cette analyse par lui-même. Bien que nous ne puissions en faire une analyse scientifique, en inspectant le code résultant, nous

sommes confiants que la syntaxe que nous avons choisie est claire et permet d'exprimer clairement les concepts sous-jacents au langage.

- Qualité de la documentation et courbe d'apprentissage. Il s'agit ici d'un autre élément que nous pouvons difficilement aborder par nous-mêmes. Cela ne sera pas pris en compte dans notre analyse.

Selon ces différents facteurs, nous voyons que nous devrons surtout nous concentrer sur la structure générale du code ainsi que sur sa complexité. Le code doit être facile à lire, tout en permettant de compacter un maximum d'information dans un minimum d'espace.

### 6.1.2 Diagramme de compilation

Notre premier dessin est tiré d'un livre de compilation. Il s'agit d'un automate assez standard et représente donc un bon diagramme "moyen" afin de voir les capacités de notre logiciel.

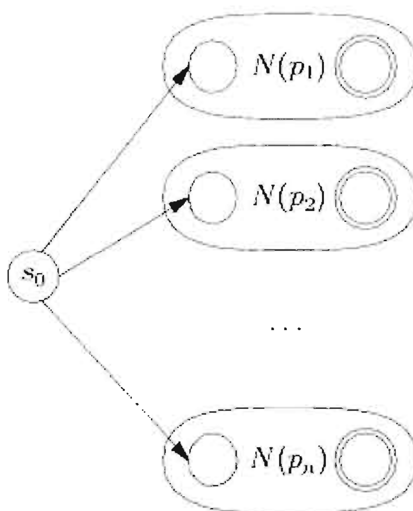



**Figure 6.1** Diagramme d'automate de compilation

Quelques points importants sont à noter dans ce dessin. Premièrement, on note la création d'une structure sous-jacente (composée d'une ellipse et deux cercles). Par la

```
circle1 = Circle with [radius = 10.0];
circle2 = Circle with [radius = 12.0];
ellipse = Ellipse with [xradius = 50.0, yradius = 20.0];
circle = put circle1 at (-30.0, 0.0) in form;
put circle1 at (30.0, 0.0) in form;
put circle2 at (30.0, 0.0) in form;
put ellipse in form;

d1 = put form at (100.0, 100.0) in main;
d2 = put form at (100.0, 50.0) in main;
d3 = put form at (100.0, -50.0) in main;
put Text with [text = "$N(p_1)$"] at (100.0, 100.0) in main;
put Text with [text = "$N(p_2)$"] at (100.0, 50.0) in main;
put Text with [text = "\dots"] at (100.0,0.0) in main;
put Text with [text = "$N(p_n)$"] at (100.0, -50.0) in main;

put Text with [text = "$s_0$"] at (0.0, 20.0) in main;
d4 = put circle1 at (0.0, 20.0) in main;

(coord of anglePoint(70.0) in main.d4)->(coord of
    anglePoint(180.0) in main.d1.circle);
(coord of anglePoint(0.0) in main.d4)->(coord of
    anglePoint(180.0) in main.d2.circle);
(coord of anglePoint(-70.0) in main.d4)->(coord of
    anglePoint(180.0) in main.d3.circle);
```

**Figure 6.2** Code pour le diagramme d'automate

suite, cette forme est appliquée à plusieurs reprises dans le dessin. Cela démontre que, dans ce cas-ci, il est possible de créer et d'utiliser des structures d'un certain degré de complexité.

On note aussi l'utilisation des variables pour récupérer les dessins afin de pouvoir les référencer plus tard. Cette étape est aussi simple que l'assignation à une variable. Par la suite, on peut noter les flèches qui sont tracées à partir de points définis à partir de certains cercles. La structure même est extrêmement explicite dans ce cas et rend la compréhension du dessin très simple. On récupère aussi finalement les points formés à un certain angle d'un cercle.

La complexité du diagramme est beaucoup plus simple qu'elle ne le serait avec des primitives de base. En effet, les seuls dessins explicites que nous devons effectuer par points sont les flèches. De plus, ces flèches sont relativement simples puisque nous n'avons rien à spécifier pour le bout de flèche, seulement les extrémités. Pour le reste, les formes sont soient définies à partir de notre hiérarchie (comme les cercles ou les ellipses) ou bien par un collage de formes plus simples (comme pour la forme `form`). Les points sont obtenus par des accès structurés et fournissent donc des valeurs exactes, rendant ces opérations extrêmement rapides à écrire pour l'usager.

On note toutefois qu'il existe certains éléments dans le code qui auraient pu être simplifiés afin de rendre le code encore plus compact. En effet, nous remarquons la présence de beaucoup de valeurs explicites alors qu'il aurait été possible de les remplacer par des variables afin de rendre le tout plus clair.

Un élément qui aurait pu amener une touche de simplification aurait été d'utiliser des listes ou des boucles afin d'insérer plus aisément les objets. Par exemple, la structure complexe aurait pu être ajoutée plusieurs fois par une boucle `for`, avec des positions calculées en fonction de l'index. Nous sommes toutefois d'avis que, malgré l'amélioration possible que cela fournit au code, il est peu probable que les utilisateurs finaux prennent le temps de faire ce genre de modification et utiliseront fort probablement plus des positions relatives avec une série d'appels, sauf si le nombre d'objets à positionner devenait grand (par exemple, une dizaine d'objets).

En général, nous pouvons toutefois considérer que ce diagramme est simplifié de beaucoup par l'utilisation de notre solution. La grande part du code est explicite et la structure est très simple à observer en regardant uniquement le code. Même si tout n'est pas parfait, il y a clairement avantage à utiliser le logiciel pour coder cet automate.

### 6.1.3 Diagramme de tableau

Notre second dessin est tiré d'un livre sur les structures de données. Il représente l'évolution d'une structure de donnée simple (un tableau) à travers certaines modifi-

cations.

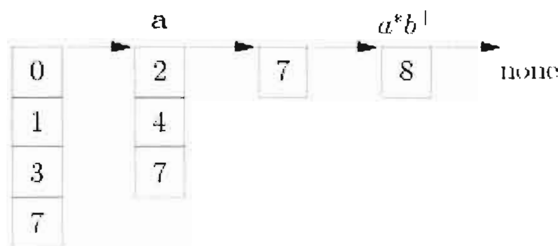



**Figure 6.3** Diagramme de tableau

Un des points les plus intéressants de ce dessin est l'utilisation d'une fonction. Elle place une forme dans un canevas à plusieurs endroits spécifiques en fonction d'une position de départ et d'un vecteur de translation. Cela nous permet de placer une forme identique (un carré) plusieurs fois de suite, ce qui nous donne l'impression qu'il s'agit en réalité d'un tableau.

Un autre point qui est très important dans ce dessin est que les nombreux carrés sont en fait tous des instances du même, défini au début. Cela permet de modifier l'ensemble des carrés aisément en ne modifiant que les propriétés de celui de base. Pour le reste du dessin, il s'agit surtout d'un ensemble de flèches et de textes positionnés à des endroits spécifiques.

En analysant le code, on voit une amélioration claire provoquée par le fait d'utiliser une fonction pour placer les carrés. Cela nous permet de simplifier de beaucoup les appels et les répétitions. Nous pouvons toutefois noter que quelques améliorations supplémentaires pourraient être apportées. En effet, une classe aurait pu être écrite en Java pour représenter un tableau à l'horizontale ou à la verticale. Cela aurait pu simplifier significativement le code du dessin en combinant dans la structure le texte et les carrés utilisés pour représenter le tableau lui-même. De plus, les flèches auraient pu être déterminées à partir de points obtenus du tableau lui-même (par exemple le coin supérieur droit ou supérieur gauche), auxquels nous aurions ajouté un vecentierteur de déplacement.

```
fun dropMany(f, n, s, v, c){
  for i from 1 to n do
    put f at (s + (i - 1) * v) in c;
  end
}

square = Square with [length = 20.0];
changeVec = (0.0, -20.0);
dropMany(square, 4, (0.0, 0.0), changeVec, main);
dropMany(square, 3, (50.0, 0.0), changeVec, main);
dropMany(square, 1, (100.0, 0.0), changeVec, main);
dropMany(square, 1, (150.0, 0.0), changeVec, main);
put Text with [text="0"] at (0.0, 0.0) in main;
put Text with [text="1"] at (0.0, -20.0) in main;
put Text with [text="3"] at (0.0, -40.0) in main;
put Text with [text="7"] at (0.0, -60.0) in main;
put Text with [text="\bf a"] at (50.0, 20.0) in main;
put Text with [text="2"] at (50.0, 0.0) in main;
put Text with [text="4"] at (50.0, -20.0) in main;
put Text with [text="7"] at (50.0, -40.0) in main;
put Text with [text="7"] at (100.0, 0.0) in main;
put Text with [text="\bf $a^*b^+$"] at (150.0, 20.0) in main;
put Text with [text="8"] at (150.0, 0.0) in main;
put Text with [text="none"] at (200.0, 0.0) in main;

(12.0, 10.0)->(38.0, 10.0);
(62.0, 10.0)->(88.0, 10.0);
(112.0, 10.0)->(138.0, 10.0);
(162.0, 10.0)->(188.0, 10.0);
```

**Figure 6.4** Code pour le diagramme de tableau

Avec ces changements, le code aurait pu être beaucoup plus compact. Bien que cette option soit réalisable, nous ne l'avons pas implémentée car il était important de limiter ce qu'un usager peut créer sans l'utilisation de Java. Cela est dû au fait que notre objectif est que seul les utilisateurs experts se servent de Java afin d'améliorer le logiciel. Le code est tout de même significativement meilleur que de placer tout à la main, ce qui nous permet de conclure à l'efficacité de notre logiciel dans le cas d'une structure générale.

### 6.1.4 Diagramme de Venn

Notre dernier dessin est une structure qui ressemble beaucoup plus à un dessin général, c'est-à-dire un diagramme de Venn. La forme d'un tel diagramme fait qu'il est extrêmement complexe d'écrire une classe qui permet de dessiner automatiquement ce type de structure et qu'il vaut mieux créer les formes une par une, comme on le ferait à la main. Cela nous permettra d'analyser si les choses simples à dessiner à la main demeurent facile à réaliser avec notre logiciel.

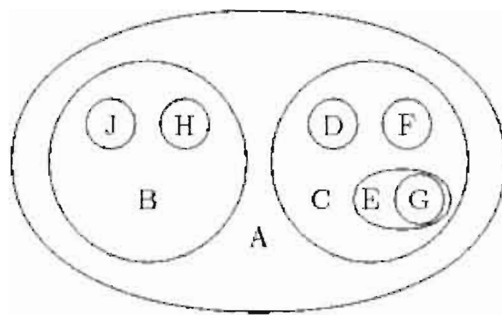



**Figure 6.5** Diagramme de Venn

Ce dessin ne possède pas beaucoup de grands points d'intérêt. En effet, la structure ne peut pas vraiment être générée automatiquement par un algorithme, alors elle est entrée manuellement. On remarque tout de même l'utilisation de plusieurs sous-canevas afin de simplifier le placement des objets qui sont à l'intérieur d'autres.

Naturellement, étant donné que la structure doit être décrite manuellement, le code n'est pas plus court que l'utilisation de primitives de base. Par contre, cela est compensé par un niveau de clarté supérieur étant donné l'utilisation des canevas afin de regrouper les éléments dans des ensembles respectifs. Cela permet de positionner tous les éléments en valeurs relatives au lieu d'absolues, ce qui simplifie de beaucoup le travail de positionnement.

De plus, le fait de pouvoir stocker les éléments dans des variables nous permet de les identifier par différents noms significatifs, ce que des primitives de base comme celles

```
circle = Circle with [radius = 10.0];
bigcircle = Circle with [radius = 40.0];
ellipse = Ellipse with [xradius = 20.0, yradius = 12.0];
bigellipse = Ellipse with [xradius = 100.0, yradius = 60.0];

put bigcircle in canvas1;
put circle at (15.0, 15.0) in canvas1;
put Text with [text = "H"] at (15.0, 15.0) in canvas1;
put circle at (-15.0, 15.0) in canvas1;
put Text with [text = "J"] at (-15.0, 15.0) in canvas1;
put Text with [text = "B"] at (0.0, -15.0) in canvas1;

put bigcircle in canvas2;
put circle at (-15.0, 15.0) in canvas2;
put Text with [text = "D"] at (-15.0, 15.0) in canvas2;
put circle at (15.0, 15.0) in canvas2;
put Text with [text = "F"] at (15.0, 15.0) in canvas2;
put Text with [text = "C"] at (-20.0, -15.0) in canvas2;
put ellipse at (13.0, -15.0) in canvas2;
put Text with [text = "E"] at (0.0, -15.0) in canvas2;
put circle at (20.0, -15.0) in canvas2;
put Text with [text = "G"] at (20.0, -15.0) in canvas2;


put canvas1 at (-45.0, 0.0) in main;
put canvas2 at (45.0, 0.0) in main;
put Text with [text = "A"] at (0.0, -30.0) in main;
put bigellipse in main;
```

**Figure 6.6** Code du diagramme de Venn

de PDF ne nous permet pas de faire. Nous pouvons donc en conclure que, même pour les structures difficiles à représenter, notre logiciel fournit tout de même un nombre significatif d'avantages.

## 6.2 Présentation du produit final

Une fois que nous avons pu confirmer que notre logiciel était efficace, il ne restait plus qu'à le faire connaître. Nous présentons donc ici un résumé des efforts qui ont été effectués afin d'amener le logiciel à la connaissance de la communauté LaTeX.

### 6.2.1 Statistiques et documentation

En premier lieu, avant de pouvoir présenter notre solution, il nous fallait la documenter correctement. En effet, l'effort de développement nous aura permis de développer un produit de très grande taille. La taille totale du travail se limite à environ 10 megs, mais cela implique plus de 6300 lignes de code développées spécifiquement pour le projet (cela incluant les commentaires, mais excluant tout le code généré par `SableCC`, qui est très exhaustif). Avec une taille aussi significative, il est tout simplement impensable de s'attendre à ce qu'un utilisateur analyse le code afin de trouver ce qui va répondre à ses besoins.

Pour cela, nous avons pris le temps de rédiger une documentation interne. Elle consiste à documenter chaque classe de notre hiérarchie, ainsi que chaque fonction disponibles. Cela fournit des renseignements complets pour quelqu'un qui cherche un élément spécifique du langage.

### 6.2.2 Présentation

Par la suite, nous avons rédigé une présentation pour la conférence TUG 2010, que nous avons présentée le 28 juin 2010 à San Francisco, devant des membres très influents de la communauté LaTeX, dont Donald Knuth, le créateur original de TeX. La conférence nous a permis de nous mettre au pouls des besoins de la communauté et de l'intérêt qu'ils portent à un tel projet. La réponse se voulait enthousiaste, particulièrement de la part du présentateur d'Asymptote, John Bowman. La conférence est disponible sur le site de River Valley (Bourgeois, 2010) (malheureusement sans vidéo étant donné un problème d'opérateur).

Bien qu'il s'agisse d'un élément très important dans la présentation d'un projet, une conférence ne rejoint pas beaucoup de monde. Pour combler ce besoin, nous avons aussi rédigé un article pour la revue TUGBoat (Bourgeois et Villemaire, 2010a), qui est la revue principale de la communauté LaTeX. Nous avons inclus l'article en annexe. Cela

nous a permis de rejoindre un public beaucoup plus grand qu'une simple présentation devant public.

### 6.2.3 Site web

Finalement, afin de faciliter la distribution de notre logiciel, nous avons développé un site web (Bourgeois et Villemaire, 2010b) pour accompagner la présentation et permettre le téléchargement. Le site contient aussi toute notre documentation ainsi que les contenus des présentations et articles que nous avons mentionnés plus haut. Nous avons aussi pris le temps de fournir quelques exemples supplémentaires afin de montrer l'attrait direct de SDDL.

## 6.3 Conclusion de notre analyse

Nous concluons que notre solution est efficace et qu'elle respecte, à différents niveaux, les critères que nous nous étions fixés au départ. Il s'agit d'une hiérarchie d'objets facile et extensible, accompagnée d'une syntaxe claire et fluide. Avec l'ajout de la gestion des erreurs et sa capacité à décrire plusieurs formes de structures, notre logiciel permet de simplifier la représentation des diagrammes classiques du domaine. Nous en venons donc à la conclusion que notre solution respecte les différents critères et nous pouvons donc considérer notre solution comme étant valide.

[Cette page a été laissée intentionnellement blanche]

# CONCLUSION

La création de diagrammes structurés, comme nous avons pu le constater tout au long de ce mémoire, est un sujet très complexe. Bien qu'il puisse, de prime abord, n'être vu que comme la capacité de faire des petits dessins simples, nous remarquons rapidement que les décrire tels que nous les voyons est beaucoup plus difficile qu'il y parait.

Entre autres, nous avons pu constater que les diagrammes sont pensés de manière extrêmement abstraites. En effet, dans un arbre, nous ne voyons pas une série de sommets reliés par des points, mais bel et bien un arbre seulement. Cette capacité à abstraire la forme du contenu est extrêmement pratique pour un être humain, mais pas très utile pour un ordinateur. Il est extrêmement difficile d'essayer de s'affranchir du besoin de spécifier la forme lorsqu'on veut créer un diagramme.

Il en est de même avec LaTeX, duquel nous nous sommes inspiré. Bien que le logiciel puisse faire la forme de manière adéquate dans plusieurs cas, ces derniers s'avèrent très spécifiques et, plus souvent qu'autrement, la possibilité d'obtenir un niveau accru de contrôle sur la forme est souhaitable. D'autre part, la question du positionnement d'un texte scientifique dans une page a été abondamment étudié (entre autre par la création de TeX), alors que le bon positionnement d'un diagramme l'a été beaucoup moins. Il est donc problématique de se débarrasser de cet élément de forme dans n'importe quel logiciel de création de diagrammes.

Cela nous amène donc à la question naturelle de savoir si notre logiciel s'avère une solution valide à notre problématique initiale. En effet, bien que notre analyse démontre que la création de dessins devient plus simple, nous ne pouvons affirmer que nous écrivons ce que nous pensons du diagramme.

Or, comme la question est de vérifier la validité de notre solution, nous répondrons par l'affirmative. Même si elle n'est pas parfaite, un degré de complexité se trouve éliminé malgré tout. Il est tout de même possible pour un utilisateur de penser en des termes plus abstraits, même s'il ne s'agit pas du niveau d'abstraction optimal. De plus, étant donné que nous avons créé notre solution pour être extensible, il est possible d'augmenter le niveau d'abstraction avec la contribution de la communauté. Bien que la solution ne soit pas parfaite et aurait pu jouir d'améliorations subséquentes, l'objectif principal était de développer un langage qui simplifiait la description de diagrammes structurés et cela fut atteint.

Naturellement, étant donné les limites de temps et de complexité, plusieurs pistes de développements supplémentaires n'ont pu être utilisées. En effet, nous nous sommes limités à la description d'une image statique. Il est toutefois très utile de pouvoir créer des images dynamiques. La possibilité de créer des séquences de mouvements à l'intérieur d'un diagramme aurait pu rendre notre outil plus intéressant. De plus, un outil de génération automatique de graphes ou d'arbres, même imparfait, pourrait fournir des choix supplémentaires à un utilisateur qui souhaite obtenir un résultat rapidement, sans toutefois rechercher un résultat parfait.

Il est toutefois bon de noter que, bien qu'il soit long et complexe d'effectuer de tels ajouts, notre logiciel a été pensé en fonction de pouvoir être étendu par l'utilisateur. Il est donc tout à fait possible de pouvoir, éventuellement, fournir de telles fonctionnalités dans une version future du programme. D'ici là, celui-ci suffira amplement à la création de diagrammes simples.

# APPENDICE A

## PRÉSENTATION DU PROJET POUR LE TEX USERS GROUP

Nous incluons ici l'article complet que nous avons rédigé à propos de notre logiciel afin d'en faire la promotion auprès de la communauté TEX et LATEX. Cet article, qui fut présenté dans `Tugboat`, la revue du TEX Users Group, tel que mentionné précédemment, complémente la présentation qui fut effectuée lors de la conférence du même groupe à San Francisco. L'article est essentiellement une introduction pragmatique pour un utilisateur final, et fournit donc plusieurs exemples intéressants d'utilisation tout en décrivant brièvement les différents concepts importants qui ont menés le développement de `SDDL`. Il s'agit de la version finale, tel que l'article apparait dans le numéro de `Tugboat`.

## Drawing structured diagrams with SDDL

Mathieu Bourgeois and Roger Villemaire

### Abstract

We present SDDL, a Structured Diagram Description Language aimed at producing graphical representations for discrete mathematics and computer science. SDDL allows combining graphical objects (circles, lines, arrows, ...) and LaTeX boxes to produce diagrams representing discrete structures such as graphs, trees, etc. with an easy-to-use domain specific language.

### 1 What is SDDL?

SDDL, *Structured Diagram Description Language*, is a high-level domain specific language tailored for diagrams that have an inherent structure. Examples of these might be trees, graphs and automata. Any diagram that is naturally structured can be described in SDDL. However, it was designed especially for data structures appearing in computer science and discrete mathematics.

The main objective of this language is to realize drawings in a natural and structured way. Mainly, one describes a drawing in SDDL in a way that is similar to drawing on a blackboard. For example, specific shapes (like circles, ellipses, texts, boxes) are drawn at positions that can be either absolute or relative to specific points on already placed shapes. Since we aim at mathematical drawings, text is handled through LaTeX.

Since exhibiting the structure of a diagram is essential to our purpose, SDDL uses an object-oriented hierarchy, completely written in Java, under our high-level language. Having shapes as objects makes structuring of diagrams easier and more intuitive since most shapes are explicitly represented by a class. Furthermore, extension by the end user is quite feasible since Java is a well-known language.

At the lower level, SDDL uses another graphical description language for LaTeX, namely Asymptote [1]. Our tool produces Asymptote code, which is finally converted to an encapsulated PostScript (eps) vector file.

### 2 Canvas and shapes

A diagram in SDDL is defined by a main `Canvas`. It is just like a standard painting canvas, in the sense that we can place (or paint) different things on it as much as we like. The things we can place inside a `Canvas` are `Shapes`. Thus, for creating a "Hello, world!" diagram, we would use the following:

```
put Text with [text = "Hello, world!"] in main;
```

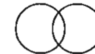
resulting in:

Hello, world!

In this example, `Text` is a `Shape`, and it possesses a property `text`, which is the LaTeX string used to render the text. Each `Shape` defines a certain number of such properties.

SDDL permits the definition of variables. It is a dynamically typed language, so variables don't have to be declared before being used. Variables make it possible to reuse the same `Shape` at multiple locations. For example,

```
a = Circle with [radius = 10.0];
put a at (-7.5, 0.0) in main;
put a at (7.5, 0.0) in main;
```



In this example, we specify the location where we want our `Shape` to be. If a location is not specified with the `at` clause, the `Shape` will be placed at the point of origin of the current `Canvas`, which is its center.

One important thing to note is that once a `Shape` has been put in a `Canvas`, it is immutable. It cannot be modified or removed. A `Shape` can be modified *after* it has been drawn on a `Canvas`, but this will have an effect only on later use of this object and will not modify the actual `Canvas` in any way; a `Shape` always appears in a `Canvas` as it was at the moment it was added.

When a `Shape` is created, every property specified by the user is set, in the given order. It is not necessary to specify all properties at once, nor to set all of them. Most of the properties have appropriate default values. However, some properties, if not set, will yield strange results. For example, a circle with no radius property set will have a default radius of 0. Usually, property order is irrelevant. For instance, giving for an ellipse the radius along the x-axis or the y-axis first will yield exactly the same result.

```
a = Ellipse;
a = a with [xradius = 20.0];
a = a with [yradius = 10.0];
put a in main;
```

### 3 Structuring multiple canvases

Every diagram consists of a *main* `Canvas`. However, we can define other `Canvas` objects if we wish. An additional `Canvas` can be introduced using a typical variable assignment. However, when a `put` command is used, SDDL checks to see if the canvas variable is

already defined. If it isn't, it automatically creates an empty Canvas for use.

One of the main reasons to use other Canvases is to create an explicit structure in our diagram. Since a Canvas is a Shape, once a sub-Canvas has been created, it can be placed inside the main Canvas, or any other one for that matter. This will ensure that everything that is defined inside our sub-Canvas will be placed at the proper position in the final diagram. However, any Canvas that has been defined, but is not linked directly or indirectly with the *main* Canvas, will not be drawn in the final diagram.

As an example, let's say we want to make a diagram that consists of two identical "eyes". Instead of defining two identical objects, we will create one in a Canvas named *form* and put it at different positions inside our *main* Canvas. Our "eye" consists of a circle and an ellipse, both with the same center. All we need to do for this is place them at the default position of the Canvas. Finally, we can take this sub-Canvas and place it at the two positions required.

```
a = Circle with [radius=10.0];
b = Ellipse with [xradius=20.0, yradius=10.0];
put a in form;
put b in form;
put form at (-20.0, 0.0) in main;
put form at (20.0, 0.0) in main;
```

## 4 Paths

SDDL offers support for the description of paths. The way in which they are described is similar in syntax with MetaPost and Asymptote, though with some variations. A Path shape is described by linking points together with specific linking symbols. To draw a line between two points, the line symbol `--` can be used. To link some points using a curve, use the curved line symbol `~`, which will use Asymptote's positioning algorithms to create a nice-looking curve which passes through those points. For the moment, user control over the curve is limited to beginning and ending tangents, but could be expanded.

```
p1 = a--b--c;
p2 = a~b~c;
```

One of the other things you may want to do with a Path is to create an Arrow out of it. SDDL defines symbols for forward, backward and bidirectional arrows for both linear and curved lines. Right now, however, support is restricted to forward arrows.

```
p1 = a<-b--c->d;
p2 = a<~b~c~>d;
p3 = a<->b;
p4 = a<~>b;
```

Once a Path is defined, it can be used as any other Shape. However, one of the main things that you want to do is to put your Path somewhere. For this, you can write something like this:

```
p = (0.0, 0.0) -- (10.0, 10.0);
put p in main;
```

This is the standard way to add Shapes to a Canvas. However, since Paths are usually used to link different Shapes together in the *main* Canvas, special handling is done in SDDL. Namely, a Path that is not explicitly assigned to a variable will be automatically placed in the *main* Canvas. Therefore the Path will be directly placed in *main* without any explicit put expression. Thus, linking has its own special syntax, which is more natural and convenient. As an example, this program

```
(0.0, 0.0) -- (10.0, 10.0);
```

yields exactly the same result as the one just above. Furthermore, points can be added and multiplied by a scalar (as vectors). Using a dot to locate the origin of the drawing, we can use the SDDL syntax to create this example:

```
a = (10.0, 10.0);
a--2.0*a->(30.0, 0.0);
a~(-1.0)*a~(30.0, 0.0);
```

Finally, as we will now see, Shapes also usually define specific points.

## 5 Drawing and linking shapes together

To draw a diagram, usually some shapes with a specific structure are first laid down. Once that is done, those objects are linked together with lines, curves and arrows. However, these links must be made between specific positions, usually derived from one or more specific Shapes. For example, we may want to link the northwestern point of a rectangle with the point on a circle at an angle of 45 degrees. These points could obviously be computed in advance. However, this becomes quite problematic when points are at peculiar angles or more complex relationships between points and Shapes are needed. Worst of all, if the position or any other property of a Shape is changed, all computations will have to be redone.

To ease object linking, SDDL provides so-called *reference points*. A reference point is a point that has no static value, unlike points defined by a pair of two real numbers. Instead, a reference point is defined by its relationship to a specific Shape appearing at a particular location. As a matter of fact, all reference

points are defined along with the Shape because they are a natural part of it. This ensures that we always have nice-looking lines at exactly the positions we want them to be.

However, a reference point's exact position in a Canvas will depend on the Shape's position. Worse, since the same Shape can appears at multiple locations in the same Canvas, we have to know which occurrence we are talking about!

Therefore, SDDL introduces a feature called a Drawing. A Drawing is an object that represents a Shape at a particular position, i.e. a specific occurrence of a Shape in a Canvas. Whenever a Shape is put inside a Canvas, a Drawing is returned and can be assigned to a variable. This gives a way to uniquely identify every occurrence of a Shape appearing in a Canvas. If the same Shape is placed twice, two different Drawings will be returned, each referring to a different occurrence of the same Shape.

```
a = Circle with [radius = 10.0];
d1 = put a at (-10.0, 0.0) in main;
d2 = put a at (10.0, 0.0) in main;
```

Once a drawing has been defined, a way to access its points is needed. SDDL defines a syntax for extracting points from drawings:

**coord of** ⟨*coord*⟩ ( ⟨*args*⟩ ) **in** ⟨*drawing*⟩

Here one specifies the kind of point to use (⟨*coord*⟩) and any particular arguments required to obtain it. The available points are Shape specific, so only those kinds of points which make sense for the particular Shape can be used. For instance one can get a point at a particular angle on a Circle. This is also an example where an additional argument is needed. For instance, anglePoint(45.0) will give the point at 45 degrees.

Finally, to reference the drawing from which we take the point, its drawing Path (a dot separated path) from the main diagram must be given.

```
a = Circle with [radius = 10.0];
d1 = put a at (-7.5, 0.0) in main;
d2 = put a at (7.5, 0.0) in main;
(coord of anglePoint(0.0) in main.d1) --
   (coord of anglePoint(180.0) in main.d2);
```

In this example, a line between two identical circles is drawn. The drawing Paths are simple here, since both drawings are made directly in *main*. Thus, only *main* followed by the drawing name is needed.

Simply giving a Drawing is not sufficient in order to make reference points non-ambiguous; complete drawing Paths are required, as we will now show. Let's go back to the two eyes we drew previously. The eye was defined by a circle and an ellipse, both put inside a sub-Canvas. Now, let's say we want to draw a line between the two circles. Each of those circles is defined in the sub-Canvas and they are, as a matter of fact, the same Drawing *a*. Adding a link between *a* and itself would add a link between the sub-Canvas' circle and itself (what could this mean?). Adding the sub-Canvas to the main Canvas twice, as we did before, would just duplicate this structure inside the main Canvas.

But since a Canvas is a Shape, we get a drawing when we put the sub-Canvas inside *main*. We can therefore identify each inner circle by listing the drawings required to access them, as shown in the following SDDL example. Thus, we can always link together Shapes that are deeply nested inside a sub-Canvas.

```
a = Circle with [radius=10.0];
b = Ellipse with [xradius=20.0, yradius=10.0];
c = put a in form;
put b in form;
f1 = put form at (-20.0, 0.0) in main;
f2 = put form at (20.0, 0.0) in main;
(coord of anglePoint(180.0) in main.f1.c) ->
  (coord of anglePoint(0.0) in main.f2.c);
```

## 6 Programming constructs and lists

SDDL also defines typical programming constructs. For instance lists are created and elements accessed in a syntax similar to most other dynamic languages. Since the content of a list is a general Java Object, you can create a list of objects that are not of the same type. Thus, an assignment like

```
l = [2.0, (0.0, 0.0), [], a--b];
```

is perfectly legal, even if not typically very useful! A more typical use of lists would be

```
list = [(0.0, 0.0), (10.0, 10.0)];
l = list[0];
l2 = list[1];
l--l2;
```

Lists are nice, but to use them properly, adequate programming constructs are needed, such as for and while loops. SDDL defines those, permitting us to draw arrays of Shapes or define points through a simple list.

The ability to use loops significantly simplifies many diagrams in computer science. In this example, we first define a list of points. Afterward, using a for loop, we link those points into a linear Path.

```
a = [(10.0, 10.0), (20.0, 10.0),
     (20.0, 20.0), (10.0, 20.0)];
for i from 0 to 2 do
```

```
  a[i]--a[i+1];
end
```

Alternatively, we could get the same result using the following while loop.

```
i = 0;
while i != 3 do
  a[i]--a[i+1];
  i = i + 1;
end
```

## 7 Functions

SDDL also permits the creation of functions inside the code. Those must be written before the diagram description. A SDDL function is defined with the keyword `fun`. The list of arguments does not need to be typed, only named. The return type (if it exists) does not need to be specified either. To call the function in the resulting code, a typical C-like call is used.

```
fun dropMany(s, i, v, n, c){
  for j from 1 to n do
    put s at i + j*v in c;
  end
}
circle = Circle with [radius = 5.0];
dropMany(circle, (0.0, 0.0),
         (5.0, 0.0), 10.0, main);
```

In this case, we define a function that drops many instances of a Shape *s* at different positions on a Canvas *c*. The positions of the Shape are determined by an initial point *i* and a translation vector *v*. Finally, the number of Shapes placed is also given as a parameter *n*. Each Shape is then placed at the initial point translated by the translation vector a certain number of times. This permits us to create many circles along one line.

One important point is the parameter for the Canvas in which we place the Shapes. However, we pass *main* as our parameter. The reason for this is that SDDL does not possess global variables. A function can only access its parameters and variables that have been defined inside the function. Thus, trying to place something in *main* from inside a function will yield an error message saying the variable *main* is unknown.

## 8 Primitives

SDDL, as mentioned before, is written in Java. One of the nice features of Java is its libraries. Many classes have been written for any number of different tasks and the number of functions and algorithms implemented is astonishing. To rewrite libraries that are already defined in Java or to link them one by one in SDDL seemed pointless. It is much nicer to directly use those functions, since they're already there.

Since Java is a reflexive language, in the sense that the program knows about itself and can modify itself at will, functions can be dynamically accessed at runtime. We already use this feature to simplify the definition of the SDDL interpreter: the interpreter doesn't have to know every possible type of Shape in order to work. Using these same mechanisms permits loading classes at runtime and gives the user the power to call from inside SDDL any Java function.

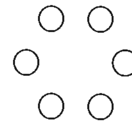
This is done through so-called primitives, which are defined using the keyword `primitive` followed by the name of the function, as it will appear in SDDL. Following that, two strings are given: one for the package in which the function is defined and one for the name of the function as it was defined in Java. Any primitive declaration must be made prior to any function definition.

Let's look at an example. One of the main things a diagram description language would require is some basic mathematics and geometry libraries. Java possesses a nice `java.lang.math` package which we would like to use, especially the `sin` and `cos` functions. For this, we will need to define two primitives.

```
primitive static sin "java.lang.Math" "sin";
primitive static cos "java.lang.Math" "cos";
```

Once they are defined, they can be called just like any other SDDL functions. In this case, we will use those functions to place circles around an invisible circle.

```
for i from 1 to 6 do
  put Circle with [radius = 5.0] at
          (20.0 * cos(i * 3.14 / 3.0),
           20.0 * sin(i * 3.14 / 3.0))
          in main;
end
```

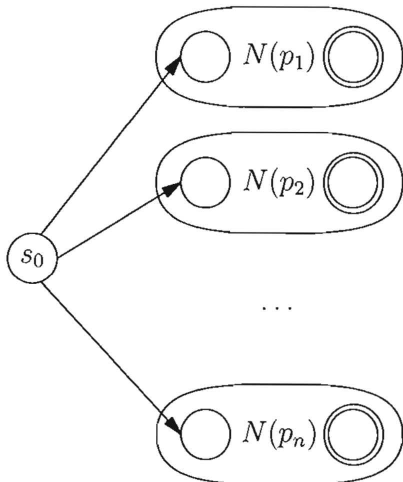


## 9 A concrete example

Now that we have all our tools, let's use SDDL to describe a simple diagram of an automaton. This is the complete code, along with the resulting image.

```
circle1 = Circle with [radius = 10.0];
circle2 = Circle with [radius = 12.0];
ellipse = Ellipse with [xradius = 50.0,
                        yradius = 20.0];
circle = put circle1 at (-30.0, 0.0) in form;
put circle1 at (30.0, 0.0) in form;
put circle2 at (30.0, 0.0) in form;
```

```
put ellipse in form;

d1 = put form at (100.0, 100.0) in main;
d2 = put form at (100.0, 50.0) in main;
d3 = put form at (100.0, -50.0) in main;
put Text with [text="$N(p_1)$"]
        at (100.0, 100.0) in main;
put Text with [text="$N(p_2)$"]
        at (100.0, 50.0) in main;
put Text with [text="\dots"]
        at (100.0,0.0) in main;
put Text with [text="$N(p_n)$"]
        at (100.0, -50.0) in main;

put Text with [text = "$s_0$"]
        at (0.0, 20.0) in main;
d4 = put circle1 at (0.0, 20.0) in main;

(coord of anglePoint(70.0) in main.d4)->
(coord of anglePoint(180.0) in main.d1.circle);
(coord of anglePoint(0.0) in main.d4)->
(coord of anglePoint(180.0) in main.d2.circle);
(coord of anglePoint(-70.0) in main.d4)->
(coord of anglePoint(180.0) in main.d3.circle);
```




## 10 Modifying the hierarchy by adding shapes

Using what has been described above, most diagrams can be created. However, many simplifications can be made and some domains may find it relevant to add classes specific to their trade. SDDL eases the modification of the Java hierarchy underneath the language by using reflexivity. Any user who respects some basic guidelines will be able to have its class work automatically in SDDL without any specific linking required.

To do this, every class must possess a certain number of properties. Those properties are defined through setters and getters. For instance, for the radius of a circle, these are called `setRadius` and `getRadius`. Any property that follows this rule will be accessible.

Furthermore, an empty constructor must also be defined that sets, as much as possible, default values for all properties of the defined `Shape`. There are also some small functions to be defined, like the `draw` and `obtainPath` functions. Any reference points must also be defined with a function to obtain them.

As an example, let's say we would like to add a class to represent a cloud. A cloud will have a number of "spikes" and a size. The class definition (without function definition) that we would require would be the following:

```
public class Cloud extends Shape{
  double size;
  int numberOfSpikes;
  public Cloud(){...}
  public void setSize(double size){...}
  public double getSize(){...}
  public void setNumberOfSpikes(int n){...}
  public int getNumberOfSpikes(){...}
  public void draw(AbstractPoint a,
                   PrintWriter w){...}
  public Path obtainPath(){...}
}
```

## 11 Future additions and availability

SDDL is available now at `http://www.info2.uqam.ca/~villemaire_r/Recherche/SDDL/`. It is functional, though not by any means complete. Many additional `Shape`s could be added (in particular tree and graph classes) and options could be added to existing classes. Development of the application continues for the time being and a more thorough version will be made available.

## References

[1] John C. Bowman and Andy Hammerlindl. Asymptote: A vector graphics language. *TUGboat: The Communications of the TEX Users Group*, 29:288–294, 2008.

◇ Mathieu Bourgeois
Université du Québec à Montréal
Montréal, Canada
`bourgeois dot mathieu dot 2 (at) courrier dot uqam dot ca`

◇ Roger Villemaire
Université du Québec à Montréal
Montréal, Canada
`villemaire dot roger (at) uqam dot ca`
`http://intra.info.uqam.ca/personnels/Members/villemaire_r`

# APPENDICE B

# PRÉSENTATION DE LA GRAMMAIRE DE SDDL

Nous présentons ici le contenu entier de la grammaire de SDDL, tel que réalisé avec SableCC. Tous les éléments du langage sont là, à l'exception de la réflexivité, dont les détails de code sont situés dans la partie Java de notre projet.

```
Package sddl.syntax;

Helpers

  any = [0..0xffff];

  tab = 9;
  lf = 10;
  cr = 13;

  upper = ['A'..'Z'];
  lower = ['a'..'z'];
  digit = ['0'..'9'];

  name_char = upper | lower | digit | '_';

  string_char = [[32..126] - '"'];

  dot = '.';

  not_star = [any-'*'];
  not_star_not_slash = [not_star - '/'];

Tokens
```

```
put_kwd = 'put';
at_kwd = 'at';
in_kwd = 'in';
fit_kwd = 'fit';
around_kwd = 'around';
if_kwd = 'if';
while_kwd = 'while';
true_kwd = 'true';
false_kwd = 'false';
then_kwd = 'then';
else_kwd = 'else';
for_kwd = 'for';
from_kwd = 'from';
end_kwd = 'end';
to_kwd = 'to';
by_kwd = 'by';
do_kwd = 'do';
with_kwd = 'with';
and_kwd = 'and';
or_kwd = 'or';
fun_kwd = 'fun';
return_kwd = 'return';
primitive_kwd = 'primitive';
call_kwd = 'call';
static_kwd = 'static';
coord_kwd = 'coord';
of_kwd = 'of';
get_kwd = 'get';

scaled_kwd = 'scaled';
slanted_kwd = 'slanted';
rotated_kwd = 'rotated';
translated_kwd = 'translated';


straight_line_kwd = '--';
curved_line_kwd = '~';
straight_fwd_arrow_kwd = '->';
curved_fwd_arrow_kwd = '~>';
straight_bwd_arrow_kwd = '<-';
curved_bwd_arrow_kwd = '<~';
straight_bidir_arrow_kwd = '<->';
curved_bidir_arrow_kwd = '<~>';
```

```
  plus = '+';
  minus = '-';
  mult = '*';
  div = '/';
  eq = '==';
  neq = '!=';
  lt = '<';
  gt = '>';
  leq = '<=';
  geq = '>=';
  not = '!';
  assign = '=';
  dot = dot;

  l_br = '{';
  r_br = '}';
  l_par = '(';
  r_par = ')';
  l_cr = '[';
  r_cr = ']';
  sc = ';';
  comma = ',';

  id = lower name_char*;
  class_name = upper name_char*;

  integer = ('-')? digit+;
  real = ('-')? digit+ dot digit+;

  string = '"' string_char* '"' | '"' '"' '"';

  blank = ' ' | tab | cr | lf | cr lf;
  comment = '/*' not_star* ('*' (not_star_not_slash not_star*)?)* '*/';

Ignored Tokens

  blank, comment;

Productions

  file =
    [primitives]:primitive* [functions]:function* [stms]:stm*;
```

```
primitive =
  primitive_kwd static_kwd? id [class_name]:string
  [func_name]:string signature? sc;

signature =
  l_par type_list? r_par;

type_list =
  string [additional_types]:additional_type*;

additional_type =
  comma string;

function =
  fun_kwd id l_par id_list? r_par l_br [stms]:stm* r_br;

id_list =
  id [additional_ids]:additional_id*;

additional_id =
  comma id;

stm =
  {exp} exp sc |
  {assign} id assign exp sc |
  {while} while_kwd exp do_kwd [stms]:stm* end_kwd sc? |
  {if} if_kwd exp then_kwd [stms]:stm* else_part? end_kwd sc? |
  {for} for_kwd id from_kwd [low_bound]:exp to_kwd [up_bound]:exp
        do_kwd [stms]:stm* end_kwd sc? |
  {return} return_kwd exp sc;

else_part =
  else_kwd [stms]:stm*;

exp =
  {or} exp or_kwd and_exp |
  {simple} and_exp ;

and_exp =
  {and} and_exp and_kwd line_exp |
  {simple} line_exp;

line_exp =
  {line} line_exp line_op eq_exp with_part? |
```

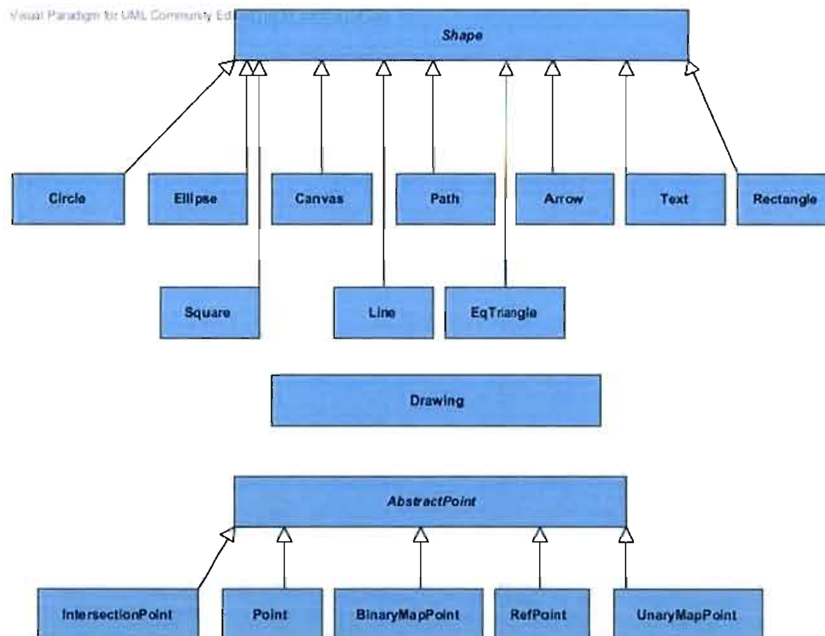
```
  {simple} eq_exp;

line_op =
  {straight_line} straight_line_kwd |
  {curved_line} curved_line_kwd |
  {straight_fwd_arrow} straight_fwd_arrow_kwd |
  {curved_fwd_arrow} curved_fwd_arrow_kwd |
  {straight_bwd_arrow} straight_bwd_arrow_kwd |
  {curved_bwd_arrow} curved_bwd_arrow_kwd |
  {straight_bidir_arrow} straight_bidir_arrow_kwd |
  {curved_bidir_arrow} curved_bidir_arrow_kwd;

eq_exp =
  {eq} [left]:cmp_exp eq [right]:cmp_exp |
  {neq} [left]:cmp_exp neq [right]:cmp_exp |
  {simple} cmp_exp;

cmp_exp =
  {lt} [left]:add_exp lt [right]:add_exp |
  {gt} [left]:add_exp gt [right]:add_exp |
  {leq} [left]:add_exp leq [right]:add_exp |
  {geq} [left]:add_exp geq [right]:add_exp |
  {simple} add_exp;

add_exp =
  {add} add_exp plus mul_exp |
  {minus} add_exp minus mul_exp |
  {simple} mul_exp;

mul_exp =
  {mult} mul_exp mult left_unary_exp |
  {div} mul_exp div left_unary_exp |
  {simple} left_unary_exp;

left_unary_exp =
  {not} not left_unary_exp |
  {get} get_kwd id from_kwd left_unary_exp |
  {put} put_kwd exp at_part? in_kwd left_unary_exp |
  {fit} fit_kwd class_name around_kwd left_unary_exp |
  {coord} coord_kwd of_kwd id l_par
          arg_list? r_par in_kwd left_unary_exp |
  {simple} right_exp;

at_part =
```

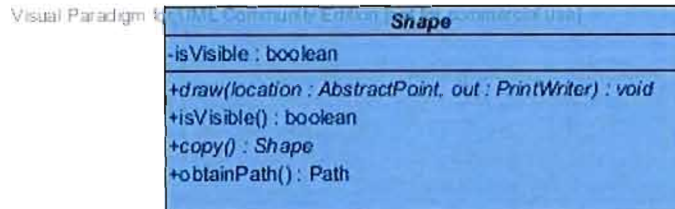
```
  at_kwd exp;

right_exp =
  {call} right_exp dot id l_par arg_list? r_par |
  {index} right_exp l_cr exp r_cr |
  {scaled} right_exp scaled_kwd l_par eq_exp r_par |
  {slanted} right_exp slanted_kwd l_par
            [xslant]:exp comma [yslant]:eq_exp r_par |
  {rotated} right_exp rotated_kwd l_par eq_exp r_par |
  {translated} right_exp translated_kwd l_par [xtranslate]:term
               comma [ytranslate]:eq_exp r_par |
  {simple} term;

term =
  {par} l_par exp r_par |
  {point} l_par [x_value]:exp comma [y_value]:exp r_par |
  {drawing} class_name with_part |
  {var} id |
  {int} integer |
  {real} real |
  {true} true_kwd |
  {false} false_kwd |
  {string} string |
  {list} l_cr arg_list? r_cr |
  {static_call} id l_par arg_list? r_par;

arg_list =
  exp [additional_args]:additional_arg*;

additional_arg =
  comma exp;

with_part =
  with_kwd l_cr options? r_cr;

options =
  option [additional_options]:additional_option*;

additional_option =
  comma option;

option =
  id assign exp;
```

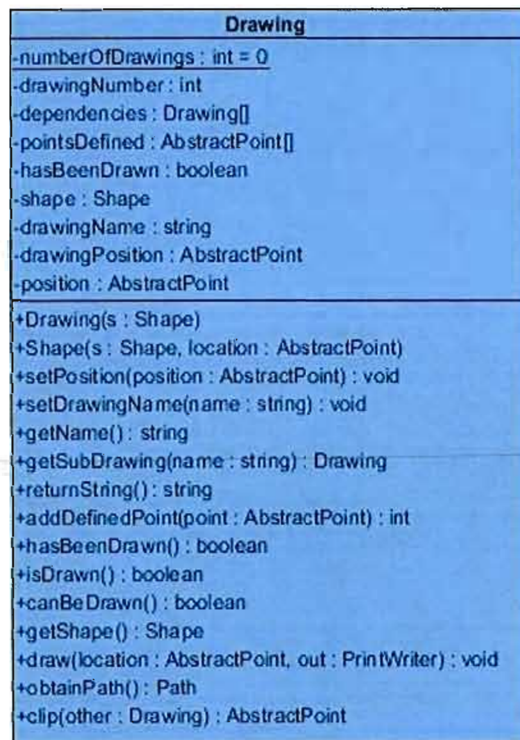
# APPENDICE C

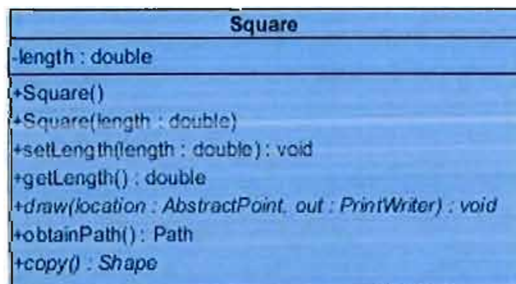
# PRÉSENTATION DE LA HIÉRARCHIE

Nous présentons ici la description complète de notre hiérarchie objet en Java. Le tout est décrit à l'aide d'un diagramme UML. Toutefois, afin de faciliter le formatage, nous avons séparé le tout en deux sections. Le premier diagramme contient les noms des classes ainsi que leurs relations d'héritage. Par après, tous les diagrammes suivants représentent le contenu spécifique de chaque classe, sans les relations entre classes. Ceci nous permet d'obtenir une vue d'ensemble des classes, tout en obtenant des détails cruciaux sur l'implémentation de chaque classe.




Visual Paradigm for UML Community Edition [not for commercial use]
Shape
-isVisible : boolean
+draw(location : AbstractPoint, out : PrintWriter) : void
+isVisible() : boolean
+copy() : Shape
+obtainPath() : Path

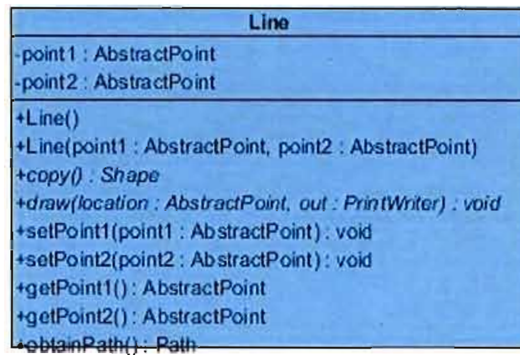

Drawing
-numberOfDrawings : int = 0
-drawingNumber : int
-dependencies : Drawing[]
-pointsDefined : AbstractPoint[]
-hasBeenDrawn : boolean
-shape : Shape
-drawingName : string
-drawingPosition : AbstractPoint
-position : AbstractPoint
+Drawing(s : Shape)
+Shape(s : Shape, location : AbstractPoint)
+setPosition(position : AbstractPoint) : void
+setDrawingName(name : string) : void
+getName() : string
+getSubDrawing(name : string) : Drawing
+returnString() : string
+addDefinedPoint(point : AbstractPoint) : int
+hasBeenDrawn() : boolean
+isDrawn() : boolean
+canBeDrawn() : boolean
+getShape() : Shape
+draw(location : AbstractPoint, out : PrintWriter) : void
+obtainPath() : Path
+clip(other : Drawing) : AbstractPoint

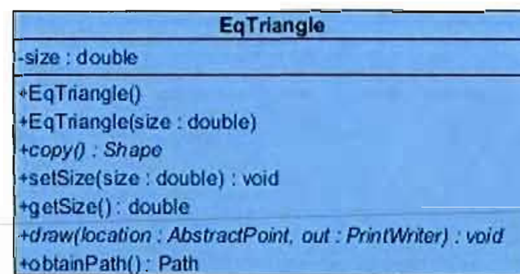

Square
-length : double
+Square()
+Square(length : double)
+setLength(length : double) : void
+getLength() : double
+draw(location : AbstractPoint, out : PrintWriter) : void
+obtainPath() : Path
+copy() : Shape

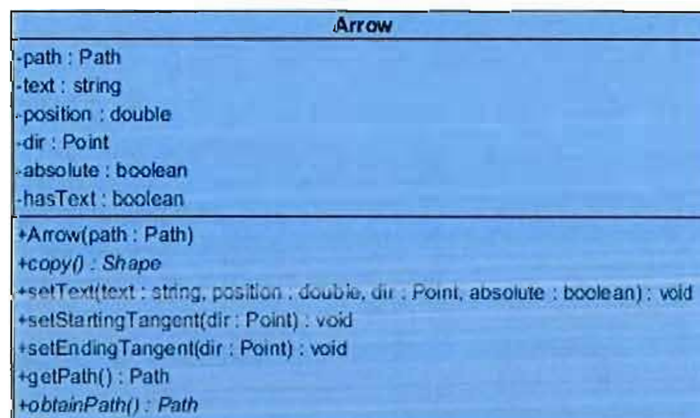
| Rectangle |
| --- |
| -length : double<br>-width : double<br>-round : boolean |
| +Rectangle()<br>+Rectangle(length : double, width : double)<br>+setLength(length : double) : void<br>+setWidth(width : double) : void<br>+setRound(round : boolean) : void<br>+getLength() : double<br>+getWidth() : double<br>+getRound() : boolean<br>+getNorth(d : Drawing) : RefPoint<br>+getSouth(d : Drawing) : RefPoint<br>+getWest(d : Drawing) : RefPoint<br>+getEast(d : Drawing) : RefPoint<br>+getNorthwest(d : Drawing) : RefPoint<br>+getNortheast(d : Drawing) : RefPoint<br>+getSouthwest(d : Drawing) : RefPoint<br>+getSoutheast(d : Drawing) : RefPoint<br>*+draw(location : AbstractPoint, out : PrintWriter) : void*<br>+obtainPath() : Path |

| Circle |
| --- |
| -radius : double |
| +Circle()<br>+Circle(radius : double)<br>*+copy() : Shape*<br>+setRadius(radius : double) : void<br>+getRadius() : double<br>+getAnglePoint(d : Drawing, angle : double) : RefPoint<br>*+draw(location : AbstractPoint, out : PrintWriter) : void*<br>+obtainPath() : Path |

| Ellipse |
| --- |
| -xradius : double<br>-yradius : double |
| +Ellipse()<br>*+copy() : Shape*<br>+setXRadius(r : double) : void<br>+setYRadius(r : double) : void<br>+getXRadius() : double<br>+getYRadius() : double<br>*+draw(location : AbstractPoint, out : PrintWriter) : void*<br>+obtainPath() : Path |


Line
-point1 : AbstractPoint
-point2 : AbstractPoint
+Line()
+Line(point1 : AbstractPoint, point2 : AbstractPoint)
+copy() : Shape
+draw(location : AbstractPoint, out : PrintWriter) : void
+setPoint1(point1 : AbstractPoint) : void
+setPoint2(point2 : AbstractPoint) : void
+getPoint1() : AbstractPoint
+getPoint2() : AbstractPoint
+obtainPath() : Path


EqTriangle
-size : double
+EqTriangle()
+EqTriangle(size : double)
+copy() : Shape
+setSize(size : double) : void
+getSize() : double
+draw(location : AbstractPoint, out : PrintWriter) : void
+obtainPath() : Path


Arrow
-path : Path
-text : string
-position : double
-dir : Point
-absolute : boolean
-hasText : boolean
+Arrow(path : Path)
+copy() : Shape
+setText(text : string, position : double, dir : Point, absolute : boolean) : void
+setStartingTangent(dir : Point) : void
+setEndingTangent(dir : Point) : void
+getPath() : Path
+obtainPath() : Path

| Canvas |
| --- |
| -canvasList : Canvas[] |
| -canvasPointList : AbstractPoint[] |
| -drawingList : Drawing[] |
| -drawingPointList : AbstractPoint[] |
| -isMain : boolean |
| -canvasPosition : AbstractPoint |
| +Canvas() |
| *+copy() : Shape* |
| +setCanvasPosition(position : AbstractPoint) : void |
| +setMain() : void |
| +isMain() : boolean |
| +addShape(s : Shape, location : AbstractPoint) : Drawing |
| +addShape(s : Shape, location : AbstractPoint, name : string) : Drawing |
| +getDrawing(i : int) : Drawing |
| +getDrawing(name : string) : Drawing |
| +getDrawingLocation(name : string) : AbstractPoint |
| +getDrawingOrNull(name : string) : Drawing |
| +getDrawingLocation(i : int) : AbstractPoint |
| +getNumberOfDrawings() : int |
| *+draw(location : AbstractPoint, out : PrintWriter) : void* |
| +drawMain(out : PrintWriter) : void |
| +drawCanvas(location : AbstractPoint, remainingDraws : Drawing [], out : PrintWriter) : void |

| Path |
| --- |
| -list : AbstractPoint[] |
| -hasAStartingTangent : boolean |
| -hasAEndingTangent : boolean |
| -startingTangent : Point |
| -endingTangent : Point |
| -isLinear : boolean |
| +Path() |
| +Path(list : AbstractPoint []) |
| +Path(list : AbstractPoint [], startingTangent : Point, endingTangent : Point) |
| *+copy() : Shape* |
| +addPoint(point : AbstractPoint) : void |
| +setLinear() : void |
| +isLinear() : boolean |
| +getPoint(i : int) : AbstractPoint |
| +getPoints() : AbstractPoint [] |
| +getNumberOfPoints() : int |
| +cycle() : void |
| +hasAStartingTangent() : boolean |
| +hasAEndingTangent() : boolean |
| +getStartingTangent() : Point |
| +getEndingTangent() : Point |
| +setStartingTangent(startingTangent : Point) : void |
| +setEndingTangent(endingTangent : Point) : void |
| +pathValue() : string |
| *+draw(location : AbstractPoint, out : PrintWriter) : void* |

**Text**

-text : string

+Text()
+Text(text : string)
+setText(text : string) : void
+getText() : string
*+draw(location : AbstractPoint, out : PrintWriter) : void*
+obtainPath() : Path
*+copy() : Shape*

***AbstractPoint***

+add(second : AbstractPoint) : AbstractPoint
+multiply(multiplier : double) : AbstractPoint
+combine(percentage : double, second : AbstractPoint) : AbstractPoint
*+solve() : string*
*+copy() : AbstractPoint*

**Point**

-x : double
-y : double

+Point()
+Point(x : double, y : double)
*+solve() : string*
+toString() : string
*+copy() : AbstractPoint*

**RefPoint**

-referenceDrawing : Drawing
-pointNumber : int

+RefPoint(referenceDrawing : Drawing, pointNumber : int)
*+solve() : string*
*+copy() : AbstractPoint*

**IntersectionPoint**

-p : Path
-q : Path

+IntersectionPoint(p : Path, q : Path)
*+copy() : AbstractPoint*
*+solve() : string*

| UnaryMapPoint |
| --- |
| -point : AbstractPoint<br>-preString : string<br>-endString : string<br>-value : double |
| +UnaryMapPoint(point : AbstractPoint)<br>*+solve() : string*<br>+setMapTypeMultiply(value : double) : void<br>*+copy() : AbstractPoint* |

| BinaryMapPoint |
| --- |
| -firstPoint : AbstractPoint<br>-secondPoint : AbstractPoint<br>-preString : string<br>-middleString : string<br>-endString : string |
| +BinaryMapPoint(firstPoint : AbstractPoint, secondPoint : AbstractPoint)<br>*+copy() : AbstractPoint*<br>*+solve() : string*<br>+setMapTypeAdd() : void |

[Cette page a été laissée intentionnellement blanche]

# APPENDICE D

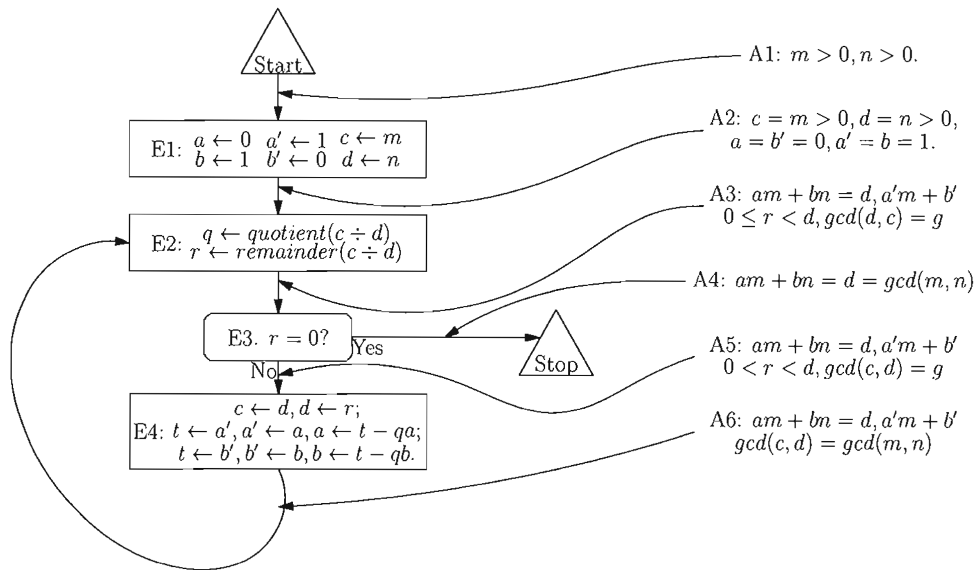
# PRÉSENTATION DES DIAGRAMMES CHOISIS

Cet annexe comprend l'ensemble des quinze (15) diagrammes que nous avons choisis comme base de qualité pour notre projet. Tel que mentionné précédemment, nous avons inclus à la fois le code de chaque diagramme en SDDL ainsi que le diagramme résultant. Aucune analyse n'est effectué pour ces dessins, à l'exception de ceux déjà analysés au chapitre 6.

```
/* Start box */
triangle = EqTriangle with [size = 40.0];
put triangle at (0.0, -10.0) in main;
put Text with [text = "Start"] at (0.0, -5.0) in main;

/* E1 box */
rectanglePos = (0.0, -50.0);
rectangle = Rectangle with [length = 160.0, width = 30.0];
put rectangle at rectanglePos in main;
put Text with [text = "E1:"] at rectanglePos - (60.0, 0.0) in main;
put Text with [text = "$a \leftarrow 0$"]
    at rectanglePos + (-30.0, 5.0) in main;
put Text with [text = "$b \leftarrow 1$"]
    at rectanglePos + (-30.0, -5.0) in main;
put Text with [text = "$a' \leftarrow 1$"]
    at rectanglePos + (10.0, 5.0) in main;
put Text with [text = "$b' \leftarrow 0$"]
    at rectanglePos + (10.0, -5.0) in main;
put Text with [text = "$c \leftarrow m$"]
    at rectanglePos + (50.0, 5.0) in main;
put Text with [text = "$d \leftarrow n$"]
```

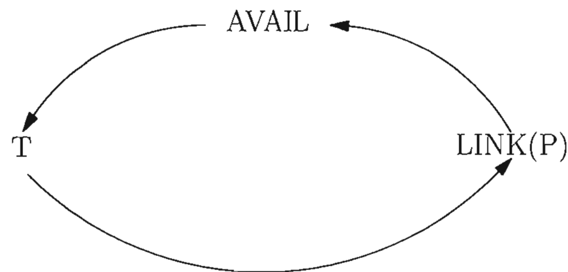
```
    at rectanglePos + (50.0, -5.0) in main;

/* E2 box */
rectangle2Pos = (0.0, -100.0);
put rectangle at rectangle2Pos in main;
put Text with [text = "E2:"] at rectangle2Pos - (60.0, 0.0) in main;
put Text with [text = "$q \leftarrow quotient(c \div d)$"]
    at rectangle2Pos + (10.0, 5.0) in main;
put Text with [text = "$r \leftarrow remainder(c \div d)$"]
    at rectangle2Pos + (10.0, -5.0) in main;

/* Round E3 box */
rectangle3Pos = (0.0, -150.0);
roundRectangle = Rectangle with [length = 80.0, width = 25.0,
                                 round = true];
put roundRectangle at rectangle3Pos in main;
put Text with [text = "E3. $r = 0$?"] at rectangle3Pos in main;

/* Stop box */
stopPos = (150.0, -170.0);
put triangle at stopPos in main;
put Text with [text = "Stop"] at stopPos + (0.0, 5.0) in main;

/* E4 box */
rectangle4Pos = (0.0, -200.0);
put Rectangle with [length = 160.0, width = 40.0]
    at rectangle4Pos in main;
put Text with [text = "E4:"] at rectangle4Pos + (-70.0, 0.0) in main;
put Text with [text = "$c \leftarrow d, d \leftarrow r$;"]
    at rectangle4Pos + (10.0, 12.0) in main;
put Text with
    [text = "$t \leftarrow a', a' \leftarrow a, a \leftarrow t - qa$;"]
    at rectangle4Pos + (10.0, 0.0) in main;
put Text with
    [text = "$t \leftarrow b', b' \leftarrow b, b \leftarrow t - qb$."]
    at rectangle4Pos + (10.0, -12.0) in main;

(0.0, -10.0) -> rectanglePos + (0.0, 15.0);
rectanglePos + (0.0, -15.0) -> rectangle2Pos + (0.0,15.0);
rectangle2Pos + (0.0, -15.0) -> rectangle3Pos + (0.0,12.5);
rectangle3Pos + (0.0, -12.5) -> rectangle4Pos + (0.0,20.0);
rectangle4Pos + (0.0, -20.0) -- rectangle4Pos + (0.0, -50.0) --
    rectangle4Pos + (-100.0, -50.0) -- rectangle2Pos + (-100.0, 0.0) ->
    rectangle2Pos + (-80.0, 0.0);
```

```
rectangle3Pos + (40.0, 0.0) -> rectangle3Pos + (141.0, 0.0);
put Text with [text = "Yes"] at rectangle3Pos + (48.0, -5.0) in main;
put Text with [text = "No"] at rectangle3Pos + (-8.0, -17.5) in main;

/* Placing text notes */
put Text with [text = "A1: $m > 0, n > 0$."] at (300.0, 0.0) in main;
put Text with [text = "A2: $c=m>0, d=n>0,$"] at (300.0, -34.0) in main;
put Text with [text = "$a=b'=0, a'=b=1$."] at (300.0, -46.0) in main;
put Text with [text = "A3: $am + bn = d, a'm + b'$"]
    at (300.0, -74.0) in main;
put Text with [text = "$0 \leq r < d, gcd(d,c) = g$"]
    at (300.0, -86.0) in main;
put Text with [text = "A4: $am + bn = d = gcd(m, n)$"]
    at (300.0, -120.0) in main;
put Text with [text = "A5: $am + bn = d, a'm + b'$"]
    at (300.0, -154.0) in main;
put Text with [text = "$0 < r < d, gcd(c,d) = g$"]
    at (300.0, -166.0) in main;
put Text with [text = "A6: $am + bn = d, a'm + b'$"]
    at (300.0, -194.0) in main;
put Text with [text = "$gcd(c, d) = gcd(m, n)$"]
    at (300.0, -206.0) in main;

(250.0, 0.0)~(200.0, -5.0)~(150.0, -15.0)~>(0.0, -20.0)
    with [startingTangent = (-1.0, 0.0), endingTangent = (-1.0, 0.0)];
(230.0, -40.0)~(200.0, -45.0)~(150.0, -65.0)~>(0.0, -70.0)
    with [startingTangent = (-1.0, 0.0), endingTangent = (-1.0, 0.0)];
(230.0, -80.0)~(200.0, -85.0)~(150.0, -115.0)~>(0.0, -120.0)
    with [startingTangent = (-1.0, 0.0), endingTangent = (-1.0, 0.0)];
(220.0, -120.0)~(200.0, -120.0)~(150.0, -125.0)
    ~>rectangle3Pos + (90.0, 0.0)
    with [startingTangent = (-1.0, 0.0), endingTangent = (-1.0, -1.0)];
(225.0, -160.0)~(200.0, -170.0)~(150.0, -190.0)
    ~(100.0, -175.0)~>(0.0, -170.0)
    with [startingTangent = (-1.0, 0.0), endingTangent = (-1.0, 0.0)];
(225.0, -200.0)~(150.0, -220.0)~>(0.0, -240.0)
    with [startingTangent = (-1.0, 0.0), endingTangent = (-1.0, 0.0)];
```



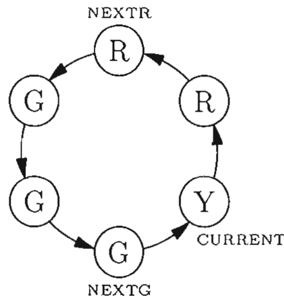

```
primitive static sin "java.lang.Math" "sin";
primitive static cos "java.lang.Math" "cos";
primitive static sqrt "java.lang.Math" "sqrt";

fun ellipse(angle, length, width){
  return (length * cos(angle * 2.0 * 3.1415 / 360.0),
          width * sin(angle * 2.0 * 3.1415 / 360.0));
}

fun ellipseDerive(angle, length, width){
  length2 = length * length;
  width2 = width * width;
  sinus = sin(angle * 3.1415 / 180.0);
  sinus2 = sinus * sinus;
  cosinus = cos(angle * 3.1415 / 180.0);
  cosinus2 = cosinus * cosinus;
  x = -1.0 * (length * sinus) /
      sqrt(width2 * cosinus2 + length2 * sinus2);
  y = (width * cosinus) / sqrt(width2 * cosinus2 + length2 * sinus2);
  return (x,y);
}

avail = Text with [text="AVAIL"];
link = Text with [text="LINK(P)"];
t = Text with [text="T"];
```

```
put avail at ellipse(90.0, 100.0, 50.0) in main;
put t at ellipse(180.0, 100.0, 50.0) in main;
put link at ellipse(0.0, 100.0, 50.0) in main;

(ellipse(7.0, 100.0, 50.0))~(ellipse(45.0, 100.0, 50.0))
    ~>(ellipse(75.0, 100.0, 50.0));
(ellipse(105.0, 100.0, 50.0))~(ellipse(135.0, 100.0, 50.0))
    ~>(ellipse(173.0, 100.0, 50.0));
(ellipse(193.0, 100.0, 50.0))~(ellipse(270.0, 100.0, 50.0))
    ~>(ellipse(355.0, 100.0, 50.0));
```




```
primitive static sin "java.lang.Math" "sin";
primitive static cos "java.lang.Math" "cos";

fun circular(angle, radius){
  return (radius * cos(3.1415 * angle / 180.0),
          radius * sin(3.1415 * angle / 180.0));
}

circleradius = 40.0;
circle = Circle with [radius=10.0];
g = Text with [text="G"];
r = Text with [text="R"];
y = Text with [text="Y"];

put circle at (circular(30.0, circleradius)) in main;
put r at (circular(30.0, circleradius)) in main;
put circle at (circular(90.0, circleradius)) in main;
put r at (circular(90.0, circleradius)) in main;
put circle at (circular(150.0, circleradius)) in main;
put g at (circular(150.0, circleradius)) in main;
put circle at (circular(210.0, circleradius)) in main;
put g at (circular(210.0, circleradius)) in main;
put circle at (circular(270.0, circleradius)) in main;
```

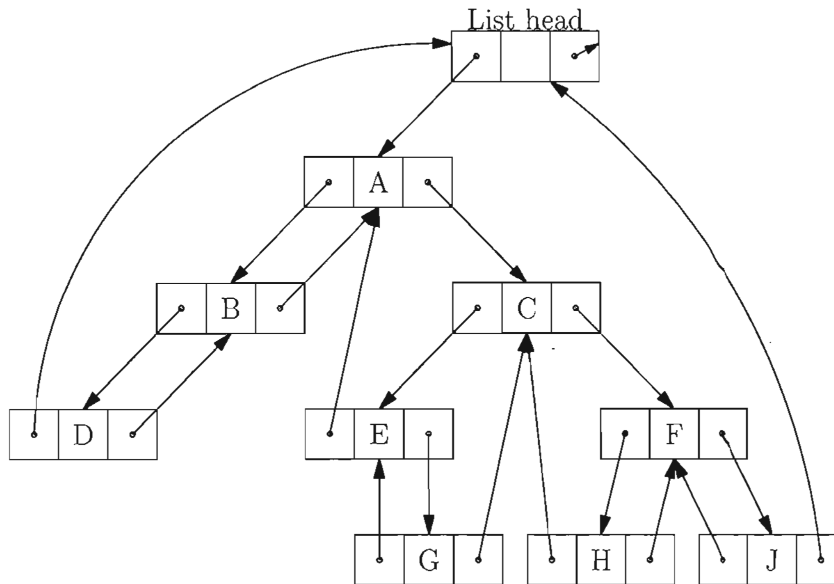
```
put g at (circular(270.0, circleradius)) in main;
put circle at (circular(330.0, circleradius)) in main;
put y at (circular(330.0, circleradius)) in main;

(circular(45.0, circleradius))~(circular(60.0, circleradius))
    ~>(circular(75.0, circleradius));
(circular(105.0, circleradius))~(circular(120.0, circleradius))
    ~>(circular(135.0, circleradius));
(circular(165.0, circleradius))~(circular(180.0, circleradius))
    ~>(circular(195.0, circleradius));
(circular(225.0, circleradius))~(circular(240.0, circleradius))
    ~>(circular(255.0, circleradius));
(circular(285.0, circleradius))~(circular(300.0, circleradius))
    ~>(circular(315.0, circleradius));
(circular(345.0, circleradius))~(circular(0.0, circleradius))
    ~>(circular(15.0, circleradius));

put Text with [text="\tiny{NEXTR}"]
    at (circular(90.0, circleradius)) + (0.0, 15.0) in main;
put Text with [text="\tiny{NEXTG}"]
    at (circular(-90.0, circleradius)) + (0.0, -15.0) in main;
put Text with [text="\tiny{CURRENT}"]
    at (circular(-30.0, circleradius)) + (15.0, -15.0) in main;
```




```
fun left(d){

  return 0.16 * coord of west in d.rectangle +
         0.84 * coord of east in d.rectangle;

}

fun right(d){

  return 0.16 * coord of east in d.rectangle +
```

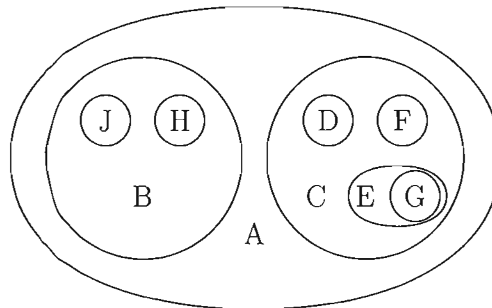
```
        0.84 * coord of west in d.rectangle;
}

rectangle1 = Rectangle with [length=60.0, width=20.0];
rectangle2 = Rectangle with [length=20.0, width=20.0];
circle = Circle with [radius=1.0];
put rectangle1 in form;
rectangle = put rectangle2 in form;
put circle at (-20.0, 0.0) in form;
put circle at (20.0, 0.0) in form;

d1 = put form at (0.0, 0.0) in main;
d2 = put form at (-60.0, -50.0) in main;
put Text with [text="A"] at (-60.0, -50.0) in main;
d3 = put form at (-120.0, -100.0) in main;
put Text with [text="B"] at (-120.0, -100.0) in main;
d4 = put form at (-180.0, -150.0) in main;
put Text with [text="D"] at (-180.0, -150.0) in main;
d5 = put form at (-60.0, -150.0) in main;
put Text with [text="E"] at (-60.0, -150.0) in main;
d6 = put form at (0.0, -100.0) in main;
put Text with [text="C"] at (0.0, -100.0) in main;
d7 = put form at (60.0, -150.0) in main;
put Text with [text="F"] at (60.0, -150.0) in main;
d8 = put form at (-40.0, -200.0) in main;
put Text with [text="G"] at (-40.0, -200.0) in main;
d9 = put form at (30.0, -200.0) in main;
put Text with [text="H"] at (30.0, -200.0) in main;
d10 = put form at (100.0, -200.0) in main;
put Text with [text="J"] at (100.0, -200.0) in main;

(left(d1))~>(coord of north in d2.rectangle);
(right(d1))~>(coord of east in d1.rectangle + (0.0, 5.0));
(left(d2))~>(coord of north in d3.rectangle);
(right(d2))~>(coord of north in d5.rectangle);
(left(d3))~>(coord of north in d4.rectangle);
(right(d3))~>(coord of south in d2.rectangle);
(left(d4))~>(coord of west in d1.rectangle + (0.0, 5.0));
(right(d4))~>(coord of south in d3.rectangle);
(left(d5))~>(coord of north in d6.rectangle);
(right(d5))~>(coord of north in d7.rectangle);
(left(d6))->(coord of south in d2.rectangle);
(right(d6))~>(coord of north in d8.rectangle);
```

```
(left(d7))~>(coord of north in d9.rectangle);
(right(d7))~>(coord of north in d10.rectangle);
(left(d8))->(coord of south in d6.rectangle);
(right(d8))->(coord of south in d5.rectangle);
(left(d9))->(coord of south in d5.rectangle);
(right(d9))->(coord of south in d7.rectangle);
(left(d10))->(coord of south in d7.rectangle);
(right(d10))~>(coord of south in d1.rectangle + (10.0, 0.0));

put Text with [text="List head"]
    at coord of north in d1.rectangle + (0.0, 5.0) in main;
```



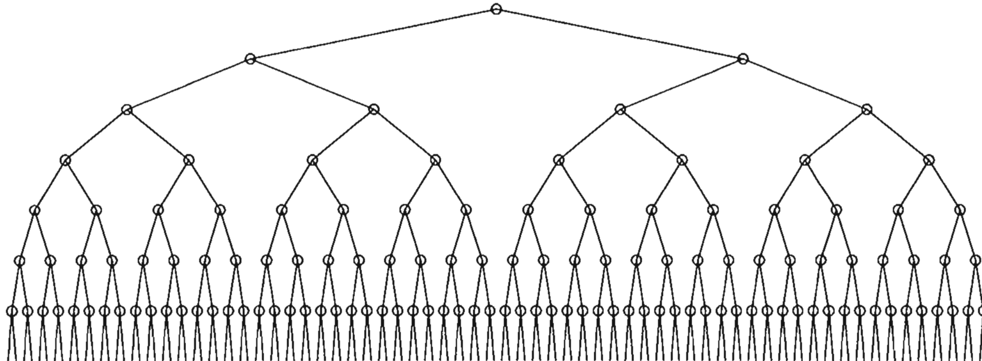

```
circle = Circle with [radius = 10.0];
bigcircle = Circle with [radius = 40.0];
ellipse = Ellipse with [xradius = 20.0, yradius = 12.0];
bigellipse = Ellipse with [xradius = 100.0, yradius = 60.0];

put bigcircle in canvas1;
put circle at (15.0, 15.0) in canvas1;
put Text with [text = "H"] at (15.0, 15.0) in canvas1;
put circle at (-15.0, 15.0) in canvas1;
put Text with [text = "J"] at (-15.0, 15.0) in canvas1;
put Text with [text = "B"] at (0.0, -15.0) in canvas1;

put bigcircle in canvas2;
```

```
put circle at (-15.0, 15.0) in canvas2;
put Text with [text = "D"] at (-15.0, 15.0) in canvas2;
put circle at (15.0, 15.0) in canvas2;
put Text with [text = "F"] at (15.0, 15.0) in canvas2;
put Text with [text = "C"] at (-20.0, -15.0) in canvas2;
put ellipse at (13.0, -15.0) in canvas2;
put Text with [text = "E"] at (0.0, -15.0) in canvas2;
put circle at (20.0, -15.0) in canvas2;
put Text with [text = "G"] at (20.0, -15.0) in canvas2;


put canvas1 at (-45.0, 0.0) in main;
put canvas2 at (45.0, 0.0) in main;
put Text with [text = "A"] at (0.0, -30.0) in main;
put bigellipse in main;
```

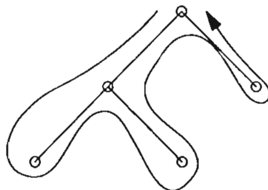



```
fun drawBinaryTree(d, s, o, dh, dv, n)
{

put s at o in d;
put o~(o + (-1.0 *  dh, -1.0 * dv)) in d;
put o~(o + (dh, -1.0 * dv)) in d;
if (n == 1)
then
return 0;
else
drawBinaryTree(d, s, o + (-1.0 * dh, -1.0 * dv), dh / 2.0, dv, n - 1);
drawBinaryTree(d, s, o + (dh, -1.0 * dv), dh / 2.0, dv, n - 1);
end

}

(0.0,0.0)--(0.0,0.0);
```

```
origin = (0.0, 0.0);
dv = 20.0;
dh = 100.0;
bullet = Circle with [radius = 2.0];
drawBinaryTree(main, bullet, origin, dh, dv, 7);
```

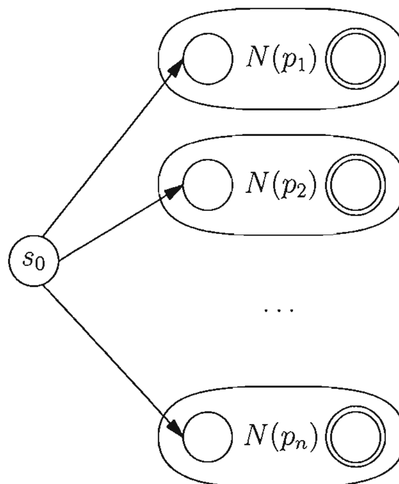


```
bullet = Circle with [radius = 2.0];
put bullet in main;
put bullet at (-30.0, -30.0) in main;
put bullet at (-60.0, -60.0) in main;
put bullet at (0.0, -60.0) in main;
put bullet at (30.0, -30.0) in main;

(0.0, 0.0)--(-30.0, -30.0);
(0.0, 0.0)--(30.0, -30.0);
(-30.0, -30.0)--(-60.0,-60.0);
(-30.0, -30.0)--(0.0, -60.0);

(-10.0, 0.0)~(-50.0, -30.0)~(-65.0, -65.0)~(-30.0, -40.0)~
    (5.0,-65.0)~(0.0,-50.0)~(-15.0,-30.0)~(-10.0, -20.0)~
    (0.0,-10.0)~(25.0,-32.0)~(35.0,-35.0)~
    (32.0, -27.0)~>(10.0, 0.0);
```



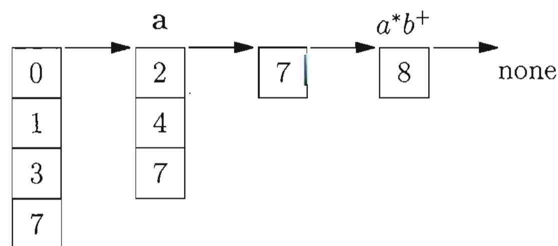
```
circle1 = Circle with [radius = 10.0];
```

```
circle2 = Circle with [radius = 12.0];
ellipse = Ellipse with [xradius = 50.0, yradius = 20.0];
circle = put circle1 at (-30.0, 0.0) in form;
put circle1 at (30.0, 0.0) in form;
put circle2 at (30.0, 0.0) in form;
put ellipse in form;

d1 = put form at (100.0, 100.0) in main;
d2 = put form at (100.0, 50.0) in main;
d3 = put form at (100.0, -50.0) in main;
put Text with [text = "$N(p_1)$"] at (100.0, 100.0) in main;
put Text with [text = "$N(p_2)$"] at (100.0, 50.0) in main;
put Text with [text = "\dots"] at (100.0,0.0) in main;
put Text with [text = "$N(p_n)$"] at (100.0, -50.0) in main;

put Text with [text = "$s_0$"] at (0.0, 20.0) in main;
d4 = put circle1 at (0.0, 20.0) in main;

(coord of anglePoint(70.0) in main.d4)->
    (coord of anglePoint(180.0) in main.d1.circle);
(coord of anglePoint(0.0) in main.d4)->
    (coord of anglePoint(180.0) in main.d2.circle);
(coord of anglePoint(-70.0) in main.d4)->
    (coord of anglePoint(180.0) in main.d3.circle);
```



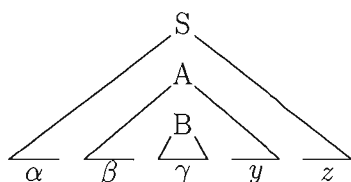

```
fun dropMany(f, n, s, v, c){
  for i from 1.0 to n do
    put f at (s + (i - 1.0) * v) in c;
```

```
  end
}

square = Square with [length=20.0];
changeVec = (0.0, -20.0);
dropMany(square, 4.0, (0.0, 0.0), changeVec, main);
dropMany(square, 3.0, (50.0, 0.0), changeVec, main);
dropMany(square, 1.0, (100.0, 0.0), changeVec, main);
dropMany(square, 1.0, (150.0, 0.0), changeVec, main);
put Text with [text="0"] at (0.0, 0.0) in main;
put Text with [text="1"] at (0.0, -20.0) in main;
put Text with [text="3"] at (0.0, -40.0) in main;
put Text with [text="7"] at (0.0, -60.0) in main;
put Text with [text="\bf a"] at (50.0, 20.0) in main;
put Text with [text="2"] at (50.0, 0.0) in main;
put Text with [text="4"] at (50.0, -20.0) in main;
put Text with [text="7"] at (50.0, -40.0) in main;
put Text with [text="7"] at (100.0, 0.0) in main;
put Text with [text="\bf $a^*b^+$"] at (150.0, 20.0) in main;
put Text with [text="8"] at (150.0, 0.0) in main;
put Text with [text="none"] at (200.0, 0.0) in main;

(12.0, 10.0)->(38.0, 10.0);
(62.0, 10.0)->(88.0, 10.0);
(112.0, 10.0)->(138.0, 10.0);
(162.0, 10.0)->(188.0, 10.0);
```



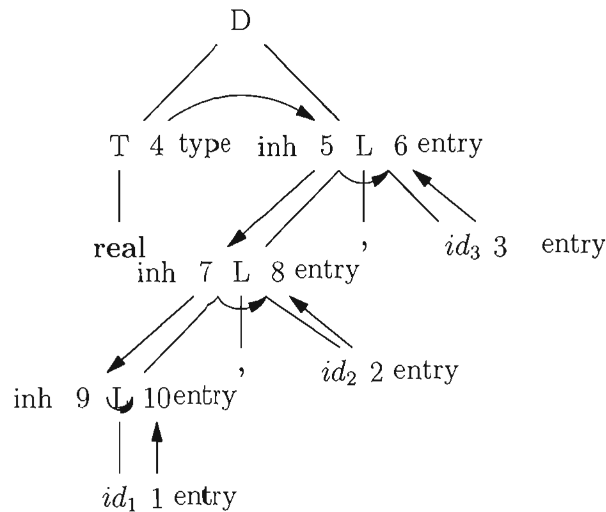

```
put Text with [text="S"] in main;
put Text with [text="A"] at (0.0, -20.0) in main;
put Text with [text="B"] at (0.0, -40.0) in main;
put Text with [text="$\alpha$"] at (-60.0, -60.0) in main;
put Text with [text="$\beta$"] at (-30.0, -60.0) in main;
put Text with [text="$\gamma$"] at (0.0, -60.0) in main;
put Text with [text="$y$"] at (30.0, -60.0) in main;
put Text with [text="$z$"] at (60.0, -60.0) in main;
```

```
(-5.0, -5.0)--(-70.0, -54.0)--(-50.0, -54.0);
(5.0, -5.0)--(70.0, -54.0)--(50.0, -54.0);
(-5.0, -25.0)--(-40.0, -54.0)--(-20.0, -54.0);
(5.0, -25.0)--(40.0, -54.0)--(20.0, -54.0);
(-5.0, -45.0)--(-10.0, -54.0)--(10.0, -54.0)--(5.0, -45.0);
```




```
put Text with [text="D"] in main;

put Text with [text="T"] at (-50.0, -50.0) in main;
put Text with [text="\bf real"] at (-50.0, -90.0) in main;
put Text with [text="4"] at (-35.0, -50.0) in main;
put Text with [text="type"] at (-15.0, -50.0) in main;

put Text with [text="L"] at (50.0, -50.0) in main;
put Text with [text="inh"] at (15.0, -50.0) in main;
put Text with [text="5"] at (35.0, -50.0) in main;
put Text with [text="6"] at (65.0, -50.0) in main;
put Text with [text="entry"] at (85.0, -50.0) in main;

put Text with [text="\bf $id_3$"] at (90.0, -90.0) in main;
put Text with [text="\bf ,"] at (50.0, -90.0) in main;
put Text with [text="3"] at (105.0, -90.0) in main;
put Text with [text="entry"] at (135.0, -90.0) in main;

put Text with [text="L"] at (0.0, -100.0) in main;
put Text with [text="inh"] at (-35.0, -100.0) in main;
put Text with [text="7"] at (-15.0, -100.0) in main;
put Text with [text="8"] at (15.0, -100.0) in main;
put Text with [text="entry"] at (35.0, -100.0) in main;

put Text with [text="\bf ,"] at (0.0, -140.0) in main;
put Text with [text="\bf $id_2$"] at (40.0, -140.0) in main;
put Text with [text="2"] at (55.0, -140.0) in main;
put Text with [text="entry"] at (75.0, -140.0) in main;

put Text with [text="L"] at (-50.0, -150.0) in main;
```

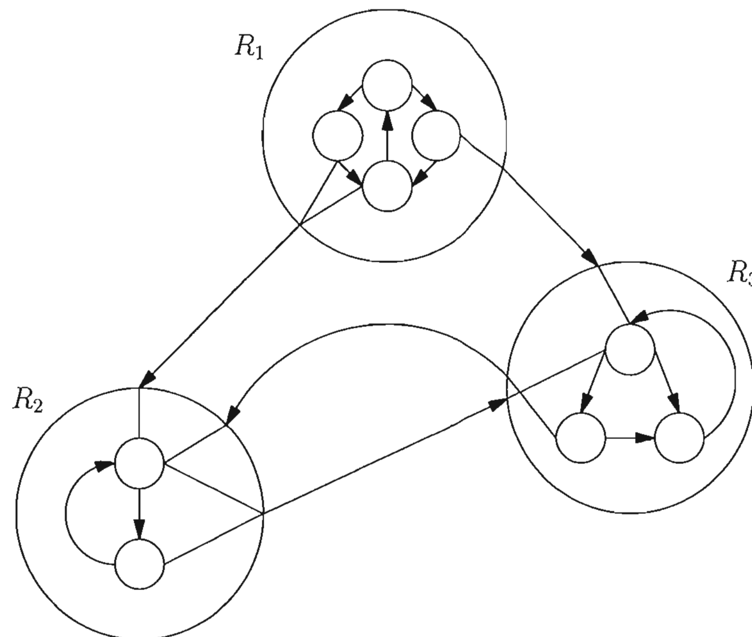
```
put Text with [text="inh"] at (-85.0, -150.0) in main;
put Text with [text="9"] at (-65.0, -150.0) in main;
put Text with [text="10"] at (-35.0, -150.0) in main;
put Text with [text="entry"] at (-15.0, -150.0) in main;

put Text with [text="\bf $id_1$"] at (-50.0, -190.0) in main;
put Text with [text="1"] at (-35.0, -190.0) in main;
put Text with [text="entry"] at (-15.0, -190.0) in main;

(-10.0, -10.0)--(-40.0, -40.0);
(-50.0, -60.0)--(-50.0, -80.0);
(10.0, -10.0)--(40.0,-40.0);
(50.0, -60.0)--(50.0, -80.0);
(60.0, -60.0)--(80.0, -80.0);
(40.0, -60.0)--(10.0, -90.0);
(10.0, -110.0)--(40.0, -130.0);
(0.0, -110.0)--(0.0, -130.0);
(-10.0, -110.0)--(-40.0, -140.0);
(-50.0, -160.0)--(-50.0, -180.0);

(-30.0, -40.0)~(0.0, -30.0)~>(30.0, -40.0);
(40.0, -60.0)~(50.0, -65.0)~>(60.0, -60.0);
(95.0, -80.0)->(70.0, -60.0);
(30.0, -60.0)->(-5.0, -90.0);
(-10.0, -110.0)~(0.0,-115.0)~>(10.0, -110.0);
(45.0, -130.0)->(20.0, -110.0);
(-20.0, -110.0)->(-55.0, -140.0);
(-55.0, -150.0)~(-50.0, -155.0)~>(-45.0, -150.0);
(-35.0, -180.0)->(-35.0, -160.0);
```




```
circle = Circle with [radius=10.0];
bigcircle = Circle with [radius=50.0];

c1 = put circle at (0.0, 20.0) in graph1;
c2 = put circle at (20.0, 0.0) in graph1;
c3 = put circle at (-20.0, 0.0) in graph1;
c4 = put circle at (0.0, -20.0) in graph1;
bc1 = put bigcircle in graph1;

c5 = put circle at (0.0, 20.0) in graph2;
c6 = put circle at (0.0, -20.0) in graph2;
bc2 = put bigcircle in graph2;

c7 = put circle at (0.0, 15.0) in graph3;
c8 = put circle at (20.0, -20.0) in graph3;
c9 = put circle at (-20.0, -20.0) in graph3;
bc3 = put bigcircle in graph3;

g1 = put graph1 in main;
g2 = put graph2 at (-100.0, -150.0) in main;
g3 = put graph3 at (100.0, -100.0) in main;

/* lines in first drawing */
(coord of anglePoint(90.0) in main.g1.c4)~>
    (coord of anglePoint(270.0) in main.g1.c1);
(coord of anglePoint(180.0) in main.g1.c1)~>
    (coord of anglePoint(90.0) in main.g1.c3);
(coord of anglePoint(0.0) in main.g1.c1)~>
```

```
    (coord of anglePoint(90.0) in main.g1.c2);
(coord of anglePoint(270.0) in main.g1.c3)~>
    (coord of anglePoint(180.0) in main.g1.c4);
(coord of anglePoint(270.0) in main.g1.c2)~>
    (coord of anglePoint(0.0) in main.g1.c4);

(coord of anglePoint(270.0) in main.g1.c3)~
    (coord of anglePoint(225.0) in main.g1.bc1);
(coord of anglePoint(180.0) in main.g1.c4)~
    (coord of anglePoint(225.0) in main.g1.bc1);
(coord of anglePoint(0.0) in main.g1.c2)~
    (coord of anglePoint(345.0) in main.g1.bc1);

/* lines in second drawing */
(coord of anglePoint(270.0) in main.g2.c5)~>
    (coord of anglePoint(90.0) in main.g2.c6);
crd1 = coord of anglePoint(180.0) in main.g2.c6;
crd2 = coord of anglePoint(180.0) in main.g2.c5;
crd1~((crd1 + crd2)/2.0 + (-20.0, 0.0))~>crd2;

(coord of anglePoint(90.0) in main.g2.c5)~
    (coord of anglePoint(90.0) in main.g2.bc2);
(coord of anglePoint(0.0) in main.g2.c5)~
    (coord of anglePoint(45.0) in main.g2.bc2);
(coord of anglePoint(0.0) in main.g2.c5)~
    (coord of anglePoint(0.0) in main.g2.bc2);
(coord of anglePoint(0.0) in main.g2.c6)~
    (coord of anglePoint(0.0) in main.g2.bc2);

/* lines in third drawing */
(coord of anglePoint(180.0) in main.g3.c7)~>
    (coord of anglePoint(90.0) in main.g3.c9);
(coord of anglePoint(0.0) in main.g3.c7)~>
    (coord of anglePoint(90.0) in main.g3.c8);
(coord of anglePoint(0.0) in main.g3.c9)~>
    (coord of anglePoint(180.0) in main.g3.c8);
crd3 = coord of anglePoint(0.0) in main.g3.c8;
crd4 = coord of anglePoint(90.0) in main.g3.c7;
crd3~(crd3 + crd4) / 2.0 + (20.0, 20.0)~>crd4;

crd4--(coord of anglePoint(105.0) in main.g3.bc3);
(coord of anglePoint(180.0) in main.g3.c7)--
    (coord of anglePoint(185.0) in main.g3.bc3);
(coord of anglePoint(180.0) in main.g3.c9)--
```

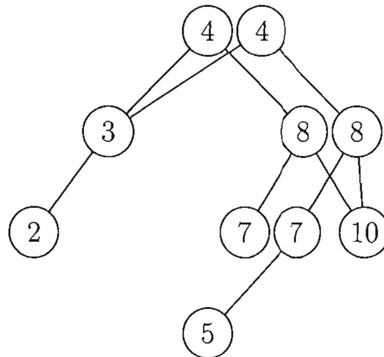
```
    (coord of anglePoint(175.0) in main.g3.bc3);

/* lines between drawings */
(coord of anglePoint(225.0) in main.g1.bc1)~>
    (coord of anglePoint(90.0) in main.g2.bc2);
(coord of anglePoint(345.0) in main.g1.bc1)~>
    (coord of anglePoint(105.0) in main.g3.bc3);
(coord of anglePoint(0.0) in main.g2.bc2)~>
    (coord of anglePoint(185.0) in main.g3.bc3);
crd5 = coord of anglePoint(175.0) in main.g3.bc3;
crd6 = coord of anglePoint(45.0) in main.g2.bc2;
crd5~(crd5 + crd6) / 2.0 + (00.0, 30.0)~>crd6;

/* Texts */
crdText1 = coord of anglePoint(135.0) in main.g1.bc1 + (-20.0, 0.0);
crdText2 = coord of anglePoint(135.0) in main.g2.bc2 + (-10.0, 10.0);
crdText3 = coord of anglePoint(45.0) in main.g3.bc3 + (10.0, 10.0);
put Text with [text="$R_1$"] at crdText1 in main;
put Text with [text="$R_2$"] at crdText2 in main;
put Text with [text="$R_3$"] at crdText3 in main;
```




```
primitive getClip "sddl.hierarchy.Drawing" "clip";
primitive setVisible "sddl.hierarchy.Shape" "setVisible";
```

```
/* clip takes 3 drawings and a canvas
   first is a line
   second and third are two shapes which intersect line
   will draw a line between the intersection points the line
   creates with the two shapes */
fun clip(d1, d2, d3, canvas)
{
d1.setVisible(false);
draw1 = put d1 in canvas;
c1 = draw1.getClip(d2);
c2 = draw1.getClip(d3);
put c1--c2 in canvas;

}

circle = Circle with [radius=10.0];
d1 = put circle in main;
put Text with [text="4"] in main;
d2 = put circle at (-40.0, -40.0) in main;
put Text with [text="3"] at (-40.0, -40.0) in main;
d3 = put circle at (-70.0, -80.0) in main;
put Text with [text="2"] at (-70.0, -80.0) in main;
d4 = put circle at (40.0, -40.0) in main;
put Text with [text="8"] at (40.0, -40.0) in main;
d5 = put circle at (15.0, -80.0) in main;
put Text with [text="7"] at (15.0, -80.0) in main;
d6 = put circle at (65.0, -80.0) in main;
put Text with [text="10"] at (65.0, -80.0) in main;
d7 = put circle at (22.0, 0.0) in main;
put Text with [text="4"] at (22.0, 0.0) in main;
d8 = put circle at (62.0, -40.0) in main;
put Text with [text="8"] at (62.0, -40.0) in main;
d9 = put circle at (37.0, -80.0) in main;
put Text with [text="7"] at (37.0, -80.0) in main;
d10 = put circle at (0.0, -120.0) in main;
put Text with [text="5"] at (0.0, -120.0) in main;

l1 = (0.0, 0.0)--(-40.0, -40.0);
clip(l1, d1, d2, main);
l2 = (-40.0, -40.0)--(-70.0, -80.0);
clip(l2, d2, d3, main);
l3 = (0.0, 0.0)--(40.0, -40.0);
clip(l3, d1, d4, main);
l4 = (40.0, -40.0)--(15.0, -80.0);
```

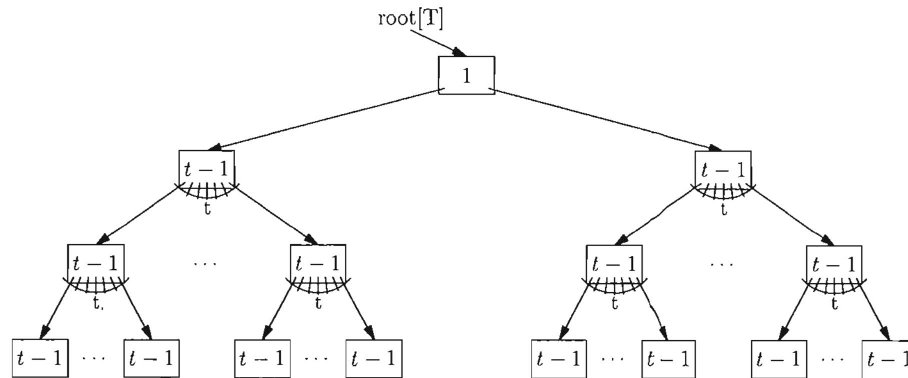
```
clip(l4, d4, d5, main);
l5 = (40.0, -40.0)--(65.0, -80.0);
clip(l5, d4, d6, main);
l6 = (22.0, 0.0)--(-40.0, -40.0);
clip(l6, d7, d2, main);
l7 = (22.0, 0.0)--(62.0, -40.0);
clip(l7, d7, d8, main);
l8 = (62.0, -40.0)--(65.0, -80.0);
clip(l8, d8, d6, main);
l9 = (62.0, -40.0)--(37.0, -80.0);
clip(l9, d8, d9, main);
l10 = (37.0, -80.0)--(0.0, -120.0);
clip(l10, d9, d10, main);
```




```
fun niceArrow(c, dMain, dLeft, dRight){
  firstPoint = coord of southwest in dMain.rectangle + (3.0, 3.0);
  secondPoint = coord of southeast in dMain.rectangle + (-3.0, 3.0);

  firstEnd = coord of north in dLeft.rectangle;
  secondEnd = coord of north in dRight.rectangle;

  put firstPoint->firstEnd in c;
  put secondPoint->secondEnd in c;

  point1 = firstPoint * 5.0 / 6.0 + secondPoint * 1.0 / 6.0;
  point2 = firstPoint * 4.0 / 6.0 + secondPoint * 2.0 / 6.0;
  point3 = firstPoint * 3.0 / 6.0 + secondPoint * 3.0 / 6.0;
  point4 = firstPoint * 2.0 / 6.0 + secondPoint * 4.0 / 6.0;
  point5 = firstPoint * 1.0 / 6.0 + secondPoint * 5.0 / 6.0;

  put (coord of southwest in dMain.rectangle + (-2.5, 2.5))~
      (coord of south in dMain.rectangle + (0.0, -5.0))~
```

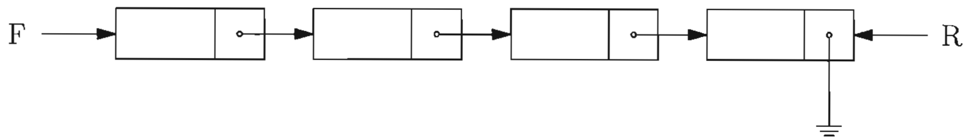
```
      (coord of southeast in dMain.rectangle + (2.5, 2.5)) in c;

  put point1--(point1 + (-1.0 * 3.0, -1.0 * 7.0)) in c;
  put point2--(point2 + (-1.0 * 1.0, -1.0 * 8.5)) in c;
  put point3--(point3 + (0.0, -1.0 * 9.0)) in c;
  put point4--(point4 + (1.0, -1.0 * 8.5)) in c;
  put point5--(point5 + (3.0, -1.0 * 7.0)) in c;

  put Text with [text="\small{t}"] at point3 + (0.0, -13.0) in c;

}

rect = Rectangle with [length=30.0, width=20.0];
r = put rect in main;
put Text with [text="1"] in main;
rectangle = put rect in form;
put Text with [text="$t-1$"] in form;

dl = put form at (-140.0, -50.0) in main;
dll = put form at (-200.0, -100.0) in main;
dlr = put form at (-80.0, -100.0) in main;
dlll = put form at (-230.0, -150.0) in main;
dllr = put form at (-170.0, -150.0) in main;
dlrl = put form at (-110.0, -150.0) in main;
dlrr = put form at (-50.0, -150.0) in main;

dr = put form at (140.0, -50.0) in main;
drr = put form at (200.0, -100.0) in main;
drl = put form at (80.0, -100.0) in main;
drrr = put form at (230.0, -150.0) in main;
drrl = put form at (170.0, -150.0) in main;
drlr = put form at (110.0, -150.0) in main;
drll = put form at (50.0, -150.0) in main;

put Text with [text="root[T]"] at (-30.0, 30.0) in main;
(-30.0, 24.0)->(coord of north in main.r);

dots = Text with [text="\dots"];
put dots at (-140.0, -100.0) in main;
put dots at (-200.0, -150.0) in main;
put dots at (-80.0, -150.0) in main;
put dots at (140.0, -100.0) in main;
put dots at (200.0, -150.0) in main;
put dots at (80.0, -150.0) in main;
```

```
(coord of southwest in main.r + (3.0, 3.0))->
    (coord of north in main.dl.rectangle);
(coord of southeast in main.r + (-3.0, 3.0))->
    (coord of north in main.dr.rectangle);

niceArrow(main, dl, dll, dlr);
niceArrow(main, dll, dlll, dllr);
niceArrow(main, dlr, dlrl, dlrr);
niceArrow(main, dr, drl, drr);
niceArrow(main, drl, drll, drlr);
niceArrow(main, drr, drrl, drrr);
```




```
rect = put Rectangle with [length=60.0, width=20.0] in form;
put Rectangle with [length=20.0, width=20.0] at (20.0, 0.0) in form;
bullet = put Circle with [radius=1.0] at (20.0, 0.0) in form;

put Text with [text="F"] in main;

f1 = put form at (70.0, 0.0) in main;
f2 = put form at (150.0, 0.0) in main;
f3 = put form at (230.0, 0.0) in main;
f4 = put form at (310.0, 0.0) in main;

(coord of anglePoint(0.0) in main.f1.bullet)->
    (coord of west in main.f2.rect);
(coord of anglePoint(0.0) in main.f2.bullet)->
    (coord of west in main.f3.rect);
(coord of anglePoint(0.0) in main.f3.bullet)->
    (coord of west in main.f4.rect);
```

```
ptW = coord of west in main.f1.rect;
ptE = coord of east in main.f4.rect;

ptW + (-30.0, 0.0)->ptW;
ptE + (30.0, 0.0)->ptE;

put Text with [text="R"] at (380.0, 0.0) in main;

ptS = coord of anglePoint(270.0) in main.f4.bullet;
ptS2 = ptS + (0.0, -35.0);

ptS--ptS2;
ptS2 + (-5.0, 0.0)--ptS2 + (5.0, 0.0);
ptS2 + (-3.5, -2.0)--ptS2 + (3.5, -2.0);
ptS2 + (-2.0, -4.0)--ptS2 + (2.0, -4.0);
```



# BIBLIOGRAPHIE


Adobe Systems Incorporated. 1994. PostScript Language Reference Manual. http ://www.adobe.com/products/postscript/pdfs/PLRM.pdf.

———. 2006. *PDF Reference : Version 1.7.* Adobe Systems Incorporated, 6th édition.

———. 2010a. Adobe Postscript 3. http ://www.adobe.com/products/postscript/.

———. 2010b. PDF Developer Corner : PDF Reference. http ://www.adobe.com/devnet/pdf/pdf_reference.html.

Aho, A. V., R. Sethi, et J. D. Ullman. 1986. *Compilers : Principles, Techniques and Tools.* Addison-Wesley.

American Mathematical Society. 2008. Why Do We Recommend LaTeX ? http ://www.ams.org/publications/authors/tex/latexbenefits.

Apple Computer, Inc. 1996. *The TrueType Reference Manual.* Apple Computer, Inc., Cupertino, California.

Bourgeois, M. 2010. Introduction to Drawing Diagrams with SDDL. http ://river-valley.tv/introduction-to-drawing-structured-diagrams-in-sddl/.

Bourgeois, M. et R. Villemaire. 2010a. « Drawing Structured Diagrams with SDDL », *TUGBoat*, vol. 31, no. 2, p. 206–210.

———. 2010b. SDDL. http ://www.info2.uqam.ca/~villemaire_r/Recherche/SDDL/.

Bowman, J. C. 2007. « The 3D Asymptote Generalization of Metapost Bézier Interpolation », *Proceedings in Applied Mathematics and Mechanics.*

Chomsky, N. 1956. « Three Models for the Description of Language », *Information Theory, IRE Transactions on*, vol. 2, no. 3, p. 113–124.

Cormen, T. A., C. E. Leiserson, R. L. Rivest, et C. Stein. 2001. *Introduction to Algorithms.* The MIT Press.

Davis, M., R. Sigal, et E. Weyuker. 1994. *Computability, Complexity, and Languages : Fundamentals of Theoretical Computer Science.* Morgan Kaufmann.

Dhu, D.-Z. et K.-I. Ko. 2000. *Theory of Computational Complexity.* Wiley.

Gagnon, É. 1998. « SableCC, an Object-Oriented Compiler Framework ». Mémoire de maîtrise, McGill University.

———. 2009. ObjectMacro - SableCC. http ://sablecc.org/wiki/ObjectMacro.

Group, O. M. 1999. Unified Modeling Language (version 1.3). Rapport, Object Management Group.

Hammerlindl, A., J. Bowman, et T. Prince. 2010a. Asymptote. http ://asymptote.sourceforge.net/.

———. 2010b. Asymptote : The Vector Graphics Language. http ://asymptote.sourceforge.net/asymptote.pdf.

Hobby, J. 1992. « A User's Manual for MetaPost », *Computing Science Technical Report*, vol. 162.

International Organization for Standardization. 2008. Document Management — Electronic Document File Format for Long-Term Preservation — Part 1 : Use of PDF 1.4 (PDF/A-1). ISO 19005-1 :2005.

Jr., F. S. H. et S. M. Kelley. 2007. *Computer Graphics using OpenGL*. Prentice-Hall, 3e édition.

Knuth, D. 1968. *The Art of Computer Programming*. Addison-Wesley.

———. 1984. *The TeXbook*. Reading, Mass, Addison-Wesley.

———. 1986. *The Metafontbook*. Reading, Mass, Addison-Wesley.

———. 1999. *Digital Typography*. Stanford, California : Center for the Study of Language and Information.

Lamport, L. et D. Bibby. 1986. *LATEX : A Document Preparation System*. T. 260. Citeseer.

Lancaster, D. 2005. Approximating a Circle or an Ellipse using Four Bezier Cubic Splines. Rapport, Synergetics.

Maplesoft. 2010. *Maple 14 User Manual*.

McNaughton, R. et H. Yamada. 1960. « Regular Expressions and State Graphs for Automata », *IRE Transactions on Electronic Computers*, no. 1, p. 39–47.

Object Management Group. 2009. UML. http ://www.uml.org/.

Oracle. 2010. Java. http ://www.java.com/.

Pierce, B. C. 2002. *Types and Programming Languages*. MIT Press.

Riskus, A. 2006. « Approximation of a Cubic Bezier Curve by Circular Arcs and Vice Versa », *Information Technology and Control*, vol. 35, no. 4, p. 371–378.

Rose, K. H. 2008. XY-pic Home Page. http ://tug.org/applications/Xy-pic/.

Rose, K. H. et al. 2010. « XY-pic User's Guide », *DIKU, Université de Copenhague*, vol. 1.

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, et W. Lorensen. 1991. *Object-Oriented Modeling and Design*. Prentice-Hall.

Saunders, J. H. 1989. « A Survey of Object-Oriented Programming Languages », *JOOP : Journal of Object-Oriented Programming*, vol. 1.

Sobel, J. et D. Friedman. 1996. *An Introduction to Reflection-Oriented Programming*. San Francisco : Reflection 96.

TeX Users Group. 2009. MetaPost - TeX User Groups. http ://tug.org/metapost.html.

TIOBE Software. 2008. Tiobe Index. http ://www.tiobe.com/index.php/content-/paperinfo/tpci/index.html.

Ullman, J., J. Hopcroft, et R. Motwani. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Van Zandt, T. 1993a. PSTricks-PostScript Macros for Generic TEX. http ://www.tex.uniyar.ac.ru/doc/pst_ug.pdf.

———. 1993b. « PSTricks : PostScript Macros for Generic TEX User's Guide », *Version 0.93 a, http ://www. tug. org/applications/PSTricks.*