

XML and Its Basic Techniques

Azri Bin Mohamed Yasin
Hu He
Phan Thi Quynh Trang
Tran Binh Thanh
Yap Ying Hui Priscilla

Hong Dai Thanh
Nguyen Quang Phuc
Shivam Pathak
Tran Thanh Phu
Zhang Xiaoyu

Table of Contents

Introduction.....	1
Chapter A XML.....	2
I. Introduction of eXtensible Markup Language	2
1. The Overview of eXtensible Markup Language (XML)	2
2. Development History	3
II. The basic structural design of XML documents	4
1. The Graph Theory: Tree	4
2. Structure of XML Document	5
III. An Overview of XML Syntax.....	6
1. XML Prologue	6
i. XML Declaration	6
ii. Document Type declaration (DTD)	7
2. XML Tags	7
i. Paring Rule	7
ii. Single tags:	8
3. XML Elements	8
4. XML Attributes	10
5. Entities	12
6. Comments	14
7. Processing Instructions	14
8. Characters	15
9. White spaces	15
10. Naming XML elements	16
IV. Namespaces in XML.....	18
1. Introduction to Namespaces.....	18
2. Declaring Namespaces	19
3. Namespaces in XML Documents.....	19
i. Default Namespaces	19
ii. Multiple Prefixed Namespaces	20
iii. Default and Prefixed Namespaces	20
V. Content Model in XML.....	23

1. Empty Content Model	23
2. Any Content Model	23
3. PCDATA Content Model	23
VI. Well-formed XML	24
1. Naming Conventions	24
2. Proper structure	24
3. Proper Tag Pairing & Matching	25
4. Root Element: One and Only One	25
5. Proper Entity References	26
VII. Example Explanation	28
VIII. Conclusion	32
Chapter B DTD.....	33
I. Introduction	33
II. Document type declaration	34
1. Internal DTD	34
2. External DTD	35
III. Document Structure	38
1. Element:	38
i. Empty Element	38
ii. Element with only text content	38
iii. Element with only child elements.....	39
iv. Element with any content.....	41
v. Element with mixed content.....	42
2. Attributes	42
i. CDATA	44
ii. Enumerated	44
iii. ID	44
iv. IDREF or IDREFS.....	45
v. NMTOKEN or NMTOKENS:	47
vi. ENTITY or ENTITIES.....	47
vii. NOTATION	47
4. XML Character Data	48
i. Parsed Character Data	48

ii. Unparsed Character Data.....	48
IV. Document Content	50
1. Notations:	50
i. Description:.....	50
ii. Usage:.....	50
iii. Syntax:.....	50
2. Entities:	51
i. Description:.....	51
ii. Parsed Entities.....	52
iii. Unparsed Entities:.....	53
V. XML Document Validation.....	55
1. General Overview	55
2. How to validate	55
VIII. Conclusion	58
Chapter C XPath.....	59
I. Introduction	59
II. Basic concepts in XPath	60
1. XPath Data Types	60
2. XPath Data Model	60
i. Root node.....	60
ii. Element node	60
iii. Attribute node.....	61
iv. Namespace node	61
v. Processing instruction node.....	61
vi. Comment node	61
vii. Text node	61
3. Document order.....	62
4. Context.....	63
III. Location path.....	64
1. Location Path	64
i. Absolute Location Paths.....	64
ii. Relative Location Paths	64
2. Location Step.....	65

3.	Evaluation Process	66
i.	How a Location Step is evaluated	66
ii.	How a Location Path is evaluated	66
4.	Axes	69
i.	Child axis	69
ii.	Descendant axis	70
iii.	Self axis.....	71
iv.	Descendant-or-self axis.....	72
v.	Attribute axis.....	72
vi.	Parent axis.....	72
vii.	Ancestor axis	73
viii.	Ancestor-or-self axis	73
ix.	Preceding axis	73
x.	Following axis	74
xi.	Preceding-sibling axis.....	74
xii.	Following-sibling axis	75
5.	Node Tests	75
i.	Type Test	76
ii.	Name Test	77
iii.	Processing-instruction.....	78
6.	Predicates.....	79
i.	Predicates of type “Boolean Statement”	79
ii.	Predicates of type Location Path	82
iii.	Compound Predicates.....	83
iv.	Multiple Predicates	84
7.	Abbreviated Syntax	85
i.	Child Axis	85
ii.	Position Predicate	85
iii.	Attribute Axis	86
iv.	Select All Descendants of the Context Nodes.....	86
v.	Select the Context Node	86
vi.	Select Parent Node	87
IV.	Conclusion	89

Chapter D XQuery.....90

I. Introduction 90

II. XQuery 91

1. XQuery Definition..... 91
2. XQuery Terminology 91
3. Basic Syntax rules..... 91
4. XQuery Path Expressions 92
 - i. Select the XML document to be queried 92
 - ii. Path expressions to address elements of interest..... 92
 - iii. Compose into a piece of query 92
5. Element Constructors and Enclosed Expressions 94
6. XQuery FLWOR expressions..... 95
7. FLWOR is an acronym for "For, Let, Where, Order by, Return" 96
 - i. for..... 97
 - ii. let 98
 - iii. where 99
 - iv. order by and return..... 100
8. Conditional Expression:..... 105
9. Quantified Expression: 107
10. Arithmetic 108
- Comparison 109
11. Built-in functions:..... 109
12. User-defined functions: 114
13. Path expressions vs FLWOR expressions 115

III. Conclusion..... 116

Bibliography118

Appendix120

Introduction

HTML, HyperText Markup Language, is the predominant language to develop websites nowadays. However, HTML has some limitations like lacking the capability to support all font styles, dependency on scripting language, disallowing user to define new structures...

XML, Extensible Markup Language, has been introduced to complement HTML, giving us more flexibility: creating new tags, defining new structures, transferring data easily across different applications and network... XML is also simple and easy to learn. More details about XML are given in the first chapter of the book.

This book provides a short and simple yet complete instruction for XML beginners. After reading this book, you will gain a basic understanding of XML structure, how to define your own block of data, performing queries to request for information...

The book includes 4 chapters. Chapter XML gives an over view of XML structure, XML syntax, name space in XML and well-formed XML document. Chapter DTD presents the rules to define the structure of an XML document and how to validate XML documents against a DTD. The next two chapters we discuss how to use XML functions to query XML data. Chapter X Path discusses XPath expressions and XPath standard functions. Finally, chapter X Query covers XQuery built-in and user defined functions.

Chapter A: XML

I. Introduction of eXtensible Markup Language

1. The Overview of eXtensible Markup Language (XML)

The Markup Language nowadays is a common terminology to specify a system of text which is marked up by some special syntactical keywords called *tags*, and formed structurally to define, present and process data and information. The meaning of using *tags* is basically to annotate the text to be distinguished and formatted. There are some popular markup languages today, like *HyperText Markup Language* (HTML), *Rich Text Format* (RTF). Figure A.I.1.1 shows an illustration of a markup language:

```
<html>
  <head>
    <title>This is the Restaurant Collection</title>
  </head>
  <body>
    <p>Kent Ridge Highlights</p>
  </body>
</html>
```

Figure A.I.1.1 Example of HTML – a Markup Language

In this book we will discuss about a markup language as well as a strong data processing technology which is called XML.

XML stands for *eXtensible Markup Language*, so that XML is a markup language, which is extensible. The importance of the extensibility of XML is that we can define the ways its structure to be performed and displayed. Moreover, with the inheritance from its forth father, *Standard Generalized Markup Language* (SGML), XML is very flexible in designing the format, and we can come up with its specification to define another Markup Language. However, unlike SGML, which is very complicated and hard to use, XML is much more handy and comfortable. (St.Laurent, 1998)

XML today is constructed by W3C – World Wide Web Consortium, and its usage has become popular as defined in XML 1.0 Recommendation.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!--This is an example of XML-->
<restaurant>
  <rname>Kent Ridge Highlights</rname>
  <rtype type_of_restaurant="FastFood" type_of_cuisine="Western Exclusive Cooking Styles"/>
</restaurant>
```

Figure A.I.1.2: an example of XML document

2. Development History

In the late 1960s, a group of researchers led by Charles Goldfarb from *International Business Machines (IBM)* started a project called *GenCode*, which was a platform inspired by the presentation of William W. Tunnickliffe on the idea of markup language. This project was then upgraded and the team developed a higher level of this plan to Generalized Markup Language in 1969. (EnzineArticle.com, 2011)

In 1974, Goldfarb developed SGML, which is the father of all the current markup languages. After 1980s, almost all of the new markup languages are derived from SGML, including HTML and XML. (Tay, 2003)

HTML was created by Sir Tim Berners-Lee in the early 1990s by using SGML as a developing platform. HTML is undeniably the most used markup language over because of its popularity in web services as well as data storage.

However, HTML has its own restriction, and there are a lot of things that HTML could not carry out. Developers need more flexibility in HTML. That led to the reason why XML was developed. (St.Laurent, 1998)

In the midst of 1990, Jon Bosak led a team in World Wide Web Consortium to develop a new markup language, XML. XML has become a powerful markup language because it also provides a platform to define other markup languages by its extensibility. XML is not only useful in the web services, but also in data structures because it supports a clear and well-formed environment to store data (SMGLsource.com, 2011).

In the 2000s, many XML-based languages were created, while a lot of them are becoming more and more popular like RSS (Really Simple Syndication) and XHTML (eXtensible HyperText Markup Language). Nowadays there are many applications that have adopted XML format like Microsoft Office and OpenOffice.org.

II. The basic structural design of XML documents

Everything has a story to begin, and so does XML. The story of XML starts with the Graph theory of Tree.

XML has a close relationship with the Graph Theory of Tree because of its basic structure. The structure of XML does not follow any arbitrary rule, but it has the formation of a tree. (Holzner, 1998)

To start with, we need to discuss about some of the basic concept in graph theory, which is really important for the understanding of XML Structure: Tree.

1. The Graph Theory: Tree

Firstly, we need to define a graph. According to Wolfram Mathworld (2011), a graph can be defined as:

In a mathematician's terminology, a graph is a collection of points and lines connecting some (possibly empty) subset of them. The points of a graph are most commonly known as graph vertices, but may also be called "nodes". Similarly, the lines connecting the vertices of a graph are most commonly known as graph edges, but may also be called "lines"

In practice, we also usually call the *node* as *vertex*, and *line* as *edge*. An example of graph is shown as figure A.II.1.1.

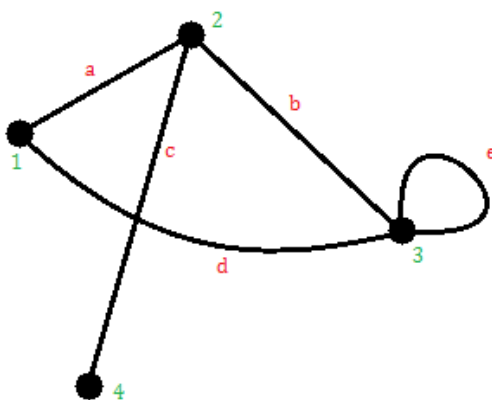


Figure A.II.1.1 - An example of Graph

1, 2, 3, 4: Vertices

a, b, c, d, e: Edges

Any graph can be categorized as 2 main types: directed and undirected graph. (Koehler, 2002). In the Graph Theory, a Tree is an undirected graph that any pair of its vertices is connected by exactly one simple path (no duplication). We can define any node as a root. We can see a demonstration in figure A.II.1.2.

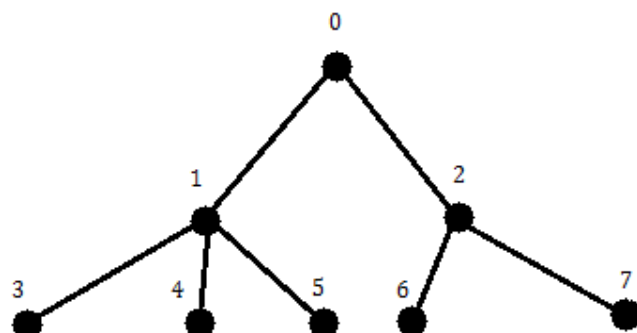


Figure A.II.1.2 – example of a Tree
The node 0 is the root of the tree

2. Structure of XML Document

Any XML Document has a well-formed structure. As each element can hold other elements, and so on, XML structure works like a tree and it is naturally easy to organize the data. (Tay, 2003)

Root: every XML document has only one root element.

Ancestors: every element that stays at a higher level of a particular element is its ancestor.

Descendants: every element that stays at a lower level of a particular element is its descendant. We can recognize the descendants of an element easily by looking to the content within the 2 tags of an element (Please refer to section 3.2)

Parent: every element, except the root element, has only one parent that is the element that covers it at one level higher.

Child: every element may have more than one child. The children of an element are the elements that stay within its tags at exact one level below.

Siblings: elements that are at the same level under a parent, which cannot be either ancestors or descendants of others.

Figure A.II.2.1 shows an example of an XML document as a tree.

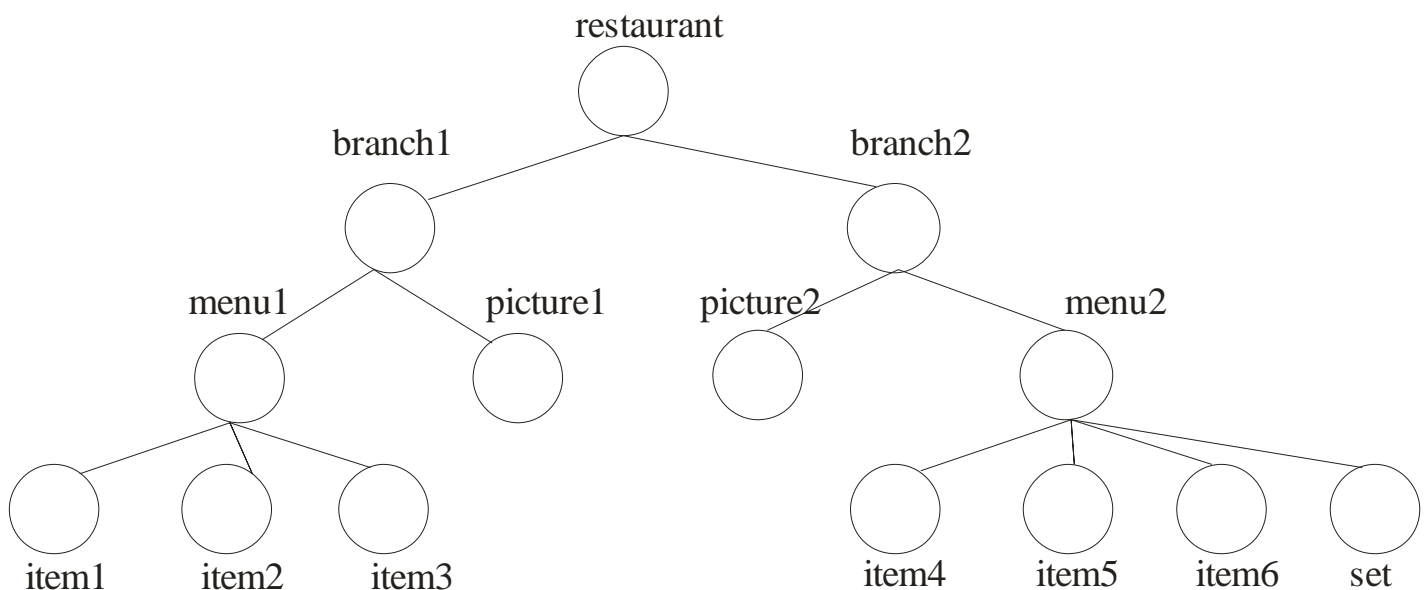


Figure A.II.2.1

Sample explanations of figure A.II.2.1:

The root is the restaurant node. Restaurant, branch1 & menu1 are ancestors of item1, item2, item3. The nodes item4, item5, item6, set & menu2 are together descendants of branch2. The branch1 node is the parent of picture1, and picture1 is the child of branch1. The picture2 and menu2 are siblings.

III. An Overview of XML Syntax

1. XML Prologue

i. XML Declaration

All XML documents should begin with an XML declaration. XML declaration must be located at the first position of the first line in the XML document. XML declaration consists of version number, encoding declaration and standalone status as in figure A.III.1.1.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

Figure A.III.1.1 – XML

We notice that the XML declaration has no closing tag, that is `</?xml>`, as we consider the declaration as a special kind of XML tags. The tags will be covered later in this chapter. Figure A.III.1.2 gives us some possible attributes that we can use in XML declaration. We notice that the Document Type declaration (DTD) will be covered later in the next chapter.

Attribute Name	Required	Possible Attribute Value	Attribute Description
version	YES	1.0	Specifies the version of the XML standard that the XML document refers to.
encoding	NO	UTF-8, UTF-16, ISO-10646-UCS-2, ISO-10646-UCS-4, ISO-8859-1 to ISO-8859-9, ISO-2022-JP, Shift_JIS, EUC-JP	The encoding attribute is used to specify the character set that is used for encoding the document. The default value is "UTF-8."
standalone	NO	yes, no	Denote whether or not the document requires a DTD in order to be processed. The default value is 'Yes'. Use 'yes' if the XML document has an internal DTD. Use 'no' if the XML document is linked to an external DTD, or any external entity references.

Figure A.III.1.2 – Possible Attribute

In figure A.III.1.1, we can see that the version which is being referred in this XML document later on is "1.0". The encoding declaration is "UTF-8". The standalone status equals to "no" means that all the elements, attributes and entities of this XML document is

imported from external DTD. The terminologies and functions of elements, attributes, entities and external DTD will be covered later in this chapter.

ii. Document Type declaration (DTD)

Besides having the above basic declaration, declaration may include Document Type declaration (DOCTYPE declaration) to define the legal building blocks of an XML *document*. The DTD is used to link an internal set of declarations with the document, or to link the XML document to an external DTD file for validation (Simon St. Laurent, Robert Biggar, 1999). Figure A.III.1.3 is an example of Document Type declaration to link the XML document to an external DTD file for validation.

```
<!DOCTYPE restaurant_collection SYSTEM "restaurant_collection.dtd">
```

Figure A.III.1.3

We will discuss DOCTYPE declaration in more details in the chapter DTD later.

2. XML Tags

In every other mark-up language, a tag is used to mark data up. In XML, tags play the key role to structure and store data. XML tags mark data into parts, like a tree system in the graph theory.

i. Paring Rule:

Unlike HTML, the tags in XML are mostly in pairs. Whenever we open a tag, we must close it. XML tags have the form:

```
<One_tag> some data </One_tag>
```

Different from HTML where there is a certain number of tag names and are fixed, while the tag names in XML are chosen flexibly and all up to the favourites of the users (Holzner, 1998). Therefore, the tags in XML not only display data, but also tell the users what are the meanings of them. We can see the difference in figures A.III.2.1 and A.III.2.2

```
<h1> Berry-Berry Belgian Waffles </h1>  
<p> light Belgian waffles covered with an assortment of fresh berries and whipped cream </p>
```

Figures A.III.2.1: HTML tags

```
<name>Berry-Berry Belgian Waffles</name>  
<description>  
light Belgian waffles covered with an assortment of fresh berries and whipped cream  
</description>
```

Figure A.III.2.2: XML tags

From the example above, we can see that the “Berry-Berry Belgian Waffles” is the information being enclosed by the tags <name> and </name>, which shows the name of the dish. Similarly, the phrase “light Belgian waffles covered with an assortment of fresh berries and whipped cream” is the description of the dish, enclosed by tags <description> and </description>. In comparing to the examples of HTML tags, we can see clearly that the XML tags are more expressive.

Moreover, the name of the tag in XML is case-sensitive (World Wide Web Consortium, 2011). See the figure A.III.2.3, we can see that the two tags are completely different, although they have the same meaning.

```
<name>Berry-Berry Belgian Waffles</name>  
<NAME>Berry-Berry Belgian Waffles</NAME>
```

Figure A.III.2.3: an XML tag names which have case-sensitive difference

ii. Single tags:

XML also has single (or solo) tags which do not carry data. However, they can carry some special information within the tags themselves, which we will cover in the part Attributes. (Harold, Means, 2004)

These tags have another form of syntax to help the processors recognize that it is completed. The syntax of this tag is just revised a bit from opening tag: <Single_tag/>

```
<type course = "Appetizer"/>
```

Figure A.III.2.4: a self-closing XML tag

3. XML Elements

XML elements are determined as everything inclusively from the start tag to the end tag. (Holzner, 1998)

In XML, each element may contain the two tags, the data information between the tags (figure A.III.3.1), and attributes with values (figure A.III.3.2) (which will be covered later). XML element also may contain other child elements inside (figure A.III.3.3).

```
<!--a typical element with a tag pair -->  
<description>Steamed Sweet Tapioca with Coconut Creame</description>
```

Figure A.III.3.1: a typical element with a tag

```

<!--a typical element with attributes -->
<branch branch_id="10" address="87 Kent Ridge Road" hotline="66109666">
</branch>

```

Figure A.III.3.2 a typical element with attributes

```

<!--an element with child elements inside -->
<item item_id="101">
  <name>Strawberry Belgian Waffles</name>
  <price>&sgd; 6.50</price>
  <description>light Belgian waffles covered with strawberries </description>
  <type course = "Appetizer"/>
</item>

```

Figure A.III.3.3 an element with child elements inside

Empty element is defined by putting the end-tag immediately after the start-tag. Empty element is also defined by omitting the end-tag and adding the slash at the end of the start-tag (self-closing or solo tag) as in figure A.III.3.4

```

<!--an empty element-->
<type course = "Dessert"/>

```

Figure A.III.3.4 – an empty element

According to the XML 1.0 Recommendation (2011), any well-formed XML document should have one, and only one root element which called document element. This root element contains all the other elements. As we can see in the grand example, the root element is the <restaurant_collection>. The well-formedness of XML document will be discussed in more details at the end of this chapter.

At this time we look at the 2 figures A.III.3.5 and A.III.3.6 to see the difference between a non well-formed document and well-formed document.

```

<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<branch branch_id="10" address="87 Kent Ridge Road" city="Singapore"
hotline="66109666">
</branch>
<branch <branch branch_id="20" address="145 Thompson Road" city="Singapore"
hotline="66106999">>
</branch>

```

Figure A.III.3.5 - non well-formed

```

<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<item item_id="100">
  <name>Berry-Berry Belgian Waffles</name>
  <price>&sgd; 7.95</price>
  <description>
    light Belgian waffles covered with an assortment of fresh berries
  </description>
  <type course = "Appetizer"/>
  <picture source="www.flickr.com/kentridge/item100.png" type="image-svg"/>
</item>

```

Figure A.III.3.6- well-formed document

In figure A.III.3.6, we can see that ‘item’ is an element which has an attribute, that is item_id is equal to 100. Within the element, there are such other sub-elements, named ‘name’, ‘price’, ‘description’, ‘type course’ and ‘picture’. These sub-elements can definitely be considered as the properties of the ‘item’ element.

The element ‘name’ is an element which is “Berry-Berry Belgian Waffles”, shows the name of the dish.

The element ‘price’ is an element which is “\$7.95”, shows the price of the dish.

The element ‘description’ is an element which is “light Belgian waffles covered with an assortment of fresh berries and whipped cream” shows the yummy description of the dish.

The element ‘type course’ is an element which has only on tag, and an attribute “Appetizer”, shows the role of the dish.

The element ‘picture’ is a self-closing tag with the attribute ‘source’ which is “www.flickr.com/kentridge/item100.png” and attributes ‘type’ is “image-svg”. The attributes and their usage will be discussed later.

So, from the example above, we can see that an XML document can effectively use meaningful tag names to show the meanings of its data, rather than just store it.

4. XML Attributes

In XML, as well as HTML, attributes are used to give additional information about an element. XML attributes are not part of the data, but they give important information about the elements. They are put within the start tag (or the only tag if the tag is self-closing) to provide the properties of the element. The attributes usually have values declared after the equal sign, enclosed within the quotes. One tag may have zero, or more than one attribute. (Holzner, 1998)

The use of XML is very similar to HTML in the sense of showing more information of the elements. However, in XML the attributes are strictly required the values together with, while some HTML attributes do not need values (for example, the Boolean attributes). (St.Laurent, 1998)


```

<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<item item_id="203">
  <name>Chicken Steam Hotplate</name>
  <price>&sgd; 8.70</price>
  <description>A Chicken drumstick steamed with hot white rice</description>
  <type course = "MainDish"/>
  <picture title="item num 203" source="www.flickr.com/kentridge/item203.png"
    type="image-svg"/>
</item>

```

figure A.III.4.1 - XML Attributes

From the *figure A.III.4.1*, we can see that the `<item>` tag has one attribute that is `item_id` (value is 203), the `<type>` tag has one attribute, that is `course` (value is “MainDish”). An element can also have more than one attribute, as the element within the tag `<picture>` has 2 attributes, one is `title` (value is “item num 203”) and another one is `source` (value is the URL “www.flickr.com/kentridge/item203.png”).

Since XML can provide flexible tag names (Williamson, 2001), so an XML attribute can be considered as another tag within a parent tag. We can compare the use of these different ways to get better understanding why XML Attributes are not preferred. The figure A.III.4.2 shows an equivalent way to the figure A.III.4.1

```

<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<item>
  <item_id>203</item_id>
  <name>Chicken Steam Hotplate</name>
  <price>&sgd; 8.70</price>
  <description>A Chicken drumstick steamed with hot white rice</description>
  <type>
    <course>MainDish</course>
  </type>
  <picture>
    <title>item num 203</title>
    <source>www.flickr.com/kentridge/item203.png</source>
    <type>image-svg</type>
  </picture>
</item>

```

figure A.III.4.2 - Equivalent XML elements

Actually, there are some reasons that we should use child elements instead of attributes.

Firstly, the XML attributes definitely can hold one and only one value, while we can have multiple same child elements which depend on the declaration in the DTD part. We can see this difference in the grand example: the `<menu>` element has many `<item>` elements, as well as many `<set>` elements.

Secondly, according to World Wide Web Consortium (2011), the attributes do not follow the tree structure, which is an advantage of XML document. So that it is easier to organize the data, and expand it later on by using child elements instead of the attributes. Consider the *figure A.III.4.3* with the use of attributes and the equivalent *figure A.III.4.4* XML code with the use of child elements we can see the difference in their structures. The *figure A.III.4.4* can be expanded into more detailed data, in *figure A.III.4.5* while the original *figure A.III.4.3* could not be able to do that.

```
<rtype type_of_restaurant="FastFood" type_of_cuisine="Western Exclusive Cooking Styles"/>
```

figure A.III.4.3

```
<rtype>
  <type_of_restaurant>Fast Food</type_of_restaurant>
  <type_of_cuisine> Western Exclusive Cooking Styles</type_of_cuisine>
</rtype>
```

figure A.III.4.4

```
<rtype>
  <type_of_restaurant>
    <type_name>Fast Food</type_name>
    <scale>Medium</scale>
  </type_of_restaurant>
  <type_of_cuisine>
    <cuisine_name>Western Exclusive Cooking Styles</cuisine_name>
    <operation_method>Western Food</operation_method>
    <material_origin>Australia</material_origin>
  </type_of_cuisine>
</rtype>
```

figure A.III.4.5 with the expansion in bold

5. Entities

As we have discussed so far, XML is not only a markup language, but also a way to store data. In real life definitely there is a lot of information that is repeated many times within one document. Since no one wants to repeat a boring job just to write a bunch of data (text or binary file) every time, XML Entity is the solution. Once an XML entity is created, it works as a shortcut of those data in the rest of the document (Entities - XML Tutorial, 2011).

Moreover, XML also can refer to entities from other source files (Please see Chapter DTD for more information).

An XML Entity must be created in the Document Type Definition (DTD). The syntax of declaring entity is:

```
<!ENTITY entity_name "the text you want to be replaced">
```

figure A.III.5.1 – Entity creation

```
<!ENTITY chickenrice "A Chicken drumstick steamed with hot white rice">
```

figure A.III.5.2 Example Entity declaration

An entity after being created in DTD could be referred in the elements. We enclose the entity_name between & and ; respectively to make a reference to that entity.

```
<!ENTITY chickenrice "A Chicken drumstick steamed with hot white rice">  
<DESCRIPTION>&chickenrice;</DESCRIPTION>
```

figure A.III.5.3 Example Entity declaration and using

This example is equivalent to

```
<DESCRIPTION>A Chicken drumstick steamed with hot white rice</DESCRIPTION>
```

figure A.III.5.4 Example without Entity, but this way will cost time repeating the whole sentence

These following entities are predefined in XML, which every XML processor should have the ability to recognize these entities (StylusStudio.com, 2011). However, for the safe of interoperability, a valid XML document should declare these entities like any other entities before referring to them. The 5 predefined entities in XML are in figure A.III.5.5

Entities	Stands for
<	<
>	>
&	&
'	'
"	"

figure A.III.5.5 – predefined entities

6. Comments

In every programming language, comments are very important because they are the parts that the developer can explain in words what they are proposing to do. Comments in XML are simply texts that are completely ignored by the XML parser, and thus we can put anything in comments. According to World Wide Web Consortium (2011), a comment should begin with the `<!--` flag and end with `-->` flag as in figure A.III.6.1.

```
<!--this is an XML comment-->
```

figureA. III.6.1

```
<!--define the location of the external DTD using a relative URL address-->
```

figure A.III.6.2

```
<!--define the location of the external DTD using a relative URL address-->  
<!--define the location of the external DTD using a relative URL address-->  
<!--define the location of the external DTD--using a relative URL address-->
```

Figure A.III.6.3

In figure A.III.6.2, this is a valid comment in XML, whereas in figure A.III.6.3, there are some invalid comments in XML. We should remember that the string "--" (double-hyphen) must not occur within comments, which may cause troubles in parsing the document with processors. (Holzner, 1998)

7. Processing Instructions

Processing Instructions (PIs) enable us to pass additional information on to the applications using our XML, beyond the XML Parser. When the XML parser is reading XML document, and it encounters a PI, it simply hands that information off to the application. PIs are not a part of the document's character data and thus the parser doesn't care about them. PIs take the form as in figure A.III.7.1.

```
<?TARGET content ?>
```

Figure A.III.7.1

The target is similar to the name of the Processing Instruction, and is used to identify what application the instruction belongs to. The content is the instruction for that application for how it should perform. Importantly, we cannot use any combination of the letter xml for the name of target since it is reserved for W3C (Tay, 2003). Actually, an XML declaration is also considered as a processing instruction as in figure A.III.7.2.

```
<?xml version="1.0" ?>
```

Figure III.7.2

The target indicates that the instruction applies to xml, and the information provided is the version of XML we are using. Figure A.III.7.3 is another example of processing instruction.

```
<?image-gif scale="50%" ?>
```

Figure A.III.7.3

Figure A.III.7.3 is a PI that processes the image-gif image and this indicates that how image are to be scaled by the application.

8. Characters

A character is an atomic unit of text. Legal characters are tab, carriage return, line feed, and the legal characters of Unicode defined by the International Standard ISO/IEC 10646. We also can use character reference in XML as in figure A.III.8.1. A character reference refers to a specific character in the ISO/IEC 10646 character set. (Tay, 2003)

```
<!--nnnn is a code point in decimal form-->
<character>#nnnn;</character>

<!--hhhh is a code point in hexadecimal form-->
<character>#xhhhh;</character>
```

Figure A.III.8.1

For example, instead of typing dollar sign, we can use character that refers to the dollar symbol as in figure A.III.8.2

```
<price>#36; 4.95</price>
<price>#x24; 4.95</price>
```

Figure A.III.8.2

In this example, the decimal code for dollar sign is 36, and the hexadecimal code for dollar sign is 24. Therefore instead of typing \$, we can define the symbol by character reference $ or $.

9. White spaces

According to World Wide Web Consortium (2011), white space consists of one or more space () characters, carriage returns, line feeds, or tabs. Figure A.III.9.1 is the example of white space.

```
<!--description element has whitespace content-->

<description>          </description>

<description>
</description>
```

Figure A.III.9.1

10. Naming XML elements

These rules describe below are not solely used for XML elements, but we apply this for any XML component that needs a valid XML name. (Holzner, 1998)

First, XML names may contain any alphanumeric character. This includes the Standard English letters as well as the digits. XML names may also include non-English letters, numbers, and ideograms. They may also include the period hyphens, underscores, or colons as in figure A.III.10.1.

```
<!--The period may be used in an element name-->
<restaurant.collection>

<!--The hyphen may be used in an element name-->
<restaurant-collection>

<!--The underscore may be used in an element name-->
<restaurant_collection>

<!--The colon may be used in an element name-->
<restaurant:collection>
```

Figure A.III.10.1

We notice that the colon is a valid character to be used in XML names; however, it is also a special character that is reserved for use with XML Namespaces. There for we should avoid using colon in our element names. XML Namespaces will be cover later in this chapter.

XML names may not contain other punctuation characters such as quotation marks, apostrophes, dollar signs, ampersand, semi-colon, etc. XML names should not begin with a number and may not contain whitespace of any kind. Finally, all names beginning with the string XML (in any combination of case) are reserved for W3C specifications, according to World Wide Web Consortium (2011). Figure A.III.10.2 is some valid XML names, whereas figure A.III.10.3 is some invalid XML names.

```
<!--Digits are acceptable within an element name-->  
<item4sale>  
  
<!--Hyphens are fine after letters-->  
<restaurant-collection>  
  
<!--Accents and diacritics are acceptable-->  
<nhà hàng>
```

Figure A.III.10.2

```
<!--Names must not begin with a digit-->  
<4sale>  
  
<!--Names cannot start with special characters such as a hyphen or period-->  
<.restaurant>  
  
<!--Accents Names cannot contain special characters other than the period,  
hyphen, underscore, and colon-->  
<restaurant+collection>
```

Figure A.III.10.3

IV. Namespaces in XML

1. Introduction to Namespaces

First, we use Namespaces to keep track of components and to ensure that people don't create elements and attributes that conflict with each other.

In figure A.IV.1.1, we have an XML document for storing restaurant branch. This is a straightforward XML document with a few elements to store restaurant branch, with picture and menu element for the actual data. (Tay, 2003)

```
<branch branch_id="10" address="87 Kent Ridge Road" city="Singapore" hotline="66109666">
  <picture source="www.flickr.com/kentridge/logo.gif" type="image-gif"/>
  <menu>
    <item item_id="100">
      <name>Berry-Berry Belgian Waffles</name>
      <price>&sgd; 7.95</price>
      <type course = "Appetizer"/>
    </item>
  </menu>
</branch>
```

Figure A.IV.1.1

If we wanted to include some HTML mark-ups along with our XML data, like in figure A.IV.1.2

```
<branch branch_id="10" address="87 Kent Ridge Road" city="Singapore" hotline="66109666">
  <html>
    <body>
      <picture source="www.flickr.com/kentridge/logo.gif" type="image-gif"/>
      <menu>
        <item item_id="100">
          <menu>
            <li><name>Berry-Berry Belgian Waffles</name></li>
            <li><price>&sgd; 7.95</price></li>
            <li><type course = "Appetizer"/></li>
          </menu>
        </item>
      </menu>
    </body>
  </html>
</branch>
```

Figure A.IV.1.2

The preceding example has a big problem: two <menu> tags. One of the <menu> tags represents the HTML <menu> whereas the other is the <menu> element for our XML

restaurant branch. When we mix the two together, there is no way for an XML parser or a browser to tell the difference between the two (Holzner, 1998). That is why we need Namespaces.

2. Declaring Namespaces

Namespaces are specified using the `xmlns` attribute to declare the Namespace with an element. The attribute takes the general form as in figure A.IV.2.1

```
<element xmlns:prefix="Namespace URI">
```

Figure A.IV.2.1

element is the element for which we are declaring the Namespace, the *prefix* is the prefix we will use with our element names, and the *Namespace URI* which may be a URL or simply a string, is the identifier of the Namespace itself. The prefix is optional, if it is left out, the Namespace declaration is considered the default Namespace, and all the elements in the document will be treated as members of that Namespace unless they are marked otherwise. (Williamson, 2001)

```
<restaurant_collection xmlns="my_default_namespace">
```

Figure A.IV.2.2

In figure A.IV.2.2 we declare `restaurant_collection` as our default namespace.

3. Namespaces in XML Documents

i. Default Namespaces

According to World Wide Web Consortium (2011), to declare a default Namespace for your document, we simply use the `xmlns` attribute with the document's root element as in figure A.IV.3.1

```
<?xml version="1.0"?>
<restaurant xmlns="my_default_namespace">
  <branch branch_id="10" address="87 Kent Ridge Road" city="Singapore" hotline="66109666">
    <picture source="www.flickr.com/kentridge/logo.gif" type="image-gif"/>
    <menu>
      <item item_id="100">
        <name>Berry-Berry Belgian Waffles</name>
        <price>&sgd; 7.95</price>
        <type course = "Appetizer"/>
      </item>
    </menu>
  </branch>
</restaurant>
```

Figure A.IV.3.1

ii. Multiple Prefixed Namespaces

To make all the Namespaces in the document require a prefix, we use the same basic structure as declaring a default Namespace, only we add the : *prefix* to the xmlns attribute as in figure A.IV.3.2 (World Wide Web Consortium, 2011)

```
<?xml version="1.0"?>
<branch:restaurant xmlns:branch="my_branch_namespace"
  xmlns:html="standard_html ">
  <branch:branch branch_id="10" address="87 Kent Ridge Road" city="Singapore" hotline="66109666">
    <html:html>
      <html:body>
        <branch:picture source="www.flickr.com/kentridge/logo.gif" type="image-gif"/>
        <branch:menu>
          <branch:item item_id="100">
            <html:menu>
              <html:li>< branch:name>Berry-Berry Belgian Waffles</ branch:name></html:li>
              <html:li>< branch:price>&sgd; 7.95</ branch:price></html:li>
              <html:li>< branch:type course = "Appetizer"/></html:li>
            </html:menu>
          </branch:item>
        </branch:menu>
      </html:body>
    </html:html>
  </branch:branch>
</branch:restaurant>
```

Figure A.IV.3.2

In figure A.IV.3.2, we declare two Namespaces in the root element (element restaurant in this case) which are branch Namespace and html Namespace. Therefore, in the XML document any element starts with the prefix branch will be considered a child element of the root element restaurant, and any element starts with the prefix html will be considered as an HTML markup part. We also notice that *the root element restaurant also has the Namespace prefix*.

iii. Default and Prefixed Namespaces

We declare both Namespaces globally by declaring them in the root element. The default Namespace applies to any element without a prefix in the document (Holzner, 1998). Besides, any element in the document can be made part of that Namespace by appending the prefix as in figure A.IV.3.3

```

<?xml version="1.0"?>
<restaurant xmlns="my_branch_namespace"
  xmlns:html="standard_html ">
  <branch branch_id="10" address="87 Kent Ridge Road" city="Singapore" hotline="66109666">
    <html:html>
      <html:body>
        <picture source="www.flickr.com/kentridge/logo.gif" type="image-gif"/>
        <menu>
          <item item_id="100">
            <html:menu>
              <html:li><name>Berry-Berry Belgian Waffles</name></html:li>
              <html:li><price>&sgd; 7.95</price></html:li>
              <html:li><type course = "Appetizer"/></html:li>
            </html:menu>
          </item>
        </menu>
      </html:body>
    </html:html>
  </branch>
</restaurant>

```

Figure A.IV.3.3

In figure A.IV.3.3, we declare restaurant as a default Namespace by declaring a namespace without the prefix. Then any element does not start with any prefix will be considered an element of the restaurant element. On the other hand, we also globally declare html Namespace by specifying its prefix in the root element of the XML document and thus any element starts with html prefix will follow the HTML standards. (Tay, 2003)

We could also declare the default Namespace globally, but then have the second Namespace declared locally as in figure A.IV.3.4

```

<?xml version="1.0"?>
<restaurant xmlns="my_branch_namespace">
  <branch branch_id="10" address="87 Kent Ridge Road" city="Singapore" hotline="66109666">
    <picture source="www.flickr.com/kentridge/logo.gif" type="image-gif"/>
    <menu>
      <item item_id="100">
        <html:html xmlns:html="standard_html ">
          <html:menu>
            <html:li><name>Berry-Berry Belgian Waffles</name></html:li>
            <html:li><price>&sgd; 7.95</price></html:li>
            <html:li><type course = "Appetizer"/></html:li>
          </html:menu>
        </html:html>
      </item>
    </menu>
  </branch>
</restaurant>

```

Figure A.IV.3.4

In figure A.IV.3.4, we declare the html Namespace locally by declaring it inside the element named item. Therefore we are just able to use any HTML markup inside the item element.

We could also eliminate the prefix altogether by declaring a local default Namespace as in figure A.IV.3.5

```
<?xml version="1.0"?>
<restaurant xmlns="my_branch_namespace">
  <branch branch_id="10" address="87 Kent Ridge Road" city="Singapore" hotline="66109666">
    <picture source="www.flickr.com/kentridge/logo.gif" type="image-gif"/>
    <menu>
      <item item_id="100">
        <html xmlns="standard_html ">
          <menu>
            <li><name>Berry-Berry Belgian Waffles</name></li>
            <li><price>&sgd; 7.95</price></li>
            <li><type course = "Appetizer"/></li>
          </menu>
        </html>
      </item>
    </menu>
  </branch>
</restaurant>
```

Figure A.IV.3.5

V. Content Model in XML

A content model is a specification of the internal structure of an element's content. Below is some content model in XML, its definition and examples.

1. Empty Content Model

According to World Wide Web Consortium (2011), the empty content model which is the simplest content model indicates that the parent element has no sub-elements and no character data content. We will explore the definition of character data content in chapter DTD. Figure A.V.1.1 and figure A.V.2.2 are two examples of empty content model in XML.

```
<Restaurant/>
```

Figure A.V.1.1

```
<Restaurant id="restaurant1"/>
```

Figure A.V.1.2

Empty content model may or may not have the attribute. Figure A.V.1.1 is the empty content model without attribute, whereas figure A.V.1.2 is the empty content model with an attribute.

2. Any Content Model

Any content model allows the element to contain parsed character data or any other elements as content (Holzner, 1998). We will discuss parsed character data in chapter DTD. Figure A.V.2.1 is example of any content model in XML.

```
<picture source="www.flickr.com/kentridge/logo.gif" ftype="image-gif">
  This is a picture
<price>$7</price>
</picture>
```

Figure A.V.2.1

In figure A.V.2.1 the content of element picture contains a parsed character data (this is a picture) and other element (price element).

3. PCDATA Content Model

The PCDATA content model allows the element to contain only parsed character data. We will cover PCDATA in more detail in chapter DTD. Figure A.V.3.1 illustrates PCDATA Content Model.

```
<name>Strawberry Belgian Waffles</name>
```

VI. Well-formed XML

XML document must be well-formed in order to be considered as XML. According to World Wide Web Consortium (2011), the XML 1.0 Recommendation states all conditions that the XML must follow which are also called constraints. If the document fails to meet these constraints, it is not an XML document. (Holzner, 1998)

There are some important rules that make our XML document well-formed:

1. Naming Conventions

All element and attribute names must follow the conventions for XML naming that we have mentioned in Naming XML elements part. Figure A.VI.1.1 indicates a well-formed element and a non well-formed one. (Tay, 2003)

```
<!--This is a well-formed element. Hyphens are fine after letters-->
<restaurant-collection>

<!--This is not a well-formed document since it violates the naming convention-->
<-restaurant-collection>
```

Figure A.VI.1.1

2. Proper structure

Elements must be properly nested in figure A.VI.2.1. However, figure A.VI.2.2 is not a good XML document.

```
<restaurant_collection>
  <restaurant>
    <rname>Kent Ridge Highlights</rname>
    <branch></branch>
  </restaurant>
</restaurant_collection>
```

Figure A.VI.2.1

```
1: <restaurant_collection>
2: <restaurant>
3: <rname>Kent Ridge Highlights</rname>
4: <branch>
5: </restaurant>
6: </branch>
7: </restaurant_collection>
```

Figure A.VI.2.2

As we see in figure A.VI.2.2, restaurant and branch element are not properly nested. As in line 5 and 6 <restaurant> and <branch> tag are not closed in the correct order. We should close the <branch> tag before closing <restaurant> tag.

3. Proper Tag Pairing & Matching

Every start tag must have an end tag, or take the form of the empty element. We have mentioned the syntax of tag of XML elements in the previous part. Figure A.VI.3.1 is the example of wrong tag in XML

```
<!--branch tag does not close-->
<restaurant_collection>
  <restaurant>
    <rname>Kent Ridge Highlights</rname>
    <branch>
  </restaurant>
</restaurant_collection>
```

Figure A.VI.3.1

All tags must properly match case as in figure A.VI.3.2

```
<!--tags properly match case-->
<restaurant_collection>
</restaurant_collection>

<!--tags dose not properly match case-->
<restaurant_collection>
</Restaurant_Collection>
```

Figure A.VI.3.2

4. Root Element: One and Only One

A well-formed document must have one and only one root element. The root element is not necessary to be big, but it must contain all the other elements in the XML document. As we have discussed in the 2.B Structure of XML Document, the examples in figures A.VI.4.1 are samples of well-formed and non well-formed XML Documents:

```

<!--well-formed XML document-->
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<restaurant_collection>
  <branch>
  </branch>
</restaurant_collection>

<!-- non well-formed XML document -->
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<branch branch_id="10" address="87 Kent Ridge Road" city="Singapore" hotline="66109666">
</branch>
<branch <branch branch_id="20" address="145 Thompson Road" city="Singapore" hotline="66106999">>
</branch>

```

Figure A.VI.4.1

5. Proper Entity References

All entities must be properly referenced. All the entities which are referenced in the XML document should be declared correctly in the DTD part (Williamson, 2001). The entities which are not properly referenced will cause the error in well-form checking of the XML Document. Figure A.VI.5.2 shows the correct and error examples of an XML Document when the entities are used to reference. The declaration has been made in the DTD part in figure A.VI.5.1

```
<!ENTITY sgd "&#x24;";>
```

Figure A.VI.5.1

```

<!--correct entity reference -->
<price>&sgd; 1.95</price>

<!--incorrect entity reference -->
<price>&sg; 1.95</price>

<!--incorrect entity reference -->
<price>&usd; 1.95</price>

```

Figure A.VI.5.2

The XML document is said to be valid must satisfy the some requirements. First of all it must be well-formed XML document. Then the document must refer to a Document Type

declaration which follows all the DTD rules. This part will be discussed carefully in the chapter of Document Type declaration. (Simon St. Laurent, Robert Biggar, 1999).

VII. Example Explanation

We look at the first part of the XML example in the appendix A:

```
<restaurant_collection>
<restaurant>
<name>Kent Ridge Highlights</name>
<rtype type_of_restaurant="FastFood" type_of_cuisine="Western Exclusive Cooking
Styles"/>

<!--first branch of Kent Ridge Highlights-->
<branch branch_id="branch01" address="87 Kent Ridge Road" city="Singapore"
hotline="66109666">
<picture title="branch logo representation" source="www.flickr.com/kentridge/logo.gif"
ftype="image-gif"/>
<menu>
<item item_id="item100">
<name>Berry-Berry Belgian Waffles</name>
<type course = "Appetizer"/>
<price>&sgd; 7.95</price>
<description>light Belgian waffles covered with an assortment of fresh berries and whipped
cream</description>
<picture title="item num 100" source="www.flickr.com/kentridge/item100.png"
ftype="image-svg"/>
</item>
<item item_id="item101">
<name>Strawberry Belgian Waffles</name>
<type course = "Appetizer"/>
<price>&sgd; 6.50</price>
<description>light Belgian waffles covered with strawberries and whipped
cream</description>
<picture title="item num 101" source="www.flickr.com/kentridge/item101.png"
ftype="image-svg"/>
</item>
<item item_id="item102">
<name>Homestyle Exclusive</name>
<type course = "MainDish"/>
<price>&sgd; 12.50</price>
<description>two eggs, bacon or sausage, toast, and our ever-popular hash
browns</description>
<picture title="item num 102" source="www.flickr.com/kentridge/item102.png"
ftype="image-svg"/>
</item>
<item item_id="item103">
<name>Sweet Tapioca</name>
<type course = "Dessert"/>
<price>&sgd; 4.95</price>
<description>Steamed Sweet Tapioca with Coconut Creame</description>
</item>
<item item_id="item104">
```

```

<name>Coke</name>
<type course = "Drink"/>
<price>&sgd; 1.95</price>
</item>
</menu>
</branch>

<!--this is second branch-->
<branch branch_id="branch02" address="145 Thompson Road" city="Singapore"
hotline="66106999">
<picture title="restaurant logo representation" source="www.flickr.com/kentridge/logo.gif"
ftype="image-gif"/>
<picture title="branch logo representation" source="www.flickr.com/kentridge/20/logo.gif"
ftype="image-gif"/>
<menu>
<item item_id="item200">
<name>Belgian Waffles</name>
<type course = "Appetizer"/>
<price>&sgd; 4.95</price>
<description>two of our famous Belgian Waffles with plenty of real maple
syrup</description>
<picture title="item num 200" source="www.flickr.com/kentridge/item200.png"
ftype="image-svg"/>
</item>
<item item_id="item201">
<name>French Toast</name>
<type course = "Appetizer"/>
<price>&sgd; 4.50</price>
<description>thick slices made from our homemade sourdough bread</description>
</item>
<item item_id="item202">
<name>Black Beans And Red Peppers</name>
<type course = "MainDish"/>
<price>&sgd; 6.80</price>
<picture title="item num 202" source="www.flickr.com/kentridge/item202.png"
ftype="image-svg"/>
</item>
<item item_id="item203">
<name>Chicken Steam Hotplate</name>
<type course = "MainDish"/>
<price>&sgd; 8.70</price>
<description>A Chicken drumstick steamed with hot white rice</description>
<picture title="item num 203" source="www.flickr.com/kentridge/item203.png"
ftype="image-svg"/>
</item>
<item item_id="item204">
<name>Chocolate Ice Cream</name>
<type course = "Dessert"/>
<price>&sgd; 2.50</price>
</item>

```

```

<item item_id="item205">
<name>Chocolate Caramel Pecan ' Cheesecake</name>
<type course = "Dessert"/>
<price>&sgd; 2.70</price>
</item>
<item item_id="item206">
<name>Milk Shake</name>
<type course = "Drink"/>
<price>&sgd; 3.95</price>
</item>
<item item_id="item207">
<name>
<![CDATA[Fruit juice (orange, apple, lime & mango) ]]>
</name>
<type course = "Drink"/>
<price>&sgd; 2.20</price>
</item>
<set price="&sgd; 25.9">
<item_set item_id="item200 item203 item204 item206"/>
<gift>
<toy name="Ball pen and notebook">an exclusive ball pen and notebook with logo of our
restaurant
<picture source="www.skyphotos.net/toy.jpg" ftype="image-jpg "/>
</toy>
</gift>
</set>
<set price="&sgd; 30.8">
<item_set item_id="item201 item202 item203 item205 item207"/>
<gift>
<voucher id="voucher220" value="&sgd; 10" expired="one month"/>
</gift>
</set>
<set price="15.8">
<item_set item_id="item200 item203 item207"/>
</set>
</menu>
</branch>

</restaurant>

...

</restaurant_collection>

```

In this chapter we will look through a complete example of XML Document, an example of the “Restaurant Collection”. The “Restaurant Collection” has 5 restaurants, in this Chapter the first restaurant will be discussed and explained.

The First Restaurant has information is within the tags <restaurant> </restaurant>. The Restaurant has elements, first the name is *Kent Ridge Highlights* in the tags <name>

</rname>. The type of restaurant is *FastFood* and the type of cuisine is *Western Exclusive Cooking Styles*, they are attributes of the single tag <rtype/>. Moreover, the restaurant also has 2 branches, which are included within tags <branch> and </branch>.

The tag <branch> has attributes, which show the additional information of the branch. For the first branch, the ID of the branch is *branch01*, this must be unique among all the branches (we will see this later in the DTD Chapter). The address is *87 Kent Ridge Road*, city is *Singapore*, hotline is *66109966*. This branch has 2 elements, the first is picture, a single tag with attributes. The attributes show more info about the picture: title *branch logo representation*, source *www.flickr.com/kentridge/logo.gif* the type of image is *image-gif*. Another element is menu, which is within the tag <menu>.

The menu of the branch01 of the first restaurant has 5 items, represent for 5 dishes. 5 dishes have the same kinds of child elements. For instance, the first dish, the item ID is *item100*, which also is unique. The name of the item is *Berry-Berry Belgian Waffles*, type is *Appetizer* (attribute), and price is *7.95* and followed by the description of the dish. The last element of the item is the single tag <picture/> with attributes show the information of the picture is title = *item num 100* source *www.flickr.com/kentridge/item100.png* and type *image-svg*. The same information applied for other dishes.

The second branch also has similar features, except that it basically has 2 pictures. This is allowed because the declaration of the usage of the XML Document depends on the DTD (which is covered in the DTD Chapter). In the 8th item, there is one special data, that is <![CDATA[Fruit juice (orange, apple, lime & mango)]]>. The CDATA protects the use of ampersand ‘&’ as it will not be parsed by the parser and will not cause any error. In this branch, in addition to the items as the first branch, it also has another type of element that is <set>.

The <set> pair tags has the attribute shows the list of the items it comprises within the items of the branch. For example at the first set, the items are *item200 item203 item204 item206*. There are also another element that is <gift>. The gift may be a <toy> or a <voucher> (as in the second set). The <toy> element has additional information name as its attribute. Moreover, it also has another child element, which is picture. The picture in this <set> tag is also the same type of the picture of the branch.

For the second <set>, the <voucher> child element only has attributes. They are id *voucher220* value *10* and expired in *one month*.

VIII. Conclusion

XML (eXtensible Markup Language) is a Markup Language, but itself is also a standardized and self-describing way of encoding data. XML is strong and simple enough to be used as a common format for exchanging data through various platforms, languages and applications. An XML document is well-formed and structured as a tree, so that it is easy to access and organize the elements. Therefore, XML is actually a good approach for the database solutions.

From XML, today there are many good XML-based technologies: DTD, XQuery, XML Schema and XPath. Furthermore, XML is also the platform for some well-known applications:

- XHTML: how HTML is supported and extended from XML, the future of Web technology
- RDF (Resource Description Framework): The framework to form the resources on the Internet.
- RSS (Really Simple Syndication): the family of web feeds using XML as a platform

Chapter B: Document Type Definition

I. Introduction

Every XML document must be associated with exactly one Document Type Definition (DTD). Basically, DTD is the set of rules that defines the structure, the allowable contents of a XML document. To be more specific, DTD defines the types, attributes and the constraints that every element in the XML document must follow.

XML parsers check whether a XML document is well-formed based on the constraints specified in DTD document.

In this chapter we are going to discuss:

Part II – Document Type Declaration: How we can associate a XML document with a DTD document.

Part III – Document Structure: Various different types and attributes of element - the main block of data of an XML document.

Part IV – Document Content: Usage of entity and notation – shortcuts to any internal or external content for the XML document.

Part V – XML Document Validation: How we can check the validity of a XML document based on its associating DTD.

Part VI – An Overview on the book Document Type Definition – Restaurant Collections

II. Document type declaration

Document Type Declaration is the process of associating the XML documents with the Document Type Definition file. It can contains the directly the DTD, or references to an external files containing the DTD, or both.

Document Type Declaration is an essential and must-be unique part in a XML document. Its position is between the XML prolog and the root element of the document.

As listed above, there are two different ways for Document Type Declaration. The DTD directly contained in the declaration is knows as Internal DTD, whereas the latter is External DTD.

1. Internal DTD

The declaration of internal DTD begins with the keyword “!DOCTYPE ”, followed by the name of root element. Moreover, Element declaration can be added, which specifies the child elements that root element has. Details of syntax of declaration of internal DTD are given in Figure B.II.1.1 (Simon St. Laurent, Robert Biggar, 1999)

```
<!DOCTYPE root-element [element-declarations]>
```

Figure B.II.1.1: Format for Internal DTD declaration

```
<!DOCTYPE restaurant [  
  <!ELEMENT restaurant (name , address, type)>  
  <!ELEMENT name (#PCDATA)>  
  <!ELEMENT address(#PCDATA)>  
  <!ELEMENT type (#PCDATA)>  
>
```

Figure B.II.1.2: Example of Internal DTD declaration

Figure B.II.1.2 shows an example of Internal DTD declaration. The root element restaurant is declared to have another 3 child elements called name, address, and type respectively. For the details in the syntax as well as the meaning of these elements, we will discuss them in the later part of this chapter. For now, a small part of an XML document is given in Figure B.II.1.3 so that you can understand better the effect of the DTD in Figure B.II.1.2


```
<restaurant>
<name> Asian Cuisine </name>
<address> 380 KentRidge Road </address>
<type > fastfood </type>
</restaurant>
```

Figure B.II.1.3: Corresponding XML document to the DTD in Figure B.II.1.2

2. External DTD

Similar to internal DTD, the syntax for external begins with “!DOCTYPE ” followed by name of root element, then key word SYSTEM or PUBLIC. If the reading of the file requires more than 1 application, public identifier is required. At the end, the URI for DTD file is compulsory. Specifically, we have 3 different ways to declare the external Document Type Definition.

The first type uses the SYSTEM identifier right after the key word “!DOCTYPE” and the root element name, and followed by the URI referencing the external file containing the DTD. This is shown in Figure B.II.2.1 (Simon St. Laurent, Robert Biggar, 1999)

```
<!DOCTYPE rootElement SYSTEM "URIforExternalSubset">
```

Figure B.II.2.1: Format for External DTD Declaration using SYSTEM identifier

Figure B.II.2.2 and B.II.2.3 respectively give two examples on the usage of SYSTEM identifier with different viewpoint of the URI. Figure B.II.2.2 declares the root element McDonald with the DTD containing in the file McDonald.dtd store in the same folder as the XML document. On the other hand, Figure B.B.II.2.3 gives a specific online URL to the restaurant.dtd file for the DTD of RESTAURANT root element.

```
<!DOCTYPE McDonald SYSTEM "McDonald.dtd">
```

Figure B.II.2.2: Example of External DTD declaration using System identifier

```
<!DOCTYPE RESTAURANT SYSTEM "http://www.example.com/restaurant.dtd">
```

Figure B.II.2.3: Example of External DTD declaration using System identifier

The other 2 types include the use of PUBLIC identifier, which is the document processing used to identify a document type that spans more than 1 application. According to (Williamson, 2001), Public Identifier is intended to all systems that read it. Public identifier consists of 3 different parts separated by double forward slashes with the syntax given in Figure B.II.2.4.

"-//Owner Identifier//Text Identifier or Organization Name//Language"

Figure B.II.2.4: Format for Public identifier

Basically, Public identifier starts with a hyphen and a double forward slashes, and then it comes the first part that contains the Owner identifier. If the owner has not been "registered" then it will be replaced by a hyphen. After that, if the owner has been recognized, then second part contains only text identifier (such as DTD or ENTITIES...). If not, it would contain the name of the organization that public that DTD. And finally, identifier ends with the language used (EN for English, FR for French, etc.). Last but not least, the public identifier must be contained with two double quote characters.

Figure B.II.2.5 shows the detailed syntax of the first type of using PUBLIC identifier. Starting with the "<!DOCTYPE" keyword and the root element name, this declaration then followed by the "PUBLIC" keyword, the public identifier, and finally the URI to the external file of the DTD.

<!DOCTYPE *rootElement* PUBLIC "*PublicIdentifier*" "*URIforExternalSubset*">

Figure B.II.2.5: Format for External DTD Declaration using PUBLIC identifier

Following the syntax in Figure B.II.2.5, an example is given in Figure B.II.2.6 which shows the declaration of root element RESTAURANT using the external DTD available at the URL <http://www.example.com/restaurant.dtd> and the public identifier with the owner identifier of EXAMPLE and text identifier as DTD RESTAURANT using English as the language.

**<!DOCTYPE RESTAURANT PUBLIC "-//EXAMPLE//DTD RESTAURANT//EN"
" <http://www.example.com/restaurant.dtd> ">**

Figure B.II.2.6: Example of External DTD declaration using Public identifier

When the parser is able to interpret the actual public identifier to DTD, we do not need to use the URL. That is our third type of external DTD declaration using only the public identifier. Details are shown in Figure B.II.2.7

```
<!DOCTYPE rootElement PUBLIC "PublicIdentifier">
```

Figure B.II.2.7: Format for External DTD declaration using PUBLIC identifier

Figure B.II.2.8 give one similar example to the one in Figure B.II.2.6 is given, provided that this public identifier is recognized and parse-able by the parser.

```
<!DOCTYPE RESTAURANT PUBLIC "-//EXAMPLE//DTD RESTAURANT//EN">
```

Figure B.II.2.8: Example of External DTD Declaration using Public identifier

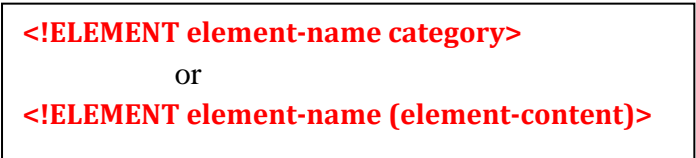
III. Document Structure

1. Element:

As described in the previous chapter, Element is the main content block of XML which can contain text, other elements, combination of text and other elements or nothing.

Element can be declared only once in the DTD document. Element declaration means specifying the name and its allowable contents (text, other child element, or both). The elements in the XML document should follow the defined rule for that element.

Element declaration begins with keyword “!ELEMENT”, then element name and element’s category (EMPTY, ANY ...) or element’s content (text data or child elements). General syntax is described in Figure B.III.1.1 (W3Schools, 1999)



```
<!ELEMENT element-name category>  
or  
<!ELEMENT element-name (element-content)>
```

Figure B.III.1.1: Format for Element declaration

The followings specifying syntax for each kind of elements:

i. Empty Element

As the element is empty, we should put key word EMPTY into the (element-content) of the general syntax in Figure B.III.1.1. Figure B.III.1.2 describes the syntax for declaring empty element (W3Schools, 1999).



```
<!ELEMENT element-name EMPTY>
```

Figure B.III.1.2: EMPTY element declaration

A classic example of the corresponding part of the XML file would be `
`. Moreover, if we want the element just to contain attributes (e.g. an image or audio element) then we can let the content of the element empty.

ii. Element with only text content

For this category, the only allowed content is characters. And in order to specify an element is text-only content, we use the key word #PCDATA for the element content. As XML supports Unicode, characters in XML can be Chinese, Arabic, Vietnamese, etc. characters. Figure B.III.1.3 shows the declaration of text-content only element (W3Schools, 1999).

```
<!ELEMENT element-name (#PCDATA)>
```

Figure B.III.1.3: Text only element declaration

Figure B.III.1.4 and B.III.1.5 respectively give an example on using text only element and its corresponding part in XML document. Here, restaurant_name is a text-content only element, and when it is used in XML document, the text is written within the 2 opening and closing tags.

```
<!ELEMENT restaurant_name(#PCDATA)>
```

Figure B.III.1.4: Example on Text-only element

```
<restaurant_name> 中国面条 </restaurant_name>
```

Figure B.III.1.5: Corresponding part of III.1.4 in XML document

From the Figure B.III.1.5, 中国面条 contains some characters not in ASCII, however it is still acceptable as long as we declare the DTD to use the Chinese language.

iii. Element with only child elements

An element can contain other elements inside called child elements. The name of child elements should be declared in order as described in Figure B.III.1.6 (W3Schools, 1999)

```
F <!ELEMENT element-name (child1,child2,...)> ion
```

Similarly to previous section, Figure B.III.1.7 and B.III.1.8 show an example of using element with child elements and its corresponding part in the XML document. To be specific, Figure B.III.1.7 illustrates an element named restaurant with 3 other child elements name, address, and type. All 3 of them are text only element.

```
<!ELEMENT restaurant (name , address, type)>  
<!ELEMENT name (#PCDATA)>  
<!ELEMENT address(#PCDATA)>  
<!ELEMENT type (#PCDATA)>
```

Figure B.III.1.7: Example on Element with Child element content

```

<restaurant>
  <name> Asian Cuisine </name>
  <address> 380 KentRidge Road </address>
  <type > fastfood </type>
</restaurant>

```

Figure B.III.1.8: Corresponding part of III.1.7 in XML document

However, it is not always that every time we declare a new instance of an element, we want to put every piece of information into it. Some child elements would like to consider themselves as optional; some may need to appear more than once. Moreover, some child elements may not exist under the same parent element at the same time. We can actively resolve this problem by further specifying the restriction on number of occurrence of every child element. Figure B.III.1.9 shows in details the syntax rules for this restriction (Williamson, 2001).

A B	either A or B occurs
A,B	both A and B occur , in the specified order
A?	A can occur 0 or 1 time
A[*]	A occur 0 or more times , no upper limit
A⁺	A occur at least 1 time

Figure B.III.1.9: Rules on restriction for number of occurrence of child element

Here we would like to introduce some small example on the usage of the above rules.

Figure B.III.1.10 illustrates the declaration of element gift in which its child can be either a toy element or a voucher element by using the syntax A|B in the element content.

```

<!ELEMENT gift (TOY | VOUCHER )>

```

Figure B.III.1.10: Example of using number of child element occurrence restriction

Figure B.III.1.11 gives an example of element order in which its child elements are item, set and voucher. Note that for each child element, we have different restriction for them. According to the declaration in III.1.11, an order must have at least 1 item. In addition, the customer may want, but not essentially, to order more sets of meal. And finally he or she can use at most one voucher to pay.

<!ELEMENT order (item+, set*, voucher?)>

Figure B.III.1.11: Example of using number of child element occurrence restriction

iv. Element with any content

Once declared with the keyword ANY, an element can contain any kind of content: text, child elements or both. There is no restriction for the number of appearances of text or child elements. Syntax for any content-element is shown in Figure B.III.1.12 which is basically replacing the element-content with the key word ANY (W3Schools, 1999).

<!ELEMENT element-name ANY>

Figure B.III.1.12: ANY content Element declaration

Example of any element is described in Figure B.III.1.113 and B.III.1.14

**<!ELEMENT ITEM (NAME,PRICE)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT PRICE (#PCDATA)>

<!ELEMENT VOUCHER (VALUE)>
<!ELEMENT VALUE (#PCDATA)>

<!ELEMENT MENU ANY >**

Figure B.III.1.13: Example on Element with Any content

```
<MENU>
  <ITEM>
    <NAME>qwert</NAME>
    <PRICE>3.5</PRICE>
  </ITEM>

  <ITEM>
    <NAME>McDonald</NAME>
    <PRICE>2.5</PRICE>
  </ITEM>

  <VOUCHER>
    <VALUE>80</VALUE>
  </VOUCHER>

</MENU>
```

Figure B.III.1.14: Corresponding part of III.1.10in XML document

Figure B.III.1.13 shows that the element MENU is declared as ANY content element, so in the example of Figure B.III.1.14, it can have 2 ITEM elements and 1 VOUCHER element.

v. Element with mixed content

Similarly as in part d, mixed content element allows us to have a various kinds of the element content, be it the text #PCDATA or other child elements. However, by using mixed content element, we can have the power to set the specific of named elements to be included as the children.

Figure B.III.1.15 gives us the general syntax to declare a mixed content element. The element content has the structure of firstly the #PCDATA, and then other different child elements. All of them are separated by a straight slash (|) which indicates only one of them can be chosen. And finally, we put a * outside the brackets meaning the whole block can occur 0 or more times with no upper limit. This allows us to have as many text content and different child elements as possible regardless of their relative order. Also note that #PCDATA must come first in the declaration, otherwise it would return an error as lack of well-formness in XML syntax

```
<!ELEMENT element-name ( #PCDATA |child element1| child element2 |.....)*>
```

Figure B.III.1.15: Mixed content element declaration (Document Type Definition, 2011)

```
<!ELEMENT toy (#PCDATA | picture)*>
```

Figure B.III.1.16: Example of mixed content element

Figure B.III.1.16 gives us an example of mixed content element. It shows the declaration of toy element in which its content can be text, or a picture element or even both. And this does not have any limits on the number of occurrence. In other words, one may choose to use as many text descriptions and pictures as possible in order to best-describe the toy.

2. Attributes

An element can have 0 or more attributes, which specify its characteristics and provide further information about its contents or usage.

Very similar to element, since Attributes play an important role in enhancing the functionality and use of XML document, it is essential to have a proper declaration in DTD. That begins with keyword “!ATTLIST” then the NAME of that element. A list of attributes follows in the following order: Attribute name then attribute type. Default value for attributes

```
<!ATTLIST element-name attribute-name attribute-type default-value>
```


may also be added. General syntax to declare Attribute list is described in Figure B.III.2.1 (W3Schools, 1999)

Figure B.III.2.1: Format for attribute list declaration

Here we are interested in finding different usage of various data types provided for the attributes. According to (W3Schools, 1999), all the available attribute types and attribute default values are summarized in Figure B.II.2.2 and B.II.2.3 respectively.

Type	Description
CDATA	The value is character data
enumerated	The value must be one from an enumerated list (<i>en1 en2 ...</i>)
ID	The value is a unique id
IDREF or IDREFS	The value is the id of another element or a list of other ids
NMTOKEN or NMTOKENS	The value is a valid XML name or a list of valid XML names
ENTITY or ENTITIES	The value is an entity (we will see it in next chapter) or a list of entities
NOTATION	The value is a name of a notation (we will see it in next chapter)
xml:	The value is a predefined xml value

Figure B.II.2.2: List of available data types for Attribute

Value	Explanation
<i>Value</i>	The default value of the attribute
#REQUIRED	The attribute must be included
#IMPLIED	The attribute is optional
#FIXED <i>“value”</i>	The attribute value cannot be changed

Figure B.II.2.3: List of settings for attribute default values

Now we will go through every different data types. For each of them, we can discuss on the usage as well as different examples implementing together with the default value settings.

i. CDATA

Similarly to type #PCDATA of elements, type CDATA for attributes are allowed to contain characters only. However, CDATA is ignored by the XML parser while PCDATA is not.

An element of type PCDATA can be replaced by an EMPTY element with CDATA attribute if only text data is needed. As CDATA is ignored by xml parser, CDATA is used when the text contain a lot of special symbols e.g. “< “ or “&”. If these two symbols appear in the content of a PCDATA element, error will be generated.

An example on usage of CDATA is illustrated in Figure B.II.2.4

ii. Enumerated

A list of possible values is given. We can only choose one of these to be the value of attribute. Figure B.II.2.4 also helps to make it clearer on the usage of data type Enumerated.

iii. ID

ID attribute name must start with characters or underscore and cannot contain any spaces. Moreover, the attribute of ID type must be unique throughout the XML file. In other words, once an ID attribute is created, it becomes a global ID in the whole XML document. The significant of this kind of data type is that it allows other attributes to reference to it, hence reduce the redundancy in the XML document.

And if you are quick enough, you may have noticed that data type ID must go along with the setting #REQUIRED and can never be with #FIXED “value” due to its uniqueness in XML. Moreover, since the ID attribute helps to uniquely identify the element in the XML document, it is impossible to have 2 or more ID attributes concurrently for the same element.

Figure B.II.2.4 will help us to wrap up all the idea in the previous 3 sub-parts about CDATA, enumerated and ID data types.

<!ELEMENT BRANCH (#PCDATA)>				
<!ATTLIST BRANCH				
ADDRESS	ID	#REQUIRED		
CITY	CDATA	#FIXED	"Singapore"	
HOTLINE	CDATA	#IMPLIED		
RATING	(unrated 1 2 3 4 5)		"unrated"	
>				
Example of XML document :				
<BRANCH ADDRESS ="87 KentRidge Road" CITY="Singapore"				
HOTLINE="67846732" RATING="4">Mc Donald 1</BRANCH>				

Figure B.II.2.4: Example of Attribute List

Figure B.II.2.4 shows the declaration of a text-content element name BRANCH and its attribute list consisting of 4 different attributes: ADDRESS, CITY, HOTLINE and RATING. Since each restaurant branch can only locate at one position, its address can be made its ID attribute. Note that this ID attribute is required in every instance of this element, otherwise we could fail in distinguish two restaurant branch in the same city, happen to have same hotline number (maybe same handphone number of the owner) and have the same rating score. Moreover, the CITY attribute is fixed to the value "Singapore" as we are considering the context of this country. Hotline is text-content value, but it is not compulsory to have in every branch. And lastly, the rating for the branch can be chosen from an agreed standard, with the default value to be "unrated" for new restaurants.

After the example, we can see that if an attribute has data type as ID, then its setting must be #REQUIRED. However, if an attribute setting is #REQUIRED, it is not necessary to have data type of ID.

iv. IDREF or IDREFS:

When an attribute is of type IDREF, it is simply just has the same ID value of the previous-defined attributes of ID type. Same thing applies for IDREFS, yet this attribute can have multiple of values for the previous ID attributes (Williamson, 2001).

Figure B.II.2.5 and B.II.2.6 show us more different scenarios of making use of the data type IDREF and IDREFS. We can see that the SET element has two attributes, and the ITEM_ID is of data type IDREF, which can be referenced to any previous defined ID

attribute, in this case is ITEM_ID of the ITEM. So here SET is special to have the same item as ITEM, yet maybe different price. And SET can reference to an available instance of ITEM without having to create a new yet same one.

```

<!ELEMENT MENU ( ITEM+, SET* )>
<!ELEMENT ITEM EMPTY>
<!ATTLIST ITEM
    ITEM_ID      ID          #REQUIRED
    PRICE        CDATA       #REQUIRED
>
<!ELEMENT SET EMPTY>
<!ATTLIST SET
    ITEM_ID      IDREF       #REQUIRED
    PRICE        CDATA       #REQUIRED
>

```

Figure B.II.2.5: Example of Attribute List

```

<MENU>
  <ITEM ITEM_ID="I1" PRICE="3.5"/>
  <ITEM ITEM_ID="I2" PRICE="2.5" />
  <SET ITEM_ID="I2" PRICE="2.2" />
</MENU>

```

Figure B.II.2.6: Corresponding part of II.2.5 in XML document

Note that one of the weaknesses of XML is that it does not distinguish among the global ID attributes. And hence in the example of Figure B.II.2.5 and B.II.2.6, if we add another element BRANCH with an ID attribute BRANCH_ID, the attribute ITEM_ID of the SET element can still take BRANCH_ID as its value. Of course, this will not make any sense at all. So it depends on the user of the DTD to justify himself on the correct way to use IDREF.

In addition, we can use data type IDREFS for attribute ITEM_ID of the SET element meaning that this ITEM_ID can take multiple values of previous defined ITEM_ID of ITEM element.

```

<!ELEMENT SET EMPTY>
<!ATTLIST ITEM_SET
    ITEM_ID      IDREFS      #REQUIRED
    PRICE        CDATA       #REQUIRED
>

```

Figure B.II.2.7: Example of Attribute list

```

<MENU>
  <ITEM ITEM_ID="I1" PRICE="3.5"/>
  <ITEM ITEM_ID="I2" PRICE="2.5" />
  <SET ITEM_ID="I1 I2" PRICE="5.8" />
</MENU>

```

Figure B.II.2.8: Corresponding part of II.2.8 in XML document

This can be helpful if we want to represent the real-life set of meals where buying 2 or 3 items at the same time can help to reduce the total price. As shown in Figure B.II.2.7 and II.2.8, ITEM_ID of SET now has the data type of IDREFS and hence can take in more than 1 value of ITEM_ID of ITEM element.

v. NMTOKEN or NMTOKENS:

This type is used to restrict the use of values of attributes to valid xml names.

They are text that must begin with letters or underscore. They can contain underscore, letters or digits and must not have space (Williamson, 2001).

vi. ENTITY or ENTITIES

Entity may be a text string or an external object (image file, audio file ...). The attributes contain the defined entity name in DTD that links to that text or file. ENTITY links to one entity while ENTITIES link to several entities within the XML document. More details on Entity will be discussed later in the next part of this chapter.

vii. NOTATION

Similar to Entity, Notation data type allows us to reference to a declared notation name identifying the format used with non-XML information such as video, audio and image files. Details on syntax and usage of Notation will be discussed later.

Figure B.II.2.9 and B.II.2.10 help us to imagine the usage of ENTITY and NOTATION attribute types.

```

<!ELEMENT PICTURE EMPTY>
<![ATTLIST PICTURE
    SOURCE      ENTITY          #REQUIRED
    TYPE        NOTATION (image-gif)  #REQUIRED
>
<!NOTATION image-gif PUBLIC "image/gif">
<!ENTITY sample SYSTEM "sample.gif" NDATA image-gif>

```

Figure B.II.2.9: Example of Attribute list

```
<PICTURE SOURCE="sample" TYPE="image-gif" />
```

Figure B.II.2.10: Corresponding part of II.2.9 in XML document

As we may have notice there are two different types of text character in XML, one is #PCDATA used in declaration of element content, the other is CDATA used in declaration of attribute data type. What are the similarities and differences between them? Hence it comes the next part of this chapter.

4. XML Character Data

i. Parsed Character Data

XML documents are read and processed by a specific piece of software called an XML parser. When a document is processed by the XML parser, each character in the document is read, in order to create a representation of the data.

Any text that gets read by the parser is Parsed Character Data, or PCDATA. In the other word, PCDATA refers to texts stay within normal XML tags that could be extracted out. For example element content is considered either other elements or PCDATA and attribute values are considered as PCDATA.

PCDATA should not contain any &, < characters; these need to be represented by the & < entities respectively.

```
<name>Fruit juice (orange, apple, lime & mango)</name>
```

Figure II.3.1

```
<name>Fruit juice (orange, apple, lime &amp; mango)</name>
```

Figure II.3.2

In Figure B.II.3.1, the name element will cause an error since the parser will interpret & as a reference to the entity and in wrong syntax (&entity;). Therefore if we want to include this character, we have to use an equivalent entity, it is & in this case as in Figure B.II.3.2.

ii. Unparsed Character Data

In XML, there are some characters which are illegal, meaning that the XML parser will produce an error when reading these characters, like & and <

The character & refers to an entity, so that when it is not used properly for an entity, the processor will interpret wrongly and consider it as an error. Similarly, the character < is considered as the start of a new tag, as well as an element.

Therefore we can use CDATA section to denote the content that should not be parsed. A CDATA section starts with `<![CDATA[` and ends with `]]>` as in Figure B.V.2.1

```
<![CDATA[part should not be parsed]]>
```

Figure II.3.3

For example, instead of using equivalent entity to the character `&` like in Figure B.V.1.2, we can put the whole text in CDATA section as in Figure B.V.2.2. Therefore this part is not read by the parser and thus will not cause errors.

```
<name>  
  <![CDATA[Fruit juice (orange, apple, lime & mango)]]>  
</name>
```

Figure II.3.4

We can use CDATA section any where PCDATA occurs, for example element content. However, attribute values are always parsed unless they are specified as CDATA in DTD.

IV. Document Content

1. Notations:

i. Description:

According to (Simon St. Laurent, Robert Biggar, 1999), Notations are declarations made in DTD so that they are assignment of a label to some fixed content in a document. Since they are going to be used widely in DTD as well as the whole XML document applied that particular DTD, notations must be unique. And this has to hold in both internal and external subset of DTD.

ii. Usage:

Notation with its simple implementation can be used with various purposes. Yet it is usually used by the DTD authors to indentify file types.

Sometimes we want to use some file types that the XML parser does not support. Going on the old way will result an error notification when the parser reads through the document. This is when notations come in handy. Notation identifier references to a descriptive standard whose interpretation is left to the application, not the XML parser anymore. Only when the application can understand the standard, it can continue to process the information associated with that standard in later part of DTD as well as the XML document. However, even if the application does not support the standard, it just simply ignore the notations used and does not bother loading any external source declared in the notations. Significantly this will help to avoid time, bandwidth and data wasting. (Document Type Definition, 2011)

Other than this, notations can be used with attributes to identify resources referenced by element.

iii. Syntax:

Notation has two different ways of formatting. One can use either of the two kinds of identifiers SYSTEM and PUBLIC.

<!NOTATION *notationName* SYSTEM "*SystemIdentifier*">

Figure B.IV.1.1: Format for Notation Declaration using System identifier

Figure B.IV.1.1 shows the basic syntax for notation declaration using SYSTEM identifier. The format starts with the key word “<!NOTATION”, followed by the notation name used in the document, and then the key word “SYSTEM”, and then the system identifier indicating the data type this notation represents for, and finally ends with “>”.

One example is shown in Figure B.IV.1.2 on how to include GIF image data in your XML document by first introducing the data type GIF. The declaration first starts with “<!NOTATION” and then the name “gif”, followed by key word “SYSTEM” and finally “image/gif” indicating the data type “GIF” of the notation.

```
<!NOTATION gif SYSTEM "image/gif">
```

Figure B.IV.1.2: Example of Notation using System identifier

Moreover, there is another way of declaring Notations, which is to use PUBLIC identifier. The syntax is the same as above except that we are using “PUBLIC” instead of “SYSTEM” and we also need to include the public identifier and a specific URI to indicate the data type of notation. The details of this format is given in Figure B.IV.1.3

```
<!NOTATION notationName PUBLIC "PublicIdentifier" "URI">
```

Figure B.IV.1.3: Format for Notation Declaration using Public identifier

An example of declaring notation type-*image-svg* using public identifier is given in Figure B.IV.1.4. Here we use the public ID “-//W3C//DTD SVG 1.1//EN” and the URI “<http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd>” (Document Type Definition, 2011)

```
<!NOTATION type-image-svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
```

Figure B.IV.1.4: Example of Notation using Public identifier

Clearer example on usage of Notations will be discussed after we have looked into Entities.

2. Entities:

i. Description:

Entity is another markup declaration of XML, and it is part of the DTD (if any), in both internal and external subset. And it works in the same way as notation, after declared in DTD, it can be widely used in the rest of the DTD as well as the XML document applying that particular DTD. (Simon St. Laurent, Robert Biggar, 1999)

The main purpose of using Entities is for replacement of long or hard to remember content throughout the whole document. As a result, this helps a lot to increase the legibility of the document.

When we look into the classification of entities, there are several different views:

- + Internal and external entities
- + Parsed and unparsed entities

There is another type of entity which is called Parameter entity, yet is not a focus at the level of this book.

ii. Parsed Entities

a. Internal

According to (Document Type Definition, 2011), all internal entities are parsed entities. They associate a declared name (from any subset of DTD) with text content, so that the parser will replace those entity references immediately with the text content in the rest of DTD and document.

The syntax for declaring internal parsed entities is given in Figure B.IV.2.1 (Simon St. Laurent, Robert Biggar, 1999). It starts with the key word “<!ENTITY”, the entity name and finally its content within a pair of double quotes.

`<!ENTITY entityName "entityContent">`

Figure B.IV.2.1: Format for internal parsed entities

In order to make use of the declared entity in the document, we have to apply the syntax shown in Figure B.IV.2.2, starting with ampersand character (&), then the entity name, and finally the semi colon character (;).

`&entityName ;`

Figure B.IV.2.2: Syntax of using an internal parsed entity

Figure B.IV.2.3 shows us one example on the usage of internal parsed entities. If keeping writing the word “Appetizer” in a large sets of menu in a Restaurant collection is so tedious, one may consider declare an internal entity to help him/her.

in DTD: **<!ENTITY A "Appetizer">**
 Later in the document:
<SENTENCE>&A; is the first dish in a formal meal.</SENTENCE>
 It will be resolved into
<SENTENCE>Appetizer is the first dish in a formal meal. </SENTENCE>

Figure B.IV.2.3: Example of Internal Parsed Entity

Note that this is case sensitive.

Besides, XML has some Pre-defined name character entities which is described in the previous chapter. They are used in the same way as internal parsed entities.

b. External:

This kind of entities is the same as internal parsed entities, yet their content is stored in separate file, and must be well-formed (Simon St. Laurent, Robert Biggar, 1999).

For the syntax, we also have two ways of declaration, using SYSTEM and PUBLIC identifiers. Figure B.IV.2.4 gives us the summary of the format of external parsed entities. It also starts with the key word “<!ENTITY”, then the entity name, the key word “SYSTEM” or “PUBLIC”, and then the public identifier if we are using PUBLIC key word, and finally the URI to the content for the entity.

<!ENTITY *entityName* SYSTEM "*URItoContent*">
 or
<!ENTITY *entityName* PUBLIC "*PublicIdentifier*" "*URItoContent*">

Figure B.IV.2.4: Format for External Parsed Entity Declaration

External parsed entities help to break up large document into small parts, and make it easier to maintain and reuse any repetitive ones. (Simon St. Laurent, Robert Biggar, 1999)

iii. Unparsed Entities:

All unparsed entities are External entities. One important difference is that they are not processed at the same time with other parsed entities. Instead, all of them are left for the application to handle with later. Upon receiving the entity content reference and the content type (declared by using Notations), application will check if it cannot support the content type, then ignore all those entities. Otherwise, the entities will then be interpreted, retrieved and parsed separately. We need to emphasize again thanks to the use of notations and unparsed external entities that even when the application cannot understand the notation, the document parsing process will not interrupt (Document Type Definition, 2011).

Declaration syntax: Unparsed external entities are declared in the same way as that for Parsed external entities with an addition of “NDATA” and description of content type (notation identifier). Details are given in Figure B.IV.2.5.

<!ENTITY *entityName* SYSTEM "*URltoContent*" NDATA *NotationName*>
or
<!ENTITY *entityName* PUBLIC "*PublicIdentifier*" "*URltoContent*" NDATA *NotationName* >

Figure B.IV.2.5: Format for External Unparsed Entity

One classic example is the reference of file picture.gif through a picture entity shown in Figure B.IV.2.6. First the notation must be predefined in the DTD. And then we can declare the picture entity by following the syntax given in Figure B.IV.2.5. It starts with the key word “<ENTITY”, then “picture” as the entity name, then the key word “SYSTEM”, after that followed by the entity content which is an URI to the picture, and then the key word “NDATA” and finally a declared notation name representing the data type of the reference file.

<!NOTATION gif SYSTEM "image/gif">
<!ENTITY picture SYSTEM "picture.gif" NDATA gif>

Figure B.IV.2.6: Example of External Unparsed Entity and Notation

According to (Simon St. Laurent, Robert Biggar, 1999), in the XML document, the use of unparsed entities is a bit different from parsed ones. We cannot use &EntityName; in the element content but have to use entity as the content of an attribute of the element. Figure B.IV.2.7 gives a specific example on how to use the picture entity declared above in a Picture element.

<PICTURE SOURCE="picture"/>

Figure B.IV.2.7

V. XML Document Validation

1. General Overview

Briefly a DTD defines all the possible elements to be found within your document, what is the formal shape of your document tree (by defining the allowed content of an element; either text, a regular expression for the allowed list of children, or mixed content i.e. both text and children). The DTD also defines the valid attributes for all elements and the types of those attributes. And once your XML document has declared to follow a DTD, you have to follow every “grammar” rules set up by the DTD authors.

XML Document Validation is a process of checking two important properties of a XML document: Well-formness and Validity. Well-formness here addresses that the document has to follow every single syntactical rules of XML, and these rules are universal ones that apply to all XML documents. On the other hand, each XML document declared which DTD it is following at the beginning of the file, and Validity addresses that the document also needs to respect all the rules dictated in that chosen DTD (XML validation, 2011)

2. How to validate

According to (XML validation, 2011), the first way to be mentioned is to use the *xmllint* program included with **libxml** available in the Unix or Linux environment. The *--valid* option turns-on validation of the files given as input, and the *--noout* is used to disable output of the resulting tree. For example, Figure B.V.2.1 shows the validation of a copy of the first revision of the XML 1.0 specification:

```
xmllint --valid --noout test/valid/REC-xml-19980210.xml
```

Figure B.V.2.1: Example of XML document validation

Some other easier ways you may consider following is to use the available online XML Validators at www.xmlvalidation.com or <http://www.stg.brown.edu/service/xmlvalid/>. Last but not least, you can also install some XML editor such as XML Spy for a stable installed version.

VI. Complete DTD Example

Here we would like to introduce to all of you the Document Type Definition used throughout our book, namely Restaurant.dtd

Restaurant.dtd defines the DTD for the Restaurant Collection which consists of famous and popular restaurants in Singapore. Following are some of the main ideas and requirements associating with each restaurant:

- Each restaurant must have a specific name, types of service and cuisine, as well as its network of branches.
- There is also the case that some large fine-dining restaurants do not have any branches.
- For each branch of a restaurant, we are interested in its picture of venue, and also the list of dishes it provides. Additionally, as for standardization, we need to implement an index system for easier identification. Moreover, there are some more information about the address, the hotline, and the rating of a particular branch.
- For the menu, although the traditional serving is dish by dish, there exist a lot of fast-food restaurants that provide set of dishes with a lower total price to attract their customers. Hence we would like to have 2 different presentations in the menu: list of dishes, and list of pre-arranged set of dishes (by the restaurant owners). Maybe when you order the meal sets, you can get some promotion from the restaurant, a gift as a toy or some vouchers.
- A toy would need a specific name as well as text description or picture. A voucher would need some basic information about the value and expiry date. It would also need some ID numbers so that the owner can keep track well of the information.
- And finally, for each dish, customers may be interested in its name, its price, its role in a meal, and best of all, some sample description or pictures would be more attractive.

First of all, we need to identify the most needed to be child element. Clearly since each restaurant may have many branches, and even each branch has a lot of information to store, we can make branch be the child element of the restaurant. For the restaurant type, since they can be chosen from fixed enumerated list, it is more suitable to declare them as attribute.

For each branch, since all the data about address, phone number, rating are hardly expanding in terms of long time, we can set them branch's attributes, while picture and menu are more complex, they should be the child elements.

Similar reasoning can be applied to menu, set, item, toy and voucher. Any pieces of data that are more inclined to metadata (data about data) should be set as attributes, while others that are more likely to contain further information should be set as child elements. As a result, we can continue to implement the code for DTD. Figure B.VI.1.1 shows the complete version of DTD for the Restaurant Collection. Needless to say there would be more than one way to define this DTD, and each of them has its own advantages based on different needs of users.

```

<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT restaurant_collection (restaurant+)>

<!ELEMENT restaurant (rname, rtype, branch+)>
<!ELEMENT rname (#PCDATA)>
<!ELEMENT rtype EMPTY>
<!ATTLIST rtype
    type_of_restaurant      (FastFood | FamilyStyle | FineDining | CasualDining)      #IMPLIED
    type_of_cuisine         CDATA                                                    #IMPLIED
>

<!ELEMENT branch (picture*, menu)>
<!ATTLIST branch
    branch_id      ID          #REQUIRED
    address        CDATA      #REQUIRED
    city           CDATA      #FIXED      "Singapore"
    hotline        CDATA      #REQUIRED
    rating         (unrated | 1 | 2 | 3 | 4 | 5)      "unrated"
>

<!NOTATION image-gif SYSTEM "image/gif">
<!NOTATION image-jpg SYSTEM "image/jpg">
<!NOTATION image-png SYSTEM "image/png">
<!NOTATION image-svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<!ELEMENT picture ANY>
<!ATTLIST picture
    title      CDATA          #IMPLIED
    source     CDATA          #REQUIRED
    ftype      NOTATION (image-gif | image-svg | image-png | image-jpg) #REQUIRED
>

<!ELEMENT menu (item+, set*)>
<!ELEMENT set (item_set, gift?)>
<!ATTLIST set
    price      CDATA #REQUIRED
>

<!ELEMENT item_set EMPTY>
<!ATTLIST item_set
    item_id    IDREFS #REQUIRED
>

<!ELEMENT gift (toy | voucher)>
<!ELEMENT toy (#PCDATA | picture)*>
<!ATTLIST toy
    name      CDATA #REQUIRED
>

<!ELEMENT voucher EMPTY>
<!ATTLIST voucher
    id        ID          #REQUIRED
    value     CDATA      #REQUIRED
    expired   CDATA      #IMPLIED
>

<!ELEMENT item (name, type, price, description?, picture?)>
<!ATTLIST item
    item_id    ID          #REQUIRED
>
<!ELEMENT name (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT type EMPTY>
<!ATTLIST type
    course     (Appetizer | MainDish | Dessert | Drink) #REQUIRED
>

```

Figure B.VI.1.1: Complete DTD for Restaurant Collection

VIII. Conclusion

Overall, Document Type Definition is an essential part in the building of XML document. Throughout the whole chapter, we have learnt about the 4 main components of the Markups Declaration which are Element, Attribute, Notation and Entity. Covering this chapter with their basic declaration and usage syntax in various different scenarios and combinations, we hope to help you grab the core idea in constructing a DTD file as well as a wellformed and valid XML document. For interested readers, if you are keen on reading more and deeper, DTD is a good start for the newer generation – Schema.

Yet, it has been said that “For now, it's still worth learning about DTDs. They're the best tools available at present, and knowing DTDs will make transitions to schemas much easier. DTDs aren't perfect, but their replacement isn't available yet.” (Simon St. Laurent, Robert Biggar, 1999)

Chapter C: XPath

I. Introduction

XPath is short for XML Path Language. It is an integral component of XML technology and is the foundation of other related XML technologies such as XQuery and XSLT.

There are 2 main capabilities of XPath, namely, node addressing and computation on selected nodes:

- Addressing nodes: XPath provides a mechanism by which we are able to “point to” or “select” desired node(s) in an XML document. This is the primary purpose of XPath (Clark J., DeRose S., 1999). It is worth noting why this functionality is basic and essential: before performing any operation on the data contained in the XML nodes, we need a reliable way to specify which nodes we desire to work with.
- Expressing computation: XPath also provides a simple library of built-in functions for “expressing computation” on the nodes already “addressed” earlier. This allows for “extraction” or “aggregation” of data. However, the data addressed by XPath is “read-only”. XPath is not able to change the value of the nodes in the XML document directly. This particular functionality overlaps significantly with XQuery (see chapter D). As XQuery’s implementation is more mature in this regard, we leave this aspect for XQuery chapter to cover.

Some functionality in XPath requires knowledge about XML namespace. Since XML namespace is a big topic in and of itself, not all namespace-related concepts will be covered in this chapter. We will only give explanation on namespace-related concepts which are necessary for the understanding of another important concept, or just for the sake of completion. Interested reader can consult the official document on Namespaces in XML 1.0 (<http://www.w3.org/TR/xml-names/>) and XPath 1.0 (<http://www.w3.org/TR/xpath/>) for the complete coverage on XML namespace.

II. Basic concepts in XPath

1. XPath Data Types

An expression in XPath is evaluated to yield an object, which has one of the following four basic types:

- node-set (an unordered collection of nodes without duplicates)
- boolean (true or false)
- number (a floating-point number)
- string (a sequence of UCS characters)

2. XPath Data Model

XPath operates on an XML document as a tree. The data model on which an XPath expression operates is essentially a read-only version of the XML document. By keeping only the essential parts and abstracting out insignificant syntax details, we are left with a simple tree model which is more suitable for the purposes of XPath.

The way XPath models an XML document as a tree is largely similar to, but not the same as how other standards (e.g. DOM) model an XML document. In this section, we will explain and clarify some of the potentially confusing and counter-intuitive aspects of the data model of XPath.

The tree model contains nodes. There are seven types of node:

- Root nodes
- Element nodes
- Text nodes
- Attribute nodes
- Namespace nodes
- Processing instruction nodes
- Comment nodes

i. Root node

The root node is the root of the tree. There is one and only one root node in the tree. Note that the root element of the XML document is not the root node, but a child of the root node. The root node also has children processing instruction and comment nodes that occur before and after the start and end tag of the root element. The root node can be considered as an entity that represents the whole XML document, and directly or indirectly contains the rest of the nodes of the XML document.

ii. Element node

Every element in the XML document corresponds to an element node. The children of an element node are the element nodes, comment nodes, processing instruction nodes and text nodes for its content.

An element node may have a unique identifier (ID). This is the value of the attribute that is declared in the DTD as type ID. We only consider unique IDs when the document has a DTD.

iii. Attribute node

"Each element node has an associated set of attribute nodes; the element is the parent of each of these attribute nodes; however, an attribute node is not a child of its parent element" (W3C XPath 1.0). This counter-intuitive fact will be referred to again later in the chapter, so please take note of it.

"An element has attribute nodes only for attributes that were explicitly specified in the start-tag or empty-element tag of that element or that were explicitly declared in the DTD with a default value" (W3C XPath 1.0).

iv. Namespace node

"Each element has an associated set of namespace nodes [...]. The element is the parent of each of these namespace nodes; however, a namespace node is not a child of its parent element" (W3C XPath 1.0).

v. Processing instruction node

"There is a processing instruction node for every processing instruction, except for any processing instruction that occurs within the DTD" (W3C XPath 1.0).

"The XML declaration is not a processing instruction. Therefore, there is no processing instruction node corresponding to the XML declaration" (W3C XPath 1.0).

vi. Comment node

"There is a comment node for every comment, except for any comment that occurs within the DTD" (W3C XPath 1.0).

vii. Text node

"Character data is grouped into text nodes. As much character data as possible is grouped into each text node: a text node never has an immediately following or preceding sibling that is a text node. [...] A text node always has at least one character of data" (W3C XPath 1.0).

Note that characters inside comments, processing instructions and attribute values do not produce text nodes.

As we go through the definition of the seven types of nodes in XPath, these properties can be observed in the data model:

- Since the model is a tree, every node except root node has exactly one parent, and no node will be shared.
- The parent of a node, if any, is either the root node or an element node.
- Attribute nodes, namespace nodes, processing instruction nodes, comment nodes and text nodes are always leaf nodes of the tree model.

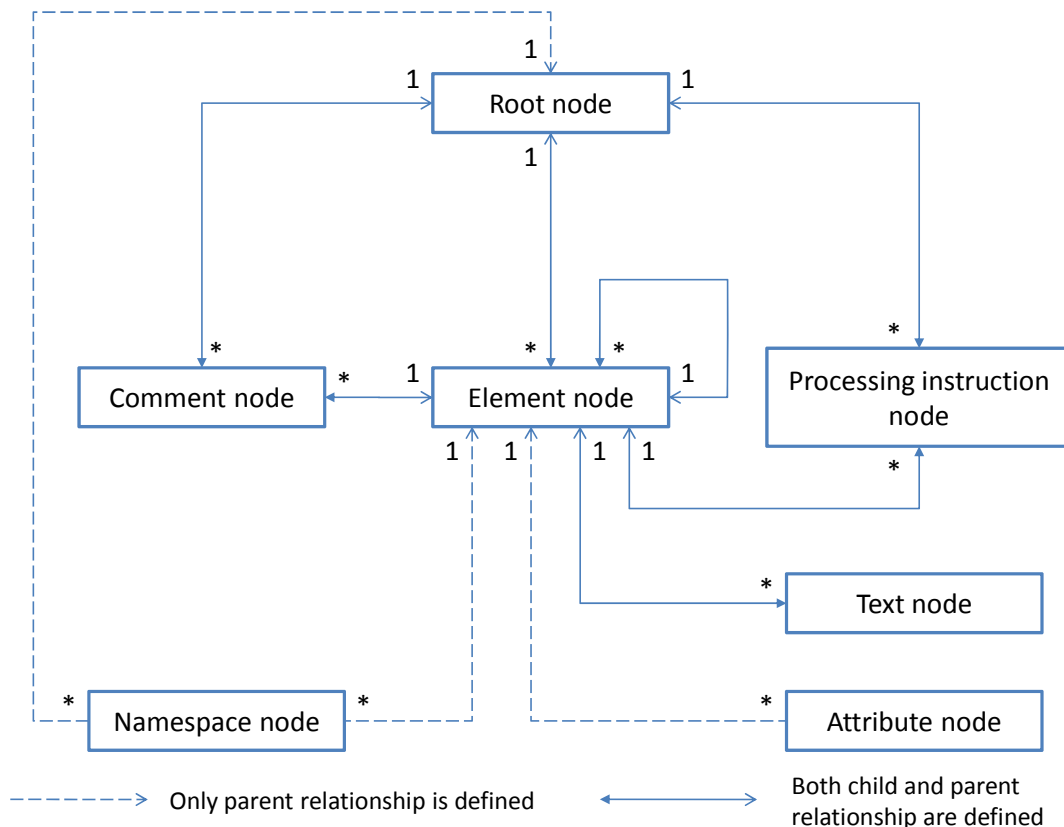


Figure C.II.2.1 Parent – Child relationship between 7 types of nodes

Figure C.II.2.1 summarizes the parent-child relationship between different types of nodes in XPath data model. A solid arrow represents “has as child” relationship, while a hollow arrow represents “has as parent” relationship.

3. Document order

"There is an ordering, document order, defined on all the nodes in the document corresponding to the order in which the first character of the XML representation of each node occurs in the XML representation of the document after expansion of general entities" (W3C XPath 1.0). In other words, document order is an order defined on all nodes in the document corresponding to their order of appearance starting from the beginning to the end of the document.

By the definition, these properties can be observed in the document order:

- Root node is the first node.
- Element nodes come in order of the occurrence of their start tag in the XML document.
- Among the nodes with the same parent node, namespace nodes occur before attribute nodes, and attribute nodes occur before element nodes.

However, W3C leaves up to implementation the relative ordering among the attribute nodes, and the relative ordering among the namespace nodes.

In XMLSpy, the attribute nodes of an element are ordered according to the order of declaration.

4. Context

The context consists of:

- *a node - the context node*

As we will see later in the chapter, during the evaluation process of a path expression, axis and node-test will generate an initial node-set, and then each predicate will further filter the node-set. During each of the aforementioned processes, each and every node in the resulting node-set from previous step will become the context node for the current step of evaluation.

- *a pair of (non-zero) positive integers - the context position and the context size*

The context size indicates the size of the node-set, i.e. number of nodes in the node-set, that the context node belongs to. Context size can be queried with function *number last()*.

The context position is the position of the context node in the node-set ordered in document order if the axis is a forward axis and ordered in reverse document order if the axis is a reverse axis. In other words, the context position reflects how close the context node is to the previous context node which generates the current node-set. The context position starts counting from 1. Context position can be queried with function *number position()*.

- *a set of variable bindings*
- *a function library*
- *the set of namespace declarations in scope for the expression*

In this book, we are not going to go into detail about these last 3 items.

III. Location path

1. Location Path

Location path refers to the full XPath expression that achieves the first functionality of XPath. It is a means of addressing the nodes in the tree model that corresponds to an XML document. As such, a location path will evaluate to a node-set, an unordered collection of nodes without duplicate, which is, specifically, a subset of all the nodes in the tree model. “Unordered” and “without duplicate” follow naturally from set theory, in which elements are not distinguished by their positions in the set, and each element in the set is unique.

LocationPath	::=	RelativeLocationPath AbsoluteLocationPath
AbsoluteLocationPath	::=	 '/' RelativeLocationPath ? AbbreviatedAbsoluteLocationPath
RelativeLocationPath	::=	Step RelativeLocationPath '/' Step AbbreviatedRelativeLocationPath

Figure C.III.1.1: Grammar of Location Path (W3C XPath 1.0)

Figure C.III.1.1 describes the grammar of a location path. Putting aside the abbreviated syntax, which will be covered in detail in section C.III.7, a location path consists of one or more location steps ([Step](#)) separated by a “/”, and the location path may or may not be preceded by a “/”.

/
Step
/ Step
Step / Step / Step
/ Step / Step / Step

Figure C.III.1.2: Generalized examples of Location Path

Figure C.III.1.2 shows 5 generalized examples that fit the grammar of Location Path. As we can see, a single “/” can already be qualified as a location path. A location path may have one or three, or any positive number of location steps ([Step](#)) separated by “/”, and the location path may or may not be preceded by a “/”.

It can also be observed from figure C.III.1.1 that there are 2 types of location paths: absolute location paths and relative location paths.

i. Absolute Location Paths

If the location path starts with a “/”, it is an absolute location path.

An absolute location path is evaluated starting from the root node. In other words, the initial context node is the root node.

ii. Relative Location Paths

On the contrary, if the location path does not start with a “/”, it is a relative location path.

A relative location path can only be evaluated if provided with a context. Without a context, a relative location path is meaningless. The context is usually naturally provided by the previous step of evaluation (figure C.III.1.1 shows that a relative location path can be part of an absolute location path).

As we explore deeper and deeper, we will realize that location path is quite powerful by itself. However, a single location path does not always do the trick. Sometimes, we need to combine the result of two location paths together. Therefore, let us take a look at how we can combine the results of two location paths together.

UnionExpr	::=	PathExpr UnionExpr ' ' PathExpr
-----------	-----	--

Figure C.III.1.3 Grammar of UnionExpr

As we can see in figure C.III.1.3, this is accomplished using the union operator, a vertical-bar character (|), to delimit the constituent paths (Simpson, J. E., 2008).

/child::restaurant_collection/child::restaurant/child::branch/child::picture /child::restaurant_collection/child::restaurant/child::branch/child::menu/child::item/child::picture
--

Figure C.III.1.4 Example of union operator

An example of the usage of the union operator is shown in figure C.III.1.4. As to the meaning of the path expression, we are going to see this expression again in section C.III.4 Axes under Descendant axis.

2. Location Step

To understand how the full location path works, we need to first understand the constituent component: the location steps.

Step	::=	AxisSpecifier NodeTest Predicate *
		AbbreviatedStep
AxisSpecifier	::=	AxisName '::'
		AbbreviatedAxisSpecifier

Figure C.III.2.1: Grammar of Location Step (Step)

The unabbreviated syntax of a location step consists of 3 parts, as shown in Figure C.III.2.1. Each part selects/filters the set of nodes according to various criteria:

- An **axis**, which specifies the tree relationship with the context node (e.g. child nodes).
- A **node test**, which specifies the node type and name of the nodes (e.g. comment nodes).
- Zero or more **predicates**, which further filter the set of nodes according to arbitrary selection criteria.

child::item[name='Coke']

Figure C.III.2.2: An example of a location step

Let us look at the location step shown in figure C.III.2.2: “child” is the axis, “item” is the node test (in this case, we test the node by name), and “name=‘Coke’” is the predicate.

3. Evaluation Process

i. How a Location Step is evaluated

A location step is always evaluated relative to a context node. Context node refers to the basis on which the axis is applied.

Consider the following location step which is part of some location path.

descendant::item

What does this text segment mean? The cryptic syntax can easily be deciphered using a mnemonic, which “magically” translates it to plain English.

Magical Mnemonic:

<u>SELECT</u> <i>axis</i> <u>OF</u> <i>context node(s)</i> <u>WHOSE</u> (<i>type</i>) <u>IS</u> <i>nodeTest</i>

The text in brackets, “(type)”, indicates the specific node test type being used. In this case, it’s a “name” node test.

Performing the appropriate substitutions gives us,

<u>SELECT</u> <i>descendant(s)</i> <u>OF</u> <i>context node(s)</i> <u>WHOSE</u> (<i>name</i>) <u>IS</u> “item”.
--

This will select all the descendant nodes of each of the context node(s) having name “item”.

Even after the mnemonic substitution, we still do not know the “context node”. Without the context, we cannot specify exactly which nodes are being selected in the **restaurant.xml** document. To determine what the context node(s) is, we need to know what comes before the location step.

ii. How a Location Path is evaluated

A location path is evaluated in a series of steps. All the nodes in the resulting node-set become the new context nodes for the next step of evaluation. This process can be simply referred to as “cascading evaluation”.

In this section only, for the sake of simplicity, we will use the tree model in figure C.III.3.1 instead of restaurant.xml to demonstrate the evaluation process of a location path.

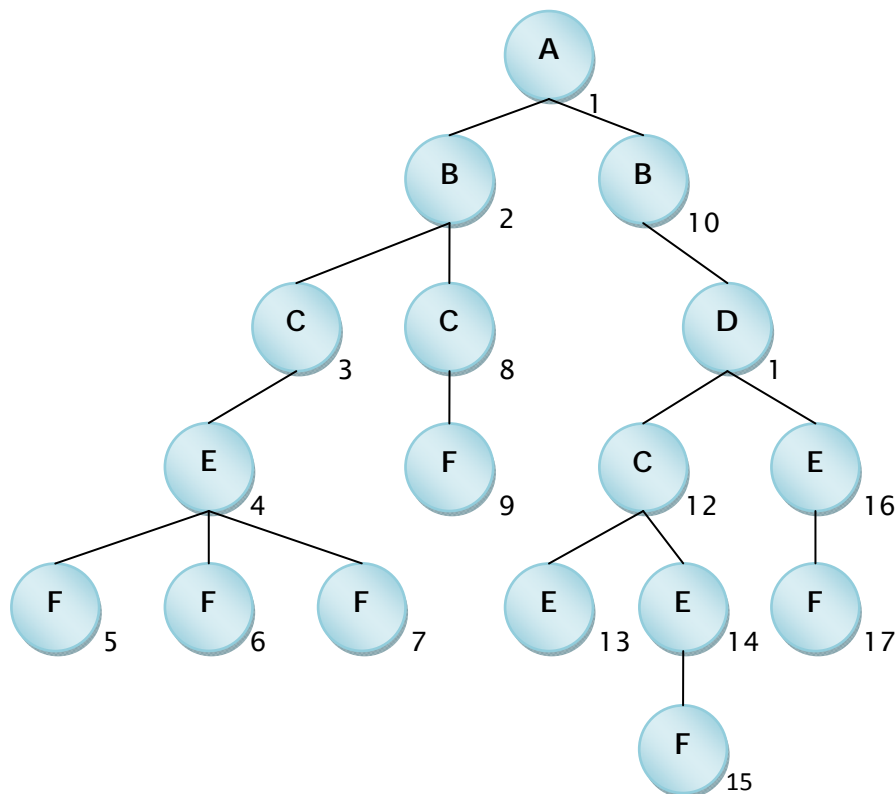


Figure C.III.3.1: The tree model for the example in this section (Schwartzbach, A. M. (2006))

Now, let us extend the “magical” mnemonic we used earlier to evaluate more complicated location paths involving multiple location steps.

Let us consider the following 3-step location path:

/descendant::C/child::E/child::F

Step 0

Firstly, don't panic!

Note that this is an absolute location path because it starts with a “/”. This means that the context node is the root node.

Recall that the root node in an XML document does not refer to node “A1”. It is the node at the very top of the XML tree hierarchy, one level above “A1”. As such, the root node cannot be found physically within the XML document. The function of the root node is to hold all the other nodes that is specified by the user in an XML document so as to conform to the tree structure containing a single root at the top.

The location path can be broken down into 3 separate location steps, the “/” being the delimiter:

descendant::C
child::E
child::F

Step 1

Consider the first location step:

descendant::C

Using our mnemonic, we have:

SELECT descendant(s) OF "A1" WHOSE (name) IS "C".

Referring to the figure, we have the 1st node-set generated: C3, C8, C12

Step 2

We now proceed to evaluate the second location step.

child::E

Using the same mnemonic, we have:

SELECT child(s) OF context nodes WHOSE (name) IS "E"
= SELECT child(s) OF 1st node-set WHOSE (name) IS "E"

Note that the context node of the current location step is no longer the "root node". It consists of each of the nodes generated from the previous sequence evaluation; the 1st node-set.

Since the 1st node-set consists of 3 nodes, we can expand the mnemonic further.

<u>SELECT child(s) OF "C3" WHOSE (name) IS "E".</u>	→	"E4"
<u>SELECT child(s) OF "C8" WHOSE (name) IS "E".</u>	→	∅
<u>SELECT child(s) OF "C12" WHOSE (name) IS "E".</u>	→	"E13" & "E14"

The child axis is thus, applied to each of the nodes in the context node-set separately, resulting in the nodes listed on the right. Thereafter, the result is combined to make up the subsequent resulting node-set: E4, E13, E14.

Step 3

We now reach the third or last location step.

child::F

Similarly, the last location step consists of a different set of context nodes.

SELECT child(s) OF 2nd node-set WHOSE (name) IS "F".

This can be expanded to:

<u>SELECT child(s) OF "E4" WHOSE (name) IS "F".</u>	→	"F5", "F6", "F7"
<u>SELECT child(s) OF "E13" WHOSE (name) IS "F".</u>	→	∅
<u>SELECT child(s) OF "E14" WHOSE (name) IS "F".</u>	→	"F15"

Final node-set generated: F5, F6, F7, F15.

Summary

`/descendant::C/child::E/child::F`

In summary, the above location path is equivalent to:

`child of (child of (descendant of "root node" named "C") named "E") named "F"`

The context nodes are placed in the inner brackets. Hence, as in precedence arithmetic, we are to evaluate the innermost brackets first.

4. Axes

The evaluation of an XPath location step starts with its axis. Axis specifies the tree relationship between the node(s) to be selected and the context node.

Except for self axis (the axis that contains only the context node), an axis is either a forward axis or a reverse axis. A forward axis is an axis that only contains the context nodes or nodes that are after the context nodes in document order. A reverse axis is an axis that only contains the context nodes or nodes that are before the context nodes in document order. By these definitions, one cannot specify nodes that precede the context node and nodes that succeed the context node in the same location step. To achieve that, we will use union operator `|` as introduces in section C.III.1.

AxisName	::=	'ancestor'
		'ancestor-or-self'
		'attribute'
		'child'
		'descendant'
		'descendant-or-self'
		'following'
		'following-sibling'
		'namespace'
		'parent'
		'preceding'
		'preceding-sibling'
		'self'

Figure C.III.4.1 Grammar of AxisName

Figure C.III.4.1 lists out the name of the 13 axis in XPath. All the axes will be explained, except for namespace axis.

i. Child axis

This axis contains the children of the context node. As explained in section C.II.2 XPath Data Model, since there is no children relationship between an attribute and the corresponding element, the child axis will not contain attribute nodes of the context node.

Child axis is a forward axis.

`/child::restaurant_collection/child::restaurant/child::branch`

Figure C.III.4.2 Example of child axis

An example of child axis is demonstrated in Figure C.III.4.2. Starting from root, we select the child named `restaurant_collection` of the root node, then we go on to select all children named `restaurant` of `restaurant_collection` node, and finally, we select all children named `branch` of all the resulting `restaurant` nodes. The meaning of the query is "find all the branches of all the restaurants in the restaurant collection".

ii. Descendant axis

This axis contains the descendants of the context node. As explained in section C.II.2 XPath Data Model, since an attribute node is not a child of its parent element, the descendant axis will not contain attribute nodes of the context node. Namespace nodes are also not included in descendant axis of the context node for the same reason.

Descendant axis is a forward axis.

One of the many usages of descendant axis is to directly select all element nodes which have the same name and are descendants of different levels with respect to the context node, saving us the trouble of going down the document tree level by level and merging separate results together.

/descendant::branch

Figure C.III.4.3 Example of descendant axis

An example of descendant axis is demonstrated in figure C.III.4.3. The expression selects all the descendants named `branch` starting from root node. Thanks to the way the data is organized in this restaurant example (`branch` elements only appear as children of `restaurant` elements and `restaurant` elements only appear as children of `restaurant_collection` element), we end up with the same result as traversing the tree structure level by level with child axis (figure C.III.4.2).

Let us look at another example of descendant axis.

/descendant::picture
/child::restaurant_collection/child::restaurant/child::branch/child::picture
/child::restaurant_collection/child::restaurant/child::branch/child::menu/child::item/child::picture
/descendant::branch/child::picture
/descendant::item/child::picture
/child::restaurant_collection/child::restaurant/child::branch/child::picture /child::restaurant_collection/child::restaurant/child::branch/child::menu/child::item/child::picture

Figure C.III.4.4 Descendant axis vs. Child axis

The example in Figure C.III.4.4 demonstrates the usage of descendant axis to shorten the path expression, and also shows the pitfall that one may encounter when trying to shorten the path expression.

Before discussing the example, let us review the DTD document of the restaurant example. We realize that: 1) a `branch` element can only be a child of a `restaurant` element, and a `restaurant` element can only be a child of `restaurant_collection` element; 2) a `picture` element can either be a

child of an item element or a branch element; 3) an item element can only be a child of a menu element, and a menu element can only be a child of a branch element. The example will revolve around these 3 key points.

The first path expression selects all descendants named picture from root node. Without considering the DTD, the meaning of the query is "find all picture records in the collection of restaurants". However, if we take the 3 key points above into consideration, the meaning of the query would be "find any picture records of restaurant branches and menu items". There are 44 element nodes in the result of this query.

The second path expression selects all child nodes named picture of all the nodes named branch by traversing the document tree level by level with child axis. Taking the DTD into consideration, this query means "find any pictures records of all the branches". There are 13 element nodes in the result of this query.

The third path expression selects all child nodes named picture of all the nodes named item by traversing the document tree level by level with child axis. Taking the DTD into consideration, this query means "find any pictures records of all the items in the menus of all the branches". There are 31 element nodes in the result of this query.

If the intention of the query is the same as the second or third expression, we should not blindly use descendant axis to shorten the expression to the first expression and directly jumps to the result. Instead, we must consult the DTD to find out whether the meaning of the shortcut expression is the same as the original one with child axis. In this case, the second and third expression can be shortened up as the fourth and fifth expression respectively without changing the intention of the query. The converse also holds true. If the intention of the query is the same as the first expression, we should look at the DTD to see whether picture element can be the child element of any element other than branch and item. If there does exist other element which is parent of picture element, then the sixth expression with union operator to merge the node-set containing pictures of branches and the node-set containing pictures of items is the safe option to go for.

iii. Self axis

This axis contains only the context node.

In practice, self axis is usually seen in the expressions inside predicates in order to connect the context from the outer expressions with the inner expression. One example of such use would be the self axis' combination with descendant-or-self axis in their abbreviated forms (see section C.III.7 Abbreviated Syntax).

`/child::restaurant_collection/child::restaurant/branch::restaurant/self::restaurant`

Figure C.III.4.5 Example of self axis

Figure C.III.4.5 shows an example of self axis. This is not a very good example of self axis since the location step containing the self axis is totally redundant. The result of this query will be the same with or without the location step containing the self axis.

iv. Descendant-or-self axis

This axis contains the context node and the descendants of the context node. In other words, the content of this axis is union of the content of descendant and self axis of the same context node.

Descendant-or-self axis is a forward axis.

The application of descendant-or-self axis is similar to descendant axis. The only difference is that the context node is also included in this axis.

v. Attribute axis

This axis contains the attributes of the context node; the axis will be empty unless the context node is an element.

Attribute axis is the only way in XPath to access the attributes of an element.

```
/child::restaurant_collection/child::restaurant/child::branch/attribute::address
```

Figure C.III.4.6 Example of attribute axis

An example of attribute axis is shown in figure C.III.4.6. The first 3 location steps of this example are basically the example used in child axis (figure C.III.4.2). From the node-set returned by the first 3 location steps, we continue to select attributes named address. The meaning of this query would be "find the addresses of all the branches of all the restaurants there are in the restaurant collection".

```
/child::restaurant_collection/child::restaurant/child::branch/child::address
```

Figure C.III.4.7 Attribute is not child of the associated element

As mentioned in section C.II.2 XPath Data Model, an attribute node is not a child of its parent element. The example in figure C.III.4.7 tries to query the child named address of the element nodes named branch; the query will return an empty node-set.

vi. Parent axis

This axis contains the parent of the context node, if there is one. Specifically, this axis contains nothing if the context node is the root node, and it contains the single parent node of the context node for the rest of the cases.

Parent axis is a reverse axis.

```
/descendant::picture/parent::item
```

Figure C.III.4.8 Example of parent axis

An example of parent axis is shown in figure C.III.4.8. From the resulting node-set returned by the first location step, we continue to select parent nodes named item. The meaning of the query after taking DTD into consideration is "find the list of items from all the menus of all the branches of restaurants that have a picture".

vii. Ancestor axis

This axis contains the ancestors of the context node. The root node is always included in the ancestor axis, unless the context node is the root node.

Ancestor axis is a reverse axis.

Similar to descendant axis, ancestor axis let us directly select all element nodes which are ancestors of many levels with respect to the context node, saving us the trouble of going up the document tree level by level.

`/descendant::toy/ancestor::set/attribute::price`

Figure C.III.4.4.9 Example of ancestor axis

The example in figure C.III.4.9 demonstrates the usage of ancestor axis. Starting from root, the expression selects all the descendants named toy, then we go on to select the ancestor named set, and finally we go on to select the attribute named price. The meaning of this query, taking the DTD into consideration, would be “find the price of all the meal set that comes with toy”.

viii. Ancestor-or-self axis

This axis contains the context node and the ancestors of the context node. In other words, the content of this axis is union of the content of ancestor and self axis of the same context node.

Ancestor-or-self axis is a reverse axis.

The usage of this axis is similar to ancestor axis. We only have to keep in mind that the context node is also included for ancestor-or-self axis.

ix. Preceding axis

This axis "contains all nodes in the same document as the context node that are before the context node in document order, excluding any ancestors, attribute nodes and namespace nodes" (W3C XPath 1.0).

The less precise but more visual definition would be: all elements nodes whose closing tags (or empty element tags) appear before the opening tag of the context node in the XML document are in the context node's preceding axis.

Preceding axis is a reverse axis.

The examples for the last 4 axes will revolve around this single node selected by this expression the path expression in figure C.III.4.10:

`/descendant::menu[position()=2]/item[position()=3]`

Figure C.III.4.10 Common path expression for the last 4 axes

The meaning of the expression in figure C.III.1.10 is “selects the 3rd item in the 2nd menu in the collection of restaurant”. As to why the expression has the above meaning, we leave it as an exercise for the reader after reading section C.III.6 Predicate.

```
/descendant::menu[position()=2]/item[position()=3]/preceding::description
```

Figure C.III.4.11 Example of preceding axis

Figure C.III.4.11 gives an example for preceding axis. The path expression selects nodes preceding the 3rd item element in the 2nd menu element named description. Taking the DTD into consideration, this path expression queries the descriptions of the items preceding the 3rd item in the 2nd menu. The result contains the descriptions of items in different menus from the menu that the context node belongs to.

x. Following axis

This axis "contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants and excluding attribute nodes and namespace nodes" (W3C XPath 1.0).

The less precise but more visual definition would be: all elements nodes whose opening tags (or empty element tags) appear after the closing tag of the context node in the XML document are in the context node's following axis.

Following axis is a forward axis.

```
/descendant::menu[position()=2]/item[position()=3]/following::description
```

Figure C.III.4.12 Example of following axis

Figure C.III.4.12 gives an example of preceding axis. The path expression selects nodes following the 3rd item element in the 2nd menu element named description. Taking the DTD into consideration, this path expression queries the descriptions of the items following the 3rd item in the 2nd menu. The result contains the descriptions of items in different menus from the menu that the context node belongs to.

xi. Preceding-sibling axis

This axis "contains all the preceding siblings of the context node; if the context node is an attribute node or namespace node, the preceding-sibling axis is empty" (W3C XPath 1.0).

Preceding-sibling axis is a reverse axis.

```
/descendant::menu[position()=2]/item[position()=3]/preceding-sibling::item
```

Figure C.III.4.13 Example of preceding-sibling axis

Figure C.III.4.13 gives an example of preceding-sibling axis. The path expression selects preceding siblings named item of the 3rd item element in the 2nd menu element. The meaning of the path expression is "find all the items in the 2nd menu that strictly come before the 3rd item".

Preceding-sibling (and following-sibling) axes are usually used when the XML document contains a series of elements with same name which are children of the same parent element.

xii. Following-sibling axis

This axis "contains all the following siblings of the context node; if the context node is an attribute node or namespace node, the following-sibling axis is empty" (W3C XPath 1.0).

Following-sibling axis is a forward axis.

```
/descendant::menu[position()=2]/item[position()=3]/following-sibling::item
```

Figure C.III.4.14 Example of following-sibling axis

Figure C.III.4.14 gives an example of following-sibling axis. The path expression selects following siblings named item of the 3rd item element in the 2nd menu element. The meaning of the path expression is "find all the items in the 2nd menu that strictly come after the 3rd item".

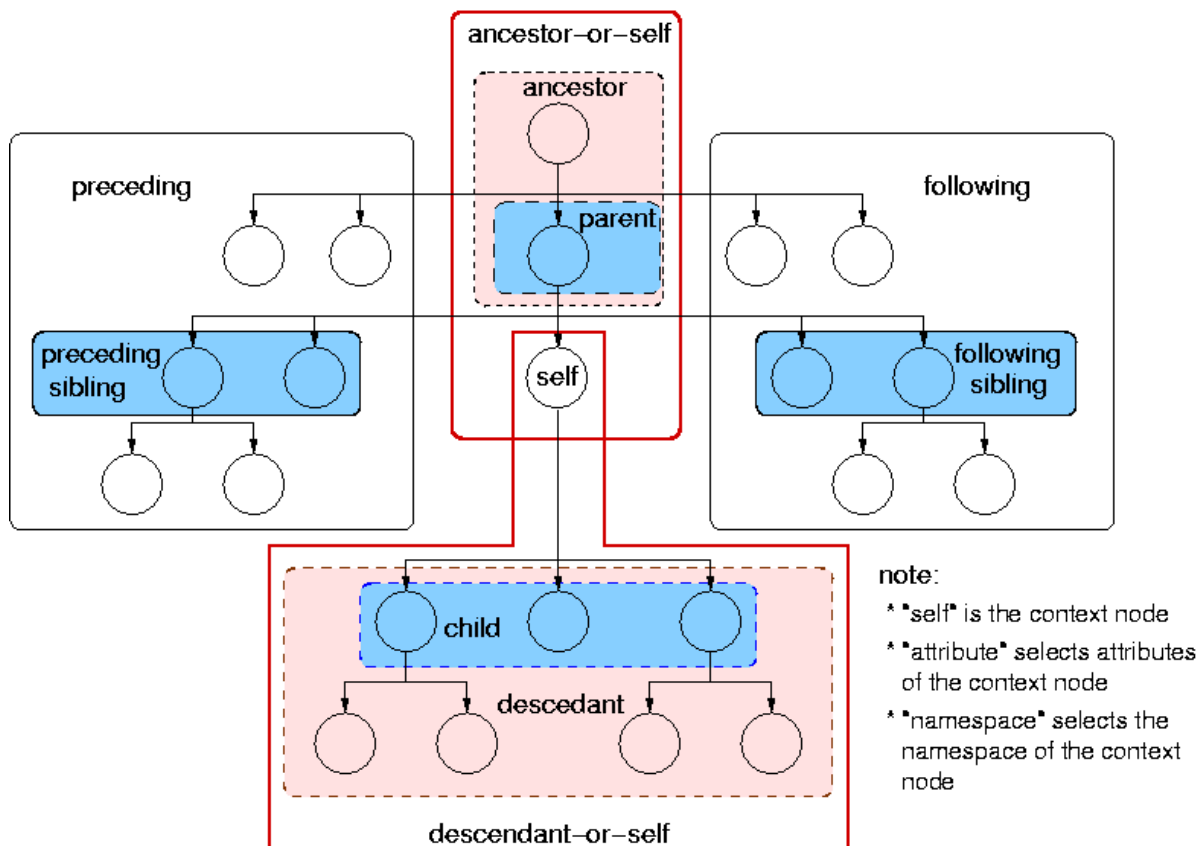


Figure C.III.4.15 Summary of the axes in XPath and their coverage
(2010, Scott, P., Martin, M.)

Figure C.III.4.15 gives a visual summary of the scope of the axes. This diagram sums up the fundamental knowledge of the axes in XPath.

5. Node Tests

The second part of a location step is the node test. The primary function of the node test is to further filter the set of nodes selected by the axis. This is done by specifying the type of nodes desired. For example, specifying either text or comment nodes will only select nodes of that type.

NodeTest	::=	NameTest
		NodeType '(' '
		'processing-instruction' '(' Literal ')'

Figure C.III.5.1 Grammar of NodeTest

Figure C.III.5.1 shows that there are 3 types of node tests in total. The first 2 lines above refer to the main types of node tests, namely, “name test” and “type test” respectively. The last one is in fact, a special case. It is a mixture of first 2 types and will hence, be dealt with separately.

i. Type Test

The “type test” tests for nodes matching a given type. If the test is met, the node is included in the successive node-set, as per the evaluation scheme detailed in the location path section above.

NodeType	::=	'comment'
		'text'
		'processing-instruction'
		'node'

Figure C.III.5.2 Grammar of Node Type

Figure C.III.5.2 shows 4 options for type test. However, we are going to cover only 3 type test in this section; the type test processing-instruction will be in a different section of its own.

- **node()**

The simplest type test is “node()” which evaluates to true for any node of any type whatsoever.

For example:

/descendant-or-self::node()

In this case, “node()” is the node test.

Use our mnemonic:

<u>SELECT</u> <u>axis</u> <u>OF</u> <u>context</u> <u>node(s)</u> <u>WHOSE (type)</u> <u>IS</u> <u>nodeTest</u>

We have:

<u>SELECT</u> <u>descendant or self</u> <u>OF</u> <u>root node</u> <u>WHOSE (type)</u> <u>IS</u> <u>any node</u>
--

Notice that the words in brackets still remain as “type”. By default, this refers intuitively to the second type of node test which is the “type test”. The “nodeTest” section of the mnemonic is replaced according to the node type being used at the moment.

- **comment()**

This node test evaluates to true for comment nodes only.

For example:

/descendant-or-self::node()/child::comment()
--

Using our mnemonic, we have:

```
SELECT child(s) OF /descendant-or-self::node() WHOSE (type) IS comment
```

Building upon the first example, we first select all nodes in the first location step. After evaluating the first location step, we have all the nodes in the document. Thereafter, using this as the context, we evaluate the second location step. Applying the “comment()” node test in the second location step, means that we're testing all of the nodes selected previously. Only comment nodes will pass the test. Hence, the XPath expression simply selects all comment nodes in the document.

- **text()**

This node test evaluates to true for text nodes only.

For example:

```
/descendant-or-self::node()/text()
```

Using our mnemonic, we have,

```
SELECT /descendant-or-self::node() OF root node WHOSE (type) IS text
```

ii. Name Test

A “name test” tests for nodes matching a given name. If a node passes the name test, it is included in the successive node-set. Note that the name test only returns true when the name is exactly the same as specified.

- **QName**

The simplest name test is the “qualified name”. The term “qualified” has to do with namespaces; a topic we will not delve into in this document. Suffice to say, it can be taken simply as any name that is legally allowed to identify an XML element as defined in the XML section of this document.

This form of node test is frequently used earlier while introducing location path evaluation as it is the most intuitive.

For example:

```
/descendant::picture
```

Here, “picture” is the node test.

Using our mnemonic:

```
SELECT axis OF context node(s) WHOSE (type) IS nodeTest
```

We have:

```
SELECT descendant OF root node WHOSE (name) IS “picture”
```

The type specified in the bracket is replaced with “name” to indicate that the node test is a “name test”.

In the case of a name test, the concept of “principal node type” is relevant. The “principal node type” of an axis refers to the type of elements the axis can contain. By default, for all axes, the principal node type is element node. However, for the attribute axis, the principal node type is attribute node; for the namespace axis, the principal node type is namespace node. The name test will only select nodes that correspond to the “principal node type” of the current axis.

For example:

```
/descendant-or-self::node()/attribute::node()
```

Using our mnemonic, we have,

```
SELECT attribute OF /descendant-or-self::node()/ WHOSE (type) IS any
```

Although the “node()” node test refers to all nodes of any type, not all nodes are selected in the final second location step. This is because the attribute axis's principal node type is the attribute. Hence, only attribute nodes are allowed to be included in the resulting node-set.

The node test checks each node in the current evaluation step against a certain criteria, generally characterised by type. Only those nodes matching the criteria are included in the new node-set. The rest are disregarded or ignored.

- **Asterisk**

The “*” node test is true for any node of the principal node type.

For example:

```
/descendant-or-self::node()/child::*
```

Using our mnemonic, we have:

```
SELECT child OF /descendant-or-self::node()/ WHOSE (type) IS principal-node
```

Hence, this selects all the elements in the document since the principal node of the child axis is the element.

At this point it is instructive to contrast this with the following:

```
/descendant-or-self::node()/child::node()
```

The asterisk node test is simply replaced with the node() node test.

Using our mnemonic, we have:

```
SELECT child OF /descendant-or-self::node()/ WHOSE (type) IS any node
```

Note that in this case all nodes including comments and processing-instructions will be selected, not only elements.

iii. Processing-instruction

```
processing-instruction()
```

This node test evaluates to true only for processing-instruction nodes. This last node test is unique. It's the only one that can accept an optional parameter.

For example:

```
/descendant-or-self::node()/processing-instruction()
```

Using our mnemonic, we have:

```
SELECT descendant or self OF root node WHOSE (type) IS processing-instruction
```

This selects all the processing-instructions in the whole document.

To only select specific processing-instruction by name, we can put the name in the brackets as arguments.

```
/descendant-or-self::node()/processing-instruction('image-gif')
```

This specifically selects all the processing instructions titled “image-gif”. In our restaurant.xml document, it refers to the following element.

```
<?image-gif scale="50%"?>
```

Note that the “processing-instruction” node test in this form is performing a name test in addition to a type test. The type test refers to filtering for processing-instruction nodes. The name test refers to filtering for the processing-instruction nodes of a specific title as listed in the argument.

- **Closing remarks:** Note that the “()” in each the node tests above are necessary. The presence of parentheses serves to clearly distinguish the type test from name test.

6. Predicates

The final part of a location step consists of zero or more predicates enclosed by square brackets.

Predicate	::=	[' PredicateExpr ']
PredicateExpr	::=	Expr

Figure C.III.6.1 Grammar of Predicate and PredicateExpr

As defined in figure C.III.6.1, whatever is in the brackets is termed a “predicate expression” ([PredicateExpr](#)), which is essentially the general “expression” ([Expr](#)) in a predicate bracket construction. As such, the predicate expression can be of 2 types: a location path on its own or a boolean statement.

i. Predicates of type “Boolean Statement”

First, let us consider the case where the PredicateExpr is a boolean statement.

Syntax:

```
[LocationPath Operator Value]
```

Syntax of Operator:

```
"=" | "!=" | ">=" | "<" | "<="
```

The symbols above are the familiar boolean mathematical operators like “equal to” and “greater than”. Note that if the XPath expression is in an XML document, the operators need to be replaced by its “escape” equivalents as defined in the XML section. For instance, character “<” will need to be escaped as “<”.

Syntax of Value: Value can either be a string literal, a number (floating point or decimal). The grammar for Literal and Number are given in figure C.III.6.2.

Literal	::=	"" [^"]* "" "" ['']* ""
Number	::=	Digits ('.' Digits)? '.' Digits
Digits	::=	[0-9]+

Figure C.III.6.2 Grammar of Literal, Number and Digits

A typical example of a predicate is:

[attribute::hotline="66109666"]

A predicate filters a node-set with respect to the axis and the node test (context), to produce a new node-set. For each node in the node-set to be filtered, the PredicateExpr is evaluated with that node as the context node; if the expression evaluates to true for that node, the node is included in the new node-set; otherwise, it is not included. Notice that the position of the context nodes being selected does not change, as in the case of adding another location step through the "/" operator. In predicate evaluation, the set of all nodes at the current context is merely being "filtered".

In summary, a location step's predicate sets forth a boolean test: if the boolean test returns true, then this node will be selected for inclusion in the resulting node-set. In all other cases, the node is disregarded.

Here, it is useful to extend the mnemonic we have been using to predicates.

Magical mnemonic 1:

SELECT axis OF context node(s) WHOSE (type) IS nodeTest

Magical mnemonic 2a:

SELECT axis OF context node(s) WHOSE (type) IS nodeTest
PREDICATED UPON [Proposition]

Comparing the two, we observe that the second form is essentially equivalent to the first except for the addition of the "PREDICATED UPON" expression. What does this mean? As explained in the summary above, the predicate is essentially a proposition which has a truth value; either true or false (boolean). This proposition is used as a condition for the filtering process.

From the location path evaluated up till "mnemonic 1", we have a node-set as illustrated in the section above, where we evaluated a full location path. For every node in this node-set, we further test it against the proposition. Only if the proposition yields a true will that node be included in the new resulting node-set.

Magical Mnemonic 2a (Predicate Section):

[Proposition]
= [LocationPath Operator Value]
= {SELECT axis OF context node(s) WHOSE (type) IS nodeTest} Operator Value

Consider the following location path with a boolean statement as predicate:

```
/restaurant_collection/restaurant/branch[attribute::hotline="66109666"]
```

Use our 2nd mnemonic, we have:

```
SELECT child(s) OF "/restaurant_collection/restaurant/" WHOSE (name) IS "branch"  
PREDICATED UPON  
{SELECT attribute(s) OF "/restaurant_collection/restaurant/" WHOSE (name) IS  
"hotline"} = "66109666"
```

Note that the context of the predicate follows the same context established previously by the portions of the location steps that precede the predicate.

To illustrate the difference between a location step separated by "/" and predicates, consider what happens if we replaced the same predicate expression above with a "/" instead.

```
/restaurant_collection/restaurant/branch/attribute::hotline
```

Applying our mnemonic again:

```
SELECT attribute(s) OF "/restaurant_collection/restaurant/branch" WHOSE (name)  
IS "hotline"
```

Note now that the context of the location path has now changed, it has gone down into the "hotline" attribute branch as opposed to staying at "branch". Now, different (context) nodes are selected upon evaluation. The difference is clearly illustrated by comparing the pink highlighted text in the 2 expressions concerned.

At this point, the usefulness of predicates should be clear. Predicates allow us to test surrounding nodes without actually moving there.

❖ *Positional Predicates*

Positional predicates are a commonly example of "boolean statements" predicates. Because of its widespread use, we would be touching on this even though it involves the use of node-set function.

The position() function returns the position of the node in the node-set according to the natural order appearing in the XML document.

For example, in **restaurant.xml**, the "restaurant" branch contains 2 child "branch" nodes.

```
<restaurant>  
  <branch branch_id="branch01" ... hotline="66109666"> ①  
  <branch branch_id="branch02" ... hotline="66106999"> ②  
  ...  
</restaurant>
```

The position of each of "branch" nodes is indicated right beside the code. This numbering is derived from the document order as defined in section C.II.3 Document Order.

Before giving an example XPath expression involving positional predicates, let us consider the following first. Notice that the predicate appears in the middle of the XPath expression is followed by another location step.

```
/restaurant_collection/restaurant[rname="Kent Ridge Highlights"]/branch
```

The first part of the path expression is evaluated as illustrated previously.

```
/restaurant_collection/restaurant[rname="Kent Ridge Highlights"]
```

Only the restaurant having rname="Kent Ridge..." is selected. The resulting node-set contains only one node.

Thereafter, for each node in the node-set selected above, a further shift in context is applied to step down into the branch node(s) of each of the nodes. In this case, since there's only one node, the procedure applies only to that one node.

The final resulting node-set hence, contains the 2 "branch" nodes of the restaurant named "Kent Ridge Highlights". The result is equivalent to the XML code segment quoted above.

Now, what if we want to select only one of the "branch" nodes? Suppose the 2nd one?

For this, we need to use the position() function.

```
/restaurant_collection/restaurant[rname="Kent Ridge Highlights"]/branch[position()=2]
```

By adding the positional predicate, we further apply a test on the node-set obtained from the earlier expression which contains only 2 "branch" nodes. Recall that the position() returns the position of the node as indicated clearly in the XML code segment above. Hence, it is clear that the boolean test will lead to a "true" only if the position returns a "2". This only occurs for the 2nd branch node.

Therefore, only the 2nd branch node will be included in the final resulting node-set.

ii. Predicates of type Location Path

The second type of predicates involves a single location path.

Syntax:

```
[LocationPath]
```

Though simpler in form relative to the first type, this form is fact, less intuitive. It is equivalent to the first without the operator or the value.

When predicates are used in this way, it "tests" for the existence of a node. If the node(s) specified in the predicate exists relative to the context, the test is deemed to have passed. Hence, the context node is included in the new node-set. Otherwise, it is excluded.

Why does this work? This works because of the way XPath treats an empty node-set. If the node specified in the location path in the predicate doesn't exist, i.e. the location path evaluates to an empty node-set, false is returned. Otherwise, for a non-empty node-set, true is returned, thus including the context node being tested for in the new node-set (Simpson, J. E (2008)).

Again, it is useful to extend the mnemonic we have been using to predicates.

Magical Mnemonic 1:

```
SELECT axis OF context node(s) WHOSE (type) IS nodeTest
```

Magical Mnemonic 2b:

```
SELECT axis OF context node(s) WHOSE (type) IS nodeTest  
PREDICATED UPON [LocationPath]
```

Magical Mnemonic 2b (Predicate Section):

```
[LocationPath]  
= [SELECT axis OF context node(s) WHOSE (type) IS nodeTest]
```

The predicate section simply corresponds to a location path. Which we now know that when evaluated as a predicate returns true or false as follows. It returns true if at least one node specified by the location path exists, and returns false if none exist. Essentially the predicate in this form acts as an “existential quantifier”.

Consider the following example where we want to find all the “branch(s)” having at least one “picture” child element.

```
/restaurant_collection/restaurant/child::branch[picture]
```

Using our mnemonic, we have,

```
SELECT child(s) OF "/restaurant_collection/restaurant/" WHOSE (name) IS "branch"  
PREDICATED UPON  
{SELECT child(s) OF "/restaurant_collection/restaurant/" WHOSE (name) IS  
"picture"}
```

Notice again that the context of the predicate is inherited from the preceding location step.

The XPath expression test each branch node for the existence of a “picture” child element, returning true if it contains at least one. For each branch node tested as true, it is included in the new node-set. Hence, the resulting node-set contains all the branch nodes containing at least one “picture” child element.

Extending the above example, what if we want to show the opposite? In other words, find all the “branch(s)” containing no “picture” child elements at all.

```
/restaurant_collection/restaurant/child::branch[not(child::picture)]
```

By negating the predicate expression we are testing for, we are effectively testing for the non-existence of the “picture” element instead!

iii. Compound Predicates

You can test for multiple conditions in a single predicate by delimiting the multiple conditions with logical and/or operators (Simpson, J. E (2008)). Both conditions must be satisfied for the node(s) to be included in the resulting node-set.

For example,

```
/restaurant_collection/restaurant/child::branch[child::picture and  
attribute::hotline="66109666"]
```

This selects the “branch” node(s) of restaurants provided they have at least a child with one “picture” element AND an attribute “hotline” matching value “66109666”.

iv. Multiple Predicates

As specified in the grammar of a location step ([Step](#)) (figure C.III.2.1), many predicates can be included in the same location step.

However of the operation of multiple predicates is slightly different from compound predicates. The difference lies in the change of context during evaluation. Each succeeding predicate is evaluated in terms of the narrowed context provided by the preceding one.

Consider the following example:

```
restaurant_collection/restaurant/branch/menu/item[child::picture][position()=4]
```

Up till the first predicate, the resulting node-set after evaluation contains all the “item” node(s) having at least one child “picture” element as explained earlier. This node-set is further tested by the second predicate. As such, the resulting node-set only contains the 4th node among the many node(s) in the earlier node-set.

What happens if we switch the position of the 2 predicates?

```
restaurant_collection/restaurant/branch/menu/item[position()=4][child::picture]
```

Notice that we no longer get the same result. Here, we first select the 4th “item” of all the “menus” in the “restaurants”. This node-set contains much lesser nodes than its counterpart. Thereafter, this node-set is further tested by the second predicate, thus, eliminating any nodes having no “picture” child element.

Take note of the difference:

Former: Selects the 4th element, of all the “items”, containing at least one “picture” element. Later: Selects the 4th “item”, of all the “menus”, which contains at least one “picture” element.
--

7. Abbreviated Syntax

So far, we have been using unabbreviated XPath expressions. As the query becomes more and more complex, the expression can get extremely long and unwieldy. To make location paths concise and manageable, XPath defines abbreviated syntax of commonly used expressions. Figure C.III.7.1 shows 4 out of 6 of the abbreviated syntaxes we are going to cover.

AbbreviatedAbsolutePath	::=	'/' RelativeLocationPath
AbbreviatedRelativeLocationPath	::=	RelativeLocationPath '/' Step
AbbreviatedStep	::=	'.'
		'..'
AbbreviatedAxisSpecifier	::=	'@'?

Figure C.III.7.1 Grammar of abbreviated syntax

There are 6 abbreviated syntax defined in total.

i. Child Axis

child::

The “child” axis is optional.

The most commonly used abbreviation is that “child::” can be omitted from a location step. In effect, child is the default axis.

The following example lists the unabbreviated form followed by the abbreviated form.

Long form:

/child::restaurant_collection/child::restaurant

Short form:

/restaurant_collection/restaurant

These two path expressions give the same result upon evaluation.

ii. Position Predicate

position()

The position() function is optional in predicates.

When a number is placed in a predicate in isolation, it is assumed that the number is meant to be tested against the position() function. Recall that the position() function returns the position of the context node relative to the other nodes in the node-set.

For example:

Long form:

/restaurant_collection/restaurant/branch/menu/item[position()=3]

Short form,

```
/restaurant_collection/restaurant/branch/menu/item[3]
```

The “position()=” text segment is effectively appended before the digit to form a proposition which evaluates to true only when the child nodes are in position 2 relative to the context node.

iii. Attribute Axis

```
attribute:: → @
```

The “attribute::” text segment in an XPath expression can be simply replaced by “@”.

For example:

Long form:

```
/restaurant_collection/restaurant/branch[attribute::city='Singapore']
```

Short form:

```
/restaurant_collection/restaurant/branch[@city='Singapore']
```

iv. Select All Descendants of the Context Nodes

```
/descendant-or-self::node()/ → //
```

The “/descendant-or-self::node()” text segment in an XPath expression can be simply replaced by “//”.

As per the mnemonic we derived, the expression above means,

```
SELECT descendants or self OF root node WHOSE (type) IS any node
```

This selects the context node and its descendants, whatever their node type.

For example:

Long form:

```
/descendant-or-self::node()/item[name="Coke"]
```

Short form:

```
//item[name="Coke"]
```

v. Select the Context Node

```
self::node() → .
```

The “self::node()” text fragment can be replaced by just a “.”

Recall that the self axis simply selects the context node, and the node test “node()” selects any node type.

A trivial example follows:

Long form:

```
/restaurant_collection/restaurant/branch/self::node()
```

Short form:

```
/restaurant_collection/restaurant/branch/.
```

Note also that this is equivalent to:

```
/restaurant_collection/restaurant/branch
```

The “./” is redundant since the current context nodes are automatically selected.

As we have mentioned in section C.III.4 Axis, a more practical use of the self axis is to pass the context of the main path expression into the predicates.

Consider the following example:

This expression selects all the menu items which contain a “picture” node as one of its descendants.

Long form:

```
/restaurant_collection/restaurant/branch /menu/item[self::node()//picture]
```

Short form:

```
/restaurant_collection/restaurant/branch/menu/item[./picture]
```

Note that the “self::node()” or the “.” here is crucial. If it was missing, as shown below,

```
/restaurant_collection/restaurant/branch/menu/item[//picture]
```

The expression would select all the item nodes regardless of whether it has a “picture” node as a descendant. Reason being, the predicate is no longer evaluated relative to the “item” context node. Rather, the context node is taken as the root node by default. Note that this is only the case for expressions starting with “/”.

```
/restaurant_collection/restaurant/branch[attribute::city="Singapore"]
```

In this particular example, the context of “attribute::” is given by the context of its “parent” (branch) as per normal.

```
/restaurant_collection/restaurant/branch[./attribute::city="Singapore"]
```

Hence, the “./” is unnecessary.

vi. Select Parent Node

```
parent::node()
```

→

```
..
```

The “parent::node()” text segment in any XPath expression can simply be replaced by “..”

Recall the expression selects the parent of the current context node, whatever the node type.

Again let's start with a trivial example.

Long form:

```
/restaurant_collection/restaurant/branch/parent::node()
```

Short form:

```
/restaurant_collection/restaurant/branch/..
```

This is equivalent to:

```
/restaurant_collection/restaurant
```

The parent::node() thus, allows us to traverse up the XML tree.

A more practical use example follows.

```
//item[name="Coke"]/../../../rname
```

Here, we want to list all the restaurant names selling “Coke”. The expression first selects all the items named “Coke”. Thereafter, for each of the context nodes, it goes back to the parent of the current context node. It backtracks 3 steps up before it reaches the “restaurant” as their context nodes. At this point it selects the “rname” element which contains the restaurant names.

➤ **Closing remarks:** The abbreviated syntax of XPath bears a striking resemblance to UNIX file path. Hence, XPath abbreviations may be initially difficult to understand because of the cryptic nature of the shortcuts used. But after learning them, one would definitely appreciate its concision and natural ease of use.

IV. Conclusion

We have covered the core concepts of XPath location paths as a tool to address nodes in an XML document. Coverage includes the location paths and steps, absolute and relative location paths, the concept of context node(s), cascading evaluation, detailed description of axes, node tests and predicates and finally culminating in abbreviated forms.

We hope that our introductory chapter to XPath has given you a solid ground in the fundamental concepts of XPath.

For further reading, do look into XML namespace and the built-in function library available even in XPath 1.0. More advanced readers may want to explore XPath 2.0 with its new features such as sequence data model and enhanced type checking.

Chapter D: XQUERY

I. Introduction

“Data extraction, conversion, transformation and integration are well-understood database problems, and their solutions rely on a query language.” (Deutsch *et al*, n.d.) Nowadays, as XML data proliferates on the Web, more applications that can integrate and aggregate these data from multiple source as well as transform them to facilitate exchange are expected (Deutsch *et al*, n.d.), and that brings our concern to a query language of XML— XQuery. XQuery is specially designed to meet the properties of XML data, which is fundamentally different from that of relational and object-oriented data we have seen so far. It is not rigidly structured and the schema exists with the data as tags instead of independently.

“XQuery is to XML what SQL is to database tables.”(XQuery Tutorial, 1999) This statement asserts the importance and indispensability of XQuery. Up to now, XQuery is supported by all major databases as an alternative search tool to SQL to query information with XML data embedded.

In this chapter, one will explore technical details of XQuery with topics focusing on Basic syntax rules, XQuery path expressions, FLWOR expressions, Element constructors, XQuery Conditional and Quantified Expressions, Built-in Functions as well as User-Defined Functions.

II. XQuery

We will be using “**restaurant.xml**” as the example to demonstrate and explain XQuery expressions. Details of “**restaurant.xml**” can be found in Appendix. For the layout of query results in this chapter, we follow the format displayed in software *Oxygen XML Editor*. XML spy is an alternative choice.

1. XQuery Definition

Xquery is the designated query language for XML, which operates on the abstract, logical structure of an XML document, rather than its surface syntax. (Boag *et al*, 2010) The logical structure mentioned here has already been introduced to you as the concept of data model in Chapter XPath.

2. XQuery Terminology

Most of the terminologies in XQuery are coherent with terms defined in XML and XPath. XML documents are treated as trees of nodes in XQuery. XQuery is aim to select and display the information stored in or derived from targeted nodes.

3. Basic Syntax rules

According to W3schools.com (1999), syntax rules set for XQuery are simple and easy to follow.

- XQuery is case-sensitive.
Key words from FLWOR expressions and built-in functions are in lowercase form.
- XQuery elements, attributes, and variables must be valid XML names.
XQuery operates on the abstract, logical structure of an XML document, rather than its surface syntax. Before making a query, familiarise yourself about the document structure.
- An XQuery string value can be in single or double quotes.
String values include document names, attribute values, contents, etc.
- An XQuery variable is defined with a ‘\$’ symbol followed by a user-given name.
For instance, \$x with x represents nodes in the addressed level.
- XQuery comments are delimited by (: and :)
e.g. (: XQuery Comments :)

4. XQuery Path Expressions

We shall start by recalling some XPath expressions first as "every XPath expression is a valid XQuery"(Kay, 2005).

Path expressions from XPath 2.0 are adopted as a functional unit of XQuery. In a sense, XQuery needs to be able to access any nodes in an XML document. Path expressions directly support this ability by looking at the special structure of XML with both hierarchical and sequential features. Hence, in its simplest form, a piece of XQuery can be written in pure path expressions.

Think of a problem scenario where you are starving but only with four dollars in your pocket. You are desperate to find out something that is affordable to eat as quickly as possible. Given an XML document containing useful information about several restaurants, you can save yourself by constructing a path expression.

i. Select the XML document to be queried

This is accomplished with a built-in function *doc()*, e.g. *doc("restaurants.xml")* or *doc("http://SingaporeFood.com/restaurants.xml")*. Both are methods of sourcing an XML document. This will be further explained in the section of built-in functions.

ii. Path expressions to address elements of interest

With the knowledge of XPath, one can easily browse through the content, do filtering and address to nodes of your interest, i.e. *//menu/item [price<4]/name* , or *//item/[price<4]/name* equivalently.

iii. Compose into a piece of query

`doc("restaurant.xml")//menu/item[price<4]/name`

Figure D.II.4.1: A Simple Path Expression in XQuery

The above expression selects all the name elements under the item elements that have a price element with value less than 4 in document *restaurant.xml*. Note that "element" is used for description and in fact, the output of an XQuery statement can be a collection of XML elements if the query targets on an XML document. For instance, when the above path expression is executed, result returned will look like what is shown in Figure D.II.4.2.

```

<name>Coke</name>
<name>Chocolate Ice Cream</name>
<name>Chocolate Caramel Pecan ' Cheesecake</name>
<name>Milk Shake</name>
<name>Fruit juice (orange, apple, lime & mango) </name>
<name>Coke</name>
<name>Milk Shake</name>
<name>Milk Shake</name>
<name>Milk Shake</name>
<name>Coke</name>
<name>Coke</name>
<name>Avocado Milksake</name>
<name>Finger Fish</name>
<name>Avocado Milksake</name>
<name>Sweet Corn Soup</name>
<name>Coke</name>
<name>Mineral Water</name>

```

Figure D.II.4.2: A general output format of XQuery

Look into the list of food returned in Figure D.II.4.2. Most of them are actually drinks rather than edible stuff. Hence, we need to refer back to the XML structure and limit the query to non-drinks by filtering data according to the "course" attribute denoting the food types.

```
doc("restaurant.xml")//menu/item[price<4 and type/@course != "Drink"]/name
```

Figure D.II.4.3: Example of multiple filtering conditions in XQuery path expressions

Query in Figure D.II.4.3 will return fewer yet more relevant results. Notice that "&&" cannot be used in predicate to connect two statements, we use "and" instead.

```

<name>Chocolate Ice Cream</name>
<name>Chocolate Caramel Pecan ' Cheesecake</name>
<name>Finger Fish</name>
<name>Sweet Corn Soup</name>

```

Figure D.II.4.4: Results of query in Figure D.II.4.3

This kind of plain display style in Figure D.II.4.2 and Figure D.II.4.4 is referred as "flat lists" conventionally. It can be improved by building hierarchy into the result document which can be taken as a new XML file, or introduce meaningful contexts into the results for better understanding. Element constructors and enclosed expressions are the techniques involved.

5. Element Constructors and Enclosed Expressions

XQuery provides constructors that can create XML structures within a query. Constructors are provided for element, attribute, document, text, comment, and processing instruction nodes (Boag *et al*, 2010). In this section, we mainly focus on Element Constructors. As the name indicated, an element constructor is used to create an element node.

Results shown in Figure D.II.4.4 can be transformed into Figure D.II.5.1 by introducing element constructors.

```
<Collection_of_Food>
  <item>
    <name>Chocolate Ice Cream</name>
    <name>Chocolate Caramel Pecan ' Cheesecake</name>
    <name>Finger Fish</name>
    <name>Sweet Corn Soup</name>
  </item>
</Collection_of_Food>
```

Figure D.II.5.1: Example of improved output by element constructors

The original query should now be written as in Figure D.II.5.2.

```
<Collection_of_Food>
  <item>
    {(doc("restaurant.xml"))//menu/item[price<4 and type/@course != "Drink"]/name)}
  </item>
</Collection_of_Food>
```

Figure D.II.5.2: Example of XQuery embedded in element constructors

In an element constructor, curly brackets “{ }” are used to enclose function expressions, delimiting and distinguishing them from the surrounding literal text. The statement contained is thus named “enclosed expressions”. Enclosed expressions are portions to be evaluated and replaced by the evaluated results when displaying results. An alternative and slightly better version of Figure D.II.5.2 is given in Figure D.II.5.3 with output shown in Figure D.II.5.4.

```

<Collection_of_Food>
{ doc("restaurant.xml")//menu/item[price<4 and type/@course != "Drink"]/<item>{name}</item> }
</Collection_of_Food>

```

Figure D.II.5.3: Alternative way of using element constructors to Figure D.II.5.2

```

<Collection_of_Food>
  <item>
    <name>Chocolate Ice Cream</name>
  </item>
  <item>
    <name>Chocolate Caramel Pecan ' Cheesecake</name>
  </item>
  <item>
    <name>Finger Fish</name>
  </item>
  <item>
    <name>Sweet Corn Soup</name>
  </item>
</Collection_of_Food>

```

Figure D.II.5.4: Output of query in Figure D.II.5.3

One may notice that the output is still not very informative and constrained by the simplicity of path expressions. What if the food hunter wants to see the restaurant names and addresses associated with each item displayed? The problem is easily solved if one is familiar with XQuery FLWOR expressions.

6. XQuery FLWOR expressions

FLWOR expressions are merited as the workhorse of the XQuery language (Kay, 2005). The five declarative clauses build up a framework for a piece of query and make it possible for additional query features, i.e. Conditional Expressions, Quantified Expressions, Comparisons, Arithmetic, Built-in Functions and User-Defined Functions.

To start with, let us recall previous XPath expression example (Figure D.II.4.1) which prints the names of the food whose prices are less than 4 dollars in “restaurant.xml”. This can be achieved by a piece of simple FLWOR expression as well. (Figure D.II.6.1). Here \$x represents the nodes which we input into the *for* iteration. In *for* clause, we defines the path to node \$x as `doc("resturant.xml")//branch/item` using a key word *in*. Following that, a *where* clause defines the conditions that price of \$x must be less than 4 dollars and \$x cannot be of the type "Drink". Lastly, a *return* clause outputs the names of food that satisfies the requirements in *where* clause.

```
for $x in doc("restaurant.xml")//branch/item
where $x/price<4 and $x/type/@course != "Drink"
return $x/name
```

Figure D.II.6.1: A simple FLWOR Expression

However, cautions should be taken over the path expressions in FLWOR. Here we are referring to the path expressions appeared in a certain clause rather than a single path expression query. If one selects the input node *\$x* as branch elements in the *for* clause (Figure D.II.6.1), it does not give the desired results though appeared to be the same logic. Under this expression, the returned results (Figure D.II.6.3) will be all the items under each branch as long as there is one item from this branch meets the requirements in *where* clause as *\$x* now denotes the branch elements instead of the item elements.

```
for $x in doc("restaurant.xml")//branch
where $x//price<4 and $x//type/@course != "Drink"
return $x//name
```

Figure D.II.6.2: FLWOR Expression with improper path expressions

```
<name>Berry-Berry Belgian Waffles</name>
<name>Strawberry Belgian Waffles</name>
<name>Homestyle Exclusive</name>
<name>Sweet Tapioca</name>
<name>Coke</name>
<name>Belgian Waffles</name>
<name>French Toast</name>
<name>Black Beans And Red Peppers</name>
<name>Chicken Steam Hotplate</name>
<name>Chocolate Ice Cream</name>
<name>Chocolate Caramel Pecan ' Cheesecake</name>
.....
```

Figure D.II.6.3: Partial output of query in Figure D.II.6.2

7. FLWOR is an acronym for "For, Let, Where, Order by, Return"

Now we explain FLWOR expressions in a more comprehensive manner.

For, *let*, *where*, *order by*, and *return* are key words of five clauses taking place in a piece of FLWOR expression. The *return* clause is the one that always presents and it must work with at least one *for* or *let* clause. The rest of the clauses are optional. We will explain one at a time to build up understanding.

i. for

According to Kay (2005), “*for* clause iterates over an input sequence and calculates some value for each item in that sequence, returning a sequence obtained by concatenating the results of these calculations.” A *sequence* is an ordered collection of zero or more items in XQuery. It is possible to have one output item for every input item in simple cases (Figure D.II.7.1). The *for* clause iterates variable $\$x$ over the integer sequence (1, 2, 3). Then, in the *return* clause, it calculates the $\$x * \x for each number in (1, 2, 3) and returns the sequence (1, 4, 9).

```
for $x in (1 to 3)
return $x * $x
```

Figure D.II.7.1: Example of *for* clause in FLWOR Expressions

The input items and the output items in this example (Figure D.II.7.1) are atomic values. In XQuery, numbers, strings, dates, boolean values, and URIs are examples of *atomic values*. Sequences may contain XML nodes and atomic values in XQuery data model. Both of them could be used in *for* expression.

Let us look at an example which takes nodes as the input items, however, the output items are numbers. In the example (Figure D.II.7.2) which counts the number of different food which are selling by all the branches of the restaurant *Kent Ridge Highlights*, $\$x$ takes nodes under `doc("restaurant.xml")//branch` as the input. It returns 13 as the number of different food in the menu of the restaurant. Function *count()* and *distinct-values()* are built-in functions here, which we will discuss later in the section of built-in functions.

```
for $x in doc("restaurant.xml")//restaurant
where $x/rname = "Kent Ridge Highlights"
return count(distinct-values($x//name))
```

Figure D.II.7.2: FLWOR Expression with nodes as input and numbers as output

We can also input items as numbers and have nodes as the output. In Figure D.II.7.3, the query defines input $\$x$ with the sequence (1,2,3,4,5), aiming to select the first five food items in restaurant.xml. The results are shown in Figure D.II.7.4.

```
for $x in 1 to 5
return (doc("restaurant.xml")//name)[$x]
```

Figure D.II.7.3: FLWOR Expression with numbers as input and nodes as output

```
<name>Berry-Berry Belgian Waffles</name>  
<name>Strawberry Belgian Waffles</name>  
<name>Homestyle Exclusive</name>  
<name>Sweet Tapioca</name>  
<name>Coke</name>
```

Figure D.II.7.4: Output of query in Figure D.II.7.3

ii. let

The usage of *let* clause is to declare a variable with a certain value. When declaring the variable in *let* clause, name of the variable is on the left side of the “:=” sign, while the value which assigned to it is on the right side. The syntax of *let* clause is presented in Figure D.II.7.5.

```
Let $VariableName := AssignedValue
```

Figure D.II.7.5: Syntax of a let clause

For instance, Let \$price := 500, assigns 500 to the variable called \$price in the *let* clause.

Both *for* and *let* clauses are used to produce a tuple stream in FLWOR expressions. At least one bound variable is in each tuple.

According to Taylor (2007), "The *for* clause iterates over the items in the binding sequence, binding the variable to each item in turn." However, "a *let* clause binds each variable to the result of its associated expression, without iteration. ... If there are no *for* clauses, the *let* clauses generate one tuple containing all the variable bindings."

If we define the input items \$x with the sequence (8, 7, 6, 5) in the *for* clause and declare \$y with the assigned value - the sequence ("x", "y", "z"), when returns the combination of (\$x, \$y), it will iterate over each of the item in the sequence (8, 7, 6, 5) and binds the item with the “x, y, z” as a whole. “8 x y z 7 x y z 6 x y z 5 x y z” is the result returned by the query. The details of the expression are given in Figure D.II.7.6.

```
for $x in (8, 7, 6, 5)  
let $y := ("x", "y", "z")  
return ($x, $y)
```

Figure D.II.7.6: Mechanisms of *for* and *let* clause

If we declare two variables \$x and \$y as the sequences (8, 7, 6, 5) and ("x", "y", "z") in two *let* clauses respectively (Figure D.II.7.7), it will print “8 7 6 5 x y z” as the results. Besides, one can have any number of *let* and *for* clauses in a piece of FLWOR expression.

```
let $x := (8, 7, 6, 5)
let $y := ("x", "y", "z")
return ($x, $y)
```

Figure D.II.7.7: Example with two *let* clauses in a query

iii. where

The *where* clause in XQuery serves as a filter to remove unwanted items and retains the elements of interest. For those readers who has studied SQL before, it is quite similar to the SQL WHERE Clause in a SELECT statement.

If one wants to print the names of the food whose price is higher than 20 dollars, \$x as the input variable is defined in the *for* clause under the path *doc("restaurant.xml")//item* firstly. After that, we set the condition under the *where* clause to filter the unrelated item and prints the name of the desired food in the return clause. It shows us that *Honey Mustard Chicken* and *Lamp Chops* are the items we are looking for. The details of this example and the return values are given in the Figure D.II.7.8 and Figure D.II.7.9 respectively.

```
for $x in doc("restaurant.xml")//item
where $x/price>20
return $x/name
```

Figure D.II.7.8: Example of a where clause

```
<name>Honey Mustard Chicken</name>
<name>Lamp Chops</name>
```

Figure D.II.7.9: Output of query in Figure D.II.7.8

If one wants to find out the restaurant that sells *Strawberry Belgian Waffles*, the name "Strawberry Belgian Waffles" is set as the condition under *where* clause. At last, the name of the restaurant is given by the *return* clause. *Kent Ridge Highlights* is the desired restaurant (Figure D.II.7.11). Details of the expression can be found in Figure D.II.7.10.

```
for $x in doc("restaurant_finalversion.xml")//restaurant
where $x//name ="Strawberry Belgian Waffles"
return $x/rname
```

Figure D.II.7.10: Example of the where clause

```
<rname>Kent Ridge Highlights</rname>
```

Figure D.II.7.11: Output of query in Figure D.II.7.10

iv. order by and return

order by clause is used to sort the query results in either ascending or descending order.

return clause must be presented at the end of every FLWOR expression. The items which are included in the result are generated and defined by using *return* clause.

Let us take a look at the expression in Figure D.II.7.12, which selects all the name elements under the item elements and have a price element with value larger than 18 and sorts the result according to names.

Under the *for* clause, the variable *\$x* serves as a representation of all the selected item elements from the restaurant table.

As for the *where* clause, it creates a condition to only select the items with the price higher than 18.

In this case, the *order by* clause will order the result in an ascending fashion, the default order unless stated otherwise.

The *return* clause generates what should be returned. In this example, the name of the restaurants (rname elements), names of the food (name elements) and their corresponding prices (price elements) will be returned. It is worth to note that "(" and "," can be used when returning multiple expressions at one time. Additionally, the path expression ".." is used here to trace back to the parent nodes of *\$x* for "rname" retrieval. We will introduce more details about this later when we talk about element constructors for FLWOR expressions.

```
for $x in doc("restaurant.xml")//restaurant/branch//item
where $x/price>18
order by $x/name
return ($x/../../../rname,$x/name,$x/price)
```

Figure D.II.7.12: Example of the *order by* and *return* clause

The result of the XQuery expression (Figure D.II.7.12) is given in Figure D.II.7.13.

```
<lname>No Name Restaurant</lname>  
<name>Creamed Lobster With Puff Pastry Shells</name>  
<price>18.95</price>  
<lname>Nanyang Watermark Lumenouis Outlet</lname>  
<name>Fish Baked In a Bag</name>  
<price>19.99</price>  
<lname>Nanyang Watermark Lumenouis Outlet</lname>  
<name>Honey Mustard Chicken</name>  
<price>24</price>  
<lname>Nanyang Watermark Lumenouis Outlet</lname>  
<name>Lamp Chops</name>  
<price>26.99</price>
```

Figure D.II.7.13: Output of query in Figure D.II.7.12

In Summary, *for*, *let*, *where*, *order by*, *return* are the five clauses of FLWOR. The *for* and *let* clauses can appear any number of times in any order. The *where* and *order by* clauses are optional, but *where* clause must appears prior to *order by* clauses. Finally, a *return* clause must always appear in the end of each FLWOR expression.

As a functional language, XQuery expressions could be nested inside any other expressions. Moreover, it could pass the result of an expression or a function into another expression or function to calculate a value. A FLWOR expression is just an expression which does not require being at the top level of the query. Therefore, it can be used anywhere as an expression whenever it is allowed.

Now, with basic knowledge equipped of FLWOR expressions, one should be able to construct simple queries, e.g. search the address or hotline of a certain branch according to your favourite food, search by price thresholds, search by food types, order by ratings and prices, etc. Now it is time to think of query with more complicated logic and develop ways to organise the results to make more sense.

Let's apply the element constructors to improve the result display in FLWOR. First, we embed it with the starting example of FLWOR expressions, for the food hunter to see corresponding restaurant names and branches addresses associated with available food.

```

for $x in doc("restaurant.xml")//restaurant/branch//item
where $x/price<4 and $x/type/@course != "Drink"
return

```

```

<restaurant>
  { $x/../../../rname }
  <branch>
    { $x/../../@address }
    <item>
      { $x/name }
      { $x/price }
    </item>
  </branch>
</restaurant>

```

Figure D.II.7.14: FLWOR expression *return* clause modified by element constructors

In Figure D.II.7.14, ".." is applied to trace back to the parent nodes of \$x, which are the <rname> and <branch> elements to find the corresponding restaurant names and branch addresses. As discussed for Figure D.II.6.2, \$x must be specifically addressed to <item> level rather than staying above at <branch> or <restaurant> level, path expression ".." is thus important for locating nodes with higher hierarchy. The outcome for query in Figure D.II.7.14 is shown in Figure D.II.7.15.

```

<restaurant>
  <rname>Tamil Srika Haani Food Republic</rname>
  <branch address="204 Anderson Highway">
    <item>
      <name>Finger Fish</name>
      <price>3.90</price>
    </item>
  </branch>
</restaurant>
<restaurant>
  <rname>Tamil Srika Haani Food Republic</rname>
  <branch address="46 Orchard Boulevard">
    <item>
      <name>Sweet Corn Soup</name>
      <price>3</price>
    </item>
  </branch>
</restaurant>
.....

```

Figure D.II.7.15: Partial output of query in Figure D.II.7.14

With this, one can modify all FLWOR expressions shown previously to enhance their display styles. For instance, the example in Figure D.II.7.2 with *count()* function now can be transformed into Figure D.II.7.16.

Similarly, to order all the branches by their ratings from highest to the lowest and display the associated the nameS of the restaurants and branches addresses, we can write query as in Figure D.II.7.17. The list of qualified restaurants is displayed in Figure D.II.7.18 with details.

```
for $x in doc("restaurant.xml")//branch
return
    <restaurant>
        {$x/../rname}
        <branch>
            {$x/@address}
        <number_of_items>{count($x//name)}</number_of_items>
        </branch>
    </restaurant>
```

Figure D.II.7.16: Query in Figure 3 enhanced by element constructors

```
for $x in doc("restaurant.xml")//branch
order by $x/@rating descending
return
    <restaurant>
        {$x/../rname}
        <branch>
            {$x/@branch_id,$x/@rating}
        </branch>
    </restaurant>
```

Figure D.II.7.17: Order by ratings with element constructors

```

<restaurant>
  <lname>Tamil Srika Haani Food Republic</lname>
  <branch branch_id="branch823" rating="3"/>
</restaurant>
<restaurant>
  <lname>Kent Ridge Highlights</lname>
  <branch branch_id="branch02" rating="2"/>
</restaurant>
<restaurant>
  <lname>No Name Restaurant</lname>
  <branch branch_id="branch003" rating="1"/>
</restaurant>
.....

```

Figure 1: Partial output of query in Figure D.II.7.17

Query in Figure D.II.7.17 gives a hint on how to display information in attributes. To simplify the problem, suppose $\$x$ is now at `<branch>` level, and we want to know only the attribute values of `<branch>`. Query in Figure D.II.7.19 will return a list of `<branch address="@#%">`, however in fact, only `address="@#%"` is from the original XML document and it is merged with external `<branch>` tag from element constructors.

```

<branch>
  { $x/@address }
</branch>

```

Figure D.II.7.19: Display information in attributes by element constructors

Cautions on path expressions should be taken again once to apply this in FLWOR expressions. A failed query is shown in Figure D.II.7.20. An error will be reported upon its execution with error message showing "cannot create an element having two attributes with the same name @price".

```

for $x in doc("restaurant.xml")//menu
return
  <set>{ $x/set/@price }</set>

```

Figure D.II.7.20: Failed query with ignorance on path expressions

Let's interpret the query with respect to its path expression. Now $\$x$ is at `<menu>` level, when *for* clause iterates through every `<menu>` element, `$x/set/@price`, however, will return price information of every set under a certain menu at the same time, which the parser will tend to add into `<set>` at the same time and result in error. To resolve the problem, one can let $\$x$ go down to `<set>` level and let the *for* clause to iterate through each set.

Always bear in mind that, though FLWOR is a kind of declarative language, the inner mechanism might be rather different from its surface syntax. Sometimes this requires deep thought with your path expressions.

8. Conditional Expression:

An "If-Then-Else" statement could help to set a condition for returning results. Thus, it is found at under the *return* clause in FLWOR expressions.

Figure D.II.8.1 shows the syntax of conditional expressions, slightly modified from MSDN – conditional (2011), so as to make the syntax more meaningful. Instead of showing the word “expression”, the steps “Whatever you wanted to display” and “it is optional to display anything”, that provides the meaning is used. The Figure D.II.8.2 presents a query example of conditional expressions. The query in Figure D.II.8.2 is to find all the food priced greater than \$17.00. So according to the “restaurant.xml” example, the result is showed in Figure D.II.8.3.

```
if (<condition>)
then
  <Whatever you wanted to display>
else
  <it is optional to display anything>
```

Figure D.II.8.1: Syntax of conditional expressions

```
for $x in doc("restaurant.xml")//item
return
  if ($x/price > 17)
  then ($x/name, $x/price)
else ()
```

Figure D.II.8.2: Example of if-else query statement

```
<name>Creamed Lobster With Puff Pastry Shells</name>
<price>18.95</price>
<name>Honey Mustard Chicken</name>
<price>24</price>
<name>Lamp Chops</name>
<price>26.99</price>
<name>Fish Baked In a Bag</name>
<price>19.99</price>
```

Figure D.II.8.3: Results of query statement in Figure D.II.8.2

Notes: parentheses around the ‘if’ expression are required. ‘else’ is required, but it can be just else (), if there is nothing to return as shown in Figure II.8.2. The expression following *then* and *else* statement should be either enclosed by element constructors or "()" to avoid syntax error. Examples are found in Figure D.II.8.4a & D.II.8.4b.

```
for $x in doc("restaurant.xml")//gift
return
  if($x/toy)
  then
    <toy>{$x/toy/@name}</toy>
  else
    <voucher>{$x/voucher/@id}</voucher>
```

Figure D.II.8.4a: If-Then-Else statement with element constructors

```
for $x in doc("restaurant.xml")//gift
return
  if ($x/toy)
  then ($x)
  else
    <gift>No toy for this set</gift>
```

Figure D.II.8.4b: If-Then-Else statement with “()” to enclose expressions

Notice that *else* includes all possibilities that are excluded from if, and once else is not made use of and reduced to else() form, an if-then-else statement can be expressed in a basic FLWOR expression with where clause. Examples are found in Figure D.II.8.5a and D.II.8.5b.

```
for $x in doc("restaurant.xml")//gift
return
  if ($x/toy)
  then($x)
  else ()
```

Figure D.II.8.5a: Comparison of If-Then-Else statements with simple FLWOR expressions


```
for $x in doc("restaurant.xml")//gift
where $x/toy
return $x
```

Figure D.II.8.5b: Comparison of If-Then-Else statements with simple FLWOR expressions

9. Quantified Expression:

Quantified expressions in XQuery concern the use of universal and existential quantifiers in XQuery semantics. It assists to determine whether at least one item in a sequence satisfies a condition, or whether every item in a sequence satisfies a condition.

Figure D.II.9.1 shows the syntax of quantified expressions in XQuery. Take a look at Figure D.II.9.2, which presents an example of existential quantifier in XQuery (MSDN - quantified, 2011). The translation of it is to find all food with prices greater than \$17.00, however uses the quantifier expression. The result displayed is shown in Figure D.II.9.3.

```
( some | every ) <variable> in <Expression> (,...) satisfies <Expression>
```

Figure D.II.9.1: Syntax of quantified expressions in XQuery

```
let $d1 := doc("restaurant.xml")
for $t1 in $d1//item
where some $t2 in (
    for $t3 in $d1//item
    return $t3 )
satisfies $t1 = $t2 and $t1/price>17
return
<result> { $t1//name }</result>
```

Figure D.II.9.2: Example of quantified expression

```

<result>
  <name>Creamed Lobster With Puff Pastry Shells</name>
</result>
<result>
  <name>Honey Mustard Chicken</name>
</result>
<result>
  <name>Lamp Chops</name>
</result>
<result>
  <name>Fish Baked In a Bag</name>
</result>

```

Figure D.II.9.3: Result of Figure D.II. 9.2 query statement

Do note that the implementation of quantified expressions depends on the usage, what the user wish to obtain. It may be used as a means of check, as shown in Figure D.II.7.9 or it may be used in *for* clause to just display whichever items that fits the conditions given.

10.Arithmetic

XQuery supports the usual arithmetic operations like +, -, *, div, idiv, mod.

Notice that there is a different between *div* and *idiv*, where *div* performs division on any numeric type, whereas *idiv* require integer arguments and returns an integer as result, rounding towards 0. An example of arithmetic usage is `2 + <int> { 2 } </int>` the result is 4.

However, if the operand is an untyped data, it will cast to a double, raising an error if the cast fail.

```

let $p := doc("restaurant.xml")//item/price
return <result>{$p[1] + $p[2]}</result>

```

Figure D.II.10.1: Example of arithmetic in FLWOR expression

Displayed result will be like `<result>14.45</result>`.

Result is adding first price and second price of the data, returning a value in double type.

Comparison

In XQuery there are two ways of comparing values.

1. General comparisons: =, !=, <, <=, >, >=
2. Value comparisons: eq, ne, lt, le, gt, ge

Examples of the two ways are shown in Figure D.II.11.1 & D.II.11.2.

```
let $p := doc("Restaurant.xml")//item/price > 17  
  
OR  
  
for $p in doc("Restaurant.xml")//item/price >17
```

Figure D.II.11.1: Example of comparison statements in FLWOR

The above expression (Figure D.II.11.1) will return true if **any** price attributes have a value greater than 17. According to the "restaurant.xml" example, the result displayed will be "true". Indeed there is food item priced above \$17.00.

```
let $p := doc("Restaurant.xml")//item/price gt 17
```

Figure D.II.11.2: Alternative example of comparison statements in FLWOR

This expression however returns true if **there is only one** price attribute returned by the expression, and its value is greater than 17. **If more than one price is returned, an error occurs.** Using the "restaurant.xml" example, the result displayed will be "false", however if to change 17 to 24, then it will display "true". Reason is that there is exactly one food item priced above \$24, that is \$26.99 item "Lamp Chop".

11. Built-in functions:

XQuery 1.0 and XPath 2.0 share the same built-in function library. Built-in functions are pre-defined functions in XQuery which can be directly embedded into a piece of query to meet more sophisticated search requirements. W3C has documented them clearly at <http://www.w3.org/TR/xpath-functions>. Generally, they can be categorised into aggregate functions, functions on numerical values, strings, durations, dates and times, nodes, sequences, etc. Here we give some examples on those frequently appeared ones, especially the powerful aggregate functions.

Aggregate Functions	Meaning
fn:count	Returns the number of items in a sequence.
fn:avg	Returns the average of a sequence of values.
fn:max	Returns the maximum value from a sequence of comparable values.
fn:min	Returns the minimum value from a sequence of comparable values.
fn:sum	Returns the sum of a sequence of values.

fn: denotes the namespace of those functions. Since all built-in functions share the same namespace of *fn*, we do not have to indicate it explicitly when calling them in a query.

Function *count()* has already appeared in Figure D.II.7.2 & D.II.7.16 to count the number of names of food in each case.

With function *max()*, to write a query retrieving the most expensive dish becomes a piece of cake as compared to the hard work with SQL. The query to find the most expensive food among all restaurants would be like in Figure D.II.12.1, with the result \$26.99.

```
for $x in doc("restaurant.xml")/restaurant_collection
return max($x//price)
```

Figure D.II.12.1: Most expensive food
price of restaurant_collection

In a similar manner, it is easy to find out the maximum price of dish in each restaurant or in each branch of one the restaurant by simply altering the path expressions in *for* clause.

Now think of this, what will be the result if we write query like in Figure D.II.12.2?

```
for $x in doc("restaurant.xml")//item
return max($x/price)
```

Figure D.II.12.2

Additionally, if one wants to display the name of the most expensive food, an additional variable \$y is required. This is to avoid the improper path expressions that we have seen previously as now \$x is at level <restaurant_collection>, which is too far away from <item> and will not return us the correct name. One attempt could be like in Figure D.II.12.3. Notice that there are two *for* clauses to have two separate iterations for x and y.

```
for $x in doc("restaurant.xml")/restaurant_collection
for $y in doc("restaurant.xml")//item
where $y/price = max($x//price)
return $y/name
```

Figure D.II.12.3: Example of two *for* clauses in a piece of query

The result would be like in Figure D.II.12.4.

```
<name>Lamp Chops</name>
```

Figure D.II.12.4: Output of query in Figure D.II.12.3

A *let* clause is not applied in the case of Figure D.II.12.3 as it takes the assigned value as a whole chunk at one time rather than iterates over it. This, however, can be overcome by introducing iterating variables into a *let* clause. The example below is a good demonstration and it makes use of another aggregate function named *avg()*. The average price of each restaurant is computed by query in Figure D.II.12.5. Nevertheless, this trick cannot be applied to Figure D.II.12.3 as \$x and \$y should iterate independently in their levels.

```
for $x in doc("restaurant.xml")//restaurant
let $a := avg(doc("restaurant.xml")//restaurant[rname = $x/rname]//item/price)
return
  <Restaurant>
    {$x/rname}
    <avgprice> {$a} </avgprice>
  </Restaurant>
```

Figure D.II.12.5: Example of using *avg()* function in query and introducing a iterative variable into a *let* clause

Here we give two more tables of functions on numeric values and strings.

Functions on Numerical Values	Meaning
fn:abs	Returns the absolute value of the argument.
fn:ceiling	Returns the smallest number with no fractional part that is greater than or equal to the argument.
fn:floor	Returns the largest number with no fractional part that is less than or equal to the argument.
fn:round	Rounds to the nearest number with no fractional part.
fn: round-half-to-even	Takes a number and a precision and returns a number rounded to the given precision. If the fractional part is exactly half, the result is the number whose least significant digit is even.

Functions on String	Meaning
fn:codepoints-to-string	Creates an xs:string from a sequence of Unicode code points.
fn:string-to-codepoints	Returns the sequence of Unicode code points that constitute an xs:string.
fn:compare	Returns -1, 0, or 1, depending on whether the value of the first argument is respectively less than, equal to, or greater than the value of the second argument, according to the rules of the collation that is used.
fn:concat	Concatenates two or more xs:anyAtomicType arguments cast to xs:string.
fn:substring	Returns the xs:string located at a specified place within an argument xs:string.
fn:string-length	Returns the length of the argument.
fn:upper-case	Returns the upper-cased value of the argument.
fn:lower-case	Returns the lower-cased value of the argument.

Additionally in XQuery, if one is not sure about exact name to search and wants to query with a keyword, the function *contains()* is useful. Example in Figure D.II.12.6 shows what to do if one would like to know all varieties of the waffles offered. Unfortunately, as XQuery is case-sensitive, results will only be returned if "Waffles" is used, nothing if "waffles". The result would be like in Figure D.II.12.7.

```
for $x in doc("restaurant.xml")//item
where contains($x/name,"Waffles")
return ($x/name, $x/price)
```

Figure D.II.12.6: Example of keyword search in XQuery by built-in function *contains()*

```
<name>Berry-Berry Belgian
Waffles</name>
<price>7.95</price>
<name>Strawberry Belgian Waffles</name>
<price>6.50</price>
<name>Belgian Waffles</name>
<price>4.95</price>
```

Figure D.II.12.7: Output of query in Figure D.II.12.5

Another useful built-in function is *data()*. It can be used to retrieve atomic values of elements, leaving off the the original schema associated. *Data()* offers a way to adjust the schema of the result display. With query in Figure D.II.12.8, one would expect result in Figure D.II.12.9 instead of Figure D.II.12.10.

```
for $x in doc("restaurant.xml")//restaurant
return <R>{data($x/rname)}</R>
```

Figure D.II.12.8: Renaming tags of elements by *data()* function

```

<R>Kent Ridge Highlights</R>
<R>No Name Restaurant</R>
<R>Crescent Lodge Country</R>
<R>Nanyang Watermark Lumenouis Outlet</R>
<R>Tamil Srika Haani Food Republic</R>

```

Figure D.II.12.9: Output of query in Figure D.II.12.7

```

<rname>Kent Ridge Highlights</rname>
<rname>No Name Restaurant</rname>
<rname>Crescent Lodge Country</rname>
<rname>Nanyang Watermark Lumenouis Outlet</rname>
<rname>Tamil Srika Haani Food Republic</rname>

```

Figure D.II.12.10: Results with original schema

Besides altering schema, it's also possible to alter the query by removing unwanted element tags. Here we give an example which is derived from the previous query in Figure D.II.8.2, and the results returned put name and price together under the same element tag `<item>`. It is obvious that the size of returned elements (Figure D.II.12.12) is significantly reduced compared with Figure D.II.8.3.

```

for $x in doc("restaurant.xml")//item
return
  if ($x/price > 17)
  then <item>{data($x/name), data($x/price)}</item>
  else ()

```

Figure D.II.12.11: Alteration of query in Figure D.II.8.2 with function `data()`

```

<item>Creamed Lobster With Puff Pastry Shells 18.95</item>
<item>Honey Mustard Chicken 24</item>
<item>Lamp Chops 26.99</item>
<item>Fish Baked In a Bag 19.99</item>

```

Figure D.II.12.12: Output of query in Figure D.II.12.10

12. User-defined functions:

User-defined functions are introduced into XQuery for the convenience sake of the user, especially when a query gets larger and more complex. User-defined functions can be

defined right in the query or in a separate library. The advantage of a library is that these functions could then be re-used in other query. The syntax is given in Figure D.II.13.1.

```
declare function prefix:function_name($parameter as datatype)
as returnDatatype
{
  ...function code here...
};
```

Figure D.II.13.1: Syntax of user-defined funtions

A recursive function that compute the depth of one XML document can be defined as in Figure D.II.13.2 using user-defined function technique. *Local* is a prefix and *depth* is function name with *empty* being another built-in function. For the insight explanation of **item type** (*node()*, \$x takes value of nodes) and **occurrence indicator** (*, denotes any number of nodes returned as \$x) appeared in the function definition, one can read more if interested with a good online reference addressed at http://www.stylusstudio.com/xquery/xquery_functions.html.

```
declare function local:depth($x as node(*) as xs:integer
{
  if (empty($x/*))
  then 1
  else (max(local:depth($x/*)) + 1)
};
local:depth(doc("restaurant.xml"))
```

Figure D.II.13.2: Example of a user-defined function

The content in Figure D.II.13.2 can appear directly in a query file. The last line calls the function and applies it to the restaurant example. Result shows an integer number 8 which has been justified correct by tracing through the hierarchy of “restaurant.xml”. One trivial thing to note is that, the semicolon after the closing brace at the end of the declared function is compulsory for the parser.

13. Path expressions vs FLWOR expressions

At this stage, one may raise a common question about the trade-off between Xquery path expressions and FLWOR expressions as they appear to be two things when introduced and FLWOR seems to take the lead in XQuery. Well, generally speaking, FLWOR has higher flexibility to adapt itself to larger number of query cases and thus more frequently used. However, one should notice that there are cases that XPath expressions and FLWOR make little difference and cases that mixed expressions appeared in query, e.g. Figure D.II.12.5. In

fact, there are no objective rules set to differentiate between them and reasons to justify the use of a certain one could be rather subjective. In the first place, the choice is dependent on personal habits. If one is more acquainted with path expression, there is higher chance that he (she) will write most of simple query in path expression unless FLWOR expression is necessary. On the other hand, under certain circumstances, FLWOR gives more power to create a well-formed query, e.g. when multi-variables are expected or when ordering of result is required. This kind of uncertainty in writing query rises from the adaptability of XQuery and it does give the user much space to establish their creativity. As a rule of thumb, since "all roads lead to Rome", all we need to pay attention is to follow the syntax and make sure we understand the internal mechanisms of every operation in query.

III. Conclusion

Overall, XQuery is a functional language which can be used to query XML. Through this chapter, we have seen simple XQuery path expressions and specially designed FLWOR expressions for basic query. Element constructors and enclosed expressions, conditional and quantified expressions, built-in and user-defined functions have been introduced to add in additional features to XQuery. Finally, for readers who want to do further study, the next step to learn about could be XLink and XPointer, which are used to define the standard way to create hyperlinks in XML documents.

For a long-term perspective of XQuery, it is replacing proprietary middleware languages, Web Application development languages, complex Java or C++ programs, as it is simpler to work with and easier to maintain than many other alternatives. (Marchiori, M.& Quin, L., 2011)

For more standards set by W3C, please refer to the documents below.

[XQuery 1.0: An XML Query Language]

[XML Syntax for XQuery 1.0]

[XML Query 1.0 Requirements]

[XQuery 1.0 and XPath 2.0 Functions and Operators]

[Namespaces in XML]

Conclusion

Out of the consideration of objectiveness, this book emphasizes exclusively on the "hard facts" of XML. With main references being W3C recommendations, the writing group made efforts in compacting contents related with topics of XML, DTD, XPath and XQuery after some work with literature reviews. In short, XML, progressing to be the basis of a new era of data processing, has been dissected and discussed with respect to its syntax, namespaces property, usage of CDATA and well-formness. Following that, we have seen how to use DTDs to define and constrain the structure and content of XML data. Chapter of XPath gives a detailed view on XPath Data Model and how XPath expressions and location paths work. In the end, to realise query to XML documents, path expressions, FLWOR expressions, sometimes integrated with built-in functions, have been demonstrated in chapter of XQuery with case studies.

Bibliography

Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., & Siméon, J. (2010). *XQuery 1.0: An XML Query Language (Second Edition)*. Retrieved March 30, 2011, from <http://www.w3.org/TR/xquery/>

Deutsch, A., Fernandez, M., Florescu, D., Levy, A., Suciu, D. (n.d.). *A Query Language for XML*. Retrieved March 29, 2011, from <http://www8.org/w8-papers/1c-xml/query/query.html>

Document Type Definition. (2011). Retrieved March 30, 2011, from Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Document_Type_Definition

Extensible Markup Language (XML): Predefined Entities. (n.d.). Retrieved from stylusstudio.com: <http://www.stylusstudio.com/w3c/xml11/sec-predefined-ent.htm>

Holzner, S. (1998). *XML Complete*. New York: McGraw - Hill.

James Clark, Steve DeRose. (1999). *XML Path Language (XPath) Version 1.0 W3C Recommendation (1999)* Retrieved April 12, 2011, from w3.org: <http://www.w3.org/TR/xpath/>

John E. Simpson. (2008). *XPath and Xpointer*. Retrieved April 12, 2011, from http://commons.oreilly.com/wiki/index.php/XPath_and_XPointer

Kay, M. (2005). *Blooming FLWOR - An Introduction to the XQuery FLWOR Expression*. Retrieved March 28, 2011, from Progress Data Direct: <http://www.xquery.com/tutorials/flwor/>

Kay, M. (2008). *XSLT 2.0 and XPath 2.0 Programmer's Reference*. John Wiley and Sons.

Koehler, K. R. (2002). *Directed Graphs*. Retrieved March 21, 2011, from <http://www.rwc.uc.edu/koehler/comath/33.html>

Marchiori, M., Quin, L. (2011). *W3C XML Query (XQuery):30,000 foot view*. Retrieved March 28, 2011, from <http://www.w3.org/XML/Query/>

MathWorld, W. (n.d.). *Graph*. Retrieved March 13, 2011, from [mathworld.wolfram.com: http://mathworld.wolfram.com/Graph.html](http://mathworld.wolfram.com/Graph.html)

Conditional Expression (XQuery). (2011). Retrived March 15, 2011, from MSDN: <http://msdn.microsoft.com/en-us/library/ms190171.aspx>

Quantified Expression (XQuery). (2011). Retrived March 15, 2011, from MSDN: <http://msdn.microsoft.com/en-us/library/ms189927.aspx>

Quackit. (n.d.). *Quackit Webmaster Tutorial*. Retrieved February 8, 2011, from www.quackit.com

- Schwartzbach, A. M. (2006). *An Introduction to XML and Web Technologies*. Addison Wesley.
- Simon St. Laurent, Robert Biggar. (1999). *Inside XML DTDs*. McGraw-Hill Professional.
- St.Laurent, S. (1998). *Why XML?*. Retrieved March 22, 2011, from simonstl.com:
<http://www.simonstl.com/articles/whyxml.htm>
- Taylor, A.G. (2007). *SQL All-in-One Desk Reference for Dummies: SQL and XML* (pp.544-555). Indiana: Wiley.
- Tizag.(n.d.). *Tizag Tutorials*. Retrieved February 14, 2011, from <http://www.tizag.com/>
- W3Schools Online Web Tutorials. (1999). Retrieved March 30, 2011, from w3schools.com:
<http://www.w3schools.com>
- Wikipedia. (2011). *XML validation*. Retrieved March 30, 2011, from Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/XML_validation
- Williamson, H. (2001). *XML: The complete Reference*. Osborne/McGraw-Hill.
- World Wide Web Consortium (W3C). (n.d.). *W3C workshop*. Retrieved March 30, 2011, from <http://www.w3.org/>
- xmlwriter.net. (n.d.). *Xml Declaration*. Retrieved March 16, 2011, from xmlwriter.net:
http://xmlwriter.net/xml_guide/xml_declaration.shtml
- Xml Tutorial. (1999). Retrieved February 8, 2011, from w3schools.com:
<http://www.w3schools.com/xml>
- Xpath axis picture (n.d.). *XPath axis*. Retrieved April 15, 2011, from
<http://home.lu.lv/~mihails/tts/111/xpath/index.html>
- XQuery Syntax. (1999). Retrieved March 30, 2011, from w3schools.com:
http://www.w3schools.com/xquery/xquery_syntax.asp
- XQuery Tutorial. (1999). Retrieved March 29, 2011, from w3schools.com:
<http://www.w3schools.com/xquery/default.asp>

Appendix

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<!--define the location of the external DTD using a relative URL address-->
<!DOCTYPE restaurant_collection SYSTEM "restaurant.dtd">

<!--this is a processing instruction-->
<?image-gif scale="50%"?>

<restaurant_collection>
  <!--restaurant definition-->
    <restaurant>
      <name>Kent Ridge Highlights</name>
      <rtype type_of_restaurant="FastFood" type_of_cuisine="Western Exclusive Cooking Styles"/>

      <!--first branch of Kent Ridge Highlights-->
      <branch branch_id="branch01" address="87 Kent Ridge Road" city="Singapore" hotline="66109666" rating="4">
        <picture title="branch logo representation" source="www.flickr.com/kentridge/logo.gif" type="image-
gif"/>

        <menu>
          <item item_id="item100">
            <name>Berry-Berry Belgian Waffles</name>
            <type course = "Appetizer"/>
            <price>7.95</price>
            <description>light Belgian waffles covered with an assortment of fresh berries and
whipped cream</description>

            <picture title="item num 100" source="www.flickr.com/kentridge/item100.png"
type="image-svg"/>

          </item>
          <item item_id="item101">
            <name>Strawberry Belgian Waffles</name>
            <type course = "Appetizer"/>
            <price>6.50</price>
            <description>light Belgian waffles covered with strawberries and whipped
cream</description>

            <picture title="item num 101" source="www.flickr.com/kentridge/item101.png"
type="image-svg"/>

          </item>
          <item item_id="item102">
            <name>Homestyle Exclusive</name>
            <type course = "MainDish"/>
            <price>12.50</price>
            <description>two eggs, bacon or sausage, toast, and our ever-popular hash
browns</description>

            <picture title="item num 102" source="www.flickr.com/kentridge/item102.png"
type="image-svg"/>

          </item>
          <item item_id="item103">
            <name>Sweet Tapioca</name>
            <type course = "Dessert"/>
            <price>4.95</price>
            <description>Steamed Sweet Tapioca with Coconut Creame</description>
          </item>
          <item item_id="item104">
            <name>Coke</name>
            <type course = "Drink"/>
            <price>1.95</price>
          </item>
        </menu>
      </branch>
    </restaurant>
  </restaurant_collection>
```

```

        </menu>
    </branch>

    <!--this is second branch-->
    <branch branch_id="branch02" address="145 Thompson Road" city="Singapore" hotline="66106999" rating="2">
        <picture title="restaurant logo representation" source="www.flickr.com/kentridge/logo.gif"
type="image-gif"/>

        <picture title="branch logo representation" source="www.flickr.com/kentridge/20/logo.gif"
type="image-gif"/>

        <menu>
            <item item_id="item200">
                <name>Belgian Waffles</name>
                <type course = "Appetizer"/>
                <price>4.95</price>
                <description>two of our famous Belgian Waffles with plenty of real maple
syrup</description>

                <picture title="item num 200" source="www.flickr.com/kentridge/item200.png"
type="image-svg"/>
            </item>
            <item item_id="item201">
                <name>French Toast</name>
                <type course = "Appetizer"/>
                <price>4.50</price>
                <description>thick slices made from our homemade sourdough bread</description>
            </item>
            <item item_id="item202">
                <name>Black Beans And Red Peppers</name>
                <type course = "MainDish"/>
                <price>6.80</price>
                <picture title="item num 202" source="www.flickr.com/kentridge/item202.png"
type="image-svg"/>
            </item>
            <item item_id="item203">
                <name>Chicken Steam Hotplate</name>
                <type course = "MainDish"/>
                <price>8.70</price>
                <description>A Chicken drumstick steamed with hot white rice</description>
                <picture title="item num 203" source="www.flickr.com/kentridge/item203.png"
type="image-svg"/>
            </item>
            <item item_id="item204">
                <name>Chocolate Ice Cream</name>
                <type course = "Dessert"/>
                <price>2.50</price>
            </item>
            <item item_id="item205">
                <name>Chocolate Caramel Pecan ' Cheesecake</name>
                <type course = "Dessert"/>
                <price>2.70</price>
            </item>
            <item item_id="item206">
                <name>Milk Shake</name>
                <type course = "Drink"/>
                <price>3.95</price>
            </item>
            <item item_id="item207">
                <name>
                    <![CDATA[Fruit juice (orange, apple, lime & mango) ]]>
                </name>
                <type course = "Drink"/>
                <price>2.20</price>
            </item>
        </menu>
    </branch>

```

```

        <set price="&sgd; 25.9">
            <item_set item_id="item200 item203 item204 item206"/>
            <gift>
                <toy name="Ball pen and notebook" description="an exclusive ball pen
and notebook with logo of our restaurant"/>
            </gift>
        </set>
        <set price="&sgd; 30.8">
            <item_set item_id="item201 item202 item203 item205 item207"/>
            <gift>
                <voucher id="voucher220" value="&sgd; 10" expired="one month"/>
            </gift>
        </set>
        <set price="15.8">
            <item_set item_id="item200 item203 item207"/>
        </set>
    </menu>
</branch>
</restaurant>

<!-- second restaurant -->
<restaurant>
    <rname>No Name Restaurant</rname>
    <rtype type_of_cuisine="Sea Food Special"/>

    <!--first branch of No Name Restaurant-->
    <branch branch_id="branch001" address="98 Kallang Road" hotline="67109688" rating="3">
        <picture title="branch logo representation" source="www.comp.nus.edu.sg/~tbthanh/noname/logo.gif"
type="image-gif"/>
        <menu>
            <item item_id="item1000">
                <name>Lobster Meat On a Salad</name>
                <type course = "MainDish"/>
                <price>16.95</price>
                <description>Salad made with rock lobster, celery, almonds, green onion,
mayonnaise, lemon juice, and other ingredients</description>
            </item>
            <item item_id="item1001">
                <name>Lobster Stew with Butter and Cream</name>
                <type course = "MainDish"/>
                <price>14.50</price>
                <description>lobster stew with lobster meat, heavy cream, milk, and butter. An
easy and wonderful dish for a special celebration, lobster stew.</description>
            </item>
            <item item_id="item1002">
                <name>Chilled Lobster With Mimosa Dressing</name>
                <type course = "Appetizer"/>
                <price>10.50</price>
                <description>A chilled lobster first course or appetizer with sweet mimosa dressing
or sauce</description>
                <picture title="item num 1002"
source="www.comp.nus.edu.sg/~tbthanh/noname/item1002.png" type="image-svg"/>
            </item>
            <item item_id="item1003">
                <name>Creamed Lobster With Puff Pastry Shells</name>
                <type course = "MainDish"/>
                <price>18.95</price>
                <description>This creamed lobster recipe makes a delicious and special meal, with
chunks of lobster, seasonings, Parmesan cheese, and puff pastry shells</description>
            </item>
            <item item_id="item1004">
                <name>Coke</name>

```



```

        <type course = "Drink"/>
        <price>3.95</price>
    </item>
    <item item_id="item1005">
        <name>Lobster Supreme</name>
        <type course = "MainDish"/>
        <price>13.50</price>
        <description>Lobster with macaroni and a cheese sauce, along with cooked green
peas. Lobster supreme casserole bake with macaroni. </description>
        <picture title="item num 1005"
source="www.comp.nus.edu.sg/~tbthanh/noname/item1005.png" type="image-svg"/>
    </item>
    <item item_id="item1006">
        <name>Lobster Egg Sandwich</name>
        <type course = "Appetizer"/>
        <price>13.50</price>
        <description>Lobster meat and eggs are scrambled with bacon and onion and
peppers in butter</description>
        <picture title="item num 1006"
source="www.comp.nus.edu.sg/~tbthanh/noname/item1006.png" type="image-svg"/>
    </item>
    <item item_id="item1007">
        <name>Milk Shake</name>
        <type course = "Drink"/>
        <price>2.5</price>
    </item>
    <set price="&sgd; 29.95">
        <item_set item_id="item1002 item1003 item1004"/>
        <gift>
            <toy name="Lobster Souvenir" description="an exclusive present with
logo of our restaurant"/>
        </gift>
    </set>
    <set price="&sgd; 29.95">
        <item_set item_id="item1002 item1003 item1007"/>
    </set>
    <set price="&sgd; 29.95">
        <item_set item_id="item1002 item1000 item1007"/>
    </set>
    <set price="&sgd; 40.95">
        <item_set item_id="item1000 item1005 item1007 item1006 item1003"/>
        <gift>
            <voucher id="vouche1220" value="&sgd; 15" expired="one month"/>
        </gift>
    </set>
</menu>
</branch>

<!--this is second branch-->
<branch branch_id="branch002" address="25 Clementi Road" city="Singapore" hotline="66106988" rating="5">
    <picture title="restaurant logo representation"
source="www.comp.nus.edu.sg/~tbthanh/noname/logo.gif" type="image-gif"/>
    <picture title="branch logo representation"
source="www.comp.nus.edu.sg/~tbthanh/noname/20/logo.gif" type="image-gif"/>
    <picture source="www.comp.nus.edu.sg/~tbthanh/noname/20/logo2.png" type="image-svg"/>
    <picture source="www.comp.nus.edu.sg/~tbthanh/noname/20/logo3.png" type="image-gif"/>
    <menu>
        <item item_id="item2000">
            <name>Lobster Egg Sandwich</name>
            <type course = "Appetizer"/>
            <price>10.95</price>
            <description>Lobster meat and eggs are scrambled with bacon and onion and

```

```

peppers in butter</description>
<picture title="item num 2000"
source="www.comp.nus.edu.sg/~tbthanh/noname/20/item2000.png" type="image-svg"/>
</item>
<item item_id="item2001">
<name>Lobster Corn Chowder</name>
<type course = "Appetizer"/>
<price>9.95</price>
<description>This tasty lobster chowder recipe is made with chunks of cooked
lobster, green onions, celery, a little red pepper, corn, and seasonings</description>
</item>
<item item_id="item2002">
<name>Lobster Roll </name>
<type course = "MainDish"/>
<price>6.80</price>
<description>A delicious lobster roll with cooked lobster, mayonnaise, and celery,
along with seasonings.</description>
<picture title="item num 202"
source="www.comp.nus.edu.sg/~tbthanh/noname/20/item202.png" type="image-svg"/>
</item>
<item item_id="item2003">
<name>Lobster Salad with Celery and Mayonnaise</name>
<type course = "MainDish"/>
<price>8.70</price>
<description>This lobster salad is made with cooked lobster, mayonnaise, celery,
and lemon juice.</description>
<picture title="item num 203"
source="www.comp.nus.edu.sg/~tbthanh/noname/20/item203.png" type="image-svg"/>
</item>
<item item_id="item2004">
<name>Chocolate Ice Cream</name>
<type course = "Dessert"/>
<price>4.50</price>
</item>
<item item_id="item2005">
<name>Boiled Lobster</name>
<type course = "MainDish"/>
<price>12.8</price>
<description>The traditional cooking method, serve with plenty of melted
butter.</description>
</item>
<item item_id="item2006">
<name>Milk Shake</name>
<type course = "Drink"/>
<price>2.75</price>
</item>
<item item_id="item2007">
<name>Lobster Supreme</name>
<type course = "MainDish"/>
<price>11.20</price>
<description>Lobster with macaroni and a cheese sauce, along with cooked green
peas. Lobster supreme casserole bake with macaroni. </description>
<picture title="item num 1005"
source="www.comp.nus.edu.sg/~tbthanh/noname/item1005.png" type="image-svg"/>
</item>
<set price="&sgd; 23.9">
<item_set item_id="item2000 item2003 item2004"/>
<gift>
<toy name="Lobster Souvenir" description="an exclusive present with
logo of our restaurant"/>
</gift>
</set>

```

```

        <set price="&sgd; 30.8">
            <item_set item_id="item2000 item2001 item2003 item2006"/>
            <gift>
                <voucher id="voucher2200" value="&sgd; 8" expired="one month"/>
            </gift>
        </set>
        <set price="&sgd; 25.8">
            <item_set item_id="item2000 item2005 item2007"/>
        </set>
    </menu>
</branch>

<!-- this is third branch of No Name Restaurant -->
<branch branch_id="branch003" address="125 Changi Road" city="Singapore" hotline="66106977" rating="1">
    <picture title="restaurant logo representation"
source="www.comp.nus.edu.sg/~tbthanh/noname/logo.gif" type="image-gif"/>
    <picture title="branch logo representation"
source="www.comp.nus.edu.sg/~tbthanh/noname/30/logo.gif" type="image-gif"/>
    <menu>
        <item item_id="item3000">
            <name>Lobster Egg Sandwich</name>
            <type course = "Appetizer"/>
            <price>10.95</price>
            <description>Lobster meat and eggs are scrambled with bacon and onion and
peppers in butter</description>
            <picture title="item num 3000"
source="www.comp.nus.edu.sg/~tbthanh/noname/20/item2000.png" type="image-svg"/>
        </item>
        <item item_id="item3001">
            <name>Lobster Corn Chowder</name>
            <type course = "Appetizer"/>
            <price>9.95</price>
            <description>This tasty lobster chowder recipe is made with chunks of cooked
lobster, green onions, celery, a little red pepper, corn, and seasonings</description>
        </item>
        <item item_id="item3002">
            <name>Chilled Lobster With Mimosa Dressing</name>
            <type course = "Appetizer"/>
            <price>9.50</price>
            <description>A chilled lobster first course or appetizer with sweet mimosa dressing
or sauce</description>
            <picture title="item num 2002"
source="www.comp.nus.edu.sg/~tbthanh/noname/item1002.png" type="image-svg"/>
        </item>
        <item item_id="item3003">
            <name>Lobster Bisque</name>
            <type course = "MainDish"/>
            <price>10.70</price>
            <description>Made with lobster meat, dry sherry, butter, milk, and
seasonings.</description>
            <picture title="item num 2003"
source="www.comp.nus.edu.sg/~tbthanh/noname/20/item2003.png" type="image-svg"/>
        </item>
        <item item_id="item3004">
            <name>Creamed Lobster</name>
            <type course = "MainDish"/>
            <price>11.50</price>
            <description>Creamed lobster recipe is made with lobster meat, Swiss cheese, and
white sauce. Serve creamed lobster over English muffins.</description>
        </item>
        <item item_id="item3005">
            <name>Chocolate Ice Cream</name>

```

```

        <type course = "Dessert"/>
        <price>5.50</price>
    </item>
    <item item_id="item3006">
        <name>Milk Shake</name>
        <type course = "Drink"/>
        <price>2.75</price>
    </item>
    <item item_id="item3007">
        <name>Coke</name>
        <type course = "Drink"/>
        <price>1.20</price>
    </item>
    <item item_id="item3008">
        <name>Creamed Lobster with White Sauce and Swiss Cheese</name>
        <type course = "MainDish"/>
        <price>7.5</price>
        <description>Creamed lobster recipe is made with a white sauce, English muffins,
Swiss cheese and bread crumbs</description>
    </item>
    <set price="&sgd; 30.5">
        <item_set item_id="item3000 item3001 item3003 item3005 item3007"/>
        <gift>
            <toy name="Lobster Souvenir" description="an exclusive present with
logo of our restaurant"/>
        </gift>
    </set>
    <set price="&sgd; 20.8">
        <item_set item_id="item3000 item3003 item3005"/>
    </set>
    <set price="&sgd; 22.8">
        <item_set item_id="item3001 item3004 item2007"/>
    </set>
</menu>
</branch>
</restaurant>

<!--this is third restaurant-->
<restaurant>
    <rname>Crescent Lodge Country</rname>
    <rtype type_of_restaurant="FamilyStyle"/>

    <!--first branch of Crescent Lodge Country-->
    <branch branch_id="branch111" address="12 West Coast Road" city="Singapore" hotline="65799777" rating="4">
        <picture title="branch logo representation" source="www.comp.nus.edu.sg/~tbthanh/cresent/logo.gif"
type="image-gif"/>
    </branch>
    <menu>
        <item item_id="item1100">
            <name>Tomato and Mozzarella Napoleon</name>
            <type course = "Appetizer"/>
            <price>9</price>
            <description>beefsteak tomatoes layered between slices of fresh mozzarella,
topped with peslo sauce.</description>
            <picture title="item num 1100"
source="www.comp.nus.edu.sg/~tbthanh/cresent/item1100.png" type="image-svg"/>
        </item>
        <item item_id="item1101">
            <name>BBQ Duck</name>
            <type course = "Appetizer"/>
            <price>10</price>
            <description>bacon wrapped BBQ duck breast, served over a bed of sauteed
mustard cabbage.</description>
        </item>
    </menu>
</branch>
</restaurant>

```

```

        <picture title="item num 1101"
source="www.comp.nus.edu.sg/~tbthanh/cresent/item1101.png" type="image-svg"/>
    </item>
    <item item_id="item1102">
        <name>Fried Brie</name>
        <type course = "Appetizer"/>
        <price>8</price>
        <description>served with a raspberry horseradish sauce</description>
        <picture title="item num 1102"
source="www.comp.nus.edu.sg/~tbthanh/cresent/item1102.png" type="image-svg"/>
    </item>
    <item item_id="item1103">
        <name>Sweet Tapioca</name>
        <type course = "Dessert"/>
        <price>5.95</price>
        <description>Steamed Sweet Tapioca with Coconut Creame</description>
    </item>
    <item item_id="item1104">
        <name>Coke</name>
        <type course = "Drink"/>
        <price>1.95</price>
    </item>
    <item item_id="item1105">
        <name>Jumbo Shrimp and Crab</name>
        <type course = "Appetizer"/>
        <price>12 </price>
        <description>chilled jumbo shrimp and colossal crab served with our zesty Cocktail
sauce</description>
    </item>
    <item item_id="item1106">
        <name>Vegetarian sautee</name>
        <type course = "MainDish"/>
        <price>16 </price>
        <description>darden hardened vegetables sauteed with gralic, gingger and soy
over pasta</description>
    </item>
    <item item_id="item1107">
        <name>Roast Duckling</name>
        <type course = "MainDish"/>
        <price>14 </price>
        <description>crisp half long island duckling served with orange grand mamier
sauce</description>
    </item>
    <item item_id="item1108">
        <name>Chicken Parmesan</name>
        <type course = "MainDish"/>
        <price>16 </price>
        <description>breaded chicken cutlet with a tomato basil sauce and melted
mozzarella</description>
    </item>
    <item item_id="item1109">
        <name>Cocktail</name>
        <type course = "Appetizer"/>
        <price>5 </price>
    </item>
    <item item_id="item1200">
        <name>Milkshake</name>
        <type course = "Drink"/>
        <price>4 </price>
    </item>
</menu>
</branch>

```

```

</restaurant>

<!-- This is the fourth restaurant -->
<restaurant>
    <rname>Nanyang Watermark Lumenouis Outlet</rname>

    <!-- This Restaurant does not have Restaurant Type specification -->
    <rtype></rtype>

    <!-- The first branch -->
    <branch address="01 Nanyang Boulevard" branch_id="branch1" hotline="99997472" city="Singapore" rating="3">

        <!-- This branch does not have picture -->
        <menu>
            <item item_id="item01">
                <name>Live Baby Octopus Karatomenis Hotsauce</name>
                <type course="Appetizer"></type>
                <price>13</price>
                <description>Live baby octopus half-cooked in traditional karatomenis
hotsauce</description>

                <picture source="www.facebook.com/nanyangwatermark/liveoctopus.jpg"
type="image-svg"/>
            </item>
            <item item_id="item02">
                <name>Spanakopita Triangles</name>
                <type course="MainDish"></type>
                <price>17</price>
                <description>Secret recipe on making triangle wonton-like rice paper, rolled with
spanakopita lamp meat</description>

                <!-- This item does not contain picture -->
            </item>
            <item item_id="item03">
                <name>Honey Mustard Chicken</name>
                <type course="MainDish"></type>
                <price>24</price>
                <description>Red Skin Mashed Potatoes with garlic and Petite Green Peas

                <picture source="www.facebook.com/nanyangwatermark/honeymustard.jpg"
type="image-svg"/>
            </item>
            <item item_id="item04">
                <name>Milkshake</name>
                <type course="Drink"></type>
                <price>5</price>
            </item>
            <item item_id="item05">
                <name>Cocktail</name>
                <type course="Appetizer"></type>
                <price>7</price>
            </item>
        </menu>
    </branch>

    <!-- This is second branch -->
    <branch address="56 Keppel Highway" branch_id="branch2" hotline="99992637" rating="4">
        <picture source="www.photobucket.com/56keppel.bmp" type="image-gif"/>
        <menu>
            <item item_id="item1">
                <name>Easy Mixed Skillset Lasagna</name>
                <type course="MainDish"></type>
                <price>16.99</price>
                <description>Mixed beef and lasagna salads in onion steam pot</description>

```

```

<picture source="www.photobucket.com/56keppel/lasagna.jpg" type="image-
svg"/>

</item>
<item item_id="item2">
  <name>Lamp Chops</name>
  <type course="MainDish"></type>
  <price>26.99</price>
  <description>Bone in grilled lamb chops served with creamy artichoke and mint
sauce, roasted nuts, chilli and fresh mint</description>
</item>
<item item_id="item3">
  <name>Bresaola rainbow beef salad</name>
  <type course="Appetizer"></type>
  <price>15.99</price>
  <description>Finely sliced beef with roasted beetroot, rocket, fennel, horseradish,
Parmesan, balsamic and extra virgin olive oil</description>
  <picture source="www.photobucket.com/56keppel/bresaola.jpg" type="image-
gif"/>
</item>
<item item_id="item4">
  <name>Fish Baked In a Bag</name>
  <type course="MainDish"></type>
  <price>19.99</price>
  <description>Fantastic sustainable fish of the day from Devon and Cornwall. Baked
in its own juices with clams, mussels, smashed fennel, chilli and anchovy. Served with Sicilian scented cracked wheat</description>
</item>
<item item_id="item5">
  <name>Chocolate Creammies</name>
  <type course="Dessert"></type>
  <price>8.99</price>
  <description>Chocalate pie baked with delicious ice-cream on top</description>
</item>
<item item_id="item6">
  <name>Milkshake</name>
  <type course="Drink"></type>
  <price>8.99</price>
</item>
<set price="&sgd;49.99">
  <item_set item_id="item3 item4 item5 item6"></item_set>
  <gift>
    <toy name="Zurich Marco Lego"></toy>
  </gift>
</set>
<set price="&sgd;59.99">
  <item_set item_id="item1 item2 item5 item6"></item_set>
  <gift>
    <voucher value="&sgd;" id="voucher109"/>
  </gift>
</set>
</menu>
</branch>
</restaurant>

<!-- This is the fifth restaurant -->
<restaurant>
  <rname>Tamil Srika Haani Food Republic</rname>
  <rtype type_of_restaurant="FineDining" type_of_cuisine="Indian Delights"/>
  <branch address="28 Prince George's Park Avenue 3" branch_id="branch101" hotline="67897365" rating="5">
    <menu>
      <item item_id="item001">
        <name>Bhindhi Jaipuri</name>
        <type course="Appetizer"></type>

```

```

        <price>7.50</price>
        <description>Cauliflower with potatoes cooked together with onions, spices and
tomatoes</description>
        <picture source="www.pictureyours.net/haani/haani.jpg" type="image-gif"/>
    </item>
    <item item_id="item002">
        <name>Onion Pakora</name>
        <type course="Appetizer"></type>
        <price>5.30</price>
        <description>Sliced onion with green chilli coated in chick pea flour crisp
fried</description>
    </item>
    <item item_id="item003">
        <name>Attukal Soup</name>
        <type course="MainDish"></type>
        <price>11.90</price>
        <description>A spicy soup of bone of leg of lamb simmered with peppercorn and
coriander</description>
        <picture source="www.pictureyours.net/haani/attukal" type="image-svg"/>
    </item>
    <item item_id="item004">
        <name>Chicken Rasam</name>
        <type course="MainDish"></type>
        <price>14.70</price>
        <description>A spicy soup of chicken with bones simmered with peppercorn,
coriander and mint</description>
    </item>
    <item item_id="item005">
        <name>Masala Dosai</name>
        <type course="Dessert"></type>
        <price>6</price>
        <description>Crepes filled in spiced potatoes and onions</description>
    </item>
    <item item_id="item006">
        <name>Kali Mirch Kebab</name>
        <type course="Dessert"></type>
        <price>7</price>
        <description>Bones chicken pieces marinated in yoghurt, fresh herbs and
peppercorns masala. Served with salad and sauce.</description>
    </item>
    <item item_id="item007">
        <name>Avocado Milksake</name>
        <type course="Drink"></type>
        <price>3.50</price>
    </item>
    <item item_id="item008">
        <name>Cocktail</name>
        <type course="Drink"></type>
        <price>4.20</price>
    </item>
    <set price="&sgd;25.99">
        <item_set item_id="item001 item003 item005 item007"/>
        <gift>
            <toy name="Brabasa Plastic Fish"></toy>
        </gift>
    </set>
    <set price="&sgd;29.99">
        <item_set item_id="item002 item004 item006 item008"></item_set>
    </set>
    <set price="23.50">
        <item_set item_id="item001 item003 item008"></item_set>
        <gift>

```



```

                                <voucher value="&sgd;4" id="voucher123"/>
                            </gift>
                        </set>
                    </menu>
                </branch>

<!-- This is the second branch -->
<branch address="204 Anderson Highway" branch_id="branch102" hotline="67896329" rating="5">
    <picture source="www.pixhub.net/anderson/main.jpg" type="image-gif"/>
    <menu>
        <item item_id="item11">
            <name>Finger Fish</name>
            <type course="Appetizer"></type>
            <price>3.90</price>
            <description>Deep fried crumb fish finger, served with tartar sauce</description>
        </item>
        <item item_id="item22">
            <name>Ginger Fish</name>
            <type course="Appetizer"></type>
            <price>5</price>
            <description>Cubes of fish in a ginger flavored sauce</description>
            <picture source="www.pixhub.net/anderson/cube.jpg" type="image-svg"/>
        </item>
        <item item_id="item33">
            <name>Chicken Tikka Salad</name>
            <type course="MainDish"></type>
            <price>8</price>
            <description>Sliced boneless chicken tikka with chopped onion and mint
leaves</description>
            <picture source="www.pixhub.net/anderson/tikka.jpg" type="image-gif"/>
        </item>
        <item item_id="item44">
            <name>Crème Soup</name>
            <type course="MainDish"></type>
            <price>5</price>
            <description>Tomato, Chicken, Mushroom or Vegetable</description>
        </item>
        <item item_id="item55">
            <name>Fish Coriander Goa Style</name>
            <type course="MainDish"></type>
            <price>7</price>
            <description>Fresh seasonal fish cooked in coriander leaf and onion in delicious
mild gravy</description>
        </item>
        <item item_id="item66">
            <name>Prawn Karobary Goa Style</name>
            <type course="MainDish"></type>
            <price>6</price>
            <description>Succulent prawns flavored with makhan masala and onion
gravy</description>
            <picture source="www.pixhub.net/anderson/goa.jpg" type="image-svg"/>
        </item>
        <item item_id="item77">
            <name>Egg Masala</name>
            <type course="Dessert"></type>
            <price>4</price>
        </item>
        <item item_id="item88">
            <name>Kadhai Paneer</name>
            <type course="Dessert"></type>
            <price>7</price>
            <description>pieces of home made cheese, marinated and sauteed with capsicum

```

in a Kadai</description>

```
</item>
<item item_id="item99">
  <name>Avocado Milksake</name>
  <type course="Drink"></type>
  <price>2.50</price>
</item>
<set price="&sgd;15">
  <item_set item_id="item11 item22 item44 item99"></item_set>
  <gift>
    <toy name="Barbie dolls"></toy>
  </gift>
</set>
<set price="&sgd;15">
  <item_set item_id="item66 item77 item99"></item_set>
  <gift>
    <voucher value="&sgd;7" id="voucher164" expired="3 months"/>
  </gift>
</set>
</menu>
</branch>

<!-- This is the third branch -->
<branch address="46 Orchard Boulevard" branch_id="branch823" hotline="67892635" rating="3">
  <menu>
    <item item_id="items1">
      <name>Vegetable Samosa</name>
      <type course="Appetizer"></type>
      <price>5</price>
      <description>Assorted mixed vegetables in a crisp pastry</description>
      <picture source="www.grandphotos.com/orchard/samosa.jpg" type="image-svg"/>
    </item>
    <item item_id="items2">
      <name>Sweet Corn Soup</name>
      <type course="Appetizer"></type>
      <price>3</price>
    </item>
    <item item_id="items3">
      <name>Butter Chicken Masala</name>
      <type course="MainDish"></type>
      <price>6</price>
      <description>Tandoori chicken cooked in rich tomato butter gravy</description>
      <picture source="www.grandphotos.com/orchard/tandoori.jpg" type="image-gif"/>
    </item>
    <item item_id="items4">
      <name>Maa Dee Jinga</name>
      <type course="MainDish"></type>
      <price>8</price>
      <description>Fresh prawns cooked in rich tomato onion gravy</description>
    </item>
    <item item_id="items5">
      <name>Coke</name>
      <type course="Drink"></type>
      <price>1</price>
    </item>
    <item item_id="items6">
      <name>Mineral Water</name>
      <type course="Drink"></type>
      <price>2</price>
    </item>
    <item item_id="items7">
      <name>Navrathana Korma</name>
```

```

        <type course="Dessert"></type>
        <price>4</price>
        <picture source="www.grandphotos.com/orchard/korma.jpg" type="image-svg"/>
    </item>
    <item item_id="items8">
        <name>Vegetable Khajana</name>
        <type course="Dessert"></type>
        <price>6</price>
        <description>Classic north Indian dish with bell peppers, vegetables and cashew
nuts cooked in masala</description>
    </item>
    <set price="&sgd;14">
        <item_set item_id="items1 items2 items3 items5"></item_set>
        <gift>
            <voucher value="&sgd;6" id="voucher15"/>
        </gift>
    </set>
    <set price="&sgd;25">
        <item_set item_id="items1 items2 items3 items4 items7 "></item_set>
        <gift>
            <toy name="Colorado Plastic Monkey"></toy>
        </gift>
    </set>
    <set price="&sgd;19">
        <item_set item_id="items2 items4 items6 items8"></item_set>
    </set>
</menu>
</branch>
</restaurant>
</restaurant_collection>

```