Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Inférence sur RDF et génération de XML en Mercury

Degrave, François

Award date:
2006

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

Inférence sur RDF et génération de XML en Mercury

François Degrave

Mémoire présenté en vue de l'obtention du grade de
Maître en Informatique
Année académique 2005 - 2006

Résumé

Les applications dites "*light client*", privilégiées par le marché depuis la fin des années 90, sont entièrement exécutées par un serveur ; le rôle du client se limite à afficher, via un navigateur Web, l'interface permettant d'interagir avec ce dernier. Si cette architecture présente d'évidents avantages pour la maintenance, ceux-ci sont malheureusement contrebalancés par une ergonomie souvent médiocre et une qualité d'interaction très perfectible (à cause des latences induites par les "*Round-Trip delay Times*" et le rafraîchissement des pages dans le navigateur).

Depuis environ la fin de l'année 2004, Mission Critical - c'est ainsi que se nomme l'entreprise dans laquelle s'est déroulé le projet - s'est donc tournée vers le concept de *Rich Web Client*, qui a comme particularité de partager quelque peu les tâches entre le serveur et le client. Ce dernier est ainsi amené à opérer certains traitements locaux, ne demandant pas d'assistance de la part du serveur. Chez Mission Critical, ce concept s'est intégré grâce, notamment, à l'utilisation de *templates* développés en XUL pour générer l'interface ; le XUL (*XML User Interface Language*) est un langage de création d'interfaces propre à Mozilla. Le mécanisme de *templates* mentionné ici permet en réalité de générer et de modifier des interfaces en effectuant des requêtes portant sur un fichier RDF (*Resource Description Framework*, modèle de données pour lequel il existe une syntaxe XML). Le XUL a par contre comme inconvénients de n'être utilisable qu'avec les navigateurs de Mozilla et de comporter des mécanismes dont la sémantique est parfois à la limite du compréhensible, ce qui rend assez difficile son utilisation quotidienne.

Le but du projet à la base de cet ouvrage était donc de recréer un langage de *templates* semblable au XUL (en essayant d'améliorer ses défauts sémantiques), et d'implémenter en Mercury - langage logique/fonctionnel utilisé chez Mission Critical - l'application capable de l'interpréter, et de générer ainsi du XML automatiquement à partir d'un fichier RDF, de façon indépendante du navigateur utilisé. Cet ouvrage contient donc un bref aperçu du langage Mercury, du modèle de données RDF et de sa syntaxe XML non-abrégée, et décrit de façon assez détaillée le nouveau langage de *templates* que nous avons défini.

Abstract

In client-server architecture networks, so-called "light client" applications are very popular since the end of the 90's. In this kind of applications, the client's computer system depends primarily on the central server for processing activities ; the client's only job is to show the user interface on the screen, thanks to a Web browser. This architecture has many advantages, especially for maintenance, but has also many drawbacks : a high cost of development, a poor ergonomy and a very perfectible interaction's quality (because of the Round-Trip delay Times and pages refreshes).

Since about the end of the year 2004, Mission Critical - as is called the company where the projet was held - chose to use the "Rich Web Client" concept, which has the particularity to transfer some processing activities to the client. This client can thus process some local jobs without needing any help from the server. In Mission Critical, this concept was integrated thanks to XUL to generate the interface. The XUL language (*XML User Interface Language*) has been developed by the Mozilla company to create interfaces. Its interesting mechanism is called *templates* ; it allows us to generate and modify interfaces by making requests on RDF data files (*Resource Description Framework*, a data model that can be written in a special XML syntax). On the other hand, XUL has several drawbacks : it has to be used with Mozilla's browsers, and its semantics are sometimes really unclear, which makes XUL quite difficult to use in large applications.

The purpose of the project was to recreate a template's language similar to XUL (with improved semantics) and to implement in Mercury - a logic/functional programming language used by Mission Critical - the application able to interpret it, and thus to generate XML automatically from a RDF data file independently from the Web browser used. This report contains an overview of the Mercury programming language, of the RDF data model and its non-abbreviated XML syntax, and describes quite precisely the new template's language we defined.

Remerciements

Au terme de cet ouvrage et du stage qui a précédé sa rédaction, il me vient à l'esprit beaucoup de monde à qui exprimer ma reconnaissance ; je les citerai ici dans un ordre que je souhaite le moins maladroit possible :
Le promoteur de ce mémoire, Wim Vanhoof, pour son soutien inconditionnel et les nombreuses et précieuses corrections qu'il a apportées à cet ouvrage ;
Michel Vanden Bossche, pour m'avoir si chaleureusement accueilli chez Mission Critical et m'avoir soutenu tout au long du projet ;
Vinciane Capelle, dont les rires et la joie de vivre ont égayé mon stage à Erps-Kwerps ;
Maxime Van Assche, qui a suivi de près l'avancée du projet, émis de nombreuses (et fructueuses) idées et m'a soutenu lors de la réalisation de l'*engine* ;
Paul Massey, dont les conseils au quotidien ont été indispensables à la progression du travail, et dont la conversation a été un véritable plaisir durant nos trajets à vélo entre Leuven et Erps-Kwerps ;
Thomas Fazekas, qui m'a aidé à chaque fois que j'en ai eu besoin, et qui m'a accueilli dans sa voiture lorsqu'il faisait trop froid pour rentrer à vélo avec Paul ;
Toutes les personnes de chez Mission Critical, qui font régner dans l'entreprise une ambiance de travail détendue et conviviale, qui donne l'envie et le courage d'offrir tous les jours le meilleur de soi ;
Emmanuelle Boilan, qui m'accompagne chaque jour dans ma vie et dans mes études, et dont la passion et le dévouement envers son travail me servent d'exemple pour réaliser le mien ;
Mes parents, qui ont soutenu chacun de mes choix, même les plus difficiles, et qui ont accessoirement relu et corrigé l'orthographe de cet ouvrage malgré leur notable incompréhension du sujet traité ;
Tous mes collègues de l'Institut d'Informatique, pour leur amitié et leur aide précieuse tout au long de mon cursus ;
Le dictionnaire de synonymes en ligne de l'Université de Caen, qui a suppléé à mon manque de vocabulaire ;
Le système LaTeX, car sans doute n'aurais-je pas pu terminer cet ouvrage dans les temps en utilisant une autre méthode de rédaction.

Table des matières

1 Introduction **7**
1.1 Systèmes *fat client* et *light client* 7
1.2 Les ontologies 9
1.3 L'approche de Mission Critical 10
1.4 Le projet 11

2 Le langage Mercury **13**
2.1 Les paradigmes de programmation 13
2.1.1 Langages impératifs et effets de bord 13
2.1.2 Langages fonctionnels 14
2.1.3 Langages logiques 15
2.2 Le langage Mercury 15
2.2.1 Vue d'ensemble 15
2.2.2 Les avantages du Mercury 16
2.3 La programmation en Mercury 17
2.3.1 Les modules 18
2.3.2 L'interface 18
2.3.3 L'implémentation 19
2.3.4 Les prédicats 19
2.3.5 Les fonctions 20
2.3.6 Les importations de modules 21
2.3.7 Les types de base et la librairie standard 21
2.3.8 Les types définis par l'utilisateur 23
2.3.9 Les modes 26
2.3.10 Le déterminisme 27

3 Le RDF **28**
3.1 Introduction 28
3.2 Des faits au RDF 29
3.2.1 Les faits 29
3.2.2 Les faits et les structures de données 30

3.2.3 Prédicats et triples 30
3.2.4 Faits et graphes 32
3.3 La syntaxe XML du RDF 35
3.3.1 Les balises RDF 36
3.3.2 Les identifiants 37
3.3.3 Les balises à la loupe 41
3.4 Quelques applications du RDF 45

4 Le *template engine* 46
4.1 Le concept de *template* 46
4.1.1 Requêtes RDF et unification 46
4.1.2 Les *gabarits* 50
4.2 Le langage de *templates* 52
4.2.1 L'élément *template* 52
4.3 La structure d'une règle 52
4.3.1 L'élément *conditions* 53
4.3.2 L'élément *bindings* 54
4.3.3 L'élément *action* 56
4.3.4 La référence 57
4.4 Plusieurs règles dans un *template* 58
4.4.1 L'attribut `"comparenode"` 60
4.5 Les *templates* imbriqués 64
4.6 Les *templates* récursifs 67
4.7 Défauts et améliorations 70

5 Conclusion 73
5.1 Bilan 73
5.2 Perspectives 75

Chapitre 1

Introduction

Cette introduction a pour but de faire comprendre les motivations originelles qui ont donné naissance au projet qui fut à la base de cet ouvrage. Ce projet a été mené à bien dans une entreprise appelée MISSION CRITICAL, focalisée sur le développement de solutions informatiques dites "critiques" (une application critique pour une entreprise est une application indispensable à son fonctionnement). Sur le marché du logiciel, la plupart des demandes provenant des entreprises portent sur des applications du type client-serveur. Une application client-serveur met en jeu, comme son nom l'indique, un serveur qui fournit des services à des ordinateurs clients ; il est possible de catégoriser les systèmes de ce genre selon le volume de tâches traitées localement par le client. Ainsi, nous distinguerons à première vue deux types de systèmes, l'un étant appelé "*fat client*" et l'autre, en opposition, nommé "*light client*".

1.1 Systèmes *fat client* et *light client*

Comme nous l'avons mentionné dans le paragraphe précédent, un client est un ordinateur doté d'un système permettant d'accéder à un service (distant) sur un autre ordinateur (le serveur), via un quelconque type de réseau.

Jusqu'il y a quelques années, les applications développées étaient toutes dites "*fat client*" (se dit aussi "*thick client*", ou en français "client lourd"). Un "client lourd" est un client qui exécute lui-même le travail exigé par toute opération de traitement de données ; il ne se repose sur le serveur que pour le stockage de données. Les applications faisant usage d'un "*fat client*" - l'existence de telles applications remonte au Mac (1984), à X Windows (1988), à Windows (1990), etc. - jouissent d'une excellente ergonomie.

Depuis la fin des années 90, dans la foulée de l'émergence d'Internet, le marché privilégie les architectures dites "*light client*" : le client s'en remet au serveur pour la plupart des traitements de données. Typiquement, le seul rôle du client est d'afficher graphiquement, grâce aux navigateurs HTML, les informations fournies par le serveur. Le principal avantage est d'éviter de devoir installer du code localement, et donc de grandement faciliter la maintenance du système ; avec des "clients lourds", l'installation - ou ne fût-ce que la modification de la configuration - d'une application est susceptible d'exiger un déplacement physique sur chaque poste de travail des utilisateurs. Avec des "clients légers", les tâches de maintenance sont centralisées sur le serveur et nécessitent de n'être faites qu'une seule fois.

Par contre, ces avantages "informatiques" sont malheureusement contrebalancés de nombreux inconvénients. L'expérience de MISSION CRITICAL a par exemple mis en évidence que le coût de développement de ces systèmes se révèle généralement très élevé. L'interface utilisateur présente habituellement une ergonomie médiocre, en raison des limitations qu'imposent les langages d'interfaçage Web tels que le HTML. De plus, la qualité d'interaction reste très perfectible ; en effet, toute interaction avec le système est soumis aux latences induites par les "*Round-Trip delay Times*" (RTT, temps d'aller-retour d'une requête vers le serveur) et le rafraîchissement des pages dans le navigateur.

Pour ces raisons, MISSION CRITICAL s'est donc tourné, depuis environ la fin de l'année 2004, vers le concept de *Rich Web Client*. Ce concept est un hybride entre le "*fat client*" et le "*light client*", dans lequel le volume de traitement de données est plus équilibré entre le serveur et le client. Dans cette approche, l'interface client permet d'effectuer un certain nombre de traitements locaux sans l'assistance d'un serveur.

Chez MISSION CRITICAL, ce concept s'est intégré dans un système conférant au client le moyen de générer et de modifier l'interface utilisateur de manière autonome. S'il est capable d'effectuer de tels traitements, c'est parce qu'il dispose de fichiers - envoyés auparavant par le serveur - lui fournissant des informations concernant la conceptualisation du domaine d'application ; cette conceptualisation se fait, dans notre cas, grâce à des *ontologies*. Découvrons d'abord un peu plus précisément en quoi consiste cette notion d'ontologie ; nous essaierons ensuite d'éclaircir ce que nous venons de décrire ici, à savoir la forme sous laquelle le concept de système *Rich Web Client* s'est intégré dans les applications développées par MISSION CRITICAL.

1.2 Les ontologies

Le terme "ontologie" a une très longue histoire en philosophie, branche dans laquelle il renvoie au sujet de l'existence. Dans le contexte du partage de connaissances, ce terme est utilisé pour désigner la spécification d'une conceptualisation d'un domaine de connaissance [GRU01].

Une ontologie est donc la conceptualisation d'un domaine, c'est-à-dire un choix quant à la manière de décrire ce domaine, une vue simplifiée située à un niveau d'abstraction choisi, du monde que nous aimerions représenter dans un quelconque but. Il s'agit aussi, d'autre part, de la spécification explicite de cette conceptualisation, c'est-à-dire sa description formelle.

Lorsqu'il s'agit de conceptualiser un domaine, il faut donc pouvoir identifier et modéliser les concepts pertinents (des objets auxquelles sont associées des propriétés), identifier les relations pertinentes entre ces concepts (*sous-classe-de*, *est-un*, *partie-de*, etc.) et définir les règles propres aux concepts et aux relations (*partie-de* est transitive,...)[COR].

Les informations sur Internet sont, pour la plupart, incompréhensibles par les machines. Une des conséquences directes de cela est le niveau très bas d'expressivité possible dans un moteur de recherche, qui s'arrête à la notion de "mot-clé". Afin de rendre interopérables des applications qui échangent des informations sur l'Internet, et de faciliter le traitement automatique de données des ressources Web, les ontologies peuvent être d'une grande aide, puisqu'elles permettent une capture sémantique de ces données [SWM04].

Cette vision d'un nouvel Internet, plus "intelligent" grâce aux ontologies, est celle du Web sémantique. Le Web sémantique repose sur la possibilité de définir, en XML, des systèmes de balisage personnalisés ; les recommandations liées au Web sémantique est une famille en pleine croissance dont la composition (ici classée en ordre croissant de niveau d'abstraction) est [MvH04] :

- Le langage XML, qui procure une syntaxe aux documents structurés mais n'impose aucune contrainte sémantique sur leur signification.
- RDF, un modèle de données pour des objets et leurs relations, qui fournit une sémantique simplifiée pour ces modèles de données, qui peuvent être représentés dans une syntaxe XML.
- Le RDF Schema, qui est un vocabulaire permettant de décrire les propriétés et les classes des ressources RDF, avec une sémantique pour des hiérarchies de généralisation de ces propriétés et de ces classes.

– Le langage OWL, qui renforce le vocabulaire de description des propriétés et des classes : entre autres, les relations entre les classes (par exemple la disjonction entre classes), la cardinalité ("un seul exactement",...), l'égalité, le typage enrichi des propriétés, les caractéristiques des propriétés (comme la symétrie) et les classes énumérées.

Typiquement, le langage OWL (qui ne sera pas décrit dans cet ouvrage, contrairement au RDF que nous examinerons au chapitre 3), sert à décrire le domaine de données grâce à des classes et leurs relations ; le RDF permettra alors de créer des instances de ces classes.

1.3 L'approche de Mission Critical

Afin de développer l'interface utilisateur, la solution de Mission Critical fait usage du langage XUL. Les initiales XUL (prononcer "Zoûle", en référence à un personnage du film *Ghostbusters*) signifient "*XML User Interface Language*" ; comme ce nom l'indique, il s'agit d'une approche XML de définition d'interfaces. Il fut créé, au départ, pour les besoins du navigateur Mozilla. Le mécanisme qui nous intéressera plus particulièrement, et qui constitue la particularité du langage XUL, est celui de *template* ; celui-ci permet de définir une requête afin d'extraire certaines données depuis une source de données RDF (modèle de données que nous avons cité à la section 1.2, et que nous étudierons plus en détail au chapitre 3), et d'utiliser ces données dans le but de générer du nouveau code de définition d'interface. Ceci permet donc de générer des interfaces, et de les modifier localement, sans imposer à chaque fois au client d'effectuer des appels vers le serveur. Le module de Mozilla capable d'interpréter les requêtes portant sur le RDF et de générer le code souhaité est nommé le *template engine*.

En vue de fixer quelque peu les idées, essayons d'illustrer nos propos par un exemple simple. Imaginons que l'application, créée pour l'entreprise Mercedes Benz, ait pour but de permettre à l'utilisateur de commander une voiture. La conceptualisation ontologique dégagera assurément le concept de modèle de voiture, dont une des propriétés sera la palette de couleurs disponibles pour ce modèle. Ces concepts pourront être concrétisés grâces à des classes OWL ; il sera alors possible de constituer des instances de ces classes dans le format RDF.

Le client, lorsqu'il accèdera à l'application exécutée sur le serveur afin de commander une voiture, ne recevra pas une page HTML contenant des tableaux, des champs, des boutons, etc. mais un fichier RDF contenant des descriptions des voitures disponibles, conformément à la conceptualisation ontologique qui a été faite, ainsi qu'un fichier XUL correspondant à cette source de données RDF. Lors

du choix du modèle de la voiture à commander, le *template engine* du navigateur MOZILLA sera capable d'interpréter les *templates* du fichier XUL et d'extraire ainsi, à partir des données RDF, les couleurs correspondant à ce modèle, et de modifier ou de générer une nouvelle interface afin d'afficher la palette de couleurs correpondant à ce modèle (sans aucune assistance de la part du serveur). Supposons par exemple que la Mercedes SLK ne soit disponible qu'en gris, en noir ou en bleu ; si l'utilisateur sélectionne ce modèle, l'interface se modifiera pour afficher, par exemple sous forme de liste, le choix des couleurs grise, noire et bleue. S'il décide ensuite de sélectionner un modèle différent dans la liste - disons une Mercedes classe C -, et que ce modèle est disponible en d'autres couleurs - choisissons gris, vert et bordeaux -, alors l'interface pourra s'adapter, sans assistance du serveur, et afficher dans la liste des couleurs disponibles le gris, le vert et le bordeaux.

Notons aussi que nous occultons ici complètement une partie assez importante du système, afin de nous focaliser sur les sujets qui nous intéressent plus particulièrement. Le XUL n'est en réalité capable que de générer des interfaces, mais en aucun cas de leur "donner vie" - d'associer aux éléments affichés des "actions" - afin de les rendre utilisables ; cet objectif est en réalité atteint grâce au JavaScript, que nous n'étudierons pas dans cet ouvrage.

Les résultats obtenus grâce à cette méthode sont excellents en terme d'ergonomie et de performance ; malheureusement, l'approche requiert l'utilisation d'un navigateur supportant le langage XUL, à savoir un navigateur MOZILLA. Ceci ne constitue pas un problème lorsque la population d'utilisateurs est ciblée, et lorsque l'utilisation est récurrente, mais c'est un obstacle pour une population occasionnelle qui ne souhaite pas changer de navigateur. En effet, même si *Internet Explorer* (IE) de MICROSOFT a vu sa part de marché baisser ces dernières années, elle reste tout de même la plus importante (environ 75% en mai 2006, comme le montre le site `http://www.w3schools.com`). Il existe bien un *plugin* XUL pour IE, mais les performances et la stabilité ne sont pas satisfaisantes.

De plus, le XUL présente plusieurs défauts majeurs. En effet, sa sémantique est, pour certains mécanismes, intuitivement incompréhensible, et parfois totalement absurde. Il n'offre, de plus, strictement aucun outil de détection d'erreurs ; le debugging se fait par conséquent complètement à l'aveugle.

1.4 Le projet

L'objectif du projet était donc de conserver la démarche de génération à partir d'une définition ontologique mais de cibler, non plus XUL couplé avec le JavaScript, mais du HTML couplé avec le JavaScript. Cette approche permet un gain de

portabilité évident, puisqu'elle autorise l'utilisation de n'importe quel navigateur du marché.

A cette fin, l'idée a été de construire un nouveau "*template engine*" en Mercury - il s'agit d'un langage logique/fonctionnel, utilisé chez MISSION CRITICAL pour l'implémentation des applications -, avec des caractéristiques différentes de celles de l'*engine* de MOZILLA. Tout d'abord, pour éviter l'installation d'un module chez le client, celui-ci fonctionnerait du côté serveur ; cette perspective a bien sûr un prix, qui est celui d'une moins grande performance (plus de *Round-Trips Delays*). Cependant, il serait indépendant du navigateur utilisé, indépendant du langage XML choisi pour la création d'interface (il permettrait, par exemple, la génération de HTML aussi bien que de XUL), posséderait une sémantique plus claire et plus intuitive que le XUL, et permettrait la création d'outils de détection d'erreurs.

Les points importants du projet que nous avons mené cette année étaient donc de définir un langage de *templates* pour des sources de données RDF et d'implémenter l'*engine* correspondant, permettant ainsi l'interprétation des requêtes et la génération de code.

En vue de définir un nouveau langage, nous avons donc examiné plusieurs langages existants de requêtes fonctionnant sur RDF. Le RDQL (*RDF Query Language*) et le SPARQL, par exemple, sont des langages de requêtes intentionnellement ressemblant au langage de requêtes SQL (voir [SEA04] et [PS06]). Le RDFT (*RDF Template Language 1.0*) ressemble un peu plus, dans sa conception, à la définition de *templates* faites par le XUL, mais s'appuie plus sur les concepts de "noeuds" et d'"arcs" du graphe que constitue le source de données RDF [DAV03].

Finalement, il a été décidé de conserver les *templates* tels que MOZILLA les a définis, mais en s'imposant d'analyser chacun des mécanismes afin de pallier certaines faiblesses flagrantes dans leur utilisation. Nous aurons l'occasion d'examiner ceci plus en détail au chapitre 4 ; avant cela, nous présenterons une vue d'ensemble du langage Mercury - qui a permis de développer l'*engine* correspondant au langage que nous avons défini - et nous examinerons le modèle de données RDF, afin de comprendre quelle forme prennent les données sur lesquelles sont effectuées les requêtes. Le chapitre 4 exposera le fonctionnement du principe de *template*, et présentera ensuite en détail le langage défini, et mettra en évidence quelques différences fondamentales avec les *templates* tels que définis dans XUL.

Chapitre 2

Le langage Mercury

2.1 Les paradigmes de programmation

2.1.1 Langages impératifs et effets de bord

Le paradigme de la programmation impérative se base sur les modèles de la machine de Turing et la machine de Von Neumann, avec des registres et une mémoire centrale ; au coeur de ces machines se trouve le concept de modification de l'état de cette mémoire grâce à des assignations successives [AAB04] [WIK03]. Il est possible de représenter un programme par une machine à états, mettant en évidence les états successifs de la mémoire ; le programmeur doit donc connaître à tout instant l'état de la mémoire - ou un modèle de celui-ci - que le programme modifie. Cette tâche peut être facilitée par plusieurs techniques ; parmi celles-ci, nous pouvons citer l'utilisation de variables dont la portée lexicale se limite à la procédure dans laquelle elles ont été définies et qui sont désallouées par le compilateur à la sortie de la procédure, ou l'encapsulation de données, à l'origine de la programmation structurée et de la programmation orientée objet.

Si la portée d'une variable est le plus souvent limitée à la procédure dans laquelle elle a été définie, il peut bien sûr arriver qu'une variable ou une zone de mémoire dispose d'une portée qui dépasse la procédure qui la contient, afin d'être accessible par d'autres procédures. Or, il arrive que des procédures doivent mettre à jour certaines variables ou zones de mémoire dans un but qui n'est pas directement lié à leur fonction, uniquement afin que les données partagées restent dans un état prévu par le programmeur [BOR03]. L'effet de bord est défini par cette capacité d'une fonction de modifier l'état d'une valeur (variable globale ou statique, argument d'une autre fonction, affichage ou écriture de données) autre que celle qu'elle renvoie. Ces effets de bord, qui sont plus la règle que l'exception en programmation impérative, sont source de nombreux *bugs* et rendent les programmes difficilement compréhensibles. En effet, le programme peut facilement se retrouver dans un état

imprévu : il suffit pour cela d'oublier de mettre à jour certaines données partagées, de mal ordonner les différentes parties du programme ou de désallouer une zone mémoire au mauvais moment...

2.1.2 Langages fonctionnels

La programmation fonctionnelle, elle, s'affranchit de ces effets de bord puisqu'elle interdit toute opération d'assignation [AAB04][BOR03]. Ce n'est pas des machines à états qui seront utilisées pour décrire un programme, mais un emboîtement de fonctions qui peuvent être vues comme des "boîtes noires" qui peuvent s'embriquer les unes les autres. Chaque fonction peut prendre plusieurs paramètres en entrée, mais ne retourner qu'une seule valeur ; le programme lui-même calcule dès lors une application (au sens mathématique du terme) qui ne donne qu'un seul résultat pour un ensemble de valeurs en entrée. Notons que la gestion de la mémoire est faite de façon automatique (via un *garbage collector*) ce qui simplifie donc la tâche au programmeur.

Certains langages fonctionnels - dits *impurs* - permettent l'utilisation de programmation impérative en offrant la possibilité de créer certaines variables assignables, cela car certains algorithmes s'expriment aisément avec des machines à états. Ces pratiques introduisent donc, éventuellement, des effets de bord.

Une propriété intéressante des langages fonctionnels est que ceux-ci respectent la transparence référentielle. La notion de transparence référentielle [HAB06] est définie, en mathématique, par le fait que la seule chose qui importe pour une expression est sa valeur, et toute sous-expression peut être remplacée par une autre, égale en valeur (on utilise l'expression "remplacer des égaux par des égaux"). Il existe donc une différence essentielle entre les notations mathématiques et les programmes impératifs, qui violent, lorsqu'il y a effets de bord, le principe qui voudrait que les manipulations algébriques, arithmétiques et logiques respectent la transparence référentielle [AAB04]. Pourtant, cette propriété de transparence référentielle est très utile ; en effet, le remplacement d'une fonction par une autre dans un programme ne respectant pas cette propriété peut s'avérer extrêmement coûteux, puisqu'il est possible que cette opération exige d'autres changements à différents endroits du code, ce qui peut nécessiter de retester entièrement ce code, car il n'est plus sûr, dès lors, que ce changement n'a pas modifié le comportement global du programme.

Les langages fonctionnels utilisent un puissant mécanisme nommé *fonctions d'ordre supérieur*. Les fonctions sont dites d'ordre supérieur lorsqu'elles peuvent prendre une ou plusieurs autre(s) fonction(s) comme argument(s), les stocker dans

des structures de données, et/ou renvoyer des fonctions comme résultats ; il est par exemple possible, en programmation fonctionnelle, de calculer la fonction dérivée d'une fonction passée en argument. Les fonctions d'ordre supérieurs permettent la *currification*, une technique dans laquelle une fonction est appliquée à ses arguments un par un, chaque application renvoyant une nouvelle fonction prête à accepter l'argument suivant. De plus, les langages fonctionnels emploient des types et des structures de données de haut niveau, comme les listes extensibles ; grâce à ceux-ci et à l'utilisation de la récursivité, il est possible de réaliser des opérations complexes en un nombre de lignes de code très réduit par rapport à ce que l'on réaliserait dans un langage impératif. Ce haut degré d'expressivité est, de plus, soutenu par le mécanisme d'*évaluation paresseuse* dont sont souvent dotés les langages fonctionnels : une expression ne sera évaluée que lorsque le résultat de l'évaluation est nécessaire au bon déroulement du programme.

2.1.3 Langages logiques

Tout comme la programmation fonctionnelle, la programmation logique est dite *déclarative*, car elle consiste à énoncer les propriétés d'un système de résolution (à les déclarer) plutôt qu'à décrire les opérations à effectuer comme dans le cas de la programmation impérative [FOU] ; par contre, elle diffère de l'approche fonctionnelle sur plusieurs points. Comme son nom l'indique, la programmation logique se base sur les théories de la logique mathématique pour définir les procédures. Ainsi, si un programme fonctionnel est composé de fonctions, le programme logique constitue, de son côté, un ensemble d'axiomes logiques décrivant le domaine sous-jacent au problème et les suppositions nécessaires à le résoudre ; ce problème sera formalisé par une assertion logique (la *requête*), alors qu'il sera plutôt mis sous forme d'expression en langage fonctionnel. L'exécution se fera ensuite par inférences successives au départ de la requête, et non par évaluation des expressions.

Tout comme les langages fonctionnels, les langages logiques permettent un haut degré d'expressivité, ceci grâce leur formalisme puissant (sous-ensemble de la logique du premier ordre) et à leur mécanisme d'exécution (résolution, recherche et *backtracking*).

2.2 Le langage Mercury

2.2.1 Vue d'ensemble

Le Mercury est un langage de programmation, créé et implémenté en Australie par des chercheurs de l'Université de Melbourne. Il est basé sur le paradigme de la programmation purement déclarative, et a été conçu pour pouvoir être utilisé

afin de créer des applications robustes de grande taille. Il intègre les paradigmes fonctionnels et logiques, puisqu'il utilise une syntaxe traditionnelle des langages de programmation logique, tout en donnant la possibilité à l'utilisateur de définir ses propres fonctions ; il évite aussi le recours à des constructions provenant du paradigme impératif.

Mercury impose au programmeur de spécifier le type, les modes et les déterminismes[1] des prédicats et de fonctions qu'il effectue. Le compilateur vérifie ensuite ces déclarations, et refuse le programme si celui-ci ne les satisfait pas.

De plus, pour faciliter la programmation "large échelle", permettre la compilation séparée et gérer une forme d'encapsulation, Mercury est doté d'un système simple de modules. Certains modules sont prédéfinis dans la librairie standard, afin de faciliter les tâches courantes en programmation [HCS+06].

2.2.2 Les avantages du Mercury

Bien que les langages logiques et fonctionnels soient théoriquement dotés d'une expressivité plus haute que les langages impératifs tels que le C ou le Pascal, les langages de ce type existant jusqu'alors - tel que le Prolog - ont été très peu utilisés pour écrire des programmes ayant des applications dans l'industrie, et ce pour plusieurs raisons majeures.

- Tout d'abord, les compilateurs de ces langages détectent généralement moins d'erreurs que ne le font les compilateurs des langages impératifs, ce qui oblige les utilisateurs à trouver les erreurs eux-même. Cela a donc un réel effet néfaste sur la productivité.
- Les implémentations de langages de programmation logique sont significativement plus lentes que celles de langages comme le C, ce qui pousse les *designers* à préférer ces langages dans le cas d'applications pour lesquelles la performance est importante.

L'utilisation du Mercury permet de résoudre ces deux problèmes à la fois. En effet, comme nous l'avons fait remarquer plus tôt, le Mercury offre un système de typage strict, de modes et de déterminisme qui permettent au compilateur de détecter une grande partie des erreurs de programmation au moment de la compilation, ce qui améliore la fiabilité des programmes, puisque beaucoup de types d'erreurs ne peuvent tout simplement pas apparaître dans un programme dont la compilation a réussi. La productivité est, elle aussi, fortement améliorée grâce au compilateur, qui est capable de repérer des erreurs dont la localisation exigerait un *debugging* manuel dans d'autres langages déclaratifs. Mercury garde, de plus, les

[1] On appelle déterminisme la caractérisation du nombre de réponses attendu de la part d'un prédicat ou d'une fonction (voir les points suivants pour plus de détails).

avantages de la programmation déclarative pure et permet, par exemple, d'éviter les effets de bord, et ainsi une large classe d'erreurs liées à ceux-ci, alors que ce type d'erreurs constitue une vraie plaie pour les programmes en langage impératif.

De plus, d'après les tests effectués par l'équipe de développement du Mercury, l'implémentation du Mercury serait près de deux fois plus rapide que le plus rapide des systèmes de programmation logique existant, et environ cinq fois plus rapide que SICSTUS-Prolog. Ceci est dû en partie au fait que le compilateur exploite la garantie de correction des déclarations pour améliorer de manière significative le code qu'il génère [HCS+06]. Pourtant, Mercury ne sacrifie pas la portabilité pour la vitesse puisque le compilateur Mercury génère du code C qui peut être compilé sur presque toutes les plateformes soft- et hardware [BEC06].

Au-delà de ces raisons techniques et théoriques, il existe d'autres raisons d'utiliser Mercury plutôt qu'un langage impératif. Comme il l'a déjà été dit précédemment, le Mercury est, par sa nature de langage déclaratif, de "plus haut niveau" que les langages impératifs ; il est donc possible de programmer la même chose qu'en C par exemple, mais avec moins de lignes. Ce qui est écrit dans le code ne dicte pas à la machine *comment* faire les choses, mais lui énonce plutôt *ce qu'*elle est censée faire. Le code est donc moins long, et surtout plus explicite pour quelqu'un qui le lira (car il ressemble à la spécification du code elle-même). Il sera donc plus aisé à comprendre, ce qui facilitera les tâches de maintenance.

L'écriture de code en Mercury impose par contre une certaine réflexion avant la création de fonctions et de prédicats ; ceci a tout de même comme avantage de rendre la tâche un tout petit peu moins "machinale" qu'en programmation impérative. Beaucoup d'erreurs "idiotes" sont donc évitées de cette façon, ce qui permet un gain de temps supplémentaire.

Examinons maintenant, sans trop entrer dans les détails syntaxiques, et en ignorant de nombreux mécanismes particuliers, comment ces différentes spécificités s'intègrent dans l'implémentation de Mercury.

2.3 La programmation en Mercury

Un programme Mercury consiste en un ensemble de modules, contenant chacun des suites de déclarations et de clauses. Ces clauses peuvent définir soit des fonctions, soit des prédicats. Prenons l'exemple d'un module unique, définissant une procédure bien connue.

Append

```
:- module append.

:- interface.
:- import_module list.
:- pred append(list(T),list(T),list(T)).

:- implementation.
:- mode append(in,in,out) is det.
:- mode append(in,in,in) is semidet.
append([],L2,L3):-
  L3=L2.
append([Debut|Fin],L2,L3):-
  L3=[Debut|Linterm],
  append(Fin,L2,Linterm).
```

2.3.1 Les modules

Un module commence toujours par la déclaration : `:- module` *nom*. Le nom du module est une lettre minuscule suivie de zero ou plusieurs lettres (majuscules ou minuscules) et *underscores*. Le compilateur Mercury s'attend à ce qu'un module appelé "`append`" se trouve dans un fichier nommé "`append.m`". Il existe toujours deux parties dans un module : l'interface (qui peut éventuellement être vide) et l'implémentation.

2.3.2 L'interface

L'interface contient les **déclarations** de tous les types, de tous les prédicats et toutes les fonctions qui seront exportées, et donc accessibles via "l'extérieur", c'est-à-dire via d'autres modules qui importeront celui-ci. Mercury s'assure de cette façon que les **définitions** (implémentations) des prédicats et des fonctions restent privées pour un module, puisqu'elles ne peuvent apparaître dans la section "`interface`" d'un module. Les définitions de types peuvent, elles, apparaître intégralement dans l'interface, mais il existe un mécanisme de "types abstraits" qui permet le même genre de dissimulation d'information (voir le point 2.3.8 à propos des types définis par l'utilisateur). Tous les programmes Mercury doivent exporter un prédicat "`main`", que le compilateur considère comme le point de départ de la totalité du programme Mercury. Le module "`append`" donné en exemple ne peut donc pas constituer, à lui seul, un programme Mercury puisqu'il n'exporte pas de

"`main`" ; pour pouvoir utiliser le prédicat qu'il définit, il faudra l'importer dans un autre module.

2.3.3 L'implémentation

Tout ce qui se trouve après une déclaration d'implémentation est considéré comme du détail d'implémentation privé, et est donc invisible pour les autres utilisateurs du module. Cette partie "*implémentation*" contiendra donc les définitions des fonctions et prédicats déclarés dans l'interface, ainsi que les définitions des types abstraits déclarés dans l'interface ; elle contiendra aussi les déclarations et implémentations des prédicats et fonctions dont on ne souhaite pas qu'elles soient exportées. Dans le cas où l'écriture de l'une de celles-ci nécessite la définition d'un nouveau type, ce type sera généralement déclaré et défini dans la partie implémentation, puisqu'il n'est utile que pour ce qui se trouve dans la partie privée du module. Il est toutefois possible de partager un type sans qu'il soit pour autant utilisé par une fonction ou un prédicat déclaré dans l'interface, et ceci afin que d'autres modules puissent s'en servir.

2.3.4 Les prédicats

Chaque prédicat est constitué d'une déclaration définissant le type de ses arguments et d'au moins une autre déclaration définissant les différents modes ou usages du prédicat, ainsi que de clauses. Dans le cas du module donné en exemple, le seul prédicat qu'il y ait est le prédicat "`append`". La déclaration des types des arguments est : "`:- pred append(list(T),list(T),list(T)).`". Cette déclaration nous informe sur le fait que nous allons définir un prédicat "`append`" prenant trois arguments du type "`list(T)`" (voir le point 2.3.7 à propos des types de base). Si nous examinons les modes possibles pour l'utilisation du prédicat, nous constatons que nous avons le choix d'utiliser les deux premiers arguments en entrée et le second en sortie dans le cas du mode "`:- mode append(in,in,out) is det.`", soit de passer les trois arguments en entrée, comme l'indique la déclaration de mode "`:- mode append(in,in,in) is semidet.`"(voir le point 2.3.9 à propos des modes). La partie "`is det`" du premier mode nous indique que ce prédicat est *déterministe* dans ce cas, ce qui signifie que "`append`" n'échouera jamais dans ce mode-là, et calculera toujours le même *output* pour des *inputs* égaux. La partie "`is semidet`" du second mode nous indique que ce prédicat devient, dans ce mode, *semi-déterministe*, ce qui signifie que "`append`" pourra éventuellement échouer ; nous verrons cela plus en détail à la section 2.3.10.

Une clause d'un prédicat est toujours de la forme "`tête :- corps`" ; dans le cas de notre "`append`", il en existe deux, définies par :

```
append([],L2,L3):-
  L3=L2.
append([Debut|Fin],L2,L3):-
  L3=[Debut|Linterm],
  append(Fin,L2,Linterm)..
```

Une clause comprend donc une tête et un corps, séparés par le symbole ":-". La tête de la clause indique qu'il s'agit d'une définition de "`append`" et que les noms de ses trois arguments sont "`L1`", "`L2`" et "`L3`". Les symboles "`( ... ; ... )`" dans le corps de la clause signifient qu'il faut effectuer un choix entre l'exécution des buts situés avant le ";" ou après. Ce choix se fait ici sur base de la valeur de "`L1`".

2.3.5 Les fonctions

Comme dans le cas des prédicats, une fonction est constituée d'une déclaration et d'une clause. Puisqu'il n'y a pas de fonction définie dans l'exemple précédent, écrivons ici une fonction simple calculant la suite de Fibonacci :

```
:- func fib(int) = int.
    fib(N) = X :-
    ( if N =< 2
      then X = 1
      else X = fib(N - 1) + fib(N - 2)
    ).
```

La déclaration de cette fonction *fib* est : "`:- func fib(int) = int.`". Cette déclaration indique que la fonction définie prend un argument du type "`int`", et que le résultat de la fonction est de type "`int`" aussi. Remarquons qu'il n'y a pas ici de déclaration de mode ; ceci est dû au fait que le compilateur a, dans ce cas, un comportement par défaut qui satisfait l'utilisation que l'on souhaiterait faire de cette fonction. Nous verrons cela plus en détail par la suite.

La clause d'une fonction n'a pas la même forme que celle d'un prédicat ; en effet, celle-ci se présente sous la forme : "`tête = résultat :- corps`". Il est même parfois possible de l'écrire sous la forme "`tête = résultat`" ; ainsi, nous pouvons transformer la clause de l'exemple précédent en :

```
    fib(N) = ( if N =< 2
               then 1
               else fib(N - 1) + fib(N - 2)
             ).
```

Le choix de l'utilisation d'un prédicat ou d'une fonction pour la réalisation d'une procédure est plutôt une question de goût qu'une question de bonne pratique ; toutefois, il est généralement conseillé de remplacer par une fonction les prédicats déterministe n'ayant qu'un argument en sortie.

2.3.6 Les importations de modules

De la même façon qu'il est possible d'exporter, dans un module, les types, fonctions et prédicats dont les déclarations se trouvent dans la partie "`interface`" du module, il est possible d'importer les types, fonctions et prédicats dont les déclarations ont été placées dans les interfaces d'autres modules. Cette importation se fait grâce à la ligne "`:- import_module` *nom_du_module*." placée au début de l'interface ou de l'implémentation.

La question à se poser est bien sûr : "dans quels cas placer cette ligne dans l'interface, et dans quels cas la placer dans l'implémentation ?". La réponse est assez simple : il ne faut importer les modules dans le section implémentation que si ce que ceux-ci exporte est utilisé uniquement dans cette section. Tous les modules importés dans l'interface sont donc censés être utilisés dans l'interface ; par contre, les modules utilisés à la fois dans l'interface et dans l'implémentation ne doivent être importés qu'une seule fois, dans la première citée.

2.3.7 Les types de base et la librairie standard

Afin d'utiliser n'importe quel type de base, tel que les chaînes de caractères, les entiers, les liste, etc., il est nécessaire d'importer les modules définissant ces types ; ces modules se trouvent dans la librairie standard de Mercury.

L'importation d'un module définissant un type dans un programme Mercury ne permet pas seulement d'utiliser ce type et de le manipuler, mais aussi d'utiliser une variété de fonctions et prédicats prédéfinis sur ce type ; ceux-ci permettront d'éviter d'avoir recours à l'écriture de mécanismes triviaux (parfois moins triviaux...), demandant parfois l'utilisation de fonctions et prédicats intermédiaires, ce qui a comme inconvénient de surcharger le code avec des procédures qui ne sont pas directement liées au problème traité, tout cela étant donc néfaste à une lecture et une compréhension aisées du code.

Parmi les très nombreux types définis se trouve le type "io", qui a une utilité assez particulière en Mercury ; celui-ci permet bien sûr d'utiliser toutes les instructions "classiques" d'entrées-sortie, comme l'affichage à l'écran, la lecture d'un

fichier, la création de canaux d'*input/output*, etc., mais il permet aussi de sauvegarder l'intégrité mathématique tout en permettant à un programme Mercury de communiquer avec le monde extérieur.

Pour éclaircir un peu ce point, prenons l'exemple classique du programme "*Hello, World!*" :

```
:- module hello.
:- interface.
:- import_module io.
:- pred main(io::di, io::uo) is det.
:- implementation.
main(IOState_in, IOState_out) :-
    io.write_string("Hello, World!\n", IOState_in, IOState_out).
```

La question à se poser en voyant le prédicat "`main`" de ce programme est bien sûr : à quoi servent les deux arguments "`IOState_in`" et "`IOState_out`" ? La réponse est que "`IOState_in`" représente l'état du monde lorsque le prédicat *main* est appelé, "`IOState_out`" représentant l'état du monde après son exécution. Chaque prédicat ou fonction qui exécute des actions d'entrées/sorties doit posséder des arguments de ce type.

Remarquons ici que la déclaration des types et des modes des arguments, ainsi que du déterminisme du prédicat sont faits en une seule ligne ; ceci est un sucre syntaxique applicable lorsqu'un prédicat n'est défini que pour un seul mode.

Examinons maintenant brièvement deux types intéressants définis dans Mercury : les listes et les *maps*.

Les listes

Le constructeur d'une liste en Mercury est "`[|]`". Ainsi, la liste "`[1,2,3]`" pourra aussi bien s'écrire "`[1,2,3|[]]`" que "`[1,2|[3]]`" ou encore "`[1|[2,3]]`". Ce type est très largement utilisé, et il existe des prédicats et fonctions très pratiques définis sur ce type. En voici un exemple :

```
:- pred list.foldl(pred(L, A, A), list(L), A, A).
```

L'appel `list.foldl(Pred, Liste, Debut, Fin)` appelle le prédicat "`Pred`" avec chacun des éléments de la liste "*List*" (partant de la gauche vers la droite, pour faire le contraire il existe le prédicat `list.foldr`) et un accumulateur (avec la valeur initiale "`Debut`" - un accumulateur peut être une liste, un entier, un tableau, etc.), et renvoie la valeur finale de cet accumulateur dans "`Fin`".

Il existe plusieurs prédicats et fonctions permettant l'exécution de procédures similaires ; ils permettent de réduire considérablement le temps de codage et le nombre de lignes nécessaires pour diverses tâches, et de rendre le code bien plus clair puisque leur utilisation permet d'éviter l'écriture de code qui n'est pas directement lié au problème traité.

Dans notre module "*append*", nous pouvons remarquer que la déclaration de prédicat "`:- pred append(list(T),list(T),list(T))`" utilise des listes d'éléments du type "`T`". Les types commençant par une lettre majuscule, comme c'est le cas ici, sont considérés comme des types polymorphiques. Les trois listes déclarées ici devront donc être des listes contenant des éléments d'un même type, n'importe lequel.

Les *maps*

Le type "*map*" est aussi connu sous le nom de "tableau associatif" ; il s'agit d'une collection de paires (clé - donnée), qui permet d'obtenir n'importe quel élément de donnée à partir de la clé correspondante, qui requiert bien sûr d'être unique. Les *maps* sont implémentées dans Mercury grâce à l'utilisation d'arbres binaires équilibrés.

Comme dans le cas des listes, il existe beaucoup de prédicats définis sur ce type ; ceux-ci servent notamment à créer la *map*, y ajouter ou en supprimer des paires, mettre à jour les données associées à une clé, rechercher une donnée à partir d'une clé, etc., ainsi que d'effectuer certaines actions similaires à celles que l'on peut effectuer avec des liste. Le prédicat `map.foldr`) existe donc, et permet de faire, avec des *maps*, quelque chose de semblable à ce que permet `list.foldr`) avec des listes.

2.3.8 Les types définis par l'utilisateur

La définition de nouveaux types se base sur le concept d'union discriminée. Cela permet de définir une structure de données utilisée pour des données qui peuvent être de plusieurs types différents, mais fixés. Par exemple, nous pourrions définir un type permettant une représentation des cartes à jouer :

```
:- type carte_a_jouer ---> carte(rang, blason) ; joker.
:- type rang ---> as ; deux ; trois ; quatre ;
                  cinq ; six ; sept ; huit ;
                  neuf ; dix ; valet ; dame ; roi.
:- type blason ---> coeurs ; piques ; carreaux ; trefles.
```

Le constructeur de données définissant les valeurs de l'union discriminée se trouve à droite de la flèche : le blason a quatre valeurs différentes, le rang en a 13 et la

carte à jouer, 53 (52 cartes possibles plus le joker).

Les champs d'un constructeur de données peuvent être nommés, comme c'est le cas dans l'exemple suivant, uilisé pour représenter un compte bancaire :

```
:- type compte_bancaire ---> compte( nom :: string,
                                     no_compte :: int,
                                     fonds :: float ).
```

Ces noms de champs sont pratiques, car ils permettent d'accéder aux champs directement sans avoir à "décomposer" le compte bancaire précédemment. Par exemple, à la place de devoir écrire :

```
Cpte = compte(Nom, Num, Fonds),
( if Fonds >= SommeDemandee then
... debit SommeDemandee de Cpte ...
else
... rejet demande debit ...
)
```

Il est possible d'écrire :

```
( if Cpte^fonds >= SommeDemandee then
... debit SommeDemandee de Cpte ...
else
... rejet demande debit ...
)
```

Notons que cette technique permet non seulement d'accéder mais aussi de mettre à jour les champs d'une variable. Notons aussi qu'il n'est pas obligatoire de nommer chacun des champs du constructeur de données ; dans ce cas, seuls les champs nommés seront accessibles sans décomposition et recomposition de de la donnée.

Il n'est pas permis, dans un module, de définir plusieurs types utilisant les mêmes noms de champs ; les lignes suivantes ne sont donc pas correctes :

```
:- type chat ---> chat(nom :: string).
:- type chien ---> chien(nom :: string).
```

Mais peuvent être écrites de cette manière :

```
:- type chat ---> chat(nom_chat :: string).
:- type chien ---> chien(nom_chien :: string).
```

L'accès à un champ peut échouer si le type de donnée a plus d'un constructeur. Par exemple :

```
:- type carte_a_jouer ---> carte(rang::rang,
                                 blason::blason) ;
                           joker.
```

Une clause utilisant l'expression `Carte^rang` pourra, dans ce cas, échouer s'il s'avère que `Carte` est un `joker`.

Remarquons aussi que l'accès aux champs peut être chaîné ; par exemple :

```
:- type employe ---> employe(id :: int, contact :: contact details).
:- type contact_details ---> contact_details(adresse :: string,
                                              telephone :: int).
```

Le champ `adresse` de `contact_details` pourra alors être accédé grâce à l'expression `Employe^contact^adresse` si `Employe` contient une valeur du type `employe`.

Types abstraits

Comme il l'a été vu plus tôt, le mécanisme d'*interface* et d'*implementation* permet d'assurer la dissimulation des détails d'implémentation d'une fonction ou d'un prédicat lors de l'exportation de ceux-ci. Puisqu'il est permis de définir entièrement un type dans la partie *interface* du module, cette dissimulation n'est pas assurée pour les définitions de types, mais elle reste possible grâce à la notion de *type abstrait*. Cela permet de n'indiquer, dans l'*interface*, que la déclaration du type, la définition se trouvant dès lors dans la partie *implementation* du module. Prenons un simple exemple pour illustrer ceci :

```
:- module dictionnaire.
:- interface.
   ...
:- type dictionnaire(Cle, Valeur).
   ...
:- implementation.
:- import module list.
:- type dictionnaire(Cle, Valeur) == list({Cle, Valeur}).
   ...
```

La déclaration de type "`:- type dictionnaire(Cle, Valeur).`" se situe dans l'*interface* ; ce type sera donc exporté, mais sa définition :

```
:- type dictionnaire(Cle, Valeur) == list({Cle, Valeur}).
```

ne sera pas accessible, puisque celle-ci se situe dans la partie *implementation*.

2.3.9 Les modes

Intéressons-nous de nouveau au prédicat "`append`" que nous avons défini au début de cette section 2.3. Celui-ci avait une déclaration des types qu'il utilisait ainsi que deux déclarations des modes possibles d'utilisation :

```
   ...
:- pred append(list(T),list(T),list(T)).
   ...
:- mode append(in,in,out) is det.
:- mode append(in,in,in) is semidet.
   ...
```

Maintenant que nous avons déjà examiné de près la ligne de déclaration des types des arguments "`:- pred ...`", penchons-nous un peu plus sur les modes de ces arguments. Ici, nous sommes face aux modes classiques "`in`" et "`out`", ce qui signifie que les arguments correspondants seront utilisés soit comme entrées, soit comme sorties du prédicat `append`. Il existe par contre des modes moins ordinaires, que nous avons déjà rencontrés dans le programme "*Hello, World!*" défini à la section 2.3.7.

```
   ...
:- pred main(io::di, io::uo) is det.
   ...
```

Les modes `di` et `uo` sont nommés ainsi car il s'agit d'abréviations pour *destructive input* et *unique output*. Ils peuvent être compris assez intuitivement en examinant la façon dont ils sont utilisés. En effet, comme il est assez insensé de réutiliser un ancien état d'entrée/sortie - une fois l'impression d'un document terminée, il est impossible de récupérer l'encre sur le papier et de la remettre dans la cartouche - les états d'entrée/sortie sont uniques. Les actions mettant en oeuvre des entrées/sorties doivent donc créer un état `io` qui remplacera le précédent. De façon similaire, comme il est impossible de "copier" l'état du monde et de créer ainsi deux univers parallèles, les états `io` nécessitent d'être uniques.

```
io.write_string("La signification de la vie est", IO0, IO1),
io.write_int(42, IO1, IO2)
```

Dans l'exemple précédent, l'écriture de "`La signification de la vie est`" va détruire `IO0` et produire `IO1`, et l'écriture de "`42`" détruira `IO1` et produira `IO2`. La réutilisation des entrées détruites, ou l'utilisation multiple des sorties uniques générerait des erreurs à la compilation.

Comme l'exemple précédent le montre, il faut trouver, pour chaque nouvel état `io`, un nouveau nom pour la variable qui le représente. Nommer tous les états

intermédiaires devenant rapidement pénible, Mercury définit un sucre syntaxique sous la forme de *variables d'états*. L'exemple précédent pourrait donc être écrit :

```
io.write_string("La signification de la vie est", !IO),
io.write_int(42, !IO)
```

Le code est transformé par le compilateur en quelque chose d'équivalent à ce que nous avons écrit précédemment ; chaque occurence de `!IO` représente en réalité deux variables normales.

2.3.10 Le déterminisme

Chacun des modes d'un prédicat ou d'une fonction est catégorisé suivant le nombre de solutions qu'ils peuvent avoir. Ainsi, si chaque appel possible à un mode particulier d'un prédicat ou d'une fonction

- a exactement une solution, alors le mode est "déterministe" (det) ;
- a soit une solution, soit aucune solution, alors ce mode est "semidéterministe" (semidet) ;
- a au moins une solution mais pourrait en avoir plus, alors ce mode est "multisolution" (multi) ;
- a zéro ou plusieurs solutions, alors ce mode est "non-déterministe" (nondet) ;
- échoue sans produire de solution, alors ce mode a le déterminisme d'un échec (failure).

Dans notre exemple "*Append*", le prédicat "`append`" était déclaré pour plusieurs modes différents ; chacun de ceux-ci avait un déterminisme associé.

```
   ...
:- mode append(in,in,out) is det.
:- mode append(in,in,in) is semidet.
   ...
```

Intuitivement, il est très facile de comprendre le "pourquoi" de ces déterminisme ; en effet, lorsque deux listes (du même type) sont passées en entrée, la concaténation de celles-ci donnera toujours une et une seule solution en sortie. Si, par contre, la troisième liste est aussi passée en entrée, la tâche du compilateur devient alors de "vérifier" si la troisième liste est bien la concaténation des deux premières ; cette tâche peut évidemment terminer soit sur un succès, soit sur un échec, ce qui justifie le déterminisme "`semidet`" de ce mode.

Chapitre 3

Le RDF

3.1 Introduction

Qu'est-ce que RDF ? Le langage RDF (*Resource Description Framework*) est une spécification créée pour le traitement des (méta)données ; son but est de rendre interopérables des applications qui échangent des informations incompréhensibles par les machines sur le Web, et de faciliter le traitement automatique de données des ressources Web [LS99]. Cette simple définition ne satisfait évidemment pas notre curiosité concernant RDF ; pour pouvoir réellement comprendre ce langage et les concepts qui le sous-tendent, le mieux est sans doute d'essayer de comprendre quel raisonnement s'est trouvé à la base de sa création.

La première étape est donc de s'intéresser à ce qu'il faut traiter, c'est-à-dire l'information. Nous pourrions faire une catégorisation arbitraire entre information du type *contenu*, du type *donnée* et du type *fait*. Les informations du type *contenu* sont généralement traitées comme un "tout" : afficher une page HTML, jouer un fichier de musique... Les informations du type *donnée* sont, elles, plutôt traitées par parties : ajouter un enregistrement dans une base de données, trier une liste d'objets... Les informations du type *fait* sont les moins communes en informatique, et sont utilisées quotidiennement autant par monsieur tout-le-monde que par les spécialistes de gestion des connaissances. Ces faits peuvent être, par exemple : "J'ai été au magasin", "Maman mange des chips au sel", "La Terre est au centre de l'Univers"... Peu importe la véracité de ces faits, l'important étant que l'on puisse les écrire et les formaliser, afin de pouvoir les traiter.

Ce chapitre a donc pour but de montrer comment, en partant de faits communs utilisés quotidiennement dans le langage écrit et parlé, il est possible de créer une spécification formelle de ceux-ci. Nous prendrons donc, pour commencer, un fait quelconque que nous traiterons par étapes, en affinant au fur et à mesure notre modélisation pour enfin mettre en place le modèle de graphes que définit RDF.

3.2 Des faits au RDF

3.2.1 Les faits

Le monde est bondé d'information du type *fait*, mais la plupart de ces faits ne sont pas formalisés, et ne peuvent donc être traités de façon automatique [McF03]. Le RDF, lui, définit un système spécifique d'écriture de faits. Par le passé, quelques systèmes traitaient déjà un sous-ensemble de ce type d'information. Si l'on examine par exemple les marque-pages de Mozilla, on se retrouve face à une système du type *fait*, assez trivial. En voici un exemple :

```
<A HREF="http://www.mozilla.org/"
   ADD_DATE="961099870"
   LAST_VISIT="1055733093"
   LAST_CHARSET="ISO-8859-1">
</A>
```

Ce marque-page nous informe à propos d'une URL[1]. Cette URL est le sujet d'un fait exprimant plusieurs propriétés : la date à laquelle elle a été ajoutée, la dernière date de visite, etc. Ce semi-XML n'est pas un formalisme qui peut s'appliquer à n'importe quel fait, mais le RDF est là pour apporter ce nouveau formalisme.

Il est à noter que les données telles que les marque-pages sont souvent appelées "métadonnées". Ce terme est censé aider l'esprit à créer une séparation entre les données que l'URL représente (le contenu) des données qui décrivent cet URL. Malheureusement, si un programmeur doit, par exemple, créer un système de gestion des marque-pages, les soit-disant "métadonnées" deviennent la seule information intéressante, et sont donc dès lors considérées comme les données sur lesquelles travailler. Cette séparation étant source de confusion - les *données* de l'un sont les *métadonnées* de l'autre - elle a été quelque peu mise de côté en RDF, où l'on préférera ne parler que de *faits*, sans distinctions entre eux.

[1]Selon le site *Comment ça marche ?* (`http://www.commentcamarche.net` , le 10 avril 2006), "une URL (Uniform Resource Locator) est un format de nommage universel pour désigner une ressource sur Internet. Il s'agit d'une chaîne de caractères ASCII imprimables qui se décompose en cinq parties : le nom du protocole (HTTP pour la plupart), l'identifiant et le mot de passe (facultatif, permet de spécifier les paramètres d'accès au serveur), le nom du serveur (notez qu'il est possible d'utiliser l'adresse IP du serveur, ce qui rend par contre l'URL moins lisible), le numéro de port (lorsque le service Web du serveur est associé au numéro de port 80, le numéro de port est facultatif), et le chemin d'accès à la ressource (le répertoire et le nom du fichier demandé)".

3.2.2 Les faits et les structures de données

Les faits sont utilisés pour décrire des données ; la modélisation de ces données se fait grâce à des structures de données, en utilisant éventuellement des outils tels que les diagrammes UML. Examinons d'abord un exemple simple afin de comprendre de quelles manières un fait peut être modélisé ; choisissons pour cela une simple scène d'un garçon et son chien jouant avec une balle sur la plage. Pour commencer, considérons les structures de données traditionnelles. En Java, par exemple, l'information pourrait être stockée comme un objet :

```
{ garçon:"Tom", chien:"Médor", balle:"tennis", plage:"Malibu" }
```

Ceci ressemble d'ailleurs aussi à ce qui pourrait être une structure (struct) en C. Nous pourrions aussi imaginer stocker cela dans une liste de *strings* :

```
[ "Tom", "Médor", "tennis", "Malibu"]
```

Il y a en général de nombreuses façons de modéliser et de stocker des données, chaque façon ayant ses avantages et ses inconvénients (une liste, par exemple, est ordonnée et exige l'utilisation de données du même type). La seule chose qui importe, finalement, est que la modélisation choisie convienne au problème traité.

En RDF, la modélisation des données requiert l'utilisation de tuples. Un tuple est un groupement de N composants, éventuellement de types différents ; peu de langages supportent les tuples directement, comme c'est le cas du SQL par exemple. Utilisons, pour notre exemple, la notation :

```
<- Tom, Médor, tennis, Malibu ->
```

Les tuples sont ordonnés (comme une liste) mais non numérotés (contrairement aux tableaux). Ici, l'information réellement stockée dans le tuple est très limitée et se résume à "Tom, Médor, tennis et Malibu sont associés d'une certaine manière".

3.2.3 Prédicats et triples

Pour rendre l'exemple plus adapté à la suite, laissons de côté la plage sur laquelle le garçon et le chien se trouvent. Si nous voulons modéliser ce nouvel exemple de façon plus complète qu'auparavant, afin notamment de rendre explicites les relations entre les protagonistes de notre scénario, une des manières naturelles serait de créer des objets distincts :

```
var boy = { Person_id:1, name:"Tom", Dog:dog, Ball:ball };
var dog = { Dog_id:2, name:"Médor", Person:boy, Ball:ball };
var ball = { Ball_id:5, type:"tennis", color:"vert" };
```

Mais il est évident que, de cette manière, l'accent est mis sur les *protagonistes* du scénario et leurs *relations* sont totalement mises de côté. L'option habituelle pour remédier à cela est l'ajout d'objets, de tables,... selon le système de modélisation dans lequel on travaille. Dans le cas des faits, la solution est d'introduire chaque relation existante comme une terme - appelé *prédicat* - dans un tuple.

Il existe donc un groupe de tuples (de faits) ayant la particularité de contenir des termes qui retiennent des informations sur les relations entre les données. De façon naïve, nous pourrions introduire de tels termes dans notre exemple précédent :

```
<- 1, Tom, propriétaire, 2, joue-avec, 5 ->
<- 2, Médor, chien-de, 1, joue-avec, 5 ->
<- 5, tennis, vert ->
```

Dans ce dernier exemple, il apparaît clairement que l'information retenue est beaucoup plus complète que dans les tentatives précédentes, mais cette façon de faire n'est pas encore idéale ; en effet, les tuples peuvent contenir plusieurs prédicats, ce qui rend le traitement automatique des faits difficile. Si nous "séparons" les tuples afin qu'ils contiennent un prédicat au plus, cela donne :

```
<- 1, Tom, propriétaire, 2 ->
<- 1, Tom, joue-avec, 5 ->
<- 2, Médor, chien-de, 1 ->
<- 2, Médor, plays-with, 5 ->
<- 5, tennis, vert ->
```

Les prédicats sont maintenant séparés, et les tuples sont sans doute encore plus simples à lire ; ce processus peut être comparé à la normalisation en ingénierie des bases de données. Malgré tout, il est encore possible de raffiner cette solution, en décidant que chaque tuple contienne le même nombre de termes, à savoir trois. Pour cela, nous introduisons ici une dernière propriété, associant un objet (son identifiant) à son nom.

```
<- 1, est-nommé, Tom ->
<- 1, propriétaire, 2 ->
<- 1, joue-avec, 5 ->
<- 2, est-nommé, Médor ->
<- 2, chien-de, 1 ->
<- 2, joue-avec, 5 ->
<- 5, type, tennis ->
<- 5, couleur, green ->
```

Tous les tuples sont maintenant des triples ; cette écriture par triples est la façon normalisée de modéliser les faits, et la plus largement utilisée. Chaque terme des ces triples portent un nom formel : le terme de relation s'appelle toujours le *prédicat*, le premier terme est nommé le *sujet* alors que le troisième terme est appelé l'*objet*.

En observant les différents langages qui utilisent le concept de (tuples) prédicats, on constate que ceux-ci peuvent être écrits de différentes façons, comme c'est le cas en Prolog, où l'on écrira :

`prédicat(sujet, objet)` Exemple : `joue-avec(1,5)`

Alors qu'en Lisp, on rencontrera plutôt :

`(prédicat sujet objet)` Exemple : `(joue-avec 1 5)`

Le RDF est, comme nous le verrons ci-après, un modèle de graphe avant tout, mais il existe plusieurs syntaxes pour représenter ces graphes en XML, et celles-ci privilégient généralement l'ordre naturel `<-sujet prédicat objet->`.

3.2.4 Faits et graphes

Maintenant qu'il a été défini quel était le "bon format" pour écrire un fait, il serait intéressant de se pencher sur la façon dont ceux-ci seront stockés dans un ordinateur.

Une première approche, la plus simple, serait de les stocker comme des éléments indépendants. Dans le cas d'une base de données par exemple, cela signifierait l'utilisation d'une table à trois colonnes, chaque ligne contenant un triple. L'utilisation d'une telle approche a comme avantage d'être très flexible. En effet, il est possible d'ajouter ou supprimer des triples n'importe quand. Il n'y a aucune structure interne, donc aucune maintenance à y effectuer. Il est aussi possible de fusionner deux ensembles de triples sans aucun problème, et d'ajouter des faits venant de sources différentes.

La deuxième façon d'organiser les faits est de reconnaître qu'il existent des liens entre eux, créant ensemble une structure globale, qui pourrait être stockée comme une structure de données traditionnelle, avec des pointeurs et des références entre les tuples. De plus, ses liens étant assez généraux, cette structure est un graphe ; un tel graphe peut être représenté visuellement, comme c'est le cas pour notre exemple du garçon, du chien et de la balle (voir figure 3.1).

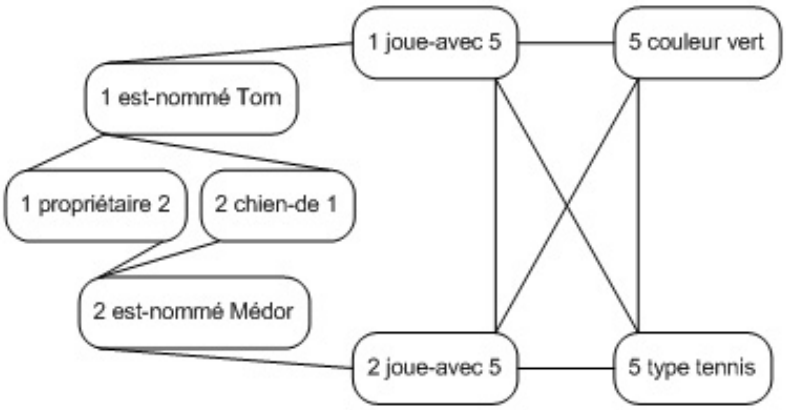

FIG. 3.1 – Modélisation des faits tenant compte des liens entre eux

Ce graphe est de toute évidence fort compliqué pour un système ne consistant qu'en trois entités ! Une première simplification possible est l'ajout de points d'intersection contenant les *id*'s contenus dans les tuples, comme on peut le voir figure 3.2.

De plus, certains tuples sont opposés l'un à l'autre (comme c'est le cas "*propriétaire-de / chien-de*" dans cet exemple) ; de tels duplicats peuvent être supprimés en utilisant des axes orientés (voir figure 3.3).

Finalement, en remarquant que chaque terme prédicat n'a qu'un seul axe entrant et un seul axe sortant, les prédicats pourraient aussi bien être indiqués sur le graphes comme les labels des axes orientés plutôt que des noeuds (voir figure 3.4).

Ce dernier graphe est très clair ; on y distingue les prédicats intermédiaires, qui ne font qu'ajouter de l'information à des identificateurs (*est nommé*, *type*, *couleur*), et les prédicats qui donnent des informations sur les relations entre ces identificateurs.

Ce graphe suit les notations RDF officielles, où des cercles (ou des ellipses) sont utilisées pour représenter les identificateurs, et des rectangles pour les valeurs littérales. Toutefois, en RDF, les identificateurs ne sont pas des chiffres mais des *URL*, comme nous le verrons un peu plus tard.

Une troisième et dernière façon d'organiser les faits se situe à mi-chemin entre les deux précédentes ; en effet, celle-ci n'est structurée que "partiellement" grâce à des *conteneurs*. Comme dans notre première approche, les prédicats sont indépendants et non-organisés. Toutefois, les conteneurs permettent de regrouper des éléments dont on juge qu'il est pertinent de les rassembler, par exemple parce qu'ils donnent tous des informations à propos d'un même thème. Si, par la suite, nous avons besoin

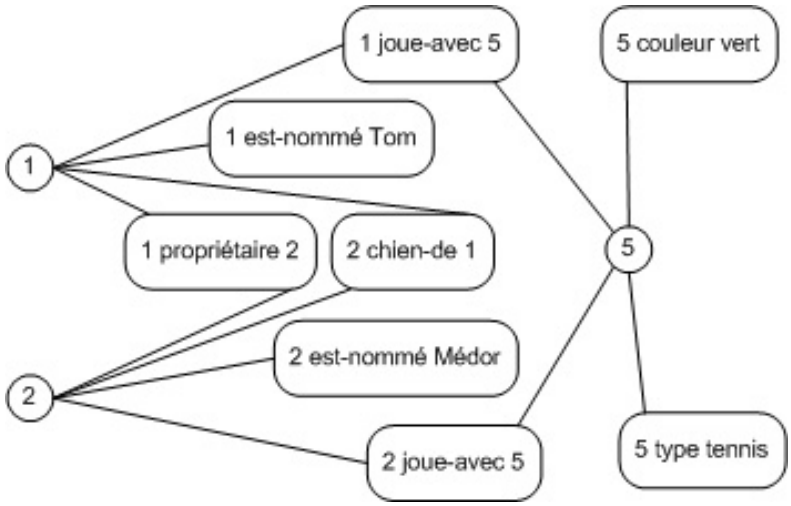

FIG. 3.2 – Modélisation des faits : utilisation d'identifiants

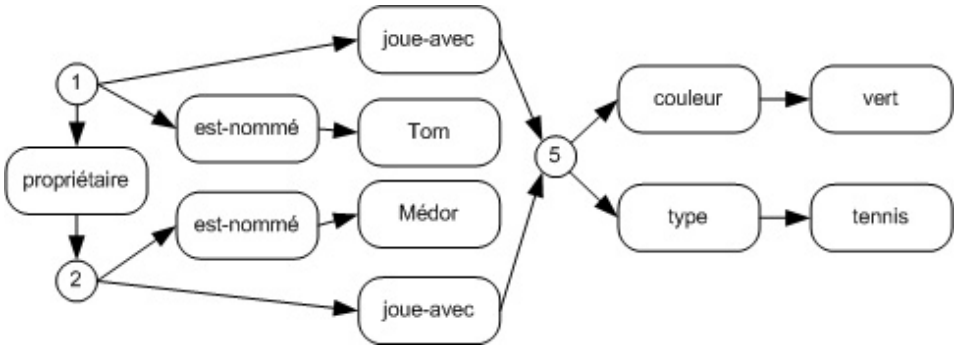

FIG. 3.3 – Modélisation des faits : utilisation d'axes orientés

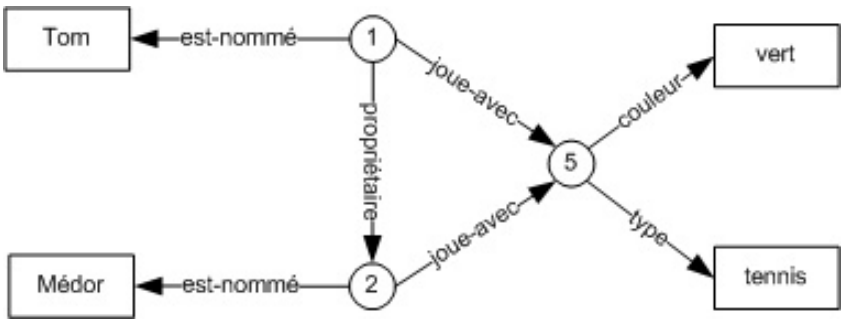

FIG. 3.4 – Modélisation des faits avec des axes orientés et nommés

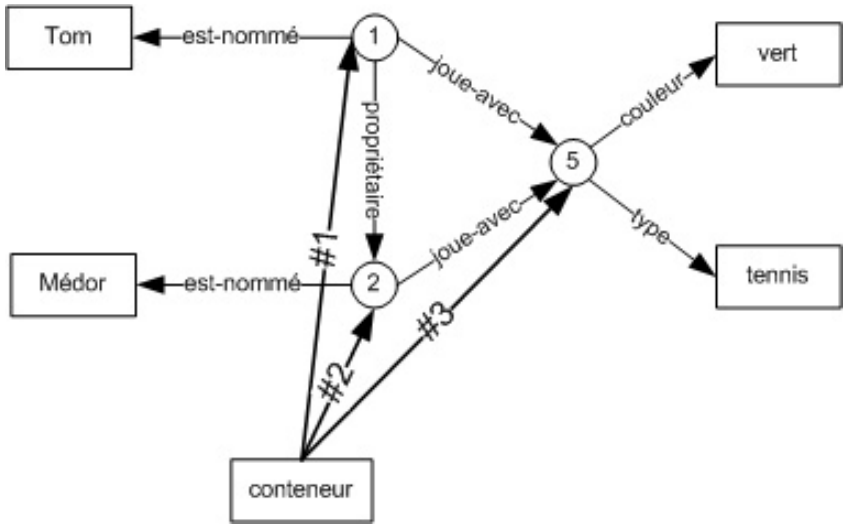

FIG. 3.5 – Modélisation graphique de faits utilisant un conteneur

de connaître des choses à propos de ce thème, il sera plus aisé de les retrouver, car le conteneur nous informera sur les faits traitant de cela.

Puisque le RDF ne peut représenter que des faits, ces conteneurs sont donc constitués de faits ; n'importe quel terme d'un fait stocké peut dès lors apparaître dans un ou plusieurs conteneurs, bien que le sujet sera généralement choisi. Le terme contenant le mot "conteneur" est le point de départ du conteneur ; chaque prédicat le concernant sera nommé automatiquement grâce à un numéro. La figure 3.5 nous donne un aperçu des faits qui seraient ajoutés si nous décidions de rassembler les trois identifiants de Tom, du chien et de la balle dans un même conteneur.

3.3 La syntaxe XML du RDF

Le RDF est donc un modèle de graphe avant tout, mais afin de pouvoir manipuler et échanger les faits modélisés, il est intéressant d'en définir une syntaxe écrite. Cette syntaxe utilise l'encodage du XML ; celui-ci apporte de plus la possibilité d'utiliser, comme nous le verrons ci-après, des *espaces de noms*, ce qui facilite l'écriture des identificateurs. Il existe en réalité deux syntaxes RDF en XML : la syntaxe complète, dite *syntaxe de sérialisation* qui exprime toutes les capacités du modèle de données d'une manière très courante, et la *syntaxe abrégée* qui inclut des constructions supplémentaires, fournissant ainsi une forme plus compacte pour représenter un sous-ensemble du modèle de données. Ici, il sera surtout question de la syntaxe complète ; la syntaxe abrégée fera l'objet de quelques points, mais il n'en sera pas fait une description exhaustive.

3.3.1 Les balises RDF

Le but premier du RDF étant d'offrir un moyen de spécifier les faits, nous pourrions imaginer que la syntaxe d'un fait soit quelque chose comme ceci :

```
<fait sujet="..." predicat="..." objet="..."/>

 ou :

<fait>
  <sujet .../>
  <predicat .../>
  <objet .../>
</fait>

 ou :

<sujet ... predicat="..." objet="..."/>
```

Mais ces formes n'existent pas en RDF ; c'est plutôt cette forme-ci qui est utilisée :

```
<fait sujet="...">
  <predicat>
    objet
  </predicat>
</fait>
```

De plus, si c'est effectivement cette forme-là qui est utilisée, les balises ne portent pas exactement les noms apparaissant ici. Ainsi, en RDF, on retrouvera :

```
<Description about="http://www.mozilla.org/">
  <NC:LastVisited>
    10 January 2004
  </NC:LastVisited>
</Description>
```

Nous savons maintenant dresser une petite table de correspondance entre les concepts précédemment étudiés et la syntaxe utilisée en RDF (voir tableau 3.1). Ainsi, dans le dernier exemple présenté, `http://www.mozilla.org/` est le sujet, le prédicat est `NC:LastVisited`, et l'objet *10 January 2004*.

Concept	Terme RDF	Syntaxe RDF
Fait	Description	`<Description>`
Sujet	Resource	`about=...`
Prédicat	Property	*Défini par l'utilisateur*
Objet	Valeur d'une propriété	`resource=...` ; *texte*

TAB. 3.1 – Correspondances faits - RDF

Finalement, la liste complète des balises spécifiques à RDF se résume tout simplement à :

```
<RDF> <Description> <Seq> <Bag> <Alt> <li>
```

La balise `<RDF>` est la balise qui encadre tous les documents RDF. La balise `<Description>` est, comme nous l'avons vu, une balise servant à annoncer la description d'un fait. Les balises `<Seq>`, `<Bag>`, `<Alt>` et `<li>` sont destinées à mettre en oeuvre les mécanismes de conteneurs que nous avons évoqués plus tôt.

3.3.2 Les identifiants

Lors de notre exemple du garçon et de son chien, nous avons défini des identifiants pour chaque protagoniste du modèle de faits. L'utilisation d'identifiants est assez importante en RDF, et ne se fait évidemment pas grâce à l'attribution de numéros... Les identifiants peuvent en fait être utilisés de deux manières.

La première manière est d'utiliser les identifiants pour identifier des faits entiers, ce qui peut être accompli en ajoutant un attribut `ID` à la balise `<Description>`. Pour identifier ce fait de manière unique dans le monde entier, il suffira alors de le désigner par l'URL du document RDF, suivi du caractère `"#"` et de la valeur de l'attribut `ID`. Le RDF lui-même peut dès lors être considéré comme un ensemble de ressources, chaque ressource étant un seul fait. Voici un exemple d'identifiant RDF :

```
<Description ID="http://www.exemple.com/#printEnabled" ... />
```

La seconde utilisation possible des identifiants en RDF est de remplacer certains littéraux locaux par ceux-ci. Si nous considérons les exemples analysés précédemment, il apparaît beaucoup de faits à propos de Tom. Le corps physique de Tom n'apparaît pas dans tous ces faits - seulement un numéro pour l'identifier, et ce numéro "représente" la personne. Le RDF fournit une méthode d'identification plus élaborée et plus pratique que ces numéros, ce sont les URL. Une URL représente la personne tout aussi efficacement ; elle peut aussi bien être une adresse mail qu'une

URL contenant des informations personnelles. Ainsi, chaque fait dont le sujet est une chaîne de caractères identique à l'URL choisie pour Tom est un fait à propos de Tom. De plus, le sujet d'un fait peut aussi être une URL, comme c'est le cas, dans nos exemples, du chien de Tom et de la balle de Tom.

De façon plus subtile, le prédicat (ou la propriété, si l'on utilise le formalisme de RDF) est lui aussi représenté par une URL. Celle-ci n'est en fait qu'un pointeur vers une ressource décrivant plus complètement la propriété. Pour la lisibilité, l'URL d'un prédicat contient généralement un mot exprimant explicitement son sens général, comme par exemple : `www.exemple.com/#proprietaire`

Les faits peuvent donc, en RDF, être entièrement exprimés grâce à des identifiants formés d'URLs. Les documents RDF sont donc généralement sources de beaucoup de confusions chez les débutants, car l'utilisation de tels fichiers semble nécessiter des interactions directes avec l'Internet. Pourtant ce n'est pas du tout le cas, et ces URL n'ont pas plus de signification que des chaînes de caractères qui contiendraient des adresses de rues. Il n'y a aucune navigation automatique sur le Web et, à moins que le logiciel qui utilise ce RDF ne soit programmé pour effectuer de telles actions, le traitement de fichiers RDF ne requiert en aucun cas de connexion à l'Internet. Malgré tout, il existe une complication possible, qui est qu'une URL utilisée dans un fichier RDF peut éventuellement identifier un fait présent dans un autre fichier RDF, mais de telles mécanismes ne sont généralement pas mis en pratique dans les applications courantes à cause de l'extraordinaire plat de spaghetti de liens que tout cela pourrait créer.

La notion d'espaces de noms

Il a été fait remarquer un peu plus tôt qu'il existait, en XML, un mécanisme très utile pour faciliter l'utilisation des URL dans la syntaxe du RDF. Il s'agit en fait des espaces de noms [VLI05]. Pour expliquer la façon dont ceux-ci fonctionnent, un exemple simple est celui des noms et prénoms. Pour identifier une personne dans un cercle de famille restreint, le prénom suffit généralement ; par contre, pour l'identifier en dehors de ce cadre, il faudra ajouter le nom de famille, qui joue ici le rôle d'*espace de noms*. Cet exemple n'est pas tout à fait exact car le nom et le prénom ne suffisent pas, dans la réalité, à identifier une personne précise, mais dans son principe il illustre bien le problème. Un autre exemple est celui des numéros téléphoniques comme ils étaient utilisés il y a encore quelques années : pour appeler à quelqu'un se situant dans la même zone, il n'y avait pas besoin de former le préfixe, le numéro seul suffisait à identifier le destinataire. Par contre, pour appeler quelqu'un se situant dans une autre zone, il fallait ajouter à ce numéro un indicatif de zone.

Dans le cas du XML, il s'agit de désigner les noms d'éléments et d'attributs dans le contexte d'un document donné. En XML, il est recommandé d'utiliser, afin d'augmenter la lisibilité d'un document, des noms issus du langage naturel. Ainsi, il sera très fréquent de retrouver des noms tels que "name" ou "nom", "title" ou "titre", "description", etc. dans des vocabulaires XML différents ; or des attributs et éléments nommés de la même façon peuvent avoir des fonctions et des contenus tout à fait différents.

Pour pouvoir comprendre la signification d'un élément ou d'un attribut, il est donc nécessaire de pouvoir spécifier à quel vocabulaire il appartient ; c'est l'objectif de la recommandation W3C "Namespaces In XML" [BHLT04]. De plus, cette recommandation s'est donné comme contrainte supplémentaire de permettre l'utilisation simultanée d'éléments et d'attributs provenant de plusieurs espaces de noms dans un même document XML, ce qui complique encore le problème ; en effet, il ne s'agit pas, dès lors, d'identifier un "type de document" dans l'entête du document XML, mais bien de donner la possibilité d'identifier, élément par élément et attribut par attribut, le vocabulaire auquel il appartient.

Puisque, dans la vision du W3C, XML est avant tout destiné à un usage orienté Web, les espaces de noms sont assimilables à des "ressources" Web, ce qui les a poussé à choisir, pour les identifier, d'utiliser les identificateurs standards, les URI[2]. Il est donc possible d'utiliser, pour identifier un espace de noms, un type d'URI nommé URN[3], tel que "`urn:oasis:xmlns:xml:catalog`", qui est par définition un nom, ce qui n'inciterait dès lors personne à essayer d'accéder à une page Web ayant cette adresse.

Malheureusement, une confusion naît du fait qu'il est aussi possible, pour identifier un espace de nom, d'utiliser une URL (les URLs sont, comme les URNs, un sous-ensemble des URIs) telle que "`http://www.w3.org/1999/xhtml`" ; cette dernière identifie non seulement l'espace de noms "XHTML", mais c'est aussi l'adresse

[2]D'après *Wikipédia* (`http://fr.wikipedia.org`, le 12 avril 2006) "une URI, de l'anglais *Uniform Resource Identifier*, soit littéralement identifiant uniforme de ressource, est un protocole mis en place pour le *World Wide Web* qui normalise la syntaxe de courtes chaînes de caractères désignant un nom ou une adresse d'une ressource, physique ou abstraite" Les URLs et les URNs sont tous deux des sous-ensembles des URIs.

[3]D'après *Wikipédia* (`http://fr.wikipedia.org`, le 12 avril 2006) : "*Uniform Resource Name*, traduit littéralement de l'anglais par « nom uniforme de ressource », est le nom d'un standard informatique dans le domaine de l'Internet qui concerne principalement le *World Wide Web*. Il donne une syntaxe de chaîne de caractères utilisable pour identifier une ressource (un document, une image, un enregistrement sonore, etc.) globalement, durant toute son existence, indépendamment de sa localisation ou de son accessibilité par Internet. Par exemple `urn:ietf:rfc:2141` est un URN identifiant le `RFC 2141`."

d'une page Web. Cette pratique est comparable à l'utilisation d'une adresse *e-mail* ou d'un numéro de téléphone pour identifier une personne, ce qui ne signifie pas que la personne identifiée ainsi est elle-même une adresse *e-mail* ! C'est donc, finalement, la chaîne de caractères représentant l'URI qui identifie l'espace de noms et non l'URI en tant que telle, ce qui dissocie totalement l'espace de noms identifié par une URL de la ressource dont c'est l'adresse. Il n'est donc pas nécessaire, pour qu'un espace de noms identifié par "`http://www.w3.org/1999/xhtml`" puisse être défini, que la page "`http://www.w3.org/1999/xhtml`" existe, et, si c'était le cas, celle-ci pourrait contenir absolument n'importe quoi, et n'avoir éventuellement aucun rapport avec l'espace de noms.

La syntaxe des espaces de noms

Deux mécanismes ont été prévu par la recommandation du *W3C* concernant les espaces de noms : les espaces de noms par défaut et les définitions de préfixes. Ceux-ci permettent d'éviter de répéter l'URI identifiant un espace de noms à propos de chaque élément et de chaque attribut, ce qui rendrait le XML illisible.

Les espaces de noms par défaut Ceux-ci constituent l'utilisation la plus simple qui peut être faite des espaces de noms : on définit un espace de noms pour le document, et chaque élément est associé à celui-ci, sauf indication contraire. Par exemple, en XHTML, on trouvera des choses du genre :

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
      <title>Ma page</title>
      .../...
  </head>
.../...
</html>
```

L'attribut "`xmlns`" permet de déclarer l'espace de noms par défaut ; il s'applique à l'élément dans lequel il est situé, ainsi qu'à tous ses descendants. Il est possible de redéfinir un espace de noms par défaut dans l'un des descendants ; il sera valable pour celui-ci et ses descendants.

Les définitions de préfixes Ce mécanisme est particulièrement utile en RDF, où plusieurs espaces de noms différents sont fréquemment utilisés dans un même document, souvent dans des éléments imbriqués, ce qui rendrait très verbeuse l'utilisation d'espaces de noms par défaut. Prenons un simple exemple de fichier RDF [DEA06] :

```
<?xml version="1.0" encoding="iso-8859-1"?>

<RDF:RDF xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:weather="http://www.xulplanet.com/rdf/weather#">

  <RDF:Description
       RDF:about="http://www.xulplanet.com/rdf/weather/city/Paris">
    <weather:name>Paris</weather:name>
    <weather:prediction>Ensoleillé</weather:prediction>
  </RDF:Description>

</RDF:RDF>
```

Les déclarations de préfixes sont :

```
xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:weather="http://www.xulplanet.com/rdf/weather#"
```

L'utilisation de l'élément "`http://www.xulplanet.com/rdf/weather#name`" sera, dès lors, exactement équivalente à l'utilisation d'un élément nommé "`weather:name`", car le préfixe "`weather`" a précédemment été associé à l'URI "`http://www.xulplanet.com/rdf/weather#`".

3.3.3 Les balises à la loupe

Nous allons maintenant introduire une synthèse des différentes balises utilisées dans la syntaxe XML du RDF, ainsi que leurs significations.

La balise <RDF>

La balise `<RDF>` est obligatoire et contient le document RDF entier. Il n'y a pas d'attributs qui soient spécifiques à cette balise. Les seules choses que l'on retrouve généralement dans celle-ci sont les préfixes des espaces de noms.

Les enfants de la balise `<RDF>` sont un ensemble de balises, qui sont obligatoirement des balises `<Description>` ou des balises de conteneurs, à savoir `<Seq>`, `<Bag>`, ou `<Alt>`.

La balise <Description>

La balise `<Description>` est un peu le coeur du RDF, car c'est elle qui sert à représenter les faits. Elle peut contenir zéro ou plusieurs enfants, chacun de ses enfants représentant un prédicat (ou une propriété, dans le formalisme RDF).

Chacun de ces enfants implique donc bien sûr l'existence d'un fait complet, dont le sujet est la valeur de l'attribut "about" de l'élément <Description> père, et l'objet est le contenu de l'enfant <propriété>.

```
<RDF:Description about=sujet>
  <propriété1 ...>objet1</propriété1>
  <propriété2 ...>objet2</propriété2>
</RDF:Description>
```

Dans cet exemple, "sujet" représente une URI. Il est possible, en RDF, de ne pas spécifier d'attribut "about" dans un élément <Description> [BM04] ; dans ce cas, nous parlerons de "description anonyme". Ce type de description est rarement utilisée telle quelle, et on le retrouvera plus fréquemment en lieu et place de l'objet du prédicat.

```
<RDF:Description about=sujet>
  <propriété1 ...>
    <RDF:Description>
      <propriété2 ...>
        ...
      </propriété2>
    </RDF:Description>
  </propriété1>
</RDF:Description>
```

Ainsi, dans cet exemple, le sujet identifié par la valeur "sujet" de l'attribut "about" est caractérisé par une certaine "propriété1", dont l'objet n'a pas d'identifiant mais est lui-même caractérisé par une certaine "propriété2".

Cette balise <Description> peut aussi contenir un attribut "ID" qui permet référencer de façon univoque le fait qu'elle décrit, comme il l'a déjà été fait remarquer plus tôt (section 3.3.2). Il est important de remarquer que, cet attribut ayant pour but de n'identifier qu'un seul fait, il ne devra être placé que dans des éléments <Description> ne contenant qu'une seule propriété, et ne décrivant de cette manière qu'un seul fait ; pour les éléments contenant plusieurs propriétés, il sera possible, comme nous le verrons dans le point suivant, de placer cet attribut "ID" directement dans la balise de propriété.

Les balises de propriétés

Les balises de propriété doivent être créées par l'utilisateur du RDF ; par contre, certains attributs pour ces balises sont prédéfinis dans RDF ; il s'agit des attributs "ID" et "parseType".

Comme nous l'avons fait remarquer dans le point précédent, l'attribut "ID" sert à identifier de manière unique un fait ; une description peut contenir plusieurs propriétés et ainsi décrire plusieurs faits, et il est donc nécessaire d'identifier ceux-ci en plaçant l'attribut "ID" directement dans les balises de propriétés.

L'attribut "parseType" permet de spécifier de quelle manière la chaîne de caractères contenue entre les balises d'une propriété doit être interprétée ; ainsi, si la valeur de l'attribut est "Literal", cela signifie que la valeur est une chaîne de caractères arbitraire. Et si la valeur de l'attribut est "Resource", cela signifie que la valeur est une URI.

Malgré tout, cette syntaxe est un peu verbeuse, et il lui sera généralement préféré son équivalent tiré de la syntaxe abrégée. Pour signaler que la valeur est une URI, on écrira donc :

```
<propriété rdf:resource=objet/>
```

Généralement, une chaîne de caractères arbitraire ne sera, elle, signalée par aucun attribut ; la chaîne de caractères contenue entre les balises d'une propriété sera donc, par défaut, considérée comme telle.

Nous avons maintenant suffisamment d'éléments pour écrire, en véritable syntaxe RDF, notre exemple du garçon, du chien et de la balle ; il existe évidemment plusieurs manières de l'écrire en RDF, en voici donc une parmi d'autres :

```
<?xml version="1.0" encoding="iso-8859-1"?>

<RDF:RDF xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ex="http://www.exemple.com/#">

  <RDF:Description RDF:about="ex:Tom">
    <ex:prenom>Tom</ex:prenom>
    <ex:proprietaire resource="ex:Medor"/>
    <ex:joue-avec resource="ex:balle"/>
  </RDF:Description>

  <RDF:Description RDF:about="ex:Medor">
     <ex:prenom>Medor</ex:prenom>
     <ex:espece>chien</ex:espece>
     <ex:joue-avec resource="ex:balle"/>
  </RDF:Description>
```

```
  <RDF:Description RDF:about="ex:balle">
    <ex:type>tennis</ex:type>
    <ex:couleur>vert</ex:couleur>
  </RDF:Description>
</RDF:RDF>
```

Les conteneurs

Il peut arriver qu'il soit nécessaire, par exemple, de préciser qu'un travail a été créé par plusieurs personnes, ou de créer une liste des étudiants inscrits à un cours, etc. Afin de faire référence à une collection de ressources, RDF offre différents types de conteneurs ; ceux-ci sont utilisés pour contenir ces listes de ressources ou de littéraux.

Un conteneur consiste en une balise `<Bag>`, `<Seq>`, ou `<Alt>`, chacune définissant un type différent de conteneur [LS99]. Le conteneur et son contenu forment ce que l'on appelle une *collection* ; celle-ci peut être placée là où se place normalement l'objet dans un fait, mais peut être aussi utilisée comme un fait indépendant. Les différents types de conteneurs ont des caractéristiques et des significations différentes.

Bag La balise `<Bag>` contient une liste non ordonnée de ressources ou de littéraux. Elle est utilisée pour déclarer qu'une propriété possède plusieurs valeurs qui n'ont pas d'ordre particulier, l'ordre dans lequel elles apparaissent n'a donc pas de signification. Il est, de plus, permis qu'un *Bag* contienne plusieurs valeurs identiques. Exemple d'utilisation d'un *Bag* :

```
<Description about="www.exemple.com/#Tom">
  <ns:proprietaire>
    <Bag ID="Chiens">
      <li>Medor</li>
      <li>Fido</li>
      <li>Mirza</li>
    </Bag>
  </ns:proprietaire>
</Description>
```

Sequence La balise `<Seq>` a les mêmes propriétés que `<Bag>`, mais cette fois, l'ordre d'apparition des valeurs a un sens. Il sera ainsi possible de conserver un

ordre alphabétique. De plus, comme le *Bag*, la *Sequence* permet la duplication de valeurs identiques. Exemple d'utilisation d'une *Sequence* :

```
<Seq ID="JSArticlesParDate">
  <li resource="http://www.dogworld.com/Aug96.doc" />
  <li resource="http://www.webnuts.net/Jan97.html" />
  <li resource="http://www.carchat.com/Sept97.html" />
</Seq>
```

Alternative La balise `<Alt>` contient une liste de ressources ou de littéraux qui représentent des alternatives pour la valeur (cette fois unique) d'une propriété. Il sera possible d'utiliser l'*Alternative* pour, par exemple, fournir les traductions du titre d'un article dans différentes langues. Une application utilisant une propriété dont la valeur est une *Alternative* saura qu'elle peut choisir une des valeurs contenues dans la liste. Exemple d'utilisation d'une *Alternative* :

```
<Alt ID="SitesMiroirs">
  <li resource="ftp://ftp.x.org"/>
  <li resource="ftp://ftp.cs.purdue.edu"/>
  <li resource="ftp://ftp.eu.net"/>
</Alt>
```

3.4 Quelques applications du RDF

Outre l'utilisation du RDF telle que décrite dans cet ouvrage, il existe beaucoup d' applications possibles de ce modèle de données. La base même de RDF, sa raison d'exister est de doter le Web d'une infrastructure de base permettant le transfert et le traitement des (méta)données [WOR99]. Tout un chacun peut, aujourd'hui, cataloguer une ressource Web en un langage compréhensible par les machines. Le RDF pourra donc servir à la découverte de ressources, ce qui permettrait d'améliorer de façon conséquente l'efficacité des moteurs de recherche. Ce langage peut servir à l'évaluation de contenu, en décrivant des ensembles de pages qui représentent un seul et unique "document", pour décrire les droits sur les propriétés intellectuelles des pages Web, pour indiquer les préférences de confidentialité d'un utilisateur ou les politiques de confidentialité des sites Web [LS99]...

Chapitre 4

Le *template engine*

Comme nous l'avons observé dans l'introduction de cet ouvrage, le but final du projet était de créer un langage de *templates*, ainsi que d'implémenter l'*engine* qui permet l'interprétation de ce langage, et de cette façon la génération automatique de XML.

Nous avons maintenant, grâce au chapitre 2 à propos de Mercury, un aperçu du langage de programmation qui a servi à l'implémentation de l'*engine* ; nous savons aussi sous quelle forme se présentent les données sur lesquelles cet *engine* doit raisonner, grâce au chapitre précédent à propos du RDF. Il reste maintenant à comprendre ce qu'est réellement un *template* - le concept sera expliqué dans la section qui suit -, et de quelle manière ce mécanisme s'intègre dans le langage que nous avons défini. Comme il l'a été mentionné dans le chapitre d'introduction, ce dernier se base sur un langage existant défini par Mozilla, le XUL, qui comporte quelques défauts considérables. Pour cette raison, la deuxième partie de ce chapitre présentera de façon assez détaillée notre nouveau langage, en s'appuyant de temps à autre sur une analyse du XUL et de certains de ses inconvénients, afin de justifier quelques-uns de nos choix.

4.1 Le concept de *template*

4.1.1 Requêtes RDF et unification

Le système de *templates* que nous allons présenter s'avère être en fait un système de règles, dans lequel chacune de celles-ci associe un *gabarit* (sous forme de XML paramétrisé - dont nous étudierons ses spécificités au point suivant (4.1.2)) à une *requête*. Il permet de rechercher parmi un ensemble de données celles qui correspondent à ce qui est spécifié dans les critères de recherche, et de générer

du code XML contenant ces données. Dans notre cas, les données que nous allons utiliser sont des faits RDF ; une requête du type *template* est donc un processus spécifique au RDF.

Les requêtes effectuées dans un *template* sont décrites sous la forme d'un *pattern-matching*. Elles procèdent donc à ce que l'on appelle de l'unification ; il s'agit là de la combinaison d'un ensemble d'éléments de données dans un résultat final. Dans notre cas, ces éléments de données sont les sujets, les prédicats et les objets d'un ensemble de faits RDF. Toute combinaison de ces éléments satisfaisant à la spécification de la requête - qui précise la "forme" que doit prendre cette combinaison - correspond à un résultat, constitué d'un nouveau tuple.

Ce système est comparable à celui du SELECT en SQL, du moins lorsque celui-ci contient une jointure (une clause FROM portant sur plus d'une table). Les noms de colonnes renvoyés dans une telle requête ne correspondent pas à une seule table, mais sont tirés de toutes les tables de la jointure.

Pour commencer notre observation des *templates*, mieux vaut sans doute éviter de commencer par leur syntaxe. Reprenons plutôt l'exemple du petit garçon, de son chien et de sa balle, introduit au chapitre 3 concernant le RDF. Celui-ci était utilisé pour décrire un système, en-dehors de toute syntaxe XML de RDF.

```
<- 1, est-nommé, Tom ->
<- 1, propriétaire, 2 ->
<- 1, joue-avec, 5 ->
<- 2, est-nommé, Médor ->
<- 2, chien-de, 1 ->
<- 2, joue-avec, 5 ->
<- 5, type, tennis ->
<- 5, couleur, green ->
```

L'information modélisée est "Tom et son chien Médor jouent avec une balle de tennis verte", et chaque protagoniste est identifié par un numéro. Nous pouvons effectuer des requêtes concernant cet ensemble de faits en utilisant une requête de fait unique ou de faits multiples.

Requêtes de fait unique

Les faits que nous avons rencontrés jusque maintenant sont tous fermés ; chaque partie de ceux-ci étaient des données littérales, ne contenant aucune inconnue. Un fait contenant une inconnue n'est pas très intéressant en tant que donnée, mais il peut constituer un point de départ pour notre système de requêtes sur les faits.

```
<- 1, propriétaire, ?IDchien ->
```

Les variables des requêtes du type *template* sont des littéraux commençant par le caractère " ?". Ici, la requête signifie : "Existe-t-il une ou plusieurs valeurs qui, mises à la place de la variable " `?IDchien`", ferait correspondre ce tuple à un tuple de notre base de connaissances ?". Lorsqu'une requête se réalise, l'unification entraîne ainsi la "fermeture" des variables, c'est-à-dire que chacune de celles-ci se retrouve associée à une ou plusieurs valeurs connues. Dans le cas de ce simple exemple, le second fait de la modélisation d'information à propos du garçon et de son chien fait correspondre la requête avec :

```
<- 1, propriétaire, 2 ->
```

Si l'on associait la variable " `?IDchien`" à la valeur 2, un fait correspondant à un fait réel serait alors construit ; la variable " `?IDchien`" peut donc être liée à la valeur 2.

Supposons que Tom a un second chien, appelé Fido. Ce nouveau fait peut donc être ajouté à notre exemple :

```
<- 1, propriétaire, 3 ->
<- 3, est-nommé, Fido ->
```

Et notre requête contenant " `?IDchien`" pourrait donc correspondre à deux faits différents :

```
<- 1, propriétaire, 2 ->
<- 1, propriétaire, 3 ->
```

La variable " `?IDchien`" peut être liée à 2 ou à 3, il existe donc deux solutions.

Un fait contenant une variable peut, comme nous l'avons vu, servir de requête ; celui-ci pourrait tout aussi bien contenir deux variables :

```
<- ?IDpersonne, propriétaire, ?IDchien ->
```

Dans le système de *templates*, il n'est par contre pas permis de remplacer le terme prédicat par une variable.

La requête précédente requiert une combinaison de valeurs qui satisfasse les deux variables à la fois. Si nous utilisons encore la même source de données que précédemment, il existe alors deux solutions à notre nouvelle requête : " `?IDpersonne`" lié à 1 et " `?IDchien`" lié à 2, ce qui satisfait un premier fait, et " `?IDpersonne`" lié à 1 et " `?IDchien`" lié à 3, satisfaisant l'autre fait.

Si nous ajoutons maintenant une personne nommée Jeanne (avec un ID de 4), qui partage Fido avec Tom (Médor restant le chien exclusif de Tom), nous pouvons rajouter deux faits supplémentaires :

```
<- 4, est-nommé, Jeanne ->
<- 4, propriétaire, 3 ->
```

Il résulte, avec les nouvelles informations acquises, qu'il existe maintenant trois solutions à notre dernière requête :

- " `?IDpersonne`" lié à 1 et " `?IDchien`" lié à 2
- " `?IDpersonne`" lié à 1 et " `?IDchien`" lié à 3
- " `?IDpersonne`" lié à 4 et " `?IDpersonne`" lié à 3

Nous dirons, dans ce cas, que l'ensemble des variables (noté comme un tuple) `<- ?IDpersonne, ?IDchien ->` est associé aux ensembles de valeurs :

```
<- 1, 2 ->
<- 1, 3 ->
<- 4, 3 ->
```

Bien que " `?IDpersonne`" puisse être lié à 4 et " `?IDchien`" lié à 2, aucune solution ne résulte de cette combinaison puisqu'il n'existe pas de fait correspondant dans l'information détenue.

En bref, une requête de fait unique tente d'associer un fait contenant des variables avec un fait réel, en liant ces variables à des termes fermés contenus dans les faits réels ; les combinaisons de valeurs récoltées de cette manière constituent les solutions.

Requêtes de faits multiples

Comme son nom l'indique, une requête de faits multiples permet de combiner deux ou plusieurs faits contenant des variables dans une même requête. Le niveau de déduction que permet le *template engine* ressemble, si l'on excepte la syntaxe, à du Prolog vraiment simpliste (semblable au DataLog [ULL05]).

Si nous tentons de savoir, en utilisant toujours notre exemple précédent, le nom du chien dont Tom est le propriétaire, la requête pourrait être constituée de trois faits qu'il faudra unifier :

```
<- ?IDpersonne, est-nommé, Tom ->
<- ?IDpersonne, propriétaire, ?IDchien ->
<- ?IDchien, est-nommé, ?NOMchien ->
```

Les requêtes de faits multiples doivent souvent, comme c'est le cas ici, introduire des variables dont l'unification ne présente pas d'intérêt direct, mais qui sont nécessaires pour trouver des faits réels correspondants à la requête. En particulier, cette requête a été construite en partant des éléments déjà identifiés ("`Tom`") et des éléments dont on souhaite connaître la valeur ("`?NOMchien`"). En examinant ensuite l'information dont nous disposons, il est possible de trouver quelle forme donner à notre requête pour que celle-ci contienne ces éléments, et soit susceptible de correspondre à des faits réels ; nous pouvons remarquer qu'à cette fin, il est nécessaire d'ajouter des tuples, et donc de nouvelles variables, afin de créer une "connexion" entre les éléments dont nous disposons au départ. Dans le cas qui nous occupe, il a donc été nécessaire de considérer les variables "`?IDchien`" et "`?IDpersonne`".

La tâche du *template engine* sera ici de découvrir les valeurs qui peuvent être attribuées aux variables de la requête et qui satisfassent tous les triples simultanément. Dans notre cas, si nous ignorons l'existence de "Fido", la seule solution à la requête est :

```
<- 1, est-nommé, Tom ->
<- 1, propriétaire, 2 ->
<- 2, est-nommé, Médor ->
```

En utilisant la notation usuelle, nous pouvons écrire que cette solution unifie l'ensemble de variables `<-?IDpersonne, ?IDchien, ?NOMchien->` à un unique ensemble de valeurs `<-1,2,Médor->`. La valeur "Médor" est la valeur recherchée ; les autres ne sont là que pour lier les faits ensemble. Cette requête montre bien pourquoi ce que les variables subissent est appelé une "unification" : toutes doivent correspondre simultanément à des valeurs avant qu'une solution ne soit trouvée.

Nous verrons dans les sections 4.3.1 et 4.3.2 comment ces deux types de requêtes s'intègrent dans le langage de *templates* que nous avons défini. Les valeurs associées aux variables de ces requêtes doivent bien sûr, ensuite, être utilisée afin de servir le but même du *template engine* : la génération de XML. Ceci se fait grâce au mécanisme de "gabarits".

4.1.2 Les *gabarits*

Un *gabarit* représente en réalité le code que nous souhaitons voir généré ; comme nous l'avons déjà fait remarquer précédemment, un gabarit est toujours associé à une requête, et forme avec celle-ci une *règle*. Ainsi, pour chaque solution de la requête (chaque ensemble de valeurs trouvé grâce à l'unification des variables),

un "fragment" de code XML sera répété - ce fragment de code est ce que nous appelons le gabarit. Comme dans n'importe quel code XML, les éléments qui se situent dans ce bout de code sont associés à des attributs, et leur contenu peut éventuellement être composée de valeurs littérales :

```
<élément attribut=...>
  Sous-élément ou valeurs littérales
</élément>
```

Nous pouvons, afin de créer un gabarit, remplacer les valeurs des attributs ou les valeurs littérales par des noms de variables que nous avons utilisées dans la requête associée. Définissons par exemple un gabarit de la forme :

```
<personne prenom=?NOMpersonne>
  Propriétaire de ?NOMchien
</personne>
```

Supposons maintenant que la requête qui lui est associée (une requête est toujours, dans une règle, placée avant le gabarit) ait provoqué une unification de l'ensemble de variables `<-?IDpersonne, ?NOMpersonne, ?IDchien, ?NOMchien->` avec les ensembles de valeurs suivants :

```
<- 1, Tom, 2, Médor ->
<- 1, Tom, 3, Fido ->
<- 4, Jeanne, 3, Fido ->
```

La combinaison de la requête qui nous a permis de trouver les trois solutions avec le gabarit précédent aura pour effet de répéter trois fois ce gabarit, en remplaçant chaque fois les variables par leurs valeurs respectives. Le code généré serait donc :

```
<personne prenom=Tom>
  Propriétaire de Médor
</personne>
<personne prenom=Tom>
  Propriétaire de Fido
</personne>
<personne prenom=Jeanne>
  Propriétaire de Fido
</personne>
```

Nous avons maintenant parcouru les principes généraux du système de *templates* ; voyons maintenant plus en détail la façon dont ces mécanismes peuvent être utilisés dans le langage de *templates*, ainsi que les techniques plus spécifiques qu'il est possible d'utiliser.

4.2 Le langage de *templates*

4.2.1 L'élément *template*

La spécification de requêtes et de gabarits s'effectuent à l'intérieur de balises `<template>` ; celles-ci peuvent en réalité contenir plusieurs règles (plusieurs paires requête - gabarit) qui seront considérées, comme nous le verrons plus tard, selon l'ordre de leur apparition. La concrétisation de ces règles se fait grâce à la balise `<rule>`. Un *template* est donc de la forme :

```
...
<template>
  <rule>
    ... ici se trouve l'intérieur de la règle (requête/gabarit) ...
  </rule>
    ... zero ou plusieurs règles peuvent s'intégrer ici ...
</template>
...
```

Il existe un attribut spécifique à la balise `<template>` : il s'agit de l'attribut `"comparenode"`. L'utilisation de celui-ci sera expliquée un peu plus tard (voir le point 4.4.1). Citons de plus l'attribut `"datasources"` qui permet de spécifier l'emplacement de la source RDF utilisée par le *template* ; toutefois, cet attribut ne doit pas obligatoirement être placé au sein de la balise `<template>`, mais peut très bien se trouver dans n'importe quel élément "père" de cette dernière. Il en va de même pour l'attribut `"ref"`, dont nous examinerons le fonctionnement au point 4.3.4.

4.3 La structure d'une règle

Une règle, dans sa forme basique, est composée de trois éléments : l'élément *conditions*, l'élément *bindings*, et l'élément *action*. Les deux premiers servent en réalité à décrire la requête à effectuer afin de récolter les données souhaitées ; l'élément *action* sert, de son côté, à contenir le gabarit que nous voudrions voir généré pour chaque solution récoltée par la requête. Notons aussi que l'élément *bindings* n'est pas obligatoire, une règle pouvant donc très bien n'être composé que des éléments *conditions* et *action*. Nous examinerons le fonctionnement de cet élément *bindings* après avoir introduit celui des conditions.

4.3.1 L'élément *conditions*

L'élément *conditions* - dans la suite, nous appellerons aussi plus simplement cet élément la "condition" - est le premier élément fils apparaissant entre des balises `<rule>`. Celui-ci contient toujours un ou plusieurs éléments *triple*. Ces derniers sont de la forme suivante :

```
<triple subject=...
        predicate=...
        object=.../>
```

L'ensemble des triples de la condition permet d'exprimer une requête ; si cet ensemble ne contient qu'un seul triple, cette requête sera une requête de fait unique. Dans le cas contraire, il s'agira d'une requête de faits multiples. Le sujet et l'objet de ces éléments *triple* peuvent donc, comme nous l'avons vu au point 4.1.1, prendre éventuellement comme valeur le nom d'une variable. Le prédicat, par contre, doit obligatoirement avoir une valeur fixe.

Prenons en exemple un modeste élément *conditions* pour illustrer nos propos, où nous nous intéresserons cette fois à des prédictions météorologiques :

```
<conditions>
  <content uri=""/>
  <triple subject="?ville"
          predicate="http://www.exemple.com/rdf/meteo#nom"
          object="?nom"/>
  <triple subject="?ville"
          predicate="http://www.exemple.com/rdf/meteo#prediction"
          object="?pred"/>
</conditions>
```

Laissons de côté, pour le moment, l'élément *content* - qui est un élément obligatoire de la condition, et dont l'utilisation sera expliquée à la section 4.3.4 - pour nous intéresser aux triples définis ici. Il est donc assez clair, au vu de ces conditions, que nous cherchons à connaître les villes auxquelles correspondent un nom, et auxquelles sont associées une prédiction météorologique. Ici, tous les sujets et objets sont des variables, mais nous aurions tout aussi bien pu décider de ne récupérer que les villes dont les prédictions météorologiques donnent un temps ensoleillé, et dès lors remplacer la variable " ?pred" par le littéral "ensoleillé" (si c'est ainsi qu'est définie une prédiction ensoleillée dans la source RDF), ou encore de fixer le nom de la ville (variable " ?nom") à "Paris", etc.

Supposons maintenant que la source de données RDF associée à l'élément *template* contenant la condition que nous venons de définir soit celle-ci :

```
<?xml version="1.0" encoding="iso-8859-1"?>

<RDF:RDF xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:meteo="http://www.exemple.com/rdf/meteo#">

  <RDF:Description
       RDF:about="http://www.exemple.com/meteo/#Paris">
    <meteo:nom>Paris</meteo:nom>
    <meteo:prediction>ensoleillé</meteo:prediction>
  </RDF:Description>

</RDF:RDF>
```

Cette source ne contient qu'une description, qui nous fournit certaines propriétés de la ville de Paris : son nom et une prédiction météorologique. Une seule solution sera donc rapportée par l'exécution de la condition précédemment définie. Cette solution consistera à associer les variables `<- ?ville, ?nom, ?pred ->` à l'unique ensemble de valeurs :

```
<- "http://www.exemple.com/meteo/#Paris", "Paris", "ensoleillé" ->
```

Remarquons que les termes sont identifiés ici en utilisant les espaces de noms complets ; il est cependant possible de remplacer ceux-ci par l'abréviation qui leur est associée dans le fichier RDF ; cette possibilité d'utiliser les identifiants RDF courts permet d'éviter de fastidieuses opérations d'écriture, ainsi qu'un allègement certain du code XML.

4.3.2 L'élément *bindings*

Il serait légitime, à ce moment de la lecture, de poser la question de l'utilité de cet élément *bindings*, puisqu'à première vue, la condition semble, à elle seule, suffire à exprimer les requêtes telles que nous les avons présentées à la section 4.1.1. Ainsi, dans notre exemple précédent, les triples placés entre les balises `<conditions>` exprimaient une requête de faits multiples ; ils devaient être nécessairement tous satisfaits simultanément par un ensemble de valeurs pour que celui-ci constitue une solution. Un élément *conditions* exprime donc une conjonction de conditions qu'il est *obligatoire* de satisfaire.

Supposons un instant que la source de données RDF ait été la suivante :

```
<?xml version="1.0" encoding="iso-8859-1"?>

<RDF:RDF xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:meteo="http://www.exemple.com/meteo/#">

  <RDF:Description
       RDF:about="http://www.exemple.com/meteo/#Kiev">
    <meteo:nom>Kiev</meteo:nom>
  </RDF:Description>
</RDF:RDF>
```

Dans ce cas, l'utilisation d'une telle source de données avec les conditions que nous avons définies dans l'exemple de la section précédente aurait eu pour conséquence l'absence pure et simple de solution.

Mais il est possible que nous voulions connaître les différentes villes qui existent dans la source RDF, et de ne recevoir les prédictions correspondant à chacune de celles-ci que dans le cas où de telles prédictions existent. C'est là que l'élément *bindings* intervient ; celui-ci est très similaire à l'élément *conditions*, mais ce qu'il exprime n'est pas une exigence. Les triples qu'il contient ne doivent pas nécessairement être satisfaits simultanément à ceux que contiennent les conditions.

Gardons un exemple similaire au précédent pour mieux comprendre ce fonctionnement :

```
<conditions>
  <content uri=""/>
  <triple subject="?ville"
          predicate="http://www.exemple.com/meteo/#nom"
          object="?nom"/>
</conditions>

<bindings>
  <binding subject="?ville"
          predicate="http://www.exemple.com/meteo/#prediction"
          object="?pred"/>
</bindings>
```

Comme nous pouvons le constater, l'élément *bindings* contient lui aussi des triples, mais ceux-ci ne se trouvent plus, comme c'est le cas pour les conditions, dans une balise `<triple>` mais bien dans une balise `<binding>`. Si nous considérons à nouveau la source de données RDF du point 4.3.1 - celle contenant la ville de Paris -, la solution obtenue avec la paire conditions - *bindings* définie ici est la même que celle obtenue si les *bindings* avaient été des conditions, à savoir que les variables `<- ?ville, ?nom, ?pred ->` seraient liées à l'ensemble de valeurs :
`<- "http://www.exemple.com/meteo/#Paris", "Paris", "ensoleillé" ->`
Avec notre nouvelle source de données - celle contenant la ville de Kiev -, la solution générée associe la variable " `?ville`" au littéral "`Kiev`", et la variable " `?pred`" au littéral vide "".

4.3.3 L'élément *action*

Entre les balises `<action>` d'une règle se situe ce que nous avons appelé plus tôt le "gabarit". L'action contient donc, en réalité, du "véritable" code XML, dans lequel il se peut que certaines valeurs - valeurs d'attributs, littéraux - portent les noms des variables qui ont été utilisées auparavant dans les éléments *conditions* et *bindings*. Pour chacune des solutions en notre possession, le code XML va être répété, en remplaçant à chaque fois les variables rencontrées par les valeurs qui leurs sont associées dans la solution considérée. Prenons par exemple l'action suivante :

```
<action>
  <information>
    La ville de ?nom connaîtra un temps ?pred demain.
  </information>
</action>
```

Supposons maintenant que notre source de données RDF soit un peu plus fournie que ne l'étaient les deux exemples qui ont précédé, et que la résolution des conditions et *bindings* nous ait donné trois solutions correspondant aux variables <- ?ville, ?nom, ?pred -> :

```
<- "http://www.exemple.com/meteo/#Paris", "Paris", "ensoleillé" ->
<- "http://www.exemple.com/meteo/#Londres", "Londres", "pluvieux" ->
<- "http://www.exemple.com/meteo/#Namur", "Namur", "" ->
```

Le code XML généré par le *template engine* serait dès lors :

```
<information>
  La ville de Paris connaîtra un temps ensoleillé demain.
</information>
```

```
<information>
  La ville de Londres connaîtra un temps pluvieux demain.
</information>
<information>
  La ville de Namur connaîtra un temps  demain.
</information>
```

Remarquons que l'ordre dans lequel apparaissent les solutions générées par l'*engine* ne correspond pas à celui de la source RDF de départ ; ceci est dû à l'utilisation de structures de données Mercury appelées "*maps*" qui ont plusieurs avantages, sauf celui de conserver l'ordre dans lequel les éléments y sont insérés.

4.3.4 La référence

La source de données RDF est la représentation XML d'un graphe orienté. La recherche de solutions se fait, dans les exemples présentés jusqu'ici, dans l'entièreté de ce graphe ; il est cependant possible de fixer un noeud de "départ" - appelé la référence - dans ce graphe. Ceci se fait grâce à l'attribut `"ref"`. La valeur de celui-ci est celle d'un sujet ou d'un objet de la source de données. Comme nous l'avons déjà fait remarquer, cet attribut peut être placé soit dans la balise `<template>`, soit dans l'un de ses éléments pères.

Grâce à l'attribut `"ref"`, nous venons donc d'apprendre une façon de fixer le noeud de départ pour la recherche dans le graphe RDF ; il faut maintenant avoir le moyen de préciser à quelle variable de nos conditions correspond ce noeud. C'est ici qu'intervient l'élément *content* que nous avions ignoré précédemment (section 4.3.1). La balise `<content>` a et ne peut avoir qu'un seul attribut. Il s'agit de l'attribut `"uri"` ; la valeur de celui-ci permettra de préciser la variable à laquelle le noeud spécifié dans l'attribut `"ref"` devra correspondre. Prenons l'exemple suivant :

```
<template ref="http://www.exemple.com/meteo/#Paris"
          datasource="previsionsmeteo.rdf">
  <rule>
    <conditions>
      <content uri="?ville"/>
      <triple subject="?ville"
            predicate="http://www.exemple.com/meteo/#prediction"
            object="?pred"/>
    </conditions>
    <action>
      ...
    </action>
  </rule>
</template>
```

Ici, nous avons donc décidé de fixer la valeur de la variable "`?ville`" au noeud "`http://www.exemple.com/meteo/#Paris`" de la source RDF. La recherche dans la source de données RDF est donc limité de cette façon par la contrainte de valeur sur l'une des variables. Il est tout à fait légitime de poser la question de l'utilité de ce mécanisme, puisque le même effet pourrait être obtenu en remplaçant simplement la variable par sa valeur directement dans la condition, comme nous pourrions le faire avec l'exemple précédent :

```
...
  <conditions>
    <content uri=""/>
    <triple subject="http://www.exemple.com/meteo/#Paris"
            predicate="http://www.exemple.com/meteo/#prediction"
            object="?pred"/>
  </conditions>
...
```

Outre le fait de rendre l'écriture moins lourde, puisqu'il nous dispense d'écrire des URLs parfois "kilométriques" (l'utilisation des espaces de noms abrégés permet déjà un allègement des identifiants), l'attribut "`ref`" trouve une réelle utilité lors de l'utilisation des *templates* de façon récursive - ceci sera vu plus en détail au point 4.6.

Remarquons enfin que le langage XUL - le langage de *templates* développé par MOZILLA - rend obligatoire la définition d'une référence et de la variable qui lui correspond dans chaque élément *template* ; il ne permet donc pas, comme c'est le cas ici, de prendre en compte la totalité du graphe RDF lorsque aucun noeud de départ n'est précisé.

L'attribut "`uri`" est ici fixé, lorsque nous ne désirons pas définir de noeud de référence, au littéral vide "" (le même effet pourrait être obtenu en lui fournissant comme valeur le nom d'une variable n'apparaissant pas dans la condition). Cette syntaxe n'est, de toute évidence, pas très élégante, et pourrait facilement être améliorée en rendant facultative la présence de l'élément *content* au sein de la balise `<conditions>` d'une règle.

4.4 Plusieurs règles dans un *template*

Nous n'avons examiné, jusqu'à présent, que des *templates* ne mettant en oeuvre qu'une seule règle (balise `<rule>`). Il est cependant possible d'y intégrer plusieurs règles ; nous allons maintenant étudier la raison de cela.

Lorsque plusieurs règles sont intégrées dans un *template*, le *template engine* les considère séquentiellement, selon leur ordre d'apparition. Elles n'ont, à part une source de données commune, aucun rapport entre elles, et le résultat de l'exécution d'un *template* de ce type sera - sauf dans le cas que nous examinerons ci-après de l'utilisation de l'attribut "comparenode" - équivalent au résultat fourni par une suite de *templates* contenant chacun une seule de ces règles. Un nouvel exemple permettra de comprendre comment utiliser ces règles multiples :

```
<template ...>
  <rule>
    <conditions>
      <content uri=.../>
      <triple subject="?ville"
            predicate="meteo:nom"
            object="?nom"/>
      <triple subject="?ville"
            predicate="meteo:prediction"
            object="?pred"/>
    </conditions>
    ...
  </rule>
  <rule>
    <conditions>
      <content uri=.../>
      <triple subject="?ville"
            predicate="meteo:nom"
            object="?nom"/>
    </conditions>
    ...
  </rule>
</template>
```

L'effet du premier élément *conditions* sera de rechercher, dans la source de données, toutes les villes comportant des noms et auxquelles des prévisions météorologiques sont associées. L'effet des conditions de la seconde règle sera, cette fois, de rechercher l'ensemble des villes auxquelles des noms sont associés - logiquement, il s'agira cette fois de la totalité des villes contenues dans la source RDF.

Ceci n'est évidemment pas d'une utilité extraordinaire, et ce que nous souhaiterions pouvoir faire est de sélectionner la règle à appliquer selon les propriétés qui sont attribuées aux données. Dans le cas de l'exemple précédent, il serait par

exemple judicieux d'avoir la possibilité de faire en sorte que la seconde condition ne sélectionne que les villes n'ayant pas déjà été sélectionnées dans la première règle. Cela permettrait notamment d'intégrer dans la première règle un élément *action* contenant :

```
<information>
  La ville de ?nom connaîtra un temps ?pred demain.
</information>
```

et de mettre entre les balises `<action>` de la seconde règle :

```
<information>
  Aucune prévision météorologique n'est associée à ?nom.
</information>
```

Une telle utilisation des *templates* à règles multiples sera rendue possible, dans notre langage, par l'attribut `"comparenode"`. Avant d'examiner celui-ci, nous tenterons d'abord de comprendre la méthode utilisée dans le langage de *templates* de MOZILLA, afin de mettre en évidence ses défauts et de comprendre ainsi ce qui a motivé la création de notre attribut `"comparenode"` et de ses spécificités.

4.4.1 L'attribut `"comparenode"`

La solution de MOZILLA et ses défauts

Dans le langage de *templates* de MOZILLA, le mécanisme permettant d'utiliser les règles multiples de la manière dont il vient d'être fait cas, consiste en l'ajout d'un attribut `"uri"` dans n'importe quel élément fils de l'élément *action*. Cet attribut a une sémantique quelque peu nébuleuse, et son utilisation entraîne plusieurs conséquences, pas toujours prévisibles.

La signification première de cet attribut `"uri"` est de donner un "noeud de comparaison", permettant de sélectionner la règle à appliquer selon les données considérées. Ce "noeud de comparaison" est nécessairement une variable ; le fait que celle-ci apparaisse en tant que valeur de l'attribut `"uri"` signifie que si les conditions d'une règle génèrent une solution associant cette variable à une certaine valeur, aucune autre solution liant cette variable à cette même valeur ne pourra être générée, dans cette règle ou dans les règles suivantes. Dans notre exemple habituel, cela s'intégrerait de cette manière :

```
...
<rule>
  <conditions>
    <content uri=""/>
    <triple subject="?ville"
            predicate="http://www.exemple.com/rdf/meteo#nom"
            object="?nom"/>
    <triple subject="?ville"
            predicate="http://www.exemple.com/rdf/meteo#prediction"
            object="?pred"/>
  </conditions>
  <action>
    <information uri=?ville>
      La ville de ?nom connaîtra un temps ?pred demain.
    </information>
  </action>
</rule>
<rule>
  <conditions>
    <content uri=""/>
    <triple subject="?ville"
            predicate="http://www.exemple.com/rdf/meteo#nom"
            object="?nom"/>
  </conditions>
  <action>
    <information uri=?ville>
      Aucune prévision météorologique n'est associée à ?nom.
    </information>
  </action>
</rule>
...
```

La première remarque à faire ici est qu'il serait, en XUL, interdit de choisir la variable "`?nom`" comme valeur de l'attribut "`uri`", car, contrairement à la variable "`?ville`", celle-ci ne sera pas associée à des URIs dans les solutions générées, mais plutôt à des littéraux tels que "`Paris`" ou "`Kiev`". Cette constatation a d'ailleurs fait l'objet d'un rapport de *bug* auprès de MOZILLA[1], puisque les tests effectués sur ce cas avaient pour conséquence le *crash* de l'application. La correction de cette erreur a eu pour effet de remplacer ce *crash* par l'absence pure et simple d'*output* lors d'une pareille utilisation de l'attribut "`uri`".

[1] `https://bugzilla.mozilla.org/show_bug.cgi?id=321091`

La seconde remarque importante à faire ici est que l'attribut "`uri`" de la seconde règle n'a strictement aucun effet sur l'exécution du *template* ; celui-ci est tout bonnement ignoré, et c'est la variable définie comme dans l'attribut "`uri`" de la première règle du *template* qui sera considérée comme "noeud de comparaison" pour l'ensemble des règles. Si chacune des variables "`?ville`" de la deuxième règle était renommée en "`?city`", les deux règles redeviendraient alors indépendantes, puisque le "noeud de comparaison" serait fixé à "`?ville`" - or cette variable n'apparaîtrait plus dans la seconde règle.

Il existe une seconde signification de cet attribut "`uri`", assez peu utile d'ailleurs ; l'élément dans lequel il est placé indique, en réalité, la partie de l'action qui doit être répétée pour chaque solution collectée par les conditions (et les *bindings*). Ce n'est donc pas le "gabarit" entier qui sera répété, mais seulement une partie de celui-ci. Les balises situées à l'intérieur de l'élément *action* et se situant autour de l'élément contenant l'attribut "`uri`" ne seront pas répétées. Ce mécanisme s'avère assez inutile, puisque le même effet pourrait être obtenu en plaçant ces balises non-répétées autour de l'élément *template* lui-même.

Nous pouvons finalement constater que la sémantique de cet attribut "`uri`" n'est pas très "logique", en tout cas très peu intuitive. C'est cette raison qui nous a poussé à définir un nouveau dispositif, sous la forme d'un nouvel attribut nommé *comparenode*.

L'utilisation de "`comparenode`"

La première constatation étant que l'attribut "`uri`" avait, dans MOZILLA, une portée s'étendant à l'entièreté des règles du *template*, il a été décidé de placer cet attribut "`comparenode`" directement dans l'élément *template* concerné. Sa signification est la suivante : la variable définie comme valeur de l'attribut "`comparenode`" devra prendre une valeur différente dans chaque solution générée par le *template* auquel cet attribut est associé. Cela a bien sûr pour conséquence que si la variable définie dans le "`comparenode`" - appelons-la "variable de comparaison" - est instanciée à une valeur X dans une règle du *template*, elle ne pourra plus être instanciée à cette même valeur dans les solutions des règles suivantes. De plus, contrairement à la solution proposée par MOZILLA, l'emploi d'une variable de comparaison susceptible d'être instanciée à une valeur littérale ne pose aucun problème.

Revenons à l'exemple précédent, auquel nous appliquons notre nouvel attribut :

```
<template comparenode=?ville
          datasource="previsionsmeteo.rdf">
  <rule>
    <conditions>
      <content uri=""/>
      <triple subject="?ville"
              predicate="meteo:nom"
              object="?nom"/>
      <triple subject="?ville"
              predicate="meteo:prediction"
              object="?pred"/>
    </conditions>
    <action>
      <information>
        La ville de ?nom connaîtra un temps ?pred demain.
      </information>
    </action>
  </rule>
  <rule>
    <conditions>
      <content uri=""/>
      <triple subject="?ville"
              predicate="meteo:nom"
              object="?nom"/>
    </conditions>
    <action>
      <information>
        Aucune prévision météorologique n'est associée à ?nom.
      </information>
    </action>
  </rule>
</template>
```

Nous pourrions utiliser "`?nom`" comme variable de comparaison, malgré que celle-ci sera normalement associée à des valeurs littérales ; cela aurait le même effet qu'avec la variable "`?ville`" dans le cas où il existerait une relation bijective entre l'identifiant et le nom d'une ville. Ceci n'étant pas toujours le cas, mieux vaut choisir "`?ville`" comme variable de comparaison, puisque, dans le cas contraire, deux villes portant le même nom ne génèreraient qu'une solution.

Le code XML résultant de l'utilisation du template précédent, si nous ne considérons que deux villes - la ville de Paris, que l'on prévoit ensoleillée, et la ville de Kiev, à laquelle aucune prévision n'est associée (ceci reviendrait à fusionner la source RDF définie section 4.3.1 avec celle de la section 4.3.2 - sera :

```
<information>
  La ville de Paris connaîtra un temps ensoleillé demain.
</information>
<information>
  Aucune prévision météorologique n'est associée à Kiev.
</information>
```

4.5 Les *templates* imbriqués

Il est possible d'intégrer un élément *template* à l'intérieur de l'action d'une règle. Cet élément peut de toute évidence contenir plusieurs attributs ; en particulier, il sera possible de lui associer de nouveaux attributs `"datasources"` - ce qui permet donc de changer localement de source de données -, `"ref"` et `"comparenode"`.

Le point intéressant à relever dans le cas d'un *template* imbriqué est que les variables utilisées dans celui-ci, et qui ont déjà été instanciées dans la *condition* correspondant à l'action dans laquelle il apparaît, prennent en toute logique les valeurs auxquelles elles ont été précédemment associées. Ceci permet de faire totalement dépendre les solutions générées par ce *template* imbriqué de la solution des conditions de la règle "mère". La valeur de l'attribut `"ref"`, par exemple, peut être fixée à celle d'une des variables définies dans la condition. Pour mieux comprendre tout l'intérêt de ce mécanisme, prenons un nouvel exemple qui permettra de montrer son utilité de manière explicite :

```
<template datasources="familleSmith.rdf" comparenode="?papa">
   <rule>
      <conditions>
         <content uri=""/>
         <triple subject="?papa"
                 predicate="famille:nom"
                 object="?nom"/>
      </conditions>
```

```
        <action>
           <homme nom="?nom">
              <template ref="?papa">
                <rule>
                    <conditions>
                      <content uri="?papa2"/>
                      <triple subject="?papa2"
                              predicate="famille:fils"
                              object="?fils2"/>
                      <triple subject="?fils2"
                              predicate="famille:nom"
                              object="?nom2"/>
                     </conditions>
                     <action>
                        <papa-de nom="?nom2"/>
                     </action>
                 </rule>
              </template>
            </homme>
          ...
  </template>
```

Supposons que la source de données associée à cet exemple contienne tous les hommes d'une famille nommée "Smith". Il existe trois générations de personnes : un grand-père (John Senior), ses trois fils (John Junior 1,2 et 3) et ses deux petits fils (John Junior Junior 1 est le fils de John Junior 1, et John Junior Junior 2 est le fils de John Junior 2) :

```
<RDF:RDF xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:famille="http://www.famillesmith.com/#">

  <RDF:Description RDF:about="GrandPère_Smith">
    <famille:nom>John Senior</famille:nom>
    <famille:fils RDF:resource="Père_1_Smith"/>
    <famille:fils RDF:resource="Père_2_Smith"/>
    <famille:fils RDF:resource="Père_3_Smith"/>
   </RDF:Description>

  <RDF:Description RDF:about="Père_1_Smith">
    <famille:nom>John Junior 1</famille:nom>
    <famille:fils RDF:resource="Fils_1_Smith"/>
   </RDF:Description>
```

```
   <RDF:Description RDF:about="Père_2_Smith">
    <famille:nom>John Junior 2</famille:nom>
    <famille:fils RDF:resource="Fils_2_Smith"/>
   </RDF:Description>

   <RDF:Description RDF:about="Père_3_Smith">
    <famille:nom>John Junior 3</famille:nom>
   </RDF:Description>

   <RDF:Description RDF:about="Fils_1_Smith">
    <famille:nom>John Junior Junior 1</famille:nom>
   </RDF:Description>

   <RDF:Description RDF:about="Fils_2_Smith">
    <famille:nom>John Junior Junior 2</famille:nom>
   </RDF:Description>
</RDF:RDF>
```

Le *template* précédent permet, à partir cette source de données, de générer le code XML suivant :

```
<homme nom="John Senior">
  <papa-de nom="John Junior 1"/>
  <papa-de nom="John Junior 2"/>
  <papa-de nom="John Junior 3"/>
</homme>
<homme nom="John Junior 1">
  <papa-de nom="John Junior Junior 1"/>
</homme>
<homme nom="John Junior 2">
  <papa-de nom="John Junior Junior 2"/>
</homme>
<homme nom="John Junior 3"/>
<homme nom="John Junior Junior 1"/>
<homme nom="John Junior Junior 2"/>
```

Les hommes n'ayant pas de fils - le troisième fils ainsi que les deux petits-fils - représentent des solutions qui, placées comme valeur de l'attribut "`ref`" du *template* imbriqué, ont comme conséquence de provoquer une absence de solutions aux conditions de ce *template* ; les éléments correspondants à ces personnes ne contiennent donc aucun sous-élément.

Il apparaît de toute évidence qu'avec les *templates* imbriqués, le nombre de niveaux de sous-éléments dans le code généré ne dépassera jamais le nombre de niveaux d'imbrication des *templates* ; c'est là l'intérêt des *templates* récursifs présentés dans la section suivante.

4.6 Les *templates* récursifs

Le système de *templates* récursifs permet d'appliquer, dans l'action d'une règle, le *template* dans lequel cette règle est définie - appelons-le le *template* courant. L'utilisation de ce procédé est soumis à plusieurs contraintes, afin d'éviter que le processus ne génère une boucle infinie.

Un *template* faisant usage de la récursivité devra donc obligatoirement utiliser le système de référence ; il faut, de ce fait, que la condition de la règle faisant appel au *template* courant contienne un attribut `"uri"` dont la valeur est une des variables utilisées dans la requête. Mais même en utilisant le système de référence, le *template* courant ne peut pas être appliqué tel quel dans l'action d'une des règles ; il faut en réalité que le noeud de départ soit redéfini dans l'appel à ce *template* courant.

Le *template engine* de MOZILLA effectue de manière tout à fait implicite cet appel ; cela est fait automatiquement, et il est impossible de l'en empêcher, ce qui a pour effet de générer des résultats parfois aussi inattendus qu'indésirables. Le raisonnement de l'application de MOZILLA - ou plutôt de ses concepteurs - est le suivant : si dans une solution découverte par les conditions - appelons cette solution la "solution mère" -, l'une des valeurs composant cette solution est susceptible de remplacer la référence et de trouver ainsi de nouvelles solutions, alors ces dernières sont recherchées. Elles servent ensuite à générer du code (grâce au gabarit contenu dans l'action) qui sera placé à l'intérieur même du code généré grâce à la solution "mère". L'emplacement exact de ce code "fils" est souvent difficile - voir impossible - à déterminer à l'avance.

Pour éclaircir un peu ce dernier point, il serait sans doute judicieux de procéder à son illustration par des exemples graphiques. Supposons que nous ayons, dans notre source de données RDF, plusieurs éléments liés comme ceux du graphe de la figure 4.1. La référence que nous définissons pour notre *template* est celle indiquée sur le graphe, à savoir l'élément `A`. La requête, elle, contient deux triples, semblables à ceux représentés à la figure 4.2.

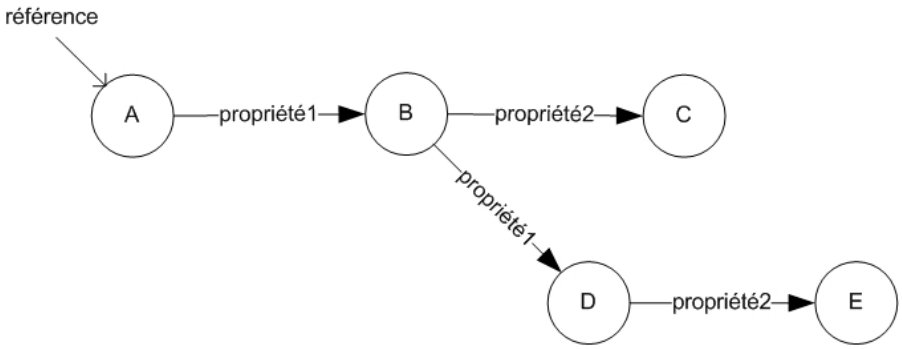

FIG. 4.1 – Source de données destinées à faire l'objet d'une requête

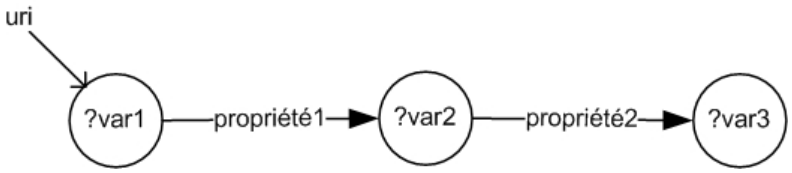

FIG. 4.2 – Représentation graphique d'une requête

Cette requête devrait nous permettre d'effectuer une seule unification, dont la solution associerait les variables `<- ?var1, ?var2, ?var3 ->` à l'ensemble de valeurs `<- A, B, C ->`. Le système de MOZILLA ajoutera à cela une sorte de "sous-solution" de cette première solution qui sera l'ensemble `<- B, D, E ->`. Imaginons que le gabarit situé dans l'action est le suivant :

```
<element apropos=?var1>
  <propriete valeur=?var3/>
</element>
```

Alors le code XML généré par le *template engine* de MOZILLA ressemblera sans doute (sans certitude...) à :

```
<element apropos=A>
  <propriete valeur=C/>
  <element apropos=B>
    <propriete valeur=E/>
  </element>
</element>
```

La sous-solution acquise lors de la requête a donc servi à créer un sous-élément dans celui correspondant à la solution "mère". Ce caractère implicite et incontrôlable de la méthode proposée par MOZILLA n'aide évidemment en rien lorsqu'il s'agit de comprendre son fonctionnement afin de l'utiliser convenablement...

Pour éviter cela, le langage de *templates* que nous avons défini ne s'aventure pas à accomplir des manoeuvres qui n'ont pas été explicitement demandées. Ainsi, pour utiliser un *template* de manière récursive, il est impératif d'introduire dans l'action une balise `<apply-rules/>` à l'endroit où l'on souhaite voir apparaître le résultat. De plus, il est obligatoire de définir, dans cette balise, le noeud qui servira de nouvelle référence lors de l'appel récursif. Pour obtenir le même résultat que celui obtenu avec le système de MOZILLA, il faudra que l'action contienne un gabarit comme celui-ci :

```
<element apropos=?var1>
  <propriete valeur=?var3/>
  <apply-rules ref=?var2/>
</element>
```

La solution de notre requête définie graphiquement associe l'ensemble des variables `<- ?var1, ?var2, ?var3 ->` à l'ensemble de valeurs `<- A, B, C ->`. La référence lors de l'appel récursif du *template*, dans notre exemple, sera donc bien `B` ; la nouvelle solution générée associera les variables `<- ?var1, ?var2, ?var3 ->` à `<- B, D, E ->`, ce qui permettra la génération du code correspondant à l'endroit où se situe la balise `<apply-rules/>`. Comme l'appel est récursif, le *template engine* tentera ensuite d'effectuer la même opération, avec, cette fois, la valeur `D` comme référence. Ce cas-là ne donnant aucune solution, le résultat final sera, comme attendu, celui-ci :

```
<element apropos=A>
  <propriete valeur=C/>
  <element apropos=B>
    <propriete valeur=E/>
  </element>
</element>
```

Nous pouvons donc constater que le résultat de l'appel récursif au *template* courant se trouve effectivement là où nous nous y attendions, c'est-à-dire là où se situait, dans notre action, la balise `<apply-rules/>`.

Supposons que le gabarit que nous avons utilisé ait été celui ci (le même que le précédent mais dans lequel la balise `<apply-rules/>` a été déplacée) :

```
<element apropos=?var1>
  <apply-rules ref=?var2/>
  <propriete valeur=?var3/>
</element>
```

Alors le code XML généré aurait été :

```
<element apropos=A>
  <element apropos=B>
    <propriete valeur=E/>
  </element>
  <propriete valeur=C/>
</element>
```

Ce dernier exemple est strictement impossible à reproduire avec le langage de *templates* de MOZILLA, car ce dernier ne permet pas de choisir l'endroit où est placé le code généré par l'appel récursif au *template* courant.

Pour terminer, nous pouvons aussi remarquer qu'outre la référence, il est aussi possible de redéfinir dans l'élément "*apply-rules*" les attributs *datasources* et *comparenode*. L'appel récursif au *template* courant pourra donc se faire en utilisant une nouvelle source de données, ou un nouveau noeud de comparaison. Ce mécanisme n'a toutefois pas, outre le défi que constituait son implémentation, d'intérêt flagrant pour l'utilisation qui sera généralement faite des *templates*.

4.7 Défauts et améliorations

S'il est évident, après avoir examiné brièvement le langage de *templates* de MOZILLA, que notre solution s'avère être intuitivement plus compréhensible, elle n'est cependant pas encore idéale. En effet, la syntaxe des règles est encore souvent très lourde, et ce pour effectuer des requêtes qui ne sont pourtant pas fort complexes. Pour alléger l'écriture de code dans notre langage, et rendre ce dernier plus riche, plusieurs mécanismes sont envisageables.

Tout d'abord, il serait bon de donner la possibilité d'effectuer un "`not`" devant un triple situé dans une requête. Reprenons notre exemple des villes et des prédictions météorologiques ; pour ne sélectionner que les villes auxquelles ne sont pas associées de prédiction, il est nécessaire d'écrire un *template* à règles multiples utilisant un attribut "`comparenode`" tel que nous en avons défini un à la section 4.4.1. Il est donc nécessaire d'écrire une règle sélectionnant toutes les villes auxquelles sont associées des prédictions, afin d'effectuer un "filtre", et d'ainsi trouver, grâce au *comparenode*, les villes n'ayant pas satisfait à cette première requête. Il serait pourtant beaucoup plus simple d'écrire quelque chose de semblable à :

```
...
  <rule>
    <conditions>
      <content uri=""/>
      <triple subject="?ville"
              predicate="meteo:nom"
              object="?nom"/>
      <NOT subject="?ville"
           predicate="meteo:prediction"
           object="?pred"/>
    </conditions>
    <action>
      ...
    </action>
  </rule>
...
```

De cette façon, il serait possible de collecter les données de la source RDF qui satisfassent les éléments *triple* et qui ne satisfassent pas les éléments *NOT* de cette requête.

De plus, comme nous l'avons compris après l'étude du mécanisme de *templates*, il existe un *AND* implicite entre chaque triple composant une condition ; peut-être serait-il utile de mettre en place un mécanisme permettant d'effectuer des *OR*, et ainsi augmenter fortement l'expressivité des requêtes. Dans un même ordre d'idées, il serait peut-être bon de s'inspirer de la condition "`LIKE`" du langage SQL, et de permettre donc de sélectionner un sujet ou un objet commençant, terminant ou contenant une certaine chaîne de caractères. Nous pourrions éventuellement imaginer une syntaxe ressemblant à :

```
...
  <triple subject="?ville"
          predicate="meteo:nom"
          object="?nom"
          objectLike="P%"/>
...
```

Dans cette perspective, une condition ne contenant que le triple que nous venons de définir permettrait de trouver toutes les solutions associant à la variable "`?nom`" des noms de villes commençant par la lettre "`P`".

Afin d'alléger l'écriture des *templates* au sein du fichier XML qui les accueille, il serait profitable des les rassembler dans un fichier séparé, et d'associer à chacun

d'eux un identifiant. L'appel d'un template donnée pourrait alors se faire très facilement grâce à une simple balise, appelons-là "*apply-template*".

```
... code XML ...
  <apply-template ID=identifiant ref=Référence
                  datasources=Source de données RDF .../>
... code XML ...
```

Ceci permettrait, de plus, de faciliter la lecture du code, particulièrement dans le cas de *templates* imbriqués car, comme nous avons pu le constater plus tôt (section 4.6), de tels *templates* sont rapidement constitués de plusieurs dizaines de lignes, malgré la simplicité des exemples proposés dans ce chapitre.

Chapitre 5

Conclusion

5.1 Bilan

Nous avons à notre disposition, au terme de ce projet, un *engine* codé en Mercury capable d'interpréter tous les types de *templates* que nous avons évoqués au chapitre précédent. Le code de cet engine est en réalité composé de huit modules assez conséquents, contenant en tout 135 prédicats, ce qui représente un peu moins de 4.000 lignes de code (commentaires compris).

Nous ne comptons pas, dans ces chiffres, le module `mexpat` (puisqu'il existait déjà), que nous avons utilisé afin d'effectuer le *parsing* de fichiers XML. Celui-ci permet, à partir d'un élément XML, de créer un élément défini par un type Mercury ; il s'agit en fait d'une structure contenant le nom de l'élément, la liste de ses attributs ainsi que la liste de ses fils. Un fils est représenté par un type *content*, qui peut lui-même être composé d'un élément ou d'un littéral. Le module `mexpat` offre le moyen de manipuler les éléments, d'extraire les fils, les attributs, etc. Il permet aussi d'effectuer la transformation inverse du *parsing* à un élement stocké dans une structure Mercury, c'est-à-dire transformer ce dernier en une chaîne de caractères contenant le code XML de cet élément.

Ce module a servi tout le long du développement du *template engine*, puisque c'est grâce à celui-ci que nous avons pu créer le module de *parsing* du RDF, nommé `rdf_parser` ; les éléments d'un fichier RDF pouvaient alors être stockés dans une nouvelle structure, tenant compte de la forme "`sujet - prédicat - objet`" des faits représentés par ces éléments.

Le fichier de *templates* lui-même nécessitait d'être l'objet d'un *parsing*, afin de pouvoir le mettre sous une forme plus adéquate (une nouvelle structure définie dans Mercury), nous permettant de distinguer chacune des règles, et dans chacune de celles-ci, les éléments *conditions*, *bindings* et *actions*. Cette opération est effectuée dans un module nommé `template_parser`.

Ayant à notre disposition une structure de données extraite du fichier RDF, et ayant, de plus, une structure permettant de manipuler facilement un *template* et les règles qui le composent, il est alors possible de confronter les données RDF aux *conditions* et *bindings* de chacune des règles. C'est là le rôle du module `interpreter`, qui nous permet d'obtenir les ensembles de valeurs qui, associées à l'ensemble de variables, constituent les solutions de la requête.

Toutefois, il est nécessaire, si un *template* contient plusieurs règles, de sélectionner la règle à appliquer avant de satisfaire une requête, particulièrement dans le cas où le *template* utilise un attribut *comparenode* (voir section 4.4.1 du chapitre précédent). Le module `rule_selector` permet de réaliser ceci.

Lorsque nous sommes en possession des solutions des requêtes, la dernière chose à réaliser est de répéter l'action correspondante pour chacune de celles-ci. Le module `constructor` sert à cela, mais sert aussi à la gestion des *templates* imbriqués et des appels récursifs au *template* courant, ce qui le rend particulièrement complexe.

Pour gérer les appels de tous les modules que nous avons cités, il existe deux modules supplémentaires, à savoir `generator` dont le but est de coordonner les appels pour l'interprétation d'un seul *template*, et le module *engine_ core*, qui gère l'entièreté du fichier XML contenant des *templates* et qui renferme le module `"main"`.

S'il n'a pas encore été testé en profondeur, l'*engine* permet tout de même de réaliser un certain nombre de tâches avec une relative facilité, la syntaxe et la sémantique du langage ayant l'avantage d'être assez facilement compréhensibles en comparaison avec les autres solutions existantes. Il existe bien sûr toujours des défauts, tels que ceux que nous avons cités à la fin du chapitre précédent, et une certaine quantité de travail reste encore à fournir si nous souhaitons pouvoir l'utiliser dans des applications commerciales.

5.2 Perspectives

Si l'*engine* a été testé durant et après son développement, il serait tout de même utile d'effectuer des tests dans des cadres d'utilisation différents, de l'employer le plus possible, avec des objectifs divers, afin d'assurer sa fiabilité. Il existe aussi, bien sûr, de nombreuses extensions que nous pourrions imaginer créer afin de compléter l'*engine* ; l'une des plus importantes parmi celles-ci serait la création d'un mode de détection d'erreurs ainsi que d'un mode *verbose* - qui permet d'afficher, en temps réel, le traitement du fichier sur la sortie standard. En effet, comme nous l'avons fait remarquer dans l'introduction, un des gros défauts du XUL de MOZILLA est le manque cruel de moyens offerts pour la détection des erreurs dans le code des *templates*. Cependant, les moyens de *debugger* et de détecter les utilisations incorrectes des *templates* offerts par notre *engine* sont, pour le moment, encore trop élémentaires ; une extension plus étoffée serait donc la bienvenue.

Nous avons mentionné l'existence, au chapitre 3, d'une syntaxe XML du RDF appelée *syntaxe abrégée*, forme plus compacte pour représenter un sous-ensemble du modèle de données ; la syntaxe complète que nous avions présentée était la *syntaxe de sérialisation*. Cette dernière est la seule que supporte le *template engine*. Il serait sans aucun doute intéressant de permettre à ce dernier d'accepter cette syntaxe abrégée, afin de pouvoir utiliser des sources de données RDF hétérogènes.

Plusieurs autres fonctionnalités restent encore à implémenter ; nous avons cité celles-ci dans la dernière section du chapitre 4. Il s'agit donc de donner la possibilité d'exprimer des opérateurs logiques entre triples dans les conditions et les *bindings*, tels que le "*not*" ou le "*or*".

Il s'agit aussi de faciliter l'utilisation des *templates* et d'alléger leur syntaxe, en permettant de placer leur définition dans un fichier séparé ; en attribuant à chacun d'eux un identifiant, il serait alors possible de n'utiliser qu'un élément `<apply_template/>` afin d'effectuer l'appel.

La dernière possibilité que nous avons imaginée est d'emprunter certains mécanismes pratiques au langage de requêtes SQL, tels que la condition LIKE, et d'augmenter ainsi les possibilités du langage de *templates*. Il est bien sûr possible de concevoir encore de nombreuses améliorations, afin de rendre toujours plus efficace et plus aisée l'utilisation du *template engine*.

Bibliographie

[AAB04] Anthony A. AABY. *Introduction to Programming languages.* http ://cs.wwc.edu/ aabyan/PLBook, Juillet 2004.

[ALL02] Jeremy ALLAIRE. *A next-generation rich client.* http ://download.macromedia.com, Mars 2002.

[BEC06] Ralph BECKET. *Mercury tutorial.* http ://www.cs.mu.oz.au/mercury, Avril 2006.

[BHLT04] Tim BRAY, Dave HOLLANDER, Andrew LAYMAN, and Richard TOBIN. *Namespaces in XML 1.1.* W3C Recommendation - http ://www.w3.org, Février 2004.

[BM04] Dave BECKETT and Brian McBRIDE. *RDF/XML Syntax Specification (Revised).* W3C Recommendation - http ://www.w3.org, Février 2004.

[BOR03] Xavier BORDERIE. *Caractéristiques des langages fonctionnels.* http ://developpeur.journaldunet.com, Septembre 2003.

[BSD01] Vaughn BULLARD, Kevin T. SMITH, and Michael C. DACONTA. *Essential XUL Programming.* John Wiley & Sons, Inc., 2001.

[BUR04] Carole BURET. *L'inévitable fusion des architectures client-serveur et web.* http ://www.zdnet.fr, Avril 2004.

[COR] Olivier CORBY. *Ontologie.* http ://www.rangiroa.essi.fr.

[DAV03] Ian DAVIS. *RDF Template Language 1.0.* http ://www.iandavis.com, Septembre 2003.

[DEA06] Neil DEAKIN. *XUL Tutorial.* http ://www.xulplanet.com, Février 2006.

[FOU] Jean-Pierre FOURNIER. *Génie Logiciel : Paradigmes de Programmation.* http ://www.infeig.unige.ch.

[GRU01] Tom GRUBER. *What is an Ontology ?* http ://www-ksl.stanford.edu/, Septembre 2001.

[HAB06] Henri HABRIAS. *Intégrité référentielle.* http ://www.iut-nantes.univ-nantes.fr, Avril 2006.

[HCS+06] Fergus HENDERSON, Thomas CONWAY, Zoltan SOMOGYI, David JEFFERY, Peter SCHACHTE, Simon TAYLOR, Chris SPEIRS, Tyson DOWD, and Ralph BECKET. *The Mercury Language Reference Manual Version 0.12.2.* http ://www.cs.mu.oz.au/mercury, Avril 2006.

[LS99] Ora LASSILA and Ralph R. SWICK. *Resource Description Framework (RDF) Model and Syntax Specification.* W3C Recommendation - http ://www.w3.org, Février 1999.

[McF03] Nigel McFARLANE. *Rapid Application Development with Mozilla.* Prentice Hall PTR, Novembre 2003.

[MvH04] Deborah L. McGUINNESS and Frank van HARMELEN. *OWL Web Ontology Language Overview.* W3C recommendation - http ://www.w3.org, Février 2004.

[ONT] *Ontologie.* http ://websemantique.org/Ontologie.

[PS06] Eric PRUD'HOMMEAUX and Andy SEABORNE. *SPARQL Query Language for RDF.* http ://www.w3.org, Avril 2006.

[RIA] *Web Application Formats Working Group.* http ://www.w3.org.

[SEA04] Andy SEABORNE. *RDQL - A Query Language for RDF.* http ://www.w3.org, January 2004.

[SWM04] Michael K. SMITH, Chris WELTY, and Deborah L. McGUINNESS. *OWL Web Ontology Language Guide.* W3C recommendation - http ://www.w3.org, Février 2004.

[ULL05] Jeffrey ULLMAN. *DataLog.* http ://www.labri.fr/, Decembre 2005.

[VAN] Wim VANHOOF. *Notes du cours INFO2109 : Programmation fonctionnelle et logique.* http ://www.info.fundp.ac.be/~wva/cours/.

[VLI05] Eric VAN DER VLIST. *Introduction aux espaces de noms XML.* http ://xmlfr.org, Septembre 2005.

[WIK03] *Functional programming.* http ://en.wikipedia.org, Août 2003.

[WOR99] Emma PLACE WORSFOLD. *Internet et la collaboration internationale autour des points d'accès par sujet.* 65th IFLA Council and General Conference - http ://www.ifla.org, Août 1999.

[XUL05] *Client Riche.* http ://www.xulfr.org, Décembre 2005.