CONCEPTION D'UN TRADUCTEUR INTELLIGENT DE RTF VERS XML

par

Yi Xu

mémoire présenté au Département d'informatique en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbooke, Québec, Canada, mars 2004

Le 8 septembre 2004,
Date

le jury a accepté le mémoire de M. Yi Xu dans sa version finale.

Membres du jury

M. Richard St-Denis
Directeur
Département d'informatique

M. Richard St-Denis
Directeur
Département d'informatique

M. Marc Frappier
Membre
Département d'informatique

M. Gabriel Girard
Président-rapporteur
Département d'informatique

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-94918-4
Our file Notre référence
ISBN: 0-612-94918-4

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Sommaire

Les fichiers de textes formatés (.doc) produits par l'outil *Microsoft Word* sont omniprésents dans la très grande majorité des ordinateurs. Ils constituent des documents digitaux dont certains sont de nature publique et leurs propriétaires aimeraient bien les publier facilement sur l'*Internet*. Une solution consiste à traduire, par exemple, ces fichiers en des fichiers dans le format HTML. Ce mémoire présente un nouveau système informatique qui permet de convertir un fichier dans le format RTF, un format proche du format .doc mais universel et lisible par un humain, en un fichier dans un format XML. Dans ce mémoire, le format XML est considéré comme un format intermédiaire puisqu'un fichier dans ce format est à son tour utilisé pour générer un fichier dans un format cible comme HTML, JSP, TeX ou D2E. En plus de présenter l'architecture et des éléments de conception de ce système, ce mémoire porte une attention particulière sur des règles de traduction, des règles de simplification et des règles de préférence mises en œuvre grâce à des techniques empruntées au domaine de la construction des compilateurs.

Acknowledgments

First and foremost I would like to express my sincere gratitude to Professor Richard St-Denis for his intellectual ideas, scientific thoughts, generous help and supervision during the preparation and writing of this thesis. Without his patient scrutiny with so many resourceful suggestions and comments, this thesis would never have been possible. I have learned many things from him including how to identify a problem and how to address it. Even during his illness, not only he spent his countless valuable hours to correct different versions of my thesis but also he has given me his patient hearing. Again I wish to express my gratitude toward him.

Thanks to my friend Mr. S.A.M. Matiur Rahman for helping me in correction of English grammar of my thesis.

Thanks are also due to the staff in the département d'informatique for their assistance during my study in Université de Sherbrooke.

Table of Contents

Summary ii

Acknowledgments iii

Table of Contents iv

List of Tables ix

List of Figures x

Introduction 1

Methodology 3

Solution 1: Converting into RTF 3

Solution 2: Extracting RTF File 4

Solution 3: Understanding Corresponding Tags 4

Solution 4: XML Representation 5

Solution 5: Generating New File 5

Contribution 5

Organization 6

Chapter 1 Translator System Structure 7

1.1 System Components 7

1.1.1 Microsoft Word 8

1.1.2 Extractor 9
1.1.3 Intelligent Translator 10
1.1.4 Generator 10
1.2 Data Flow of the System 11
1.3 RTF Syntax 16
1.3.1 Control Words 17
1.3.2 Control Symbols 17
1.3.3 Groups 18
1.3.4 Plain Text 18
1.3.5 A RTF Example: Hello, World! 19
1.4 A Compact Package of Control Words 20
Chapter 2 Overview of the Translator 24
2.1 Text Processing Domain 24
2.1.1 Character Reading 26
2.1.2 Lexical Analysis 26
2.1.3 Syntax Analysis 27
2.1.4 Semantic Analysis 28
2.1.5 Code Optimization 28
2.1.6 Code Generation 29
2.2 Binary Tree Data Structure 29
2.2.1 The Class `TreeNode` 30
2.2.2 Intermediate Nodes 31
2.2.3 Target Nodes 32
2.2.3.1 The Class `TagNode` 33
2.2.3.2 The Class `FieldNode` 34
2.2.3.3 The Class `BookMarkNode` 35
2.2.3.4 The Class `ImageNode` 35
2.2.3.5 The Class `TextNode` 36

2.2.4 The Node Representation 37

2.3 Object-Oriented System 38

2.3.1 Encapsulation 39

2.3.2 Abstraction 39

2.3.3 Inheritance 40

2.3.4 Self-Recursion 41

2.4 Algorithm 41

2.4.1 Translation Rules 42

2.4.2 Simplification Rules 42

2.4.3 Preference Rules 42

Chapter 3 Translation Rules **43**

3.1 Some Associations Between RTF, HTML, and XML 44

3.2 Translation Rules on Fonts and Paragraphs 45

3.2.1 The Font and Paragraph Attribute Table 45

3.2.2 Rules in Implementation Level 47

3.2.3 An Example Translating with the Rules on Fonts 48

3.3 Translation Rules on Headings 53

3.4 Translation Rules on Lists 54

3.4.1 The Compulsory Rules on Lists 56

3.4.2 Translation Rules on Items of a List 60

3.4.3 Nested Lists and Normal Lists 63

3.5 Translation Rules on Images 65

3.6 Translation Rules on Bookmarks 67

3.7 Translation Rules on Fields 68

3.8 Translation Rules on Tables 70

3.8.1 Basic Rules for Tables 70

3.8.2 Translation Rules for Nested Cells 72

Chapter 4 Simplification Rules **76**

4.1 Simplification Rules on Useless Blocks 76
4.2 Simplification Rules on Merging 78
Chapter 5 Code Generation 81
5.1 XML 82
5.1.1 What is XML? 82
5.1.2 The Components of XML 83
5.2 DTD 83
5.2.1 DTD Syntax 84
5.2.1.1 ELEMENT Declarations 84
5.2.1.2 ATTLIST Declarations 85
5.2.1.3 ENTITY Declarations 87
5.2.1.4 NOTATION Declarations 88
5.2.2 The DTD for the *Translator* 89
5.2.2.1 Definition of a Paragraph 90
5.2.2.2 Definition of a Heading 91
5.2.2.3 Definition of an Image 92
5.2.2.4 Definition of a List 92
5.2.2.5 Definition of a Table 93
5.3 Implementation Aspects 93
5.3.1 Identification Tags and an Activating Mechanism 94
5.3.2 Image Generation 96
5.3.3 Table Generation 96
5.3.4 Paragraph Generation 98
5.3.5 Text Generation 99
5.3.6 Heading Generation 101
5.3.7 Field Generation 101
5.3.8 Bookmark Generation 102
5.3.9 List Generation 103

5.3.10 Item Generation 103

5.3.11 Font Generation 104

5.4 Display the Translated XML 105

5.4.1 Root Node Match 108

5.4.2 Child Element Nodes Match 109

5.4.3 Display `XMLoutput.xml` 109

Conclusion **112**

Appendix A C++ Code in the *Translator* **117**

A.1 The Class `Translator.cpp` 117

A.2 The Class `TreeNode.cpp` 139

Appendix B The File `Translator.xsl` **145**

Appendix C The Test Files **150**

C.1 The RTF File `Test.rtf` 151

C.2 The Command File `Test.cmd` 153

C.3 The Text File `Test.txt` 157

C.4 The Translated XML File `XMLoutput.xml` 158

C.5 The Transformed HTML Page 161

Bibliography **163**

List of Tables

1 The translator recognizable RTF control words 22

2 The table storing font and paragraph attributes 44

3 Some associations between RTF, XML, and HTML 45

4 The **attributeTable** with default values 49

5 The **attributeTable** ending in line 1 50

6 The **attributeTable** ending in line 3 51

7 The **attributeTable** ending in line 7 52

8 The syntax of an image 65

9 The syntax of a bookmark 67

10 The syntax of a field 68

11 The syntax of a table row 70

12 Default value in ATTLIST declaration 87

List of Figures

1 The RTF file HelloWorld.rtf 9

2 Data flow of the intelligent translator system 12

3 The input file Dataflow.doc 13

4 The converted RTF file Dataflow.rtf 13

5 Internal text formatting of Dataflow.rtf 14

6 The file Dataflow.cmd to store extracted control words 14

7 The file Dataflow.txt to store extracted texts 15

8 The XML output file Dataflow.xml 15

9 The file Dataflow.html browsed by *Microsoft Internet Explorer* 16

10 A RTF file to show plain text 18

11 RTF control words for HelloWorld.rtf 19

12 The RTF file HelloWorld.rtf 20

13 The command file HelloWorld.cmd 23

14 The function **ConstructTree()** 26

15 A syntax tree constructed by the *translator* program 28

16 The class **TreeNode** 31

17 The class **BlockNode** 32

18 The class **SequenceNode** 32

19 The class **TagNode** 34

20 The class **FieldNode** 34

21 The class **BookMarkNode** 35

22 The class **ImageNode** 36

23 The class **TextNode** 37

24 Part of the syntax tree generated from the file Dataflow.cmd 38

25 Inheritance between nodes 40

26 Three files (RTF, text, and command) for testing translation rules on fonts ... 50

27 The command file testHeading.cmd 53

28 The syntax for the list table 55

29 Values in **\levelnfcN** and the corresponding meaning 57

30 Part of control words for number type of ordered lists 58

31 An ugly list with different aligned items 59

32 The RTF file testItem.rtf 61

33 The command file testItem.cmd 62

34 The text file testItem.txt 62

35 The RTF file testNestedList.rtf 64

36 The text file testNestedList.txt 64

37 The command file testNestedList.cmd 65

38 A bookmark example 67

39 The RTF file testTable.rtf for nested tables 73

40 The command file testTable.cmd for nested tables 74

41 The text file testTable.txt for nested tables 74

42 An example for simplification rules on blocks 78

43 Three files (RTF, command, and text) for testing merging 79

44 DTD for the *translator* 89

45 The function **MakeImage()** 96

46 The function **MakeCell()** 98

47 The function **MakePar()** 99

48 The function **MakeText()** 100

49 The function **MakeHeading()** 101

50 The function **MakeField()** 102

51 The function **MakeBookMark()** 102

52 The function **MakeItem()** 104

53 The function **GetFontList()** 105

54 Options to display XML 106

55 The code for the root node match 108

56 The first heading expression in `XMLoutput.xml` 110

Introduction

Text processing and transformation is a fundamental domain of computer science. It embraces a procedure mainly based on character reading, lexical analysis, syntax analysis, semantic analysis, code optimization, and code generation. Character reading does the actual reading of the source text, which is usually in the form of a stream of characters. Lexical analysis is to collect sequences of characters into meaningful units called tokens. Syntax analysis determines the structural elements of the text and their relationships with a result generally represented as a parse tree. Semantic analysis is to interpret text meaning as opposed to its syntax. Optimization is for purpose of code improvement. Finally, the step for code generation represents the production of the last and useful results. The procedure above is also called translation because it accepts as input a text in a certain language and produces as output a text in another language, while preserving the meaning of that text. A typical example of translation is the compilation of programs written in a high-level programming language into a code machine [1].

One important application in text processing and transformation is the development of format translators. With the rise of the Internet, there has been a pervasive need for file sharing between machines and between applications. However, different kinds of vendor-specific binary file formats prevent a standard to appear. Each applications

program uses its own vendor-specific format to store data. Most of the information about file formats is confidential, not well documented or not available for public use [2]. Besides, PC software evolved rapidly, leading to a proliferation of binary file formats and file format versions [3]. Both become serious problems for users, who found that file sharing was often impractical and that upgrades to application and operating system software could make their older files inaccessible. It results in the current trend toward open, Web-accessible publication formats. Therefore there is a great need for software to convert binary file formats to a universal format. Two file formats are predominant: `.doc` introduced by Microsoft is the most widely used vendor specific binary format [2], and XML (Extensible Markup Language) is being looked at for exchanging content between databases, systems, and users, as a web services protocol for enabling interoperability between systems [4]. The objective of our project is to study how to convert `.doc` files into XML. Because the internal binary format `.doc` is not easily recognizable for human and `.doc` can be transformed by *Microsoft Word* into RTF (Rich Text Format) that is an internal textual formatting, this thesis focuses on the RTF to XML conversion.

There are some conversion tools available on the market, capable of translating from a `.doc`/RTF file to XML format. For example, *RTF2FO XML* converter [5] and *NTS Word/RTF-to-XML* translator [6] are two format converters. However, they may not always satisfy particular requirements in some cases. Besides, in some applications, only part of tags/control words is often used. It is not an efficient way to parse this small number of tags with existing tools which were originally designed for a large package of complicated tags (supporting about 500 tags for a commercial product). Finally, some new format documents are required to be converted into and the current translators and generators cannot make out such results. For these reasons, we are motivated to design an efficient and flexible translator converting between different formats for particular usage. Especially we focus on the design and creation of a tool to translate from RTF to XML. This thesis introduces an intelligent algorithm used in the translator and demonstrates how it works and why it is efficient.

Methodology

The following problems are found during the stage of designing a universal translator converting .doc files to other formats:

1) How to read an internal binary formatting which is not publicly available?

2) How to get the control words from the source document?

3) How to understand the control words?

4) Which is the best candidate for target document?

5) How to generate the target document?

Problem 1 can be solved by converting a `.doc` document into a human readable RTF document that has an internal textual representation. Problem 2 can be solved by extracting content and control words of a RTF document with an extractor. Problem 3 can be solved by building some association relationships between RTF control words and a new file format. This is the main aspect of the translator. We apply techniques and knowledge in text processing and transformation introduced in the previous section to do this job. Problem 4 can be solved by adopting XML to represent translated documents. Problem 5 can be solved by a generator to produce a specific format document taking an XML file as input.

Solution 1: Converting into RTF

Microsoft offers an exchange format of its own, the Rich Text Format (RTF). RTF is the most portable file format used to exchange files across platform while still retaining all or much of its formatting [7]. RTF is a universal file format that pervades practically every PC. Because RTF is a textual representation, it is much easier to generate and process

than binary `.doc` files. The first thing is to convert a `.doc` document into a RTF with a word processor.

Solution 2: Extracting RTF File

A RTF file consists of texts, images, and control words. In our implementation, these elements are extracted and classified into three parts (texts, images, and control words) by an extractor which is already implemented. Every piece of extracted texts is associated to a label for identification when generated into the target document. The texts and images are the fundamental data about a RTF document and will be combined into the target document with the translator. A subset of the RTF file contains RTF control words with references to labels and images. The RTF control words play an important role in forming a document appearance. It is these words that arrange and decide the format of a document. The main problem is on dealing with RTF control words.

Solution 3: Understanding Corresponding Tags

RTF control words are associated with corresponding XML tags according to some association rules. In this stage, the control words will be translated according to the rules. This procedure is actually text processing. A program is coded to read RTF control words letter by letter (by reading a character stream), scan tokens (to group into control words), parse syntax tree (control words stored in a binary tree), and do semantic analysis (to find control words' meaning). Simplification rules are set for code optimization. Since there may exist more than one tags in the target file associated with one RTF control word, a preference rule would be adopted as one of the inputs along with RTF commands. The preference may be decided by users or by default.

Solution 4: XML Representation

XML is a good candidate for representing initial documents. XML has been originally designed for ease of implementation and for interoperability with both SGML and HTML. Nowadays, it is looked as a standardized file format flowing between different information systems. In our case, XML is chosen as the format of the target file of the translator which can translate many kinds of formats with different code generators.

Solution 5: Generating New File

After the corresponding XML result comes out, this result and images are inputted into a code generator and then the generator produces a new file. If the output of the translator is presented with XML, a related Document Object Model and an XSL file will be also added as part of inputs in the generator.

Contribution

This thesis discusses an algorithm for the format translator. This algorithm is based on three rules: translation rules, simplification rules, and preference rules. The translation rules are for translating the RTF control words. The simplification rules are for code optimization. Finally, preference rules are for considering preferences supplied by users.

The translator is extensible. Firstly, the input file could be other files rather than a RTF file as long as they can be divided into command parts and other parts as the RTF format does. Secondly, particularly interesting control words could be easily added in a particular case. Thirdly, the output which is currently an XML file could be changed for other

formats. Two ways could realize this goal. The translator either uses the XML file to convert any preferred file with a format generator or outputs a preferred file directly.

Organization

Besides the introduction, this master thesis consists of six parts. Chapter 1 gives readers a general overview about the whole system. It introduces the main processes and data flow between them. Chapter 2 describes the translator's binary tree data structure, a context-free grammar, the object-oriented character of the translator, and an algorithm which includes three sets of rules. Chapter 3 explains the translation rules in detail. Plenty of examples are supplied to demonstrate how the translation rules work. Also the RTF syntax is outlined in this chapter. Chapter 4 introduces the simplification rules with some samples. Chapter 5 discusses code generation. The syntaxes of XML and XSL are outlined, respectively. Then the DTD (Document Type Definition) of the XML result is described. In addition, a HTML file transformed by the XML result with an XSL file is displayed in the chapter. The conclusion part discusses the advantages and weaknesses of the translator.

Chapter 1

Translator System Structure

1.1 System Components

In this thesis, the term "translator system" refers to a whole system including several components that do translation together. The term "*translator*" on the other hand, refers to a specific program which translates a RTF file into an XML file. There are four components in the system: *Microsoft Word*, *extractor*, *translator*, and *generator*. It is a one-direction system whose data cannot flow back to the other end. Such kind of architecture is one of the most common ways to partition application logic between pairs consisting of a data sender and a data receiver. In the translator system, a sender always works on data for the following one. A receiver always processes the received data and then outputs it as the input or one of inputs to the next component that acts as another data sender. For each pair in the system, the sender is responsible for supplying data and the receiver is responsible for the application and presentation.

Each component in the system is logically independent but each functionally depends on the data supplied by the previous one. From the view-point of a system user, the whole system is a converter that converts a `.doc` document into a document written in a preferred target format. However, each component plays its own role inside the system.

Two components work as translators. *Microsoft Word* is actually a *pure* translator. It converts the source file completely without missing data. On the contrary, the *translator* in the system is an *intelligent* translator. It chooses some useful control words of RTF files and does not consider meaningless data. Its intelligence is also proven by the preference rules with which a user can select an option in the case there are two or more translation choices. If no preference is indicated by the user, a default value is taken. In some cases, some attributes, which are not important enough to affect the translated document apparently, are set by the developer for purpose of simplicity.

The other two components work as an information decomposer and a composer, respectively. Obviously, the *extractor* is such a decomposer that extracts the source file into three kinds of data: texts, images, and control words. The *generator*, which acts as a composer, takes the extracted control words and original texts and images together to generate a text file in the target format.

1.1.1 Microsoft Word

Because `.doc` files are not easily readable and accessible due to their internal binary representation, we have to consider an easier processing format. RTF is such kind of format which is tag based and text formatting. *Microsoft Word* is used for converting a `.doc` file into a RTF file. It is very easy to make a RTF file from a `.doc` file through *Microsoft Word*. The concrete steps are below. Firstly open a `.doc` file with *Microsoft*

Word then click the "File" button and then choose "Save As". Next you may keep the file name as before and select "*.RTF" as "Save as type". You may also simply change the extension of the `.doc` file into the RTF extension directly. The input data for *Microsoft Word* is a `.doc` file and *Microsoft Word* outputs the corresponding RTF file. Then this RTF file will be forwarded to the next component, the *extractor*.

1.1.2 Extractor

Although the internal representation of RFT files is much more human-recognizable than that of `.doc` files, it is still a huge job for human beings. For example Figure 1 shows a RTF document which externally consists of pure letters "**Hello, world!**". It is pretty complex internally from the view-point of human beings.

```
HelloWorld.rtf - Bloc-notes
Fichier  Edition  Format  ?
{\rtf1\ansi\ansicpg1252\uc1 \deff0\deflang1033\deflangfe1036{\fonttbl{\f0\froman\fcharset0\fprq2{\*\panose 02020603050405020304}
Times New Roman;}{\f214\froman\fcharset238\fprq2 Times New Roman CE;}{\f215\froman\fcharset204\fprq2 Times New Roman Cyr;}
{\f217\froman\fcharset161\fprq2 Times New Roman Greek;}{\f218\froman\fcharset162\fprq2 Times New Roman Tur;}{\f219\froman
\fcharset186\fprq2 Times New Roman Baltic;}}{\colortbl;\red0\green0\blue0;\red0\green0\blue255;\red0\green255\blue255;
\red0\green255\blue0;\red255\green0\blue255;\red255\green0\blue0;\red255\green255\blue0;\red255\green255\blue255;\red0\green0\blue128;
\red0\green128\blue128;\red0\green128\blue0;\red128\green0\blue128;\red128\green0\blue0;\red128\green128\blue0;
\red128\green128\blue128;\red192\green192\blue192;}{\stylesheet{\nowidctlpar\widctlpar\adjustright \fs20\cgrid \snext0 Normal;}{\*\cs10
\additive Default Paragraph Font;}}{\info{\title Hello, World}{\author pc018}{\operator pc018}{\creatim\yr2003\mo10\dy21\hr16\min56}
{\revtim\yr2003\mo10\dy21\hr16\min57}{\version1}{\edmins1}{\nofpages1}{\nofwords0}{\nofchars0}{\*\company Universit
\'e9 de Sherbrooke}{\nofcharsws0}{\vern73}}\deftab708\widowctrl\ftnbj\aenddoc\hyphhotz425\formshade\viewkind1\viewscale100\pgbrdrhead
\pgbrdrfoot \fet0\sectd \linex0\headery709\footery709\colsx709\endnhere\sectdefaultcl {\*\pnseclvl1\pnucrm\pnstart1\pnindent720\pnhang{
\pntxta .}}{\*\pnseclvl2
\pnucltr\pnstart1\pnindent720\pnhang{\pntxta .}}{\*\pnseclvl3\pndec\pnstart1\pnindent720\pnhang{\pntxta .}}{\*\pnseclvl4\pnlcltr\pnstart1
\pnindent720\pnhang{\pntxta )}}{\*\pnseclvl5\pndec\pnstart1\pnindent720\pnhang{\pntxtb (}{\pntxta )}}{\*\pnseclvl6
\pnlcltr\pnstart1\pnindent720\pnhang{\pntxtb (}{\pntxta )}}{\*\pnseclvl7\pnlcrm\pnstart1\pnindent720\pnhang{\pntxtb (}{\pntxta )}}{
\*\pnseclvl8\pnlcltr\pnstart1\pnindent720\pnhang{\pntxtb (}{\pntxta )}}{\*\pnseclvl9\pnlcrm\pnstart1\pnindent720\pnhang
{\pntxtb (}{\pntxta )}}\pard\plain \qc\nowidctlpar\widctlpar\adjustright \fs20\cgrid {\b\fs24 Hello, World!}{\fs24
\par }}
Démarrer | KeyA... | Disqu... | extra... | thesis | NSE... | WinEdt | Micro... | Invite ... | data... | datafl... | 16:39
```

Figure 1: The RTF file `HelloWorld.rtf`

There are several reasons that prevent analysis of RTF to become an easy job for human beings. Firstly, there are too many characters, 3 415, in this very simple document that includes no lists, no images, and no tables. Secondly, its internal representation lacks of logic and relationship between tags. Thirdly, most of control words are not easy to

remember and understand. For a regular RTF document, the inner code is much bigger and more complicated.

A RTF document is considered as the composition of three kinds of parts: images, texts, and control words. The *extractor* extracts a RTF document into images, texts, and control tags. Images and texts are kept unchanged for later use. The extracted RTF control words are translated with the *translator* based on some translation rules.

1.1.3 Intelligent Translator

As a general translator in common sense, the *translator* does not create extra information. It forwards the sender's information, the RTF control words, to the receiver in a way in which another side may understand. So, the *translator* collects data from its client and translates this "foreign language" into understandable words for the *generator*. In this sense, the *translator* is a middleman in communication. Actually it is located in the middle place in the system. As mentioned before, it is a producer-consumer pair based system. In the *extractor-translator* pair, the *translator* works as a client. It is the *extractor* who makes a service by supplying extracted RTF control tags to the client. But in the *translator-generator* pair, the *translator* changes its role and works as a server. The *translator* outputs an XML result to the *generator*.

1.1.4 Generator

After obtaining new file tags, the extracted images and texts can be rearranged by means of the *generator*. Control words are responsible for arranging document appearance. The *generator* is the last part in linked components inside of the system. Its role is to create the target format based on the translated XML file and extracted texts as well as images. For purpose of extensibility, we choose XML rather than other data representation to carry

data content. Because XML can be used to describe data, the *generator* can read all required data from the XML document and makes some specific format users prefer. Since the XML document carries style attributes and other information of the source document, the *generator* simply gets the information from the XML document and generates a new kind of file. Certainly, a DTD file and an XSL file are required as companion files to define and represent the XML document.

So far the *generator* is implemented as a HTML format generator for purpose of demonstration. Of course, it could be made as other format generators, D2E [8] or TeX [9] for example. The *generator* could also be an all-in-one powerful generator that is able to generate some format documents according to user's preference. In fact, we can go one step further. The *generator* can be within the system or it can be in other different systems. XML is an extensible, generalized markup language that is capable to describe data passing between different systems or applications. So the *generator* can still safely get information from the translator system without any trouble.

1.2 Data Flow of the System

Figure 2 shows the dataflow of the translator system. The input file for the whole system is in `.doc` format. We use *Microsoft Word* to convert `.doc` format into RTF format. Then the converted `.rtf` file is forwarded into the *extractor* that is already implemented by our research group. The output of the *extractor* is classified as three subsets: images (`.png` or `.jpg`), texts (`.txt`), and control words (`.cmd`). In order to recognize the commands, the RTF control words should be inputted into the intelligent *translator* as well as texts and then it yields XML result. The final stage is in the *generator*. The XML result flows into the *generator*. Also a DTD file, which is used for definition of the XML file, and an `.XSL` file, which is used for presentation of the result, are inputted into the *generator* along with images and the XML file. Because there may exist more than one

translation for one control word, a preference file (`.prf`) is also inputted into the *generator*. Finally, the *generator* outputs the final result-HTML, D2E [8], or TeX [9].

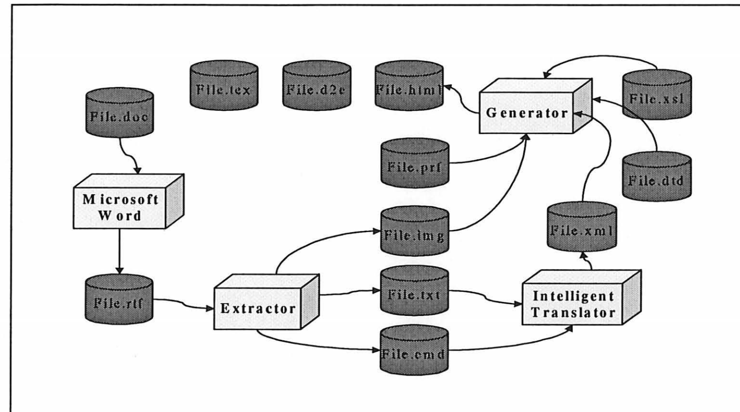


Figure 2: Data flow of the intelligent translator system

To demonstrate the data flow in the translator system, a simple *Microsoft Word* `.doc` document, `Dataflow.doc`, is taken as the input of the system. Assume the *generator* is an *XMLtoHTML* generator which takes XML as input and produces a HTML file as output.

Stage 1: Take Input as `.doc` Format

Figure 3 shows the `Dataflow.doc` input file. Because `.doc` internal code is not public available and is in a binary representation, only its appearance is illustrated. This simple document includes an unordered list, an image, and some paragraphs.

Stage 2: Convert to RTF Format

Next, a `.doc` file flows into *Microsoft Word* where it is converted into a RTF file. Although the RTF file has the same physically appearance as its `.doc` version from the

view of human beings (compare Figure 3 and Figure 4), its internal representation is quite different (see Figure 5) because one is binary and the other is textual. Unlike the binary internal formatting of `.doc`, RTF's textual representation offers an opportunity to analyze the document's "source code" and then build a translator.

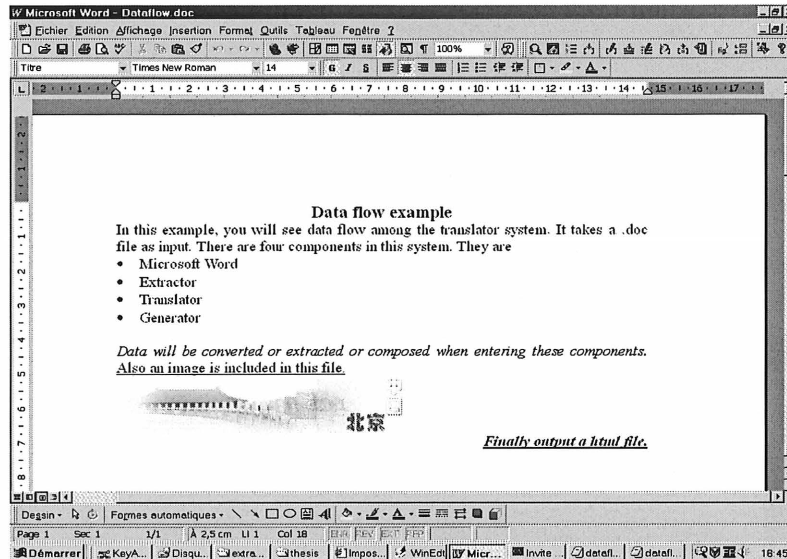


Figure 3: The input file `Dataflow.doc`

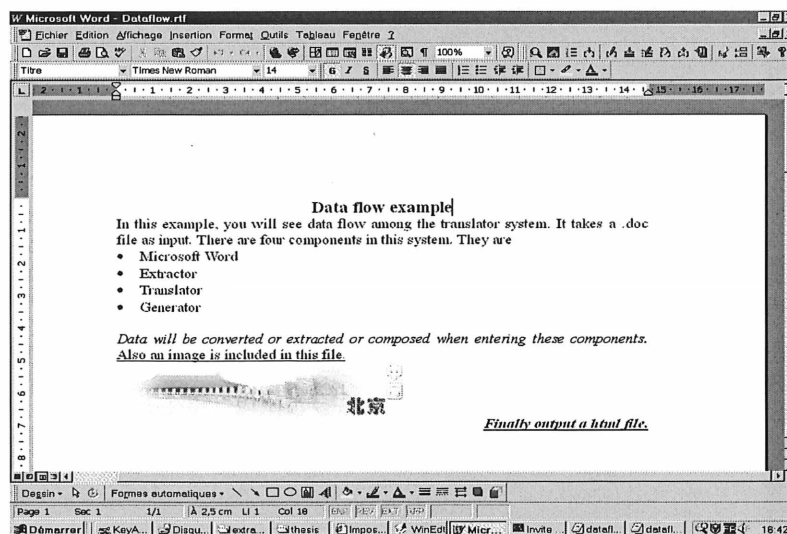


Figure 4: The converted RTF file `Dataflow.rtf`

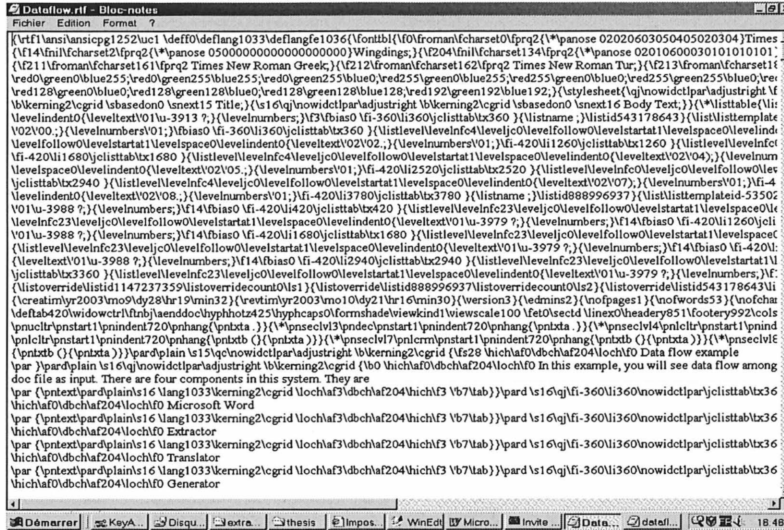


Figure 5: Internal text formatting of `Dataflow.rtf`

Stage 3: Extract Three Parts from the RTF File

The `Dataflow.rtf` file is read by the *extractor* from which information is extracted into three parts: `Dataflow.cmd` shown in Figure 6 to store extracted RTF command words (control words), `Dataflow.txt` shown in Figure 7 to store extracted texts, and `Dataflowimg_1.jpg` which contains the extracted image.

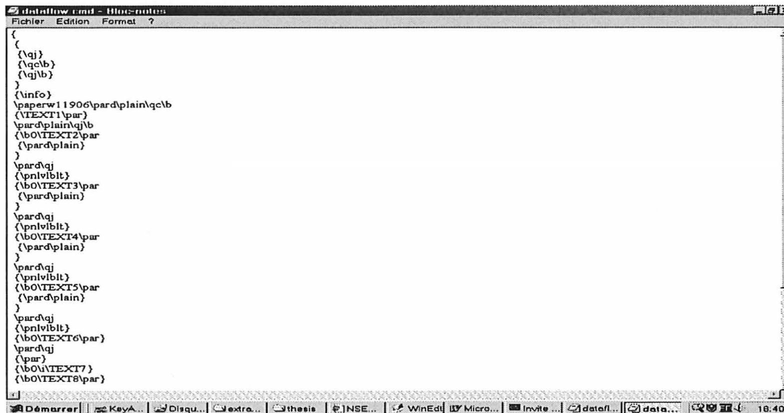
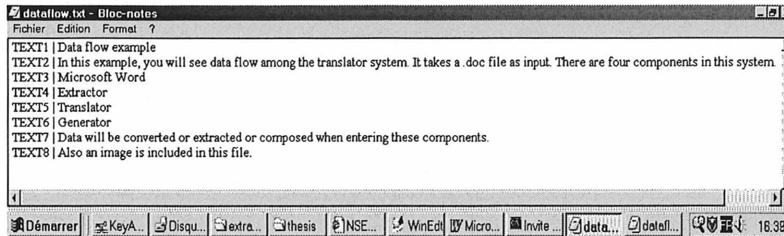


Figure 6: The file `Dataflow.cmd` to store extracted control words

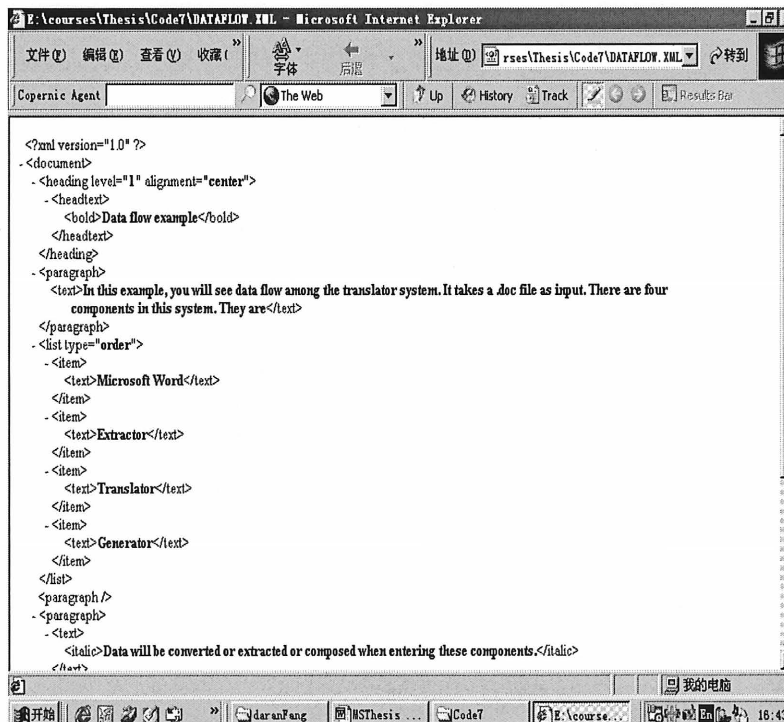


```
TEXT1 | Data flow example
TEXT2 | In this example, you will see data flow among the translator system. It takes a .doc file as input. There are four components in this system.
TEXT3 | Microsoft Word
TEXT4 | Extractor
TEXT5 | Translator
TEXT6 | Generator
TEXT7 | Data will be converted or extracted or composed when entering these components.
TEXT8 | Also an image is included in this file.
```

Figure 7: The file `Dataflow.txt` to store extracted texts

Stage 4: `Dataflow.cmd` Flows into the Translator

The command file `dataflow.cmd` is read by the *translator* which outputs an XML file: `Dataflow.xml` shown in Figure 8.



```
<?xml version="1.0" ?>
- <document>
  - <heading level="1" alignment="center">
    - <headtext>
        <bold>Data flow example</bold>
      </headtext>
    </heading>
  - <paragraph>
      <text>In this example, you will see data flow among the translator system. It takes a .doc file as input. There are four
        components in this system. They are</text>
    </paragraph>
  - <list type="order">
    - <item>
        <text>Microsoft Word</text>
      </item>
    - <item>
        <text>Extractor</text>
      </item>
    - <item>
        <text>Translator</text>
      </item>
    - <item>
        <text>Generator</text>
      </item>
    </list>
    <paragraph />
  - <paragraph>
    - <text>
        <italic>Data will be converted or extracted or composed when entering these components.</italic>
```

Figure 8: The XML output file `Dataflow.xml`

Stage 5: Code Generation

It is the last stage in which the *generator* does code generation taking `Dataflow.xml`, `Dataflow.pre`, `Dataflow_img1.jpg`, and `Dataflow.xsl` as well as `Dataflow.dtd`. Assume an *XMLtoHTML* generator is adopted. Figure 9 shows `Dataflow.html` browsed by *Microsoft Internet Explorer*.

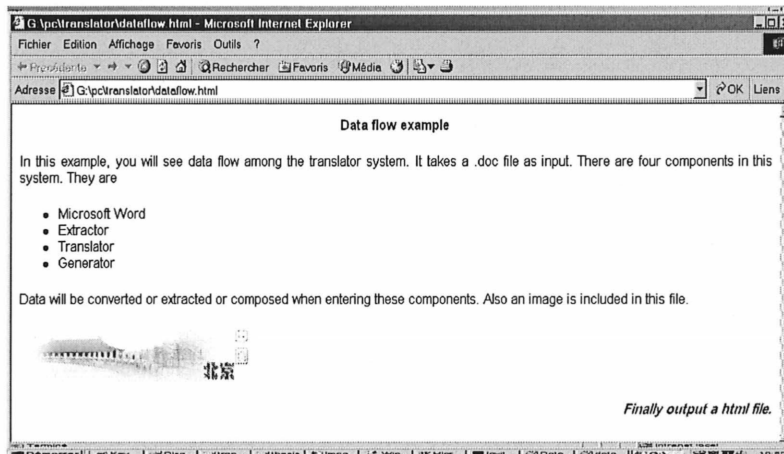


Figure 9: The file `Dataflow.html` browsed by *Microsoft Internet Explorer*

1.3 RTF Syntax

RTF is a document format. RTF is not intended to be a markup language anyone would use for coding entire documents by hands (although it could be done). Instead, it is meant to be a format for document data that all sorts of programs can read and write. By Rich Text Format Specification version 1.6 [10], RTF is a file format for encoding graphics and formatted text to permit easy transfer between different applications and operating systems. RTF breaks down into four basic categories: ***control words, control symbols, groups, and plaintext.***

1.3.1 Control Words

A control word is a specially formatted command that RTF uses to mark printer control code and information that applications use to manage documents. A control word cannot be longer than 32 characters. A control word takes the following form:

\LetterSequence<Delimiter>

Three parts are involved in each control word. The first is a backslash. Then comes an identifier that is made up of lowercase alphabetic characters (a-z). RTF is case sensitive. The final part is a delimiter. The delimiter can be a space, a digit, a hyphen (-) or any character other than a letter or a digit. For example, **\fs24** is a control words which means the size of some text is 24 half-points.

The control properties of certain control words (such as bold, italic, keep together and so on) have only two states. When such a control word has no parameter or has a nonzero parameter, it is assumed that the control word turns on the property. When such a control word has a parameter of 0, it is assumed that the control word turns off the property. For example, **\b** turns on bold, whereas **\b0** turns off bold.

1.3.2 Control Symbols

A control symbol consists of a backslash followed by a single, nonalphabetic character. Control symbols take no delimiters. There are only three control symbols that are of general interest: **\~** is the one that indicates a nonbreaking space; **\-** is an optional hyphen (a hyphenation point); and _ is a nonbreaking hyphen (that is, a hyphen that is not safe for breaking the line after). The escape ***** is also part of a construct we do not discussed now.

1.3.3 Groups

A group consists of text and control words or control symbols enclosed in braces ({ }). The opening brace " { " indicates the start of the group and the closing brace " } " indicates the end of the group. Each group contains the text affected by the group and the different attributes of that text. A RTF file can also include groups for fonts, styles, screen color, pictures, footnotes, comments (annotations), headers and footers, summary information, fields, and bookmarks, as well as document, section, paragraph, and character formatting properties. If the font, file, style, screen-color, revision mark, and summary-information groups and document-formatting properties are included, they must precede the first plain text character in the document. These groups form the RTF file header. If the group for fonts is included, it should precede the group for styles. If any group is not used, it can be omitted.

1.3.4 Plain Text

The control words, control symbols, and braces constitute control information. All other characters in the file are plain text. Figure 10 shows an example of plain text that exists within a group.

```
{\RTF1{\nofpages1}{\nofwords0}{\nofchars0}{\vern835
1}\widoctrl\ftnbj\sectd\linex0\endnhere\pard\plain
\fs20 This is plain text.\par}
```

Figure 10: A RTF file to show plain text

The phrase "This is plain text" is part of a group, but it is not a control word (no backslash before it) and is treated as a document text.

1.3.5 A RTF Example: Hello, World!

RTF is not designed for coding document by hands. But we can still do it to show the basic syntax of RTF. Figure 11 shows the source code written by hands.

```
{\RTF1\ansi\deff0 {\fonttbl {\f0 Times New Roman;}}
\qc\f0\fs24\b Hello, World!}
```

Figure 11: RTF control words for `HelloWorld.rtf`

The **\RTF1** at the start of the file means "the file that starts here will be in RTF, Version 1," and it is required of all RTF documents. The **\ansi** means that this document is in the ANSI character set.

The **\deff0** says that the default font for the document is font #0 in the font table, which immediately follows. The **{\fonttbl...}** construct is for listing the names of all the fonts that may be used in the document, associating a number with each. The **{\fonttbl...}** construct contains a number of **{\fnumber** font name;**}** constructs, for associating a number with a font name. This ends the prolog to the RTF document; everything afterward is actual text.

The **\qc**, the first part of the actual text of the document, means that this paragraph should be centered. The **\f0** means that we should now use the font associated with the number **0** in the font table, namely, Times New Roman. The **\fs24** means to change the font size to 12 points. The **\fs** command's parameter is in half-points: so **\fs24** means 12 points. As you probably inferred, the **\b** means bold on the font sytle. The space after the **\b** does not actually appear in the text, but merely ends the **\b** token.

And finally, the document ends with a **}**. While there are other **}**'s in the document, we know that it is the one that ends this document because it matches the **{** that started the document, and because there is nothing after it in this file.

To see the RTF file of this example, open a *Notepad* text editor and type it in, save this file as `HelloWorld.rtf`, and finally open it with a *Word* processor. Figure 12 shows the RTF document `HelloWorld.rtf`.

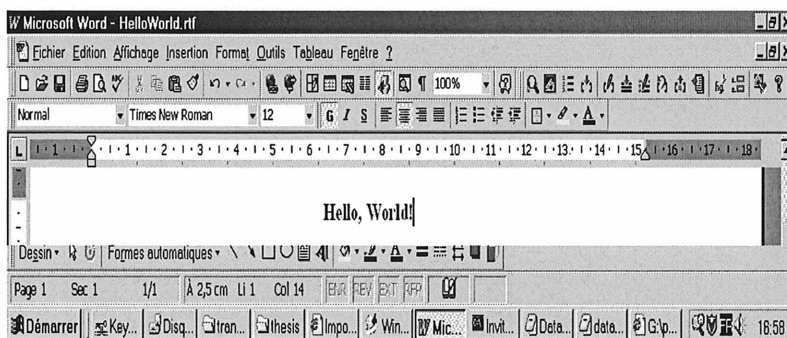


Figure 12: The RTF file `HelloWorld.rtf`

1.4 A Compact Package of Control Words

The *extractor* works on a compact package of RTF control words. Although there are hundreds of RTF control words by Rich Text Format Specification version 1.6 [10], the *extractor* only extracts 95 kinds of control words from a RTF document. The reason is that only a few number of RTF control words are related with the layout of a RTF document and therefore are considered important. Many other control words are either ignored or set by default. All RTF control words are classified into four categories by the translator system.

Useless Control Words

The first category is for those control words which are not useful at all in the *translator* implementation although they may be useful in other applications. For example, **\facenter** is a control word for font alignment. But in the *translator* design, a font is not allowed to have alignment attributes. Only a paragraph/heading/cell has alignment. So this kind of control words is useless for the translator system and is simply excluded by the *extractor* and the *translator.*

Least Important Control Words

The second category is for those control words which have minor contribution to a document construction. For example, **\levelindentN** that shows minimum distance from the left indent to the start of the paragraph text (used for *Word* 7.0 compatibility only), is not very important to build a paragraph. Such control words are also excluded.

Less Important Control Words

The third category is for those control words which are important but their value is not taken out. Instead they are assigned a default value for purpose of simplification. For instance, **\levelnfcN** is used to specify the number type for the item icon. The number **N** starts from 0 to 2 146. Each refers a number type in an ordered list, such as Arabic number or Chinese number. In the *translator* implementation, the number type of an ordered list is always set to the Arabic number system (1, 2, 3,...). This kind of control words is important for the *translator* but a fixed value is adopted. Therefore the third category is not included by the *extractor* and the *translator.*

Most Important Control Words

The final category is for those necessary to construct a document. For example, **\pnlvlbody** which shows an ordered list is such a necessary control word. With this control word, the *translator* knows there is a list in the source document and its type is ordered. The control word database of the *extractor* only contains these tags.

The *extractor* adopts only 95 RTF control words. They are key control words for fonts, paragraphs, lists, tables, and images which can basically control layout and appearance of documents written in the target format. The other control words cannot be recognized by the *extractor* unless they are added into its database of control words. Since the *translator* takes a `.cmd` file from the *extractor*, the *translator* deals with at most 95 control words. Table 1 shows all 95 RTF control words on which the *extractor* and the *translator* work.

Table 1: The translator recognizable RTF control words

bin	'	*	\0x0a	\0x0d	tab	lquote	rquote
ldblquote	rdblquote	{	}	\	plain	b	i
ul	ulw	sub	super	strike	striked	line	par
pard	ql	qc	qj	qr	outlinelevel	fldrslt	ls
pnlvlblt	pnlvlbody	shppict	jpegblip	pngblip	nonshppict	emfblip	macpict
wmetafile	dibitmap	wbitmap	picwgoal	pichgoal	pmmetafile	trowd	row
cell	cellx	intbl	brdrw	trgaph	paperw	margl	margr
info	fldinst	bkmkstart	pict	author	buptim	colortbl	comment
creatim	doccomm	fonttbl	footer	footerf	footerl	footerr	footnote
ftncn	ftnsep	ftnsepc	header	headerf	headerl	headerr	keywords
operator	pntext	printim	private1	revtim	stylesheet	rxe	subject
tc	title	txe	xe	fs	picscalex	picscaley	

The tags inside of the table are RTF control words for font size and style, paragraph alignment, page break, page numbering, page headers and footers, widow-and-orphan control, ordered list, unordered list, image, and other features as well as some important control symbols. The concrete meaning of each tag is out of this thesis although some of them are explained somewhere in this thesis when necessary. But one thing worth mention is that these tags are complete and accurate to control a normal RTF document. They are loaded into the *extractor* RTF control words' database. The *extractor* reads the words among the table and then the *translator* processes them. Those that are not in the table are simply neglected by the *translator*.

Compare two `HelloWorld.rtf` documents. The first one is created by hands and its internal code is shown in Figure 11. The second one is created in normal way: converted from `HelloWorld.doc` by *Microsoft Word*. Both have exactly content-- **Hello, World!**. But their internal code is quite different. The internal code of the second one, shown in Figure 1, is much larger and more complex than that of the first one. Why the two RTF documents with the same content have so different internal representations? The answer is too many extra words exist in the second file. A limited number of RTF control words are enough to decide a document attribute and its appearance. Figure 13 shows the control words extracted by the *extractor*. Compared to about 300 control words shown in Figure 1, only 10 are extracted and these control words combined with the extracted TEXT1 are enough to re-create `HelloWorld.rtf`.

```
{
 {
  {\qj}
 }
 {\info}
 \paperw11906\margl1800\margr1800\pard\plain\qc
 {\b\TEXT1}
 {\par}
}
```

Figure 13: The command file `HelloWorld.cmd`

Chapter 2

Overview of the Translator

2.1 Text Processing Domain

The *translator* is an application of text processing and transformation domain. Like any other applications in this domain, the *translator* works with procedures including character reading, lexical analysis, syntax analysis, semantic analysis, code optimization, and code generation. The function **`ConstructTree()`**, shown in Figure 14, is developed for character reading, lexical analysis, and syntax analysis. The function **`FirstScan()`**, shown in Appendix A, is for semantic analysis. The function **`SecondScan()`**, shown in Appendix A, is for code optimization. And the function **`ThirdScan()`**, shown in Appendix A, is for code generation. The four functions are the main functions in the *translator* program. They play important roles in text processing and transformation during translation.

```
void TranslatorTree::ConstructTree(ifstream &inFile)
{   //omit some code lines for simplification//
    value=0;// initialize the value attached in the end of a token
    int BlockID=1;
    ImageID=1;
    BlockNode* pRoot=new BlockNode(0);
    SequenceNode* child=new SequenceNode();
    pRoot->AddRightSon(NULL);//the root node has no right child
    pRoot->AddLeftSon(child);// the root node has only a sequence node
    m_tree=(TreeNode*)pRoot;
    pCurrentNode=(TreeNode*)child;
    while(!inFile.eof() )//read the whole characters of the command file
    {
      getline(inFile,mstr);//for character reading
      for(int i=0;i<mstr.length();i++)//read each letter in the string
      {
        ch=mstr.at(i);//read one letter at "i"
        if (ch=='{') //meet a block
        {
          //make a block and push into the  tree
          treeStack.push( (TreeNode*)pCurrentNode);
          pBlock=AddBlockNode( &pCurrentNode->m_pLeft,BlockID++);
          pSequence=AddSequenceNode(&pBlock->m_pLeft);
          pCurrentNode=pSequence;
        }
        else if(ch =='\\' )//a token ends
        {
          int pos=mstr.length();//last token without ending "}" or "\"
          for(int j=i+1;j<mstr.length();j++)//for making a token
             if(mstr.at(j)=='\\'||mstr.at(j)==')' || mstr.at(j)=='{'}
             { //count the position when the token ends
               pos=j;
               break;
             }
          token=mstr.substr(i+1,(pos-i-1)); //make a token
          makeDataNode();// a function to store the token into
        }
        else if(ch=='}')//End Block
        {
          //Pop Parent Node from Stack
          pCurrentNode=(SequenceNode*)treeStack.top();
          treeStack.pop();
```

```
            if(pCurrentNode!=pRoot)
            {   //make a sequence node
                pSequence=AddSequenceNode(&pCurrentNode->m_pRight);
                pCurrentNode=pSequence;
            }
}
```

Figure 14: The function **ConstructTree()**

The data structure of the *translator* is a binary tree. All characters of the `.cmd` file for a RTF document are read and loaded into the tree with the function **ConstructTree()**. The function is responsible for building the data tree. When characters are read, they are firstly grouped into different tokens. This procedure is called lexical analysis. Then tokens are classified into three categories: block, sequence, and data node. This procedure is called syntax analysis. After syntax analysis, these nodes are loaded into the tree.

2.1.1 Character Reading

Character reading in the *translator* program is to read a character stream letter by letter from a `.cmd` file supplied by the *extractor*. Because the `.cmd` file is attached with an **ifstream** object **inFile** (defined in the **main** function), the function **ConstructTree()**, shown in Figure 14, is ready to read the stream. In this function, a character variable **ch** is used to load a letter from a line of the stream. The string function **getline()** reads line one by one.

2.1.2 Lexical Analysis

Lexical analysis is a procedure to collect sequence of characters into meaningful units called tokens. Each token in the *translator* program consists of one or more characters

that are collected into a unit before further processing takes place. There are six kinds of tokens that are recognized by the *translator* program:

- control word – a sequence of lower case letters between a backslash and either another backslash or a left brace or a right brace is a control word;
- text – "TEXT" and following integer digits is a text;
- field – "FIELD" and following integer digits is a field;
- bookmark – "BOOKMARK" and following integer digits is a bookmark;
- image – "IMAGE" is an image;
- image name – a sequence of letters after IMAGE followed by an empty space and before a right brace is an image name.

In the **`ConstructTree()`** function, a character string called **`token`** is used to store a token.

2.1.3 Syntax Analysis

After tokens are produced, syntax analysis is performed which determines the structural elements of the `.cmd` file as well as their relationships [11]. The results of syntax analysis are represented as a syntax tree in the *translator* program according to a context-free grammar. Figure 15 shows a general example of a syntax tree.

Context-Free Grammar for Syntax Analysis

Block-> {Sequence}| { }

Sequence-> Block Sequence | Tag Sequence | Image Sequence | Field Sequence

| Bookmark Sequence | Text Sequence | Block | Tag | Text | Bookmark | Field | Image

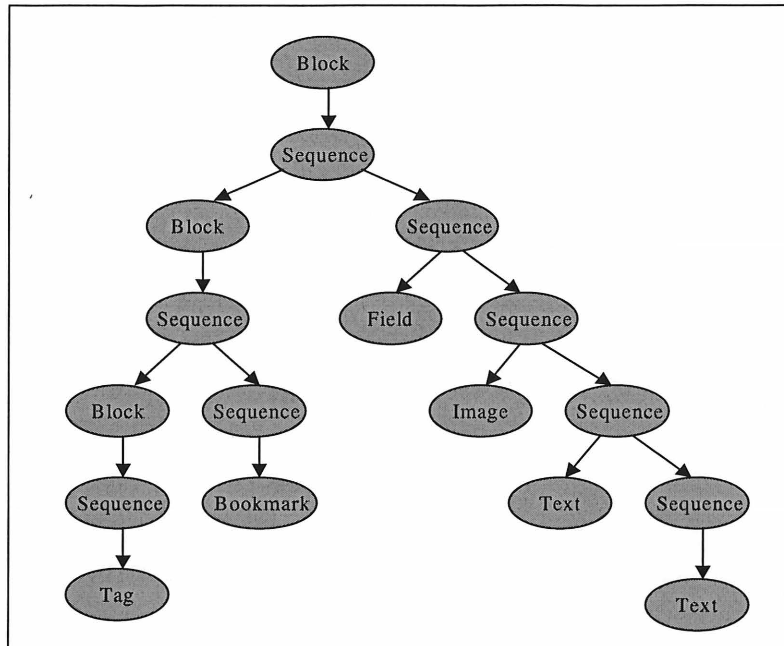


Figure 15: A syntax tree constructed by the *translator* program

2.1.4 Semantic Analysis

Semantic analysis is to interpret text meaning as opposed to its syntax. The translation is done in this procedure. In the *translator* implementation, the function **`FirstScan()`** is for translation. A set of rules, called translation rules, specifies concrete translation issues. A top-down traversal of the syntax tree is carried out in the semantic analysis phase of the *translator* program.

2.1.5 Code Optimization

The goal of code optimization, also called improvement, is to increase the efficiency of the target code [12]. The function **`SecondScan()`** is for code optimization. A top-down traversal of the syntax tree is carried out in code optimization in the *translator* program. A

set of rules, called simplification rules, specifies concrete code improvement in the program. There are two cases that can be applied with the simplification rules. The first one happens in a block which contains useless data. In this case, the block is simply jumped during the traversal. The second one happens in two or more nearby texts which have the same style attributes. In this case, these texts are merged together and share the same attributes.

2.1.6 Code Generation

This is the last stage of text processing and transformation domain. The *translator* outputs the target code XML. The function **`ThirdScan()`** is for code generation. Like semantic analysis and code optimization, code generation also adopts a top-down traversal of the syntax tree.

2.2 Binary Tree Data Structure

The data structure of the system is implemented with a binary tree. A binary tree can be implemented either explicitly in a linked list or implicitly in an array [13]. With the former it is easier to maintain a parent-child relationship but it takes more memory space because of the storage used for references. The latter is much less self-documenting but saves space because there are no references. In the translator system, the explicit linked list representation is selected. The main reason is that the binary tree constructed by the translator is not full or complete in most translation cases. The linked list representation for binary trees does not need to account for the gaps where nodes are missing. On the contrary, the array representation has to keep control of the slots which do not contain actual tree elements [14].

Each node is an instance of class **TreeNode** or its subclasses. A node keeps two **TreeNode** pointers to point to its left child and right child. The tree nodes are holders of input RTF control words. Every control word is read and then stored in this binary tree. Physically, there are eight kinds of nodes in this tree. But functionally, we can categorize them into three kinds by the functions they own. First is **TreeNode**, a parent node used for making children. Because each node is an object of the **TreeNode**'s subclasses, recursion calls which traverse the whole tree are valid. Second are intermediate nodes, **BlockNode** and **SequenceNode**, which do not hold any data for document contents. They show the structure and sequence of control words. Third are the target nodes in which control words actually locate. Without exception, target nodes are the tree leaves.

The *translator* mainly works on the `.cmd` file. A `.cmd` file has a solid format as shown in Figure 6 and Figure 13. This kind of file always starts with an open brace and ends with a close brace. Between them are a series of blocks and sequences of tags. The *translator* reads the `.cmd` file and builds a binary tree based on its contents. So the root of the tree is always a block and this block keeps a single pointer to one child only called **sequenceNode**. The **sequenceNode** maintains a pointer to its left child –a block or one of target nodes–and another pointer to its right child called **sequenceNode.** Next, the left child of the root node goes on to point to its child if it is a block. Otherwise, it is one of target nodes (a leaf) and does nothing. The right child, which is also a **sequenceNode**, points to its two children as its parent does. Then each child has its own children and points to the children, respectively. This child creation process goes on until reading the last control word in the `.cmd` file.

2.2.1 The Class TreeNode

Figure 16 shows the class **TreeNode**, a super class from which its child classes inherit. It is an abstract class with a pure virtual function and other four regular functions. There are

three attributes in the class **TreeNode**: **m_NodeType** to contain the type of each node; **m_pLeft** and **m_pRight** to point to the left child node and right child node, respectively. The pure virtual function **GetNodeType()** is overridden by the derived classes. Other four functions are implemented for getting or adding two children of a node object.

```
class TreeNode
{
public:
        NODETYPE m_NodeType;
        TreeNode* m_pLeft;
        TreeNode* m_pRight;
        TreeNode();
        ~TreeNode();
        virtual NODETYPE GetNodeType()=0;
        void AddLeftSon(TreeNode* tn);
        void AddRightSon(TreeNode* tn);
        TreeNode* GetLeftSon();
        TreeNode* GetRightSon();
}
```

Figure 16: The class **TreeNode**

2.2.2 Intermediate Nodes

Intermediate nodes are used to express a process that can identify nodes that store attribute information. Normally they are in the middle of the binary tree. Leaves are the only nodes to store RTF document attributes and contents. Intermediate nodes do not hold any document information, but they can guide a traversal function where to go and what is the next kind of node. Two kinds of intermediate nodes are used in the binary tree implementation. Figure 17 and Figure 18 show class **BlockNode** and class **SequenceNode**, respectively.

```
class BlockNode : public TreeNode
{
public:
	int id;
	bool use;
	BlockNode();
	BlockNode(int i);
	~BlockNode();
	bool GetUse();
	void SetUse(bool myUse);
	NODETYPE GetNodeType();
};
```

Figure 17: The class **BlockNode**

The Boolean variable **use** is for showing if the node has useful data in the second traversal for simplification. The function **GetUse()** and **SetUse()** are implemented for getting and setting the variable **use**. The function **GetNodeType()** is overridden and it returns the **NT_BLOCK** node type.

```
class SequenceNode : public TreeNode
{
public:
	SequenceNode();
	~SequenceNode();
	NODETYPE GetNodeType();
};
```

Figure 18: The class **SequenceNode**

2.2.3 Target Nodes

Target nodes are nodes in which useful RTF information locates. The information is something required for generating the target file. Totally, there are five kinds of important

data in a RTF document: control words (tags) to regulate document appearance, text contents (texts), images, bookmarks, and fields. The control words are RTF command tags to decide and control the appearance of a RTF document. The texts are all sequences of characters in a RTF document that have a meaning for the user. The images are pictures included in a RTF document. A bookmark is a term to refer to a location within a RTF document, referred elsewhere as an anchor.

A field is used as a placeholder for data that might change in a RTF document and for creating form letters and labels in mail-merge documents or as a hyperlink to a destination. Besides a hyperlink, other common uses for fields are the following:

- Display information about a document such as the author's name, the file size, or the number of pages. To do so, use the AUTHOR, FILESIZE, NUMPAGES, or DOCPROPERTY field.
- Add, subtract, or perform other calculations. To do so, use the = (Formula) field.
- Work with documents in a mail merge. For example, insert ASK and FILLIN fields to display a prompt as *Microsoft Word* merges each data record with the main document.

Correspondingly, five kinds of nodes are implemented to store the information.

2.2.3.1 The Class `TagNode`

This class is used for producing a tag node to hold a RTF control word. A private string variable **`m_strTag`** is used to store a control word; an integer **`value`** records the value accompanied with the control word. For example, when **outlinelevel2** is read, **outlinelevel** is stored in the string **`m_strTag`** and value **2** is stored in the variable **`value`** that regulates the level of a title. Figure 19 shows the class **`TagNode`**.

```
class TagNode : public TreeNode
{
private:
	string m_strTag;
	int value;
public:
	TagNode();
	~TagNode();
	void SetTag(string str);
	void SetValue(int m_value);
	string GetTag();
	int GetValue();
	NODETYPE GetNodeType();
};
```

Figure 19: The class **TagNode**

2.2.3.2 The Class FieldNode

This class is used for producing a field node to hold a field in a RTF document. The variable **fieldID** is used to identify the field. The Boolean variable **fieldText** is used to indicate if the field has a text inside. Figure 20 shows the class **FieldNode.**

```
class FieldNode : public TreeNode
{
private:
	int fieldID;
	bool fieldText;
public:
	FieldNode(int id);
	~FieldNode();
	void SetID(const int id);
	int GetID();
	void SetFieldText(bool flag);
	bool GetFieldText();
	NODETYPE GetNodeType();
};
```

Figure 20: The class **FieldNode**

2.2.3.3 The Class **BookMarkNode**

This class is used for producing a bookmark node to hold a bookmark in a RTF document. Variable **bookID** is used to identify the ID of the bookmark. Figure 21 shows the class **BookMarkNode.**

```
class BookMarkNode : public TreeNode
{
private:
        int bookID;
public:
        BookMarkNode(int id);
        ~BookMarkNode();
        void SetID(const int id);
        int GetID();
        NODETYPE GetNodeType();
};
```

Figure 21: The class **BookMarkNode**

2.2.3.4 The Class **ImageNode**

This class is for producing an image node to hold an image in a RTF document. The variable **imageID** is used to identify the image. The variable **ImageWidth** is used to load the width of the original image. The variable **ImageHeight** is used to load the height of the original image. The variable **ImageScaleX** is used to load the horizontal scaling value. And the variable **ImageScaleY** is used to load the vertical scaling value. Figure 22 shows the class **ImageNode.**

```
class ImageNode : public TreeNode
{
private:
	int imageID;
	string imageFile;
	int ImageWidth;  //width of original Image
	int ImageHeight; //height of original image
	int ImageScaleX; //Horizontal scaling value
	int ImageScaleY; //vertical scaling value
public:
	ImageNode(int id);
	~ImageNode();
	void SetID(const int id);
	int GetID();
	string GetFile();
	void SetFile(string str);
	void SetWidth(const int width);
	int GetWidth();
	void SetHeight(const int height);
	int GetHeight();
	void SetScaleX(const int scaleX);
	int GetScaleX();
	void SetScaleY(const int scaleY);
	int GetScaleY();
	NODETYPE GetNodeType();
};
```

Figure 22: The class **ImageNode**

2.2.3.5 The Class **TextNode**

This class is for producing a text node to hold the reference of a text in a RTF document. The variable **TextID** is used to identify the text. We use a template table **t_table** to record all attributes for font and alignment of this text. The Boolean variable **merge** is used to determine if the text is merged with the next text in a simplification process. The function **CompareText()** is used to compare if two texts have the same attributes and styles. Figure 23 shows the class **TextNode.**

```
class TextNode : public TreeNode
{
public:
	int t_table[20];
	int TextID;
	bool merge;
	int attributeNum;
	TextNode(int id);
	TextNode();
	~TextNode();
	void initialTable();
	int GetTextID();
	bool CompareText(TextNode* a, TextNode* b);
	bool GetMerge();
	NODETYPE GetNodeType();
};
```

Figure 23: The class **`TextNode`**

2.2.4 The Node Representation

The binary tree implemented in the *translator* is not a general one. Its nodes are not the same object each other although their classes are inherited from the same class. The input data of the *translator* are from `.txt` and `.cmd` files supplied by the *extractor*. The `.txt` file is used in code generation stage (see Chapter 5). In the stage of translation, only the `.cmd` file is concerned. The data from the `.cmd` file is a subset of RTF and has a similar form as `Dataflow.cmd` shown in Figure 6. Figure 24 shows the node presentation in a syntax tree. The figure displays only part of the syntax tree representation taking `Dataflow.cmd` as the input. This tree in the figure stops at the block that contains the tag node **\info** and the sequence node that contains the rest of the nodes.

Figure 24: Part of the syntax tree generated from the file `Dataflow.cmd`

2.3 Object-Oriented System

Object-oriented system is defined by Ian Graham as "inheriting the features of both object-based and class-based systems and additionally having full inheritance between classes" and "support self-recursion" [15]. Class-based is explained as abstraction plus object-based which means encapsulation as well as object identity. So, an object-oriented system could be simply defined as: object-oriented system = encapsulation + object identity + inheritance + abstraction + self-recursion. The translator system is such an OO system with characteristics of encapsulation, abstraction, inheritance, and self-recursion.

2.3.1 Encapsulation

The data structures and method implementation details of an object are hidden from other objects in the system. The only way to access an object's state is to send it a message that causes one of its methods to execute. In our implementation of the *translator*, the attributes or variables are almost set private. The access methods, **Setxxx()** and **Getxxx()**, are responsible to access an object state. For example, every node object is encapsulated as such way. To pick up control tags, the function **GetTag()** is used to do this job rather than access tags directly. The advantage of using data encapsulation is safety for future change of the translator system. For instance, the database used to store control words is replaced by a vector for some reasons. With the encapsulation design, users can still use the interface for accessing tags while alter inner database of the **TagNode**. So it is safe and efficient for future modifications.

2.3.2 Abstraction

The second character the translator system has is abstraction. Abstraction is a design technique that focuses on the essential aspects of an entity and ignores or conceals less important or non-essential aspects [16]. In the translator system, abstraction is concerned with both the attributes and behaviors of an object. The goal of abstraction in the system is to make entity (tree node) simpler, more accurate, more coherent, and more complete. Working as a general node, **TreeNode** only possesses basic attributes (two children and **m_NodeType** to show the type of a node) and attribute-access functions (to set and get data). The pure virtual function **GetNodeType()** does nothing except supplying a template for the child classes to override it. The functions **GetNodeType()** overridden in subclasses are dynamically bound at run time. Figure 16 displays the **TreeNode** implementation. For those target nodes, the specific attributes and functions are added.

For example, a **`FieldNode`** is an object to represent a field. Besides what a field already inherits from the **`TreeNode`**, two additional attributes, **`fieldId`** and **`fieldText`**, and the corresponding access functions are supplied (see Figure 20). Abstraction design applied in the translator system maintains the system as a simple efficient one.

2.3.3 Inheritance

The third character the translator system has is inheritance. Inheritance is the means by which data and behavior are transferred from one class to another. Main objects in the system are instances of the classes inherited from the base class **`TreeNode`**. An important reason to implement inheritance is that the binary tree is chosen as the data structure of the *translator*.

Tree is an excellent data structure for semantic analysis. It is often seen that each node in a tree implementation is the same object. However, images, fields, and texts are very different entities with different attributes in the *translator*. If only one class is used to instantiate these entities, many attributes would be set to NULL. Inheritance is the right approach to use. The implementation becomes more efficient and the code is easier to understand. The entities have some common ground because they have the same ancestor. So, all entities can be put on a tree and are visited with recursion calls. Figure 25 shows the relationship between entities inside the tree.

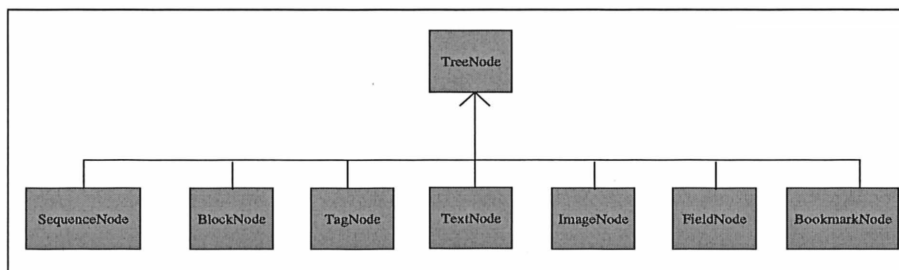


Figure 25: Inheritance between nodes

2.3.4 Self-Recursion

The last character of OO system is self-recursion or self-reference. This means that objects can send messages to their own methods recursively or send messages to themselves. Many recursion called functions are implemented in this system. **`FirstScan()`**, **`SecondScan()`**, **`DestroyTree()`**, and **`PrintTree()`** in the class **`TranslatorTree`** are such functions by which objects of the class **`TranslatorTree`** can do self-recursion. Actually, the binary tree data structure often uses recursion to traverse the tree.

2.4 Algorithm

Control words decide the appearance of a document. So, the words should be translated by means of some translation rules. Generally speaking, every RTF control word can match with one corresponding XML tag. In real programming, the function **`FirstScan()`** is implemented to traverse the tree and the RTF control words are transformed by the translation rules. If more than two results exist, one should be chosen from them by some preference rules.

Three traversals are executed in the tree. The first traversal is for translation matter guided by the translation rules. During a top-down traversal, all attributes about font and alignment for example, are collected in a template table. All or parts of attributes on the table are then copied back into one of three nodes: a **`TextNode`**, a **`FieldNode`**, or an **`ImageNode`** when one of them is met. These three nodes supply all data in generating the target document. The second traversal is for simplification. Some nodes store useless or unrelated data for translation. Some nodes and their children do not make contribution at all to the translation. Both kinds of nodes are regarded as useless nodes by

simplification rules. Sometimes two nearby text nodes are defined by two sets of attributes. The simplification rules regulate this case. The third traversal is for purpose of generating the target document. During this traversal, the useless nodes are jumped. If multi-choice situation happens, preference rules are applied. The algorithm of the translator system is based on the three rules.

2.4.1 Translation Rules

Translation rules, also called association rules, are the fundamental part of the *translator*. It maintains a relationship between RTF control words and XML tags (or other format). After RTF control words are read into the binary tree, the important RTF words are translated into something the translator knows by translation rules.

2.4.2 Simplification Rules

The simplification rules are used to simplify a complex relation into the simpler one. Some extracted RTF control words are not related to document attributes. So they are neglected and are signed **NONUSE** flags. During code generation stage in the third traversal, the nodes to store these control words are jumped over. Some neighboring texts which have the same property are separated and defined with two sets of control tags. In this case these texts can be merged together and shared with the same set of control words. The simplification job is done through the second traversal for purpose of efficiency.

2.4.3 Preference Rules

The preference rules are used for choosing one result among multi-choices specified by users. Some RTF control words have more than one corresponding control word of the target document for translation. Users can select an option or a default value is used.

Chapter 3

Translation Rules

Translation rules, also called association rules, are a set of rules to regulate translation in the translator system. There is a relationship between RTF control words and XML or HTML tags (or in other formats). Like any other rules in the world, the translation rules are a statement that describes what is true in most or all cases. As the developer of the translator system, we prescribe the body of regulations to govern the conduct of translation. During the first traversal throughout the tree (by the function **`FirstScan()`**), the translation rules are applied for concrete translation issues. Because XML tags are actually defined by a DTD as users wish, only some important control words and their associations between RTF, XML, and HTML are demonstrated in this translation system. The translation rules on fonts, paragraphs, images, lists, bookmarks, hyperlinks, fields, and tables are discussed in this chapter. Since there are some associations between RTF, HTML, and XML formats, it is possible for us to find the relations between them and enact rules for translation.

3.1 Some Associations Between RTF, HTML, and XML

Table 3 shows part of control words in the control words database (shown in Table 1) of the *extractor* and their corresponding tags in XML and HTML. Generally, all associations represented in Table 3 are the translation rules by which corresponding control words of the target file could be found. In implementation layer, more concrete rules are needed.

From this table, some RTF tags have more than one corresponding HTML tags capable to realize the same function. For example, italic font in RTF is controlled by **\i**. But there are four control tags in HTML to decide this attribute. This is where the preference rules come from.

3.2 Translation Rules on Fonts and Paragraphs

Font and alignment are the basic attributes for texts and paragraphs. This section demonstrates what the translation rules are on fonts and paragraphs and how they work with some examples.

3.2.1 The Font and Paragraph Attribute Table

Table 2: The table storing font and paragraph attributes

i	B	ul	ulw	sub	sup	strike	striked	fsN	qi	qc	qr	ql
								N				

The attributes in Table 2 are control words for font size and style, and paragraph alignment. Eight of them are about font style; four attributes are about paragraph alignment; and one attribute is about font size.

Table 3: Some associations between RTF, XML, and HTML

FUNCTION	RTF	XML	HTML
The default font size	\fs24 (default is 24)	size (default is 3)	<BASEFONT SIZE=... >
Boldface	\b	bold	B or STRONG
Italic	\i	italic	CITE or EM or I or VAR
Underline	\ul	ul	U
Underline in a word	\ulw	ulw	
A line through text	\strike	strike	S or Strike
Two lines through text	\striked	striked	
Subscript	\sub	sub	SUB
Superscript	\sup	super	SUP
Ordered list	\pnlvlbody	list (type="order")	<OL START=value>
Unordered list	\pnlvlblt	list (type= "unorder")	<UL TYPE="...">
Create paragraph	\par	paragraph	P
Insert a line break	\line		BR
Centering text	\qc	center	align= "center"
Left-aligned	\ql	left	align= "left"
Right-aligned	\qr	right	align= "right"
Justified	\qj	justified	align= "justify"
A plain text style	\plain	plain	PLAINTEXT
A table		table	table
A row of a table	\row	row	tr
A cell of a row	\cell	cell	td
The level of a heading	\outlinelevel0	level1	H1
The level of a heading	\outlinelevel1	level2	H2
The level of a heading	\outlinelevel2	level3	H3
The level of a heading	\outlinelevel3	level4	H4
A png file	\pngblip	source=.png	IMG SRC=.png
A jpeg file	\jpegblip	source=.jpeg	IMG SRC=.jpeg
The original image width	\picwgoalN	width	WIDTH=N
The original image height	\pichgoalN	height	HEIGHT=N
Horizontal scaling value	\picscalexN	scaleX	
Vertical scaling value	\picscaleyN	scaleY	
A hyperlink	\fldinst	<field href=xxx>	<A href=xxx>
Go to a bookmark	\fldrslt	<field bookName=xxx >	<A href=#xxx>
A bookmark	\bkmkstart	bookmark	<A name=xxx>

- **i** in cell 0 is for italic.
- **b** in cell 1 is for bold.
- **ul** in cell 2 is for an underline.
- **ulw** in cell 3 is for an underline of a word.
- **sub** in cell 4 is for subscript.
- **sup** in cell 5 is for superscript.
- **strike** in cell 6 is for inserting text with a line through it.
- **striked** in cell 7 is for inserting text with double lines through it.
- **fs** in cell 8 is the only control word for font size.
- **qj** in cell 9 is for justified alignment.
- **qc** in cell 10 is for center aligned.
- **qr** in cell 11 is for right aligned.
- **ql** in cell 12 is for left aligned.

Font and paragraph alignment are two basic features in a RTF document. Text is the important elements that constitute a RTF document. Moreover, font size and style are associated with every text. Paragraphs have wider meaning in RTF. A regular paragraph in a RTF document is a paragraph. Besides, each item in a list is a paragraph too. Paragraph alignment is also a basic feature in RTF construction. One important reason why font and alignment are put together into a table is that both sets of control words are changeable when they meet a new text or paragraph in the first tree traversal.

Conflict Attributes

Some attributes inside the table, called **conflict attributes**, cannot exist at the same time.

- **qc**, **qj**, **qr**, and **ql**;
- **sub** and **sup**;
- **strike** and **striked**;
- **ul** and **ulw**.

For example, a paragraph can have only one alignment attribute and should be regulated by one of four alignment tags.

Non-conflict Attributes

Some attributes inside the table, called **non-conflict attributes**, can exist at the same time. They are **fs**, **i**, **b**, one of **sub** and **sup**, one of **strike** and **striked**, one of **ul** and **ulw**, and one of align attributes. At most seven attributes can exist at the same time. For example, the expression "***font sample***", between the quotation marks, is italic, bold, and underline with the font size of 12 points.

3.2.2 Rules in Implementation Level

A RTF group is functionally a basic structure just like a sentence in English. A sentence is a piece of text with some grammars to define it. So, the sentence you are reading is active voice and present tense. By analogy, each group specifies the text and contains attributes of that text. A group can include another group inside which also has its child group. Any child group could either have its own control words to define itself or have some or no control word. In last case, the child group simply uses its parent group's attributes.

In implementation, we use a template table exactly like Table 3, called **m_table**, to store font and alignment attributes. The template table **m_table** is implemented as an integer array. Every cell in the table stores those thirteen attributes. A block node corresponds to a group in a RTF file, and a text node is used to hold font and alignment information. In fact, a field node is a special text node. So, the final goal of the first traversal of the binary data tree is to collect necessary information about a text or field node and then write into this text or field when it is met. That is what the first traversal does. For easier demonstration, the *translator* is assumed to scan the tree during the first traversal.

Based on the analysis above, we get the translation rules on fonts and paragraphs.

Rule 1: All attributes in the table are initialized to default values before the traversal begins. Every cell except **fs** is set to 0 which means *false*; 1 means *true*.

Rule 2: When the *translator* enters a block, a copy of the table is made. And this copy may be modified while the *translator* processes the nodes inside the block depending on the conflicted attributes or not.

Rule 3: When the *translator* leaves the block, the original table is taken in further scan and the copied one is thrown away. In other words, the attributes in the original table are kept unchanged unless a tag node in the same level occurs. Two nodes locating inside of the same block are considered in the same level.

Rule 4: When the *translator* meets the tag **\pard**, which means to reset paragraph properties, all paragraph attributes: **qc**, **qj** , **qr**, and **ql** are set to 0.

Rule 5: When the *translator* meets the tag **\plain**, which means to reset default font property, all font attributes are set to 0.

Rule 6: When meeting a text or a field node, the *translator* copies the current table back.

Rule 7: **\par** means a new paragraph.

3.2.3 An Example Translating with the Rules on Fonts

In real tree traversal, the top-down left-right traversal is adopted. According to this traversal, the leaf nodes that store control words and texts/fields/images in the .cmd file

are processed by an order exactly as a person reads the `.cmd` content: top-down and left-right. For example, the person reads this file from top to bottom (line 1 before line 2) and reads every control word or tag from left to right within a line. This sequence matches the tree traversal order. Because the processing order is the same as human normal reading, there is no need to show a complex tree. Instead, the real traversal is simulated with reading the `.cmd` file directly by a person. Everything between an opening brace and a closing brace is regarded as a block. Any attribute inside a block is effective for this block only. Any tag after a tag is regarded as a **SequenceNode** until there is nothing left in a block. In other examples, this simulation way is applied to demonstrate the tree traversal.

Figure 26 shows an example, which includes a RTF file and its two extracted files: a text file and a command file, used to explain the translation rules. Assume the *translator* starts to process. By Rule 1, firstly the *translator* holds a temporary table shown by Table 4 with the default value in every cell. This table, called **attributeTable**, is implemented with one dimension integer array.

Table 4: The **attributeTable** with default values

i	b	ul	ulw	sub	sup	strike	striked	fsN	qj	qc	qr	ql
0	0	0	0	0	0	0	0	24	0	0	0	0

Line 1: For easy demonstration, suppose the *translator* starts from line 1. The first tag that the *translator* meets in line 1 is **\pard** that means setting default value on a paragraph. By Rule 4, **qj**, **qc**, **qr**, and **ql** are reset to 0 although they are already in zero. The second tag is **\plain** that means to reset all font styles to zero. So **i**, **b**, **ul**, **ulw**, **sub**, **sup**, **strike**, and **striked** are set to zero by Rule 5. The third one is **\qr** so the *translator* sets the cell that corresponds to **qr** in **attributeTable** to 1. After finishing the first line, the *translator* gets the table shown in Table 5.

--RTF file:

First paragraph

Second paragraph and then next

Three paragraph

--Text file:

```
TEXT1 | First paragraph
TEXT2 | Second paragraph
TEXT3 | and then next
TEXT4 | Three paragraph
```

--Command file:

```
{
 {
  {\ql}
  {\qj\outlinelevel0}
  {\ql\outlinelevel1\b}
 }
 {\info}
 \pard\plain\qr                    //line 1
 {\b\fs24\TEXT1\par}               //line 2
 \pard\plain\qc                    //line 3
 {\i\ul\fs28\TEXT2}                //line 4
 {\fs28\TEXT3}                     //line 5
 {\i\par}                          //line 6
 \pard\plain\ql\b\fs36             //line 7
 {\i\TEXT4\par}                    //line 8
}
```

Figure 26: Three files (RTF, text, and command) for testing translation rules on fonts

Table 5: The **`attributeTable`** ending in line 1

i	b	ul	ulw	sub	sup	strike	striked	fsN	qj	qc	qr	ql
0	0	0	0	0	0	0	0	24	0	0	1	0

Line 2: The *translator* continues to traverse on line 2. A block is found by meeting an opening brace. By Rule 2, the *translator* gets a copy of the original table and gives the copy with the name "**copyTable**". The first tag the *translator* meets inside the block is **\b**. So, the second cell of **copyTable** corresponding to **\b** is set to 1. The next tag is **\fs24**. No change on it. Then the *translator* meets a text node. By Rule 6, the *translator* copies **copyTable** into the text node in which the integer array **m_table** of the text node loads every cell value in **copyTable**. The last control word in line 2 is **\par** so the current paragraph finishes and a new paragraph is made by Rule 7. Although the *translator* does not change the attribute table, with **\par** it is known that TEXT1 alone consists of a paragraph. This control word is very useful in the third tree traversal for code generation. Finally, the *translator* meets a closing brace which means ending the block. By Rule 3, **copyTable** is thrown away. Only **attributeTable** is carried further for traversal. With the table stored within the TEXT1, TEXT1 is known as bold and its size is 24. And it is right aligned in its paragraph.

Line 3: The first control word is **\pard**. By Rule 4, all alignment attributes are reset. Next is **\plain**. All fonts are reset. Finally, the *translator* meets **\qc**. So, **\qc** cell of the table is set to 1. Table 6 shows the **attributeTable** when the *translator* ends scan on line 3.

Table 6: The **attributeTable** ending in line 3

i	b	ul	ulw	sub	sup	strike	striked	fsN	qj	qc	qr	ql
0	0	0	0	0	0	0	0	24	0	1	0	0

Line 4: Again, it is a block. So, by Rule 2, the *translator* copies **attributeTable** into a new table called **copyTable**. The first three control words are **\i\ul\fs28**. **\i** and **\ul** are both non-conflict attributes so italic and underline can exist at the same time. The *translator* sets both cells to 1. And the font size is altered to 28. Finally, the *translator*

copies **copyTable** into TEXT2 because the *translator* meets a text node. By Rule 3, **copyTable** finishes its job and is deleted. Then, the *translator* goes to the next line with **attributeTable**.

Line 5: This line is almost the same as line 4 except no italic and underline. So the *translator* only changes the font size into 28 as it did in line 4 and copies the new table into TEXT3. Finally, it keeps **attributeTable** unchanged and goes to the next line.

Line 6: The *translator* meets a block. As the *translator* did in line 4 and line 5, it gets a copy of **attributeTable** and traverses along the line. Since meeting **\i**, it wants to change the italic cell. However, there is no existing text node. So, the *translator* does nothing on copying new attributes. Again the *translator* carries **attributeTable** for the next line.

Line 7: There is no brace in this line. So, the *translator* does not need to get a copy of the original table. It carries **attributeTable** for further traversal. It meets **\pard\plain\ql\b\fs36**, respectively. Because alignment attributes are conflict ones, **qr** is reset. Finally, the *translator* gets the new **attributeTable** shown in Table 7.

Table 7: The **attributeTable** ending in line 7

i	b	ul	ulw	sub	sup	strike	striked	fsN	qj	qc	qr	ql
0	1	0	0	0	0	0	0	36	0	0	0	1

Line 8: The *translator* firstly gets a copy called **copyTable**. The only font and alignment control word is **\i**. So, the *translator* just changes the italic cell of **copyTable** into 1. Then the table is copied back into TEXT4. The *translator* finally deletes **copyTable** and finishes the traversal.

3.3 Translation Rules on Headings

A heading (title) is a special paragraph. Like a paragraph, a heading contains some piece of texts with an alignment attribute. The difference between a paragraph and a heading is that a heading has a fixed font size for every component within it.

Rule 1: A heading is a specific paragraph identified by the control word **\outlinelevel**.

Rule 2: The font size of a heading is uniform and is decided by **N** in **\outlinelevelN**. The regular font size control word **\fs** within the heading is neglected.

Figure 27 shows the command file `testHeading.cmd`. By Rule 1, a heading is obtained. This heading includes three texts only. Each has a different font size attribute. However, all real font sizes are actually controlled by **\outlinelevel0** as prescribed by Rule 2. Therefore the font sizes for TEXT1, TEXT2, and TEXT3 are the same, which are **0** levels.

```
{
 {
  {\qj\fs21}
  {\qj\b\fs44}
 }
 {\info}
 \paperw11906\margl1800\margr1800\pard\plain\qj
 \outlinelevel0\b\fs44
 {\TEXT1}
 {\fs28\TEXT2}
 {\fs36\TEXT3}
 {\par}
}
```

Figure 27: The command file `testHeading.cmd`

3.4 Translation Rules on Lists

Generally speaking, there are two types of lists in RTF documents: ordered lists and unordered lists. Some lists also include lists called inner lists. Items which compose a list may have different font attributes. Before the translation rules on lists are presented, some fundamental knowledge and syntax are introduced first.

Microsoft Word 1997 and *Microsoft Word 2000* store bullets and numbering information very differently from earlier versions of *Microsoft Word.* In *Word 6.0,* for example, number formatting data is stored individually with each paragraph. From *Microsoft Word 1997* onwards, however, all the formatting information is stored in a pair of document-wide list tables and each individual paragraph stores only an index to one of the tables. The translator system deals with *Microsoft Word 1997* upwards only. There are two list tables in RTF: the list table (destination **\listtable**), and the list override table (destination **\listoverridetable**).

The List Table

The first table is the list table. A list table is a list of lists (destination **\list**). Each list contains a number of list properties that pertain to the entire list and a list of levels (destination **\listlevel**), each of which contains properties that pertain only to that level. The syntax for the list table is shown in Figure 28.

List Override Table

The list override table is a list of list overrides (destination **\listoverride**). Each list override contains the **listid** of one of the lists in the list table, as well as a list of any properties it chooses to override. Each paragraph contains a list override index (keyword

ls) which is a 1-based index into this table. Most list overrides do not override any properties. Instead, they provide a level of indirection to a list. The two types of list overrides are, in general: (1) formatting overrides, which allow a paragraph to be part of a list, are numbered together with the other members of the list, but have different formatting properties; and (2) **startat** overrides, which allow a paragraph to share the formatting properties of a list, but have a different **startat** value. The first element in the document with each list override index takes the start-at value that the list override specifies as its value, while each subsequent element is assigned the number succeeding the previous element of the list.

```
<listtable>  '{' \*\listtable <list>+ '}'
<list> \list\listemplateid & (\listsimple|listhybrid?
& <listlevel>+ & \listrestarthdn & \listid & (\listname
#PCDATA ';')
<listlevel>  <number> <justification> & \leveljcnN? &
\levelstartatN  &  (\leveloldN  &  \levelprevN?  &
\levelprevspaceN? & \levelspaceN? & \levelindentN?)?
&  <leveltext>  &  <levelnumbers>  &  \levelfollowN  &
\levellegalN? & \levelnorestartN? & <chrfmt>? & \li?
& \fi? & (\jclisttab \tx)?
<number>   \levelnfcN | \levelnfcnN | (\levelnfcN &
\levelnfcnN)
<justification>  \leveljcN | \leveljcnN | (\leveljcN
& \leveljcnN)
<leveltext>  '{' \leveltext \leveltemplateid? #SDATA
';' '}'
<levelnumbers>  '{' \levelnumbers #SDATA ';' '}'
```

Figure 28: The syntax for the list table

Among many control words on lists, only small parts of them are vital for deciding the list main structure from view of a human being. They are **\pnlvlbody**, **\pnlvlblt**, and **\ls**. The rest is ignored. In order to be able to build a list, some compulsory rules are added.

3.4.1 The Compulsory Rules on Lists

There are three translation rules for lists and they are all compulsory ones.

Rule 1: No choice for the numbering system of an ordered list.

Rule 2: No choice for the bullet icon of an unordered list.

Rule 3: All item alignment attributes in a list should be left aligned.

The reason the compulsory rules are enacted is for efficiency. Otherwise many extra control words are added in order to build a list. With the compulsory rules, many control words can be excluded by the translator system. To see the extra words and for future extensions, related control words and their meanings are listed.

Control Words for the Number Type of Lists Excluded by Rule 1

- **\levelnfcN** – specifies the number type for the level. Some common number styles are listed in figure 29. **N** is an integer from 0 to 2 146.

The listed values, shown in Figure 29 are the only legal values for number system of lists in RTF Specification version 1.6 [10]. It is not technically a problem to store those number systems and corresponding information. For example, a hash table can be used to store data. However, it would make the system larger and slower. Moreover, a majority of those number systems are seldom used in the daily activity in a specific region. For efficiency, no information exists for numbering type in the *translator* implementation. Such choice is made in the *generator* level by the preference rules.

```
0 – Arabic (1, 2, 3)
1 – Uppercase Roman numeral (I, II, III)
2 – Lowercase Roman numeral (i, ii, iii)
3 – Uppercase letter (A, B, C)
4 – Lowercase letter (a, b, c)
5 – Ordinal number (1st, 2nd, 3rd)
6 – Cardinal text number (One, Two, Three)
7 – Ordinal text number (First, Second, Third)
10 – Kanji numbering without the digit character
11 – Kanji numbering with the digit character
14 – Double Byte character
15 – Single Byte character
22 – Arabic with leading zero (01, 02, 03, ...)
24 – Korean numbering 2 (*ganada)
25 – Korean numbering 1 (*chosung)
26 – Chinese numbering 1 (*gb1)
30 – Chinese Zodiac numbering 1 (* zodiac1)
33 – Taiwanese double-byte numbering 1
37 – Chinese double-byte numbering 1
41 – Korean double-byte numbering 1
255 – No number
2146 – Phonetic double-byte Katakana characters
```

Figure 29: Values in **\levelnfcN** and the corresponding meaning

Also, there are specific control words for each specific numbering style. Some of them are shown in Figure 30. Too many control words would aggravate the translating job compared to dealing with a single control word such as **\levelnfcN** as introduced above. These control words are also ignored by the translator system.

Control Words for the Unordered List Icons Excluded by Rule 2

- **\pntxtb**: It contains the characters used for bullets.
- **\levelnfcN**: It is the same control word involved in Rule 1. Actually the value **23** in **\levelnfc23** is used to represent bullet (no number at all).

There are many icons to show unordered lists such as

o

➢

□

✓

And thousands of self-defined icons/symbols as long as users can find in a database supplied by *Microsoft Word*, such as

λ

ν

®

➔

¿

For the same reason as the one mentioned in the first rule, all options for unordered list are ignored. Actually, no control words for icon types of items are kept for purpose of efficiency. The choice for the icons is made in the *generator* level by the preference rules.

```
\pncard – Cardinal numbering (One, Two, Three)
\pndec – Decimal numbering (1, 2, 3)
\pnucltr – Uppercase alphabetic numbering (A, B, C)
\pnucrm – Uppercase Roman numbering (I, II, III)
\pnlcltr – Lowercase alphabetic numbering (a, b, c)
\pnlcrm – Lowercase Roman numbering (i, ii, iii)
\pnord – Ordinal numbering (1st, 2nd, 3rd)
\pnordt – Ordinal text numbering (First, Second, ...)
\pnbidia – Abjad Jawaz for Arabic and Biblical Standard
          for Hebrew
\pnaiu – Phonetic Katakana characters
\pnganada – Korean numbering 1 (*ganada)
\pngbnum – Chinese numbering 1 (*gb1)
```

Figure 30: Part of control words for number type of ordered lists

Control Words for the Item Alignment Excluded by Rule 3

- **\leveljcnN** –
  - 0 – left aligned is for left-to-right paragraphs and right justified for right-to-left paragraphs.
  - 1 – center aligned.
  - 2 – right aligned is for left-to-right paragraphs and left aligned for right-to-left paragraphs.
- **\pnqc** – centered numbering.
- **\pnql** – left-aligned numbering.
- **\pnqr** – right-aligned numbering.

In the *translator* implementation, different items in a list are not allowed to have different alignment attributes. All items should be left justified because center or right alignment makes no sense for practical use. Figure 31 shows a list whose items have different alignments. By Rule 3, all items in a list have only a left aligned alignment.

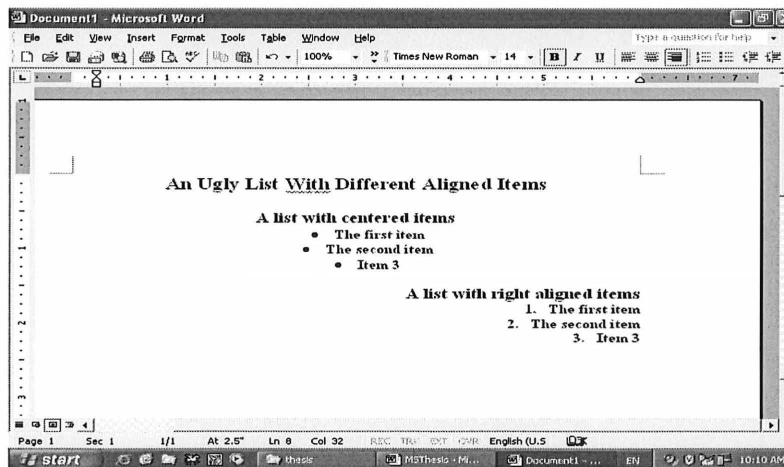


Figure 31: An ugly list with different aligned items

3.4.2 Translation Rules on Items of a List

The control words **\pnlvlbody** and **\pnlvlblt** are two important control words involved in the translation of a list. They are used to differentiate an ordered list and an unordered list, respectively.

- **\pnlvlblt** – bulleted paragraph: the actual character used for the bullet is stored in the **\pntxtb** group.
- **\pnlvlbody** – simple paragraph numbering.

In RTF specification, no specific control words are responsible for list items which compose a list. No control word can be found to show the start or end of an item. But **\pnlvlbody**, **\pnlvlblt**, and some list or paragraph attribute control words allow to identify items of a list. For example, an item is a text or a paragraph consisting of several pieces of texts in the RTF context. The idea is that all meaningful things in a paragraph (texts, bookmarks, fields, hyperlinks, tables, and images) between two list tags are probably an item although a pure text is a common form for an item. Therefore some translation rules on items of a list are obtained.

Rule 1: An item is a text or several texts which compose a paragraph. No more than one paragraph is allowed in an item.

Rule 2: A text or a paragraph between list control words **\pnlvlbody** or **\pnlvlblt** may become content of an item.

But Rule 1 and Rule 2 cannot judge an item at any case. How are a new item and a new paragraph differentiated? Other attributes of a paragraph are used to differentiate them. A RTF file shown in Figure 32 is the input file for the *extractor*.

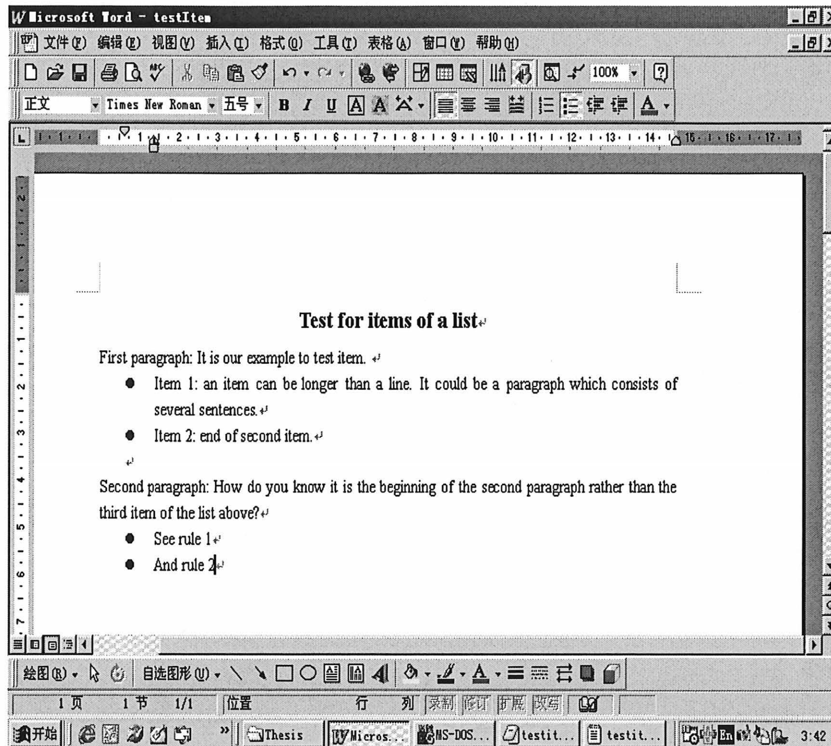


Figure 32: The RTF file `testItem.rtf`

Normally, an item is a paragraph that consists of some texts, and one of two list control words always appears before a text or some texts which compose an item in the *translator*'s input file `.cmd`. Figure 33 shows such patterns. TEXT3, TEXT4, TEXT6, and TEXT7 are items of two lists in Figure 33 and Figure 34. They are always after the list control word **\pnlvlblt**. More important thing is that a paragraph after an item should be either the next item or the new paragraph which ends the list. The control word **\par**–means new paragraph–could be used to judge it.

Rule 3: A new paragraph (a piece of text or several texts), which follows after a list control word (**\pnlvlbody** or **\pnlvlblt**), is the next item. A new paragraph or an empty paragraph without a list control word following means the ending of a list.

```
}
{
 {\ls1}
 {\ls2}
 {\ls3}
 {\ls4}
}
{\info}
\paperw11906\margl1800\margr1800\pard\plain\qc\b\TEXT1\par
\pard\plain\qj
{\TEXT2\par
 {\pard\plain}
}
\pard\qj
{\pnlvlblt\ls3}
\ls3
{\TEXT3\par
 {\pard\plain}
}
\pard\qj
{\pnlvlblt\ls3}
\ls3
{\TEXT4\par}
\pard\qj
{\par}
\pard\qj
{\TEXT5\par
 {\pard\plain}
}
\pard\qj
{\pnlvlblt\ls4}
\ls4
{\TEXT6\par
 {\pard\plain}
}
\pard\qj
{\pnlvlblt\ls4}
\ls4
{\TEXT7\par}
```

Figure 33: The command file `testItem.cmd`

```
TEXT1 | Test for items of a list
TEXT2 | First paragraph: It is our example to test item.
TEXT3 | Item 1: an item can be longer than a line. It could be a paragraph which consists of sevei
TEXT4 | Item 2: end of second item.
TEXT5 | Second paragraph: How do you know it is the beginning of the second paragraph rather than
TEXT6 | See rule 1
TEXT7 | And rule 2
```

Figure 34: The text file `testItem.txt`

Let us simulate the *translator* to translate a list based on the translation rules. After reading `testItem.cmd` file, the *translator* begins to do syntactic and semantic analysis. We jump font and alignment translation and concentrate on lists only. When the *translator* meets the first **\pnlvlblt**, it knows the first item for a list will come in. Next the *translator* scans "TEXT3" which refers to "Item 1: an item can be longer than a line..." (see Figure 33). This text starts the beginning of the first item. Then **\par** is scanned which means a new paragraph start. By Rule 1, the first item is finished. But it is not sure if there are still

other items until the *translator* scans another **\pnlvlblt**. After getting the second **\pnlvlblt** and "TEXT4", the *translator* knows the TEXT4 ("Item 2: end of second item.") is the second item. Then another **\par** is read which means ending of the second item. Is there a third item? The *translator* continues to scan until **\par**. By Rule 3, if a new paragraph comes in without following a list control word, the whole list is completed. So, when the "TEXT5" is read in, the *translator* knows it is a new paragraph after the list rather than a next item. By applying the three rules on lists, the *translator* gets another list which includes two items.

3.4.3 Nested Lists and Normal Lists

A nested list is a list that includes one or more lists inside. **\lsN**–index of the **\listoverride** in the **\listoverride** table–is used to differentiate them. This value should never be zero inside a **\listoverride**, and must be unique for all **\listoverride** within a document. The **N** value in **\lsN** is used to determine an indented item.

Rule 1: All items in a list have the same **N** value in **\lsN**.

Rule 2: If one item has a larger value **N** in its corresponding **\lsN** than the previous item and nothing (paragraphs, images, and tables) separate the item and the previous item, this item is the first item of a nested list.

Rule 3: If one item has a smaller value **N** in its corresponding **\lsN** than the previous item and the value associated to the previous list is equal to **N**, this item is one of items of the previous list.

Figure 35 shows a nested list. Each list control word is followed by a **\lsN** (see `testNestedList.txt` shown in Figure 36 and `testNestedList.cmd` shown in

Figure 37). For the first two items, both have the same value 1 for **N**. But the third item–TEXT4 has 2 in its corresponding **\lsN**. By Rule 2, this item is the first item of a nested list. Next, another item TEXT5 is scanned. Since it also has 2 in its corresponding **\lsN** as the previous item, this item is an item of the nested list by Rule 1. The final item (TEXT6) is read with **\ls1**. Because 1 in **\lsN** is smaller than the value of the previous item and it is equal to the value of the previous list, by Rule 3, this item is the third item of the previous list (outer list) and the nested list is finished at the same time.

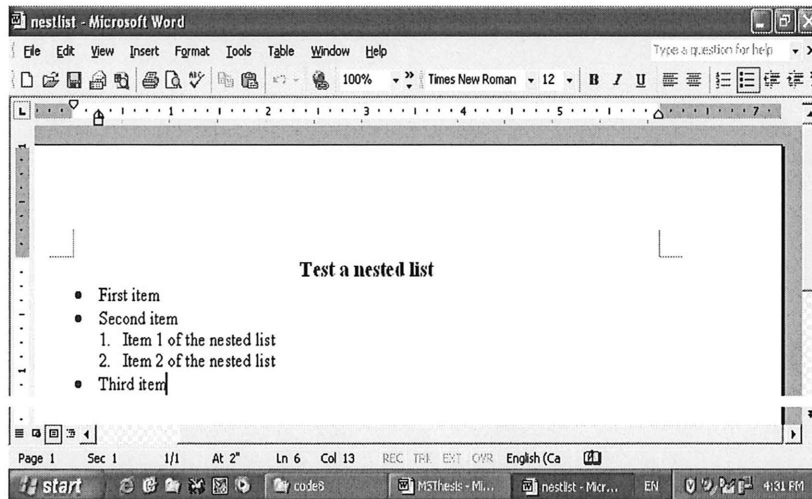


Figure 35: The RTF file `testNestedList.rtf`

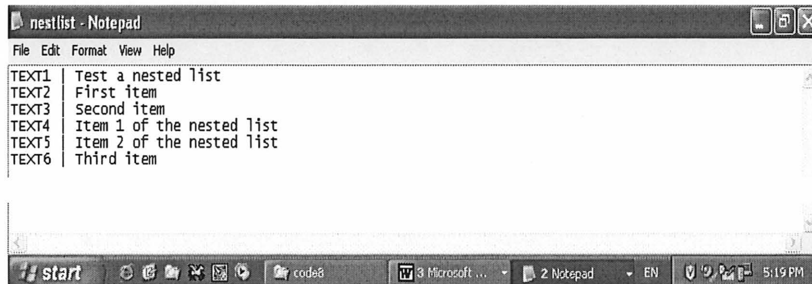


Figure 36: The text file `testNestedList.txt`

```
nestlist - Notepad
File  Edit  Format  View  Help
{
 {\ls1}
 {\ls2}
}
{\info}
\pard\plain\qc\b\fs28\TEXT1\par
{\pard\plain}
\pard\plain\qj
{\pnlvlblt\ls1}
\ls1\fs20
{\fs24\TEXT2\par
 {\pard\plain}
}
\pard\qj
{\pnlvlblt\ls1}
\ls1
{\fs24\TEXT3\par
 {\pard\plain}
}
\pard\qj
{\pnlvlbody\ls2}
\ls2
{\fs24\TEXT4\par
 {\pard\plain}
}
\pard\qj
{\pnlvlbody\ls2}
\ls2
{\fs24\TEXT5\par
 {\pard\plain}
}
\pard\qj
{\pnlvlblt\ls1}
\ls1
{\fs24\TEXT6\par}
\pard\qj
{\fs24\par}
}
```

Figure 37: The command file `testNestedList.cmd`

3.5 Translation Rules on Images

A RTF document can include images (pictures) created with other applications. These pictures can be in hexadecimal (the default) or binary format. Pictures are destinations, and begin with the **\pict** control word, which is preceded by ***\shppict** destination control keyword. A picture destination has the syntax shown in Table 8.

Table 8: The syntax of an image

<table>
<tr><td><pict></td><td>'{' \pict (<brdr>? & <shading>? & <picttype> & <pictsize> & <metafileinfo>?) <data> '}'</td></tr>
<tr><td><picttype></td><td>| \emfblip | \pngblip | \jpegblip | \macpict | \pmmetafile | \wmetafile | \dibitmap <bitmapinfo> | \wbitmap <bitmapinfo></td></tr>
<tr><td><bitmapinfo></td><td>\wbmbitspixel & \wbmplanes & \wbmwidthbytes</td></tr>
<tr><td><pictsize></td><td>(\picw & \pich) \picwgoal? & \pichgoal? \picscalex? & \picscaley? & \picscaled? & \piccropt? & \piccropb? & \piccropr? & \piccropl?</td></tr>
<tr><td><metafileinfo></td><td>\picbmp & \picbpp</td></tr>
<tr><td><data></td><td>(\bin #BDATA) | #SDATA</td></tr>
</table>

Translation on images is quite simple as long as the four control words about size are picked up and their meanings are understood. The image type or metafile information is not necessary since the *extractor* already extracted images out and assigned them with specific names as well as an extension of file such as `Dataflow_1.jpg`. The only thing should know is its size. Measurements within these control words are in twips; a twip is one-twentieth of a point.

- **\picwgoalW** – desired width of the picture in twips: the **W** argument is a long integer.
- **\pichgoalH** – desired height of the picture in twips: the **H** argument is a long integer.
- **\picscalexX** – horizontal scaling value: the **X** argument is a value representing a percentage (the default is 100).
- **\picscaleyY** – vertical scaling value: the **Y** argument is a value representing a percentage (the default is 100).

Translation rules on images are listed below:

Rule 1: The width of original image is **W** in **\picwgoalW**.

Rule 2: The height of original image is **H** in **\pichgoalH**.

Rule 3: The percentage of compression rate on the width of the original image is **X** in **\picscalexX**.

Rule 4: The percentage of compression rate on the height of the original image is **Y** in **\picscaleyY**.

Rule 5: The real size of an image in a RTF is: width= **W*X**; height= **H*Y**.

3.6 Translation Rules on Bookmarks

A bookmark is identified by two control words: **\bkmkstart**, which indicates the start of the specified bookmark, and **\bkmkend**, which indicates the end of the specified bookmark. Bookmarks have the syntax shown in Table 9.

Table 9: The syntax of a bookmark

<book>	<bookstart> | <bookend>
<bookstart>	'{*' \bkmkstart (\bkmkcolf? & \bkmkcoll?) #PCDATA '}'
<bookend>	'{*' \bkmkend #PCDATA '}'

A bookmark example is shown in Figure 38. The bookmark start and the bookmark end are matched with the bookmark tags. In the example, the bookmark tag is "paradigm". Each bookmark start should have a matching bookmark end. The bookmark start and the bookmark end may, however, be in any order.

```
\pard\plain \fs20 Kuhn believes that science, rather than
discovering in experience certain structured relationships,
actually creates (or already participates in) a presupposed
structure to which it fits the data. {\bkmkstart paradigm} Kuhn
calls such a presupposed structure a paradigm.{\bkmkend
paradigm}
```

Figure 38: A bookmark example

The control word **\bkmkcolfN** is used to denote the first column of a table covered by a bookmark. If it is not included, the first column is assumed. The control word **\bkmkcollN** is used to denote the last column. If it is not used, the last column is assumed. These controls are used within the ***\bkmkstart** destination following the

\bkmkstart control. For example, `{*\bkmkstart\bkmkcolf2\bkmkcoll5 Table1}` places the bookmark "`Table1`" on columns 2 through 5 of a table.

In the *translator* implementation, only **\bkmkstart** is extracted for efficiency reason. In order to locate where a bookmark is, a rule is enacted.

Rule 1: The location of a bookmark is the first text or field or bookmark before it.

3.7 Translation Rules on Fields

The **\field** control word introduces a field destination that contains the text of fields. Fields have the following syntax shown in Table 10.

Table 10: The syntax of a field

`<field>`	`'{' \field <fieldmod>? <fieldinst> <fieldrslt> '}'`
`<fieldmod>`	`\flddirty? & \fldedit? & \fldlock? & \fldpriv?`
`<fieldinst>`	`'{*' \fldinst <para>+ <fldalt>? '}'`
`<fldalt>`	`\fldalt`
`<fieldrslt>`	`'{' \fldrslt <para>+ '}'`

What the *extractor* actually extracts are two control words: **\fldinst** and **\fldrslt**.

- **\fldinst** – field instructions.
- **\fldrslt** – most recent calculated result of the field.

Translation on a field is very simple because the *extractor* does most of things for the *translator*. The label **\FIELD** exists in a `.cmd` file. The corresponding content of the field is in a `.txt` file extracted by the *extractor*. The *translator* takes the field from

the .txt file and then separates the field type and the field content. For example, when the *translator* scans **\FIELD8** in the testField.cmd file, it wants to take out all the strings to which the label **\FIELD8** refers in the testField.txt file. So "HYPERLINK http://www.usherbrooke.ca" is picked up. Then the *translator* separates the type and contents. The type of the field is "HYPERLINK" and the content is "http://www.usherbrooke.ca".

A field is classified into an empty field and a regular field. An empty field is a field with its type only, lacking of its field content. A regular field has both type and content which define the field.

Translation rules for fields are listed below:

Rule 1: All information about a field is the field itself, which is stored in the corresponding .txt file.

Rule 2: A field is composed of two parts. One is the field type spelled in capital letters. The other is the field content after the capital letters.

Rule 3: A field is identified by the control word **\fldrslt**. A field is regular only if some texts exist with the control word within the same block otherwise it is empty.

Several common fields are listed here:

- link to existing file;
- link to browsed page;
- link to recent file;
- link to some place in this document (go to a bookmark);
- link to an email address;
- link to a web address.

3.8 Translation Rules on Tables

There is no RTF table group in a RTF specification; instead, tables are specified as paragraph properties. A table is represented as a sequence of table rows. A table row is a continuous sequence of paragraphs partitioned into cells. The table row begins with the **\trowd** control word and ends with the **\row** control word. Every paragraph that is contained in a table row must have the **\intbl** control word specified or inherited from the previous paragraph. A cell may have more than one paragraph in it; a cell is terminated by a cell mark (the **\cell** control word), and a row is terminated by a row mark (the **\row** control word). Table rows can also be positioned. Table properties may be inherited from the previous row; therefore, a series of table rows may be introduced by a single **<tbldef>**.

A RTF table row has the syntax shown in Table 11.

Table 11: The syntax of a table row

<table>
<tr><td><row></td><td>(<tbldef> <cell>+ <tbldef> \row) | (<tbldef> <cell>+ \row) | (<cell>+ <tbldef> \row)</td></tr>
<tr><td><cell></td><td>(<nestrow>? <tbldef>?) & <textpar>+ \cell</td></tr>
<tr><td><nestrow></td><td><nestcell>+ '{*'\nesttableprops <tbldef> \nestrow '}'</td></tr>
<tr><td><nestcell></td><td><textpar>+ \nestcell</td></tr>
</table>

3.8.1 Basic Rules for Tables

Tables are the most complicated case in RTF translation. There are two kinds of translation rules on tables. The first are the basic translation rules capable to draw a simple table. The second are translation rules on the complex tables with nested cells. Some basic control words for tables are introduced in order to know tables well.

- **\trowd**: sets table row defaults.
- **\row**: denotes the end of a row.
- **\cell**: denotes the end of a table cel.
- **\trgaphN** – half the space between the cells of a table row in twips.
- **\intbl** – paragraph is part of a table.
- **\cellxN**: defines the right boundary of a table cell, including its half of the space between cells.
- **\brdrwN**: **N** is the width in twips of the pen used to draw the paragraph border line. **N** cannot be greater than 75. To obtain a larger border width, the **\brdth** control word can be used to obtain a width double that of **N**.

From these control words and table syntax, we define the basic rules on table translation.

Rule 1: A table is a sequence of table rows. Each row ends with the control words **\row**.

Rule 2: The next row follows underneath the previous row in the top-down order.

Rule 3: A row is a sequence of cells. Each cell ends with the control word **\cell**.

Rule 4: The next cell follows exactly right after the previous cell in the left-right order.

Rule 5: A cell consists of a paragraph or more. All paragraphs in cells of a row are followed by the control word **\intbl**.

Rule 6: Border line of the paragraph among cells is **N** twips in **\brdrwlN**.

Rule 7: The space between the cells of a table row is 2**N** twips in **\trgaphN**.

Rule 8: The right boundary of a table cell is defined in **\cellxN**. In other words, a cell ends at the position **N** twips (measured from left margin) in **\cellxN**.

Rule 9: The number of **\cellxN** appeared should match the number of cells in a row.

Rule 10: A table starts when the first cell in the first row starts. A row starts when the first cell in the row starts.

Rule 11: A table ends when a paragraph without **\cell** following it appears.

3.8.2 Translation Rules for Nested Cells

A nested cell is the cell that has its own cells inside. Like regular tables, tables with nested cells should abide regular translation rules for normal tables above. However, nested cell tables have particular rules.

Rule 1: A row consists of a sequence of cells. Different rows can have different number of cells. It is called the nested cell.

Rule 2: A regular cell could appear more than one times in different rows as long as the nested cells in different rows before it has same border with it.

Rule 3: An empty cell which has no text losses its top border and the cell above losses its bottom border. In other word, these two cells join together and appear as a cell.

The rules are explained with an example shown in Figure 39.

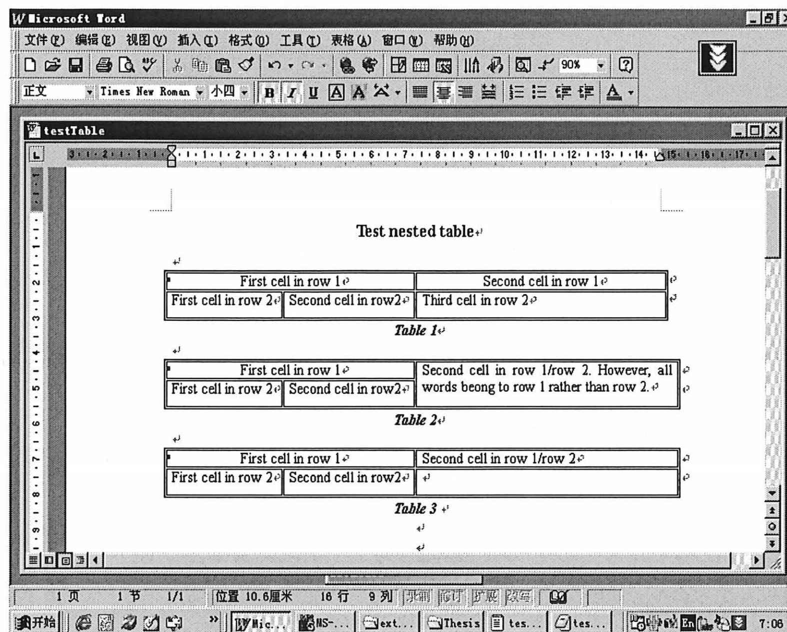


Figure 39: The RTF file `testTable.rtf` for nested tables

By common sense, a table is described with columns and rows. But there is only definition for rows in a RTF table. And the basic unit in a RTF table is a cell. On the one hand, a RTF table can be regarded as a series of rows piled as a stack. The next coming row is put under the previous one in the top-down order. On the other hand, cells in a row are put together in the left-right order. Cell position could be located by each **\cellxN**.

Let us observe how the translation works. Like previous examples, assume the *translator* processes the document. Start with the first table shown in Figure 39. When the *translator* reads **\trowd** at the first time in the `testTable.cmd` file shown in Figure 40, it knows that it is the first cell in the first row in the table by basic Rule 1. **\cellxN** appears two times before **\row** firstly appears. It means the first row has two cells only by basic Rule 1 and Rule 9. The first cell has content of TEXT2 which refers to "First cell in row 1" because the first **\cell** follows TEXT2 (see the text file in Figure 41). The first cell ends at position 4 153 twips (starting from left margin shown in **\cellx4153**) by

basic Rule 8. Then the *translator* gets the second cell with content TEXT3 with right border ended at 8 414 twips.

```
testtable - 写字板
文件(F) 编辑(E) 查看(V) 插入(I) 格式(O) 帮助(H)

\trowd\trgaph108\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10
\brdrw10\brdrw10\brdrw10\cellx4153\brdrw10\brdrw10\brdrw10\brdrw10\cellx8414\pard
\plain\qc\intbl\outlinelevel0\b
{\b0\TEXT2\cell}
\pard\plain\qc\intbl
{\TEXT3\cell}
\pard\intbl
{\b\row}
\trowd\trgaph108\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10
\brdrw10\brdrw10\brdrw10\cellx1885\brdrw10\brdrw10\brdrw10\brdrw10\cellx4153\brdrw10
\brdrw10\brdrw10\brdrw10\cellx8414\pard\qj\intbl
{\TEXT4\cell\TEXT5\cell\TEXT6\cell}
\pard\intbl
{\b\row}
\pard\qc
{\b\i\TEXT7}
\trowd\trgaph108\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10
\brdrw10\brdrw10\brdrw10\cellx4153\brdrw10\brdrw10\brdrw10\brdrw10\cellx8624\pard
\plain\qc\intbl\outlinelevel0\b
{\b0\TEXT8\cell}
\pard\plain\qj\intbl
{\TEXT9\cell}
\pard\intbl
{\row}
\trowd\trgaph108\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10
\brdrw10\brdrw10\brdrw10\cellx1885\brdrw10\brdrw10\brdrw10\brdrw10\cellx4153\brdrw10
\brdrw10\brdrw10\brdrw10\cellx8624\pard\qj\intbl
{\TEXT10\cell\TEXT11\cell\cell}

要"帮助"，请按 F1
```

Figure 40: The command file `testTable.cmd` for nested tables

```
testtable - 记事本
文件(F) 编辑(E) 搜索(S) 帮助(H)

TEXT1 | Test table
TEXT2 | First cell in row 1
TEXT3 | Second cell in row 1
TEXT4 | First cell in row 2
TEXT5 | Second cell in row2
TEXT6 | Third cell in row 2
TEXT7 | Table 1
TEXT8 | First cell in row 1
TEXT9 | Second cell in row 1/row 2. However, allwordsbeong to row 1 rather than row 2.
TEXT10 | First cell in row 2
TEXT11 | Second cell in row2
TEXT12 | Table 2
TEXT13 | First cell in row 1
TEXT14 | Second cell in row 1/row 2
TEXT15 | First cell in row 2
TEXT16 | Second cell in row2
TEXT17 | Table 3
```

Figure 41: The text file `testTable.txt` tested for nested tables

By seeing **\row** and the second **\trowd**, the *translator* knows the first row finishes and the second row begins. Because there are three **\cellxN**s before the second row appears, the *translator* knows the second row has three cells. So, this is a table with nested cells (two cells in first row and three cells in the second row). The three cells end at 1 185, 4 153, and 8 414, respectively. After the second **\row**, there is a separate paragraph which consists of TEXT7 and no **\cell** follows; the *translator* knows a normal text appears outside of the table. So the first table is finished.

Next we concentrate on translation rules on nested tables by translating the second table. From view of human beings, the second column of the table has a single cell only. And this cell (physically larger than its left cells) is neighbored by the two cells which belong to the first row and the second row at the same time. No column idea exists in a RTF table. Only rows and cells are applied in the translation. So, a table is translated row-by-row and cell-by-cell. In the second table, the first cell in the first row is located and translated as normal. Even the second cell, which contains the text "Second cell in row 1/row 2. However, all words belong to row 1 rather than row 2.", is dealt as a normal cell. By taking **\cellxN**, the *translator* knows there are two cells in the first row. Then the *translator* scans the second row and finds three cells in the second row. But the third one has no text inside. See "`{\TEXT10\cell\TEXT11\cell\cell}`" in `testTable.txt`. By Rule 2 and Rule 3 for nested cells, the *translator* gets this larger cell which belongs to the first row and the second row.

Compulsory Rule

A compulsory rule is set for efficiency reasons.

Rule 1: All borders of cells in a table have the same size.

Chapter 4

Simplification Rules

Code optimization is a very important phase in text processing and transforming domain. The target code has to be improved before it is finally generated. Such improvements include eliminating redundant or unnecessary blocks which contain no useful information and merging some piece of texts which hold the same attributes. All these improvements are regulated by a set of rules called simplification rules. The function **`SecondScan()`** is implemented to execute the second traversal of the binary data tree for simplification.

4.1 Simplification Rules on Useless Blocks

A RTF document includes hundreds of control words inside. Even the basic control words are extracted; some of them are still useless or redundant for some specific cases. Two reasons attract us to study blocks for purpose of simplification. Firstly, a block is the main unit in the structured data input file (`.cmd`) and a block may include many other

blocks. Actually the content of the command input file is a block. Another reason is that the attributes for fonts and paragraphs inside a block are independent and the block can be ignored if it is not useful. On the contrary, the attributes outside of a block are not only a copy for this block but also valid for further traversal (see section 3.2.2). So, a block is much simpler to be dealt with. A block is thrown away as long as it includes no information.

The idea is to associate a Boolean flag to each block that shows if this block contains useful data. If the block is redundant or unnecessary, the parser jumps this block and continues to do further scan. In the binary tree data structure, a block node is an intermediate node which can have child nodes such as sequence, text, and all other data nodes. To jump to useless nodes in a tree leads to code optimization in the third traversal for code generation. There are some blocks which are considered useless. Therefore corresponding rules are set for them.

Rule 1: An empty block without any node inside (no child from the view of the parser) is a useless block.

Rule 2: A block including data nodes such as texts, images, fields, and bookmarks is a good block.

Rule 3: A block including lists, tables, and paragraphs is a good block.

Rule 4: The parent of a good block is also good.

Rule 5: All other blocks are useless blocks.

An example for simplification rules on blocks is shown in Figure 42.

```
{
 {
  {\qj}
  {\qc\b}
 }
 {\info}
 \paperw11906\margl1800\margr1800\pard\plain\qc\b\TEXT1\par
 \pard\plain\qj
 {\TEXT1\par
  {\pard\plain}
 }
 {\i
 {\pnlvlblt\ls3}
 }
}
```

Figure 42: An example for simplification rules on blocks

The first block, which includes two inner blocks `{\qj}` and `{\qc\b}`, is regarded as useless because its two child blocks do not have any useful data inside–neither data nodes nor list/table/paragraph/table control words. So, the two inner blocks and their parent blocks are useless by Rule 1 and Rule 5. For the same reason, the next block `{\info}` is useless for the parser. The following block is `{\TEXT1\par {\pard\plain}}`. The inner one `{\pard\plain}` is useless. However, the outer block, which includes a text node, is a good block by Rule 2. The last block is `{\i{\pnlvlblt\ls3}}` which has an inner block. Because its child block `{\pnlvlblt\ls3}` is a good block by Rule 3, the outer block is also useful by Rule 4. Finally, the largest block which includes several useful inner blocks is evaluated as useful block by Rule 4.

4.2 Simplification Rules on Merging

Some continuous texts in a paragraph are separated but their attributes defining their font and alignment features are the same. In this case, these texts are merged into a single text in order to simplify their relationship. See an example shown in figure 43.

--the rtf file

You are reading ***two attributes***.

--the command file

```
{
 {
  {\ql\fs24}
  {\ql\fs20}
 }
 {\info}
 \pard\plain\ql\fs24
 {\TEXT1}
 {\i\fs40\ul\TEXT2}
 {\i\fs40\ul\TEXT3}
 {\TEXT4\par\par}
}
```

--the text file

```
TEXT1 | You are reading
TEXT2 | two
TEXT3 | attributes
TEXT4 | .
```

Figure 43: Three files (RTF, command, and text) for testing merging

Two texts in a sequence can be merged together in the example. They are TEXT2 and TEXT3. Both are italic, underline, and with font size of 40. Before simplification, they should be translated into something like:

```
<text size=40><italic><underline>TEXT2<\underline><\italic><\bold><\text>
```

AND

```
<text size=40><italic><underline>TEXT3<\underline><\italic><\bold><\text>
```

To improve the code, the two texts are merged like:

```
<text size=40><italic><underline>TEXT2 TEXT3<\underline><\italic><\bold>
<\text>
```

We get the simplification rules on merging.

Rule 1: Two texts in different cells in a table cannot be merged.

Rule 2: Two or more texts in a paragraph without different texts and images separating them are merged together if and only if their font and paragraph attributes are the same.

It seems to exclude a field or a bookmark between two possible merged texts. But neither has two qualified texts which can merge together. Firstly, a field is externally extracted as a real text in `.txt` file although it is also extracted as a FIELD with a field name at the same time. Two qualified texts are merged together no matter whether they are field or not. Secondly, a bookmark invisibly exists in a document. So, it does not matter if there is a bookmark between two texts. But an image does matter. Normally an image is often located in a separated paragraph. However in some case, an image could be used as small as a short text in a paragraph. So, the two texts which have a paragraph between them cannot be merged together.

In real programming, two steps are involved to judge if two texts could be merged together. The first is to collect any two neighboring texts (no **\par**, **\cell** or image between them). The second is to pick up their attribute tables and compare them cell-by-cell. If they exactly match, a merge flag is set to indicate that the two texts can be merged. The real merging job is not done in the second traversal. The code optimization is performed in code generation during the third traversal.

Chapter 5

Code Generation

The last phase of text processing and transforming is code generation. Usually, code generation is the most complex phase of an application domain because it depends not only on the characteristics of the source language but also on detailed information about the target architecture and the structure of the runtime environment [11]. The target code of the *translator* is XML. The function **`ThirdScan()`** is implemented to execute the third traversal of the binary data tree for code generation. An activating mechanism is applied to activate concrete code generation for elements and attributes of XML documents. Identification tags, which identify the elements or the attributes and then fire code generation, play an important role in the activating mechanism.

Like other applications of text processing and transforming, code generation also involves some attempts to optimize the target code. The real code optimization is done in this phase although the simplification job is completed in the second traversal. The target code XML is a user-defined file. So, a DTD file for the *translator* is discussed in this chapter. To display the XML result, an XSL file is required.

5.1 XML

5.1.1 What is XML?

Coming straight to the point, XML stands for eXtensible Markup Language. XML is a radically simplified subset of Standard Generalized Markup Language (SGML). XML is a public format and not a proprietary format of any company. The version 1.0 specification was accepted by the World Wide Web Consortium (W3C) as recommendation on February 10, 1998 [17].

XML files always clearly mark where the start and end of each of the logical parts (called *elements*) of an interchanged document occur. By defining the role of each element of a text in a formal model, known as a DTD, users of XML can check that each component of a document occurs in a valid place within the interchanged data stream. An XML DTD allows computers to check, for example, that users do not accidentally enter a third-level heading without first having entered a second-level heading, something that cannot be checked using HTML previously used to code documents that form part of the World Wide Web of documents accessible through the Internet.

XML was not designed to be a standardized way of coding text; in fact it is impossible to devise a single coding scheme that would be suitable for all languages and all applications. Instead XML is a formal language that can be used to pass information about the component parts of a document to another computer system. XML is flexible enough to describe any logical text structure, whether it is a form, memo, letter, report, book, encyclopedia, dictionary or database.

5.1.2 The Components of XML

XML is based on the concept of *documents* composed of a series of *entities*. Each entity can contain one or more logical *elements*. Each of these elements can have certain *attributes* (properties) that describe the way in which it is to be processed. XML provides a formal syntax for describing the relationships between the entities, elements, and attributes that make up an XML document, which can be used to tell the computer how it can recognize the component parts of each document.

XML differs from other markup languages in that it does not simply indicate where a change of appearance occurs, or where a new element starts. XML sets out to clearly identify the boundaries of every part of a document, whether it is a new chapter, a piece of boilerplate text, or a reference to another publication.

To allow the computer to check the structure of a document, users must provide it with a DTD that declares each of the permitted entities, elements and attributes, and the relationships between them.

5.2 DTD

A DTD provides a way of writing markup rules that specify how XML documents can be validly created. When XML 1.0 was originally specified, the DTD syntax (which is not XML-based) was inherited mainly from earlier markup languages, such as SGML and HTML. The syntax introduced in this section is mainly from MSDN Library [18] and is reorganized.

5.2.1 DTD Syntax

DTD uses a specialized non-XML vocabulary, which includes the following grammar for writing and declaring markup rules that define a specific type of an XML document structure:

- ATTLIST declarations;
- ELEMENT declarations;
- ENTITY declarations;
- NOTATION declarations.

5.2.1.1 ELEMENT Declarations

The *ELEMENT* statement is used to declare each element used within the document type defined by the DTD. It first declares an element by name, and then specifies what content is allowed for the element.

Syntax

<!ELEMENT name content >

Parameters

name

The name of the element. Exact case is required.

content

The allowable content model for the element, which must be one of the following:

- **ANY**: any content is allowed within the element. When used in an element declaration, this keyword permits an open unrestricted content model for the element and any of its child nodes.
- **EMPTY**: the element is not allowed to have content and must remain empty.
- **Declared Content Rule**: for this option, users need to write a content rule and enclose it within a set of parentheses.

5.2.1.2 ATTLIST Declarations

The *ATTLIST* (attribute lists) statement is used to list and declare each attribute that can belong to an element. It first specifies the name of the element (or elements) for which the attribute list will apply. It then lists each attribute by name, indicates whether it is required, and specifies what character data it is allowed to have as a value.

Syntax

<!ATTLIST elementName attributeName dataType default >

Parameters

elementName

The name of the element to which the attribute list applies.

attributeName

The name of an attribute. This parameter can be repeated as many times as needed to list all attributes available for use with *elementName*.

dataType

The data type for the attribute named in the *attributeName* parameter, which must be one of the following:

- **CDATA**: the attribute will contain only character data.
- **ID**: the value of the attribute must be unique. It cannot be repeated in other elements or attributes used in the document.
- **IDREF**: the attribute references the value of another attribute in the document, of ID type.
- **ENTITY**: the attribute value must correspond to the name of an external unparsed ENTITY, which is also declared in the same DTD.
- **ENTITIES**: the attribute value contains multiple names of external unparsed entities declared in the DTD.
- **NMTOKEN**: the attribute value must be a name token. Name tokens allow character data values, but are more limited than **CDATA**. A name token can contain letters, numbers, and some punctuation symbols such as periods, dashes, underscores, and colons. Name token values, however, cannot contain any spacing characters.
- **NMTOKENS**: the attribute value contains multiple name tokens. See the description for **NMTOKEN** and **ENUMERATED** for detail.
- **ENUMERATED**: the attribute values are limited to those within an enumerated list. Only values that match those listed are validly parsed. All enumerated data types are enclosed in a set of parentheses with each value separated by a vertical bar ("|").

default

The default value for the attribute named in *attributeName*. Table 12 describes the possible defaults.

Table 12: Default value in ATTLIST declaration

#REQUIRED	The attribute must appear in the XML document or a parsing error will result. To avoid a parse error, you can optionally use the **defaultValue** field directly following this keyword.
#IMPLIED	The attribute can appear in the XML document, but if omitted, no parsing error will result. In some cases you can also use the **defaultValue** field directly following this keyword.
#FIXED	The attribute value is fixed in the DTD and cannot be changed or overridden in the XML document. If this keyword is used, the **defaultValue** field directly following this keyword must also be used to declare the fixed attribute value.
defaultValue	A default or fixed value. The parser inserts this value into the XML document when the attribute is missing or is not used in the XML document. All values must be enclosed in a set of quotation marks (either single or double quotes).

5.2.1.3 ENTITY Declarations

The *ENTITY* statement is used to define entities in the DTD, for use in both the XML document associated with the DTD and the DTD itself.

Syntax

<!ENTITY [%] name [SYSTEM|PUBLIC publicID] resource [NDATA notation] >

Parameters

name

The name of the entity. Required for all entity definitions.

publicID

The public identifier for the entity. Only required if the declaration uses the *PUBLIC* keyword.

resource

The value for the entity. Required for all entity definitions.

notation

The name of a notation declared elsewhere in the DTD using the *NOTATION* statement. Only required when declaring an unparsed entity through the use of the non-XML data (*NDATA*) keyword.

5.2.1.4 NOTATION Declarations

The *NOTATION* statement is used to define notations. Notations allow an XML document to pass notifying information to external applications.

Syntax

<!NOTATION name [SYSTEM|PUBLIC publicID] resource >

Parameters

name

The name of the notation. Required for all notation definitions.

publicID

The public identifier for the notation. Only required if the declaration uses the *PUBLIC* keyword.

resource

The value for the notation. Required for all notation definitions. Typically, if the notation is public it is a URI, readable to humans but not to machines.

5.2.2 The DTD for the *Translator*

```
<!ELEMENT document((paragraph*|heading*|image*|list*|table*)+)>
<!ELEMENT heading(text+)>
<!ATTLIST heading alignment CTADA "justified" level CDATA #REQUIRED >
<!ELEMENT paragraph(text*|field*|bookmark*)>
<!ATTLIST paragraph alignment CTADA "justified">
<!ELEMENT text(italic*|bold*|(ul|ulw)*|(sub|super)*|(strike|striked)*
        |#PCDATA)>
<!ATTLIST text size CDATA "3">
<!ELEMENT italic(bold*|(ul|ulw)*|(sub|super)*|(strike|striked)*
        |#PCDATA)>
<!ELEMENT bold((ul|ulw)*|(sub|super)*|(strike|striked)*|#PCDATA)>
<!ELEMENT ul((sub|super)*|(strike|striked)*|#PCDATA)>
<!ELEMENT ulw((sub|super)*|(strike|striked)*|#PCDATA)>
<!ELEMENT sub((strike|striked)*|#PCDATA)>
<!ELEMENT super((strike|striked)*|#PCDATA)>
<!ELEMENT strike(#PCDATA)>
<!ELEMENT striked(#PCDATA)>
<!ELEMENT field (text*)>
<!ATTLIST field ref CDATA #REQUIRED>
<!ELEMENT bookmark EMPTY>
<!ATTLIST bookmark name CDATA #REQUIRED>
<!ELEMENT list (item+)>
<!ATTLIST list type (order|unorder) # REQUIRED>
<!ELEMENT item (text+|field*|bookmark*|list*)>
<!ELEMENT image EMPTY>
<!ATTLIST image
        source CDATA #REQUIRED
        width CDATA #REQUIRED
        height CDATA #REQUIRED
        scaleX CDATA #REQUIRED
        scaleY CDATA #REQUIRED>
<!ELEMENT table(row+)>
<!ELEMENT row(cell+)>
<!ELEMENT cell(text*)>
<!ATTLIST cell alignment CDATA "justified" position CDATA #REQUIRED>
```

Figure 44: DTD for the *translator*

Figure 44 shows the DTD for the *translator*. The XML output of the *translator* is defined by the DTD. The root node of the DTD is *document*. A document includes five elements: *paragraph, heading, image, list, and table*. Each element has some attributes and contains its own elements with attributes.

5.2.2.1 Definition of a Paragraph

The element *paragraph* is a regular paragraph in the target document. A paragraph has an attribute *alignment* that describes the alignment feature of the paragraph. The default value of *alignment* is "justified". In other word, the alignment of a paragraph in the target file is always justified unless a specific alignment attribute appears. A paragraph contains three elements: *text*, *field*, and *bookmark*.

Text Definition

The element *text* consists of a sequence of letters which have font and font size in the target file. In the DTD for the *translator*, the font size is defined with an attribute of *size* whose default value is 3. The font character is implemented with several elements (*italic, bold, ul, ulw, sub, super, strike, and striked*) instead of an attribute. A font element includes other font elements. For example, *italic* contains *bold*, either *ul* or *ulw*, either *sub* or *super*, and either *strike* or *striked*. *bold* contains either *ul* or *ulw*, either *sub* or *super*, and either *strike* or *striked*. *ul* or *ulw* contains either *sub* or *super*, and either *strike* or *striked*. *sub* or *super* contains either *strike* or *striked*. Although using an attribute is simpler and clearer than using several elements, an element option is adopted in the *translator*. The reason is that an element expression looks nicer in the target file.

Compare two options:

```
<text size=2><italic><bold>text content</italic></bold></text>
```

and

```
<text size=2 font="italic bold">text content</text>
```

The first expression is more popular in structured formatting files, HTML for example.

The *text* element or one of its child elements must contains a PCDATA which forms the atoms of the document, the units from which the higher-level structures are built. The PCDATA here is actually the "plain" text for *text.*

Field Definition

A field is used as a placeholder for data that might change in the target document and for creating form letters and labels in mail-merge documents or as a hyperlink to a destination. The element *field* may contain an element *text* which has exactly the same syntax as *text* in *paragraph.* The only attribute of *text* is *ref* which is a required CDATA. The attribute *ref* defines the name to which a field refers.

Bookmark Definition

A bookmark is a term to refer to a location within the target document, referred to elsewhere as an anchor. The element *bookmark* has no element. The only attribute of *bookmark* is *name* which is a required CDATA. The attribute *name* defines the name of a bookmark.

5.2.2.2 Definition of a Heading

The element *heading* is a title or subtitle in the target document. A heading has two attributes to define it. One attribute is *alignment* that describes the alignment feature of

the heading. Like p*aragraph*, the default value of *alignment* is also "justified". Another attribute for *heading* is *level* that describes the level of the font size. The element *heading* contains only one element *text* whose number is 1 or more.

5.2.2.3 Definition of an Image

An image is a picture in the target document. The element *image* does not have element. Five required CDATA attributes define an image:

- *source* – the name of the image file;
- *width* – the original width of the image size;
- *height* – the original height of the image size;
- *scaleX* – the horizontal scaling value of the image;
- *scaleY* – the vertical scaling value of the image.

5.2.2.4 Definition of a List

A list in the target document has two types: ordered and unordered. Correspondingly, *order* and *unorder* are two attributes that describe the element *list* in the DTD. A list has a single element *item* whose number is 1 or more. The element *item* contains four elements:

- *text*;
- *field*;
- *bookmark*;
- *list*.

The fourth element *list* means that a nested (indented) list exists.

5.2.2.5 Definition of a Table

The last element of document in the DTD is *table*. A table contains a single element *row* whose number is 1 or more.

The element *row* has one element *cell* and a cell has two attributes. The attribute *alignment,* whose default value is also "justified" like any other alignment attribute in the DTD, defines the alignment character of the cell. Another attribute *position* defines the right border position of the cell. A cell has its own element *text.*

5.3 Implementation Aspects

Code generation of the *translator* is implemented with the function **`ThirdScan()`** and other functions called by **`ThirdScan()`**. All related functions and variables could be found in `translator.cpp` shown in Appendix A. According to the DTD for the *translator* (see Figure 44), the target document XML is called *document.* A document may contain five elements: *paragraph, heading, image, list,* and *table.* Each element may have some attributes and contains its own elements with attributes. For example, a paragraph contains elements *field, bookmark,* and *text.* The attribute *alignment* defines the paragraph alignment feature. Also a field has one element *text* and an attribute *ref.* In the implementation level, one or more functions are responsible for generation of the elements and attributes.

The extracted "plain" data about texts, fields, and bookmarks is applied in construction of the target code. The three kinds of data are extracted by the *extractor* and are stored in the `.txt` file. In the code generation stage, the data should be retrieved. In the *translator* program, three string arrays are implemented to load the data before the beginning of code

generation. The three arrays are **text**, **field**, and **bookmark**. They are used to retrieve the related data when a text, a field, or a bookmark is generated.

5.3.1 Identification Tags and an Activating Mechanism

The first thing for code generation is to find an "identification" tag for each element and attribute during the third traversal of the data tree and then activate code generation. An identification tag in this thesis is a RTF control word or a variable that identifies a node type or the combination of a control word and some conditions used for identifying and then activating code generation. The identifying function of an identification tag is similar to a primary key to identify an entity in a database. The activating function is used to start code generation. When an identification tag is met, related code generation starts at once or later depending on the category of the identification tag. The identification tags of some elements and attributes are easily found. But some are quite complex to obtain. For example, the type of an image node, **NT_IMAGE**, is its identification tag since image nodes are unique among other nodes. Whenever **NT_IMAGE** is scanned during the third traversal, image code is generated immediately. On the contrary, more information is required to fire generation of an item of a list. The list control word **\pnlvlbody** or **\pnlvlblt** represents a list item but neither can activate item code generation correctly. A Boolean variable ***itemGoOn*** is added to compose the identification tag for an item. Whenever a list control word is parsed and ***itemGoOn*** is true, item code is produced. The last identification tags are for identifying elements or attributes only. They do not have the right to activate code generation.

Identification tags are classified into *instant*, *conditional*, and *slave* categories based on how long is the activation time or which has the right to activate code generation.

- **Instant Identification Tags**

  An instant identification tag is a RTF control word stored in a tag node or a variable for identifying a tree node. These tags are kind of firing signals. Once the tags are parsed, the corresponding code generation functions are called to produce related code at once without any delay. The identification tag of an image is in this category.

- **Conditional Identification Tags**

  A conditional identification tag is some control words plus one or more Boolean variables. When the control words are met and the Boolean variables are true, code generation starts. The identification tag of a list item is in this category. A conditional identification tag can include Boolean variables only without a control word.

- **Slave Identification Tags**

  A slave identification tag is a control word or a variable for identifying a tree node. When the control word or the type variable is parsed, the corresponding code is ready to generate. But the slave identification tag does not have the power to activate the code generation. Some entity in higher level, acting as a master, keeps control of the code generation. For example, a text is an element of a paragraph. A paragraph can include many texts. It is the function **`MakePar()`** which produces the paragraph to decide when the texts are generated.

The code generation function for the *translator* is **`ThirdScan()`**. This function realizes code generation by finding all identification tags and then calling the corresponding code generation functions directly or indirectly.

5.3.2 Image Generation

The identification tag for an image is **NT_IMAGE** and it is an instant identification tag. In the data tree, an image is uniquely represented by **ImageNode**. Whenever an image node is traversed during **ThirdScan()**, image code is generated by calling the function **MakeImage()** shown in Figure 45. The XML tags to delimit a table ("<image>" and "</image>") are written in the function. Five attributes of image, *source*, *width*, *height*, *scaleX*, and *scaleY*, are obtained by calling the five corresponding **Get()** functions.

```
void TranslatorTree::MakeImage(ImageNode* image)
{
    char* source=image->GetFile();
    //roughly convert to html unit
    int width=image->GetWidth()*0.063;
    int height=image->GetHeight()*0.063;
    int scaleX=image->GetScaleX();
    int scaleY=image->GetScaleY();
    file << "<image source=\"" << source << "\" width="
         << "\"" << width << "\" height=\"" << height
         << "\" scaleX=\"" << scaleX << "\" scaleY=\""
         << scaleY << "\">" << endl;
    file << "</image>" << endl;
}
```

Figure 45: The function **MakeImage()**

5.3.3 Table Generation

In RTF, there is no specific control word for creating a table. A RTF table is composed of a series of rows. And a row is composed of a series of cells. Because the start of a table is known by meeting the first cell at the first row of the table (see translation rules on tables in section 3.8), we can make use of this condition to build the identification tag for a table.

A Boolean variable **doTable** is used to show if it is the first time the table is produced. If it is, the *translator* does table generation and writes the string "<table>". Since there is no control word responsible for building a table and a row according to the translation rules for tables, the cell control word **/cell** and the Boolean variable **doTable** together constitute the conditional identification tag for making a table. Because the RTF control words are stored in **TagNode** in the *translator* program, a tag node should be found at first during the third traversal. Then the node is checked if it stores **/cell**. Finally, the variable **doTable** is checked if it is true. After these conditions are met, the string "<table>" is written into the XML document.

The identification tag for ending a table is two Boolean variables **doTable** and **doRow**. When **doTable** is true and **doRow** is false, the string "</table>" is written into the XML document.

Row Generation

Similar to table generation, the identification tag for row starting is also conditional. The cell control word **/cell** and the Boolean variable **doRow** together constitute the conditional identification tag for making a row. Whenever a **TagNode** that stores **/cell** is traversed and **doRow** is false, the string "<row>" is written into the XML document. The identification tag for row ending is **/row** and it is an instant identification tag. Whenever a **TagNode** that stores **/row** is traversed, the string "</row>" is written out immediately.

Cell Generation

Cells are basic units in a table. Actually, most jobs for generating a table are done in the cell generation stage. In this stage, the cell alignment are computed and cell texts are printed out in the function **MakeCell()** shown in Figure 46. The identification tag for

cell generation is **/cell** and it is an instant identification tag. Whenever **/cell** is identified, **MakeCell()** is called. In this function, alignment of the cell is firstly obtained by calling the function **GetAlignment()**. Secondly, the XML tags to delimit a cell are written in the function. Thirdly, the texts in the cell are generated by calling the function **MakeText()** in a for loop. Finally, in this function, the string "`</cell>`" is written for ending the cell.

The attribute *position* is obtained during the third traversal. The identification tag for cell *position* is **/cellx**. It is forwarded as a parameter into the function **MakeCell()**.

```
void TranslatorTree::MakeCell(TextNode*
     TextArray[],int countText,int Position)
{
     string alignment=GetAlignment(TextArray[0]);
     if(alignment=="justified")
        file << "<cell position=\"" << Position << "\">"
              << endl;
     else
        file << "<cell alignment=" << alignment
              << "position=\""<<Position << "\">" << endl;
     for(int i=0;i<countText;i++)
         MakeText(text[TextArray[i]->GetTextID()-1],
         TextArray[i]);
     file << "</cell>" << endl;
}
```

Figure 46: The function **MakeCell()**

5.3.4 Paragraph Generation

The identification tag for a paragraph is **\par** and it is an instant identification tag. Whenever **\par** is parsed in the third traversal, the function **MakePar()** shown in Figure 47 is called. The XML tags to delimit a paragraph are also written in the function. The

function **DoParagraph()** (shown in Appendix A) is used to generate elements of a paragraph. The text generation function **MakeText()** is called in **DoParagraph()**.

```
void TranslatorTree::MakePar(TreeNode*
     Nodearray[],int length)
{
     if(head!=-1)//to make a header
         MakeHead(Nodearray,length);
     else
     {
          //to make a paragraph
         if(meetTF==false||Paralignment=="justified")
            file << "<paragraph>" << endl;
         if(meetTF==true && Paralignment!="justified")
            file << "<paragraph alignment="
                  << Paralignment << ">" << endl;
         DoParagraph(Nodearray,length);
         file << "</paragraph>" << endl;
        }
}
```

Figure 47: The function **MakePar()**

A paragraph contains elements *text*, *field*, and *bookmark*. A node array, **Nodearray**, stores all elements of the paragraph during the third traversal of the data tree. The array is forwarded as a parameter into the function **DoParagraph()**. Three kinds of elements, *text*, *field*, and *bookmark*, are produced in the function.

5.3.5 Text Generation

The identification tag for a text is **NT_TEXT** and it is a slave identification tag. When a text node is parsed during the third traversal, the text node is stored in the node array **Nodearray**. Unlike image generation which outputs the code right after the identification tag is found, text generation is delayed until its parent entities call it. A text

is a basic unit in an XML document. It is the element of paragraphs, headings, fields, cells, and items. When its parent entities are generated, the text generation function **MakeText()**, shown in Figure 48, is called by its parents, respectively. Since it is a passive character, **NT_TEXT** is not an instant identification tag.

```
void TranslatorTree::MakeText(string
    textString,TreeNode* node)
{
    TextNode* temp=(TextNode*)node;
    int fontSize=GetFontSize(temp);
    string fontList=GetFont(temp);
    int defaultSize=3;
    if(head!=-1)//make text in heading
      file << "<text>" << GetFontList(textString,
           temp)<< "</text>" << endl;
    else
    {
    //make text in a normal paragraph
    if(fontSize==defaultSize && fontList=="plain")
      file << "<text>" << textString << "</text>"
           << endl;
    if(fontSize!=defaultSize && fontList=="plain")
      file << "<text size=\"" << fontSize << "\">"
           << textString << "</text>" << endl;
    if(fontSize==defaultSize && fontList!="plain")
      file << "<text>" << GetFontList(textString,temp)
           << "</text>" << endl;
    if(fontSize!=defaultSize && fontList!="plain")
      file << "<text size=\"" << fontSize << "\">"
           << GetFontList(textString,temp) << "</text>"
           << endl;
  }
}
```

Figure 48: The function **MakeText()**

5.3.6 Heading Generation

The identification tag for a heading is **\outlinelevel** and it is an instant one. Whenever **\outlinelevel** is parsed in the third traversal, the function **MakeHeading()**, shown in Figure 49, is called. The XML tags to delimit a heading are also written in the function.

```
void TranslatorTree::MakeHead(TreeNode*Nodearray[],
     int length)
{
    if(Paralignment=="justified")
       file << "<heading level=\"" << head+1<<"\">" << endl;
    else
        file << "<heading level=\"" << head+1 << "\"
        alignment=" << Paralignment << ">" << endl;
    DoParagraph(Nodearray,length);
    file << "</heading>" << endl;
    head=-1;
}
```

Figure 49: The function **MakeHeading()**

A heading is a special paragraph. By the DTD for the *translator* (shown in Figure 44), a heading contains the element *text* as a paragraph does. Unlike a paragraph, the font size for each text in a heading is uniform, regulated by the **N** in **\outlinelevelN**. And all the font sizes stored in each text node are ignored. To do this in heading text generation function **MakeHeading()**, shown in Figure 49, the string "`<text>`" is printed out no matter what the font size is. On the contrary, the regular text generation for a paragraph takes action on collecting the font size in each text node (see **MakeText()** function).

5.3.7 Field Generation

The identification tag for a field is **NT_FIELD** and it is a slave one. Like a text node, the traversed field node is stored in the node array **Nodearray** before the field generation is

done. The field generation function **MakeField()**, shown in Figure 50, is called when its parent entity (a paragraph, a heading, or an item) is generated.

```
void TranslatorTree::MakeField (TreeNode* node)
{
    int defaultSize=0;//default size of font
    FieldNode* temp=(FieldNode*)node;
    string ref=field[temp->GetID()-1];
    for(unsigned int i=0;i<ref.length();i++)
    {
        if(ref.at(i)=='"')//is not a quotation mark
        ref.erase(i,1);
    }
    GetFieldName(ref);
}
```

Figure 50: The function **MakeField()**

5.3.8 Bookmark Generation

The identification tag for a bookmark is **NT_FIELD** and it is a slave one. Like other elements (*text* and *field*) of a paragraph, *bookmark* generation has the similar model as them. The bookmark generation function **MakeBookMark()** is shown in Figure 51.

```
void TranslatorTree::MakeBookMark (TreeNode* node)
{
    BookMarkNode* temp=(BookMarkNode*)node;
    string bookName=bookmark[temp->GetID()-1];
    file << "<bookmark name=\""
         << bookName << "\"></bookmark>" << endl;
}
```

Figure 51: The function **MakeBookMark()**

5.3.9 List Generation

The identification tag for a list is a conditional one consisting of a group of control words, a Boolean variable, and a condition:

- **\pnlvlbody**: to indicate an ordered list.
- **\pnlvlblt**: to indicate an unordered list.
- **\lsN**: to give the indent distance of an indented item between the previous item.
- **\par**: to indicate the start of a new paragraph.
- **itemGoOn**: a Boolean variable to indicate item making process.
- Condition - items are indented?: to determine a new list.

The string "<list>" is generated only when either **\pnlvlbody** or **\pnlvlblt** and **\par** are met, the Boolean variable **itemGoOn** is true, and the indent distance of the current item is greater than that of the previous item.

The indent distance of the current item is set to zero by default before a list is met. When the first list comes in, the **N** in **\lsN** is normally 1 which is greater than the default value 0. The string "</list>" is generated only when either **\pnlvlbody** or **\pnlvlblt** and **\par** are met, and the Boolean variable **itemGoOn** is true, and the indent distance of the current item is less than that of the previous item.

5.3.10 Item Generation

The identification tag for an item is **\par** and a Boolean variable **itemGoOn**. So, it is a conditional identification tag. The **itemGoOn** is set to true when either **\pnlvlbody** or **\pnlvlblt** is met. When **\par** is identified and **itemGoOn** is true, the item generation function **MakeItem()**, shown in Figure 52, is called.

```
void TranslatorTree::MakeItem(TreeNode*
      Nodearray[],int length)
{
    file << "<item>" << endl;
    DoParagraph(Nodearray,length);
    file << "</item>" << endl;
}
```

Figure 52: The function **MakeItem()**

5.3.11 Font Generation

There are eight font attributes recognized by the translator. Font generation takes a master-slave pattern. The master of it is the function **MakeText()** which makes a text entity. But a font does not have corresponding identification tags. The reason is that the font attributes have been already parsed and loaded into an attribute table in a **TextNode** during the first traversal (see translation rules on font and paragraph in section 3.2). When font code is ready for generation, the font attributes are withdrawn from the text node.

The style of font code has a special format compared to other elements. All font attributes should be listed in a line although they have a child-parent relationship. All opening XML control words for font are inserted before a plain text and all closing control words are inserted after the text. "`<italic><bold><sub>a text</sub></bold></italic>`", for example, is the form in expressing font in an XML document. The function **GetFontList()**, shown in Figure 53, is responsible for font generation. The string **stringBefore** is used to store the opening font control words. The string **stringAfter** is used to store the closing font control words. A plain font list **fontList** is produced by calling **TakingFont()**. Then opening control words and closing control words are generated, respectively. Finally, **stringBefore**, **textString** (a string of the plain text), and **stringafter** are combined together.

```
string TranslatorTree:: GetFontList(string
      textString,TextNode* node)
{
    string fontList[10];//to store all font strings
    string stringAfter="";
    string stringBefore="";
    int countFont=0;//to count the font number
    int const fontStart=0;
    int const fontEnd=7;
    for(int i=fontStart;i<fontEnd+1;i++)
    {  //store all fonts in a list
        if(node->t_table[i]!=0)
            fontList[countFont++]=TakeFont(i);
    }
    for(int k=0;k<countFont;k++)
    {  // do a string before the text
        stringBefore.append("<");
        stringBefore.append(fontList[k]);
        stringBefore.append(">");
      }
    for(int m=countFont-1;m>-1;m--)
    {  // do a string after the text
        stringAfter.append("</");
        stringAfter.append(fontList[m]);
        stringAfter.append(">");
    }
    stringBefore.append(textString);
    return stringBefore.append(stringAfter);
}
```

Figure 53: The function **`GetFontList()`**

5.4 Display the Translated XML

There are basically three options to display an XML document. One is to use eXtensible Stylesheet Language (XSL) to transform to a HTML document at first and then to browse it with a regular web browser. The second option is to use an XSL display engine defined

by an XSL style sheet to display the XML document. The third one is to browse the XML document directly with an XML-enable browser defined by Cascading Style Sheets (CSS). Two options are involved with style sheets. A style sheet is a language specification which defines a simple grammar, or language. The grammar specifies what types of the statement can be made within a style sheet. The three options are shown in Figure 54.

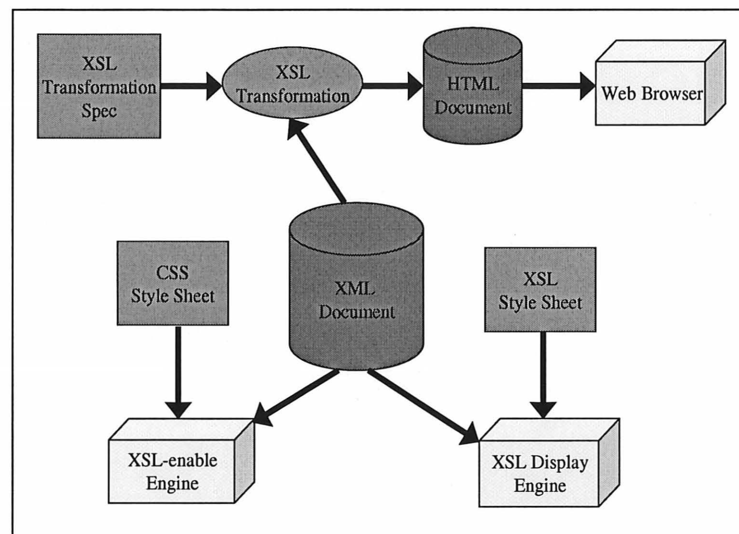


Figure 54: Options to display XML

XSL is the preferred style sheet language of XML [19]. XSL can be used to define how an XML file should be displayed by transforming the XML file into a format that is recognizable by a browser. One such format is HTML. Normally XSL does this by transforming each XML element into an element of HTML or other formats.

XSL consists of three parts: XSL Transformations (XSLT), a language for transforming XML documents; XML Path Language (XPath), a language for defining parts of an XML document; and XSL Formatting Objects (XSLFO), a vocabulary for formatting XML documents. XSLT uses XPath to define the matching patterns for transformations.

An XML document can be represented as a tree view of nodes (very similar to the tree view of folders). XPath uses a pattern expression to identify nodes in the XML document. An XPath pattern is a slash-separated list of child element names that describe a path through the XML document. The pattern selects elements that match the path.

The first option (with XSLT) is chosen in this thesis to transform and display the translated XML document into a HTML document. The XSLT document for the *translator* called `translator.xsl` is shown in Appendix B. With `translator.xsl`, any XML documents defined by the DTD for the *translator* can be transformed into HTML documents by including a processing instruction:

```
<?xml-stylesheet type="text/xsl" href="translator.xsl" media="screen"?>
```

just after the XML version declaration. The instruction means that the type is text/xsl (to link to an XSLT file, with the reference name `translator.xsl`) and media for display is screen.

The first line of an XSLT document is always about the version of XML. In the *translator* XSLT, version 1.0 is adopted by a command `<?xml version="1.0"?>`. The root element of the style sheet is the `<xsl:stylesheet>`. By the mechanism of the namespaces and schemas in XMLT Specification [20], any element beginning with the prefix `xsl:` is part of the XSL vocabulary. The only attribute of `<xsl:stylesheet>` is `xmlns:xsl`, which holds the namespace `http://www.w3.org/TR/WD-xsl` for the XSL transformation recommendation.

The `<stylesheet>` element contains 19 templates, each of which is nested within the `<template>` element, which is actually `<xsl:template>` in the style sheet because the namespace has been included. The `<template>` element has an attribute called `match`. The value of this attribute is a pattern that matches the node of the tree that the template should be applied to, in the form of an XPath expression.

5.4.1 Root Node Match

The XML processor starts in the source document tree at the root node. A template is looked for in the style sheet that matches the root node. The root node is not the `<document>` element in the translated XML, but rather the XML document itself. The root of a document can be represented by a forward slash (/) symbol. The code for the root node match is shown in Figure 55.

```
<xsl:template match="/">
    <html xmlns="http://www.w3.org/TR/xhtml1/strict">
        <head>
            <title>Translation from XML to HTML</title>
        </head>
        <body>
            <xsl:apply-templates select="document/*"/>
        </body>
    </html>
</xsl:template>
```

Figure 55: The code for the root node match

This template like other templates in the *translator* program takes the `match` attribute, which specifies the element to which the template should be applied. This is the selector. In this case, its value is the XML document's root element. Within this template, the resulting HTML document starts to produce.

The instruction `<xsl:apply-templates select="document/*"/>` is used to apply any further templates specified at this position. The XPath `"document/*"` means that all templates for the child elements of the `<document>`, which is the parent element of the target XML document, will be matched.

5.4.2 Child Element Nodes Match

The rest of the XSLT of the *translator* is about the child element nodes match. By the DTD for the *translator*, there are 18 elements/components of the `document` in the translated XML document. Therefore 18 corresponding templates are implemented to wait for match of the parsed XML document. Inside of the templates, the matched elements or contents are placed between opening and closing HTML tags. See the XSL file `translator.xsl` in Appendix B for more details.

5.4.3 Display `XMLoutput.xml`

An XML document `XMLoutput.xml`, shown in Appendix C, is used to demonstrate how to transform the XML document into a HTML document with the XSLT file `translator.xsl`. Assume an XML processor used to parse the XML document. The root node of the XML document is processed at first. The root node is the XML itself. Therefore the first line of `translator.xsl` comes. After XML version 1.0 is declared, the second line

```
<?xml-stylesheet type="text/xsl" href="translator.xsl" media="screen"?>
```

instructs that the stylesheet is XSL type and it refers to `translator.xsl` and the output can be seen by the screen media.

The first template is for the root node match in `translator.xsl`. The command `<xsl:apply-templates select="document/*"/>` is inserted between HTML tags `<body>` and `</body>` (see Figure 55). Therefore all children of the *document* in `XMLoutput.xml` should be applied as HTML body. The first child of the *document* is a heading. So the heading expression, shown in Figure 56, is then processed.

```
<heading level="1" alignment="center">
    <text><bold>Display an XML</bold></text>
</heading>
```

Figure 56: The first heading expression in `XMLoutput.xml`

The *heading* template in `translator.xsl` supplies the heading definition (see Appendix B). A heading has two attributes. The attribute *level* is responsible for the level of the heading. Six levels of a heading (from 1 to 6) are set in the DTD for the *translator*. The tags `h1` to `h6` are the HTML tags for heading level generation. Another attribute is *alignment* which decides the alignment feature of the heading. The command

```
<xsl:attribute name="align">
    <xsl:value-of select="@alignment"/>
</xsl:attribute>
```

in `translator.xsl` picks up the value of *alignment* (an XML attribute) and assigns the value into `align` (a HTML attribute) of `<h>`. Therefore, the first heading in `XMLoutput.xml` is transformed into HTML code so far

```
<h1 align="center">
    elements of the heading
</h1>
```

In this step, the content *`elements of the heading`* between `<h1>` and `</h1>` are not completely converted. The content code will be produced by matching other templates. The instruction `<xsl:apply-templates select = "./*">` in the heading template means applying all the child elements of the current node, which is the heading, between the XML tags `<h1 align="center">` and `</h1>`. The heading, shown in Figure 56, has only one element: *text*. The *text* template specifies the text definition (see Appendix B). Similarly, all elements of the text are embedded between a pair of HTML tags which are `<font>` and `</font>`. Although there is the *size* attribute for the font in the template, the attribute is jumped because the text of the heading (see Figure 56) does not have a size value inside. The XSLT instruction `<xsl:value-of select= "text()"/>` after

`<xsl:apply-templates select="./*">` applies the PCDATA (plain text) of the current node. Because there is no PCDATA in the text node, nothing is actually applied. In this step, the transformed HTML code is:

```
<h1 align="center">
     <font>
          elements of the text
     </font>
</h1>
```

The element *text* contains only one child element *bold*, which includes a PCDATA. By the *bold* template (see Appendix B), the transformed HTML code in this step is:

```
<h1 align="center">
     <font>
          <strong>
               elements of the bold
               PCDATA of the bold
          </strong>
     </font>
</h1>
```

Although a bold may contain further elements by the DTD for the *translator*, the bold in the first heading (see Appendix B) does not have any element but it contains a PCDATA "Display an XML". In this step a completed HTML code for the first heading is produced:

```
<h1 align="center">
     <font>
          <strong>
                    Display an XML
          </strong>
     </font>
</h1>
```

The first element *heading* is transformed into HTML. The other child elements of the *document* in `XMLoutput.xml` are applied by matching the related templates as the same way as `heading` transformation. The whole HTML document browsed by *Internet Explorer 5.0* is shown in Appendix C.

Conclusion

This thesis demonstrates how to design and implement an intelligent translator converting RTF to XML. It is an application of text processing and transformation domain. The translation process covers stages of character reading, lexical analysis, syntax analysis, semantic analysis, code optimization, and code generation. The advantages of the *translator* are its intelligence, efficiency, and extensibility. The main limits of the system are its dependency on the version of *Microsoft Office* and some assumptions in enacting translation rules.

The goal of the project is to implement an intelligent translator. The intelligence comes from the XML representation of the output of the *translator* and code optimization.

The *translator* can convert a RTF document into an XML document which can be transparently exchanged bewteen different systems and applications. For this reason, the *translator* could be widely applied in format transformation with a specific format generator and an XSL document. Since XML is known for its separation of content and style, to deal with XML leads to more convenience and fewer errors. Users of the *translator* could appreciate such intelligence.

Another effort we put for intelligence is to optimize the XML code. Code optimization is to improve the target code generation during the second traversal of the binary data tree by applying the simplification rules. There are two kinds of code optimization in the *translator* program before the XML code is generated. One is realized by jumping the redundant or unnecessary blocks which contain useless information during the third traversal for code generation. Another is to merge two or more texts which hold the same attributes.

The second advantage of the *translator* is efficiency. The efficiency is achieved by choosing a compact package of the RTF control words for the translation, using some facilities supplied by the standard C++ library, and adopting compulsory translation rules.

The compact package of the RTF control words avoids a lot of computing on translation. All RTF control words defined by RTF Specification version 1.6 are divided into four categories in this thesis. Only *the most important RTF control words category*, which contains 95 control words, is recognized and processed in translation. They are key control words for fonts, paragraphs, lists, tables, and images which can basically control the layout and appearance of a document written in the target format. Other control words are either ignored or set by default. Generally speaking, the more control words are included, the more nodes are constructed in the data tree. In addition, the less data is collected and the fewer data structures are implemented. By throwing hundreds of RTF control words, the *translator* program becomes efficient. Although our proach is theorically an efficient way, we do not test it yet by comparison.

Application of standard facilities also improves the efficiency of the *translator* program. The program uses some facilities in the standard C++ library. For example, all the strings in the program are defined as standard strings. Also the data structure for constructing the data tree and the data structure for the second traversal are both implemented with a

standard C++ stack. Since the standard facilities allow us to create a direct, nuts-and-bolts approach to solving problems, using the standard C++ library leads to smaller source code, more efficient coding, and faster execution.

Compulsory translation rules are the last factor resulting in the efficiency of the *translator*. For example, we set up some compulsory rules on list translation: no choice for bullet icon and the numbering style; setting item alignment as left aligned. Such design excludes extra control words in the data tree and avoids much execution on dealing with unimportant translation cases.

The third advantage of the *translator* is extensibility. The translated XML code is designed to keep the style or format of the source document as exactly as possible. Some data kept in an XML document seem to be useless in transforming XML into a HTML document, but they may be necessary in covering other formats. There are three kinds of data remaining for future extension. The first is the percentage of compression rate on the width/height of the original image. The second is the position of the right border of a cell. The third is the empty paragraph.

However, the translator has some limitations with respect to our initial goal. The goal is to design and implement a universal intelligent translator. This tool is supposed to be used in many environments with robust features. Unfortunately, we find that this goal cannot be realized at the end of implementation.

The first reason for the limitations is that the *translator* heavily depends on the version of *Microsoft Office*. As a component of the translator system, the *translator* takes the output of the *extractor* as its own input, and the *extractor* cannot always successfully extract a uniform format of contents or some important control words at all with different versions of *Microsoft Office*. For example, the list control words **\pnlvlbody** and

\pnlvlblt cannot be extracted from a list created with either *Microsoft Office 2000* or *Microsoft Office 2002*. Therefore the translator is affected by this problem. In some case, the *translator* cannot translate a correct target document, missing some contents of the source document, for example, a list expression. So, the translator system has to choose *Microsoft Office 1997* (French version) as its only RTF converting tool which is used to convert a `.doc` document into a RTF document. The *translator* can work well with this version. Other *Microsoft Office* versions may result in problems in generation of lists and some special fields.

The second reason for the limitations comes from assumptions that the `.doc` document is created with a common sense. For example, a cell is assumed not to contain images, lists, fields, bookmarks, and tables. Another example is that an item has a paragraph only. The translation rules on tables and lists are based on these assumptions. If users intentionally or unintentionally create a `.doc` document with an unusual way beyond what we expect, the translation would fail. In the worse case, this problem is fatal and leads to failure on translation because some XML tags cannot be properly produced.

The third reason for the limitations comes from the small number of control words which are recognizable by the *translator*. Everything in the world has two faces. On the one hand, the compact package of the RTF control words benefits the *translator* in efficiency. On the other hand, the design may not result in a translation of source documents in every deep aspect. Some information may be lost. For example, the bullet type of a bullet list cannot be kept as the original one. Also the border sizes of each cell in the original table are not kept either. Therefore, some subtle contents of the source document may not be translated with the *translator*. Although the compact package is our choice for purpose of efficiency and easy implementation, the *translator* has limitations on application in high quality translation.

For the three reasons, the *translator* is not a universal tool. Actually, we consider it as a prototype of a translator. Although we prepare many test files for the translation, we do not apply a formal method to test the *translator*. There is not an exception catch mechanism in the C++ program. So, it is hard to say the *translator* program is robust. It is the main weakness of the *translator*. Also, the design of the program does not follow the state of the art quite well.

The translator could be eventually marketed if some further steps are done. The first step is to resolve the compatibility problem on different versions of *Microsoft Office*. If the problem cannot be removed, two versions of the translator should be separately implemented for different *Microsoft Office* versions. One version is for *Microsoft Office 1997* and the other for *Microsoft Office 2000* and upwards. The second step is to throw away the assumption on the translation rules on tables and lists. All situations are allowed in creation of a `.doc` document even if it is in an unusual and a non-standard style. The reason to do so is to allow the *translator* to be a bug-free program. With the current *translator*, the translation would fail on the unusual document. At least, a mechanism should be set to alarm users that such document may not successfully be translated. The third is to apply formal test methods and use more programming standards in order to implement a more robust translator.

Appendix A

C++ Code in the *Translator*

The *translator* is written in C++. The classes `Translator.cpp` and `TreeNode.cpp` as well as their header files compose the translator project that implements the *translator*. Classes for the seven kinds of tree nodes are also included in the class `TreeNode.cpp`. However, two header files: `Translator.h` and `TreeNode.h`, are not listed in Appendix A for saving space.

A.1 The class `Translator.cpp`

```
#include "translator.h"
#include <string>
using namespace std;

/***************************************************************************
**
**        class Translator implementation
```

```
**
****************************************************************************/
Translator::Translator()
{
   m_tree=NULL;
   initialTable();
   meetTBFI=false;

   num=13;                        //number of attributes in the font table
   previousText=NULL;             //default
   meetField=false;               //to show if to meet a field
   head=-1;                       //initialize head
   int user_choice;               //choice for output
   coutDataNode=0;                //NO. of text/field/bookmark in a paragraph
   fieldEnd=false;                //to show if to need to print out end of a field
   Paralignment="justified";      //default value of alignment for a paragraph
   meetTF=false;                  //to show if a paragraph has texts or fields
   makeList=false;                //to show if to begin to make a list
   itemGoOn=false;                //if still to go on to make an item
   previousIndent=-1;             //default value of indentation of a list
   countList=0;                   //default value of number of lists
   countCell=0;                   //to record the number of cells in a row
   countText=0;                   //to count number of texts in the textArray
   cellOrder=0;                   //to show order of cells (first or second)
   addTextTable=false;            //to show if add text to an array for table generation
   doRow=false;                   //to show if row generation is doing
   doTable=false;                 //to show if table generation is doing
   TextArray[0]=new TextNode;
   cout << "To display XML as HTML, type 1. Input other to get an XML file " << endl;
   cin >> user_choice;
   if(user_choice==1)
      file.open("HTMLoutput.xml");
   else
      file.open("XMLoutput.xml");
   file << "<?xml version=\"1.0\" ?>" << "\n";  //initialize file
   if(user_choice==1)
      file << "<?xml-stylesheet type=\"text/xsl\" href=\"translator.xsl\"";
   file << " media=\"screen\"?>" << endl;
   file << "<document>" << "\n";
}

Translator::~Translator()
{
   DestroyTree(m_tree);
}

void Translator::DestroyTree(TreeNode* pParent)
{
   if(pParent==NULL)
      return;
   DestroyTree(pParent->GetLeftSon());
   DestroyTree(pParent->GetRightSon());
   delete pParent;
   pParent=NULL;
}

void Translator:: initialTable()
{
   /* 0-italic,1-bold,2-ul,3-ulw,4-sub,5-super,6-strike,
      7-striked,8-fs,9-qj,10-qc,11-qr,12-ql */
```

```
   for(int i=0; i< num; i++)
      m_table[i]=0;
   m_table[8]=3;//default value of font size
   m_table[9]=1;//default value of alignement is justified
}

SequenceNode* Translator::AddSequenceNode(TreeNode** pParent)
{
   SequenceNode* pNode=new SequenceNode();
   *pParent=(TreeNode*)pNode;
   return pNode;
}

TagNode* Translator::AddTagNode(TreeNode** pParent, string p, int value)
{
   TagNode* pNode=new TagNode();
   If(NonmeetNum==true)
      value=-1;
   pNode->SetTag(p);
   pNode->SetValue(value);
   *pParent=(TreeNode*)pNode;
   NonmeetNum=true;
   return pNode;
}

TextNode* Translator::AddTextNode(TreeNode** pParent, int id)
{
   TextNode* pNode=new TextNode(id);
   *pParent=(TreeNode*)pNode;
   return pNode;
}

FieldNode* Translator::AddFieldNode(TreeNode** pParent, int id)
{
   FieldNode* pNode=new FieldNode(id);
   *pParent =(TreeNode*)pNode;
   return pNode;
}

BookMarkNode* Translator::AddBookMarkNode(TreeNode** pParent, int id)
{
   BookMarkNode* pNode=new BookMarkNode(id);
   *pParent=(TreeNode*)pNode;
   return pNode;
}

ImageNode* Translator::AddImageNode(TreeNode** pParent, string str,int id)
{
   ImageNode* pNode=new ImageNode(id);
   pNode->SetFile(str);
   *pParent=(TreeNode*)pNode;
   return pNode;
}

BlockNode* Translator::AddBlockNode(TreeNode** pParent,int id)
{
   BlockNode* pNode=new BlockNode(id);
   *pParent=(TreeNode*)pNode;
   return pNode;
}
```

```
void Translator::makeDataNode()
{
   string str=token;
   string imageFile;
   bool meetImage=false;
   int pos=token.find_first_of(' ');
   if(pos!=string::npos)
   {
      str=token.substr(0,pos);
      imageFile=token.substr(pos+1);
      if(str=="IMAGE")
      {
         pImage=AddImageNode(&pCurrentNode->m_pLeft,imageFile,ImageID++);
         meetImage=true;
      }
    }
   for(int i=0; i<token.length();i++)
      if(isdigit(token.at(i)))
      {
         NonmeetNum=false;
         str=token.substr(0,i);
         for(int j=i;j<token.length();j++)
            value=value*10+int(token.at(j)-'0');
         break;
      }
   if(str=="TEXT")
      pText=AddTextNode(&pCurrentNode->m_pLeft,value);
   else if(str=="FIELD")
      pField=AddFieldNode(&pCurrentNode->m_pLeft,value);
   else if(str=="BOOKMARK")
      pBookMark=AddBookMarkNode(&pCurrentNode->m_pLeft,value);
   else if(!meetImage)
      pTag=AddTagNode(&pCurrentNode->m_pLeft, str,value);
   NonmeetNum=true;
   pSequence=AddSequenceNode(&pCurrentNode->m_pRight );
   pCurrentNode=pSequence;
   value=0;
   token="";
   meetImage=false;
}

string Translator::CheckIllegal(string line)
{
   int length=line.length();
   for(int i=0;i<length;i++)
   {
      int k=line.at(i);
      cout << "position" << i << "is " << k  << endl;
      if(line.at(i)=='<')   //read illegal XML character "<"
      {
         line.replace(i,1,"&lt;");
         i=i+3;
         length=length+3;
      }
      else if(line.at(i)=='>')   //read illegal XML character ">"
      {
         line.replace(i,1,"&gt;");
```

```
            i=i+3;
            length=length+3;
        }
        else if(line.at(i)=='&')//read illegal XML character "& "
        {
            line.replace(i,1,"&amp;");
            i=i+4;
            length=length+4;
        }
        else if(line.at(i)=='\'')//read illegal XML character "'"
        {
            line.replace(i,1,"&apos;");
            i=i+5;
            length=length+5;
        }
        else if(line.at(i)==-108 || line.at(i)==-109)
        {
            //read illegal XML character "" "
            line.replace(i,1,"&quot;");
            i=i+5;
            length=length+5;
        }
    }//end for loop
    return line;
}

void Translator::ReadText()
{
    int pos=line.find_first_of('|');
    if(pos==string::npos)
        cout << "Error. Lack of separated signs in the text file." << endl;
    else
    {
        string temp=line.substr(pos+2);
        text[indexText++]=CheckIllegal(temp);
    }
}

void Translator::ReadField()
{
    int pos=line.find_first_of('|');
    if(pos==string::npos)
        cout << "Error. Lack of separated signs in the text file." << endl;
    else
    {
        string temp=line.substr(pos+2);
        field[indexField++]=CheckIllegal(temp);
    }
}

void Translator::ReadBookMark()
{
    int pos=line.find_first_of('|');
    if(pos==string::npos)
        cout << "Error. Lack of seperated signs in the text file." << endl;
    else
    {
        string temp=line.substr(pos+2);
        bookmark[indexBook++]=CheckIllegal(temp);
    }
```

```
}


void Translator::ReadData(ifstream &in)
{
   indexText=0;                //number of string in a text array
   indexField=0;               //number of string in a field array
   indexBook=0;                //number of string in a bookmark array
   while(getline(in,line,'\n'))
   {
      //read texts,fields, and book marks from text file
      indexTemp=0;             //index of temp array
      if(line.at(0)=='T')      //read a text
         ReadText();
      If(line.at(0)=='F')      //read a field
         ReadField();
      If(line.at(0)=='B')      //read a book mark
         ReadBookMark();
   }//end while
}

void Translator::ConstructTree(ifstream &inFile)
{
   DestroyTree(m_tree);
   NonmeetNum=true;
   char ch;
   bTag=false;
   index=0;
   value=0;
   int BlockID=1;
   ImageID=1;
   BlockNode* pRoot=new BlockNode(0);
   SequenceNode* child=new SequenceNode();
   pRoot->AddRightSon(NULL);
   pRoot->AddLeftSon(child);
   m_tree=(TreeNode*)pRoot;
   pCurrentNode=(TreeNode*)child;
   while(!inFile.eof())
   {
      getline(inFile,mstr); //read command file line by line
      for(int i=0;i<mstr.length();i++)
      {
         //read the line character by character
         ch=mstr.at(i);
         if(ch=='{')   //Begin Block
         {
            //Push Parent Node To Stack
            treeStack.push( (TreeNode*)pCurrentNode );
            //Add Block Node To Tree
            pBlock=AddBlockNode( &pCurrentNode->m_pLeft,BlockID++ );
            //Add Sequence Node To Tree
            pSequence=AddSequenceNode( &pBlock->m_pLeft );
            pCurrentNode=pSequence;
         }
         else if(ch=='\\')
         {
            int pos=mstr.length();// last token without ending "}" or "\"
            for(int j=i+1;j<mstr.length();j++)
               if(mstr.at(j)=='\\' || mstr.at(j)==')' || mstr.at(j)=='{')
               {
```

```
                    pos=j;
                    break;
                }
            token=mstr.substr(i+1,(pos-i-1));
            makeDataNode();
        }
        else if(ch == '}')    //End Block
        {
            //Pop Parent Node From Stack
            pCurrentNode=(SequenceNode*)treeStack.top();
            treeStack.pop();
            if( pCurrentNode!=pRoot)
            {
                pSequence=AddSequenceNode(&pCurrentNode->m_pRight);
                pCurrentNode=pSequence;
            }
        }
    }//end for loop
  }//end while
}

void Translator::PrintTree(TreeNode* pParent, int cnt)
{
  if(pParent==NULL)
    return;
  TagNode* pTag;
  BlockNode* pBlock;
  TextNode* pText;
  BookMarkNode* pBookMark;
  FieldNode* pField;
  ImageNode* pImage;
  char* p=new char[255];
  for(int i=0; i<cnt; i++)
    cout << '-';
  if(pParent->GetNodeType()==NT_SEQUENCE )
    cout << "Sequence" << endl;
  else if(pParent->GetNodeType()==NT_BLOCK)
  {
    pBlock=(BlockNode*)pParent;
    if(pBlock->GetUse()==true)
      cout << "Block" << pBlock->id << "--use" << endl;
    else
      cout << "Block" << pBlock->id << "--nonuse" << endl;
  }
  else if(pParent->GetNodeType()==NT_TAG)
  {
    pTag = (TagNode*)pParent;
    if(pTag->GetValue()>=0)
      cout << pTag->GetTag() << pTag->GetValue() << endl;
    else
      cout << pTag->GetTag() << endl;
  }
  else if(pParent->GetNodeType() == NT_FIELD)
  {
    pField=(FieldNode*)pParent;
    cout << "FieldNode" << pField->GetID() << endl;
  }
  else if( pParent->GetNodeType()==NT_BOOKMARK)
  {
    pBookMark=(BookMarkNode*)pParent;
```

```
      cout << "BookMarkNode" << pBookMark->GetID() << endl;
   }

   else if(pParent->GetNodeType()==NT_IMAGE)
   {
      pImage=(ImageNode*)pParent;
      cout << "ImageNode" << pImage->GetID() << endl;
   }
   else if(pParent->GetNodeType()==NT_TEXT )
   {
      pText=(TextNode*)pParent;
      int id=pText->GetTextID();
      cout << "TextNode" << id << endl;
      for(int i=0;i<num;i++)
         cout << pText->t_table[i];
      cout << endl;
   }
   delete p;
   PrintTree(pParent->GetLeftSon(), cnt+1);
   PrintTree(pParent->GetRightSon(), cnt+1);
}

void Translator::FirstScan(TreeNode* pRoot,int table[],int length)
{
   if(pRoot !=NULL)
   {
      if(pRoot->GetNodeType()==NT_TAG)
      {
         string str;//a string to store tag
         TagNode* temp;
         temp=(TagNode*)pRoot;
         str=temp->GetTag();
         //do font and paragraph issues
         if(str=="pard")
            SetPard(table);
         if(str=="b" && temp->GetValue()==-1)//not number follows
            SetBold(table);
         if(str=="b" && temp->GetValue()==0) //zero follows
            ResetBold(table);
         if(str=="i" && temp->GetValue()==-1)
            SetItalic(table);
         if(str=="i" && temp->GetValue()==0)
            ResetItalic(table);
         if(str=="ul" && temp->GetValue()==-1)
            SetUnderline(table);
         if(str=="ul" && temp->GetValue()==0)
            ResetUnderline(table);
         if(str=="ulw"&& temp->GetValue()==-1)
            SetUlw(table);
         if(str=="ulw" && temp->GetValue()==0)
            ResetUlw(table);
         if(str=="sub" && temp->GetValue()==-1)
            SetSub(table);
         if(str=="sub" && temp->GetValue()==0)
            ResetSub(table);
         if(str=="super" && temp->GetValue()==-1)
            SetSuper(table);
         if(str=="super" && temp->GetValue()==0)
            ResetSuper(table);
         if(str=="strike" && temp->GetValue()==-1)
```

```
        SetStrike(table);
    if(str=="strike" && temp->GetValue()==0)
        ResetStrike(table);
    if(str=="striked" && temp->GetValue()==1)
        SetStriked(table);
    if(str=="striked" && temp->GetValue()==0)
        ResetStriked(table);
    if(str=="fs" && temp->GetValue()!=-1)
    {
        int fontsize=temp->GetValue()/8; //convert to html unit
        SetOutlevel(table,fontsize);
    }
    if(str=="plain")
        SetPlain(table);
    if(str=="qj")
        SetQJ(table);
    If(str=="qc")
        SetQC(table);
    if(str=="qr")
        SetQR(table);
    if(str=="ql")
        SetQL(table);
    if(str=="picwgoal")
        ImageWidth= temp->GetValue();
    if(str=="pichgoal")
        ImageHeight= temp->GetValue();
    if(str=="picscalex")
        ImageScaleX= temp->GetValue();
    if(str=="picscaley")
        ImageScaleY= temp->GetValue();
    if(str=="fldrslt" && currentSeq->GetRightSon()->GetLeftSon()!=NULL)
        currentField->SetFieldText(true);
}
if(pRoot->GetNodeType()==NT_IMAGE)
{
    ImageNode* pImage;
    pImage=(ImageNode*)pRoot;
    pImage->SetWidth(ImageWidth);
    pImage->SetHeight(ImageHeight);
    pImage->SetScaleX(ImageScaleX);
    pImage->SetScaleY(ImageScaleY);
}
if(pRoot->GetNodeType()==NT_TEXT)
{
    TextNode* pText;
    pText=(TextNode*)pRoot;
    for(int i=0;i<num;i++)
        pText->t_table[i]=table[i];
}
if(pRoot->GetNodeType()==NT_SEQUENCE)
    currentSeq=(SequenceNode*)pRoot;
if(pRoot->GetNodeType()==NT_FIELD)
    currentField=(FieldNode*)pRoot;
if(pRoot->GetNodeType()==NT_BLOCK)
{
      int BlockTable[20]; //a copy of font and paragraph table
    for(int i=0;i<length;i++)
        BlockTable[i]=table[i];
    FirstScan(pRoot->GetLeftSon(), BlockTable,length);
    FirstScan(pRoot->GetRightSon(), BlockTable,length);
```

```
        }
        else//Use previous table to scan further
        {
            FirstScan(pRoot->GetLeftSon(), table,length);
            FirstScan(pRoot->GetRightSon(), table,length);
        }
    }
}

void Translator::SecondScan(TreeNode* pRoot)//for simplificatin rule
{
    if( pRoot==NULL)
        return;
    if(pRoot->GetNodeType()==NT_BLOCK)//set default value of a block to nonuse
        treeStack.push((TreeNode*)pRoot);
    if(pRoot->GetNodeType()==NT_TEXT||pRoot->GetNodeType()==NT_FIELD
        || pRoot->GetNodeType()==NT_BOOKMARK)
    {
        if(pRoot->GetNodeType()==NT_FIELD)
        meetField=true;
        meetTBFI=true;//a block includes text/bookmark/field is regarded use
    }
    if(pRoot->GetNodeType()==NT_IMAGE)
    {
        meetTBFI=true;//a block includes image is regarded use
        previousText=NULL;//don't compare two texts seperated by an image
    }
    if(pRoot->GetNodeType()==NT_TAG)
    {
        TagNode* temp=(TagNode*)pRoot;
        //a block includes control words for list, table, or paragraph
        if(temp->GetTag()=="pnlvlblt" || temp->GetTag()=="pnlvlbody" ||
            temp->GetTag()=="cell" || temp->GetTag()=="row" || temp->GetTag()=="par")
        {
            meetTBFI=true;
            if(temp->GetTag()=="par")//in a new paragraph, start from bottom
                meetField=false;
        }
        //two tags in different cells or paragraphs or seperated by a field
        if(temp->GetTag()=="cell" || temp->GetTag()=="par"
            || temp->GetTag()=="fldinst" || temp->GetTag()=="picwgoal")
            previousText=NULL;
    }//end tag
    if(pRoot->GetNodeType()== NT_TEXT)
    {
        TextNode* tempText=(TextNode*)pRoot;
        if(previousText==NULL)//no any text can be compared
            previousText=tempText;
        else//compare two texts
        {
            if(meetField)//do not compare field name and the next text
            {
                meetField=false;
                previousText=NULL;
            }
            else if(tempText->CompareText(previousText,tempText)==true)
            {
                previousText->merge=true;
                tempText->merge=true;
            }
```

```
            previousText=tempText;
        }
    }//end text
    if(pRoot->GetNodeType()==NT_SEQUENCE &&pRoot->GetLeftSon()==NULL)
    {
        //ends a block reading and signs its use or unuse flag
        BlockNode* block=(BlockNode*)treeStack.top();
        treeStack.pop();
        if(meetTBFI==true)
        {
            block->SetUse(true);
            meetTBFI=false;
        }
        else if(block->GetUse()==false)
            block->SetUse(false);
        if(block->GetUse()==true && treeStack.size()>0)//set its parent to true
        {
            BlockNode* b=(BlockNode*)treeStack.top();
            b->SetUse(true);
        }
    } //for ending block
    SecondScan(pRoot->GetLeftSon());
    SecondScan(pRoot->GetRightSon());
}

void Translator::ThirdScan(TreeNode* pRoot)
{
    if(pRoot==NULL)
        return;
    if(pRoot->GetNodeType()==NT_BLOCK )
        //jump if the block is useless
        if(((BlockNode*)pRoot)->GetUse()==false)
            return;
    if(pRoot->GetNodeType()==NT_TEXT)
    {
        TextNode* temp=(TextNode*)pRoot;
        if(addTextTable==true)//add text into an array for table generation
            TextArray[countText++]=temp;
        else
        {
            meetTF=true;
            Paralignment=GetAlignment(temp);
            NodeArray[coutDataNode++]=pRoot;//store this text
        }
    }
    if(pRoot->GetNodeType()==NT_FIELD)
    {
        meetTF=true;
        NodeArray[coutDataNode++]=pRoot;//store this FIELD
    }
    if(pRoot->GetNodeType()==NT_BOOKMARK)
    {
        NodeArray[coutDataNode++]=pRoot;//store this BOOKMARK
    }
    if(pRoot->GetNodeType()==NT_IMAGE)
    {
        MakeImage((ImageNode*)pRoot);
    }
    if(pRoot->GetNodeType()==NT_TAG)
    {
```

```
TagNode* temp=(TagNode*)pRoot;
if(temp->GetTag()=="outlinelevel")//meet heading
   head=temp->GetValue();
//----make a table title or end a list before a table-------------------//
if(temp->GetTag()=="trowd")
   addTextTable=true;
//-------------------do row issue---------------------------------------//
if(temp->GetTag()=="row")
{
   doRow=false;
    cellOrder=0;
    countCell=0;
    file << "</row>" << endl;
}
if(temp->GetTag()=="cellx")
   CellPosition[countCell++]=temp->GetValue();
If(temp->GetTag()=="cell")
{
    //end the list when a table directly follows without anything between
    if(makeList==true)
    {
       for(int i=countList-1;i>-1;i--)
       {
          file << "</list>" << endl;
       }
        makeList=false;
        countList=0;
        previousIndent=-1;//default value of indentation of a list
    }
    if(doTable==false)//first time to do table
    {
        doTable=true;
        file << "<table>" << endl;
    }
    if(doRow==false)//the first cell of a row
    {
        doRow=true;
        file << "<row>" << endl;
    }
   MakeCell(TextArray,countText,CellPosition[cellOrder++]);
   countText=0;
}
//----------------do new paragragh and ends case -----------------------//
if(temp->GetTag()=="par" && itemGoOn==false )
{
   if(doRow==false && doTable==true)//end a table
    {
      doTable=false;
      addTextTable=false;
      file << "</table>" << endl;
    }
    if(makeList==true)//end the list
    {
      for(int i=countList-1;i>-1;i--)
         file << "</list>" << endl;
      makeList=false;
      countList=0;
      previousIndent=-1;//default value of indentation of a list
    }
    //do normal paragraph task
```

```
      MakePar(NodeArray,coutDataNode);//to make a paragraph
      coutDataNode=0;//reset the cout
   }
   if(temp->GetTag()=="picwgoal")
   {
      //end a list when an image directly follows without anything bewteen
      if(makeList==true)
      {
         for(int i=countList-1;i>-1;i--)
            file << "</list>" << endl;
         makeList=false;
         countList=0;
         previousIndent=-1;//default value of indentation of a list
      }
      //end a table which directly follows an image with nothing seperated
      if(doTable==true)
      {
         doTable=false;
         file << "</table>" << endl;
      }
   }
   //------do list issue and end a table directly before the list----------//
   if(temp->GetTag()=="pnlvlbody")
   {
      makeList=true;
      itemGoOn=true;
      orderList=true;
      if(doTable==true)
      {
         //end a table which directly follows a list with nothing seperated
         doTable=false;
         file << "</table>" << endl;
      }
   }
   if(temp->GetTag()=="pnlvlblt")
   {
      makeList=true;
      itemGoOn=true;
      orderList=false;
      if(doTable==true)
      {
         //end a table which directly follows a list with nothing seperated
         doTable=false;
         file << "</table>" << endl;
      }
   }
   //--------------------------do list issue-------------------------------//
   if(temp->GetTag()=="ls")
      currentIndent=temp->GetValue();
   //to make a list
   if(temp->GetTag()=="par" && itemGoOn==true)
   {
      itemGoOn=false;
      //asume \ls should be before \par otherwise program cannot work
      if(currentIndent==previousIndent)//do an item only
      {
         MakeItem(NodeArray,coutDataNode);//to make an item
         coutDataNode=0;//reset the cout
      }
      if(currentIndent>previousIndent)//do a new list
```

```
            {
               countList++;//increase the number of lists

               if(orderList)
                  file << "<list type=\"order\">" << endl;
               else
                  file << "<list type=\"unorder\">" << endl;
               MakeItem(NodeArray,coutDataNode);//to make an item
               coutDataNode=0;//reset the cout
            }
            if(currentIndent<previousIndent)//nested list ends
            {
               file << "</list>" << endl;
               countList--;//decrease the number of lists
               MakeItem(NodeArray,coutDataNode);
               coutDataNode=0;//reset the cout
            }
            previousIndent=currentIndent;
         }
      }
   ThirdScan(pRoot->GetLeftSon());
   ThirdScan(pRoot->GetRightSon());
}

void Translator::MakeCell(TextNode* TextArray[],int countText,int Position)
{
   string alignment=GetAlignment(TextArray[0]);
   if(alignment=="justified")
      file << "<cell position=\"" << Position << "\">" << endl;
   else
      file << "<cell alignment=" << alignment <<" position=\"" << Position << "\">"
           << endl;
   for(int i=0;i<countText;i++)
      MakeText(text[TextArray[i]->GetTextID()-1],TextArray[i]);
   file << "</cell>" << endl;
}

void Translator::MakeImage(ImageNode* image)
{
   string source=image->GetFile();
   int width=image->GetWidth()*0.063;//roughly convert to html unit
   int height=image->GetHeight()*0.063;//roughly convert to html unit
   int scaleX=image->GetScaleX();
   int scaleY=image->GetScaleY();
   file << "<image source=\"" << source << "\" width=" << "\"" << width << "\" height=\""
       << height << "\" scaleX=\"" << scaleX << "\" scaleY=\"" << scaleY << "\">" <<endl;
   file << "</image>" << endl;
}

void Translator::SetPlain(int m_table[])
{
   /* 0-italic,1-bold,2-ul,3-ulw,4-sub,5-super,6-strike,7-striked,
      8-fs,9-qj,10-qc,11-qr,12-ql */
   //reset all font to zero
   m_table[0]=0;
   m_table[1]=0;
   m_table[2]=0;
   m_table[3]=0;
   m_table[4]=0;
   m_table[5]=0;
```

```
    m_table[6]=0;
    m_table[7]=0;
}
void Translator::SetItalic(int m_table[])
{
    m_table[0]=1;
}

void Translator::ResetItalic(int m_table[])
{
    m_table[0]=0;
}

void Translator::SetBold(int m_table[])
{
    m_table[1]=1;
}

void Translator::ResetBold(int m_table[])
{
    m_table[1]=0;
}

void Translator::SetUnderline(int m_table[])
{
    m_table[2]=1;
}

void Translator::ResetUnderline(int m_table[])
{
    m_table[2]=0;
}

void Translator::SetUlw(int m_table[])
{
    m_table[3]=1;
}

void Translator::ResetUlw(int m_table[])
{
    m_table[3]=0;
}

void Translator::SetSub(int m_table[])
{
    m_table[4]=1;
}

void Translator::ResetSub(int m_table[])
{
    m_table[4]=0;
}

void Translator::SetSuper(int m_table[])
{
    m_table[5]=1;
}

void Translator::ResetSuper(int m_table[])
{
```

```
    m_table[5]=0;
}

void Translator::SetStrike(int m_table[])
{
    m_table[6]=1;
}


void Translator::ResetStrike(int m_table[])
{
    m_table[6]=0;
}

void Translator::SetStriked(int m_table[])
{
    m_table[7]=1;
}

void Translator::ResetStriked(int m_table[])
{
    m_table[7]=0;
}

void Translator::SetOutlevel(int m_table[],int level)
{
    m_table[8]=level;
}

void Translator::SetQJ(int m_table[])
{
    m_table[9]=1;
    m_table[10]=0;
    m_table[11]=0;
    m_table[12]=0;
}

void Translator::SetQC(int m_table[])
{
    m_table[9]=0;
    m_table[10]=1;
    m_table[11]=0;
    m_table[12]=0;
}

void Translator::SetQR(int m_table[])
{
    m_table[9]=0;
    m_table[10]=0;
    m_table[11]=1;
    m_table[12]=0;
}

void Translator::SetQL(int m_table[])
{
    m_table[9]=0;
    m_table[10]=0;
    m_table[11]=0;
    m_table[12]=1;
}
```

```
void Translator::SetPard(int m_table[])
{
   m_table[9]=0;
   m_table[10]=0;
   m_table[11]=0;
   m_table[12]=0;
}

void Translator:: MakeItem(TreeNode* Nodearray[],int length)
{
   file << "<item>" << endl;
   DoParagraph(Nodearray,length);
   file << "</item>" << endl;
}

void Translator::DoParagraph(TreeNode* Nodearray[],int length)
{
   string mergedString;      //a string to store merged string
   int mergedFontSize;       //the common font size
   string mergedFont;        //the common font attributes
   mergedString=="";         //initialization
   bool mergeFlag=false;     //to show a merging process
   TextNode* temp;
   for(int i=0;i<length;i++)
   {
      if(Nodearray[i]->GetNodeType()==NT_BOOKMARK)
         MakeBookMark(Nodearray[i]);
      if(Nodearray[i]->GetNodeType()==NT_FIELD)
      {
         FieldNode* field=(FieldNode*)Nodearray[i];
         //when merged texts before a field
         if(mergeFlag==true)
         {
            MakeText(mergedString,temp);
            mergedString=="";//reset for the next pair merged texts
            mergeFlag=false;
         }
         MakeField(Nodearray[i]);
         if(field->GetFieldText()==false)
         {
            file << "</field>" << endl;
            fieldEnd=false;
         }
         else fieldEnd=true;
      }
      if(Nodearray[i]->GetNodeType()==NT_TEXT)
      {
         temp=(TextNode*)Nodearray[i];
         //when two or more texts can do merge
         if(temp->GetMerge()==false && mergeFlag==true)
         {
            MakeText(mergedString,Nodearray[i-1]);//bring previous text's fonts
            mergedString="";//reset for the next pair merged texts
            mergeFlag=false;
         }
         if(temp->GetMerge())//a merged text comes in
         {
            if(mergedString!="")
```

```
                mergedString.append(" ");//add one space between
                mergedString.append(text[temp->GetTextID()-1]);
                mergedFontSize=GetFontSize(temp);
                mergedFont=GetFont(temp);
                mergeFlag=true;
            }
            //when two merged strings in the last of Nodearray
            if(mergeFlag==true && i==length-1)
            {
                MakeText(mergedString,Nodearray[i-1]);//bring previous text's fonts
                mergeFlag=false;
            }
            if(temp->GetMerge()==false)//do normal text generation
            {
                MakeText(text[temp->GetTextID()-1],Nodearray[i]);
                mergeFlag=false;
            }
            if(fieldEnd==true)//print after field ending
            {
                file << "</field>" << endl;
                fieldEnd=false;
            }
        }
    }
}

void Translator::MakeHead(TreeNode* Nodearray[],int length)
{
    if(Paralignment=="justified")
        file << "<heading level=\"" << head+1 << "\">" << endl;
    else
        file <<"<heading level=\"" << head+1 << "\" alignment=" << Paralignment << ">"
             << endl;
    DoParagraph(Nodearray,length);
    file << "</heading>" << endl;
    head=-1;
}

void Translator:: MakePar(TreeNode* Nodearray[],int length)
{
    if(head!=-1)//to make a header
        MakeHead(Nodearray,length);
    //empty paragraph or default alignment
    else
    {
        if(meetTF==false||Paralignment=="justified")
            file << "<paragraph>" << endl;
        if(meetTF==true && Paralignment!="justified")
            file << "<paragraph alignment=" << Paralignment << ">" << endl;
        DoParagraph(Nodearray,length);
        file << "</paragraph>" << endl;
    }
}

void Translator::MakeBookMark (TreeNode* node)
{
    BookMarkNode* temp=(BookMarkNode*)node;
    string bookName=bookmark[temp->GetID()-1];
    file << "<bookmark name=\"" << bookName << "\"></bookmark>" << endl;
}
```

```
void Translator::MakeText(string textString,TreeNode* node)
{
   TextNode* temp=(TextNode*)node;
   int fontSize=GetFontSize(temp);
   string fontList=GetFont(temp);
   int defaultSize=3;//default size of font is 3 in html
   if(head!=-1)//make heading
      file << "<text>" << GetFontList(textString,temp) << "</text>" << endl;
   else
   {
      //make normal paragraph
      if(fontSize==defaultSize && fontList=="plain")
         file << "<text>" << textString << "</text>" << endl;
      if(fontSize!=defaultSize && fontList=="plain")
          file << "<text size=\"" << fontSize << "\">" <<textString <<"</text> << endl;
      if(fontSize==defaultSize && fontList!="plain")
         file << "<text>" << GetFontList(textString,temp) << "</text>" << endl;
      if(fontSize!=defaultSize && fontList!="plain")
         file << "<text size=\"" << fontSize << "\">" << GetFontList(textString,temp)
             << "</text>" << endl;
   }
}

string Translator:: GetFontList(string textString,TextNode* node)
{
   string fontList[10];//to store all font strings
   string stringAfter="";//a font string after "text"
   string stringBefore="";//a font string before "text"
   int countFont=0;//to count the font number
   int const fontStart=0;
   int const fontEnd=7;
   //store all fonts in a list
   for(int i=fontStart;i<fontEnd+1;i++)
   {
      if(node->t_table[i]!=0)
         fontList[countFont++]=TakeFont(i);
   }
   // do a string before the text
   for(int k=0;k<countFont;k++)
   {
      stringBefore.append("<");
      stringBefore.append(fontList[k]);
      stringBefore.append(">");
   }
   // do a string after the text
   for(int m=countFont-1;m>-1;m--)
   {
      stringAfter.append("</");
      stringAfter.append(fontList[m]);
      stringAfter.append(">");
   }
   //combine text and the two strings together
   stringBefore.append(textString);
   return stringBefore.append(stringAfter);
}
```

```
string Translator::TakeFont(int index)
{
   switch(index)
   {
      case 0:
         return "italic";
      case 1:
         return "bold";
      case 2:
         return "ul";
      case 3:
         return "ulw";
      case 4:
         return "sub";
      case 5:
         return "super";
      case 6:
         return "strike";
      case 7:
         return "striked";
   }
   return "error";
}

void Translator::MakeField (TreeNode* node)
{
   int defaultSize=0;//default size of font
   FieldNode* temp=(FieldNode*)node;
   string ref=field[temp->GetID()-1];
   for(unsigned int i=0;i<ref.length();i++)
   {
      if(ref.at(i)=='"')//is not a quotation mark
         ref.erase(i,1);
   }
   GetFieldName(ref);
}

void Translator::GetFieldName(string simple)
{
   string HREF;                           //a string to store first name in field
   string rest="not a standard field";    //a string to store after field name
   int pos=simple.find_first_of(' ');
   if(pos==string::npos)
      cout << "Error. Lack of space in the field input line." << endl;
   else
   {
      HREF=simple.substr(0,pos);
      rest=simple.substr(pos+1);
      if(HREF=="HYPERLINK")
      {
         int linkpos=rest.find_first_of('\\');
         if(linkpos!=string::npos)
            file << "<field bookName=\"" << rest.substr(linkpos+3) << "\">" << endl;
         else
            file << "<field href=\"" << rest << "\">" << endl;
      }
      else
         file << "<field ref=\"" << rest << "\">" << endl;
```

```
    }
}
int Translator:: GetFontSize(TextNode* node)
{
    int const FontSizeIndex=8;//font size index in attribute table
    return node->t_table[FontSizeIndex];
}

string Translator::GetAlignment(TextNode* node)
{
    //font&paragraph table stores alignment attributes from 9 to 12
    int const alignStart=9;
    int const alignEnd=12;
    int alignIndex=9;//initialization
    for(int i=alignStart;i<alignEnd+1;i++)
    {
        if(node->t_table[i]!=0)
        {
            alignIndex=i;
            break;
        }
    }
    switch(alignIndex)
    {
        case 9:
            return "\"justified\"";
        case 10:
            return "\"center\"";
        case 11:
            return "\"right\"";
        case 12:
            return "\"left\"";
    }
    return" failed to find alignment attributes";
}

string Translator::GetFont(TextNode* node)
{
    //font&paragraph table stores font attributes from 0 to 7
    int const fontStart=0;
    int const fontEnd=7;
    fontString="";
    for(int i=fontStart;i<fontEnd+1;i++)
    {
        if(node->t_table[i]!=0)
            fontString=DoFont(i,fontString);
    }
    if(fontString=="")//the default value of font is plain
        return "plain";
    else
        return fontString;
}

string Translator::DoFont(int index,string fontString)
{
    switch (index)
    {
        case 0:
            return fontString.append("italic");
        case 1:
```

```
            return fontString.append(" bold");

        case 2:
            return fontString.append(" ul");
        case 3:
            return fontString.append(" ulw");
        case 4:
            return fontString.append(" sub");
        case 5:
            return fontString.append(" super");
        case 6:
            return fontString.append(" strike");
        case 7:
            return fontString.append(" striked");
    }
}

void Translator::EndGeneration()
{
    file << "</document>" << endl;
}

void Translator::FinishList(int countList)
{
    for(int i=countList-1;i>-1;i--)
        file << "</list>" << endl;
    makeList=false;
    countList=0;
}

/***************************************************************************
**
**        main function
**
***************************************************************************/
int main(int argc, char* argv[])
{
    static const int tableLength=13;//the number of font and paragraph attributes
    Translator* trans = new Translator();
    ifstream inFile("input.cmd");
    cout << "...load a command file" << endl;
    trans->ConstructTree(inFile);
    trans->FirstScan(trans->m_tree,trans->m_table,tableLength);
    cout << "...do translation" << endl;
    trans->SecondScan(trans->m_tree);
    cout << "...do simplification" << endl;
    trans->PrintTree(trans->m_tree, 1);
    ifstream in("input.txt");
    cout << "...load a text file" << endl;
    trans->ReadData(in);//read input.txt and classify into texts, fields and book marks
    trans->ThirdScan(trans->m_tree);
    cout << "...generate XML codes" << endl;
    if(trans->makeList==true)
        trans->FinishList(trans->countList);
    trans->EndGeneration();
    cout << "Translation is over!" << endl;
    delete trans;
    return 0;
}
```

A.2 The Class TreeNode.cpp

```
#include "TreeNode.h"
#include <string>
using namespace std;

/***************************************************************************
**
**        class TreeNode implementation
**
***************************************************************************/
TreeNode::TreeNode()
{
   m_pLeft = NULL;
   m_pRight = NULL;
}

TreeNode::~TreeNode()
{

}

void TreeNode::AddLeftSon(TreeNode* tn)
{
   m_pLeft = tn;
}

TreeNode* TreeNode::GetLeftSon()
{
   return m_pLeft;
}

void TreeNode::AddRightSon(TreeNode* tn)
{
   m_pRight = tn;
}

TreeNode* TreeNode::GetRightSon()
{
   return m_pRight;
}

/***************************************************************************
**
**        class BlockNode implementation
**
***************************************************************************/
BlockNode::BlockNode(int i) : TreeNode()
{
   m_NodeType=NT_BLOCK;
   id=i;
   use=false;
}
```

```
BlockNode::BlockNode() : TreeNode()
{
   m_NodeType=NT_BLOCK;
   use=false;
}

BlockNode::~BlockNode()
{

}

void BlockNode::SetUse(bool myUse)
{
   use=myUse;
}

bool BlockNode::GetUse()
{
   return use;
}

NODETYPE BlockNode::GetNodeType()
{
   return m_NodeType;
}

/***************************************************************************
**
**        class SequenceNode implementation
**
***************************************************************************/
SequenceNode::SequenceNode() : TreeNode()
{
   m_NodeType=NT_SEQUENCE;
}

SequenceNode::~SequenceNode()
{

}

NODETYPE SequenceNode::GetNodeType()
{
   return m_NodeType;
}

/***************************************************************************
**
**        class FieldNode implementation
**
***************************************************************************/
FieldNode::FieldNode(int id) : TreeNode()
{
   m_NodeType=NT_FIELD;
   fieldID=id;
   fieldText=false;
}
```

```
FieldNode::~FieldNode()
{

}

void FieldNode::SetID(const int id)
{
   fieldID=id;
}

void FieldNode::SetFieldText(bool flag)
{
   fieldText=flag;
}

bool FieldNode::GetFieldText()
{
   return fieldText;
}

int FieldNode::GetID()
{
   return fieldID;
}

NODETYPE FieldNode::GetNodeType()
{
   return m_NodeType;
}

/************************************************************************
**
**        class BookMarkNode implementation
**
************************************************************************/
BookMarkNode::BookMarkNode(int id) : TreeNode()
{
   m_NodeType=NT_BOOKMARK;
   bookID=id;
}

BookMarkNode::~BookMarkNode()
{

}

void BookMarkNode::SetID(const int id)
{
   bookID=id;
}

int BookMarkNode::GetID()
{
   return bookID;
}

NODETYPE BookMarkNode::GetNodeType()
{
```

```
   return m_NodeType;
}
/**************************************************************************
**
**        class ImageNode implementation
**
*************************************************************************/
ImageNode::ImageNode(int id) : TreeNode()
{
   m_NodeType=NT_IMAGE;
   imageID=id;
}

ImageNode::~ImageNode()
{

}

void ImageNode::SetID(const int id)
{
   imageID=id;
}

void ImageNode::SetFile(string str)
{
   imageFile=str;
}
string ImageNode::GetFile()
{
   return imageFile;
}

int ImageNode::GetID()
{
   return imageID;
}

void ImageNode:: SetWidth(const int width)
{
   ImageWidth=width;
}
int ImageNode:: GetWidth()
{
   return ImageWidth;
}

void ImageNode:: SetHeight(const int height)
{
   ImageHeight=height;
}
int ImageNode:: GetHeight()
{
   return ImageHeight;
}

void ImageNode:: SetScaleX(const int scaleX)
{
   ImageScaleX=scaleX;
}
```

```
int ImageNode:: GetScaleX()
{
   return ImageScaleX;
}

void ImageNode:: SetScaleY(const int scaleY)
{
   ImageScaleY=scaleY;
}

int ImageNode:: GetScaleY()
{
   return ImageScaleY;
}

NODETYPE ImageNode::GetNodeType()
{
   return m_NodeType;
}

/***************************************************************************
**
**        class TagNode implementation
**
***************************************************************************/
TagNode::TagNode() : TreeNode()
{
   m_NodeType=NT_TAG;
}

TagNode::~TagNode()
{

}

void TagNode::SetTag(string str)
{
   m_strTag=str;
}

string TagNode::GetTag()
{
   return m_strTag;
}

void TagNode::SetValue(int m_value)
{
   value=m_value;
}

int TagNode::GetValue()
{
   return value;
}

NODETYPE TagNode::GetNodeType()
{
   return m_NodeType;
```

```
}

/*************************************************************************
**
**      class TextNode implementation
**
*************************************************************************/
TextNode::TextNode(int id) : TreeNode()
{
   m_NodeType=NT_TEXT;
   TextID=id;
   initialTable();
   merge=false;
   attributeNum=13;//the number of font and paragraph attributes
}

TextNode::TextNode() : TreeNode()
{
   m_NodeType=NT_TEXT;
   TextID=0;
   initialTable();
   merge=false;
   attributeNum=13;//the number of font and paragraph attributes
}

TextNode::~TextNode()
{
}

void TextNode:: initialTable()
{
   /*0-italic,1-bold,2-ul,3-ulw,4-sub,5-super,6-strike,7-striked,8-outlevel
     9-qj,10-qc,11-qr,12-ql*/
   for(int i=0;i<attributeNum;i++)
      t_table[i]=0;
}

bool TextNode::CompareText(TextNode* a,TextNode* b)
{
   for(int i=0;i<attributeNum;i++)
      if(a->t_table[i]!=b->t_table[i])
         return false;
   return true;
}

int TextNode::GetTextID()
{
   return TextID;
}

bool TextNode::GetMerge()
{
   return merge;
}

NODETYPE TextNode::GetNodeType()
{
   return m_NodeType;
}
```

Appendix B

The File `Translator.xsl`

The XSLT file `translator.xsl` is used to transform the translated XML document into a HTML document. Nineteen templates in this file are applied to transform all elements and attributes defined by the DTD for the *translator*.

The File `Translator.xsl`

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

   <xsl:template match="/">
      <html xmlns="http://www.w3.org/TR/xhtml1/strict">
         <head>
            <title>The translated page from XML to HTML</title>
         </head>
         <body>
            <xsl:apply-templates select="document/*"/>
         </body>
      </html>
   </xsl:template>
```

```
<xsl:template match="image">
   <img>
      <xsl:attribute name="src">
         <xsl:value-of select="@source"/>
      </xsl:attribute>
      <xsl:attribute name="width">
         <xsl:value-of select="@width"/>
      </xsl:attribute>
      <xsl:attribute name="height">
         <xsl:value-of select="@height"/>
      </xsl:attribute>
   </img>
</xsl:template>

<xsl:template match="list[@type ='order']">
   <ol>
      <xsl:apply-templates select="./*"/>
   </ol>
</xsl:template>
<xsl:template match="list[@type ='unorder']">
   <ul>
      <xsl:apply-templates select="./*"/>
   </ul>
</xsl:template>

<xsl:template match="item">
   <li>
      <xsl:apply-templates select="./*"/>
   </li>
</xsl:template>

<xsl:template match="table">
   <table border="1">
      <tbody>
         <xsl:for-each select="row">
            <tr>
               <xsl:for-each select="cell">
                  <td>
                     <xsl:attribute name="align">
                        <xsl:value-of select="@alignment"/>
                     </xsl:attribute>
                     <xsl:apply-templates select="./*"/>
                  </td>
               </xsl:for-each>
            </tr>
         </xsl:for-each>
      </tbody>
   </table>
</xsl:template>

<xsl:template match="paragraph">
   <P>
      <xsl:choose>
         <xsl:when test="@alignment">
            <xsl:attribute name="align">
               <xsl:value-of select="@alignment"/>
            </xsl:attribute>
            <xsl:apply-templates select="./*"/>
         </xsl:when>
```

```
            <xsl:otherwise>
                <xsl:apply-templates select="./*"/>
            </xsl:otherwise>
        </xsl:choose>
    </P>
</xsl:template>

<xsl:template match="heading">
    <xsl:choose>
        <xsl:when match=".[@level='1']">
            <h1>
                <xsl:attribute name="align">
                    <xsl:value-of select="@alignment"/>
                </xsl:attribute>
                <xsl:apply-templates select="./*"/>
            </h1>
        </xsl:when>
        <xsl:when match=".[@level='2']">
            <h2>
                <xsl:attribute name="align">
                    <xsl:value-of select="@alignment"/>
                </xsl:attribute>
                <xsl:apply-templates select="./*"/>
            </h2>
        </xsl:when>
        <xsl:when match=".[@level='3']">
            <h3>
                <xsl:attribute name="align">
                    <xsl:value-of select="@alignment"/>
                </xsl:attribute>
                <xsl:apply-templates select="./*"/>
            </h3>
        </xsl:when>
        <xsl:when match=".[@level='4']">
            <h4>
                <xsl:attribute name="align">
                    <xsl:value-of select="@alignment"/>
                </xsl:attribute>
                <xsl:apply-templates select="./*"/>
            </h4>
        </xsl:when>
        <xsl:when match=".[@level='5']">
            <h5>
                <xsl:attribute name="align">
                    <xsl:value-of select="@alignment"/>
                </xsl:attribute>
                <xsl:apply-templates select="./*"/>
            </h5>
        </xsl:when>
        <xsl:otherwise>
            <h6>
                <xsl:attribute name="align">
                    <xsl:value-of select="@alignment"/>
                </xsl:attribute>
                <xsl:apply-templates select="./*"/>
            </h6>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

```
<xsl:template match="field">
   <A>
      <xsl:choose>
         <xsl:when test="@href">
            <xsl:attribute name="href">
               <xsl:value-of select="@href"/>
            </xsl:attribute>
            <xsl:apply-templates select="./text"/>
         </xsl:when>
         <xsl:when test="@bookName">
            <xsl:attribute name="href">
               #<xsl:value-of select="@bookName"/>
            </xsl:attribute>
            <xsl:apply-templates select="./text"/>
         </xsl:when>
         <xsl:otherwise>
            <xsl:apply-templates select="./*"/>
         </xsl:otherwise>
      </xsl:choose>
   </A>
</xsl:template>

<xsl:template match="bookmark">
   <A>
      <xsl:attribute name="name">
         <xsl:value-of select="@name"/>
      </xsl:attribute>
   </A>
</xsl:template>

<xsl:template match="text">
   <font>
      <xsl:attribute name="size">
         <xsl:value-of select="@size"/>
      </xsl:attribute>
      <xsl:apply-templates select="./*"/>
      <xsl:value-of select="text()"/>
   </font>
</xsl:template>

<xsl:template match="italic">
   <i>
      <xsl:apply-templates select="./*"/>
      <xsl:value-of select="text()"/>
   </i>
</xsl:template>

<xsl:template match="bold">
   <strong>
      <xsl:apply-templates select="./*"/>
      <xsl:value-of select="text()"/>
   </strong>
</xsl:template>

<xsl:template match="ul">
   <U>
      <xsl:apply-templates select="./*"/>
      <xsl:value-of select="text()"/>
   </U>
```

```
    </xsl:template>
    <xsl:template match="ulw">
        <U>
            <xsl:apply-templates select="./*"/>
            <xsl:value-of select="text()"/>
        </U>
    </xsl:template>

    <xsl:template match="sub">
        <SUB>
            <xsl:apply-templates select="./*"/>
            <xsl:value-of select="text()"/>
        </SUB>
    </xsl:template>

    <xsl:template match="super">
        <SUP>
            <xsl:apply-templates select="./*"/>
            <xsl:value-of select="text()"/>
        </SUP>
    </xsl:template>

    <xsl:template match="strike">
        <STRIKE>
            <xsl:apply-templates select="./*"/>
            <xsl:value-of select="text()"/>
        </STRIKE>
    </xsl:template>

    <xsl:template match="striked">
        <STRIKE>
            <xsl:apply-templates select="./*"/>
            <xsl:value-of select="text()"/>
        </STRIKE>
    </xsl:template>

</xsl:stylesheet>
```

Appendix C

The Test Files

We prepare a file, called `Test.rtf`, to test the *translator*. This file contains all fundamental elements and attributes which could constitute a simple RTF document. The corresponding command file `Test.cmd` and the text file `Test.txt`, which are both extracted by the *extractor*, are also attached in this appendix. There are two output files from the translation process. One is the translated XML file, called `XMLoutput.xml`, produced by the *translator* if the user chooses the XML presentation option. The other is `HTMLoutput.xml` produced by the *translator* if the user chooses the HTML presentation option. The file `HTMLoutput.xml` is not shown with its source code which is almost the same with `XMLoutput.xml`. Instead, we paste the browsed result by *Microsoft Internet Explorer*. Because the result shown as a HTML page is quite long, we seperate two parts of it and show them together.

C.1 The Test File `Test.rtf`

Display an XML

Xu Yi

Part 1: Test for font and font size

Small big bigger biggest. A book mark here:

Plain *italic* **bold** ***italic and bold and underline*** superstrike ~~sub and striked~~

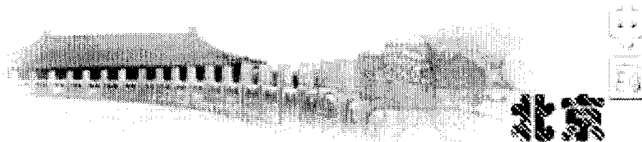
Part 2: Test for a table and alignment

The first cell at row 1	The second cell at row 1	The third cell at row 1	empty
The first cell at row 2	Left alignment	Right alignment	
The first cell at row 3	Center alignment	Justified alignment	

Part 3: Test for simplification rules

Two texts with **same font** attributes and size *are merged* into together. As you will see in input.txt file and input.xml file, **same** and **font** are two separated texts in.txt file but they are merged in XML file.

Part 4: Test for an image



Part 5: Test for lists and fields

- Item 1 of the first list. This is a hyperlink : ***www.google.ca***
- Item 2 of the first list. Another field here : ***mailto:frankxuyi@yahoo.com***
  1. The first item of an inner list begins.
  2. Item2 of the inner list
     - The most inner list
     - Second item of the most inner list
  3. Itme3 of the first inner list
- Item 3 of the first list. Go to ***mybookmark***

C.2 The Command File `Test.cmd`

```
{
 {
  {\ql\fs20}
  {\qj\outlinelevel0\fs32}
  {\qj\outlinelevel1\fs28}
  {\ql\fs24}
  {\ql\outlinelevel3\b\fs28}
  {\qc\b\fs40}
  {\qc\b\i\fs28}
  {\ul}
  {\ul}
 }
 {
  {\ls1}
  {\ls2}
  {\ls3}
  {\ls4}
 }
 {\info}
 \margr333\pard\plain\qc\outlinelevel3\b\fs28
 {\TEXT1\par}
 \pard\qc\outlinelevel3
 {\b0\i\TEXT2\par}
 \pard\plain\ql\fs20
 {\par}
 \pard\plain\ql\outlinelevel3\b\fs28
 {\TEXT3\par}
 \pard\plain\qj\fs20
 {\fs16\TEXT4}
 {\fs28}
 {\fs24\TEXT5}
 {\fs28}
 {\fs36\TEXT6}
 {\fs28}
 {\fs72}
 {\fs96\TEXT7}
 {\fs24\TEXT8}
 {\fs96
  {\bkmkstart\BOOKMARK1}
  \par
 }
 {\fs24\TEXT9}
 {\i\fs24\TEXT10}
 {\fs24}
 {\b\fs24\TEXT11}
 {\fs24}
 {\b\i\fs24\ul\TEXT12}
 {\fs24}
 {\strike\fs24\super\TEXT13}
 {\fs24}
 {\fs72\sub\striked1\TEXT14\par}
 {\fs24\sub\striked1\par}
 \pard\plain\ql\outlinelevel3\b\fs28
```

```
{\TEXT15\par}
\pard\plain\ql\fs20
{\par}
\trowd\trgaph70\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10
\brdrw10\brdrw10\brdrw10\cellx2268\brdrw10\brdrw10\brdrw10\brdrw10\cellx4962
\brdrw10\brdrw10\brdrw10\brdrw10\cellx7371\brdrw10\brdrw10\brdrw10\brdrw10
\cellx8222\pard\plain\qj\intbl\b\i\fs28
{\b0\i0\fs24\TEXT16\cell\TEXT17\cell\TEXT18\cell\TEXT19\cell}
\pard\plain\ql\intbl\fs20
{\trowd\trgaph70\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10
 \brdrw10\brdrw10\brdrw10\cellx2268\brdrw10\brdrw10\brdrw10\brdrw10\cellx4962
 \brdrw10\brdrw10\brdrw10\brdrw10\cellx7371\brdrw10\brdrw10\brdrw10\brdrw10
 \cellx8222\row}
\trowd\trgaph70\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10
\brdrw10\brdrw10\cellx2268\brdrw10\brdrw10\brdrw10\brdrw10\cellx4962\brdrw10
\brdrw10\brdrw10\brdrw10\cellx7371\brdrw10\brdrw10\brdrw10\brdrw10\cellx8222
\pard\plain\qj\intbl\b\i\fs28
{\b0\i0\fs24\TEXT20\cell}
\pard\ql\intbl
{\b0\i0\fs24\TEXT21\cell}
\pard\qr\intbl
{\b0\i0\fs24\TEXT22\cell\cell}
\pard\plain\ql\intbl\fs20
{\trowd\trgaph70\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10
 \brdrw10\brdrw10\cellx2268\brdrw10\brdrw10\brdrw10\brdrw10\cellx4962\brdrw10
 \brdrw10\brdrw10\brdrw10\cellx7371\brdrw10\brdrw10\brdrw10\brdrw10\cellx8222\row}
\trowd\trgaph70\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10
\brdrw10\brdrw10\cellx2268\brdrw10\brdrw10\brdrw10\brdrw10\cellx4962\brdrw10
\brdrw10\brdrw10\brdrw10\cellx7371\brdrw10\brdrw10\brdrw10\brdrw10\cellx8222\pard
\plain\qj\intbl\b\i\fs28
{\b0\i0\fs24\TEXT23\cell}
\pard\qc\intbl
{\b0\i0\fs24\TEXT24\cell}
\pard\qj\intbl
{\b0\i0\fs24\TEXT25\cell\cell}
\pard\plain\ql\intbl\fs20
{\trowd\trgaph70\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10\brdrw10
 \brdrw10\brdrw10\cellx2268\brdrw10\brdrw10\brdrw10\brdrw10\cellx4962\brdrw10
 \brdrw10\brdrw10\brdrw10\cellx7371\brdrw10\brdrw10\brdrw10\brdrw10\cellx8222\row}
\pard\ql
{\b\fs28\par}
\pard\plain\ql\outlinelevel3\b\fs28
{\TEXT26\par}
\pard\plain\qj\fs20
{\par}
\pard\plain\qj\b\i\fs28
{\b0\i0\fs24\TEXT27}
{\i0\fs32\TEXT28}
{\b0\i0\fs24}
{\i0\fs32\TEXT29}
{\b0\i0\fs24\TEXT30}
{\b0\fs24\TEXT31}
{\b0\i0\fs24}
{\b0\fs24\TEXT32}
{\b0\i0\fs24\TEXT33\par\TEXT34}
{\i0\fs32\TEXT35}
{\b0\i0\fs24\TEXT36}
{\i0\fs32\TEXT37}
{\b0\i0\fs24\TEXT38\par\TEXT39\par}
\pard\qj
```

```
{\i0\fs32\par}
\pard\plain\ql\outlinelevel3\b\fs28
{\TEXT40\par}
\pard\plain\ql\fs20
{\par\par}
\pard\plain\qj\b\i\fs28
{\i0\fs20
 {\shppict
  {\picscalex100\picscaley100\picwgoal5655\pichgoal1185\pngblip
   {\IMAGE input_img1.png}}
 }
 {\nonshppict
  {\picscalex100\picscaley100\picwgoal5655\pichgoal1185}
 }
}
{\b0\par}
\pard\plain\ql\fs20
{\b\fs28\par}
\pard\plain\ql\outlinelevel3\b\fs28
{\TEXT41\par}
\pard\plain\ql\fs20
{\par
 {\pard\plain}
}
\pard\plain\qj
{\pnlvlblt\ls2}
\ls2\b\i\fs28
{\b0\i0\fs24\TEXT42}
{
 {\fldinst
  {\b0\i0\fs24\FIELD1}
  {\fs20}
 }
 {\fldrslt
  {\ul\TEXT43}
 }
}
{\fs24\par
 {\pard\plain}
}
\pard\qj
{\pnlvlblt\ls2}
\ls2
{\b0\i0\fs24\TEXT44}
{
 {\fldinst
  {\b0\i0\fs24\FIELD2}
  {\b0\i0\fs20}
 }
 {\fldrslt
  {\ul\TEXT45}
 }
}
{\b0\i0\fs24\par
 {\pard\plain}
}
\pard\qj
{\pnlvlbody\ls3}
\ls3
{\b0\i0\fs24\TEXT46\par
```

```
 {\pard\plain}
}
\pard\qj
{\pnlvlbody\ls3}
\ls3
{\b0\i0\fs24\TEXT47\par
 {\pard\plain}
}
\pard\qj
{\pnlvlblt\ls4}
\ls4
{\b0\i0\fs24\TEXT48\par
 {\pard\plain}
}
\pard\qj
{\pnlvlblt\ls4}
\ls4
{\b0\i0\fs24\TEXT49\par
 {\pard\plain}
}
\pard\qj
{\pnlvlbody\ls3}
\ls3
{\b0\i0\fs24\TEXT50\par
 {\pard\plain}
}
\pard\qj
{\pnlvlblt\ls2}
\ls2
{\b0\i0\fs24\TEXT51}
{\b0\i0\fs24\TEXT52}
{
 {\fldinst
  {\b0\i0\fs24\FIELD3}
  {\b0\i0\fs20}
 }
 {\fldrslt
  {\ul\TEXT53}
 }
}
{\b0\i0\fs24\par}
}
```

C.3 The Text File `Test.txt`

```
TEXT1 | Display an XML
TEXT2 | Xu Yi
TEXT3 | Part 1: Test for font and font size
TEXT4 | Small
TEXT5 | big
TEXT6 | bigger
TEXT7 | biggest.
TEXT8 | A book mark here:
BOOKMARK1 | mybookmark
TEXT9 | Plain
TEXT10 | italic
TEXT11 | bold
TEXT12 | italic and bold and underline
TEXT13 | superstrike
TEXT14 | sub and striked
TEXT15 | Part 2: Test for a table and alignment
TEXT16 | The first cell at row 1
TEXT17 | The second cell at  row 1
TEXT18 | The third cell at row 1
TEXT19 | empty
TEXT20 | The first cell at row 2
TEXT21 | Left alignment
TEXT22 | Right alignment
TEXT23 | The first cell at row 3
TEXT24 | Center alignment
TEXT25 | Justified alignment
TEXT26 | Part 3: Test for  simplification rules
TEXT27 | Two texts with
TEXT28 | same
TEXT29 | font
TEXT30 | attributes and size
TEXT31 | are
TEXT32 | merged
TEXT33 | into together. As you will
TEXT34 | see in input.txt file and input.xml file,
TEXT35 | same
TEXT36 | and
TEXT37 | font
TEXT38 |  are two separated texts in .txt
TEXT39 | file but they are merged in XML file.
TEXT40 | Part 4: Test for an image
TEXT41 | Part 5: Test for  lists and fields
TEXT42 | Item 1 of the first list. This is a hyperlink:
FIELD1 | HYPERLINK http://www.google.ca
TEXT43 | www.google.ca
TEXT44 | Item 2 of the first list. Another field here:
FIELD2 | HYPERLINK "mailto:frankxuyi@yahoo.com"
TEXT45 | mailto:frankxuyi@yahoo.com
TEXT46 | The first item of an inner list begins.
TEXT47 | Item2 of the inner list
TEXT48 | The most inner list
TEXT49 | Second item of the most inner list
TEXT50 | Itme3  of the first inner list
TEXT51 | Item 3 of the first list.
TEXT52 | Go to
FIELD3 | HYPERLINK  \l "mybookmark"
TEXT53 | mybookmark
```

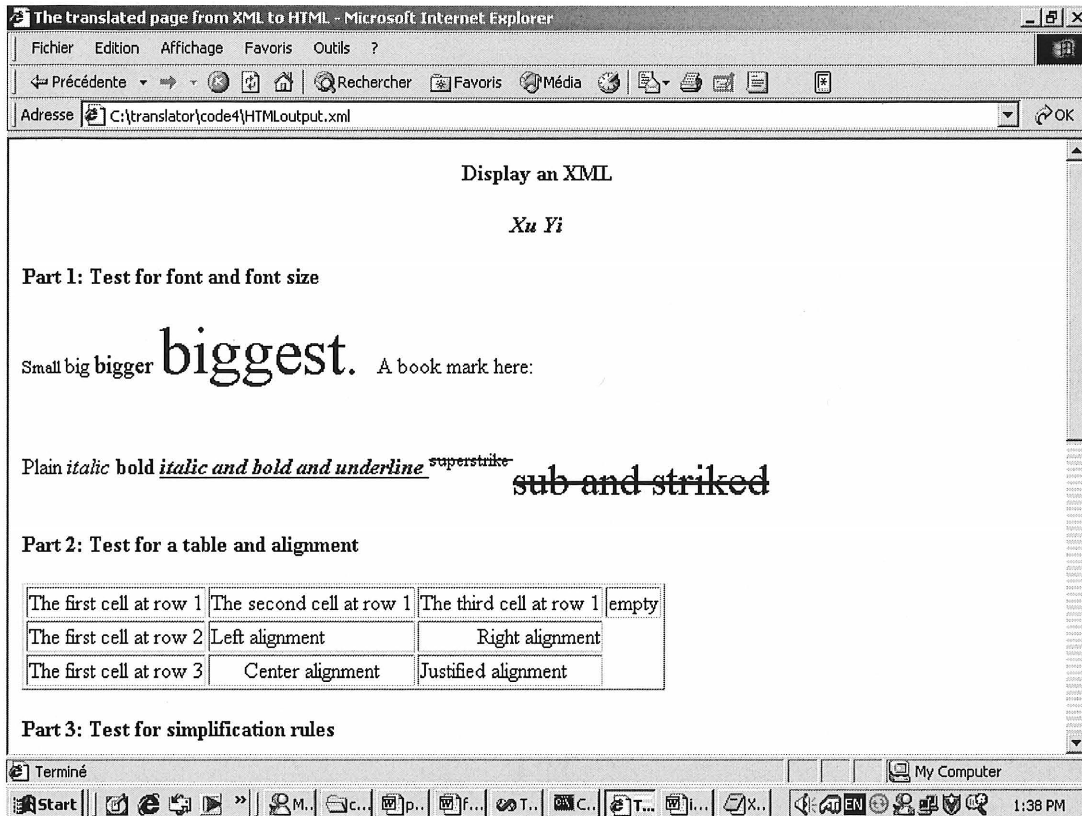
C.4 The Translated XML File `XMLoutput.xml`

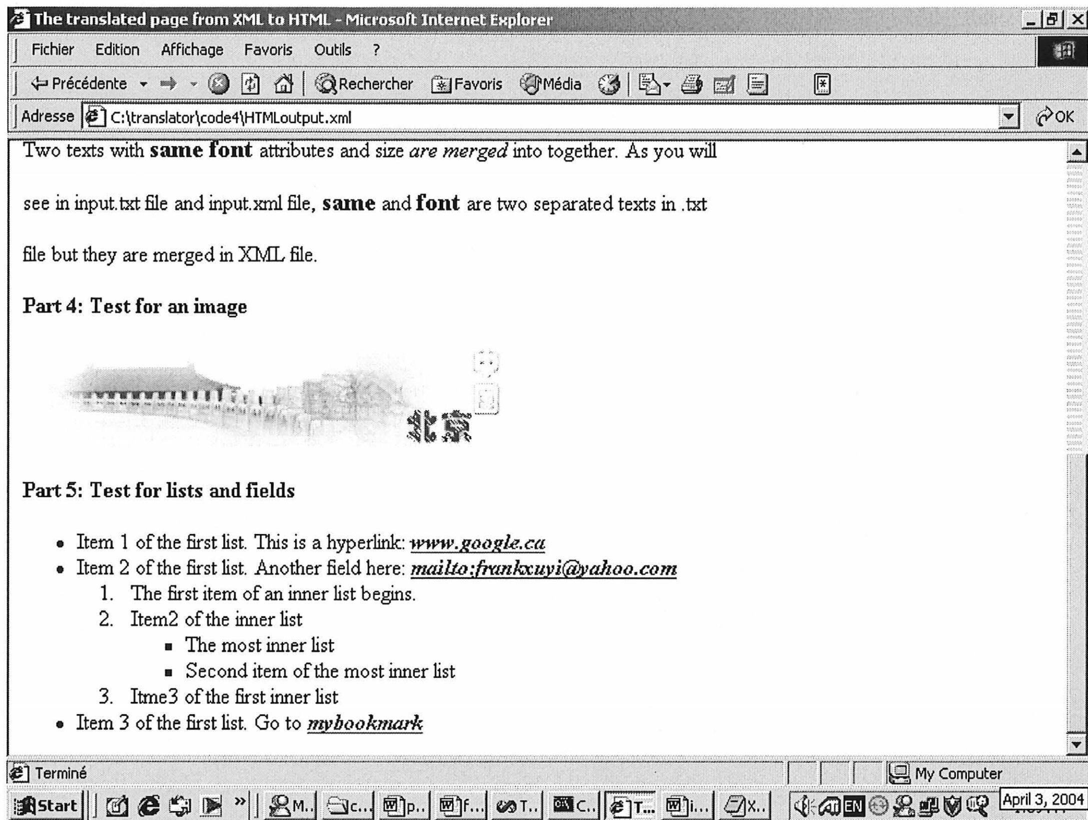
```
<?xml version="1.0" ?>
<document>
<heading level="4" alignment="center">
<text><bold>Display an XML</bold></text>
</heading>
<heading level="4" alignment="center">
<text><italic>Xu Yi</italic></text>
</heading>
<paragraph alignment="center">
</paragraph>
<heading level="4" alignment="left">
<text><bold>Part 1: Test for font and font size </bold></text>
</heading>
<paragraph>
<text size="2">Small</text>
<text>big</text>
<text size="4">bigger</text>
<text size="12">biggest. </text>
<text>A book mark here:</text>
<bookmark name="mybookmark"></bookmark>
</paragraph>
<paragraph>
<text>Plain </text>
<text><italic>italic</italic></text>
<text><bold>bold</bold></text>
<text><italic><bold><ul>italic and bold and underline</ul></bold></italic></text>
<text><super><strike>superstrike</strike></super></text>
<text size="9"><sub><striked>sub and striked</striked></sub></text>
</paragraph>
<paragraph>
</paragraph>
<heading level="4" alignment="left">
<text><bold>Part 2: Test for a table and alignment</bold></text>
</heading>
<paragraph alignment="left">
</paragraph>
<table>
<row>
<cell position="2268">
<text>The first cell at row 1</text>
</cell>
<cell position="4962">
<text>The second cell at  row 1</text>
</cell>
<cell position="7371">
<text>The third cell at row 1</text>
</cell>
<cell position="8222">
<text>empty</text>
</cell>
</row>
<row>
<cell position="2268">
```

```
<text>The first cell at row 2</text>
</cell>
<cell alignment="left" position="4962">
<text>Left alignment</text>
</cell>
<cell alignment="right" position="7371">
<text>Right alignment</text>
</cell>
<cell alignment="right" position="8222">
</cell>
</row>
<row>
<cell position="2268">
<text>The first cell at row 3</text>
</cell>
<cell alignment="center" position="4962">
<text>Center alignment</text>
</cell>
<cell position="7371">
<text>Justified alignment</text>
</cell>
<cell position="8222">
</cell>
</row>
</table>
<paragraph alignment="left">
</paragraph>
<heading level="4" alignment="left">
<text><bold>Part 3: Test for  simplification rules</bold></text>
</heading>
<paragraph alignment="left">
</paragraph>
<paragraph>
<text>Two texts with </text>
<text size="4"><bold>same font</bold></text>
<text>attributes and size </text>
<text><italic>are merged</italic></text>
<text>into together. As you will</text>
</paragraph>
<paragraph>
<text>see in input.txt file and input.xml file, </text>
<text size="4"><bold>same</bold></text>
<text>and </text>
<text size="4"><bold>font</bold></text>
<text> are two separated texts in .txt</text>
</paragraph>
<paragraph>
<text>file but they are merged in XML file.</text>
</paragraph>
<paragraph>
</paragraph>
<heading level="4" alignment="left">
<text><bold>Part 4: Test for an image</bold></text>
</heading>
<paragraph alignment="left">
</paragraph>
<paragraph alignment="left">
</paragraph>
<image source="input_img1.png" width="356" height="74" scaleX="100" scaleY="100">
</image>
```

```
<paragraph alignment="left">
</paragraph>
<paragraph alignment="left">
</paragraph>
<heading level="4" alignment="left">
<text><bold>Part 5: Test for  lists and fields</bold></text>
</heading>
<paragraph alignment="left">
</paragraph>
<list type="unorder">
<item>
<text>Item 1 of the first list. This is a hyperlink: </text>
<field href="http://www.google.ca ">
<text><italic><bold><ul>www.google.ca</ul></bold></italic></text>
</field>
</item>
<item>
<text>Item 2 of the first list. Another field here: </text>
<field href="mailto:frankxuyi@yahoo.com ">
<text><italic><bold><ul>mailto:frankxuyi@yahoo.com</ul></bold></italic></text>
</field>
</item>
<list type="order">
<item>
<text>The first item of an inner list begins. </text>
</item>
<item>
<text>Item2 of the inner list</text>
</item>
<list type="unorder">
<item>
<text>The most inner list</text>
</item>
<item>
<text>Second item of the most inner list</text>
</item>
</list>
<item>
<text>Itme3  of the first inner list</text>
</item>
</list>
<item>
<text>Item 3 of the first list.  Go to </text>
<field bookName="mybookmark ">
<text><italic><bold><ul>mybookmark</ul></bold></italic></text>
</field>
</item>
</list>
</document>
```

C.5 The Transformed HTML Page




The translated page from XML to HTML - Microsoft Internet Explorer
Fichier Edition Affichage Favoris Outils ?
Précédente Rechercher Favoris Média
Adresse C:\translator\code4\HTMLoutput.xml
OK
Two texts with same font attributes and size are merged into together. As you will
see in input.txt file and input.xml file, same and font are two separated texts in .txt
file but they are merged in XML file.
Part 4: Test for an image
北京
Part 5: Test for lists and fields
Item 1 of the first list. This is a hyperlink: www.google.ca
Item 2 of the first list. Another field here: mailto:frankxuyi@yahoo.com
1. The first item of an inner list begins.
2. Item2 of the inner list
The most inner list
Second item of the most inner list
3. Itme3 of the first inner list
Item 3 of the first list. Go to mybookmark
Terminé
My Computer
Start
April 3, 2004

Bibliography

[1] D. Grune, H. E. Bal, C. J. Jacobs and K. G. Langendoen. *Modern Compiler Design.* John Wiley&sons, Chichester, England, 2000.

[2] G. Born. *File Formats Handbook.* International Thomson Computer Press, London, England, 1995.

[3] P. A. Mansfield. How to Make a File Format Translator Using XML. *Proceeding of XML of XML Conference & Exposition*, Baltimore, Maryland, USA, December 8-13, 2002. http://www.idealliance.org/papers/xml02/dx_xml02/papers/05-04-04/05-04-04.html (available on March 16, 2004).

[4] C. Wheedleton. The Role of XML in a New Intelligence Paradigm. *Proceeding of XML of XML Conference & Exposition*, Baltimore, Maryland, USA, December 8-13, 2002. http://www.idealliance.org/papers/xml02/dx_xml02/papers/03-02-01/03-02-01.html (available on March 16, 2004).

[5] A. G. Adwired. *RTF to XML Converter.* http://www.rtf2fo.com/references.html, (available on March 16, 2004).

[6] North Atlantic Publishing Systems, Inc. *North Atlantic Announces NTS: Quark XML Conversion Product Streamlines Cross-Media Publishing.* http://www.napsys.com/NTS%20Announcement.html (available on March 16, 2004).

[7] S. M. Burke. *RTF Pocket Guide.* O'Reilly & Associates, Cambridge, MA, USA, 2003.

[8] B. Bébubé and R. St-Denis. Web-Site Design Using D2Engine. *Proceedings of the IASTED International Conference on Internet and Multimedia Systems and Applications*, Honolulu, Hawaii, USA, 128-132, August 2001.

[9] D. E. Knuth. *TEX and METAFONT: New Directions in Typesetting*. The American Mathematical Society and Digital Press, Providence, R.I. USA, 1979.

[10] Microsoft Word. Rich Text Format (RTF) Specification version 1.6. http://msdn.microsoft.com/library/default.asp?url=/library/enus/dnrtfspec/html/rtfspec.asp(available on March 16, 2004).

[11] K. C. Louden. *Compiler Construction Principles and Practice*. PWS Publishing Company, Boston, MA, USA, 1997.

[12] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. *Compiler Design Theory*. Addison Wesley Publishing Company, Reading, MA, USA, 1976.

[13] N. Dale. *C++ Plus Data Structures*. Jones and Bartlett Publishers, Sudbury, MA, USA, 1999.

[14] N. Dale, D. T. Joyce, and C. Weems. *Object-Oriented Data Structures Using Java*. Jones and Bartlett Publishers, Sudbury, MA, USA, 2002.

[15] I. Graham. *Object-Oriented Methods*. Addison-Wesley, Reading, MA, USA, 2001.

[16] D. Kafura. *Object-oriented Software Design & Construction with Java*. Prentice Hall, Upper Saddle River, NJ, USA, 2000.

[17] W3C. *Extensible Markup Language 1.0*. http://www.xml.com/axml/testaxml.htm (available on March 16, 2004).

[18] MICROSOFT. *MSDN Library*. http://msdn.microsoft.com/library/default.asp?url=/library/enus/xmlsdk/htm/dtd_dev_2bw8.asp (available on March 16, 2004).

[19] J. E. Refsnes. *Introduction to XSL*. http://xmlfiles.com/xsl/xsl_intro.asp (available on March 16, 2004).

[20] W3C. XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt (available on March 16, 2004).