Main: HomePage

Vorlesungsnotizen Markupsprachen und SSD SS2008

1 XML Basics

2 Data Type Specifications

3 Standard Query and Transformation Languages for XML

4 Indexing XML Data (Hinweis: Dieses Kapitel ist zur Ergänzung und wird in der Vorlesung nicht behandelt)

5 Xcerpt: A Query and Transformation Language for Web and Semantic Web Applications

6 The Semantic Web, RDF

7 RDF Query Languages

Für die Vorlesung im SoSe 2008 wurde das (ursprünglich als HTML vorliegende) Skript in die Wiki-Syntax von **PmWiki** konvertiert. In einigen Fällen kann es dabei zu Fehlern in der automatischen Umwandlung gekommen sein. Für Hinweise auf solche und andere Fehler oder eigenständige Korrekturen sind wir natürlich sehr dankbar.

Main: 1XMLBasics

XML and Databases: 1 XML Basics

1 Markup Languages: Origins and Typology

1.1 "Mark up" vs. "Markup"

"Mark up" (in two words): typography annotation.
"Markup": formatting and/or structure instructions used in word processing languages.

1.2 Purposes of Markup Languages

Not all markup languages have the same purposes. Markup languages have been defined for:

- **formatting texts.**
  Such markup languages have formatting or printing instructions such as "bold", "centered", "justify", etc.
- **representing the logical (or semantical) structure of texts (or documents).**
  Such markup languages might have instructions for different kinds of titles. Such markup language are used by text processors.
- **data interchange.**
  Such markup languages are used for the interchange of documents on the Internet and on the WWW, between database management systems, between text processors, etc.

1.3 Kinds of Markup Languages

One distinguishes between:

- **Specific or layout-oriented markup languages**
- **Generalized or structure-oriented markup languages**
- **Generic markup languages or "meta markup languages"**

1.3.1 Specific or layout-oriented markup languages

E.g. PostScript (from Adobe). Layout-oriented markup languages usually have instructions of three kinds:

- instructions changing the appearance of the character immediately preceding or following it.
- instructions changing the appearance of all characters up till an associated end instruction.
- instructions changing the appearance of the page they occur in or following them.

Often, well-formedness conditions are not or insufficiently specified and/or required. E.g. a sequence such as

(italics)aaaa(bold)oooo(end-italics)cccc(end-bold)

might be allowed and might yield cccc in bold, or not in bold. As a consequence, loss of information is often

observed when the layout-oriented markup of a text is translated from one layout-oriented markup language to another.

1.3.2 Generalized or structure-oriented markup languages

Generalized or structure-oriented markup languages aim at specifying only the logical (or semantical) structure of documents (or texts). The idea is that a layout for a document should be derived from the document structure. E.g. a document might specify "sections", "section titles" and "paragraphs". How these are to be rendered on the screen or on paper is not specified within the document but somewhere else -- in a so-called "style sheet", a denomination suggested by the menus offered by many text processors for choosing layout options.

The denomination "generalized markup" refers to using names for markup elements which identify text objects in a "descriptive" or "generic" manner.

The separation of document structure from layout has been first proposed in a research project at IBM at the end of the 60es. The language DCF GML (Document Composition Facility Generalized Markup Language) **[GMP70]** has first realized this separation. The language DCF GML offers:

- Concepts such as "chapter" and "paragraph" for a specification of a logical document structure.
- Instructions for the specification of document parts to be automatically numbered like chapters and list items, or to be automatically referred to like literature references.
- A layout (or formatting) of document parts like "chapter" and "paragraph" giving rise to some limited choices (e.g. of fonts and of number styles like Arabic or Roman).

It is worth mentioning that with DCF GML both, the document structure and the interpretation (or translation) of this structure in some layout were fixed and could not be modified.

Markup languages like RTF (standardized by Microsoft and supported by most word processors including Microsoft Word) and LaTeX combine aspects of specific and generalised markup languages. They have for example markup for structure notions such as titles, but also for layout such as bold.

1.3.3 Generic markup languages or "meta markup languages"

A "meta markup language" is a language which is used to specify layout-oriented or structure-oriented markup languages. Thus, a "meta markup language" does not specify any markup itself, but instead is a means for defining markup, i.e. markup tags (like e.g. in HTML <b>, bold, and <li>, list item) as well as the structure (e.g. in HTML, the contents of an unordered list consists of zero, one or more list items).

For specifying markups and structures, "meta markup languages" rely either

- on (context-free) grammars (cf. the DTD (Document Type Definitions) of SGML and XML), or
- on type definitions similar to the algebraic data type definitions as in programming languages like ML and Haskell (cf. XML Schema **[XMLSch0]**, **[XMLSch1]**, **[XMLSch2]**).

Currently, SGML **[SGML]** and XML **[XML]** are the only "meta markup languages".

Historically, SGML goes back to DCF GML as follows: The ANSI (American National Standard Institute) first specified a generic markup language inspired from DCF GML. Later, this generic markup language was refined and resulted in SGML, a standard of the ISO (International Standard Organization). The following remark on the origins of SGML is worth citing:

> It appears certain to me that at least these three ideas were common already in the 1960's, often within distinct communities which rarely talked to each other: (a) the notion of separating "content and structure" encoding from specifications for [print] processing; (b) the notion of using names for markup elements which identified text objects "descriptively" or "generically"; (c) the notion of using a (formal) grammar to model structural relationships between encoded text objects.

Robin Cover, **[SGML1]**

XML **[XML]** has been derived from SGML **[SGML]** so as to fulfill the following goals:

- To be simpler than SGML (in particular no "tag minimization", cf. infra).
- To be used not only for modeling documents (or texts) but also data items (as found e.g. in databases)
- To make documents (or data items) without grammar (or data schema) possible.
- To have a Hypertext model.

Note that the SGML standard does not include any Hypertext model. Hypertext models can be specified for SGML, and this has been done, but such Hypertext models are not part of the SGML standard.

In the "SGML and XML world", the syntax of the markup of a generalized markup language L (specified using the "meta markup language" SGML or XML) is called "abstract syntax", the structure of a document specified in the language L is called "concrete syntax". This terminology might be misleading, for in compilers, "abstract syntax" denotes the tree structure obtained from parsing a program (or document), and "concrete syntax" the linear syntax in which programs (or documents) are written.

2 Structure of an XML Document

The denomination "XML document" or short "document" is used for any kind of XML structured data item, regardless of its contents. Thus, an XML document might deliver data of any kind such as textual data, numerical data, molecular biology data, astronomy data, etc.

An XML document consists of:

- A "document prolog" (at the beginning of the document)
- The (linear) specification of a "tree of elements", the so-called "document tree" or "document instance" (following the document prolog)
- Optional "processing instructions" (that might occur in the prolog and in the element tree specification)

The purpose of processing instructions is to provide processing software, so-called "application software", with application specific indications. Thus, the form and meaning of processing instructions cannot be specified within the XML standard.

2.1 Document Prolog

An XML document prolog consists of:

- A mandatory "XML declaration" stating the XML version and the character set used in the XML document
- Optional processing instructions
- An optional declaration of a DTD (Document Type Definition)

The formalism XML Schema improves over DTD in many ways (cf. Chapter 2 "Data Type Specification "). XML Schema aims at replacing DTD. However, both in version 1.0 (**XML10**)and version 1.1 (**XML**) of XML a DTD but no schema (after XML Schema) might be specified in an XML document prolog.

Example:

```
<?xml version="1.1" encoding="ISO-8859-15" standalone="yes"?>
<!DOCTYPE Name [Declarations] >
```

This XML document prolog refers to the version 1.1, i.e. the current version, of the XML standard and to the character set "ISO-8859-15" **[ISO8859]**, i.e. the "Western, West European character set" (extended with the Euro symbol €) convenient for European languages like German, French, Italian and British English with their accentuated characters. Cf. Section **2.6 Character Sets** for more detail on characters sets.

The attribute-value pair standalone="yes" means that the XML document does not refer to any external file such as a DTD.

The DTD is given as "Declaration" with a syntax explained below. "Name" stands for both, the outermost tag of the document, i.e. the start symbol of the grammar specified as "Declaration", and the name of the Document Type. (Such a usage of the same denomination for two distinct notions might be criticized). The syntax of a DTD is explained in Chapter 2.

Example:

```
<?xml version="1.1" encoding="ISO-8859-15" standalone="no"?>
<!DOCTYPE Name SYSTEM "File-Name" >
```

In this second example, the DTD of the XML document is assumed to be given in a file referred to (with the operating system considered) as "File-Name": "File-Name" can stand for a file name, or more generally for a path identifying a file (in the syntax of the operating system used), or, more generally, for a URI (or "Uniform resource Identifier", cf. infra).

Example:

```
<?xml version="1.1" encoding="ISO-8859-15" standalone="no"?>
<!DOCTYPE Name SYSTEM "File-Name" [Declarations] >
```

In this third example, the DTD of the XML document is assumed to be specified in both, an external file referred to as "File-Name" and internally with the value of "Declarations". If both, the external and internal data type definitions contain a definition for the same notion (e.g. two grammar rules for the same element name), then the internal definition is the one to be considered. Thus, an internal DTD is a means to modify an external DTD locally (within an XML document).

2.2 Elements

An element consists of:

- an opening tag like e.g. <email> or <email type="professional">
- a content like e.g. francois.bry@example.com, any possible text, or a sequence of texts and elements
- a closing tag like e.g. </email>

An "empty element" is an element the content of which is an empty string like e.g. the following element <a>. Such an empty element can also be denoted <a/>. (Note that most XHTML processor do not correctly interprete, say,
</br> or
 but only
.)

The string occurring in a (opening or closing) tag, also called "tag name" or "tag label", must start with an alphabetical character or with underscore character "_" except the string "xml" (which is reserved for a special usage). It might include any alphanumerical character, the underscore character "_", the minus sign "-", the dot ".", and the colon ":". Note that in tags, ":" is reserved for a special usage (cf. infra "namespaces").

The content of an element might be empty, a string, another element, or one or more strings and/or elements. Thus, XML elements are **well-formed** in the sense of terms or programming language blocs. E.g. a sequence as the following is not well-formed, for the closing tag precedes the closing tag :

Example:

```
<a> <b>     </a>   </b>
```

The ordering of elements and/or strings in the content of an element is relevant. This is reasonable if texts are to be modeled. If data items are modeled, this might be too stringent. Uses of XML in databases often assume that this ordering is irrelevant or extend XML with the possibility to specify whether the ordering of an element content is

relevant or not.

Under the name of "CDATA Section" XML provides with escape facility for blocks of text containing characters which would otherwise be recognized as markup. The string to be "escaped" is to be included between

```
<![CDATA[
```

and

```
]]>"
```

E.g. the following string:

```
<![CDATA[<first-name>François</first-name>]]>
```

contains no markup.

2.3 Document Tree

After the document prolog, an XML document contains a single element (the content of which in general contains further elements). If an element A is seen as a node and an element or a string B is part of the content of A, A and B form a parent-child relation. Therefore an XML document specifies a tree, wich is called "document tree associated with the XML document".

The single element of an XML document which specifies a document tree is often referred to as the "document instance". This denomination comes from the fact that, often but not always, this element is an instance of the structure specified in the DTD given or referred to in the document prolog. Note that an XML document might have no DTD and the document tree (or document instance) of an XML document with a DTD does not have to conform to that DTD.

Concerning XML document trees, three remarks deserve to be made:

- Some nodes of an XML document tree are not elements but strings.
- An XML document specifies a (document) tree, but is itself not a tree. After its prolog, an XML document consists in a (well-formed, linear) sequence of (opening and closing) tags.
- The document tree associated with an XML document is obtained by parsing the (linear) XML document.

Nonetheless, one commonly refers to the "document tree" for denoting the (linear) part of an XML document following the document prolog.

2.4 Attributes

Pairs of the form

Attribute-Name = "Attribute-Value"

or of the form

Attribute-Name = 'Attribute-Value$_1$ ... Attribute-Value$_n$'

can be attached to an element as shown in the following example. Quotation marks " and apostrophes can be used indifferently (but consistently) for enclosing attribute values: "d'accord", "text '123' aaa", and 'text "123" aaa' are possible but not 'text "123' aaa". In "text '123' aaa", '123' is an attribute value (and the single quotes are part of that value). Similarly, in 'text "123" aaa', "123" is an attribute value (and the double quotes are part of that value). The attribute name-value(s) pairs of an element occur within the opening tag of this element. In case several attribute values are given, they are separated by blank characters and their order is irrelevant.

Example:

```
<address-book>
   <person friend='yes' xml:lang="en de fr">
        <last-name>...</last-name>
        <first-name>...</first-name>
        <phone-number type="home">
           ...
        </phone-number>
        <phone-number type="office">
           ...
        </phone-number>
        <phone-number type='cellular'>
           ...
        </phone-number>
   </person>

   ...

</address-book>
```

Attribute names obey the same syntax rules as tag names: They must start with an alphabetical character or with underscore character "_" except the string "xml" (which is reserved for a special usage). They might include any alphanumerical character, the underscore character "_", the minus sign "-", the dot ".", and the colon ":". The character ":" is reserved for a special usage (cf. infra "namespaces").

An attribute value is a string enclosed in " and " or in ' and ' (an opening " and a closing ' or the other way around is not allowed) which cannot include the characters < and &. An attribute value enclosed in " (', resp.) might include some ' (", resp.)

The ordering of the attribute name-value pairs is irrelevant, thus the following opening tags can be used indifferently:

```
<person  friend='yes'  xml:lang="en de fr">

<person  xml:lang="en de fr"  friend='yes'>
```

Except that they are not ordered, attributes might be seen as flat elements. For this reason, they have been often criticized as follows: Attributes do not contribute to make XML more expressive, but they make it more complicated. Whether to choose an attribute or an element in modeling data is a matter of taste.

XML has two special attribute types "identifier" denoted ID and "identifier reference" denoted IDREF. An attribute of type "identifier reference" is used for pointing to an element having an attribute of type "identifier". Assuming that the attribute "person-ref" is of type IDREF (identifier reference) and that the attribute key is of type REF (reference), the following example shows how the (empty) friend subelement of the first person element points to the second person element:

Example:

```
<person-list>
  <person key="bry">
    <first-name>François</first-name>
    <last-name>Bry</last-name>
    <friend person-ref="schaffert"/>
  </person>
  <person key="schaffert" >
    <first-name>Sebastian</first-name>
    <nickname>Wastl</nickname>
    <last-name>Schaffert</last-name>
     <friend person-ref="bry"/>
    <friend person-ref="schaffert"/>
  </person>
</person-list>
```

Attributes of type ID (identifier) or IDREF (identifier reference) are declared in a DTD or in a schema (cf. Chapter 2 Data Type Specification).

Note that XML offers several reference techniques:

- ID (identifier) and IDREF (identifier reference) attributes for references within a same XML document
- URI (cf. Section **2.5 Resources, Entities and Notations**) in Hypertext links for references within a same XML document or between different XML documents as well as for pointing to non-XML data
- Macro-expressions called "general entities" (cf. infra Chapter 2) that can be used for references to portions of text within an XML document
- Macro-expressions called "parameters entities" (cf. infra Chapter 2) that can be used for references to portions of text as well as file specifications within the document type definition of an XML document
- Links after the standard XLink (cf. **[XLink]**) that are not necessarily Hypertext links.

This multiplicity of similar, yet different referencing techniques may be seen as a drawback of XML.

Using ID (identifier) and IDREF (identifier reference) attributes, URI, or Links one can specify rooted graphs that are no trees, more precisely, (document) trees specifying non-tree, rooted graphs.

2.5 Resources, Entities, and Notations

2.5.1 Resources and Resource Identifiers

In XML, a "resource" is a data item or a data providing service which can be located in a non-ambiguous manner by an identifier, e.g. (but not only) a "Uniform Resource Identifier (URI)". Note that this definition is circular: A resource is what has a (resource) identifier and a (resource) identifier is something that uniquely characterizes a resource.

An example of resource identifiers are ISBN numbers. ISBN numbers are unique machine-readable identification numbers, which mark any book unmistakably. ISBN numbers have been introduced in the 70es. As of 2002, about 160 countries and territories are officially using ISBN numbers and are member in the ISBN organization **[ISBN]**.

Another example of resource identifers are the official numbers of ISO norms **[ISO]** such as the above-mentioned norm ISO8859 **[ISO8859]**.

On the Internet, resources can be identified with "URL" (Uniform Resource Locators), URN (Uniform Resource Names), and URI (Uniform Resource Identifiers), cf. **[NA]** for an overview.

URLs uniquely identify files or services accessible through an Internet protocol such as gopher (in the past), http, ftp, mailto, etc. A URL starts with a so-called "scheme" such as gopher:, http:, ftp:, or mailto:. Examples of URL are:

```
http://www.pms.ifi.lmu.de
mailto:francois.bry@example.com
```

Within the present XHTML page, the literature reference **[ISBN]** is identified with the URL #ISBN which is specified as follows:

```
<a name="ISBN" id="ISBN">[ISBN]</a>
```

The norms RFC 1738 and 1808 (cf. **[NA]**) specify the syntax of URLs.

Nowadays, URL are still informally referred to, but they are no longer referred to in formal technical documentations. Instead of URL, technical documentation nowadays refer to URI, cf. **[NA]**. Practically, one can see a URI as a URL or URN. The syntax of URI is specified in RFC 2396 (cf. **[NA]**).

A URN is a URI that has an institutional commitment to persistence and availability. A URN should be intended as a persistent, location-independent, resource identifier. A URN begins with the scheme "urn:", specified by RFC2141 (cf. **[NA]**).

A further kind of resource identifiers are the FPI (Formal Public Identifiers) inherited from SGML. FPI provide with an indirection: Instead of a location, a FPI specifies a resource that can be obtained from several locations. Thus, if the resource is not obtainable from one location, another location can be used. Note that not all XML applications (i.e. software) are capable to handle FPI. In XML, a FPI is defined in a DTD. This is addressed in Chapter 2.

2.5.2 Entities

An "entity" is a physical object such as a file, a text, a picture, etc. Referring to an entity in an XML document makes it possible to include it in the document.

In order to include in the content of an element characters such as < and > that are used in the markup, XML provides with "predefined entities":

code	character	name
<	<	lighter
>	>	greater
&	&	ampersand
'	'	apostrophe
"	"	quotation mark
€	€	Euro currency sign

"Internal entities" can be defined in a DTD (cf. Chapter 2). An internal entity denotes a portion of text. Internal entities are useful as shorthand notations, or for easing adapting documents. E.g. with an internal entity defined in the DTD of a renting contract, one can in one single place specifying the name of the tenant. All references to the internal entity in the renting contract are resolved in the name of the tenant. How "internal entities" are defined is explained in Chapter 2.

"External entities" serve two purposes:

- Splitting large XML documents into smaller parts
- Including XML data, i.e. element content and/or elements, into an XML document

External entities are localized (or addressed) by URI or FPI. Like internal entities, external entities are specified in the DTD or schema of an XML document. This is explained in Chapter 2.

The name "external entity" is often restricted to XML data. "Non-XML external entities" are often (but improperly) called "notations" after the keyword used in the declaration binding a name to such a resource (cf. infra Chapter 2, Section 4.2.2 Attribute-List Declarations).

2.5.3 Notations

Strictly speaking, a "notation" is a name declared in a DTD or schema for denoting a non-XML external entity, such as a picture. Sometimes, the expression "notation" is used to denote non-XML external entities, i.e. the name and named object are confused.

The name notation comes from the keyword used in the declaration binding a name, the notation, to a non-XML external resource (cf. infra Chapter 2, Section 4.2.2 Attribute-List Declarations).

2.6 Character Sets

XML uses the character sets of the ISO norm ISO 10646. This norm is often confused with the similar (but not identical) norm Unicode.

In ISO 10646, a character has a name and is encoded on 4 bytes. Thus, ISO 10646 can define more than 4 thousand millions characters. As of 2002, ISO 10646 defines only about 38 thousand characters.

The following table lists a few major character sets of ISO-8859 **[ISO8859]** and their regional coverage, i.e. the Latin alphabets as well as a few non-Latin alphabets with a similar design philosophy:

ISO-8859-1	Western, West European
ISO-8859-2	Central European, East European
ISO-8859-3	South European, Esperanto
ISO-8859-4	North European
ISO-8859-5	Cyrillic
ISO-8859-6	Arabic
ISO-8859-7	Greek
ISO-8859-8	Hebrew
ISO-8859-9	Turkish
ISO-8859-10	Nordic
ISO-8859-11	Thai
ISO-8859-13	Baltic Rim
ISO-8859-14	Celtic
ISO-8859-15	Euro symbol, revision of ISO-8859-1
ISO-8859-16	Romanian

As the Western Europe Latin alphabet, called Latin-1 (ISO-8859-1) **[ISO8859]** found wide adaption in West European countries, it became apparent that some characters were missing from it and others were less needed. Latin-9 (ISO-8859-15) **[ISO8859]** was defined as successor of Latin-1, replacing some of its characters and adding others, especially the Euro symbol (or "Euro currency sign") €, which didn't exist at the time Latin-1 (ISO-8859-1) was created.

What are major character sets not mentioned in the table above? The Chinese characters (with the cultural variations used in the Republic of China (or Taiwan), in the People's Republic of China, and in Japan) as well as the characters of the two alphabets used in Japan in combination with Chinese characters and the several alphabets widely used in India.

3 XML vs. SGML

XML is both a simplification and an extension of SGML:

1. XML simplifies SGML inasmuch that XML does not allow the markup minimizing of SGML.
2. XML extends SGML with a Hypertext model.
3. XML also extends SGML because an XML document does not necessarily have a DTD, and if it has it, it does not have to be structured as specified in this DTD.

Consider the following XML element modeling in XML an excerpt of an address book:

Example:

```
<persons>
  <person id="bry">
    <title>Prof. Dr.</title>
```

```
    <first-name>François</first-name>
    <last-name>Bry</last-name>
  </person>
  <person id="eisinger">
    <title>Dr.</title>
    <first-name>Norbert</first-name>
    <last-name>Eisinger</last-name>
  </person>
  <person id="schaffert">
    <first-name>Wastl</first-name>
    <last-name>Schaffert</last-name>
  </person>
</persons>
```

This XML element can be defined identically in SGML. In addition SGML allows the following simplification of the markup called "markup minimization":

Example:

```
<persons>
  <person>
    <title>Prof. Dr.</title>
    <first-name>François</first-name>
    <last-name>Bry</last-name>
  </person>
  <person>
    Dr.</title>
    <first-name>Norbert</>
    <last-name>Eisinger
  </person>
  <>
    Dr.</title>
    Slim</first-name>
    <last-name/Abdennadher/
</persons>
```

The definition of markup minimization is rather complicated, because it cannot be expressed with a context-free grammar. Indeed, those simplifications (or "minimizations") of the markup are allowed that, thanks to the context they occur in, do not lead to ambiguities.

SGML has no (standardized) Hypertext model. This means that if SGML is used for specifying a Hypertext application, not all SGML applications (i.e. software or tools) will be capable of rendering the SGML data as Hypertext data as intended.

In contrast, a Hypertext model is part of the definition of XML. The Hypertext model of XML has been inspired from the Hypertext model HyTime **[HyTime]**. The Hypertext model of XML is known as the "XML Linking Language (XLink)" **[XLink]**. The XLink Hypertext model offers:

- Hypertext links with several origins and/or sources
- The possibility of typing Hypertext links
- The possibility of defining a Hypertext link outside the resources involved in the link
- The possibility of defining various behaviors (e.g. stretch text or new windows) when (an arc of) a link is followed.

4 XML vs. HTML

The essential difference between XML and HTML is that XML is a generic or meta markup language, while HTML merely is a specific or layout-oriented markup language. In other words, with XML the markup can be freely defined, while it is predefined in HTML; with XML the logical structure of a document (or data item) can be specified, while with HTML the rendering (or layout) of a document can be defined.

HTML has been redefined in XML as XHTML **[XHTML]**. Noticeable differences between XHTML and former

versions of HTML (such as HTML 4.01 **[HTML401]**) is the syntax of empty elements (such as
 (break)) and the systematic use of elements (such as paragraph element ...).

5 References

[GMP70]	C. F. Goldfarb, E. J. Mosher, and T. I. Peterson. An online system for integrated text processing, Proc. American Society for Information Science, 7, 147-150, 1970
[HTML401]	D. Raggett, A. Le Hors,, I. Jacobs, ed. HTML 4.01 Specification **http://www.w3.org/TR/html401**
[HyTime]	S. R. Newcomb, N. A. Kipp, V. T. Newcomb. The "HyTime" hypermedia/time-based document structuring language. Communication of the ACM, 67-83, Nov. 1991
[ISBN]	isbn.org **http://www.isbn.org/standards/home/index.asp**
[ISO]	ISO -- International Organization for Standardization **http://www.iso.ch/iso/en/ISOOnline.openerpage**
[ISO8859]	ISO/IEC 8859 Information technology -- 8-bit single-byte coded graphic character sets, International Organization for Standardization
[NA]	Naming and Addressing: URIs, URLs, ... Overview of W3C materials related to Naming and Addressing. **http://www.w3.org/Addressing/**
[SGML]	ISO 8879:1986, 1986
[SGML1]	SGML: General Introductions and Overviews. **http://xml.coverpages.org/general.html#faq**
[XHTML]	XHTML™ 1.0: The Extensible HyperText Markup Language A Reformulation of HTML 4 in XML 1.0, W3C Recommendation, 26 January 2000 **http://www.w3.org/TR/xhtml1**
[XLink]	S. DeRose, E. Maler, D. Orchard, eds. XML Linking Language (XLink) Version 1.0, W3C Recommendation, 27 June 2001 **http://www.w3.org/TR/xlink/**
[XML]	T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, J. Cowan, eds. Extensible Markup Language (XML) 1.1 (Second Edition), W3C Recommendation, 16 August 2006 **http://www.w3.org/TR/2006/REC-xml11-20060816/**
[XML10]	T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, eds. Extensible Markup Language (XML) 1.0 (Fourth Edition), W3C Recommendation, 16 August 2006 **http://www.w3.org/TR/2006/REC-xml-20060816/**
[XMLSch0]	D. C. Fallside, ed. XML Schema Part 0: Primer, W3C Recommendation, 2 May 2001 **http://www.w3.org/TR/xmlschema-0/**
[XMLSch1]	H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, eds. XML Schema Part 1: Structures, W3C Recommendation, 2 May 2001 **http://www.w3.org/TR/xmlschema-1/**
[XMLSch2]	P. V. Biron, A. Malhotra, eds. XML Schema Part 0: Primer, W3C Recommendation, 2 May 2001 **http://www.w3.org/TR/xmlschema-2/**

Main: 2DataTypeSpecifications

XML and Databases: 2 Data Type Specifications

1 Features of Standard Data Models: indispensable, exceptions excluded, and not self-explanatory

A standard database management system (DBMS) — i.e. a hierarchical, network, relational, entity-relationship, or object database management system — requires a "data model" to be specified when a database is to be managed by the system. A data model specifies the structure of the data items to be stored in the database. E.g. a data model for a relational database specifies the relation (or table) names, and for each relation, an arity, the attribute names, and the types of the attributes; a data model for an object database specifies the object types (i.e. how the objects are internally structured) and the inheritance rules (i.e. which object classes are subclasses of other object classes).

With standard DBMS, data models are indispensable. Standard DBMS cannot be used for storing data items with varying structures. In particular, exceptional data items not conforming to the data model cannot be handled by standard DBMS. This is a serious restriction, for most practical databases encounter exceptions. E.g. while a data model for a database on the employees of a company might require for every employee a family name, a first name, and a data of birth to be given, it might happen that not all of these data are available when the data on some employee are to be stored in the database.

A data item of a database managed with a standard DBMS is not self-explanatory. E.g. tuple (AF123, 1230, Munich) might denote a flight (flight number, starting time, and airport) as well a good from a catalog (coding, price, place of availability).

2 Advantages of Dispensable Data Schemas and Self-Explanatory Data

Two features of XML are especially attractive as far as database modeling is concerned:

- An XML document (i.e. data item) does not necessarily have a DTD or schema (i.e. data model), and in case it has one, its structure might depart from that specified in that DTD or schema.
- The names used for markup elements make XML documents (e.g. data items) self-explanatory.

E.g. a data item like that of the following example cannot be mistaken for the description of a good.

Example:

```
<flight>
   <flight-number>AF123</flight-number>
   <departure-time>1230</departure-time>
   <departure-airport>Munich</departure-airport>
</flight>
```

Dispensable data schemas (or data models) and self-explanatory data turn out to be especially useful in distributed and heterogeneous databases and information systems for which a centralized data management and/or data

modeling is not desirable and often impossible. Note that the Web can be seen as a distributed and heterogeneous information system.

3 Semistructured Data

"Semistructured Data" are XML-like, i.e. self-explanatory (or self-describing) data with or without schema (or data-model). The denomination "semistructured" refers to the XML-like structure conveyed by tags combined with (possibly large) unstructured portions of text.

Considering semistructured data in databases has been first proposed and investigated in a few research projects in the mid 90es. The book **[ABS]** introduces in this direction of research.

"Semistructured Expressions" are used as an abstraction of XML. This abstraction has no attributes and only one form of references (instead of the many kinds of references of XML, e.g. ID and IDREF attributes, URI and Links, etc. cf. Section 2.4 Attributes). This is no significant restriction, for attributes can be seen as "flat (unordered) elements" and one single kind of references suffices to an abstract representation of most practical applications. Semistructured expressions rely on a syntax inspired from logic and/or association lists, a data structure formerly used in artificial intelligence.

Consider the XML element of the previous example. It can be represented by the following semistructured expression:

Example:

```
flight [
   flight-number      [ "AF123"  ],
   departure-time     [ "1230"   ],
   departure-airport  [ "Munich" ]
]
```

Semistructured expressions also give rise to references as follows. Consider the following XML elements (from Section 2.4 Attributes):

Example:

```
<persons>
  <person key="bry">
    <firstname>François</firstname>
    <lastname>Bry</lastname>
    <friend person-ref="schaffert"/>
  </person>
  <person key="schaffert" >
    <firstname>Sebastian</firstname>
    <nickname>Wastl</nickname>
    <lastname>Schaffert</lastname>
    <friend person-ref="bry"/>
    <friend person-ref="schaffert"/>
  </person>
</persons>
```

The XML element of this example can represented as semistructured expressions as follows:

Example:

```
persons [
  &1 ^ person [
    first-name [ "François" ],
    last-name [ "Bry" ],
    friend [ &2 ]
  ],
  &2 ^ person [
    first-name [ "Sebastian" ],
    nickname [ "Wastl" ],
```

```
        last-name [ "Schaffert" ],
        friend [ &1 ],
        friend [ &2 ]
      ]
    ]
```

Expressions of the form &n are so-called "object identifiers" or short "oid". An occurrence of an oid such as &2 in "&2 ^ person[...]" is called an "oid definition". An occurrence of an oid such as &1 in "friend [&1]" is called an "oid reference".

Concerning oid occurrences in semistructured data, the following well-formedness conditions are necessary:

- Every oid &n referred to (i.e. occurring at some other place than left of ^) in a semistructured expression, is also defined (i.e. occurs left of ^) in this semistructured expression.
- An oid &n is defined (i.e. occurs left of ^) at most once in a semistructured expression.

It might simplify the processing of semistructured expressions to also require the following additional condition:

- Every oid &n defined (i.e. occurring left of ^) in a semistructured expression is also referred to (i.e. occurs at some other place than left of ^) at least once in this semistructured expression.

Note that like XML elements, semistructured data can be seen as trees specifying rooted graphs that are not necessarily acyclic (i.e. that are not necessarily trees).

Documents and texts require an ordering of their components. In contrast, it is often beneficial not to require an ordering of the components of (non-textual) database items, because this gives the freedom to reorder them so as to optimize their storage. Thus, it is convenient to provide semistructured expressions with two kind of sequences, ordered sequences noted "[...]" and unordered sequences "{ ... }". In the following example, the children of the flight element are unordered:

Example:

```
flight {
   flight-number      [ "AF123"  ],
   departure-time     [ "1230"   ],
   departure-airport  [ "Munich" ]
}
```

The syntax of semistructured expressions (sse) can be defined as follows using a context-free grammar:

```
            <sse> := ( "&" oid "^" )? ( label | label <list>) .
           <list> := <ordered-list> | <unordered-list> .
   <ordered-list> := "[" <sse-or-reference> ( "," <sse-or-reference> )* "]" .
 <unordered-list> := "{" <sse-or-reference> ( "," <sse-or-reference> )* "}" .
<sse-or-reference> := <sse> | """ string  """ | "&" oid .
```

In this grammar, expressions between < and > are non-terminal symbols (or variables). "[" , "{", "]" and "}" are terminal symbols. "oid" and "label" denote object identifiers and tag names, respectively. As usual in the formalization of programming languages, "string", "oid" and "label" are terminal symbols (to which values are assigned).

The well-formedness conditions on occurrences of oid are not expressed in the grammar: Indeed, these conditions cannot be expressed with a context-free grammar.

Semistructured expressions give rise to expressing tuples of relational databases. E.g. the first example of this section gives the representation as a semistructured expression of a tuple of a relation "flight" with attributes "flight number", "departure time", and "departure airport".

Semistructured expressions also give rise to expressing structured data items, shared subexpressions, and cycles

(like e.g. the self-reference within the person element with oid & of the previous example) like objects (of object databases) do. However, semistructured expressions cannot express inheritance properties. To this aim, a schema (or data model) formalism is necessary.

In the literature, a few other formalisms are used for semistructured expressions. E.g. in **[ABS]**, one-element lists are represented without parentheses: e.g. instead of first-name["François"], **[ABS]** writes first-name: "François". Also, in **[ABS]**, both oid definitions and references are similarly noted to the right of ";". Furthermore, in **[ABS]**, only unordered sequences are considered (denoted "(...)"). The formalism given above has been retained for the sake of expressivity and clarity.

4 DTD

4.1 Introduction

DTD stands for "document type definition" (not declaration). Document type declaration, DOCTYPE for short, means the `<!DOCTYPE`-entity of an XML document. Document type definition is the actual grammar that is either contained or referred to in the document type declaration. See also: **http://en.wikipedia.org /wiki/Document_Type_Declaration** and **http://en.wikipedia.org/wiki/Document_Type_Definition** . (We'll correct the text below soon to reflect this properly)

DTD stands for "document type declaration". In the definition of XML **[XML]**, this notion is defined as follows:

> An XML document type declaration contains or points to markup declarations that provide a grammar for a class of documents.

The words "or points to" refer to the possibility of specifying all or part of a DTD in an external file. In **[XML]**, "markup declarations" are defined as follows:

> A markup declaration is an element type declaration, an attribute-list declaration, an entity declaration, or a notation declaration.

4.2 Markup Declarations

4.2.1 Element Type Declarations

An element declaration has the form:

```
<!ELEMENT element-name content-model>
```

An "element name", i.e. the value of the variable element-name, is a string occurring in an opening or closing element tag. As mentioned in Chapter 1, Section 2.2 Elements,

> it must start with an alphabetical character or with underscore character "_" except the string "xml" (which is reserved for a special usage). It might include any alphanumerical character, the underscore character "_", the minus sign "-", the dot ".", and the column ":". Note that in tags, ":" is reserved for a special usage (cf. infra "namespaces").

The "content model" of an element declaration specifies what might occur between the opening and closing tag of such an element (the attributes of an element are specified in a special declaration, cf. below). The types that might be used in the "content model" of an element declaration are as follows:

- (#PCDATA)
- Children elements
- Mixed content
- Free content

These content models types are as follows:

(#PCDATA): This expression, read as "parsable character data", means text (or string) without markup. Note that a content of type (#PCDATA) is no element.

Children elements: The following formalism (inspired from grammars) is to be used:

- **(child$_1$, ..., child$_n$)** expresses an (ordered) sequence of objects of types child$_1$, ..., child$_n$, respectively.
- **(child$_1$ | ... | child$_n$)** expresses the choice of an object of type child$_1$, ..., or child$_n$.
- **child ?** expresses an option: an object of type child does not occur or occurs exactly once.
- **child *** expresses a repetition: an object of type child does not occur or occurs a finite number of times.
- **child +** expresses a repetition: an object of type child occurs a non-zero, finite number of times.

In such specifications, the variables **child** and the **child$_i$** stands for element names (each of them in turn must have a markup declaration in the DTD).

Mixed content: a combination of children elements and text (i.e. "parsable character data" or #PCDATA). The structure of a mixed content cannot be specified. If the elements involved in a mixed content are to be objects of types **child$_1$**, ..., or **child$_n$**, then the mixed content can only be specified as

(#PCDATA | child$_1$ | ... | child$_n$)*

Note that **#PCDATA** must be the first choice of the list. Thus, one cannot express e.g. that a mixed content must include, say, one object of one of the types **child$_1$** and **child$_3$**.

Free content expressed by **ANY**: Every element defined in the DTD, any combination of elements defined in the DTD, or a mixed content is a possible content.

Assume that an address book consists of zero, one, or more cards, that each card consists in the following order of exactly one person description, one or more email addresses, and possibly of a comment the content of which might be anything. Assume also that a person is described by exactly one last name, exactly one first name, zero or one nickname. Assume further that names are strings. This can be expressed by the following markup definitions:

Example:

```
<!ELEMENT address-book card* >
<!ELEMENT card (person, email+, comment?) >
<!ELEMENT person (last-name, first-name, nickname?) >
<!ELEMENT last-name (#PCDATA) >
<!ELEMENT first-name (#PCDATA) >
<!ELEMENT nickname (#PCDATA) >
<!ELEMENT email (#PCDATA) >
<!ELEMENT comment ANY >
```

Note that the formalism does not permit to express which strings are valid email addresses.

4.2.2 Attribute-List Declarations

Recall (cf. Chapter 1, Section 2.4 Attributes) that

- attributes are pairs of the form "Attribute-Name = 'Attribute-Value'" or "Attribute-Name = 'Attribute-Value$_1$ ...Attribute-Value$_n$'" which can be attached to an element,
- the ordering of the attribute name-value(s) pairs attached to an element is irrelevant.

An attribute-list declaration for an element named "element-name" has the form:

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

The variable "attribute-name" takes as value the name of an attribute. As mentioned in Chapter 1 Section 2.3, this names

> obey the same syntax rules as tag names: They must start with an alphabetical character or with underscore character "_" except the string "xml" (which is reserved for a special usage). They might include any alphanumerical character, the underscore character "_", the minus sign "-", the dot ".", and the column ":". The character ":" is reserved for a special usage (cf. infra "namespaces").

The variable "attribute-type" stands for one of:

- **CDATA**
- **ID**
- **IDREF**
- **ENTITY** or **ENTITIES**
- **NOTATION**
- **NMTOKEN** or **NMTOKENS**
- **(token$_1$ | ... | token$_n$)**

CDATA is the attribute type for attribute values that are strings.

ID is the attribute type for attribute values that are identifiers (cf. Chapter 1, Section 2.4 Attributes).

IDREF is the attribute type for attribute values that are identifier references (cf. Chapter 1, Section 2.4 Attributes)

ENTITY or **ENTITIES** are the attribute types for attribute values that are a notation or several notations. Recall that a notation is a name given (in a "notation declaration", cf. infra Section 4.2.5 Notation declarations) to a "non-XML external entity" (cf. Chapter 1, Section 2.5.3 Notations). If an attribute of type ENTITY or ENTITIES is declared in a DTD, then in this DTD one or several notations must also be declared (as possible values for the attribute of type ENTITY or ENTITIES).

NOTATION is the attribute type for an attribute binding a name to a notation (cf. Chapter 1, Section 2.5.3 Notations), i.e. a "non-XML external entity".

NMTOKEN or **NMTOKENS** (for name token(s)) are the attribute types for attribute values that are one or several tokens. An XML "token" is any string conforming to the same syntax conditions as tags and attribute names. E.g. the widespread language names "en", "fr", "de" are possible XML tokens.

(token$_1$ | ... | token$_n$) specifies an attribute type with attribute values **token$_1$**, ..., and **token$_n$**, where each of **token$_i$** is a "token", i.e. a string conforming to the same syntax conditions as tags and attribute names. In this case, a attribute value is one of the **token$_i$**.

The variable "default-declaration" is one of:

- a default value for the attribute
- **#REQUIRED** if the attribute must occur in every element
- **#IMPLIED** if the attribute is optional (sic!)
- **#FIXED Value**, where "**Value**" is an attribute value, if the attribute always takes this value.

Consider the example given above of an address book and assume that each person element is always to be attached an attribute specifying the language spoken by this person and an optional attribute specifying the country where this person lives. Assume furthermore that only the languages English, French, and German might occur. This can be specified as follows:

Example:

```
<!ELEMENT person (last-name, first-name, nickname?) >
```

```
<!ATTLIST person lang (en | fr | de) #REQUIRED
                 country CDATA #IMPLIED >
```

Note that several attribute lists might be specified for the same element. Thus, the following specification has the same meaning as the previous one:

Example:

```
<!ELEMENT person (last-name, first-name, nickname?) >
   <!ATTLIST person lang (en | fr | de) #REQUIRED >
   <!ATTLIST person country CDATA #IMPLIED >
```

4.2.3 Entity Declarations

Under the denomination of "entity declarations" the definition of macro-expressions of two kinds is meant:

- "General entities" are "internal entities" (cf. Chapter 1, Section 2.5.2), i.e. macro-expressions, to be used in the element (or document tree) of an XML document.
- "Parameter entities" are "internal entities" (cf. Chapter 1, Section 2.5.2), i.e. macro-expressions, to be used in a DTD.

An entity declaration has one of the following forms, depending on the kind "general entity" or "parameter entity" of the entity:

```
<!ENTITY General-Entity-Name "string" >
<!ENTITY % Parameter-Entity-Name1 "string" >
<!ENTITY % Parameter-Entity-Name2 SYSTEM 'uri' >
<!ENTITY % Parameter-Entity-Name2 PUBLIC 'fpi' "uri" >
```

where "string" denotes a string, "uri" a URI (cf. Chapter 1, Section 2.5.1 Resources and Resource Identifiers), and "fpi" a formal public identifier (cf. Chapter 1, Section 2.5.1 Resources and Resource Identifiers). Note that " (quotation mark) and ' (apostrophe) can be indifferently used, provided that the same symbol is used as opening and closing symbol.

For example, a general entity defining a copyright notice is as follows:

Example:

```
<!ENTITY cn "&#169; 2002" >
```

The code © denotes in the character set ISO-8859 **[ISO8859]** the character ©. Each occurrence of

&cn;

in the element of the XML document the DTD of which contains the entity declaration of the above example is expanded into

© 2002

Parameter entities are similarly used within a DTD. E.g. consider the following declaration of a parameter entity:

Example:

```
<!ENTITY % uri "http://www.pms.informatik.uni-muenchen.de" >
```

Each occurrence of

%uri;

in the DTD where the parameter entity uri is defined is expanded into

http://www.pms.informatik.uni-muenchen.de

In spite of the denomination, an entity declaration does not serve to declare a non-XML external entity. Notation declarations serve this purpose. Rather confusing is the fact that an attribute with values that are notations, i.e. names of non-XML external entities, is of type ENTITY or ENTITIES (cf. supra Section 4.2.2 Attribute-List Declarations).

4.2.4 Evaluation of General and Parameter Entities

The evaluation of general entities is lazy, i.e. a general entity is evaluated by need, while the evaluation of parameter entities is eager, i.e. all parameter entities are evaluated before the DTD is used.

The following conditions must be satisfied when general and/or parameter entities are used:

- Every general and/or parameter entity referred to must be declared (i.e. defined) in the DTD.
- Parameter entities can be referred to only in the DTD.
- General entities can be referred to only in the element of the document.
- Recursive or circular declarations of (general or parameter) entities are forbidden.
- General entities may be declared and referred to in any order. In contrast, parameter entities must be defined (in the DTD) before the are referred to (in the DTD). (The rational is that the evaluation of general entities is lazy while the evaluation of parameter entities is eager).
- A parameter entity is allowed in an element, attribute-list, or entity declaration only if this parameter entity is declared in an external part of the DTD (cf. infra Section 4.3 Composition of a DTD), i.e. not in the part where it is referred to.

4.2.5 Notation declarations

Recall that a notation is a name assigned (within a DTD in a notation declaration) to a non-XML external entity, i.e. an entity defined outside the XML document considered which is non-XML data, i.e. multimedia data such as a picture (cf. Chapter 1, Section 2.5.3 Notations).

A notation declaration binds a name to a notation, i.e. to a "non-XML external entity".

A notation declaration has the following form:

<!NOTATION name SYSTEM 'uri' >

where "uri" denotes a URI. In the XML document the "name" appears as attribute value.

Example:

```
<!NOTATION photo-of-francois-bry SYSTEM
   'http://www.pms.informatik.uni-muenchen.de/mitarbeiter/bry/photo-2001-small.gif' >
```

E.g. an empty element "person" with an attribute "picture" of type ENTITY gets assigned the picture available at the given URI if the attribute "picture" is assigned as follows the value "photo-of-francois-bry":

Example:

```
<person picture="photo-of-francois-bry"/>
```

4.3 Composition of a DTD

The DTD of an XML document can be specified as follows:

- Locally to the document, i.e. within the document prolog, or
- In one external file (which is referred to in the document prolog), or
- Both, within the document prolog and in one external file (which is referred to in the document prolog), or
- Within the document prolog as well as in several external files (which are referred to in the local part of the DTD).

When the DTD is specified from many sources, e.g. locally in the document prolog and in one or several external files, inconsistencies between different sources might occur, e.g. if a same element has different declarations in different sources. In such a case, a local declaration overrides the external declarations.

The following example shows a locally defined DTD, i.e. a DTD defined within the document prolog.

Example:

```
<?xml version="1.0" encoding="ISO-..." standalone="yes"?>
<!DOCTYPE book [
   <!ELEMENT book (chapter)+>
   <!ELEMENT chapter (#PCDATA)>
]>
<book>
  <chapter>This is a text.</chapter>
</book>
```

Note the value "yes" of the attribute "standalone" in the XML declaration. Note also the double occurrence of the name "book" in the DTD:

1. As name of the document type (to the right of the keyword DOCTYPE)
2. As name of the element of the document (to the right of the keyword ELEMENT)

XML requires that both of these names are identical.

In the following example, the same DTD is specified in an external file referred to in the document prolog. Note the value "no" of the attribute "standalone" in the XML declaration indicating that an external file (that containing the DTD) is referred to in the document):

Example:

```
<?xml version="1.0" encoding="ISO-..." standalone="no"?>
<!DOCTYPE book SYSTEM "book.dtd">
<book>
  <chapter>This is a text.</chapter>
</book>
```

The file "book.dtd" is as follows. Note that another character encoding than that of the document might be used in the file "book.dtd".

Example:

```
<?xml version="1.0" encoding="ISO-..." standalone="no"?>
<!ELEMENT book (chapter)+>
<!ELEMENT chapter (#PCDATA)>
```

Note that the file "book.dtd" contains no XML data. The fact that DTD are not specified with an XML syntax has often been criticized. This makes e.g. impossible to use an XML editor for editing a DTD, or to use an XML syntax checker for DTD.

In the following example, the same external DTD of file "book.dtd" is referred to and locally modified (an attribute

is added to the element book).

Example:

```
<?xml version="1.0" encoding="ISO-..." standalone="no"?>
<!DOCTYPE book SYSTEM "book.dtd" [
   <!ATTLIST book lang (en | fr | de) #REQUIRED>
]>
<book lang="en">
  <chapter>This is a text.</chapter>
</book>
```

In the following example, the external DTD are used. The first one, contained in the file "book.dtd", specifies the structure of a book. The second DTD, contained in the file "table.dtd", specifies the structure of tables. The third DTD, contained in the file "bib.dtd", specifies the structure of a bibliography list.

Example:

```
<?xml version="1.0" encoding="ISO-..." standalone="no"?>
<!DOCTYPE book SYSTEM "book.dtd" [
   <!ENTITY % tb SYSTEM "table.dtd">
   <!ENTITY % bb SYSTEM "bib.dtd">
   %tb;
   %bb;
]>
<book>
  <chapter>This is a text.</chapter>
</book>
```

"tb" and "bb" are parameter entities (cf. supra Section 4.2.3 Entity declarations), i.e. macro-expressions to be used in a DTD. They are declared (in the <!ENTITY % ...> expressions) within the local DTD and referred to (in the expressions %tb; and %bb;) in this same local DTD.

In the previous example, parameter entities are used for realizing a module inclusion which, as such, is not provided by the DTD formalism.

Recall that the evaluation of parameter entities is eager and therefore a parameter entity has to be declared before it is referred to.

4.4 Namespaces

XML element and attribute names are so-called qualified names (short QNames). A qualified name is either prefixed, consisting of a namespace prefix and a local name (e.g. xhtml:div) or unprefixed, consisting only of a local name (e.g. div). When namespace prefixes are used within an XML document, they must be declared.

Namespaces are a means for avoiding name conflicts, i.e. they are a means for uniquely naming elements and attributes in XML documents. Their exact usage and semantics is defined in the W3C Recommendation "Namespaces in XML".

Namespaces can also be used to differentiate between two element or attribute types declared in external DTDs. E.g. it might happen that two external DTD are used, a first one defining books with elements "fn" for footnotes, and a second one defining mathematical expressions with elements "fn" for functions.

Namespaces are used as follows:

1. In the document referring to several external DTD, a name (or prefix) is assigned to the URI of each external DTD used in this document.
2. This name is used as a prefix for the element and attribute names referred to in the document.

Consider a document referring to one internal DTD defining the structure of research report, and to two external

DTD, a first external DTD defining a book structure with elements "fn" for footnotes, and a second external DTD defining mathematical expressions with elements "fn" for functions. The following example shows how prefixes are specified and used:

Example:

```
<?xml version="1.0" encoding="ISO-..." standalone="no"?>
<!DOCTYPE research-report SYSTEM "research-report.dtd" [
   <!ENTITY % book-dtd SYSTEM "book.dtd">
   <!ENTITY % math-dtd SYSTEM "http://www.thedtdcompany.com/math/math.dtd">
   %book-dtd;
   %math-dtd;
]>
<research-report xmlns:book="book.dtd"
         xmlns:math="http://www.thedtdcompany.com/math" >
  <chapter>
     This is a text with both a
     <book:fn>footnote</book:fn>
     and a function
     <math:fn>f(x) = 2</math:fn>.
  </chapter>
</research-report>
```

The prefixes are specified as attribute-value pairs of the element they should occur in. In such a specification of a prefix "p", "p" itself prefixed with the reserved expression "xmlns" (for "XML namespace") serves as attribute name. Note the use of the (reserved) symbol ":" in prefixing.

Inheritance takes place: A prefix specified in an element might be used within descendant elements at any depth (up till a re-definition of this prefix). Thus, the following is also possible, if this is compatible with the book DTD.

Example:

```
<?xml version="1.0" encoding="ISO-..." standalone="no"?>
<!DOCTYPE research-report SYSTEM "research-report.dtd" >
<research-report xmlns:book="book.dtd"
         xmlns:math="http://www.thedtdcompany.com/math" >
  <chapter>
     This is a text with both a
     <book:fn>footnote including a function
         <math:fn>f(x) = 2</math:fn>
     </book:fn>.
  </chapter>
</research-report>
```

A prefixed name such as "math:fn" stands for "**http://www.thedtdcompany.com/math:fn**". As a (disappointing) consequence, the use of namespaces is almost incompatible with validity verification (of a document main element with respect to a DTD).

4.5 Well-Formed vs. Valid XML Documents

An XML document is said to be "well-formed" if its main element is well-formed (cf. Chapter 1, Section 2.2 Elements).

An XML document is said to be valid if it is assigned a DTD and the structure of its main element enforces the document DTD.

It is worth mentioning that XML requires documents to be well-formed, but not to be valid. This gives rise e.g. to use XML for documents without a DTD, or to depart from a DTD.

4.6 Expressive Power of DTD

Basically, a DTD is a context-free grammar, more precisely a so-called regular tree-grammar, in disguise. As a

consequence, context-dependent definitions are not possible with a DTD. E.g. one cannot specify an address book with address elements of two different kinds for persons and companies both called "address". The context-freeness of the DTD formalisms requires to use two different element names, e.g. "person-address" and "company-address". In practice, this turns out to be a serious limitation.

The DTD formalism is more stringent than that of context-free grammars, i.e. not every context-free grammar can be represented as a DTD. Indeed, in a DTD, with every non-terminal symbol, i.e. with every element name "Name", two terminal symbols <Name> and </Name> occur in the definition of the non-terminal symbol. As a consequence, a DTD does not produce the empty word. Moreover, a DTD reminds of a LL(1) grammar. If the use of the choice construct (in the specification of an element content or of attributes) is conveniently restricted in a DTD, the DTD specifies a LL(1) grammar.

LL(1) grammars have been defined as those that can be parsed with a so-called LL(1) parser, i.e. a parser working as follows:

- top-down
- one symbol look-ahead
- no backtracking

Top-down means that, beginning with the rules for the start symbol of the (context-free) grammar, the rules of the grammar are tried against the document to parse.

One symbol look-ahead means that comparing the right hand side of a grammar rule with only the first of the forthcoming symbols of the document to parse always suffices to select a grammar rule among all those for the non-terminal symbol reached so far.

No backtracking means that any choice of (an alternative in the right hand side of) a grammar rule never has to be revised.

Recall that a grammar which does not produce the empty word is a LL(1) grammar if it fulfills the following condition:

> For every rule $V \to w_1 \mid \ldots \mid w_n$ and for every distinct i and j in $\{1, \ldots, n\}$, $first(w_i)$ and $first(w_j)$ have an empty intersection.

As a consequence of DTDs being (rather simple) grammars, the validity of (the main element of an) XML document can be verified with fast parsers (reminding of LL(1) parsers) that need little memory. E.g. the Simple API for XML "SAX" **[SAX]** is based on such a parser.

The article **[BB]** is a formal investigation of the languages specified by DTDs. It shows the following:

- A language specified by a DTD basically has a unique grammar.
- Properties that are undecidable for general context-free languages are decidable for languages specified by DTD.

5 XML Schema

5.1 Introduction

XML Schema is a novel standard of the W3C published as a "recommendation" in 2001. The XML Schema standard consists of three complementary parts **[Sch1]**, **[Sch2]**, and **[Sch3]**. Instead of XML Schema, the name XSchema is sometimes used. Precursors of XML Schema were named XSD (i.e. XML Schema Definition), XSchema, and DDML (i.e. Document Description Markup Language).

XML Schema aims at data type definitions like those of programming languages such as Standard ML (or SML)

[SMLa] **[SMLb]** **[SMLc]**, or Haskell **[Hask]**.

XML Schema departs from DTD as follows:

- With XML Schema, complex data types are not defined using production rules of a context-free grammar, but instead by using a kind of patterns reflecting the data structure.
- XML Schema gives rise to local type specifications. E.g. the structure of an address element might differ depending whether this element occurs within a person or a company element.
- XML Schema provides a large set of built-in (i.e. predefined) simple types (i.e. basic, or elementary, or unstructured types) either inspired from programming languages (such as boolean, float, decimal, integer, etc.), or tailored for Web applications (such as duration, time, date, etc.).
- XML Schema makes it possible to specify new (user-defined) data types as modifications of (predefined or user-defined) types. Modifications serving to specify new types can be scope restrictions of simple types (e.g. a type "age" can be defined as an integer between 0 and 120) or extension of complex types (i.e. structured types) with additional elements and/or attributes.
- XML Schema allows to specify integrity constraints (of some restricted kind) on (user-defined) types.
- XML Schema is expressed in an XML syntax. As a consequence, XML tools such as editors, parsers, and browser can be used with schemas expressed in the XML Schema formalism.

5.2 Built-in Simple Types of XML Schema

XML Schema offers a large set of simple types inspired mostly from the basic types of SQL and Java. XML Schema distinguishes between "primitive" and "derived" built-in (or predefined) simple types.

5.2.1 Built-in Primitive Types of XML Schema

The primitive built-in types of XML Schema include:

- string (in the usual sense)
- boolean (in the usual sense; the values are true, false, 0, and 1 with the obvious meaning)
- base64Binary (Base64-encoded binary data)
- hexBinary (hex-encoded binary data)
- float ("the IEEE single-precision 32-bit floating point type"; has lexical and canonical representations)
- decimal (i.e. arbitrary precision decimal numbers; has lexical and canonical representations)
- double ("EEE double-precision 64-bit floating point type"; has lexical and canonical representations)
- anyURI (absolute or relative URI)
- QName ("represents XML qualified names" of the form (namespace name, local part))
- Notation (NOTATION attribute type of XML **[XML]**)
- duration (a duration is a 6-tuple specifying in that order a year, a month, a day, an hour, minutes, and seconds according to the Gregorian calendar; has a lexical representation (e.g. P1Y2M3DT10H30M for 1 year, 2 months, 3 days, 10 hours, and 30 minutes or -P120D for minus 120 days))
- dateTime ("represents a specific instant of time". E.g. to indicate 1:20 pm on May the 31st, 1999 for Eastern Standard Time which is 5 hours behind Coordinated Universal Time (UTC), one would write: 1999-05-31T13:20:00-05:00.)
- time ("represents an instant of time that recurs every day"; has lexical and canonical representations)
- date (Gregorian calendar date understood as a "one-day long, non-periodic instance" such as 1999-10-26; has a lexical representation)
- gYearMonth ("represents a specific Gregorian month in a specific Gregorian year"; has a lexical representation)
- gYear (Gregorian year)
- gMonthDay (recurring Gregorian date such as the 15th of August)
- gDay (recurring Gregorian date such as the 15th of every month)
- gMonth (Gregorian Month recurring every Gregorian year)

The built-in (or predefined) primitive types of XML Schema are listed and explained in **[Sch3]** under **http://www.w3.org/TR/xmlschema-2/#built-in-datatypes**.

Each built-in primitive type is assigned a number of so-called "constraining facets", short "facets", like e.g. length, minLength, maxLength, or enumeration for string or totalDigits, fractionDigits, enumeration, maxInclusive, maxExclusive, minInclusive, or minExclusive for decimal. These facets can be assigned values expressing restrictions of the type considered. This way, new types can be derived from built-in primitive types. XML Schema specifies a large number of built-in types derived in this manner from the built-in primitive types. Facets can also serve to specify user-defined types as restrictions of existing (built-in or user-defined) types.

5.2.2 Built-in Derived Types of XML Schema

A derived type is derived by restriction or by forming lists from a "base type". The built-in derived types of XML Schema include:

- normalizedString (i.e. strings without carriage return (#xD), line feed (#xA) and tab (#x9) characters; has string as base type)
- token (tokenized string, i.e. strings without line feed (#xA) and tab (#x9) characters, with no leading or trailing spaces (#x20) and with no internal sequences of two or more spaces; has normalizedString as base type)
- language (natural language identifiers as defined by **[RFC1766**], e.g. "it" for Italian; has token as base type)
- NMTOKEN (NMTOKEN attribute type of **[XML**]; has token as base type)
- NMTOKENS (list of values of type NMTOKEN; has NMTOKEN as base type)
- Name (XML name as defined in **[XML**]; has token as base type)
- NCName ("non-colonized" Name, i.e. the name following a namespace prefix; has Name as base type)
- ID (ID attribute type of **[XML**]; has NCName as base type)
- IDREF (IDREF attribute type of **[XML**]; has NCName as base type)
- IDREFS (IDREFS attribute type of **[XML**], i.e. list of IDREF attributes; has IDREF as base type)
- ENTITY (ENTITY attribute type of **[XML**]; has NCName as base type)
- ENTITIES (ENTITIES attribute type of **[XML**], i.e. list of ENTITY attributes; has ENTITY as base type)
- integer (decimal with 0 decimal digits, e.g. -1, 0, 12678967543233, +100000; has decimal as base type)
- nonPositiveInteger ("derived from integer by setting the value of maxInclusive to be 0"; has integer as base type)
- negativeInteger ("derived from nonPositiveInteger by setting the value of maxInclusive to be -1"; has nonPositiveInteger as base type)
- long ("derived from integer by setting the values of maxInclusive to be 9223372036854775807 and of minInclusive to be -9223372036854775808")
- int ("derived from long by setting the value of maxInclusive to be 2147483647 and minInclusive to be -2147483648")
- short ("derived from int by setting the value of maxInclusive to be 32767 and minInclusive to be -32768")
- byte ("derived from short by setting the value of maxInclusive to be 127 and minInclusive to be -128")
- nonNegativeInteger ("derived from integer by setting the value of minInclusive to be 0")
- unsignedLong ("derived from nonNegativeInteger by setting the value of maxInclusive to be 18446744073709551615")
- unsignedInt ("derived from unsignedLong by setting the value of maxInclusive to be 4294967295")
- unsignedShort ("derived from unsignedInt by setting the value of maxInclusive to be 65535")
- unsignedByte ("derived from unsignedShort by setting the value of maxInclusive to be 255")
- positiveInteger ("derived from nonNegativeInteger by setting the value of minInclusive to be 1")

Like the built-in primitive types, each built-in derived type of XML Schema has a number of constraining facets, short facets, using which further (application-dependent) simple types can be defined. E.g. the facets of the built-in derived types normalizedString, NCName, and Name are length, minLength, maxLength, pattern, enumeration, and whiteSpace; the built-in derived type integer has the constraining facets totalDigits, fractionDigits, pattern, whiteSpace, enumeration, maxInclusive, maxExclusive, minInclusive, and minExclusive.

5.3 Simple Type Definitions

Using the constraining facets (or facets) of the built-in (primitive or derived) types, new (application-dependent) simple types can be defined (derived from the simple type considered). The following example is a definition of a simple type "last-name".

Example:

```
<xsd:simpleType name="last-name">
   <xsd:restriction base="xsd:string">
      <xsd:length value="35"/>
   </xsd:restriction>
</xsd:simpleType>
```

In the previous as well as in the following examples, the prefix xsd is assumed to be bound to the namespace of XML Schema. This is achieved by encompassing as follows the simple type definition of the previous example in an XML document specifying a schema after XML Schema:

Example:

```
<?xml version='1.0' encoding="iso-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xml:lang="en">

   <xsd:annotation>
      <xsd:documentation>
        Example of a schema after XML Schema
        for the course "XML and Databases".
        Copyright 2002 François Bry
      </xsd:documentation>
   </xsd:annotation>

   <xsd:simpleType name="last-name">
      <xsd:restriction base="xsd:string">
         <xsd:length value="35"/>
      </xsd:restriction>
   </xsd:simpleType>

</xsd:schema>
```

The following example is the definition of a simple type age as a restriction of the built-in derived type integer:

Example:

```
<xsd:simpleType name="age">
   <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="1"/>
      <xsd:maxExclusive value="121"/>
   </xsd:restriction>
</xsd:simpleType>
```

The previous type definition has the same meaning as the following one:

Example:

```
<xsd:simpleType name="age">
   <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="1"/>
      <xsd:maxInclusive value="120"/>
   </xsd:restriction>
</xsd:simpleType>
```

The constraining facet "enumeration" can be used as follows:

Example:

```
<xsd:simpleType name="vowel">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="a"/>
      <xsd:enumeration value="e"/>
      <xsd:enumeration value="i"/>
```

```
      <xsd:enumeration value="o"/>
      <xsd:enumeration value="u"/>
   </xsd:restriction>
</xsd:simpleType>
```

The following example is the definition of a simple type room-number from the built-in primitive type string using its pattern constraining facet:

Example:

```
<xsd:simpleType name="room-number">
   <xsd:restriction base="xsd:string">
      <xsd:pattern value="[1-9][0-9]*[a-z]?"/>
   </xsd:restriction>
</xsd:simpleType>
```

5.3.3 The Type Hierarchy

Every built-in simple type of XML Schema is a subtype of a special type named anySimpleType. Every (built-in or user-defined) derived type is a subtype of its base type. Furthermore, every (built-in or user-defined) type is a subtype of a special type named anyType. Thus, every built-in type of XML Schema as well as every (user-defined) type in a schema after XML Schema occupies a position in a type hierarchy.

5.4 Complex Types Definitions

In contrast to simple types, complex types specify objects having a tree structure, i.e. consisting of XML elements themselves possibly containing further elements and/or text and possibly having attributes. A definition of a complex type consists in

- the specification of a "content model", and/or
- the specification of attributes

5.4.1 Content Model Specification

A content model can consist in

- a "sequence" of elements corresponding to the DTD construct **($child_1$, ..., $child_n$)**
- a "choice" of elements corresponding to the DTD construct **($child_1$ | ... | $child_n$)**
- a "all" grouping of elements expressing that, like in a sequence each element specified must occur exactly once, but unlike in a sequence in any order. Note that the "all" construct has no counterpart in the DTD formalism in which it would be rather complicated to express.

A "mixed content", i.e. a content containing both children elements and text is expressed by attaching the attribute-value pair

```
mixed="true"
```

to the schema element specifying the content model.

The following examples illustrate how content models can be specified. The first example specifies the structure of lecture notes. A sequence is used, for the order is relevant. "Occurrence constraints" with an obvious meaning specify how many of each element should occur: E.g. a lecture note must have exactly one title, but might have any (non-zero) number of chapters.

Example:

```
<xsd:complexType name="lecture-notes">
   <xsd:complexContent>
```

```
      <xsd:restriction base="xsd:anyType">
         <xsd:sequence>
           <xsd:element name="title"
                              minOccurs="1"
                              maxOccurs="1"
                              type="xsd:string"/>
            <xsd:element name="author"
                              minOccurs="0"
                              type="xsd:string"/>
            <xsd:element name="abstract"
                              minOccurs="0"
                              maxOccurs="1"
                              type="xsd:string"/>
            <xsd:element name="chapter"
                              minOccurs="1"
                              type="xsd:string"/>
         </xsd:sequence>
      </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>
```

The following example specifies the structure of an address, as it might occur in a database. In this example, each component of an address must occur exactly once, but the order is irrelevant. Therefore, the grouping "all" is used.

Example:

```
<xsd:complexType name="address">
   <xsd:complexContent>
      <xsd:restriction base="xsd:anyType">
         <xsd:all>
           <xsd:element name="street"
                               type="xsd:string"/>
            <xsd:element name="street-number"
                               type="xsd:nonNegativeInteger"/>
            <xsd:element name="zip-code"
                               type="xsd:nonNegativeInteger"/>
            <xsd:element name="city"
                               type="xsd:string"/>
         </xsd:all>
      </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>
```

Definitions of complex types might be nested. The following example refines the previous definition of the type lecture-notes requiring chapter elements to consist of nonempty sequences of paragraphs and text (i.e. mixed content).

Example:

```
<xsd:complexType name="lecture-notes">
   <xsd:complexContent>
      <xsd:restriction base="xsd:anyType">
         <xsd:sequence>
           <xsd:element name="title"
                        minOccurs="1"
                        maxOccurs="1"
                        type="xsd:string"/>
            <xsd:element name="author"
                        minOccurs="0"
                        type="xsd:string"/>
            <xsd:element name="abstract"
                        minOccurs="0"
                        maxOccurs="1"
                        type="xsd:string"/>
            <xsd:element name="chapter">
               <xsd:complexType>
                  <xsd:complexContent>
                     <xsd:restriction base="xsd:anyType"
                                      mixed="true">
```

```
                        <xsd:sequence>
                           <xsd:element name="paragraph"
                                        minOccurs="1"
                                        maxOccurs="1"
                                        type="xsd:string"/>
                        </xsd:sequence>
                     </xsd:restriction>
                  </xsd:complexContent>
               </xsd:complexType>
            </xsd:element>
         </xsd:sequence>
      </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>
```

It might be preferable not to specify the chapter type within the specification of the content model of lecture-notes e.g. for readibility reasons or if the chapter type is to be used within several type definitions. This can be achieved as follows:

Example:

```
<xsd:complexType name="lecture-notes">
   <xsd:complexContent>
      <xsd:restriction base="xsd:anyType">
         <xsd:sequence>
           <xsd:element name="title"
                       minOccurs="1"
                       maxOccurs="1"
                       type="xsd:string"/>
            <xsd:element name="author"
                       minOccurs="0"
                       type="xsd:string"/>
            <xsd:element name="abstract"
                       minOccurs="0"
                       maxOccurs="1"
                       type="xsd:string"/>
            <xsd:element name="chapter"/>
         </xsd:sequence>
      </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="chapter">
   <xsd:complexContent>
      <xsd:restriction base="xsd:anyType"
                              mixed="true">
        <xsd:sequence>
          <xsd:element name="paragraph"
                               minOccurs="1"
                               maxOccurs="1"
                               type="xsd:string"/>
          </xsd:sequence>
      </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>
```

Specifying a chapter type at the top level does not preclude to re-specify a type with the same name within a type definition. In such a case, the type name re-specified within a type definition overrides (within this type specification) the type name previously specified, i.e. shadowing takes place. Thus, types and subtypes can be specified in a "context-dependent manner". Note, however, that an XML Schema specification can be expressed by a context-free grammar, more precisely by a regular tree-grammar. Note also that specifications local to type definitions overriding former specifications are not possible in the DTD formalism.

In the previous examples, element declarations have been used. In general, an element declaration consists in the specification of an (element) name or of a reference to another element declaration and of a type. It is possible to specify occurrence constraints (minOccurs, maxOccurs) and a default (use="default" value="default-value") or fixed value (use="fixed" value="fixed-value").

5.4.2 Attribute Specification

An attribute is specified by giving an attribute name, a (built-in or user-defined) simple type for the attribute values, and by specifying a usage and possibly a default value (or a pre-set value if the usage is "fixed").

Examples of attribute specifications are as follows.

Example:

```
<xsd:attribute name="currency" type="xsd:token" use="required" value="Euro">
<xsd:attribute name="key" type="xsd:ID" use="optional">
<xsd:attribute name="creation-time" type="xsd:dateTime" use="required">
<xsd:attribute name="author" type="xsd:string" use="fixed" value="François Bry">
<xsd:attribute name="country" type="xsd:NMTOKEN" use="fixed" value="Spain">
```

Such attribute specifications can be part of complex type specification, e.g. as in the following example:

Example:

```
<xsd:complexType name="address">
   <xsd:complexContent>
      <xsd:restriction base="xsd:anyType">
         <xsd:all>
           <xsd:element name="street"
                                type="xsd:string"/>
            <xsd:element name="street-number"
                                type="xsd:nonNegativeInteger"/>
            <xsd:element name="zip-code"
                                type="xsd:nonNegativeInteger"/>
            <xsd:element name="city"
                                type="xsd:string"/>
         </xsd:all>
      </xsd:restriction>
   </xsd:complexContent>
   <xsd:attribute name="country" type="xsd:NMTOKEN" use="fixed" value="Spain"/>
</xsd:complexType>
```

Often, several types have the same attributes. Therefore, XML Schema makes it possible to specify attribute groups. The following example shows the specification and use of an attribute group.

Example:

```
<xsd:attributeGroup name="price-attributes">
   <xsd:attribute name="currency" type="xsd:NMTOKEN" use="required" value="Euro">
   <xsd:attribute name="price" type="xsd:decimal" use="required">
</xsd:attributeGroup>

<xsd:complexType name="catalog-item">
   <xsd:all>
       <xsd:element name="name" type="xsd:string"/>
       <xsd:element name="description" type="xsd:string"/>
   </xsd:all>
   <xsd:attributeGroup ref="price-attributes"/>
</xsd:complexType>
```

5.5 Further Features

XML Schema allows

- to specify integrity constraints expressing that some attribute or element value must be unique within a certain scope specified using the selection language XPath (cf. unique elements) and specifying keys for elements (cf. key and keyref elements)
- local and global type definitions

- import and export type definitions

5.6 Integrity Constraints

The specifications of most of the nowadays ubiquitous information systems include integrity constraints, i.e. conditions rejecting so-called *invalid* or *inconsistent* data. XML Schema and DTDs provide a notion of *validity* based on structure and simple value types.

In relational database terminology, the term *integrity constraint* is used to describe various constraints over data:

- referential integrity
- identity integrity

A relational database is usually made up of one or many relations, also called tables. Each attribute of the relation (or each column of a table) has defined types, i.e. string, number etc.

Example: a database of students

firstName	lastName	topic	matrikel-nr
Rosa	Aguilar	demography	44283917
Christoph	Wieser	computer science	1939293

To unambiguously identify a tuple, one or some attributes need to be unique. This attribute (or group of attributes) is called a key. In the student database example the **matrikel-nr** attribute is suited as a key.

Identity integrity is fulfilled, if

- For every tuple the key attributes have a value assigned, eg. are not null (in the student database example: if every student has a valaue in the matrikel-nr attribute)
- There are not two tuples in the same table with equal values for the key attributes.

It is possible to refer to a tuple in eg. another table using the key of the tuple in mind. The referring tuple contains a so-called *foreign key*. *Referential integrity* is fulfilled, if for all foreign keys, there is an instance of a tuple with that key referenced. In the following example of a lecture database, the referential integrity constraint on student references is violated, as the third tuple uses a foreign key not defined in the student table (as given in the former example).

Example: a lecture database table referencing students

lecture	student
markup	1939293
markup	44283917
markup	99833200
statistics	44283917

5.6.1 Integrity Constraints in XML Schema

An equivalent to the key/foreign key based reference mechanism in XML are the ID/IDREF attributes. When using DTDs, it is possible to declare an attribute to be of type ID and hence identity integrity will be enforced on validation. The use of IDREF, according to the XML recommendation, however, will not enforce referential integrity. It is hence possible to have a valid XML document, where a reference uses an undefined key. Currently, XML Schema neither supports identity integrity on identity types, nor referential integrity on reference types while validating.

5.6.2 Current Approaches of Constraints in XML Schema

XML Schema provides various features that come near to check of referential integrity and identity integrity, yet they are not applicable to the types ID and IDREF, which are arguably XML's dedicated reference mechanism. XML Schema provides constraints for

- uniqueness
- uniqueness of composite fields
- key reference check

The use of constraints relies on XPath, an embeddable selection language (cf. Chapter 5, Section 2 Data Selection with XPath).

Uniqueness

The uniqueness constraint requires, that there are not two elements of the given type in the containing content model that are identical. There is also a so called "key" constraint — it corresponds to the uniqueness constraint with the additional `minOccurence="1"` occurence constraint.

Uniqueness of Composite Fields

Uniqueness can be defined on groups of elements. If, eg., it is desireable to have uniqueness of names in a database, but names are split up in first- middle- and last-name elements, uniqueness can be defined upon the composition of those elements.

Key Reference

Syntactically similar to the uniqueness constraint. The key ref constraint checks, that there is actually an element under the same parent node that matches the given path expression and that fulfills the uniqueness (or key) constraint.

Example:

```
<xsd:element name="bibliography">
        <xsd:complexType>
                <xsd:element name="book">
                        <xsd:attribute name="isbn" type="xs:string"/>
                </xsd:element>
                <xsd:element name="author">
                        <xsd:attribute name="firstName" type="xs:string"/>
                        <xsd:attribute name="lastName" type="xs:string"/>
                        <xsd:attribute name="book" type="xs:string"/>
                </xsd:element>
        </xsd:complexType>
        <xsd:unique name="book">
                <xsd:selector xpath="book"/>
                <xsd:field xpath="@isbn"/>
        </xsd:unique>
        <xsd:unique name="author">
                <xsd:selector xpath="author"/>
                <xsd:field xpath="@firstName"/>
                <xsd:field xpath="@lastName"/>
        </xsd:unique>
        <xsd:keyref name="author" refer="book">
                <xsd:selector xpath="book"/>
                <xsd:field xpath="@isbn"/>
        </xsd:keyref>
</xsd:element>
```

5.6.3 Other Approaches for Validation of Integrity Constraints for XML

There is an ISO standard called *Schematron*, "a language for making assertions about patterns found in XML

douments", which is well suited to check integrity constraints **[SCHMTR]**. Schematron is not introduced in detail at this point, as it relies on XPath and XSLT, languages to be introduced later in this lecture.

6 References

[ABS]	S. Abiteboul, P. Buneman, D. Suciu. Data on the Web — From Relations to Semistructured Data and XML, ISBN 1-55860-622-X, Morgan Kaufmann, 2000
[BB]	J. Berstel, L. Boasson. XML Grammars. Proc. of the 25th International Symposium "Mathematical Foundations of Computer Science", Lecture Notes in Computer Science 1893, Springer-Verlag, ISBN 3-540-67901-4, pp. 182-191, 2000. **http://www-igm.univ-mlv.fr/~berstel/Articles/MFCSgrammars.ps** (PostScript)
[Hask]	The Haskell home page. **http://www.haskell.org/**
[ISO8859]	ISO/IEC 8859 Information technology — 8-bit single-byte coded graphic character sets, International Organization for Standardization
[RFC1766]	H. Alvestrand, ed. Tags for the Identification of Languages 1995. **http://www.ietf.org/rfc/rfc1766.txt**
[SAX]	SAX — Simple API for XML **http://www.saxproject.org/**
[Sch1]	D. C. Fallside, ed. XML Schema Part 0: Primer. W3C Recommendation, 2 May 2001. **http://www.w3.org/TR/xmlschema-0/**
[Sch2]	H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, eds. XML Schema Part 1: Structures. W3C Recommendation, 2 May 2001. **http://www.w3.org/TR/xmlschema-1/**
[Sch3]	P. V. Biron, A. Malhotra, eds. XML Schema Part 2: Datatypes. W3C Recommendation, 2 May 2001. **http://www.w3.org/TR/xmlschema-2/**
[SCHMTR]	Schematron — A language for making assertions about patterns found in XML documents. **http://www.schematron.com/**
[SMLa]	Robert Harper. Programming in Standard ML. **http://www.cs.cmu.edu/People/rwh/introsml/**
[SMLb]	Stephen Gilmore. Programming in Standard ML '97: An On-line Tutorial. **http://www.dcs.ed.ac.uk/home/stg/NOTES/**
[SMLc]	Lawrence Paulson. ML for the Working Programmer. MIT Press, 2nd Edition, 1996. ISBN 0-521-57050-6 (hardback), ISBN 0-521-56543-X (paperback). **http://www.cl.cam.ac.uk/users/lcp/MLbook/**
[XML]	T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, eds. Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, 6 October 2000, **http://www.w3.org/TR/REC-xml**

Main: 3StandardQueryAndTransformationLanguagesForXML

XML and Databases: 3 Standard Query and Transformation Languages for XML

1 Need for Transformations

Querying XML or semistructured data means both

- retrieving XML (or semistructured) data from the Web
- structuring the data retrieved into new XML (or semistructured) data items

Retrieving XML (or semistructured) data from the Web calls for languages permitting

- to access distinct Web sites
- to select data from partly specified data items at partly specified positions, since Web data are highly heterogeneous

Re-structuring or "transforming" the data retrieved into new data items is necessary, since in contrast to relational databases XML and semistructured data do not have a common structure. Data re-structuring naturally takes place in widespread applications like e.g.

- collecting railways or flight connections from different servers and/or airlines and/or railways companies so as to set-up the timetable of a trip,
- generate (possibly CSS-styled) HTML data from XML data for rendering with a standard browser,
- generating a table of content or a term index for a document.

In some cases, re-structuring transformations more or less keep the tree structure of a document. This is e.g. the case when HTML data are styled with CSS: New nodes might be added to the document tree (e.g. a portion of text is inserted at the beginning of every list item), other nodes might be removed (e.g. when some HTML elements are not to be rendered), but the overall tree structure is not modified.

In other cases, re-structuring transformations considerably modify the tree structure of a document. This is e.g. the case if a publication list structured by authors is to be generated from a publication list structured by years of publication. In such a case, some nodes of the document tree have to be duplicated (the nodes corresponding to publications having more than one author) and the ordering of the nodes in the final document might considerably differ from their ordering in the original document.

2 Data Selection with XPath

With XPath **[XPath]** one can localise (sets of) nodes within the tree associated with an XML document, i.e. elements, attributes, and text.

A language very similar to XPath is XPointer. With XPointer **[XPointer]**, one can localise regions of a document. Note that a region in a document is not necessarily included in one document node. This is e.g. the case of the region beginning with the character "A" and finishing with the character "Z" in the following document.

Example:

```
<?xml version="1.0" encoding="ISO-8859-15" standalone="yes"?>

<a>
   <b>
      abcd A efgh
   </b>
    <c>
```

```
      ijkl Z mnop
    </c>
 </a>
```

Both XPath and XPointer are inspired from so-called regular path expressions. XPointer is essentially an extension of XPath. How XPointer extends XPath is briefly explained at the end of this section on XPath.

2.1 Regular Path Expressions

Regular path expressions are regular expressions serving to describe a traversal of a tree from a "context node". For example, a.b._*.c denotes the set of c-labeled nodes that can be reached from the node considered as the current context by traversing an a-labeled node which is a child of the context node and itself has a b-labeled child, which in turn has descendants (at any depth (*) and with unspecified labels (_)) one of them containing a c-labeled node (which is to be selected).

The syntax of regular path expressions (rpe) can be defined as follows using a context-free grammar:

```
rpe ::= label | "_" |
        "(" rpe ")" |
        rpe "." rpe | rpe "|" rpe |
        rpe "?" | rpe "*" | rpe "+" .
```

The variable *label* denotes an element label (i.e. name or tag). "_" is a wildcard, i.e. it matches with every element label. "." expresses a parent/child relationship. "|" is an alternative. "?" expresses an option (i.e. 0 or 1 occurrence). "*" denotes Kleene's closure operator, i.e. it denotes 0, 1 or more occurrences. "+" denotes 1 or more occurrences. Note that _* (_+, resp.) matches with any, possibly empty (non-empty, resp.), sequence of labels.

The language of regular path expression might be extended with so-called qualifiers, build using extra language constructs not further specified here, expressing conditions on the nodes selected. E.g. the following regular path expression with qualifiers might select

```
_*.chapter[title/_["xml"]].section
```

section of chapters having a title in which the string "xml" occurs.

2.2 XPath

XPath **[XPath]** uses so-called "axis specifiers" and "node tests" in lieu of regular path expressions. The axis specifier, e.g. descendant or child, specifies the traversal of the document tree. A node test corresponds to a label in a regular path expression. Axis specifier and node test are connected with "::". (be careful, attributes and node tests like "text()" are treated like axes)

E.g.

```
descendant::a
```

corresponds to the regular path expression

```
_ *.a
```

selecting a-labeled nodes at any depth.

The nodes selected by an XPath expression can be further constrained by so-called predicates expressing requirements on the nodes to be selected, e.g. the presence of an attribute, or of an attribute-value pair, or of certain children, descendant, preceeding or following elements.

XPath, as well as many other XML related formalisms, refers to the so-called "document order" of XML documents. Recall that an XML document specifies an ordered tree the nodes of which correspond to the elements of the document. Ordered trees, hence XML documents, induce as follows a total order on their nodes, i.e. elements:

- Sibling nodes are ordered in their sibling order (i.e. like in the tree).
- If n_1 and n_2 are sibling nodes and if n_1 precedes n_2, then every descendant of n_1 precedes every descendant of n_2.

Note that the document order of the elements of an XML document is the order in which these elements are specified in the (textual) XML document as well as the order of a depth-first leftmost traversal of the (ordered) document-tree associated with that XML document.

2.2.1 Axis Specifiers

The axis specifiers give rise to a navigation from the context node in all directions (to the leaves, to the root, to nodes preceding or following in "document order").

/	**select the document root (which is considered the parent of the document element)**
ancestor	**proper ancestor of current node**
ancestor-or-self	**current node or proper ancestor of current node**
attribute	**attribute of current node**
child	**immediate descendant (child) of current node**
descendant	**proper descendant of current node**
descendant-or-self	**current node or proper descendant of current node**
following	**node following the current node in document order**
following-sibling	**node following the current node in document order and at the same depth as the current node**
preceding	**node preceding the current node in document order**
preceding-sibling	**node preceding the current node in document order and at the same depth as the current node**
namespace	**namespace node of the current node**
parent	**immediate ancestor (parent) of current node**
self	**current node**

Note that according to the Common Data Model of XQuery and XPath **[XQDM]**, a namespace specification is represented by a node.

2.2.2 Node Tests

name	**matches elements of type *name***
*****	**matches every element**
namespace*:*name	**matches elements of type *name* from the given namespace**
namespace*:	**matches every element from the given namespace**
comment()	**matches comment nodes**
text()	**matches text nodes**
processing-instruction()	**matches processing-instructions**
processing-instruction(target)	**matches processing-instructions of the form <?*target* ... ?>**
node()	**matches every node**

Note that according to the Common Data Model of XQuery and XPath **[XQDM]**, a comment is represented by a node.

2.2.3 Predicates

Predicates express conditions on the nodes to be selected. They appear after the node test between [and]. In the following table, *nodetest* denotes a node test.

***nodetest*[position()=1]**	**matches the first node**
***nodetest*[position()=last()]**	**matches the last node**
***nodetest*[position() mod 2 = 1]**	**matches every second node**
***nodetest*[attribute::*attribute-name*]**	**matches nodes with attribute *attribute-name***
***nodetest*[attribute::*attribute-name*=value]**	**matches nodes with attribute-value pair *attribute-name*=value**
***nodetest*[attribute::*attribute-name*!=value]**	**matches nodes with attribute-value pair *attribute-name*=value$_1$ such that *value* and *value*$_1$ are different**
***nodetest*[not(attribute::*attribute-name*)]**	**matches nodes with no attribute *attribute-name***

In a predicate, "self::node()" denotes the current node. In a predicate, "=" ("!=", resp.) can be used for expressing the inclusion (non-inclusion) in a set of values.

Nested predicates are possible. E.g. child::a[child::b[child::c]] selects a elements having at least one b element that in turn has at least one c element.

Sequences of predicates are possible. E.g. child::a[child::a1][child::a2] selects a elements having at least one a1 child and at least one a2 child.

In a predicate, full XPath expressions are allowed making it possible to specify subdocuments below (i.e. rooted at) every node. E.g. child::a[child::a1/child::b1][child::a2/descendant::b2]/child::c selects all c elements of a elements having at least one a1 child that has in turn a b1 element and at least one a2 child that has in turn a b2 descendant.

Example:

```
<bib>
   <book>
      <author> ... </author>
      <title> ... </title>
      ...
   </book>
   <book lang='en'>
      <author> ... </author>
      <title> ... </title>
      ...
   </book>
   ...
</bib>
```

Example:

```
1)   descendant::author[position()=1]

2)   /descendant::author[position()=1]

3)   (child::bib/descendant::author)[position()=1]

4)   descendant::book[attribute::lang="en"]

5)   descendant::book[attribute::lang!="en"]

6)   /descendant::author[not(self::node()=/descendant::book[attribute::lang!="en"]/descendant::author)]
```

Applied to the XML data given above and assuming that the current node is the root, the XPath (XPath 1.0) expressions of the previous example select the following sets of nodes:

1. The set of all first authors of books
2. The set of all first authors of books (whatever the current node is, always starts the navigation from the root)
3. The singleton set containing the first author mentioned in the bib element
4. All books with an attribute-value pair 'lang="en"'
5. All books with no attribute-value pair 'lang="en"'
6. Starting from the root of the document, all authors at any depth that are no authors of some book with an attribute-value pair 'lang="*value*"' such that *value* is not 'en'.

2.2.4 Abbreviated Syntax

XPath also has an abbreviated syntax which basically is as follows:

long	*short*
child::*label*	***label***
child::*	*
/descendant-or-self::node()/	//
self::node()	.
parent::node()	..
attribute::	@
[position()=*n*]	**[*n*]**

The six examples of XPath expressions given above can be expressed in abbreviated syntax as follows:

Example:

```
1)   .//(author[1])

2)   //(author[1])

3)   (bib//author)[1]

4)   .//book[@lang="en"]

5)   .//book[@lang!="en"]

6)   //author[not(.=//book[@lang!="en"]//author)]
```

Note that the translation of `//` is `/descendant-or-self::node()/` and that this has some maybe unexpected effects:

- `//author` it is *not* the same as `/descendant-or-self::author/`. The former selects only descendants (children, grandchildren, etc.) of the context node with the label `author`. The latter *additionally* selects the context node if it has the label `author`.
- `//author` might seem the same as `/descendant::author/`; and while this is true when `//author` is the full path, it is not true

anymore when it occurs as part of a path. For this consider the following.

- `//author[1]` is not the same as `/descendant::author[1]/`. Since the former stands for `/descendant-or-self::node()/author[1]`, it selects all descendants of the context node that are the *first child of their respective parent*. The latter however selects the *single* descendant of the context node that comes *first in document order*.

Also note that in `//author` the context node is the document root (since the unabbreviated translation starts with a `/`). In order to start from the current context node, one has to write `.//author`. (This is an important motivation for introducing `.` into the abbreviated syntax.)

Note that the axes preceding, preceding-sibling, following, and following-sibling have no counterpart in the abbreviated syntax.

The abbreviated and unabbreviated syntaxes might be mixed in a location path.

2.2.5 XPath 2.0

XPath Version 2.0 (short XPath 2.0) **[XPath2.0]** is in preparation. XPath 2.0 will extend XPath 1.0 with:

- Support for XML Schema built-in data types
- Explicit existential (some, for any) and universal (all, every) quantifiers
- FR (for return) construct from XQuery FLWOR expressions
- Extended set of aggregation functions (e.g. min, max, avg)
- Conditional expressions (if-then-else)
- Node set intersection and difference functions (intersect, except)
- Regular expressions for string matching

2.3 XPointer

XPointer **[XPointer]** is based on XPath **[XPath]** which it extends, so as

- to localize portions of documents ("ranges"),
- to localize positions or ranges in documents using string matching,
- to allow localization expressions in URI.

XPointer is not further addressed in this course, for it is not directly relevant to query and transformation languages.

3 The Transformation Language XSLT

XSLT is part of XSL (the Extensible Stylesheet Language) **[XSL]**. XSL consists of:

- A language for transforming XML documents: XSLT (XSL Transformations Version 1.0) **[XSLT]**.
- A localization language for XML documents: XPath **[XPath]**.
- An "XML vocabulary" for specifying formatting semantics: XSL-FO (XSL Formatting Objects) **[XSL-FO]**.

A new version of XSLT is under development **[XSLT2.0]**. The following presentation refers to XSLT Version 1.0 **[XSLT]**.

The program examples given in this section were inspired from **[Gro]**.

3.1 XSLT Basics

XSLT is a (sort of) functional language.

An XSLT program (also called XSLT style-sheet), is an XML document with one of the following forms:

Example:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   ...
</xsl:stylesheet>

<?xml version="1.0" encoding="iso-8859-1"?>
<eg:transform xmlns:eg="http://www.w3.org/1999/XSL/Transform" version="1.0">
  ...
</eg:transform>
```

Consider the following XML document:

Example:

```
<?xml version='1.0'?>
<paragraph>
      Some words are <emphasize>emphasized</emphasize>.
      Some <emphasize>other</emphasize> also are.
</paragraph>
```

The following XSLT program transforms this XML document into an XHTML document:

Example:

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

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

   <xsl:template match="paragraph">
      <p><xsl:apply-templates/></p>
   </xsl:template>

   <xsl:template match="emphasize">
     <em><xsl:apply-templates/></em>
   </xsl:template>

</xsl:stylesheet>
```

The output of the transformation is as follows:

Example:

```
<?xml version='1.0'?>
<p>
       Some words are <em>emphasized</em>.
       Some <em>other</em> also are.
</p>
```

The XSLT program given above is evaluated as follows. At the beginning, the current node is the root of the document (which is considered to be a parent of the document instance, i.e. of the element "paragraph"). The first "template" matches with the element "paragraph". The current nodes are now the children of that element, i.e. the texts and the element "emphasize" it contains. The recursive call "apply-templates" applies the templates of the XSLT program to these current elements. The second template of the XSLT program matches with the element "emphasize", generating XHTML "em" elements. The recursive call "apply-templates" in the body of the template matching with elements "emphasize" has no effect: The evaluation terminates.

Note that every element occurring in a template and which is not specified in the XSLT namespace is copied into the output.

The prolog of an XHTML document is missing in the output of the XSLT program. In order to generate it, the XSLT program can be transformed as follows:

Example:

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

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:output method="xml" encoding="iso-8859-1"
                  doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
                  doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"/>

   <xsl:template match="paragraph">
      <html xmlns="http://www.w3.org/1999/xhtml">
         <head>
            <title>A Document</title>
         </head>
         <body>
            <p>
                 <xsl:apply-templates/>
            </p>
         </body>
      </html>
   </xsl:template>

   <xsl:template match="emphasize">
     <em><xsl:apply-templates/></em>
   </xsl:template>

</xsl:stylesheet>
```

3.2 The Recursive Computation Model of XSLT

Consider the following XML document:

Example:

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

<doc>
   <paragraph>
      Here some <emphasize>words</emphasize> are
      <emphasize>recursively
            <emphasize>
                  emphasized
            </emphasize>
      </emphasize>.
   </paragraph>
</doc>
```

The following XSLT program transforms this XML document in XHTML in such a way, that simply emphasized texts are transformed into texts in italics, doubly emphasized texts are transformed into texts in bold:

Example:

```
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="xml" encoding="iso-8859-1"
                  doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"
                  doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"/>

<xsl:template match="doc">
  <html>
     <head>
         <title>A Document</title>
     </head>
     <body>
         <xsl:apply-templates/>
     </body>
  </html>
</xsl:template>

<xsl:template match="paragraph">
  <p><xsl:apply-templates/></p>
</xsl:template>

<xsl:template match="emphasize">
  <i><xsl:apply-templates/></i>
</xsl:template>

<xsl:template match="emphasize/emphasize"/>
  <b><xsl:apply-templates/></b>
</xsl:template>
</xsl:stylesheet>
```

The patterns in `<xsl:template match="`*pattern*`">` such as "`emphasize/emphasize`" are written in a subset of abbreviated XPath (see Sec 5.2 of the XSLT Recommendation for a full grammar). Matching of patterns and nodes is defined in the XSLT Recommendation based on the interpretation of the pattern as an XPath expression as follows:

"A pattern is defined to match a node if and only if there is a possible context such that when the pattern is evaluated as an expression with that context, the node is a member of the resulting node-set."

Note that the formulation is "if there is a context", i.e., the XPath expression is not just evaluated from the current node, but from any of its ancestor nodes. For example: the inner `<emphasize>` element in the example document is matched by `emphasize/emphasize` using as context the outer `<emphasize>` element.

To understand a given pattern it is usually more intuitive to forget about this precise definition based on expression evaluation. Instead one can simply read the pattern "backwards". For example, a pattern `a/b//c/d` matches a given `<d>` element, if it has as parent a `<c>` element, which in turn has an ancestor that is a `<b>` element with an `<a>` parent element.

The restriction of patterns to using only the child/descendant axes -- as opposed to allowing any XPath expressions that yield a node-set -- is probably for efficiency concerns. (Although the XSLT recommendation does not explicitly say so.) The "if there is a possible context..." in the formal definition naively means a search through all nodes in the document. However, due to the restriction to child/descendant axes, the search can be constrained to only the ancestors of the current node. Furthermore, instead of evaluating the pattern starting from all potential context nodes, we can evaluate it backwards, i.e., search for a context node starting from the current node.

Applied to the previous XML document, this XSLT program generates the following XHTML document:

Example:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
                            "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
   <head>
       <title>A Document</title>
   </head>
   <body>
      Here some <i>words</i> are
      <i>recursively
            <b>
                  emphasized
            </b>
      </i>.
   </body>
</html>
```

The recursive computation model of XSLT is a root-to-leaves traversal of the document. Recursive procedures or functions in the common sense cannot be specified with XSLT. More generally, it is not possible to specify procedures or functions in the common sense. Furthermore, it is not possible to write an XSLT program expressing stepwise transformations of an XML document because the output of an XSLT program or procedure cannot - within an XSLT program - be used as input for that program or procedure. Indeed XSLT has no notion of procedure (or user-defined function). Contradicting claims of the authors of XSLT and of some supporters of this language, this turns out to be a serious limitation of XSLT.

XSLT has a notion of modules.

In developing XSLT programs, one often encounters the following two difficulties:

- The resolution of conflicts between several templates that might apply, i.e. whose patterns match with the current element (cf. Section 3.2.3 Conflict Resolution).
- Processing of elements with same names in different contexts differently (cf. Section 3.2.4 Modes).

3.2.1 Selection expressions

The attribute `select="`*expression*`"` can be used in `<apply-templates>` in order to apply the templates to other nodes than the children of the current node (which would be used by default when there is no `select` attribute). The *expression* is an XPath expression.

Since XPath makes it possible to reach every node of a document from every other node, it is possible to write XSLT templates acting on remote nodes, thus overcoming (or simply departing) from the recursive root-to-leaves traversal of the input document (i.e. the recursive computational model).

Consider the following XML document:

Example:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<address-book>
   <entry type="person">
      <first-name> ... </first-name>
      <last-name> ... </last-name>
      <email> ... </email>
      <telephone> ... </telephone>
   </entry>
   <entry type="company">
      <name> ... </name>
       <telephone> ... </telephone>
   </entry>
   <entry type="person">
      <first-name> ... </first-name>
      <last-name> ... </last-name>
      <telephone> ... </telephone>
   </entry>
</address-book>
```

The following XSLT template can be used to generate a list without duplicates of all first names occurring in the previous document:

Example:

```
<xsl:template match="/address-book">
   <first-name-list>
      <xsl:apply-templates select="first-name[not(.=preceding::*)]">
```

```
         <xsl:sort />
      </xsl:apply-templates>
   </first-name-list>
</xsl:template>
```

Note the use of the XPath expression in the apply-templates instruction calling a sort procedure. Selection expressions are also used in other places, e.g., for binding variables (see below).

3.2.2 Variables

Variables can be used in XSLT programs. In the following template (referring to the previous XML document), a variable "lastname" is declared and bound to the (text) content of the selected last-name element, then an element labeled with the variable value is created, the content of which is that of the first-name element of the input document:

Example:

```
<xsl:template match="//entry">
   <xsl:variable name="lastname" select="last-name"/>
   <xsl:element name="$lastname">
      <xsl:value-of select="first-name"/>
   </xsl:element>
</xsl:template>
```

The following example is a variation of the previous example where the first name also appears as value of an attribute named first:

Example:

```
<xsl:template match="//entry">
   <xsl:variable name="lastname" select="last-name"/>
   <xsl:element name="$lastname">
      <xsl:attribute name="first">
         <xsl:value-of select="first-name"/>
      </xsl:attribute>
      <xsl:value-of select="first-name"/>
   </xsl:element>
</xsl:template>
```

3.2.3 Conflict Resolution

As already mentioned, in developing XSLT programs, one often encounters the difficulty of resolving conflicts between several templates whose patterns match with the current element.

XSLT provides means for conflict resolution based on ideas like the following:

1. giving priority to local templates over templates from imported modules
2. giving higher priority to so-called qualified children or attributes
3. a qualified child with a namespace has Priority 0.25
4. the priority of a template can be set in the program

Whether the XSLT approach to conflict resolution gives rise to "natural" programs is a debated issue.

3.2.4 Modes

As already mentioned, in developing XSLT programs, one often encounters the difficulty of processing of elements with same names in different contexts differently. To this aim, templates can be assigned so-called modes.

Consider e.g. the following document:

Example:

```
<?xml version='1.0'?>
<chapter id="xsl-basics">
   <title>XSL Basics</title>
   <para>This chapter is self-referential:
      <ref to="xsl-basics"/>.
   </para>
</chapter>
```

Assume that this document is to be transformed as follows:

Example:

```
<?xml version="1.0" encoding="..."?>
<html> <body>
<h2>XSL Basics</h2>
<p>
   This chapter is self-referential:
   <i>XSL Basics</i>.
</p>
</body> </html>
```

A naive XSLT program would be as follows:

Example:

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

   <xsl:template match="chapter">
      <html> <body>
         <xsl:apply-templates/>
      </body> </html>
   </xsl:template>

   <xsl:template match="chapter/title">
      <h2><xsl:apply-templates/></h2>
   </xsl:template>

   <xsl:template match="ref">
      <xsl:variable name="to" select="@to"/>
      <xsl:apply-templates select="//*[@id=$to]/title"/>
   </xsl:template>

   <xsl:template match="para">
     <p><xsl:apply-templates/></p>
   </xsl:template>

</xsl:stylesheet>
```

This program is naive, because it does not process the title element differently, depending on the context in which it occurs. This naive program generates the following output:

Example:

```
<?xml version="1.0" encoding="utf-8"?>
<html> <body>
  <h2>XSL Basics</h2>
  <p>This chapter is self-referential:
    <h2>XSL Basics</h2>.
  </p>
</body> </html>
```

A correct XSLT program uses "modes" as follows:

Example:

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

   <xsl:template match="chapter">
      <html> <body>
         <xsl:apply-templates/>
      </body> </html>
   </xsl:template>

   <xsl:template match="chapter/title">
      <h2><xsl:apply-templates/></h2>
   </xsl:template>

   <xsl:template match="chapter/title" mode="crossref">
      <i><xsl:apply-templates/></i>
   </xsl:template>

   <xsl:template match="ref">
      <xsl:variable name="to" select="@to"/>
      <xsl:apply-templates
              select="//*[@id=$to]/title"
              mode="crossref"/>
   </xsl:template>

   <xsl:template match="para">
```

```
      <p><xsl:apply-templates/></p>
   </xsl:template>

</xsl:stylesheet>
```

3.3 The Imperative Computation Model of XSLT

XSLT also has an imperative computational model with constructs such as loops (for-each)

```
<xsl:for-each select="row">
   <tr><xsl:apply-templates/></tr>
</xsl:for-each>
```

assignment of a name to a template

```
<xsl:template name="NN"> ... </xsl:template>
```

assignment of a parameter to a template

```
<xsl:template name="NN">
    <xsl:param name="Param1"> Default Value 1 </xsl:param>
    <xsl:param name="Param2"> Default Value 2 </xsl:param>
    ...
</xsl:template>
```

call of a named template

```
<xsl:call-template name="NN" />
```

call of a named template with parameters (beware: "with-param" instead of "param")

```
<xsl:call-template name="NN">
     <xsl:with-param name="Param1"> Value1 </xsl:with-param>
     <xsl:with-param name="Param2"> Value2 </xsl:with-param>
     ...
</xsl:call-template>
```

Example:
Input:

```
<?xml version='1.0'?>
<table>
  <row>
     <entry>a1</entry><entry>a2</entry>
  </row>
  <row>
     <entry>b1</entry><entry>b2</entry>
  </row>
  <row>
     <entry>c1</entry><entry>c2</entry>
  </row>
</table>
```

XSLT program:

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="table">
  <table>
    <xsl:for-each select="row">
      <tr>
        <xsl:for-each select="entry">
          <td><xsl:apply-templates/></td>
        </xsl:for-each>
      </tr>
    </xsl:for-each>
  </table>
</xsl:template>
</xsl:stylesheet>
```

Output:

```
<table>
   <tr>
      <td>a1</td><td>a2</td>
   </tr>
   <tr>
```

```
      <td>b1</td><td>b2</td>
   </tr>
   <tr>
      <td>c1</td><td>c2</td>
   </tr>
</table>
```

Example:
Input:

```
<chapter>
   <warning>
      <para>Using a damaged extension cord may cause a fire.</para>
   </warning>
   <caution>
      <para>Freshly brewed coffee is hot.</para>
   </caution>
</chapter>
```

XSLT program:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="html"/>

<xsl:template name="admonition">
  <xsl:param name="type">Warning</xsl:param>
  <table border="1">
    <tr><th><xsl:value-of select="$type"/>:</th></tr>
    <tr><td><xsl:apply-templates/></td></tr>
  </table>
</xsl:template>

<xsl:template match="warning">
  <xsl:call-template name="admonition"/>
</xsl:template>

<xsl:template match="caution">
  <xsl:call-template name="admonition">
    <xsl:with-param name="type">Caution</xsl:with-param>
  </xsl:call-template>
</xsl:template>

<xsl:template match="para">
  <p><xsl:apply-templates/></p>
</xsl:template>

<xsl:template match="emphasis">
  <i><xsl:apply-templates/></i>
</xsl:template>
</xsl:stylesheet>
```

Output:

```
<table border="1">
   <tr>
      <th>Warning:</th>
   </tr>
   <tr>
      <td>
         <p>Using a damaged extension cord may cause a fire.</p>
      </td>
   </tr>
</table>

<table border="1">
   <tr>
      <th>Caution:</th>
   </tr>
   <tr>
      <td>
         <p>Freshly brewed coffee is hot.</p>
      </td>
   </tr>
</table>
```

3.4 Further Constructs

3.4.1 Construction of the output

<xsl:text>: copies the content of the input in the output, the white spaces are kept.

<xsl:value-of> computes a value to be inserted in the output.

<xsl:copy>: copies the current node of the input in the output.

<xsl:copy-of>: copies the selected node of the input in the output.

<xsl:element>: for naming an element of the output, e.g.

```
<xsl:element name="">
   <xsl:apply-templates/>
</xsl:element>
```

<xsl:attribute>: for attaching an attribute to the immediately encompassing element of the output, e.g.

```
<table>
   <xsl:if test="@pgwide='1'">
      <xsl:attribute name="width">100%</xsl:attribute>
   </xsl:if>
   ...
</table>
```

3.4.2 Conditional

- <xsl:if>

```
<xsl:if test="">
   <xsl:text>
     this text only gets used if $somecondition is true()
   </xsl:text>
</xsl:if>
```

- <xsl:choose>

```
<xsl:choose>
   <xsl:when test="$count &gt; 2"><xsl:text>, and </xsl:text></xsl:when>
   <xsl:when test="$count &gt; 1"><xsl:text> and </xsl:text></xsl:when>
   <xsl:otherwise><xsl:text> </xsl:text></xsl:otherwise>
</xsl:choose>
```

3.4.3 Miscellaneous

<xsl:number>: for evaluating a numerical expression and converting the result into a string.

<xsl:sort>: sorts nodes, might be called recursively, allowed as child of a xsl:apply-templates or xsl:for-each element.

<xsl:message>: for generating error messages, e.g.

```
<xsl:message>
   <xsl:text>Error: no ID found for linkend: </xsl:text>
   <xsl:value-of select="@linkend"/>
   <xsl:text>.</xsl:text>
</xsl:message>
```

<xsl:variable>: for declaring a variable and binding it to string, a node, or part of a tree. Note that a variable can be bound to a value only once. Furthermore, the usual static scoping rules hold.

4 The Query Language XQuery

4.1 Origin of XQuery

The fact that with XQuery and XSLT there are two expressive XML query languages that can be used for mainly the same purposes is somewhat surprising. While XSLT was developed by the `document community`, XQuery is an XML Query language originating from the Database community, which can already be observed by its syntactical similarity to SQL. XQuery, also called XML Query, is derived from a query language called Quilt **[Quilt]**. Quilt had been conceived by Jonathan Robie (Software AG), Daniela Florescu (INRIA) and Don Chamberlain (IBM Almaden Research Center) as a query and transformation language for XML and semistructured data integrating most of the ideas of prototype query and transformation languages designed since the mid 90es -- hence the name Quilt.

Like Quilt and XSLT, XQuery uses XPath for locating nodes and/or text in the input data.

XQuery has adopted the type system of XML Schema **[Sch1]**, **[Sch2]**, **[Sch3]**.

4.2 XQuery Principles

Like XSLT, XQuery is a (sort of) functional language.

XQuery is case-sensitive and all keywords are written in lower-case. However this has only been decided relatively late in the standardization process. Therefore you will often find keywords like `for`, `let`, etc. written in upper-case in the literature (including earlier versions of this material).

4.2.1 The Common Data Model of XQuery and XPath

A data model has been specified for XQuery (and XPath) in **[XQDM]**. Basically, this data model names the concepts XQuery (and XPath) rely upon.

According to this data model, a document is a tree of nodes. With this data model, one can not only refer to (or model) complete documents, but also to

- fragments of documents,
- sequences of documents,
- sequences of fragments of documents.

The basic notion of the data model (called instance of the data model) is that of an ordered sequence of nodes. A sequence consisting of one node is identical with this node. Nested sequences are not allowed, but are flattened to ordinary sequences. The data model has a function called `fn:unordered` that specifies, that the order of a sequence is not important, and that an XQuery engine may change the order of the sequence for optimization purposes. In the case that there is an index for certain kinds of nodes in an XML document which are selected by an XQuery query, it is often easier to return the nodes in the order in which they appear within the index rather than in document order.

Also, in this data model, the tree associated with (or represented by) an XML document is assumed to have a root which is parent of the node corresponding to the outermost element of the document. The datamodel provides a function called `fn:root` to retrieve the root of a document.

An XQuery expression is built up from the following components:

- path expressions
- element constructors
- FLWOR (spoken "flower") expressions
- operator and function applications
- conditional expressions
- quantified expressions
- type tests or conversion

4.2.2 Path Expressions

XPath 2.0 is a subset of XQuery. In the past, only XPath 1.0 expressions were allowed within XQuery expressions, but today XPath 2.0 expressions are used.

Instead of the prefix dereferencing function id of XPath, XQuery uses the infix dereferencing operator =>. The left hand side of => must be an attribute of type IDREF (or IDREFS). The right hand side of => is a "name test", i.e. the name of the element searched for. As a consequence, the dereferencing operator => can only be used if a DTD or a schema (after XML Schema) assign the required type to the attribute used. Examples such as the following show the rational of using an infix dereferencing operator. The first example selects the figure that is referenced by the first figure reference that is a child of the context node:

Example:

```
child::figref[1]/attribute::refid=>figure
```

This second example selects the manager of the manager of the context node, assuming that the context node is an emp node and that each emp node has a manager attribute that references the emp node of the employee's manager:

Example:

```
attribute::manager=>emp/attribute::manager=>emp
```

4.2.3 Element Constructors

A common XML element, which appears within an XQuery program, is an XQuery element constructor. XQuery provides so-called *direct element constructors* and also *computed element constructors*. An element constructor can be embedded in an XQuery expression which is no element constructor.

Example:

```
<book isbn="isbn-0060229357">
    <title>Harold and the Purple Crayon</title>
    <author>
        <first>Crockett</first>
        <last>Johnson</last>
    </author>
</book>
```

The same XML fragment can be constructed by using computed element constructors:

```
element book {
   attribute isbn {"isbn-0060229357" },
   element title { "Harold and the Purple Crayon"},
   element author {
      element first { "Crockett" },
      element last {"Johnson" }
   }
}
```

In a direct element constructor, curly braces { } delimit enclosed (XQuery) expressions, distinguishing them from text. Enclosed expressions are evaluated and replaced by their value, whereas material outside curly braces is simply treated as text:

Example:

```
<example>
   <p> Here is a query. </p>
   <eg> $i//title </eg>
   <p> Here is the result of the above query. </p>
   <eg>{ $i//title }</eg>
</example>
```

The above query might generate the following result:

Example:

```
<example>
  <p> Here is a query. </p>
  <eg> $i//title </eg>
  <p> Here is the result of the above query. </p>
  <eg><title>Harold and the Purple Crayon</title></eg>
</example>
```

Also XML comments and XML processing instructions are allowed to appear in an XQuery element constructor. XQuery comments starting with `(:` and ending with `:)` may be nested and should not be confused with XML comments.

XQuery variables (beginning with $) must occur between { and }, if they are not be treated as literal text. XQuery variables may occur within markup. Assume that the variable $e is bound to an element the content of which has a numerical type. The following XQuery expression constructs an element the name of which is the name of the (element) value of $e and the content of which is twice as the content of the (element) value of $e:

Example:

```
<{name($e)}>
     {2 * number($e)}
</>
```

If the element to be constructed is supposed to have all the attributes of the value of $e, the query must be as follows:

Example:

```
<{name($e)}>
     {$e/@*}
     {2 * number($e)}
</>
```

`name(element)` and `number(element)` are XPath functions.

4.2.4 FLWOR Expressions

The acronym FLWOR, read "flower", results from the keywords used in the FLWOR expressions:

F: `for`
L: `let`
W: `where`
O: `order by`
R: `return`

A FLWOR expression declares and binds the variables that are used in element constructors.

- `for` and `let` clauses serve to declare and bind variables in two different manners.

Example:

```
for $x in /library/book return element result { $x }
```

In this example, the variable $x is successively bound to each of the book elements selected by the path expression `/library/book`, which is called the binding sequence of the `for` clause. This is called "iteration". The body of the `for` clause is subsequently evaluated for each variable binding, generating a list of values -- one for each variable binding.

Example:

```
let $x := /library/book return element result { $x }
```

In this second example, the variable $x is bound to the sequence of book elements selected by the path expression `/library/book`, and the body of the `let` clause is evaluated only once for this single binding of the variable `$x`, generating only one `result` element.

A FLWOR expression may contain several FOR and LET expressions, and arbitrary nestings of the different kinds of clauses.

- `where` clause: A FLWOR expression may (but does not have to) contain a single WHERE clause. It has the same function as a `where` clause in an SQL expression, i.e. to filter variable bindings. A `where` clause may contain itself FLWOR expressions, but also boolean expressions connected with `and` or `or`.

Example:

```
for $b in /www.buy-all-online.com/library/book
let $p := /www.buy-all-online.com/library/book/price
where $b/publisher = "Springer-Verlag"
      and
      avg($p) < 100
```

The filtering through a WHERE clause does not change the order of the nodes.

- `return` clause: returns the result of an XQuery expression. A result may be a node, a sequence of nodes or a primitive value (in the sense of the primitive data types of XML Schema **[Sch1]** **[Sch2]** **[Sch3]**).

Element constructors usually occur within RETURN clauses, but they may also appear within the binding sequence of FOR or LET clauses, or even within filter expressions.

Example:

```
for $b in fn:doc("bib.xml")//book
where $b/publisher = "Springer-Verlag"
      and $b/year = "2002"
return $b/title
```

In the following example, the function `unordered` expresses that the order does not have to be kept (which gives rise to optimization of the evaluation).

Example:

```
for $b in unordered(fn:doc("bib.xml")//book)
where $b/publisher = "Springer-Verlag"
      AND $b/year = "2002"
return $b/title
```

With XQuery, two elements have so-called "duplicate values" if their names (tags, labels), attribute-value pairs, and their (normalized) contents are identical. The XQuery function `fn:distinct-values` keeps from a sequence of elements exactly one value for all duplicates. The application of the `distinct-values` function to a sequence may change the order of the sequence.

Example:

```
for $p in distinct-values(fn:doc("bib.xml")//publisher)
let $a := avg(fn:doc("bib.xml")//book[publisher = $p]/price)
return
     {$p/text()}
```

- More on FLWOR expressions

A FLWOR expression may also occur within an element constructor. The following example uses the XQuery function `count`:

Example:

```
<price-list>
  {
   for $p in distinct-values(fn:doc("bib.xml")//publisher)
   let $b := fn:doc("bib.xml")//book[publisher = $p]
   where count($b) > 100
   return $p
  }
</price-list>
```

Example:

```
{
  for $a in distinct-values(fn:doc("bib.xml")//author)
  return
    {$a/text()}
    {
      for $b in fn:doc("bib.xml")//book[author = $a]
      return $b/title
    }
}
```

What does the previous query compute?

The XQuery function `attribute` serves to construct an attribute-value pair. Assume that the variable $e is bound to an element the children of which contain only text.

Example:

```
<{name($e)}>
   {
     for $c in $e/*
     return attribute(name($c), string($c))
   }
   {
     for $a in $e/@*
     return
        <{name($a)}>
            {string($a)}
   }
```

The previous query constructs an element with the same name as $e. In this new element, the attributes of $e are child elements, and the elements of $e are attributes of the new element.

4.2.5 Sorting

Sequences can be sorted in increasing (default) or decreasing order with respect to a sorting expression. Sorting expressions are allowed for all atomic types, for which an order is defined.

Example:

```
fn:doc("bib.xml")//publisher sortby name
```

Sorting with respect to several sorting expressions (in lexicographical order) is possible.

Example:

```
fn:doc("bib.xml")//book[price > 100] sortby (author[1], title) descending
```

4.2.6 Operator and Function Application

Binary arithmetic operators: `+ - * div mod`

Unary arithmetic operators: `+ -`

Value comparison operators: `eq, ne, lt, le, gt, ge`

General comparison operators: `=, !=, <, <=, >, >=` (can be applied to node sequences as well)

Node identity: `is`

Order operators: `<<, >>`

Logical operators: `and or`

XQuery has no NOT operator, but it has a Boolean function `not`.

Sequence operator:
`()` denotes the empty sequence
Operator `to`:
`4 to 1` denotes the sequence 4, 3, 2, 1
`1 to 4` denotes the sequence 1, 2, 3, 4
Operators `union, intersect, except, before,` and `after`

4.2.7 Conditional Expressions

`if` expression `then` expression `else` expression

4.2.8 Quantified Expressions

`some` variable `in` expression `satisfies` expression, the value is true or false.

`every` variable `in` expression `satisfies` expression, the value is true or false.

4.2.9 Data Types

The built-in data types of XQuery are those of XML Schema **[Sch1]** **[Sch2]** **[Sch3]**.

The user-defined data types of XQuery are those of XML Schema **[Sch1]** **[Sch2]** **[Sch3]**.

$var `instanceof` gives rise to test whether the value of a variable is an instance of a type.

`typeswitch` (expression) `case` ... serves to test dynamic types.

```
typeswitch($customer/billing-address)
   case $a as element(*, USAddress) return $a/state
   case $a as element(*, CanadaAddress) return $a/province
   case $a as element(*, JapanAddress) return $a/prefecture
   default return "unknown"
```

`cast` serves to an explicit type conversion.

4.2.10 Functions

XQuery gives rise to define functions. User defined functions might be recursive.

```
declare function local:summary($emps as element(employee)*)
   as element(dept)*
{
   for $d in fn:distinct-values($emps/deptno)
   let $e := $emps[deptno = $d]
   return

           {fn:count($e)}
           {fn:sum($e/salary)}
```

```
};

local:summary(fn:doc("acme_corp.xml")//employee[location = "Denver"])
```

5 References

[Gro]	P. Grosso XSL Concepts and Practical Use. Tutorial t the International Conference "XML Europe 2000", Paris, France. **http://nwalsh.com/docs/tutorials/xsl/**
[Quilt]	D. Chamberlin, J. Robie, D. Florescu. Quilt: an XML Query Language for Heterogeneous Data Sources. In Proceedings of WebDB 2000 Conference, Lecture Notes in Computer Science, Springer-Verlag, Dec. 2000. **http://www.almaden.ibm.com/cs/people/chamberlin/quilt_lncs.pdf** **http://www.almaden.ibm.com/cs/people/chamberlin/quilt.html**
[Sch1]	D. C. Fallside, ed. XML Schema Part 0: Primer. W3C Recommendation, 2 May 2001. **http://www.w3.org/TR/xmlschema-0/**
[Sch2]	H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, eds. XML Schema Part 1: Structures. W3C Recommendation, 2 May 2001. **http://www.w3.org/TR/xmlschema-1/**
[Sch3]	P. V. Biron, A. Malhotra, eds. XML Schema Part 2: Datatypes. W3C Recommendation, 2 May 2001. **http://www.w3.org/TR/xmlschema-2/**
[Xcerpt]	Xcerpt home page. **http://www.xcerpt.org/**
[Xcerpt1]	François Bry, Sebastian Schaffert. A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML. **http://www.pms.informatik.uni-muenchen.de/publikationen/#PMS-FB-2002-11**
[XPath]	J. Clark and S.DeRose, eds. XML Path Language (XPath) Version 1.0. W3C Recommendation, 16 November 1999 **http://www.w3.org/TR/xpath**
[XPath2.0]	A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, J. Siméon. XML Path Language (XPath) 2.0, W3C Working Draft, 30 April 2002. **http://www.w3.org/TR/xpath20**
[XPointer]	S. DeRose, E. Maler, R. Daniel Jr., eds. XML Pointer Language (XPointer) Version 1.0. W3C Candidate Recommendation, 11 September 2001 **http://www.w3.org/TR/xptr**
[XQuery]	S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, M. Stefanescu, eds. XQuery 1.0: An XML Query Language. W3C Working Draft, 30 April 2002. **http://www.w3.org/TR/xquery**
[XQD]	M. Fernández, J. Marsh, M. Nagy, eds. XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft, 30 April 2002. **http://www.w3.org/TR/query-datamodel/**
[XQFS]	D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, P. Wadler, eds. XQuery 1.0 Formal Semantics. W3C Working Draft, 26 March 2002. **http://www.w3.org/TR/query-semantics/**
[XQUC]	D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, J. Robie, eds. XML Query Use Cases. W3C Working Draft, 30 April 2002. **http://www.w3.org/TR/xmlquery-use-cases**
[XQueryX]	A. Malhotra, J. Robie, M. Rys, eds. XML Syntax for XQuery 1.0 (XQueryX). W3C Working Draft, 7 June 2001. **http://www.w3.org/TR/xqueryx**
[XSL]	The Extensible Stylesheet Language (XSL). Document Format domains of W3C. **http://www.w3.org/Style/XSL/**
[XSL-FO]	S. Adler, A. Berglund, J. Caruso, S. Deach, T. Graham, P. Grosso, E. Gutentag, A. Milowski, S. Parnell, J. Richman, S. Zilles, eds. Extensible Stylesheet Language (XSL) Version 1.0. W3C Recommendation, 15 October 2001.
[XSLT]	J. Clark, ed. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, 16 November 1999 **http://www.w3.org/TR/xslt**
[XSLT2.0]	M. Kay, ed. XSL Transformations (XSLT) Version 2.0. W3C Working Draft, 30 April 2002 **http://www.w3.org/TR/xslt20**

Main: 4IndexingXMLData

XML and Databases: 4 Indexing XML Data

All refrences to this chapter are given in **[Wei]**.

4.1 Basics: Tag and/or Keyword Indexing

Goal of indexing methods: define a data structure (i.e. physical organization of data + primitive operations for data structure creation, data storage, and data retreival) so as to make the data retreival more efficient.

Obviously, the design or choice of a convenient indexing method depends on the kind of data retrieval.

Srarting point of indexing method for XML and senmistructured data: Indexing methods for textual data (or document). The name of this field is "information retrieval". The emphasis is on "unstructured texts", i.e. texts the structure of which (like titles, chapters, etc.) is not (or almost not) taken into account. Retrieval aimed at: text retrieval depending on the textual content reflecting the semantics).

4.1.1 Word indexes

Word indexes very much like indexes for human readers: Words from the text collected and the occurrences of a word in the text (expressed as e.g. address) is given in regard to the word. The following echniques are used for reducing the size of the index and increasing its value:

- Stemming: Instead of various morphological variations of a same stem, the stem is collected in the index. In English, satemas mostly are word prefixes, which makes stemming easier.
- Stop word lists: Some words, in general so-called "stop words" or function words such as preposition or articles, or more generally words occurring too frequently in the documents concerned for discriminasting between these documents, are excluded from the word index.
- Weighting: Words ("keywords") retained after application of stemming and/or filtering out of elements of stop word lists are assigned weights expressing their importance for discriminating between the documents concerned. A keyword weight might either be constant for all the contexts (i.e. documents) it occurs (one spekas of "plain word weight"), or vary depending on the context ("word-document weight"). E.g. a word document weight might take the same of the documents into account (e.g. term (i.e. term) frequency x inverted document frequency").

4.1.2 Signatures

A signature is an apporximate represenbtation of an index requiring less memory space than the (full, exact) index. Signatures (for a word index) are bit strings of a uniform length, each encoding (preferably a unique) one word. The signatures of all keywords occurring in a document are combined by bitwise disjunction, yielding the document signature. Obviously, a document signature might be ambigous, i.e. might not uniquely/exactly characterise the keywords occurring in a document.

4.1.3 Index Structures

Index structures are physical organisation of an word index having certain advantage for e.g. data retrieval.

4.1.3.1 Inverted Files

Inverted files are simoilar to indexes found in books for human readers: In regards with each keyword, they store the list of references of its occurrences in the document. The occurrences might be refered to ase.g. (memory) page addresses, byte offsets.

4.1.3.2 Document/Term Matrix

A document/term matrix is a two-dimensional matrix, the one dimension of which represents the document collection, the other dimension the set of keywords. A matrix entry contains a 1 (0, resp.) if the keywords occurs (does not occur, resp.) in the document. A document/term matrix improves over an inverted file inasmuch that it has two (instead of a single) look-up pattern: keyword of document (instead of keyword only). However, for large collections of doicuments and/or hightly discriminating keywords, document/term matrices are sparse.

4.1.3.3 Trie

A trie (read "try" for avoiding confusions with "tree") is an edge labelled tree aiming at the retrieval of keywords over an alphabet. E.g.

```
    d i e        (Warning: lalbels on edages)
    i n d e e d
            x
.
    m i d i
        n
    x m l
      c e r p t
```

Nodes might be annotated to references to occurrences of the keywords they represent.

4.1.3.4 PATRICIA Trie

PATRICIA tries enhance ordinary tries with string compression. TThe idea is to avoid chains of index nodes with only one outgoing edge (like

```
      i n d e
```

in the previous example). Such chains are not needed for dismabiguation and can therefore be compacted. E.g., in the previouis exap+mle, the chain

```
      i n d e
```

can be replaced by thje single node i, for (in this example) all keywords beginning with i have the prefix "inde". Such a compression can be performed not only for keyword prefixes, but at every node of an ordinary trie. In oder to prperly distingush between characher sequences, nodes following such a compacting are lalbellled with the length of the compacted chain (i.e. number of skipped characters + 1).

4.1.3.5 Suffix Tree

A suffix tree is a trie build over the suffixes of a set of (key)words. Note that a suffix tree contains all possible suffixes of the considered set of keywords, not only the keywords. Suffix trees are usuefull with languages like English in which most words are derived from a stem by suffxing a string.

4.1.3.6 PAT Tree

A PAT tree can be defined as the PATRICIA tree which can be obtained from a suffix tree.

4.2 Element Indices

Recall that semistructured data means data that may contain both, unstructured texts and structred texts. For such data, word indexes are not fully satisfying, for they do not consider the structure at all.

Element Indices (a name of ours) are methods for indexing element names (i.e. tags).

4.2.1 Text index

A text index indexes keywords occurring in the (textual) content of an element. Thus, a text index maps a keywords to (references of) the elements in which it occurs.

A text index might be restricted to "direct text containment", i.e. refer only the lowest elements (or nodes) in the document tree that contain the keyword (but does not refer to their ancestors).

This restriction is harmful is there is no other mean to (efficiently) check for ancestor-descendant relationships between elements.

A text index have been implemented using inversted files, tries, or B-trees. Often, linguistic normalization techniques such as stemmingt and stop word lists are used to reduce the number of keywords occurring in the text index.

4.2.2 Value Index

The value index is a text index augmented with "data type"-specific matching operations, e.g. "contains" for the type string, < for the type indeger. In the first case, an index similar to a text index is built giving (references to occurrences of) strings that are contained in some selected strings. In the second case, an index similar to a text index is built giving

(references to occurrences of) integers that are strictly smaller than some selected integers.

Like text indices might be restricted to direct containment, value indices for transitive relations such as string containment or < for integer might be restricted to maximal (minimal resp.) values.

Techniques referring to "type corersion" have been investigated for handling ambiguously typed values in value indices.

4.2.3 Label Index

A label index is an index (e.g. an inverted file) for element labels.

4.2.4 Combined Vocabulary

A combined vocabulary is a combination of text, value, and/or label indices, so as to avoid to have to join over several indices when the type (text, label, value, etc.) of the object search for isunknown or partly known. In a combined vocabulary, the (refetences to) different categories of objects are attached to each keyword, for each object, its category is given. This is done either by typing the (refetences to) objects, or by using a hierarchical structure: The index maps each keywords to the different categories of objects it occurs in, each category maps to the (references to the) objects of that category containing the keyword.

4.2.5 Parent/Child Index

Parent/child indices are used for path reconstruction. A parent/child index is an index relating each parent to its children.

4.2.6 An Application of Element Indices: XISS

The XML Indexing and Stroage System (XISS) combined various element indices for an efficient evaluation of regular path expressions. It relies upon:

- a so-called element index: a label index
- an attribute index: like a label index but for attributes. With look-up to a valaue table for the attribute values.
- a structure index: a parent/child index. With a EA-join for computing joins either on element and/or attribute nodes.
- an ancestor/descendant index: A method called EE-jointo similar EA-join for determining which lements of a set of elements are ancestors of nodes from another set of elements.
- a method called KC-join for evaluatuing the regular path expression constructor Kleene closure (or star).

4.3 Path Indices

Since most data selection languages (such as regular path expressions, XPath, and XPointer) as well as established query and transformation languages (such as XSLT and XQuery) build upon path expressions for data retrieval, methods have been developed for indexing paths through trees and rooted (acyclic and/or cyclic) graphs.

4.3.1 Annotated Path Indices

4.3.1.1 Element Locator Scheme

Table matching keywords to the simple (i.e. no advance construct from regular paths) label paths by which it can be reached. Since there are in general several elements (nodes) with the same path, the Element Locator Scheme also maps to the sibling (ordinal) number of the element.

4.3.1.2 CIS Index

Like the Element Locator Scheme, the CIS Index matches keywords to the simple label paths by which it can be reached. For dismabiguing, the CIS Index uses unique node identifiers ((symbolic) node addresses).

4.3.2 Context Index

The Context Index is similar to the CIS Index but also includes a structure filter.

The path along which a keyword (or label) can be reached is encoded as a bit string built as follows: a unique position for each label, 1 (0, resp.) at this position if the label is present (absent, resp.) in the path) called "path signature". Note that path signatuires in general yieled no exact retrieval.

Techniques have been investigated for reducing the storage needed by a context index.

4.3.3 BitCube

The BitCube is a three-dimensional bitmap index adding as a third dimension to the document/term matrix the set of simple paths in all documents of the document collection considered.

Advanced clustering techniques have been developped for the BitCube so as to efficiently store sparce BitCubes.

Also a notion of similarity between documents based on similarity of paths has been developped which builds upon the BitCube.

4.4 Navigation Indices: Data Guides, Signature Files, and Improvements Thereof

Navigation Indices aim at an efficient evaluation of complex path expressions like regular path expressions and XPath expressions. Navigation indices usually rely on so-called schema trees or graphs, i.e. trees the edges of which a labelled with the labels occurring in the considered documents, the nodes of which are labelled with (references to) nodeds of the considered documents. If the document considered are general (rooted) graphs imstead of trees, ie.e. might contained (directed or undirected) cycles, schema graphs usually can be built in different manners that affect the performances of the index methods.

4.4.1 BUS

The BUS (Bottom-Up Scheme) index supports structure as well as content retrieval and yields ranked results. It combines the following three data structures:

- schema tree: A tree the edges of which are labelled by the element labels as they occur in database items, the node of which are labelled with so-called "element path numbers" that can be seen as simple path identifiers (cf. picture p. 61).
- content index: maps each keyword to the (references of the) nodes they (immediately) occur in, to the number of theitr occurrences in the document, and to the identifiers (element path numbers) of the simple paths they are reachable from.
- accumulator table: table used and modified during look-up for ranking the returned documents. Basically, during a look-up, the accumulator table for maps each (reference of a) node to the (possibly weighted) number of hits so far encountered.

A look-up for a e.g. the XPath expression a/b/c["xml"] is performed as follows. The schema tree yields the relevant element path numbers. These are used with the keyword "xml" for finding in the content index the (references to the) relevant nodes. The accumaulator table is used for ranking the results.

The BUS index explicitely preculdes documents with mixed content (i.e. requires that all text appear in leaves), but this restriction does not seem to be necessary for the approach.

4.4.2 Signature File Hierarchy

The signature file hierarchy combines signature files and tries for indexing both, the structure of documents and keywords. It relies on a schema tree (liked the BUS index) and on bit strings ("signatures") to represent both, keywords and "contexts" (i.e. simple paths).

Using standard Information Retrieval techgniques (cf. 4.1), discriminating keywords are selected for the documents considered. Each keyword is assigned a signature, i.e. is encoded as a bit string of fixed length. The set of keyword occurring in a leaf is encoded as the bitwise disjunction of the signatures of the keywords the leaf contain. An inner node is assigned a signatures obtained as the bitwise disjunction of the signatures of the nodes this inner node contain. Thus, the signature of a leave or inner node indicate which keywords might occur (but not necessarily do occur) in the node.

The Signature File Hierarchy is a a schema tree similar to that of the BUS index the node of which are not labelled with (references to) document nodes, but by pairs consisting of the (reference to a) document node and the signatures of this nodes. The prfesence of the nodes signatures give rise to select only those nodes possibly containing the keywords search for while evaluating regular path expressions such a the XPath expressions a/b/c["xml"] or a//c["xml"].

In the same way as signatures are assigned to document nodes, signatures can also be assigned to the nodes of a query tree. The resulting data structure is called Query Signature Hierarchy.

A look-up consists in matching the Query Signature Hierarchy (of the considered query) with the Signature File Hierarchy in a depth-first traversal (of both trees).

For efficiency reasons, trie-based representations of the signature hierachies have been proposed.

4.4.3 DataGuide

The Data Guide is an index structure for both, tree-shaped and (rooted) graph-shaaed documents. It relies on a schema graph built in different manners in case of cycles: week and strong DataGuide. (cf. Figure on p. 70). It might be used fvor both, query ebvaluation and schema-.browsing.

Basically, a DataGuide for a collection of documents is a graph the edges of which represent any possible edges of a (graph representing a) document of the considered collection , the nodes of which are labelled by (references to) the nodes of the considered documents. Thus, a DataGuide is a schema graph.

Some DataGuides might yiled simple paths leading to wrong node references, i.e. document nodes that cannot be reached (in the documents) by such a path (cf. figure 6.11 (p. 70). Such DataGuides are called weak and DataGuides that are not weak ate called strong. Methods have been proposed for generating strong DataGuides from collecteions of (rooted graph-shaped) documents thaqt have been shown to correspond to generating a deterministic finite automaton from a non-deterministic one.

Like schema graphs, strong DataGuides might grow exponentially (in the number of document nodes, edges, and node references).

DataGuides have been often used in repositories of (i.e. query and management systems) for semistructured data such -as LORE and ToX.

4.4.4 T-Index

The T-Index is a non-deterministic structure index for both tree- and graph-shaped data. It generalizes and improves the (weak) DataGuide. The generalization consists in supporting relative path queries instead of only absolute path queries.

A T-Index conasists of a 1-Index indexing absolute paths, and a 2-Index indexing relative paths. The 1-Index is similar to a (weak) DataGuide with the difference that it grows linearly (instead of exponentially) in the size of the considered collection of documents in the number of document nodes, edges, and node references).

Instead of all simple paths, path templates are considered making it possible to index a subset of simple paths called "priviledged paths". Basically, the 1-Index builds a schema graph the nodes of which are labelled by the (references to the) document nodes that are equivalent in the sense that they are reachable by the same set of simple paths. Note that two nodes that are both reachable by only one common simple path are not necessaqrily equivalent in the sense considered here.

A 2-Index can be seen as an index for ancestor/descendant relationships between the nodes of the documents considered. Like with the 1-Index, the 2-Index builds a graph the of which are labelled by the (references to the) document nodes that are equivalent in the sense that they have the same descendants along the same set of simple paths.

4.4.5 IndexFabric

The IndexFabric indexes both path and content of tree-shaped data in a balanced hierarchy of PATRICIA trees. The hierarchy result from splitting a PATRICIA tree in blocks of similar sizes amainalbe to efficient paging.

4.4 References

[Wei] Felix Weigel.
A Survey of Indexing Techniques for Semistructured Documents. Institute for Computer Science, University of Munich, 2002.
http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/master.ps.gz (1.3M),

http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/master.pdf (4.8M),
http://www.pms.informatik.uni-muenchen.de/publikationen/projektarbeiten/Felix.Weigel/master.pdf.gz(3.7M)

Main: 5XcerptAQueryAndTransformationLanguageForWebAndSemanticWebApplications

XML and Databases: 5 Xcerpt: A Query and Transformation Language for Web and Semantic Web Applications

1 Xcerpt's Principles

Xcerpt **[Xcerpt1]** **[Xcerpt2]** is an experimental query and transformation language developed at the Institute for Informatics of the University of Munich since 2002.

The goal of the Xcerpt project is to investigate ways to ease realizing Web as well as Semantic Web applications. Three main directions for research are currently followed: (1) design of high-level language using which programming Web queries is made easier and (2) both conventional Web and "Semantic Web" queries can conveniently be expressed, (3) conception of an efficient run-time system.

The "Semantic Web" is an endeavor which Tim Berners-Lee, the inventor of HTML and of HTTP, James Hendler, and Ora Lassila initiated in 2001 with an article in the Scientific American **[SemWeb]**. The "Semantic Web" vision is that of the current Web which consists of (X)HTML and XML documents being extended with meta-data specifying the meaning of these (X)HTML and XML documents in forms usable by both, human beings and computers:

The Semantic Web will bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users. **[SemWeb]**

One might see the Semantic Web meta-data added to today's Web as semantic indexes similar to encyclopedias. A considerable advantage over conventional encyclopedias printed on paper is that the relationships expressed by Web meta-data can be followed by computers, very much like hyperlinks can be followed by programs, and be used for drawing conclusions using automated reasoning methods:

For the Semantic Web to function, computers must have access to structured collections of information and sets of inference rules that they can use to conduct automated reasoning. **[SemWeb]**

Three formalisms have been developed for Semantic Web meta-data: RDF **[RDF]**, Topic Maps (cf. e.g. **[TM1]** and **[TM2]** for introductions) and OWL **[OWL]**. RDF and Topic Maps give rise to expressing binary relationships like "a text is authored by some person", OWL is derived from description logic and gives rise to expressing not only binary relationships but also relationships of arbitrary arities and to state properties of relationships such as the commutativity and/or transitivity of a relation expressing "$person_1$ is a friend of $person_2$".

A central principle of the Xcerpt language is the working hypothesis that a common query language capable of inference to be used for querying both the conventional Web and the Semantic Web is desirable and possible. This working hypothesis is one of the salient features of Xcerpt which makes it different from all Web as well as Semantic Web query languages developed so far. At the end of this section (cf. 1.8 **A Query Language for both, the Standard Web and the Semantic Web**), arguments are given for this working hypothesis.

Xcerpt's further principles are as follows.

1.1 Referential Transparency and Answer-Closedness

1.1.1 Referential transparency

This property means that, within a definition scope, all occurrences of an expression have the same value, i.e. denote the same data. Referential transparency is an essential, precisely defined trait of the rather vague notion of "declarativity": while declarativity is an intuitive notion which possibly cannot be precisely defined, referential transparency is a clearly defined notion. Referentially transparent programs are easier to understand and therefore easier to develop, to maintain, and to optimize. Referential transparency surely is one of the essential properties a query language for the Web should satisfy.

1.1.2 Answer-closedness

Let us call "answer-closed" a query language such that replacing a subquery in a compound query by possible (not necessarily actual) answers always yields a syntactically valid query. Answer-closed query languages ensure in particular that every data item, i.e. every possible answer to some query, is a syntactically valid query. Functional programs can -- but must not -- be answer-closed. Logic programming languages are answer-closed but SQL is not. Answer-closedness eases the specification of queries because it keeps limited the unavoidable shift in syntax from the data sought for, i.e. the expected answer, and the query specifying these data.

Xcerpt is answer-closed. Xcerpt elementary queries can be seen as build-up from possible answers with variables replacing some data and additional constructs added.

1.2 Answers as Arbitrary XML Data, Answer Ranking and Top-k Answers

1.2.1 Answers as arbitrary XML data

XML is the *lingua franca* of data interchange on the Web. As a consequence, answers should be expressible as every possible XML application. This includes both text without markup and freely chosen markup and structure. This requirement is obvious and widely accepted for conventional Web query languages. Semantic Web query languages, too, should be capable of delivering answers in every possible XML application so as to make it possible, e.g. to mediate between RDF and XTM data or to translate RDF data from one RDF syntax into another RDF format.

1.2.2 Answer Ranking and Top-k Answers

In contrast to queries posed to most databases, queries posed on the conventional and Semantic Web might have a rather unpredictable number of answers. As a consequence, it is often desirable to rank answers according to some application-dependent criteria. It is desirable that Web and Semantic Web query languages offer (a) basic means for specifying ranking criteria and, for efficiency reasons, (b) evaluation methods computing only the top-k answers (i.e. a given number k of best-ranked answers according to a user-specified ranking criterium).

Xcerpt supports the specification of orders on XML documents and the retrieval of k answers, possibly sorted according to a specified order, to a query.

1.3 Pattern Queries

Xcerpt uses patterns for binding variables in query expressions instead of path expressions -- as do e.g. the Web query languages XQuery and XSLT. Query patterns are especially well-suited for a visual language because they give queries a structure very close to that of possible answers. One might say that query patterns are like forms, answers like form fillings.

1.4 Incomplete Query Specifications

Incomplete queries specifying only part of data to retrieve, e.g. only some of the children of an XML element (referring to the tree representation of XML data called "incompleteness in breadth") or an element at unspecifed nesting depth (referring to the tree representation of XML data called "incompleteness in depth") are as important on the conventional Web because of its heterogeneity: one often knows part of the structure of the XML documents to retrieve.

Incomplete queries specifying only part of data to retrieve are also important on the Semantic Web. There are three reasons for this. First, "Semantic Web data" such as RDF or Topic Map data might be found in different (XML) formats that are in general easier to compare in terms of only *some* salient features. Second, the merging of "Semantic Web data", i.e. of topic maps, is often done in terms of components common to distinct data items. Third, most Semantic Web data standards such as RDF and OWL have data items with optional components. In addition, query languages for the conventional and Semantic Web should ease retrieving only part of (completely or incompletely specified) data items.

Xcerpt supports queries that are incomplete in the following senses:

- incomplete in breadth
- incomplete in depth
- incomplete with respect to element order
- incomplete because of optional elements or attributes

1.5 Incomplete Data Selections

Because Web data are heterogeneous in their structures, one is often interested in "incomplete answers". Two kinds of incomplete answers can be considered. First, one might not be interested in some of the children of an XML (sub)document retrieved by a query. Second, one might be interested in some children elements if they are available but would accept answers without such elements.

An example of the first case would be a query against a list of students asking for the name of students having an email address but specifying that the email address should not be delivered with the answer.

An example of the second case would be a query against an address book asking for names, email addresses, and if available cellular phone numbers.

Xcerpt's construct **xcept** gives rise to discard a child of an element retrieved by a query, i.e. to express queries of the first kind.

Xcerpt's construct **optional** gives rise to select elements only if available, i.e. to express queries of the second kind.

Provided queries are evaluated on the Web sites of the XML documents that are queries, both Xcerpt constructs give rise to reducing data transfer.

1.6 Rule-Based, Chaining, and Recursion

1.6.1 Rule-Based

Rules are understood here as means to specify novel, maybe virtual data in terms of queries, i.e. what is called "views" in (relational) databases, regardless of whether this data is materialized or not. Views, i.e. rule-defined data are desirable for both, conventional and Semantic Web applications. There are three reasons for this.

First, view definitions or rules are a means for achieving the so-called "separation of concern" in query programs, i.e. means for stepwise specifications of data to retrieve and/or to construct. In other words, rules and view definitions are a means for "procedural abstraction", i.e. rules (view definitions, resp.) are the Prolog and Datalog (SQL, resp.) counterpart of functions and/or procedures.

Second, rules and view definitions give rise to easily specifying inference methods needed e.g. by Semantic Web applications.

Third, rules and view definitions are means for "data exchange". Data exchange means translating data from different sources into a common format. Data mediation is needed both on today's Web and on the emerging Semantic Web because of their heterogeneity. Data mediation is especially needed on the Semantic Web because of the many (about 20) standards for RDF data.

1.6.2 Backward and Forward Chaining

On the Web backward chaining, i.e. computing answers starting from rule heads, is in general preferable to forward chaining, i.e. computing answers from rule's bodies. Indeed, backward chaining gives rise to propagating restrictions from the goals (or main queries) to the (sub)queries posed to Web sites.

1.6.3 Recursion

On the Web recursion is needed

- for querying on the standard Web when complex transformations are needed
- for querying on the Semantic Web when inference rules are involved.

Note that a free recursion is often desirable and that recursive traversals of XML documents as offered by the recursive computation model of XSLT are not sufficient.

Xcerpt supports (unrestricted) recursion on possibly cyclic data (relying on a so-called "memoization" or "tabulation" technique).

1.7 Separation of Queries and Constructions

Two standard and symmetrical approaches are widespread, as far as query and programming languages for the Web are concerned:

- queries or programs are embedded in a Web page or Web page skeleton giving the structure of answers or data returned by calls to the programs
- part of a Web page specifying the structure of the data returned to a query or program evaluation are embedded in the queries or programs.

It is a thesis of the Xcerpt project that both approaches to queries or programs are hard to read (and, therefore, to write and to maintain).

Instead of either approach, Xcerpt strictly separates queries and "constructions", i.e. expressions specifying the structure of answers. With Xcerpt, constructions are rules' heads and queries are rules' bodies. In order to relate a rule's construction, i.e. the rule's head, to a rule's query, i.e. the rule's body, Xcerpt uses (logic programming) variables.

1.8 A Query Language for both the Standard Web and the Semantic Web

A thesis underlying the Xcerpt project is that a *common* query language for both conventional Web and Semantic Web applications is desirable.

There are two reasons for this thesis.

First, in many cases, data are not inherently "conventional Web data" or "Semantic Web data". Instead, this is a usage that gives data a "conventional Web" or "Semantic Web" status. Consider for example a computer science encyclopedia. It can be queried like any other Web document using a Web query language. If its encyclopedia relationships (formalizing expressions such as "see", "see also", "use instead" commonly used in traditional encyclopedia) are marked up, e.g. using XLink or any other *ad hoc* or generic formalism as one might expect from an online encyclopedia, then the encyclopedia can also be used as "Semantic Web data", i.e. as meta-data, in retrieving computer science texts (e.g. the encyclopedia could relate a query referring to "Linux" to Web content referring to "operating systems of the 90es") or enhance the rendering of Web contents (e.g. adding hypertext links from some words to their definitions in the encyclopedia).

Second, most likely, Semantic Web applications will combine and intertwine queries (and updates) to Web data and to meta-data (or Semantic Web data) in all possible manners. There is no reason to assume that Semantic Web applications will rely only on meta-data or that querying (or updating) of conventional Web data and Semantic Web data will take place in two (or a few) successive querying phases referring each to data of one single kind, "conventional Web data" or "Semantic Web data". Consider again the computer science encyclopedia example. Instead of one single encyclopedia, one might use several encyclopedias that might be listed in a (conventional Web) document. Retrieving the encyclopedias requires a conventional Web query. Merging the encyclopedias is likely to call for specific features of a Semantic Web query language. Enhancing the rendering of a conventional Web document using the resulting (merged) encyclopedia is likely to require (a) conventional Web queries (for retrieving conventional Web documents and the addresses of the relevant encyclopedias), (b) Semantic Web queries (for merging the encyclopedias), and (c) conventional and Semantic Web queries (for adding hypertext links from words defined in the (merged) encyclopedia).

1.9 Specific Reasoning as Theories

Many practical applications require special forms of reasoning. E.g. for efficiency reasons, equality reasoning is often performed using the so-called paramodulation rule instead of the equality axioms (transitivity, substitution, and symmetry), temporal data might require conversions between different time zones and/or calendar systems that are expressed in a simpler manner and more efficiently performed using arithmetic instead of logical axioms, or reasoning with intervals of possible values instead of exact values, e.g. for appointment scheduling, is conveniently expressed and efficiently performed with constraint programming.

For this reason, it is desirable that a query language for the (conventional and Semantic) Web can be extended with so-called "theories" implementing specific forms of reasoning.

Such "theory extensions" can be realized in two manners.

First, a theory can be implemented as an extension of the run time system of the query language, additional language constructs giving rise to use the extension.

Second, a theory can be implemented using the query language itself and made available to users of this query language through program libraries.

In the Xcerpt project, both approaches are investigated. E.g. a temporal and calendric theory is currently implemented as an extension of Xcerpt's run time system and an Xcerpt program library for querying and reasoning with RDF meta-data is developed.

1.10 Two Syntaxes: XML and Compact Syntax

While it is desirable that a query language for the (conventional and/or Semantic) Web has an XML syntax, because it makes it easier to exchange query programs on the Web and to manipulate them using the query language, a second, more compact syntax easier for human beings to read and write is desirable.

Therefore, Xcerpt has two syntaxes: an XML syntax which is purely term-oriented and another one which combines term expressions with non-term expressions like most programming languages.

Both syntaxes are interchangeable (translation is a low cost process). In the following, only the "second" or "compact syntax" , i.e. the non-XML syntax, is introduced.

2 Xcerpt's Core Constructs

The constructs of the language Xcerpt can be divided into core constructs and advanced constructs. The core constructs, which are introduced in this section, are those constructs that are necessary for structural querying and construction. This basically includes query and construction patterns, rules and goals.

Advanced constructs (cf. below **3 Xcerpt's Advanced Constructs**) are constructs that are not strictly necessary for the pure task of querying and rearranging data, but make Xcerpt suitable and convenient for practical use. This includes, among others, constructs like functions and aggregations, negation and regular expressions.

An Xcerpt program consists of at least one goal and of some (maybe zero) rules. Goals and rules are composed of data, query and construct terms representing respectively XML documents, query and XML documents constructed from the answers to queries.

2.1 Data, Query, and Construct Terms

Common to all terms is that they represent tree-like (or graph-like) structures. Square brackets (i.e. [ ]) denote ordered term specification (as in standard XML), i.e. the matching subterms in the queried resource are required to be in the same order as in the query term. Curly braces (i.e. { }) denote unordered term specification (as it is common in databases), i.e. the matching subterms in the queried resource may be in arbitrary order.

Single (square or curly) braces (i.e. [ ] and { }) are used to denote that a matching term must contain matching subterms for all subterms of a term and may not contain additional subterms (total term specification). Double braces (i.e. and {{ }}) are used to denote that the data term may contain additional subterms as long as matching partners for all subterms of the query term are found (partial term specification).

Non-tree graph structures are expressed using a reference mechanism. The construct id @ t is a defining occurrence of the identifier id and the construct ^id is a referring occurrence.

2.1.1 Data Terms

Data terms are used to represent XML documents ("XML in disguise") and the data items of a semistructured database. They are similar to ground functional programming expressions and logical atoms. Data terms may only contain the single square and curly braces described above, but no partial specifications, as a data item is always considered to be complete.

In this context, a "database" is a (multi-)set of data terms (e.g. the Web). Note, however, that a single data term is often used to represent what is commonly referred to as a "database": an address book which might contain a lot of entries can be represented in a single data term (or XML document), whereas a relational database would store each entry as a separate item.

The following data term represents an article in Xcerpt syntax. For obvious reasons, most parts are omitted. Note that some parts of this document use unordered term specification (e.g. the author entries), as the order is irrelevant.

Example:

```
paper [
  front [
     title [ "Querying the Web Revisited" ],
     author {
        fname [ "Sebastian" ],
        surname [ "Schaffert" ],
        address { ... }, bio [ ... ]
     },
     author {
        fname [ "François" ],
        surname [ "Bry" ],
        address { ... },
        bio [ ... ]
     },
  ],
  body [
     section [
        title [ "Introduction" ], ],
        ...
  ],
  rear [
      acknowl [ ... ],
      bibliog {
           bibitem [
              bib [ "XQuery" ],
              pub [ "XQuery: The XML Query Language ..." ]
           ],
           ...
      }
  ]
]
```

XML attribute-value pairs are represented in Xcerpt data terms as a special kind of subterms. An XML attribute-value pair of the form **name="value"** is represented as a subterm of the form **name{"value"}**. So as to distinguish proper subterms from attributes, all attribute-value pairs are grouped inside a subterm with the reserved label **attributes**, which has to appear as the first subterm of a term (regardless of whether the term is unordered or ordered).

The XML fragment on the top of the following example (describing a figure with a unique id attribute) corresponds to the data term on the bottom of the example:

Example:

```
<figure id="fig:graph">
    <graphic scale="80" figname="figure1"/>
</figure>

figure [
     attributes { id{"fig:graph"} },
     graphic [
          attributes { scale{"80"}, figname{"figure1"} }
     ]
]
```

Note that data terms of the current version of Xcerpt do not cover all constructs found in XML. Constructs like processing instructions are intentionally left out because they do not add important information to the data represented. Adding such XML constructs to Xcerpt's current version would be an easy task.

Xcerpt deliberately offers a non-XML syntax. Experience with XSLT shows that many programmers are uncomfortable with expressing programming language constructs in a very verbose XML notation, and programs arguably become cluttered and difficult to grasp merely due to the verbose notation. The term syntax used in Xcerpt is more lightweight, and can be easily transformed into XML and vice versa using a simple lexer (with a stack to record the nesting level for the closing tags). Such a transformation can even be performed transparently by an advanced editor, like in the language visXcerpt **[visXcerpt1]** **[visXcerpt2]** **[visXcerpt3]**.

2.1.2 Query Terms

Query terms are partial patterns that are matched with data terms, augmented by an arbitrary number of variables for selecting data items from a data term. In addition to the constructs used in data terms, query terms have the following additional properties:

- *partial specifications* omitting subterms irrelevant to the query are possible (indicated by double square brackets or curly braces **{{ }}**),
- it is possible to specify subterms at *arbitrary depth* (indicated by the keyword **desc**).
- they may contain *term variables* and *label variables* to "select" data from a database (variables are written in upper case letters below)

Thus, query terms are similar to *non-ground* functional programming expressions and logical atoms.

The Xcerpt construct **X -> t** (read "X as t") serves to associate a query term to a variable, so as to specify a restriction of its bindings. The Xcerpt construct **desc** (read "descendant") is used to specify subterms at arbitrary depth.

Suppose that all articles of a conference are contained in a proceedings element. The following query term selects title and author pairs for each article:

Example:

```
proceedings {{
   article {{
       front {{
            var T -> title {{ }},
            var A -> author {{ }}
       }}
   }}
}}
```

The reference mechanism using **^id** and **id @ t** is resolved internally when a query is evaluated. Programmers thus do not need to distinguish between usual term-subterm relationships and references, they are conceptually treated in the same manner.

Query terms (in general containing variables) are *unified* with data or construct terms (in which variables may occur) using a non-standard unification called *simulation unification* which is described below (cf. **4 Simulation Unification**). Simulation unification is based on graph simulation **[ABS]** which is a relation similar to graph homomorphisms.

The outcome of unifying a query term with a data term (construct term, resp.) is a set of substitutions for the variables in the query term (construct term, resp.), where each substitution represents an alternative solution. In case the substitutions result from unifying a query term with a data term, applying each of these substitutions to the query term yields a ground query term which is simulated (in the sense of **[ABS]**) in the data term.

2.1.3 Construct Terms

Construct terms serve to reassemble variables (the bindings of which are specified in query terms) so as to construct new data terms. They may only contain single brackets (i.e. **[ ]** or **{ }**) and variables, but no partial specification (i.e no double braces or **{{ }}**) or variable restrictions (i.e. **x -> t**). The rationale of this is to keep variable specifications within query terms, ensuring a strict separation of purposes between query and construct terms.

The following construct term creates an Author-Title pair wrapped in a "result" element:

Example:

```
result {
     var A, var T
}
```

In a construct term, the Xcerpt construct **all t** serves to collect (in the construct term) all instances of **t** that can be generated by alternative substitutions for the variables in **t** (returned by the associated query terms in which they occur). Likewise, **some n t** serves to collect at most **n** instances of **t** that can be generated in the same manner.

In the following construct term, the **all** construct is used to create a list of publications for each author:

Example:

```
results {
    result {
        var A,
        all var T
     }
 }
```

If several variables are in the scope of an `all` construct, all combinations that are justified in a query are listed. The following construct term collects all title/author pairs for the previous query:

Example:

```
results {
    all result { var A, var T }
}
```

The constructs **all** and **some n** may be nested to form more complex results. Assuming the previous query, the following construct term collects all titles for each author:

Example:

```
results {
     all result { var A, all var T }
}
```

Positioning the nested **all** around the variable **A** yields "all authors for each title" as a result:

Example:

```
results {
    all result { all var A, var T }
}
```

2.2 Queries

A *query* is a connection (using **AND**s and **OR**s) of query terms. Empty queries are possible. A query is always (implicitly or explicitly) associated with a resource. A resource may be the program itself, an external Xcerpt program or an (XML or other) document specified by a URI (uniform resource identifier).

Variables occurring in more than one query term in an **and** connected query evaluate similar to an equijoin in the language SQL (for relational databases).

The following query selects all such authors that published a paper in the proceedings of both 2003 and 2004. It is assumed that the articles are all contained in a proceedings03 resp. proceedings04 element:

Example:

```
and {
    in { resource { "file:proceedings03.xml" },
        desc author {{
            fname { var First }, surname { var Last }
        }}
    },
    in { resource { "file:proceedings04.xml" },
        desc author {{
            fname { var First }, surname { var Last }
        }}
    }
}
```

If a query does not explicitly have an associated resource, the resource specification is implicit and inherited from the parent. If none of the parents have a resource specification, or the query does not have a parent, the queried resource is the program itself (i.e. the heads of the rules and possibly data terms contained in the program - see "Rule Chaining" below).

Note that it is possible to use curly and square braces in **and** and **or** connections to specify that the evaluation order is of importance (square braces) or not (curly braces). This may serve as an indication to the evaluation engine whether certain optimizations are applicable or not.

2.3 Contruct-Query Rules and Goals

An Xcerpt program consists of zero or more *construct-query rules*, one or more goals and zero or more data terms. Both rules and goals have the form

```
CONSTRUCT construct term
FROM query part
END
```

where a construct term is constructed depending on the evaluation of a query part.

A rule can be seen as a "view" (in the sense of databases) specifying how documents shaped in the form of the construct term can be obtained by evaluating the query part against a Web resource (e.g. an XML document or a database).

In addition to the form above, goals are always (explicitly or implicitly) associated with an output resource. This resource specifies where to "write" the resulting data terms. If not explicitly specified, the output resource defaults to stdout, writing all output to the console.

3 Xcerpt's Advanced Constructs

3.1 Functions and Aggregations

Many Web applications require to compute information in addition to arranging it in new structures, like adding the VAT (value added tax) to a price, calculate totals, concatenate strings, compute the average of all results, etc. Likewise, it is desirable to express conditions on such computed data in a cohesive manner, like "the price should be lower than 50". Such computations are in general expressed using *functions* and *aggregations*.

As both, normal functions and aggregations, are *constructing* new data, they are only allowed in construction parts of rules (i.e. the rule or goal heads).

The following Xcerpt rule counts the number of articles containing the keyword "XML" in the proceedings.xml file by using the aggregation function `count` in the construct part:

Example:

```
CONSTRUCT
    nr_of_articles {
        count ( all var Paper )
    }
FROM
    in {
        resource{ "proceedings.xml" },
        var Paper -> paper {{
            keywords {{
                 keyword { "XML" }
            }}
         }}
    }
END
```

3.2 Non-Pattern Conditions

One of Xcerpt's main design principles is pattern-based querying and many query conditions can be expressed in Xcerpt query terms by using partial patterns and variable restrictions. However, such conditions are limited to structural properties, like the parent-child or sibling relationships, and there are only few possibilities to work with the content (i.e. the semantics) rather than the structure of the data in such patterns.

Often such *semantic* conditions are desirable, e.g. to select all books that are published later than a certain date, etc. For such expressions, Xcerpt uses a so-called *condition box* which may be attached to query specifications (and parts of it) in rules by using the keyword **where**.

Assume the XML database of an online book store which might contain books with title, author(s) and price. The following Xcerpt program selects all books that are published in or later than 2003:

Example:

```
CONSTRUCT
    books {
        all var Book
```

```
    }
FROM
    in {
        resource{ "bib.xml" },
            bib {{
                var Book -> book {{
                    year { var Year }
            }}
         }}
    }
where {
    var Year >= 2003
    }
END
```

Conditions in this condition box only apply to the variables contained in the associated query specification, and there to each alternative binding separately, like the WHERE-part in SQL queries. As a consequence, there is no possibility to collect all alternatives using constructs like `all` or `some`.

3.3 Optional Subterms

One of the main characteristics of the Web and semistructured databases is that they are heterogeneous in structure. Nonetheless, data items of the same kind (e.g. address book entries or student information) are often very similar and differ only in that for certain entries some information is missing. For instance in an address book, the general structure of entries is very similar, but some entries might have no email address and others might have no telephone number, either because the respective persons do not have an email address/telephone number, or because the information is simply unknown.

Querying such entries with partial query patterns is very inconvenient when information is missing in some cases but is to be retrieved in the cases where it is available. For this reason, Xcerpt provides the possibility to specify subterms of a query term as *optional*, in which case the pattern matches even if the optional subterm does not have a match in the database.

Xcerpt uses the keyword **optional** to mark a subterm as optional. Occurrences of optional may also be nested. If a query term has a subterm that is marked as optional, this subterm usually also contains variables. Obviously, the optional construct yields cases where there are no variable bindings for these variables. For this reason, such variables must be marked as optional in the construction part of a rule as well.

The following Xcerpt program creates an HTML table listing all authors in the proceedings, with email address if available, or the string "not given" otherwise, sorted by last and first name:

Example:

```
CONSTRUCT
    table [
        tr [ td [ "Author Name" ], td [ "Email (optional)" ] ],
        all tr [
            td [ var First, var Last ],
            td [ optional var Email with default "not given" ]
        ] order by [Last,First] ]
FROM
    in {
         resource{ "proceedings.xml" },
         desc author {{
             fname { var First },
             surname { var Last },
             optional email { var Email }
         }}
    }
END
```

3.4 Excluding Subterms from Variable Binding: The except Construct

While incomplete query specifications (using **{{ }}** and) give rise to *positive* specifications of data to retrieve, the **except** construct gives rise to *negative* definitions, i.e. makes it possible to *exclude* data items with some (structural) properties.

The following example builds an HTML table only from those authors mentioned in the proceedings that have no email address:

Example:

```
CONSTRUCT
    table [
        tr [ td [ "Author Name" ], td [ "Email (optional)" ] ],
        all tr [
            td [ var First, var Last ],
        ] order by [Last,First] ]
FROM
    in {
         resource{ "proceedings.xml" },
         desc author {{
             fname { var First },
             surname { var Last },
             except email { var Email }
         }}
    }
END
```

Note the similarity between the queries of the last two examples: In the example of the previous section (Section 3.3), the email address is retrieved and included in the constructed HTML table if it exists, while in the example of the current section those authors with an email address are not selected. The similarity of expressions between both Xcerpt queries appropriately reflects natural languages specifications of the same queries.

3.5 Position Specification

In some applications it is desirable to query subterms only at a certain position while still being able to use partial query patterns, or to query the position of subterms in the database. For example, a query to this Extreme 2004 article could select the first and second element of all sections:

Example:

```
in {
    resource{ "proceedings.xml" },
    desc section {{
        position 1 var First,
        position 2 var Second
    }}
}
```

Obviously, a position specification is only reasonable in partial query patterns which are matched against ordered terms (otherwise there is no position associated with the subterm in the database).

The position specification may be of the following kinds:

- *positive number*. A positive number specifies the position of the matched subterm below its parent term, where 1 is the position of the first subterm
- *negative number*. A negative number specifies the position relative to the last subterm of the parent, where -1 is the position of the last subterm.
- *variable*. A variable matches with a subterm with any position and binds the variable to the position of this subterm as a positive integer number.

Under the "pattern" paradigm, it is straightforward to also permit variables at position specifications. This allows to build very complex queries. The following Xcerpt rule queries the cells in an HTML table and builds the totals for each row and column. Note how the bindings of the variables Row and Column are used to group the results with the help of nested `all` constructs.

Example:

```
CONSTRUCT
    table [
        all tr [
            attributes { row { var Row } },
            all td [ var Value ],
            td [ sum (all var Value) ]
        ],
        tr [
            all td [
                attributes { col { var Col } },
                sum (all var Value)
            ],
            td [ sum (all var Value) ]
        ]
    ]
FROM
    in {
        resource { "sum-table.html", "html" },
        html {{
            desc table {{
                position var Row tr {{
                    position var Col td {{ var Value }}
                }}
            }}
        }}
    }
END
```

3.6 Negation

Sometimes it is useful to know what is *not* in the database. For instance, a query might ask whether there is *no* train between Munich and Paris at a certain time. Also, it might be interesting which students did *not* submit their homework or did *not* fill out the registration form completely. To achieve this, it is possible to negate Xcerpt query patterns.

An important observation is that querying semistructured data requires two different notions of negation which are equivalent in expressiveness but used differently. The first kind of negation is a *subterm negation* and is indicated by the keyword without. The second kind of negation is a *query negation* which behaves like the negation in logic programming and is indicated by the keyword **not**.

Both negation constructs implement "negation as failure" (NaF). Since full NaF is infamous for its various, not always intuitive, and often enough diffiult to use declarative semantics, Xcerpt in its current version imposes a certain syntactical condition called *stratification* [**ABW**], which in a simplified sense disallows recursions over negation. The semantics of NaF is (not are) a wide subject. Discussing it in detail in the context of Xcerpt is out of the scope of this paper.

3.6.1 Subterm Negation: The without Construct

Subterm negation allows to express that a certain subterm should *not* be found at a certain position (the root node is excluded). It is thus a construct only used in query terms. A subterm is negated by prefixing it with the keyword **without**. The following Xcerpt rule uses `without` to find all unresolved citations in an article from the proceedings:

Example:

```
CONSTRUCT
    unresolved_citations {
        all var Citation
    }
FROM
    report {{
        desc var Citation -> cite {
            attributes { ref { var Ref } }
        },
        desc bibliography {{
            without entry {{
                attributes {{ id { var Ref } }}
            }}
        }}
    }}
END
```

Note that the subterm negation is only applicable to subterms (as it specifies that a term *must not have* a certain subterm) and may not be used at the root level. Furthermore, subterm negation is only reasonable in partial term specifications, and order does not have influence on the negated subterms (only on all positive subterms).

Such negation is quite common in semistructured data, as such data is often heterogeneous. For example, a query might ask for "all students that did not submit their homework" (i.e. all student elements that do not contain an element indicating that they submitted their homework). Note that in relational database systems, this negation is very similar to querying for NULL values.

Subterms negated by `without` may contain variables, but such an occurrence can never yield variable bindings, i.e. it is not possible to list all subterms that do not occur. Consequently, all variables that occur in the scope of a `without` have to appear elsewhere outside the scope of a negation construct.

Subterm negation is an *existential* negation: As long as there exists at least one term which does not contain the negated subterm (and matches with the remainder of the pattern), all other terms are irrelevant. There might be terms that contain the negated subterm, those simply do not match.

3.6.2 Query Negation: The not Construct

Query negation specifies that the negated query must not be fulfilled. Such negation is e.g. used to express that there is no train from Munich to Paris at a certain time, or that the student database does not contain a certain student.

Query negation is expressed as part of the query specification. A query is negated by prefixing it with the keyword not. The following query selects all such authors that have an article in the 2004 proceedings but not in the 2003 proceedings (compare with Section 3.2):

Example:

```
and {
    in { resource { "file:proceedings04.xml" },
       desc author {{
          fname { var First }, surname { var Last }
       }}
    },
    not in { resource { "file:proceedings03.xml" },
       desc author {{
           fname { var First }, surname { var Last }
       }}
    }
}
```

Query negation is a *universal* negation. If the query is negated, there must not exist a term with which it matches, i.e. all terms are required to not match with the pattern.

3.7 Regular Expressions

Query languages for Web data need not only to be able to query based on the *database structure*, it is often also desirable to query parts of the plain text contained in the database. This is of particular importance when the document contains mainly text content (like this paper), but is also useful for many database applications.

A powerful construct to support such text querying are *regular expressions*. Xcerpt uses regular expressions as defined in POSIX [**POSIX RE**], but with certain Xcerpt specific extensions. Like in Perl, regular expressions are enclosed in slashes (i.e. "/").

Regular expression matching integrates well with the pattern matching paradigm of Xcerpt, with one exception: so as to retrieve the data that matched a pattern, POSIX defines so-called subexpressions that are enclosed in single parentheses, i.e. (...). The matching substring is later referred to by the position of the subgroup (for instance, Perl uses the variables $1, $2, ..., $9 for this purpose). This way of retrieving matches is straightforward in imperative languages but awkward in a language that is mainly based on pattern matching and rules.

Therefore, Xcerpt extends POSIX regular expressions by *named subexpressions*, where the name is the variable to which the matching substring is bound. Subexpressions of the form (...) are written in Xcerpt as (var X ->...), if the matching substring is to be retrieved in the variable X.

Assume that conference organisers want to publish a list of authors with email addresses, if available, out of the collected proceedings. Due to the massive spam problems, it would be unwise to publish the email addresses as they are. Rather, a common practice is to create a "spamvertised form", where the "@" sign is replaced by something different, e.g. "<at>". The following Xcerpt rule uses a regular expression to separate the user from the domain part of email addresses and creates an author list with such spamvertised email addresses (ordered by last and first name):

Example:

```
CONSTRUCT
    table [
        tr [
            td{ "Name" },
            td{ "Email" }
        ],
        all [
            td{ var First, var Last },
            optional td{ var User, "<at>", var Domain }
        ] order by [Last,First] ]
FROM
    in {
        resource { "file:proceedings04.xml" },
        proceedings04 {{
            paper {{
                front {{
                    author {{
                        fname { var First }, surname { var Last },
                        optional email {
                            /^(var User ->[^@]+)@(var Domain ->[a-zA-Z0-9.-]+)/
                        }
                    }}
                }}
            }}
        }}
    }
END
```

3.8 Sorting

4 Rule Chaining and Recursion

One of the major design principles of Xcerpt is *rule chaining*, an aspect in which Xcerpt departs significantly from other rule-based query languages, like the XML query languages XML-QL **[DFFLS]** and UnQL **[BFS]**, or the relational query language SQL, all of which do not support proper rule chaining (although querying VIEWS in SQL can be seen as a restricted form of rule chaining).

Conceptually, rule chaining means that a rule may interact with another rule by querying the "result" of the other rule. In this way, it is possible to reuse code, to write recursive queries (like a Web crawler recursively querying (for some data patterns) Web sites whose URI appear in previoulsy queried Web sites), to structure complex "query programs" appropriately (so that they are easier to maintain), and even to build external "rule libraries". The following sections highlight important aspects that can be achieved by rule chaining.

4.1 Code Reuse

Many applications have code that is used in several places in a program. Repeating this code at every such occurrence is very inefficient and prone to errors. Therefore, programming languages have long been using concepts like *procedures*, *functions* or *methods*.

Xcerpt's equivalent of a procedure is a rule, which can be "called" by using rule chaining. Like a procedure in an imperative programming language, a rule can serve to encapsulate auxiliary computations that are referred to at several positions in a program into a separate construct.

4.2 Separation of Concerns

When designing applications, a good principle is *separation of concerns*, e.g. separating program logic from presentation. Whereas this separation is impossible with a single XSL stylesheet (as all results are immediately written out and cannot be queried within the same program), and very inconvenient in XQuery (by using XQuery's function mechanism), Xcerpt's rule chaining makes separation of concerns straightforward: one rule is used to specify how to process and transform the data, while another rule focuses on presentation aspects (like a stylesheet).

Conferences usually publish their program as an HTML page. With personal digital assistants becoming more widespread, it is, however, also useful to publish the program in a format more suited for such mobile devices with small screens, e.g. the "wireless markup language" (WML). The following Xcerpt program uses an intermediate rule to create the program from the proceedings and a timetable (i.e. the "program layer"), and two different goals to format the result in HTML and WML (i.e. the "presentation layer"), both of which call the rule that creates the conference program via rule chaining:

Example:

```
GOAL
    out {
        resource { "file:program.html" },
        html [
            head [ title [ "Conference Program" ]
            ],
            body [
                h1 [ "Conference Program" ],
                table [
                    tr [
                        td [ "Time" ],
                        td [ "Title" ],
                        td [ "Authors" ]
                    ],
                    all tr [
                        td [ var Time ],
                        td [ var Title ],
                        td [ all var Author ]
                    ] order by [Time]
                ]
            ]
        ]
    }
```

```
FROM
    program [[
        talk [[
            title [ var Title ],
            author [ var Author ],
            time [ var Time ]
        ]]
    ]]
END

GOAL
    out {
        resource { "file:prices.wml"},
        wml [
            all card [
                "Time: " , var Time,
                "Title: " , var Title ,
                "Authors: " , all var Authors
            ]
        ]
    }
FROM
    program [[
        talk [[
            title [ var Title ],
            author [ var Author ],
            time [ var Time ]
         ]]
     ]]
END

CONSTRUCT
    program [
        all talk [
            title [ var Title ],
            all author [ var First, var Last ],
            time [ var Time ] ] ]
FROM
    and {
        in {
            resource { "file:proceedings.xml" },
            proceedings [[
                paper [[
                    front [[
                        title { var Title },
                        author {{
                            fname { var First },
                            surname { var Last }
                        }}
                    ]]
                ]]
            ]]
        },
        in {
            resource { "file:timetable.xml" },
            timetable [[
                entry [[
                    time [ var Time ],
                    title [ var Title ]
                ]]
            ]]
        }
    }
END
```

Note also that both query terms in the and-connected query part share the variable Title to join the information from the proceedings database with the information of the timetable.

4.3 Mediation

Complex applications often need to query several databases in order to retrieve information, all of which might have different structure. For example, the booking system of a travel agency might need to query the flight schedules of different airlines. Rule chaining can be used as a mediator by using Xcerpt rules that transform data from different resources into a common structure, which can then be queried by other rules in the program.

The following rules create a unified author list for authors that published an article either in the 2003 or in the 2004 proceedings. The second and the third rule create from the proceedings04.xml and proceedings03.xml files author terms containing first and last name of an author (note that they do not contain the collection construct `all` and thus yield several alternative data terms as answers, one for each author). The first rule (goal) collects all such author terms within an authors term.

Example:

```
GOAL
    out {
        resource { "file:authors.xml" },
        authors [
            all var Author
        ]
    }
FROM
    var Author -> author {{ }}
END

CONSTRUCT
    author { var First, var Last }
FROM
    in {
        resource { "file:proceedings04.xml" },
        proceedings04 [[
            paper [[
                front [[
                    author {{
                        fname { var First },
                        surname { var Last }
                     }}
                ]]
            ]]
        ]]
    }
END

CONSTRUCT
    author { var First, var Last }
FROM
    in {
        resource { "file:proceedings03.xml" },
        proceedings03 [[
            paper [[
                front [[
                    author {{
                        fname { var First },
                        surname { var Last }
```

```
                        }}
                    ]]
                ]]
            ]]
        }
    END
```

4.4 Recursion

Besides querying other rules, Xcerpt rules can also call themselves recursively. In contrast to the inherent structural recursion in XSLT, which is limited to the tree structure of the input tree, Xcerpt allows all kinds of recursions, not only structural recursion over XML documents.

Applications of recursion in Web query languages are manifold:

- structural recursion over the input tree (like in XSLT) is necessary to perform transformations that preserve the overall document structure and change only certain things in arbitrary documents (e.g. replacing all elements in an HTML document by elements).
- recursion over the conceptual structure of the input data (e.g. over a sequence of elements) is used to iteratively compute data (e.g. create a hierarchical representation from flat structures with references).
- recursion over references to external resources (hyperlinks) is desirable in applications like a Web crawler that recursively visit Web pages.

The following rule implements a Web crawler that recursively visits Web sites, retrieves all links (i.e. a elements) from it and in turn queries each of these by recursively calling itself and with each call passes a new binding for the variable Resource.

Note that this Web Crawler provides only minimal functionality. For instance, it does not terminate in case of cyclic hyperlinks.

Example:

```
CONSTRUCT
crawler [
    linklist {
        all link { var Link1 },
        all link { var Link2 }
    },
    var Resource ]
FROM
    and {
        in {
            resource { var Resource, "html" },
            desc a {{
                attributes {{ href { var Link1 } }}
            }}
        },
        crawler [
            linklist {{ var Link2 }},
            var Link1
        ]
    }
END
```

4.5 Backward and Forward Chaining

Rule languages in general allow to evaluate rules in two directions, so-called "forward chaining" and "backward chaining".

- Forward chaining is *data driven*. Rules are evaluated iteratively against the current set of data terms until saturation is achieved. Forward chaining is useful for instance for materialising views and for view maintenance, and is widely used in deductive databases.
- Backward chaining is *goal driven*. Beginning with the query part of the goal, program rules are selected if they are relevant for "proving" a query term. The query term in question is then replaced by the query part of the selected rule. Backward chaining is useful when the expected result is small in comparison with the number of possible results of the program. It is mainly used in logic programming languages like Prolog.

In a Web environment, both forward and backward chaining are desirable. A forward chaining approach is e.g. useful when creating a static Web site (consisting of one or more Web pages) from an input document and an Xcerpt program used as a "stylesheet", i.e. "materialising the HTML view" on the input data. On the other hand, backward chaining is necessary when querying large collections of documents, in particular the Web itself, where it is impossible to begin with the complete database, which might be the whole Web.

The current Xcerpt prototype implements backward chaining only. Different aspects of backward chaining have been investigated in [BSS04]. A forward chaining algorithm is specified, but currently not implemented.

5 Simulation Unification

5.1 Motivation

Xcerpt query terms are matched against data terms using an algorithm called *simulation unification*. In general, a unification algorithm takes two terms (containing variables) and determines a substitution for the variables (the so-called *unifier*) such that the two terms are "equal" with respect to some kind of equality relation (if possible).

The most common kind of unification (and therefore often called "standard" unification) is *Robinson's unification*, which is among others used in the logic programming language Prolog.

Example: Consider the following two Xcerpt terms:

```
        f[var X,b,c]     f[a,b,var Y]

 Applying Robinson's unification to these two terms yields the following unifier:

        σ = { var X = a, var Y = c }
```

For Xcerpt's matching, Robinson's unification is not applicable: it does not respect different kinds of term specifications (ordered - unordered and total - partial), has no support for optional or negated subterms, and does not allow variable restrictions, because it merely ensures *syntactic* equality. Therefore, simulation unification is based on a different relation between terms, a so-called *rooted simulation*.

5.2 Rooted Simulation Between Ground Query Terms

Graph Simulation

Informally, a *rooted simulation* is a relation between two graphs such that the structure of the one graph is found in the structure of the other graph (similar to a *graph homomorphism*). The distinguished root nodes of the two graphs have to be part of this relation. On graphs, rooted simulation is defined as follows:

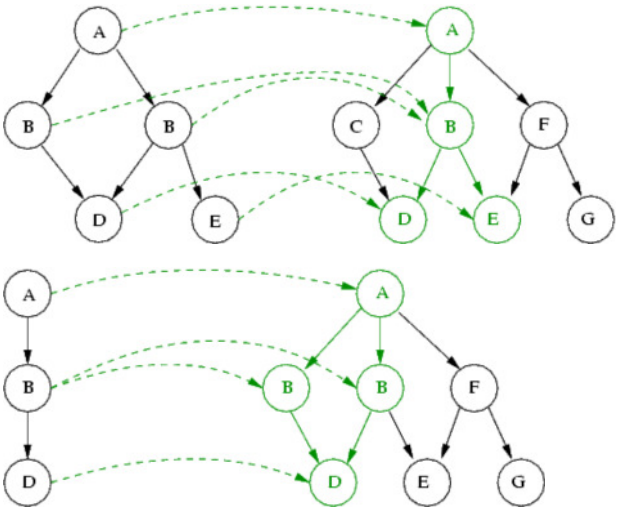
Definition:

Given two graphs $G_1 = (V_1,E_1,r_1)$ and $G_2 = (V_2,E_2,r_2)$ with sets of vertices V_i, sets of edges E_i, and distinguished vertices r_i (called the *root nodes*). A relation $\preccurlyeq$ is called a *rooted graph simulation*, if the following properties hold:

- $(r_1,r_2) \in \preccurlyeq$

- if $(v_1,v_2) \in \preccurlyeq$ then the labels of v_1 and v_2 match
- if $(v_1,v_2) \in \preccurlyeq$ and $(v_1,v_1') \in E_1$, then there exists $v_2' \in V_2$ such that $(v_1',v_2') \in \preccurlyeq$ and $(v_2,v_2') \in E_2$

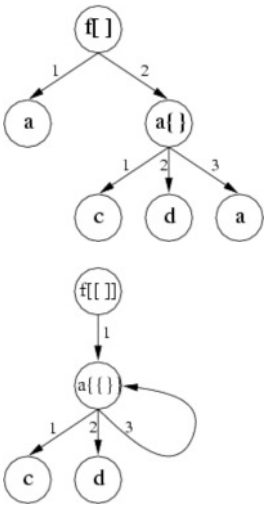
Example: Two simulations of the left term into the right term.



Ground Query Terms and Ground Query Term Graphs

A *ground query term* is a term that does not contain variables. A ground query term induces a graph in a straightforward manner:

Example: Graphs induced by f[a,a[c,d,a]] and f[[&1 @ a{{c,d,^&1}}]]



Formally, this graph is defined as follows:

Definition:

Given a ground query term t. The *graph induced by t* is a tuple $G_e = (V,E,r)$, with:

1. a set of *vertices* V defined as the set of all (immediate and indirect) subterms of t (including t itself)
2. a set of *edges* $E \subseteq V \times V \times \mathbf{N}$ characterised as follows:
   - for all terms $t_1,t_2,t_3 \in V$: if t_2 is the i^{th} subterm of t_1 and of the form ^oid (a reference), and t_3 is of the form oid @ t', with oid an identifier and t' a term, then $(t_1,t_3,i) \in E$
   - for all terms $t_1,t_2 \in V$: if t_2 is the i^{th} subterm of t_1 and *not* of the form ^oid, then $(t_1,t_2,i) \in E$
3. a distinguished vertex $r \in V$ called the *root node* with r = t

The *label* of a vertex is either the label, the string value, or the regular expression of the subterm it represents.

Ground Query Term Simulation

The graph simulation defined above can be extended to the graphs induced by ground query terms. In case a ground query term simulates in a data term, it *matches* with the data term, i.e. the query succeeds. However, this extension needs to respect the different kinds of term specifications and the special Xcerpt constructs. The following definition of a ground query term simulation covers ordered and unordered term specifications and the position construct, but the constructs without and optional are excluded, because the formalisation is rather complex. The following auxiliary notions are used in to define ground query term simulation.

Definition:

Given two sets of ground query terms $M = \{s_1,...,s_m\}$ and $N = \{t_1,...,t_n\}$. A partial or total mapping $\varpi: M \rightarrow N$ is called

- *index injective*, if for all $s_i,s_j \in M$ with $i \neq j$ holds that $\varpi(s_i) \neq \varpi(s_j)$.
- *index monotonic*, if for all $s_i,s_j \in M$ with $i < j$ holds that $\varpi(s_i) < \varpi(s_j)$.
- *index bijective*, if it is index injective and for all $t_k \in N$ exists a $s_i \in M$ such that $\varpi(s_i) = t_k$.
- *position respecting*, if for all $s_i \in M$ of the form `position j si'` holds that $\varpi(s_i) = t_k$ with k = j
- *position preserving*, if for all $s_i \in M$ of the form `position j si'` holds that $\varpi(s_i) = t_k$ such that t_k is of the form `position l tk'` and j = l.

Index monotonic mappings preserve the order of subterms in the two sets and are used for matching terms with ordered term specifications. *Index bijective* mappings ensure that all of the elements of N are mapped to and are used for total term specifications.

A *position respecting* mapping maps a subterm with position specification to a subterm with the specified position and is required if the second term (i.e. the set N) uses total and ordered term specification. A *position preserving* mapping maps a subterm with position specification to a subterm with the same position specification; it is applicable in case the second term (the set N) is

incomplete with respect to order or breadth, as the exact position cannot be determined otherwise in these cases. In particular, this ensures that a term always matches itself.

Definition:

Let t_1 and t_2 be ground (query) terms, and let G_1 and G_2 be the graphs induced by t_1 and t_2. A relation $\preccurlyeq \subseteq V_1 \times V_2$ on the sets V_1 and V_2 of immediate and indirect subterms of t_1 and t_2 is called a *ground query term simulation*, if and only if:

1. $t_1 \preccurlyeq t_2$ (i.e. the roots are in $\preccurlyeq$)
2. if $v_1 \preccurlyeq v_2$ and neither v_1 nor v_2 contain subterms of the forms *without t* or *optional t*, then the labels l and l' of v_1 and v_2 match and there exists a *total, index injective mapping* ϖ:SubT(v_1)→SubT(v_2) such that for all $s \in$ SubT(v_1) holds that $s \preccurlyeq \varpi(s)$. Depending on the kinds of subterm specifications of v_1 and v_2, ϖ in addition satisfies the following requirements:

v_1	v_2	**it holds that**
l[s_1,...,s_m]	l'[t_1,...,t_n]	ϖ is *index bijective* and *index monotonic*
l{s_1,...,s_m}	l'{t_1,...,t_n}	ϖ is *index bijective* and *position preserving*
l[[s_1,... s_m]]	l'[t_1,...,t_n]	ϖ is *index monotonic* and *position respecting*
	l'[[t_1,...,t_n]]	ϖ is *index monotonic* and *position preserving*
l{ {s_1,... s_m} }	l'{t_1,...,t_n}	ϖ is *position preserving*
	l'[t_1,...,t_n]	ϖ is *position respecting*
	l'{ {t_1,...,t_n} }	ϖ is *position preserving*
	l'[[t_1,...,t_n]]	ϖ is *position preserving*

The following examples illustrate some aspects of this definition. Note that indices have been added to subterms to distinguish the respective positions. These are not considered to be part of the subterm labels.

Example:

let t_q=f[[b_1,c_2]] and t_d=f[a_1,b_2,c_3,d_4]. $t_q \preccurlyeq t_d$, because

1. the labels match (in both cases *f*)
2. there exists a total mapping ϖ from M={b_1,c_2} to N={a_1,b_2,c_3,d_4} which is (1) *index injective* (no two suberms of M are mapped to the same subterm of N), (2) *index monotonic* (the subterms of M are mapped to subterms of N in the correct order), and (3) vacously *position respecting* (as no position constructs occur)

Example:

let t_q=f[[c_1,b_2]] and t_d=f[a_1,b_2,c_3,d_4]. $t_q \preccurlyeq t_d$ does not hold, because the only total, index injective mapping ϖ from M={c_1,b_2} to N={a_1,b_2,c_3,d_4} is ϖ= { $c_1 \rightarrow c_3$, $b_2 \rightarrow b_2$ } and thus not index monotonic.

Example:

let t_q=f'{ {c_1,b_2} } and t_d=f[a_1,b_2,c_3,d_4]. $t_q \preccurlyeq t_d$, because

1. the labels match (in both cases *f*)
2. there exists a total mapping ϖ from M={c_1,b_2} to N={a_1,b_2,c_3,d_4} which is (1) *index injective* (no two suberms of M are mapped to the same subterm of N), and (2) vacously *position respecting* (as no position constructs occur)

Example:

let t_q=f'{ {c_1,position 2 b_2} } and t_d=f[a_1,b_2,c_3,d_4]. $t_q \preccurlyeq t_d$, because

1. the labels match (in both cases *f*)
2. there exists a total mapping ϖ from M={c_1,b_2} to N={a_1,b_2,c_3,d_4} which is (1) *index injective* (no two suberms of M are mapped to the same subterm of N), and (2) *position respecting* (as the subterm b of t_d has position 2)

Example:

let t_q=f'{ {c_1,position 2 b_2} } and t_d=f'{a_1,b_2,c_3,d_4}. $t_q \preccurlyeq t_d$ does not hold, because the only mapping ϖ is not position preserving (as t_d does not contain subterms of the form `position n t`)

Example:

let t_q=f'{ {c_1,position 2 b_2} } and t_d=f'{a_1,position 2 b_2,c_3,d_4}. $t_q \preccurlyeq t_d$, because

1. the labels match (in both cases *f*)
2. there exists a total mapping ϖ from M={c_1,b_2} to N={a_1,b_2,c_3,d_4} which is (1) *index injective* (no two suberms of M are mapped to the same subterm of N), and (2) *position preserving* (as the subterm b of t_d also specifies position 2)

5.3 Substitutions and Sets of Substitutions

Substitutions

A substitution is a set of variable bindings, usually denoted by the greek letters σ and τ. A substitution can be applied to a (non-ground) term, in which case all occurrences of variables are replaced by their bindings in the substitution. Given a term t and a substitution σ the application of σ to t is denoted σ(t).

Example:

Let σ = { X → f{a}, Y → f{b} } be a substitution, and let t=g{var X, var Y, var Z} be a query term. The application of σ to t is σ(t) = g{f{a},f{b}, var Z}.

A substitution is called a *grounding substitution*, if the query term resulting from applying a substitution is ground.

Example:

Let σ = { X → f{a}, Y → f{b}, Z → h{a} } be a substitution, and let t=g{var X, var Y, var Z} be a query term. Since the application of σ to t is σ(t) = g{f{a},f{b}, h{a}}, σ is a grounding substitution for t.

Application of substitutions to query terms needs to respect the variable restrictions that occur in query terms. For instance, σ = { X → f{a} } is not a valid substitution for the query term f{ {var X -> g{{}} } }, as the binding for X is not matched by the pattern restriction g{{}}.

Sets of Substitutions

In the formalisation below, it is necessary to consider not only substitutions, but also *sets of substitutions* (denoted by the upper greek letters Σ and Φ). When applied to a query term, a set of substitutions yields a set of ground query terms, one for each substitution:

Example:

Let Σ = { { X → a }, { X → b } } be a set of substitutions and let t=f{var X} be a query term. The application of Σ to t is Σ(t) = { f{a}, f{b} }.

The issue is more complicated with construct terms containing the grouping constructs `all` or `some`. In this case, the grouping construct groups several substitutions in one term:

Example:

Let Σ = { { X → a }, { X → b } } be a set of substitutions and let t=f{all var X} be a construct term. The application of Σ to t is Σ(t) = { f{a, b} }.

5.4 Answers and (Grounding) Simulation Unifiers

Answer Substitutions

The problem of matching a query term with a data term can be expressed in terms of the ground query term simulation above. As a query term usually contains variables, it is necessary to provide a fitting grounding substitution such that the ground instance of the query term simulates into the data term. Such grounding substitutions are called *answer substitutions*:

Definition:

Given a query term t_q that does not contain the construct `except`, and a data term t_d. A grounding substitution σ for t_q is called an *answer substitution* for t_q and t_d, if and only if

1. the ground instance $\sigma(t_q)$ simulates in t_d (i.e. $\sigma(t_q) \preccurlyeq t_d$)
2. for all subterms of t_q that have the form `var X -> t` such that t is not of the form `desc t'`, it holds that σ(t) simulates in σ(X) (the binding of `X` in σ), i.e. $\sigma(t) \preccurlyeq \sigma(X)$
3. for all subterms of t_q that have the form `var X -> desc t`, it holds that σ(X) (the binding of `X` in σ) contains an immediate or indirect subterm t' such that σ(t) simulates in σ(t'), i.e. $\sigma(t) \preccurlyeq \sigma(t')$

The first requirement ensures that the substitution yields a ground instance that simulates into the data term. Requirements two and three require that the bindings for variables adhere to the pattern restrictions of the respective variables.

Example:

Given a query term t_q=f{{ var X -> g{{var Y}}, var Y}}, and a data term t_d=f{g{a},g{b},h{d},a,c}.

- the substitution σ={ X → g{a}, Y → b} is an answer substitution for t_q and t_d
- the substitution σ' = { X → h{d}, Y → d } is no answer substitution, because the binding X → h{d} does not match the restriction g{{var Y}} (the labels h and g are not equal)
- the substitution σ'' = { X → g{b}, Y → a } is no answer substitution, because the binding X → g{b} does not match the restriction g{{var Y}} (Y is mapped to a and does not match with b)

Answers

Due to incomplete term specifications (partial, unordered, descendant), there is in general more than one answer substitution for a query term t_q and a data term t_d

Example:

Let t_q=f{{var X}} and let t_d=f{a,b,c}. The substitutions σ_1={X → a}, σ_2={X → b}, and σ_3={X → c} all are answer substitutions for t_q and t_d.

An *answer* for a query term t_q and a data term t_d is therefore a *set* of answer substitutions, rather than a single substitution. Because it is often desirable to get *all* substitutions (e.g. in case of the all and some constructs in query terms), this set has to be *maximal*:

Definition:

Given a query term t_q and a data term t_d. A set of substitutions Σ is called *the set of all answer substitutions*, or short: *answer*, for t_q and t_d, if and only if

- all substitutions $\sigma \in \Sigma$ are answer substitutions for t_q and t_d
- there exists no set Φ such that Σ is a proper subset of Φ (i.e. $\Sigma \subset \Phi$ and $\Sigma \neq \Phi$) and all $\varphi \in \Phi$ are answer substitutions for t_q and t_d (maximality)

Since the set of answer substitutions represents different bindings that all yield ground instances simulating into the data term, they are also called *alternative substitutions*.

Example:

Let t_q=f{{var X}} and let t_d=f{a,b,c}. The set of substitutions Σ={ {X → a}, {X → b}, {X → c} } is the answer for t_q and t_d.

(Grounding) Simulation Unifiers

For a backward chaining evaluation, the notion of answers needs to be extended to a query term t_q and a *construct term* t_c. In this case, both terms may contain variables, and thus both terms need to be made ground to ensure a simulation. The definition of answers is extended as follows:

Definition:

Let t_1 and t_2 be two query or construct terms. Furthermore, let Σ be a set of grounding substitutions for both t_1 and t_2. Σ is called a *simulation unifier* if and only if

$\forall\ t_1' \in \Sigma(t_1)\ \exists\ t_2' \in \Sigma(t_2)$ such that $t_1' \preccurlyeq t_2'$

Example:

Let t_1 = f{{var X, b}} and let t_2 = f{a, var Y, c}. The simulation unifier for t_1 and t_2 is

Σ = { {X → a, Y → b}, {X → c, Y → b} }

$\Sigma(t_1)$ = { f{{a,b}}, f{{c,b}} }
$\Sigma(t_2)$ = { f{a,b,c} }

Example:

Let t_1 = f{{var X}} and let t_2 = f{all var Y}. A simulation unifier for t_1 and t_2 is

Σ = { {X → a, Y → a}, {X → b, Y → a}, {X → a, Y → b}, {X → b, Y → b} }

$\Sigma(t_1)$ = { f{{a}}, f{{b}} }
$\Sigma(t_2)$ = { f{a,b} }

Note that in this case, Σ is not the only conceivable simulation unifier of t_1 and t_2.

Simulation Unification

The formalisation above provides a *declarative* semantics for the evaluation of a query term against a data term. It characterises simulation unifiers, but does not specify how they can be obtained. The concrete *algorithm* for this is called *simulation unification*. Simulation unification is not further described here, but follows the declarative semantics rather closely.

6 Further Aspects of Xcerpt

6.1 Type Specification and Type Checking

6.2 Temporal and Calendric Type Specification and Checking

6.3 Semantic Web Querying and Reasoning Using Xcerpt

6.4 XChange: Reactivity on the Web and Semantic Web

7 References

[ABS]	S. Abiteboul, P. Buneman, D. Suciu. Data on the Web -- From Relations to Semistructured Data and XML, ISBN 1-55860-622-X, Morgan Kaufmann, 2000
[ABW]	Krzysztof Apt, Howard Blair and Adrian Walker Towards a Theory of Deductive Knowledge. In: Jack Minker (editor). *Foundations of Deductive Databases and Logic Programming*, Morgan-Kaufmann, 1988.
[SemWeb]	Tim Berners-Lee, James Hendler, and Ora Lassila The Semantic Web -- A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. Scientific american, May 17, 2001 **http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21**
[visXcerpt1]	Sacha Berger, François Bry, and Sebastian Schaffert A Visual Language for Web Querying and Reasoning. Proceedings of the Workshop on Principles and Practice of Semantic Web Reasoning (ICLP03), Mumbai, India (9th - 13th December 2003), LNCS 2901, 2003, ISBN 3-540-20582-9 **http://www.pms.ifi.lmu.de/publikationen#PMS-FB-2003-6**
[visXcerpt2]	Sacha Berger, François Bry, Sebastian Schaffert, and Christoph Wieser Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. Proceedings of 29th Intl. Conference on Very Large Databases (VLDB03), 2003 **http://www.pms.ifi.lmu.de/publikationen#PMS-FB-2003-2**
[visXcerpt3]	Sacha Berger Conception of a Graphical Interface for Querying XML. Diploma thesis, Institute for informatics, University of Munich, Germany, 2003 **http://www.pms.ifi.lmu.de/publikationen#DA_Sacha.Berger**
[BFS]	Peter Buneman, Mary Fernandez, and Dan Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76--110, 2000.
[DFFLS]	Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu XML-QL: A Query Language for XML. Technical report, W3C, 1998.
[TM2]	Lars Marius Garshol Metadata? Thesauri? Taxonomies? Topic Maps! ? Making sense of it all, March 2004 **http://www.ontopia.net/topicmaps/materials/tm-vs-thesauri.html**
[UseCases]	Sebastian Kraus Use Cases für Xcerpt: Eine positionelle Anfrage- und Transformationssprache für das Web. Diploma Thesis, Institute for Informatics, University of Munich, Germany (in German), 2004 **http://www.pms.ifi.lmu.de/publikationen/#DA_Sebastian.Kraus**
[OWL]	Deborah L. McGuinness and Frank van Harmelen, eds. OWL Web Ontology Language Overview W3C Recommendation, 10 February 2004 **http://www.w3.org/TR/owl-features/**
[TM1]	Steve Pepper The TAO of Topic Maps -- Finding the Way in the Age of Infoglut **http://www.ontopia.net/topicmaps/materials/tao.html**
[POSIX RE]	IEEE / The Open Group. POSIX Regular Expressions. *The Open Group Base Specifications*, Issue 6, 2001.
[RDF]	Resource Description Framework (RDF) **http://www.w3.org/RDF/**
[Xcerpt1]	**http://xcerpt.org**
[Xcerpt2]	François Bry, Sebastian Schaffert, and Paula-Lavinia Patranjan. Xcerpt and XChange: Deductive Languages for Data Retrieval and Evolution on the Web. Proceedings of Workshop on Semantic Web Services and Dynamic Networks (SWSDN2004), 2004 **http://www.pms.ifi.lmu.de/publikationen#PMS-FB-2004-10**

The Semantic Web and its Basic Data Format: RDF

Benedikt Linse

Institute for Informatics, University of Munich, Germany

June 24th 2008

Tim Berners-Lee's Semantic Web Vision (1)

The Semantic Web will bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users.

Tim Berners-Lee's Semantic Web Vision (2)

The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation.

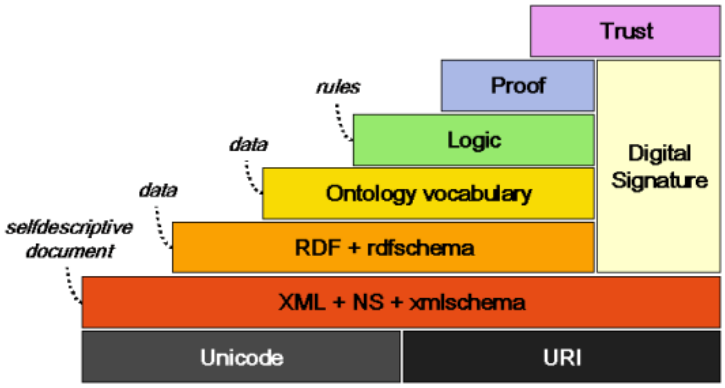
Tim Berners-Lee's Semantic Web Vision (3)

"The Semantic Web is a vision: the idea of having data on the Web defined and linked in a way that it can be used by machines not just for display purposes, but for automation, integration and reuse of data across various applications" – Tim Berners-Lee

Outline

- the Semantic Web Vision
- Semantic Web principles / RDF
  - RDF abstract data model
  - RDF serializations
- RDF semantics / model theory

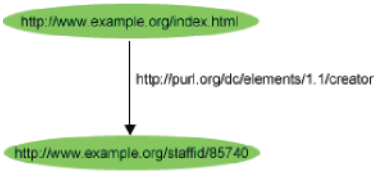
RDF in the Semantic Web Tower



On the importance of URIs

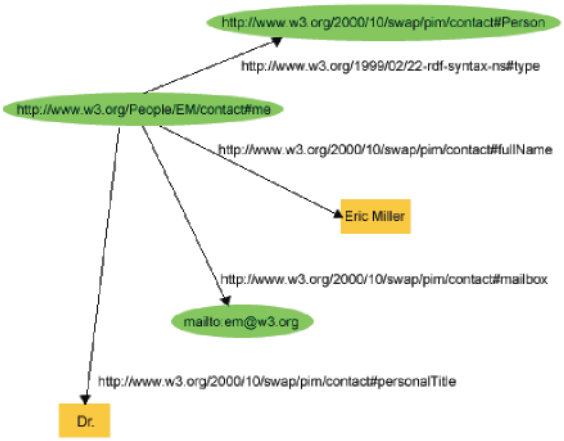
That idea of URL was the basic clue to the universality of the Web. That was the only thing I insisted upon.

RDF Abstract Data Model: Triples and URIs



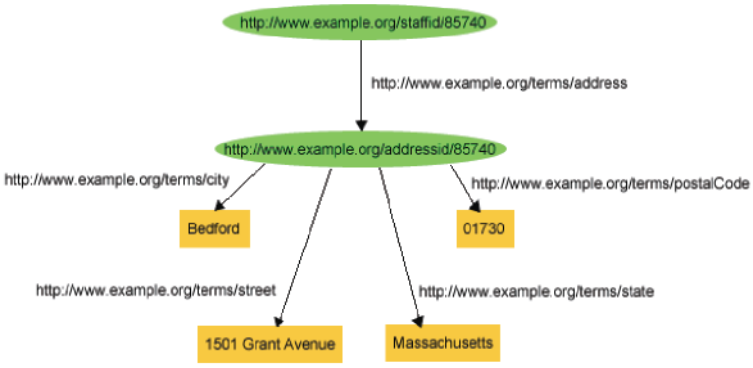
- triples made up of subject predicate object
- URIs for uniquely naming resources or concepts

RDF Abstract Data Model: Graphs and Literals



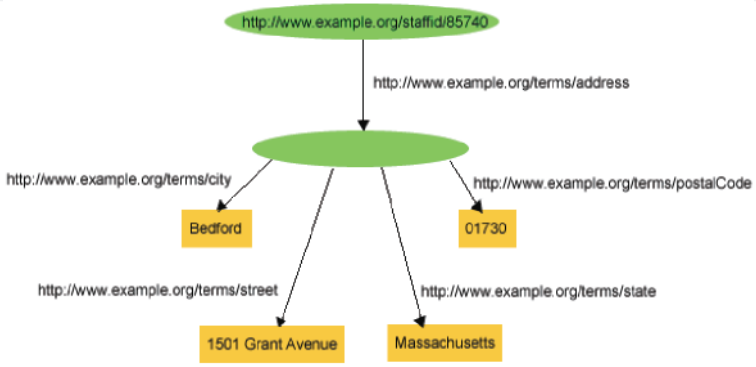
- RDF graphs are sets of RDF triples
- Resources need not be accessible on the Web
- Literals are not interpreted – stand for themselves
- Literals may only occur in object position

RDF Abstract Data Model: Structured Data and Blank Nodes



- Structured information must be encoded in triples
- One can associate a URI for a structured object
- Blank Nodes more appropriate for this purpose

RDF Abstract Data Model: Structured Data and Blank Nodes




```
exstaff:85740    exterms:address      _:johnaddress .
_:johnaddress    exterms:street       ''1501 Grant Avenue'' .
_:johnaddress    exterms:city         ''Bedford'' .
_:johnaddress    exterms:state        ''Massachusetts'' .
_:johnaddress    exterms:postalCode   ''01730'' .
```


RDF Abstract Data Model: Notes on Blank Nodes

- Blank nodes are used to encode information about concepts one does not want to associate an URI with
- In serializations, blank node identifiers start with _:
- In serializations, the scope of blank nodes is only the document.
- Scopes of URIs are world-wide!
- Blank nodes may be seen as existential variables.
- Blank nodes may introduce *redundancy* into RDF graphs.

RDF/XML Serializations

- RDF graphs must be serialized to be exchanged!
- RDF/XML: based on the recommendations *Namespaces in XML*, *XML Information Set* and *XML Base*
- N-Triples: canonical data format not based on standards

RDF Serializations: RDF/XML: Striped Syntax

```
<?xml version=''1.0''?>
<rdf:RDF xmlns:rdf=''http ... yntax-ns''
         xmlns:ex=''http://example.org/''>

  <rdf:Description>
    <ex:editor>
      <rdf:Description>
        <ex:homePage>
          <rdf:Description>
          </rdf:Description>
        </ex:homePage>
      </rdf:Description>
    </ex:editor>
  </rdf:Description>

</rdf:RDF>
```

RDF/XML: URIs in Subject and Object Position

```
<rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
  <ex:editor>
    <rdf:Description>
      <ex:homePage>
        <rdf:Description rdf:about="http://purl.org/net/dajobe/">
        </rdf:Description>
      </ex:homePage>
    </rdf:Description>
  </ex:editor>
</rdf:Description>
```


Grouping by Subject and Literals in RDF/XML

```
<rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar">
  <ex:editor>
    <rdf:Description>
      <ex:homePage rdf:resource="http://purl.org/net/dajobe/"/>
      <ex:fullName>Dave Beckett</ex:fullName>
    </rdf:Description>
  </ex:editor>
  <dc:title>RDF/XML Syntax Specification (Revised)</dc:title>
</rdf:Description>
```

RDF Literals as XML Attribute Values in RDF/XML

```
<rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
                 dc:title="RDF/XML Syntax Specification (Revised)">
  <ex:editor>
    <rdf:Description ex:fullName="Dave Beckett">
      <ex:homePage rdf:resource="http://purl.org/net/dajobe/"/>
    </rdf:Description>
  </ex:editor>
</rdf:Description>
```


XML Literals Serialized within RDF/XML

- RDF literals may be plain or typed
- typed literals are mapped from a lexical space to a value space
- ‘‘1999-08-16’’^^{xsd:date} and ‘‘18.08.1999’’^^{xsd:date} should stand for the same value
- literals may also be typed as XML fragments

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ex="http://example.org/stuff/1.0/">
  <rdf:Description rdf:about="http://example.org/item01">
    <ex:prop rdf:parseType="Literal"
             xmlns:a="http://example.org/a#"><a:Box required="true">
          <a:widget size="10" />
          <a:grommit id="23" /></a:Box>
    </ex:prop>
  </rdf:Description>
</rdf:RDF>
```

Other Typed Literals in RDF/XML

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ex="http://example.org/stuff/1.0/">
  <rdf:Description rdf:about="http://example.org/item01">
    <ex:size rdf:datatype="http://www.w3.org/2001/XMLSchema#int">123</ex:size>
  </rdf:Description>
</rdf:RDF>
```


RDF Graphs with Multiple Roots

- Encoding of blank nodes as `rdf:description` elements without `rdf:about`-attribute
- Reference to blank nodes with `rdf:nodeID`

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:ex="http://example.org/stuff/1.0/">
  <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
                   dc:title="RDF/XML Syntax Specification (Revised)">
    <ex:editor rdf:nodeID="abc"/>
  </rdf:Description>

  <rdf:Description rdf:nodeID="abc"
                   ex:fullName="Dave Beckett">
    <ex:homePage rdf:resource="http://purl.org/net/dajobe/"/>
  </rdf:Description>
</rdf:RDF>
```

Omission of Blank Nodes by rdf:parseType=‘‘Resource’’

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:ex="http://example.org/stuff/1.0/">
  <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
                   dc:title="RDF/XML Syntax Specification (Revised)">
    <ex:editor rdf:parseType="Resource">
      <ex:fullName>Dave Beckett</ex:fullName>
      <ex:homePage rdf:resource="http://purl.org/net/dajobe/"/>
    </ex:editor>
  </rdf:Description>
</rdf:RDF>
```


RDF/XML Property Attributes

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:ex="http://example.org/stuff/1.0/">
  <rdf:Description rdf:about="http://www.w3.org/TR/rdf-syntax-grammar"
                   dc:title="RDF/XML Syntax Specification (Revised)">
    <ex:editor ex:fullName="Dave Beckett" />
    <!-- Note the ex:homePage property has been ignored for this example -->
  </rdf:Description>
</rdf:RDF>
```

Syntactic Sugar for `rdf:type` Statements

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:ex="http://example.org/stuff/1.0/">
  <ex:Document rdf:about="http://example.org/thing">
    <dc:title>A marvelous thing</dc:title>
  </ex:Document>
</rdf:RDF>
```


RDF/XML Criticism

- Verbosity of the XML syntax
- Not all URIs expressible as element names because of restrictions of qualified names
- Many different possibilities to represent the same RDF graph

RDF Model Theory

- What *is* a model theory?

The Semantic Web Vision
Semantic Web Principles / RDF
RDF Model Theory
First Order Languages, Interpretations
Simple RDF Interpretations
Simple RDF Entailment: Lemmas and Decidability

RDF Model Theory

- What *is* a model theory?

REAL Model Theory

- study the interpretation of a language by means of *set-theoretic structures*
- describe the *minimal conditions* that a world must satisfy to *associate an appropriate meaning* to an expression in the language
- FO Model theory deals with FO-Formulas and structures satisfying them.
- decide whether derivation rules are valid
- *provide a dependable basis for reasoning about RDF data*

Syntax: First Order Languages

Signature of a FO language

- for each natural number $n \geq 0$ a possibly empty set Rel_L^n of n-ary relation symbols.
- for each natural number $n \geq 0$ a possibly empty set Fun_L^n of n-ary function symbols. 0-ary function symbols are called *constants*.

Note:

- Variables are not considered part of the signature, but as logical symbols.
- The sets of relation and function symbols are assumed to be *disjunct*.

Interpretations of First Order Languages

Interpretation

Let L be a FO language. An L-interpretation M is a triple (D, Map, A) where

- D is a non-empty set, the *universe* of M
- Map is a mapping defined on the signature of L such that
  - for every n-ary function symbol f of L, $Map(f) : D^n \to D$ is an n-ary function.
  - for every n-ary predicate symbol p of L, $Map(p) \subseteq D^n$ is an n-ary relation.
- A is a D-variable-assignment.

Simple RDF Interpretations

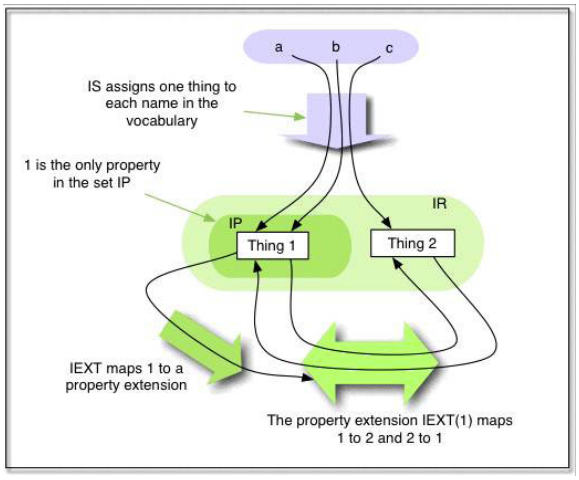
A Simple Interpretation of an RDF Vocabulary $V = U \cup B \cup L$ is defined by:

- $IR \neq \emptyset$: *domain* or *universe* of I.
- IP: the *set of properties* of I.
- $IEXT : IP \to \mathcal{P}(IR \times IR)$.
- $IS : U \to IR \cup IP$.
- IL : a mapping from typed Literals in V into IR.
- $LV \subseteq IR$: the *set of literal values*

Note:

- IR and IP are *neither* required to be disjunct *nor* is it demanded that $IP \subseteq IR$.
- a property may be applied to itself (Axiom of foundation!)
- Other kinds of interpretations: *rdf-interpretations*, *rdfs-interpretations*, *datatyped interpretations*

An Exemplary Simple RDF Interpretation



Model Relationship among FO Interpretations and FO Formulas

Let L be a FO language and $M = (D, map, A)$ an L-Interpretation. The *model relationship* $M \models F$ is defined as follows:

- $M \models \top$ is true
- $M \models \bot$ is false
- $M \models p$ iff $() \in p^M$
- $M \models p(t_1, \ldots, t_n)$ iff $(t_1^M, \ldots, t_n^M) \in p^M$ for $n \geq 1$
- $M \models \neg G$ iff $M \models G$ is false
- $M \models (G_1 \wedge G_2)$ iff $M \models G_1$ and $M \models G_2$)
- $M \models (G_1 \vee G_2)$ iff $M \models G_1$ or $M \models G_2$)
- $M \models \forall x G$ iff for all $d \in D$ holds $M[d/x] \models G$
- $M \models \exists x G$ iff there is a $d \in D$ such that $M[d/x] \models G$

Denotations of RDF Graphs

- Triples can be seen as atomic logic formulae: The triple `(example:anna rdf:type foaf:person)` could also be written as `rdf:type(example:anna, foaf:person)`.
- RDF graphs can be considered as conjunctions of atomic logical formulae, hence as logical formulae themselves.
- *Denotations* extend the interpretation mapping I from names (URIs and Literals) to graphs and blank nodes.

Denotations of *Ground* RDF Graphs and their Constituents

- if E is a plain literal "aaa" in V then $I(E) =$ `aaa`
- if E is a plain literal "aaa"@ttt in V then $I(E) =$ `<aaa,ttt>`
- if E is a typed literal in V then $I(E)$=$IL(E)$
- if E is a URI reference in V then $I(E)$=$IS(E)$
- if E is a ground triple `s p o .` then $I(E) =$ true if s, p, o are in V, $I(p)$ is in IP and $< I(s), I(o) >$ is in $IEXT(I(p))$.
- if E is a ground RDF graph then $I(E) =$ false if $I(E') =$ *false for some triple* E' *in* E, otherwise $I(E) =$ true.

Note:

- The graph $\emptyset$ is true in any denotation
- If a name in a Graph G is not in V, then $I(E)$ is false.

Extension of RDF Denotations Towards Blank Nodes

- $blank(E)$: set of blank nodes of an RDF graph E.
- let I be an interpretation, A a mapping from some set of blank nodes to the universe IR of I.
- $[I + A]$ denotes an *extended interpretation* which is like I, but maps any blank node B to $A(B)$
- if E is an RDF graph then $I(E) = true$ if $[I + A'](E) = true$ *for some mapping* A' from $blank(E)$ to IR.

Example:
`(_:x1 <ex:a> <ex:b>), (<ex:c> <ex:b> _:x1)`

Semantics of Reification

The triple `<ex:a> <ex:b> <ex:c>` is reified by a blank node `_:xxx` and the four statements

- `_:xxx rdf:type rdf:Statement`
- `_:xxx rdf:subject <ex:a>`
- `_:xxx rdf:predicate <ex:b>`
- `_:xxx rdf:object <ex:c>`

The graph of the reified statement is true in an interpretation I if: `<x,y> is in IEXT(I(rdf:subject))` just when x is a token of an RDF Triple of the form `aaa bbb ccc"` and y is in `I(aaa)`.

RDF Containers

RDF Containers comprise *sequences*, *bags* and *alternatives*.

- vocabulary for *describing* containers, rather than constructing.
- the actual containers are entities in the universe
- many natural assumptions about RDF containers are not reflected in the RDF model theory
- no special semantic conditions on container vocabulary
- `rdf:seq`, `rdf:bag`, `rdf:alt`, `rdf:_1`, `rdf:_2` ...

RDF Containers (continued)

```
_:xxx rdf:type rdf:Seq .
_:xxx rdf:_1 <ex:a> .
_:xxx rdf:_3 <ex:c> .
```

- The RDF graph above does not entail the triple `_:xxx rdf:_2 _:yyy .`
- A container may have infinitely many members.
- There is no possibility to close a container.
- Unintuively, a bag does not entail a variant obtained by permutation of its members.
- a blank node may represent a bag, sequence, alternative and a reified statement at the same time.

RDF Collections

The *existence* of a list such as `[ex:aaa, ex:bbb]` is *asserted* in RDF by the triples

- `_:c1 rdf:type rdf:List`
- `_:c1 rdf:first ex:aaa`
- `_:c1 rdf:rest _:c2`
- `_:c2 rdf:first ex:bbb`
- `_:c2 rdf:rest rdf:nil`

- The RDF Graph above is just an assertion that something of this structure exists, it does not represent a list by itself.
- RDF does not impose any well-formedness conditions on the use of RDF collections, meaning that a blank node may have multiple outgoing `rdf:first` or `rdf:rest` edges.
- RDF**S** semantics demand that any subject of `rdf:first` and any subject or object of `rdf:rest` is of `rdf:type rdf:List`.

Simple Entailment

- *S simply entails E*, if every model of *S* is also a model of *E*.
- Any transformation *S* ---> *E* is *simply valid* if *S* entails *E* in every case. Otherwise it is *invalid*.

Empty Graph Lemma

The empty set of triples is entailed by any graph.

Subgraph Lemma

A graph entails all its subgraphs.

Instance Lemma

A graph is entailed by any of its instances.

Merging Lemma

The merge of a set S of RDF graphs is entailed by S, and entails every member of S

Simple Entailment (continued)

Interpolation Lemma

S *simply entails* a graph E if and only if a subgraph of S is an instance of E.

- An RDF graph G is *lean* if it has no instance, which is a proper subgraph of G

Anonymity Lemma

Let E be a lean graph and E' a proper instance of E. Then E does not entail E'.

Compactness Lemma

If S entails E and E is a finite graph, then some finite subset S' of S entails E.

Decidability of Simple RDF Entailment

Given a set S of RDF graphs and another RDF graph G, does $S \models G$?

- find an instance of G which is a subgraph of the merge of S.
- use b-nodes as existential variables in subgraph matching

The algorithm above is valid and complete (Interpolation Lemma).
$\Rightarrow$ *simple RDF entailment is decidable*.

- Deciding entailment of simple RDF Graphs is NP-complete
- Deciding equivalence of RDF Graphs is isomorphism-complete
- Let n be the number of b-nodes within an RDF graph of size v. Then the tests can be carried out in $O(v^n)$.

Conclusion and Outlook

What we have seen today:

- Formal reasoning on the Web may dramatically change the world we live in today
- There are four ingredients for this change to happen:
  - the concept of URIs for uniquely identifying resources on the Web, but also concepts not present on the Web
  - a standardized data format with a well-defined semantics
  - querying and reasoning languages for inferring new knowledge from existing
  - software agents that use these specialized languages to fulfill reasoning tasks for people

Semantic Web Queries
SPARQL & Beyond

Slides based on Lectures by N. Henze & D. Krause, T. Furche & B. Linse & F. Bry & D. Plexousakis & G. Gottlob, M. Arenas & C. Gutierrez & B. Parsia & J. Perez & A. Polleres & A. Seaborne

Tim Furche

Vorlesung: “Markup-Sprachen und semi-strukturierte Daten”, Prof. Dr. F. Bry, SS 2008

We have XML query languages. RDF is XML. Why don’t we use XML query languages to query RDF?

- RDF aware: query for RDF triples
  - Subject, Predicate, Object
- No knowledge about the storage structure required
- Ontology support
  - Reasoning
  - support for class hierarchies
- Entailment

Why RDF query languages?

Definition (Entailment)

The set A entails the set B if and only if, in every model in which all sentences in A are true, all sentences in B are also true.

Notation: $A \models B$ - a set A entails a set B.

Entailment in RDF queries:

$$KB \models Q$$

KB: RDF knowledge base
Q: query

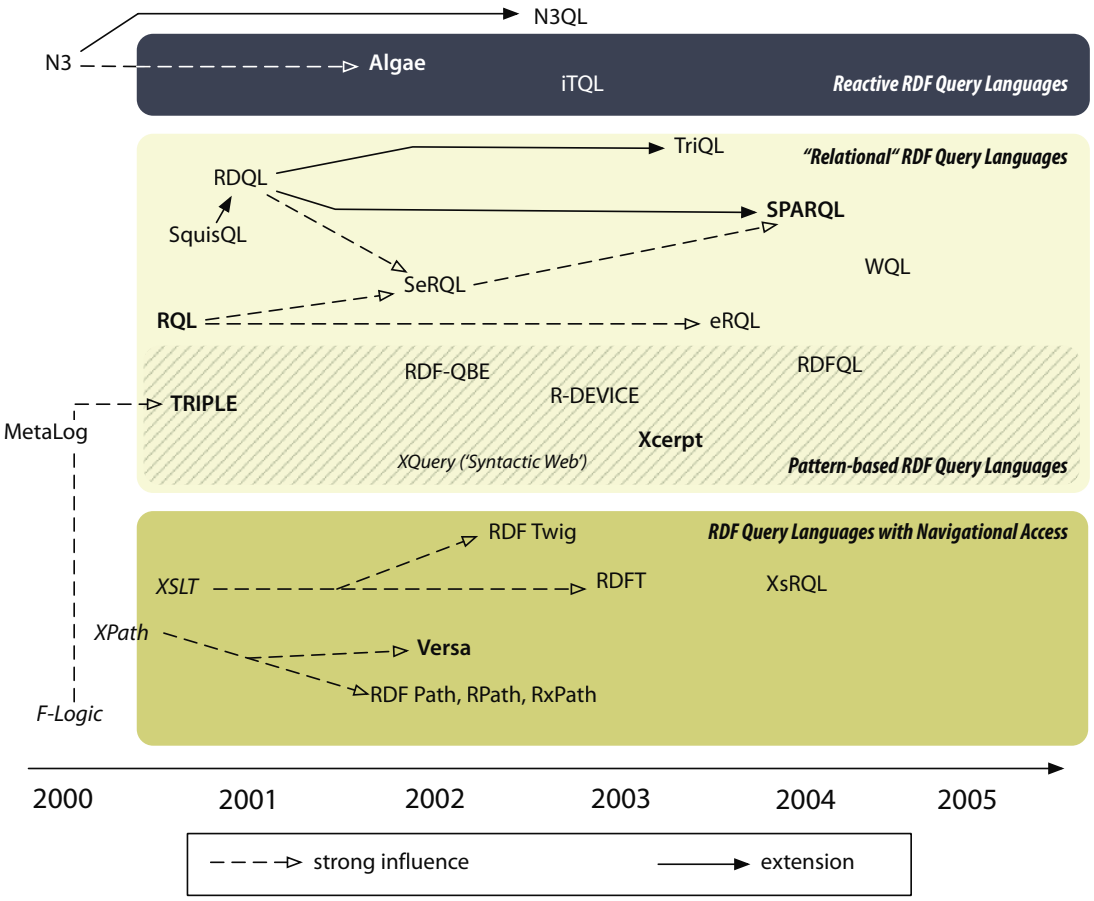
A query is true if it is entailed in the RDF source graph.

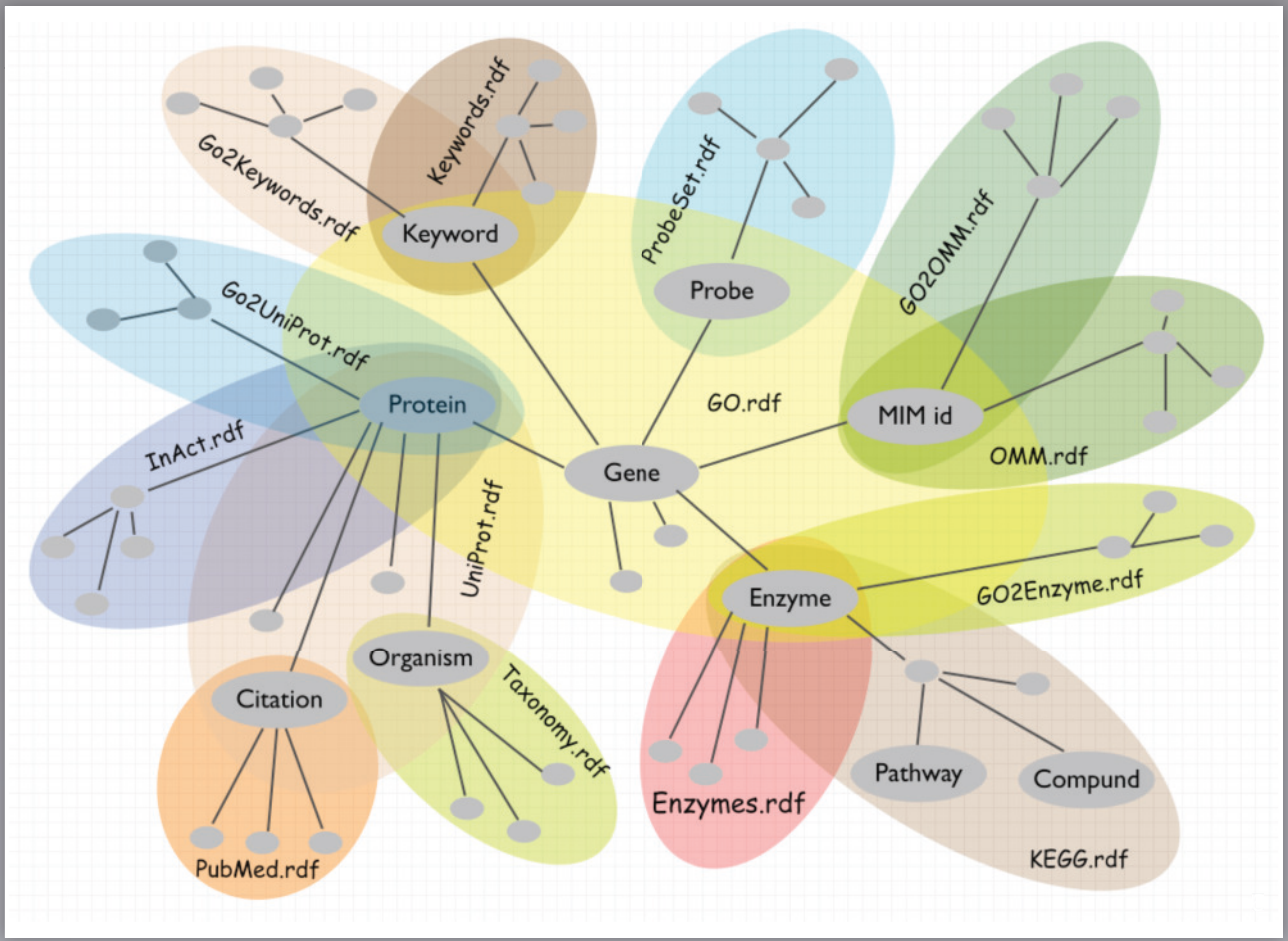
RDF **Entailment**

- Reactive rule query languages
  - Algae
- Navigational access query languages
  - Versa
- Relational query languages
  - SPARQL, RDQL, RQL, SeRQL
- Pattern-based query languages
  - TRIPLE, Xcerpt

slides are based on T. Furche et al. - RDF Querying: Language Constructs and Evaluation Methods Compared, 2006, Springer

Taxonomy for RDF query languages


N3QL
N3
Algae
iTQL
Reactive RDF Query Languages
TriQL
"Relational" RDF Query Languages
RDQL
SPARQL
SquisQL
WQL
SeRQL
RQL
eRQL
RDF-QBE
RDFQL
R-DEVICE
TRIPLE
MetaLog
Xcerpt
XQuery ('Syntactic Web')
Pattern-based RDF Query Languages
RDF Twig
RDF Query Languages with Navigational Access
XSLT
RDFT
XsRQL
XPath
Versa
RDF Path, RPath, RxPath
F-Logic
2000
2001
2002
2003
2004
2005
strong influence
extension

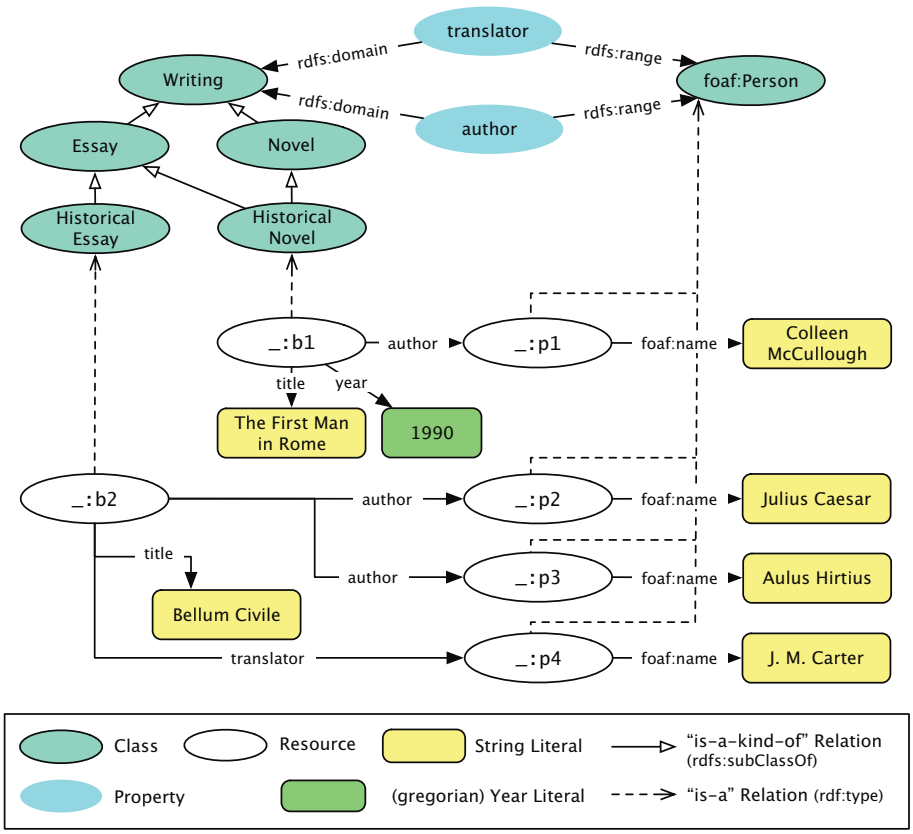

Go2Keywords.rdf
Keywords.rdf
Keyword
ProbeSet.rdf
Probe
GO2OMM.rdf
Go2UniProt.rdf
Protein
GO.rdf
MIM id
InAct.rdf
OMM.rdf
Gene
UniProt.rdf
Enzyme
GO2Enzyme.rdf
Organism
Taxonomy.rdf
Citation
Enzymes.rdf
Pathway
Compund
PubMed.rdf
KEGG.rdf

A Exemplars

Two main concepts:

1. Actions
   - `ask` - queries an RDF graph
   - `assert` - adds new statements to an existing RDF graph
   - `fwrule` - combines `ask` and `assert` actions, performs like a database Trigger
2. answers
   - bindings for query variables
   - RDF triples as proof

Reactive Rule Language – **Algae**



Example (An Algae query)

Answer:

?title	?translator	Proof

Bellum Civile	J. M. Carter	`_:1 rdf:type <http://exam...ks-rdfs#Essay>.`
		`_:1 books:author _:2.`
		`_:2 books:authorName "Julius Caesar".`
		`_:1 books:title Bellum Civile".`
		`_:1 books:translator "J. M. Carter".`

Example (fwrule)

Every writing has an author:

```
fwrule ask (?writing rdf:type <http://example.org/books#Writing>)
    assert (?writing books:author ?name)
```

More examples can be found at
`http://www.w3.org/2004/05/06-Algae/`

Algae Example II

Example (An Algae query)

Return title and optional translator of the Essay written by Julius Caesar.

```
ns rdf = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
ns books = <http://example.org/books#>

read <http://example.org/books> ()
ask ( ?essay rdf:type <http://example.org/books#Essay> .
      ?essay books:author ?author .
      ?author books:authorName "Julius Caesar" .
      ?essay books:title ?title .
     ~?essay books:translator ?translator
     )
collect( ?title, ?translator )
```

Algae Example

Algae

Critique

- No selection or extraction of **arbitrary extent** subgraphs
- No graph **construction**, no aggregation, no grouping
- Very **limited implementations**

- Same idea as XPATH (navigate trough a graph), but different implementation.
- Main technique: **traversal**
  - Forward traversal (returns objects):
    `subject selection - predicate selection -> object selection`
  - Backward traversal (returns subjects):
    `subject selection <- predicate selection - object selection`
- Powerful build-in functions: `distribute`, `filter`, `union`, `difference`, ...
- Extension by externally defined functions

Remark

Backward traversal can be simplified to:
`subject selection |- predicate selection -> object selection`

Versa examples from
`http://uche.ogbuji.net/tech/rdf/versa/versa-by-example`

Navigational access – **Versa**

Example

Get all resources:
```
all()
```

Get all author's names:
```
all() - foaf:name -> *
```

Get all author's names that contain "Ca":
```
all() - foaf:name -> contains("Ca")
```

Get the name of all translators
```
all() - books:translator -> foaf:name -> *
```

Get the names of all essays:
```
(books:Essay <- rdf:type - *) - books:title -> *
```

Versa Example I

- **distribute()** combines several queries into a list of result-lists

Example (distribute() function)

List all author's names and writing's titles

```
distribute(all(), ". - foaf:name -> *", ". - title -> *")
```

Result:

```
<List>
  <List>
    <List>
      <String>Colleen McCullough</String>
    </List>
    <List>
      <String>The First Man in Rome</String>
    </List>
  </List>
  <List>
    <List>
      <String>Julius Caesar</String>
    </List>
...
```

Versa Example II

- **filter()** similar to `distribute()` function – filters the result list

Example (filter() function)

Select all essays entitled "Bellum Civile" having a translator named "J. M. Carter"

```
filter(books:Essay <- rdf:type - *,
    ". - books:title -> eq(Bellum Civile)",
        ". -  books:translator -> foaf:name
           -> eq(J. M. Carter)"
```

Versa Example III

Versa

Critique

- lack of procedural abstraction mechanism
- very **unfamiliar** syntax
- **construction** of graphs limited

- **Selection queries** *Select all Essays together with their authors (i.e. author items and corresponding names)*
- **Extraction queries** *Select all data items with any relation to the book titled "Bellum Civile"*
- **Reduction queries** *Select all data items except ontology information and translators from the book recommender system*
- **Restructuring queries** *Invert the relation author (from a book to an author) into a relation authored (from an author to a book)*
- **Aggregation queries** *Return the last year in which an author with name "Julius Caesar" published something*
- **Combination and inference queries** *Combine the information about the book titled "The Civil War" and authored by "Julius Caesar" with the information about the book with identifier bellum_civile. Return the transitive closure of the subClassOf relation*

Expressiveness: Query Types

Query Name	Algae	Versa	SPARQL	RQL
Selection queries	yes	yes	yes	yes
Extraction queries	no	yes	no	no
Reduction queries	yes	yes	yes	yes
Restructuring queries	no	no	yes	yes
Aggregation queries	no	yes	no	yes
Combination and inference queries	yes	no	no	yes

Evaluation

B
SPARQL


```
SELECT ?Name ?Email
WHERE
{
  ?X :name ?Name
  ?X :email ?Email
}
```

- The *semantics* of simple SPARQL queries is easy to understand, at least intuitively.

> *"Give me the name and email of the resources in the datasource"*

SPARQL: **Introduction**

- Turtle: An RDF serialization
  - The RDF part of N3
  - Commonly used in examples (and tutorials and papers)
  - SPARQL uses Turtle+variables as triple pattern syntax

```
@prefix person: <http://example/person/> .
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .

person:A foaf:name "Alice" .
person:A foaf:mbox <mailto:alice@example.net> .

person:B foaf:name "Bob" .
```

SPARQL: Syntax


```
@prefix person: <http://example/person/> .
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .

person:A foaf:name "Alice" .
person:A foaf:mbox <mailto:alice@example.net> .
person:B foaf:name "Bob" .
```

```
PREFIX person: <http://example/person/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name
WHERE
  { ?x foaf:name ?name }
```

```
-----------
| name    |
===========
| "Bob"   |
| "Alice" |
-----------
```

SPARQL: Triple **Pattern**

```
@prefix person: <http://example/person/> .
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .

person:A foaf:name "Alice" .
person:A foaf:mbox <mailto:alice@example.net> .
person:B foaf:name "Bob" .
```

```
PREFIX person: <http://example/person/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name
WHERE
  { ?person foaf:mbox <mailto:alice@example.net> .
    ?person foaf:name ?name . }
```

```
-----------
| name    |
===========
| "Alice" |
-----------
```

SPARQL: **Graph Pattern**


```
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix stock: <http://example.org/stock#> .
@prefix inv: <http://example.org/inventory#> .

stock:book1 dc:title "SPARQL Query Language Tutorial" .
stock:book1 inv:price 10 .
stock:book1 inv:quantity 3 .

stock:book2 dc:title "SPARQL Query Language (2nd ed)" .
stock:book2 inv:price 20 ; inv:quantity 5 .

stock:book3 dc:title "Moving from SQL to SPARQL" .
stock:book3 inv:price 5 ; inv:quantity 0 .

stock:book4 dc:title "Applying XQuery" .
stock:book4 inv:price 20 ; inv:quantity 8 .
```

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX stock: <http://example.org/stock#>
PREFIX inv: <http://example.org/inventory#>

SELECT ?book ?title
WHERE {
 ?book dc:title ?title .
 ?book inv:price ?price . FILTER ( ?price < 15 )
 ?book inv:quantity ?num . FILTER ( ?num > 0 ) }
```

```
 ------------------------------------------------
 | book        | title                          |
 ================================================
 | stock:book1 | "SPARQL Query Language Tutorial" |
------------------------------------------------
```

SPARQL: **Filter**

```
@prefix person: <http://example/person/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

person :a foaf:name "Alice" .
person :a foaf:nick "A-online" .

person:b foaf:name "Bob" .
```

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?nick
{ ?x foaf:name ?name .
   OPTIONAL {?x foaf:nick ?nick }
}
```

```
------------------------
| name    | nick       |
========================
| "Alice" | "A-online" |
| "Bob"   |            |
------------------------
```

SPARQL: **Optional**


```
@prefix book: <http://example/book/> .
@prefix dc10: <http://purl.org/dc/elements/1.0/> .
@prefix dc11: <http://purl.org/dc/elements/1.1/> .

book:a   dc10:title "SPARQL Query Language Tutorial" .
book:b   dc11:title "SPARQL Query Language (2nd ed)" .
book:c   dc10:title "SPARQL" .
book:c   dc11:title "SPARQL" .
```

```
PREFIX dc10: <http://purl.org/dc/elements/1.0/>
PREFIX dc11: <http://purl.org/dc/elements/1.1/>

SELECT DISTINCT ?title
{
   { ?book dc10:title ?title } UNION { ?book dc11:title ?title }
}
```

```
-------------------------------------
| title                             |
=====================================
| "SPARQL Query Language Tutorial"  |
| "SPARQL"                          |
| "SPARQL Query Language (2nd ed)"  |
-------------------------------------
```

SPARQL: **Union**

```
@prefix person: <http://example/person/> .
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .

person:a foaf:name "Alice" .
person:a foaf:mbox <mailto:alice@example.net> .
person:b foaf:name "Bob" .
```

```
PREFIX person: <http://example/person/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>

CONSTRUCT { ?person vcard:FN ?name }
WHERE
  {?person foaf:name ?name . }
```

```
@prefix person: <http://example/person/> .
@prefix vcard: <http://www.w3.org/2001/vcard-rdf/3.0#>

person:a vcard:FN "Alice" .
person:b vcard:FN "Bob" .
```

SPARQL: **Construction**

c Semantics

- Triple patterns: RDF triple + variables (no bnodes for now)

(?X, name, ?$Name$)

- The base case for the algebra is a set of triple patterns

$$\{t_1, t_2, \ldots, t_k\}.$$

This is called basic graph pattern (BGP).

Example

{ (?X, name, ?$Name$), (?X, email, ?$Email$) }

SPARQL: Basic **Graph Patterns**

Definition

A mapping is a partial function from variables to RDF terms.

Given a mapping μ and a basic graph pattern P:

- $\text{dom}(\mu)$: the domain of μ.
- $\mu(P)$: the set obtained from P replacing the variables according to μ

Example

$$\mu = \{?X \to R_1, ?Y \to R_2, ?Name \to \text{john}, ?Email \to \text{J@ed.ex}\}$$

$$P = \{(?X, \text{name}, ?Name), (?X, \text{email}, ?Email)\}$$

$$\mu(P) = \{(R_1, \text{name}, \text{john}), (R_1, \text{email}, \text{J@ed.ex})\}$$

SPARQL Semantics: **Mapping**

Definition

The evaluation of the BGP P over a graph G, denoted by $[\![P]\!]_G$, is the set of all mappings μ such that:

- $\text{dom}(\mu)$ is exactly the set of variables occurring in P
- $\mu(P) \subseteq G$

Example

G

$(R_1, \text{name}, \text{john})$
$(R_1, \text{email}, \text{J@ed.ex})$
$(R_2, \text{name}, \text{paul})$

$$[\![\{(?X, \text{name}, ?Y)\}]\!]_G$$

$$\left\{ \begin{array}{l} \mu_1 = \{?X \to R_1, ?Y \to \text{john}\} \\ \mu_2 = \{?X \to R_2, ?Y \to \text{paul}\} \end{array} \right\}$$

	?X	?Y
μ_1	R_1	john
μ_2	R_2	paul

$$[\![\{(?X, \text{name}, ?Y), (?X, \text{email}, ?Z)\}]\!]_G$$

$$\{ \mu = \{?X \to R_1, ?Y \to \text{john}, ?Z \to \text{J@ed.ex}\} \}$$

	?X	?Y	?Z
μ	R_1	john	J@ed.ex

Example

G

$(R_1, \text{name}, \text{john})$
$(R_1, \text{email}, \text{J@ed.ex})$
$(R_2, \text{name}, \text{paul})$

$[\![\{(R_1, \text{webPage}, ?W)\}]\!]_G$
$\{\ \ \}$

$[\![\{(R_3, \text{name}, \text{ringo})\}]\!]_G$
$\{\ \ \}$

$[\![\{(R_2, \text{name}, \text{paul})\}]\!]_G$
$\{\ \mu_\emptyset = \{\ \}\ \}$

$[\![\{\ \}]\!]_G$
$\{\ \mu_\emptyset = \{\ \}\ \}$

SPARQL Semantics: Example II

Definition

The mappings μ_1, μ_2 are compatibles iff they agree in their shared variables:

- $\mu_1(?X) = \mu_2(?X)$ for every $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$.

$\mu_1 \cup \mu_2$ is also a mapping.

Example

	$?X$	$?Y$	$?U$	$?V$
μ_1	R_1	john		
μ_2	R_1		J@edu.ex	
μ_3			P@edu.ex	R_2
$\mu_1 \cup \mu_2$	R_1	john	J@edu.ex	
$\mu_1 \cup \mu_3$	R_1	john	P@edu.ex	R_2

$\mu_\emptyset = \{\ \}$ is compatible with every mapping.

Compatible **Mappings**

Let M_1 and M_2 be sets of mappings:

Definition

Join: $M_1 \bowtie M_2$

- $\{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1, \mu_2 \in M_2,$ and μ_1, μ_2 are compatibles$\}$
- extending mappings in M_1 with compatible mappings in M_2

will be used to define AND

Definition

Union: $M_1 \cup M_2$

- $\{\mu \mid \mu \in M_1$ or $\mu \in M_2\}$
- mappings in M_1 plus mappings in M_2 (the usual set union)

will be used to define UNION

Join and Union for SPARQL

Definition

Difference: $M_1 \setminus M_2$

- $\{\mu \in M_1 \mid$ for all $\mu' \in M_2$, μ and μ' are not compatibles$\}$
- mappings in M_1 that cannot be extended with mappings in M_2

Definition

Left outer join: $M_1 \mathbin{⟕} M_2 = (M_1 \bowtie M_2) \cup (M_1 \setminus M_2)$

- extension of mappings in M_1 with compatible mappings in M_2
- plus the mappings in M_1 that cannot be extended.

will be used to define OPT

Difference and Left Outer Join

Definition

Given a graph G the evaluation of a pattern is recursively defined

- $[\![(P_1 \text{ AND } P_2)]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$
- $[\![(P_1 \text{ UNION } P_2)]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$
- $[\![(P_1 \text{ OPT } P_2)]\!]_G = [\![P_1]\!]_G \mathbin{⟕} [\![P_2]\!]_G$

the base case is the evaluation of a BGP.

SPARQL Semantics: **General Graph Patterns**

Example (AND)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![\{(?X, \text{name}, ?N)\} \text{ AND } \{(?X, \text{email}, ?E)\}]\!]_G$$

$$[\![\{(?X, \text{name}, ?N)\}]\!]_G \bowtie [\![\{(?Y, \text{email}, ?E)\}]\!]_G$$

	?X	?N
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

$\bowtie$

	?X	?E
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

	?X	?N	?E
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex

Example (OPT)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![\{(?X, \text{name}, ?N)\} \text{ OPT } \{(?X, \text{email}, ?E)\}]\!]_G$$

$$[\![\{(?X, \text{name}, ?N)\}]\!]_G \bowtie [\![\{(?X, \text{email}, ?E)\}]\!]_G$$

	?X	?N
μ_1	R_1	john
μ_2	R_2	paul
μ_3	R_3	ringo

$\bowtie$

	?X	?E
μ_4	R_1	J@ed.ex
μ_5	R_3	R@ed.ex

	?X	?N	?E
$\mu_1 \cup \mu_4$	R_1	john	J@ed.ex
$\mu_3 \cup \mu_5$	R_3	ringo	R@ed.ex
μ_2	R_2	paul	

Example (UNION)

G : $(R_1, \text{name}, \text{john})$ $(R_1, \text{email}, \text{J@ed.ex})$ $(R_2, \text{name}, \text{paul})$ $(R_3, \text{name}, \text{ringo})$ $(R_3, \text{email}, \text{R@ed.ex})$ $(R_3, \text{webPage}, \text{www.ringo.com})$

$$[\![\{(?X, \text{email}, ?Info)\} \text{ UNION } \{(?X, \text{webPage}, ?Info)\}]\!]_G$$

$$[\![\{(?X, \text{email}, ?Info)\}]\!]_G \cup [\![\{(?X, \text{webPage}, ?Info)\}]\!]_G$$

	?X	?Info
μ_1	R_1	J@ed.ex
μ_2	R_3	R@ed.ex

$\cup$

	?X	?Info
μ_3	R_3	www.ringo.com

	?X	?Info
μ_1	R_1	J@ed.ex
μ_2	R_3	R@ed.ex
μ_3	R_3	www.ringo.com

- One of the interesting features of SPARQL is that a query may retrieve data from different sources.

Definition

A SPARQL dataset is a set

$$\mathcal{D} = \{G_0, \langle u_1, G_1\rangle, \langle u_2, G_2\rangle, \ldots, \langle u_n, G_n\rangle\}$$

- G_0 is the default graph, $\langle u_i, G_i\rangle$ are named graphs
- $\text{name}(\mathcal{D}) = \{u_1, u_2, \ldots, u_n\}$
- $d_{\mathcal{D}}$ is a function such $d_{\mathcal{D}}(u_i) = G_i$.

SPARQL Semantics: **Named Graphs**

if u is an IRI, $?X$ is a variable and P is a graph pattern, then

- (u GRAPH P) is a graph pattern
- ($?X$ GRAPH P) is a graph pattern

GRAPH will permit us to dynamically change the graph against which our pattern is evaluated.

SPARQL Semantics: **Named Graphs**

Definition

Given a dataset $\mathcal{D}$ and a graph pattern P

$$[\![(u \text{ GRAPH } P)]\!]_G = [\![P]\!]_{d_{\mathcal{D}}(u)}$$

$$[\![(?X \text{ GRAPH } P)]\!]_G = \bigcup_{u \in \text{name}(\mathcal{D})} \left([\![P]\!]_{d_{\mathcal{D}}(u)} \bowtie \{\{?X \to u\}\} \right)$$

Definition

The evaluation of a general pattern P against a dataset $\mathcal{D}$, denoted by $[\![P]\!]_{\mathcal{D}}$, is the set $[\![P]\!]_{G_0}$ where G_0 is the default graph in $\mathcal{D}$.

SPARQL Semantics: **Named Graphs**

Example (GRAPH)

$\mathcal{D}$

G_0:

$\langle$ tb, G_1: (R1, name, john) (R1, email, J@ed.ex) (R2, name, paul) $\rangle$

$\langle$ trs, G_2: (R4, name, mick) (R4, email, M@ed.ex) (R5, name, keith) (R5, email, K@ed.ex) $\rangle$

$$[\![(?G \text{ GRAPH } \{(?X, \text{name}, ?N)\})]\!]_{\mathcal{D}}$$

$$[\![\{(?X, \text{name}, ?N)\}]\!]_{G_1} \bowtie \{\{?G \to \text{tb}\}\} \cup [\![\{(?X, \text{name}, ?N)\}]\!]_{G_2} \bowtie \{\{?G \to \text{trs}\}\}$$

μ_1, μ_2:

?X	?N
R1	john
R2	paul

$\bowtie \{\{?G \to \text{tb}\}\} \cup$

μ_3, μ_4:

?X	?N
R4	mick
R5	keith

$\bowtie \{\{?G \to \text{trs}\}\}$

?G	?X	?N
tb	R1	john
tb	R2	paul
trs	R4	mick
trs	R5	keith

- Up to this point we have concentrated in the body of a SPARQL query, i.e. in the graph pattern matching expression.
- A query can also process the values of the variables. The most simple processing operation is the selection of some variables appearing in the query.

Definition

- A SELECT query is a tuple (W, P) where P is a graph pattern and W is a set of variable.
- The answer of a SELECT query against a dataset $\mathcal{D}$ is

$$\{\mu_{|W} \mid \mu \in [\![P]\!]_{\mathcal{D}}\}$$

where $\mu_{|W}$ is the restriction of μ to domain W.

SPARQL Semantics: **Select**

- A query can also output an RDF graph.
- The construction of the output graph is based on a template.
- A template is a set of triple patterns possibly with bnodes.

Example

$$T_1 = \{(?X, \text{name}, ?Y), (?X, \text{info}, ?I), (?X, \text{addr}, B)\}$$

with B a bnode

Definition

- A CONSTRUCT query is a tuple (T, P) where P is a graph pattern and T is a template.

SPARQL Semantics: **Construct I**

Definition

The answer of a CONSTRUCT query (T, P) against a dataset $\mathcal{D}$ is obtained by

- for every $\mu \in [[P]]_{\mathcal{D}}$ create a template T_μ with fresh bnodes
- take the union of $\mu(T_\mu)$ for every $\mu \in [[P]]_{\mathcal{D}}$
- discard the not valid RDF triples
  - some variables have not been instantiated.
  - bnodes in predicate positions

SPARQL Semantics: **Construct** II

Definition

The answer of a CONSTRUCT query (T, P) against a dataset $\mathcal{D}$ is obtained by

- for every $\mu \in [[P]]_{\mathcal{D}}$ create a template T_μ with fresh bnodes
- take the union of $\mu(T_\mu)$ for every $\mu \in [[P]]_{\mathcal{D}}$
- discard the not valid RDF triples
  - some variables have not been instantiated.
  - bnodes in predicate positions

SPARQL Semantics: **Construct** II

- Not a fixed output schema
  - mappings instead of tables
  - schema is implicit in the domain of mappings
- Too many NULLs
  - mappings with disjoint domains can be joined
  - mappings with distinct domains in output solutions
- SPARQL-to-SQL translations experience this issues
  - need of `IS NULL`/`IS NOT NULL` in join/outerjoin conditions
  - need of `COALESCE` in constructing output schema

Difference to **SQL**

D
Complexity

Input:

A mapping μ, a graph pattern P, and an RDF graph G.

Question:

Is the mapping in the evaluation of the pattern against the graph?

$$\mu \in [\![P]\!]_G?$$

SPARQL **Complexity**

Theorem (PAG06)

For patterns using only AND *and* FILTER *operators, the evaluation problem is polynomial:*

$$O(|P| \times |G|).$$

Proof idea

- *Check that the mapping makes every triple to match.*
- *Then check that the mapping satisfies the* FILTER*s.*

SPARQL **Complexity**

can be reduced to **conjunctive queries**

Theorem (PAG06)

For patterns using only AND *and* FILTER *operators, the evaluation problem is polynomial:*

$$O(|P| \times |G|).$$

Proof idea

- *Check that the mapping makes every triple to match.*
- *Then check that the mapping satisfies the* FILTER*s.*

SPARQL **Complexity**

Theorem (PAG06)

For patterns using AND, FILTER and UNION operators, the evaluation problem is NP-complete.

Proof idea

- *Reduction from 3SAT.*
- *A pattern encodes the propositional formula.*
- *¬ bound is used to encode negation.*

SPARQL **Complexity**

can be reduced to **unions** of **conjunctive queries**

Theorem (PAG06)

For patterns using AND, FILTER and OPT operators, the evaluation problem is PSPACE-complete.

Proof idea

- *Reduction from QBF*
- *A pattern encodes a quantified propositional formula:*

$$\forall x_1 \exists y_1 \forall x_2 \exists y_2 \cdots \psi.$$

- *nested OPTs are used to encode quantifier alternation.*
  (This time, we do not need ¬ bound.)

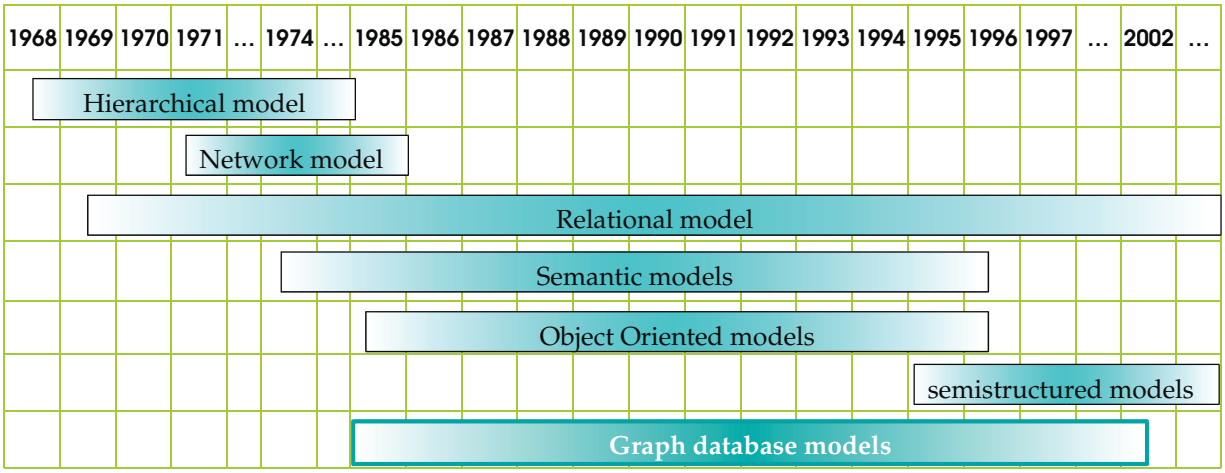
SPARQL **Complexity**

can be reduced to **unions** of **conjunctive queries** with **negation**
non-recursive datalog with negation

SPARQL
Complexity and Expressiveness

- essentially just like basic, **non-recursive SQL**
  - i.e., relational algebra
- **no** direct support for **reasoning** or RDF
- **no graph** queries

Graph Queries



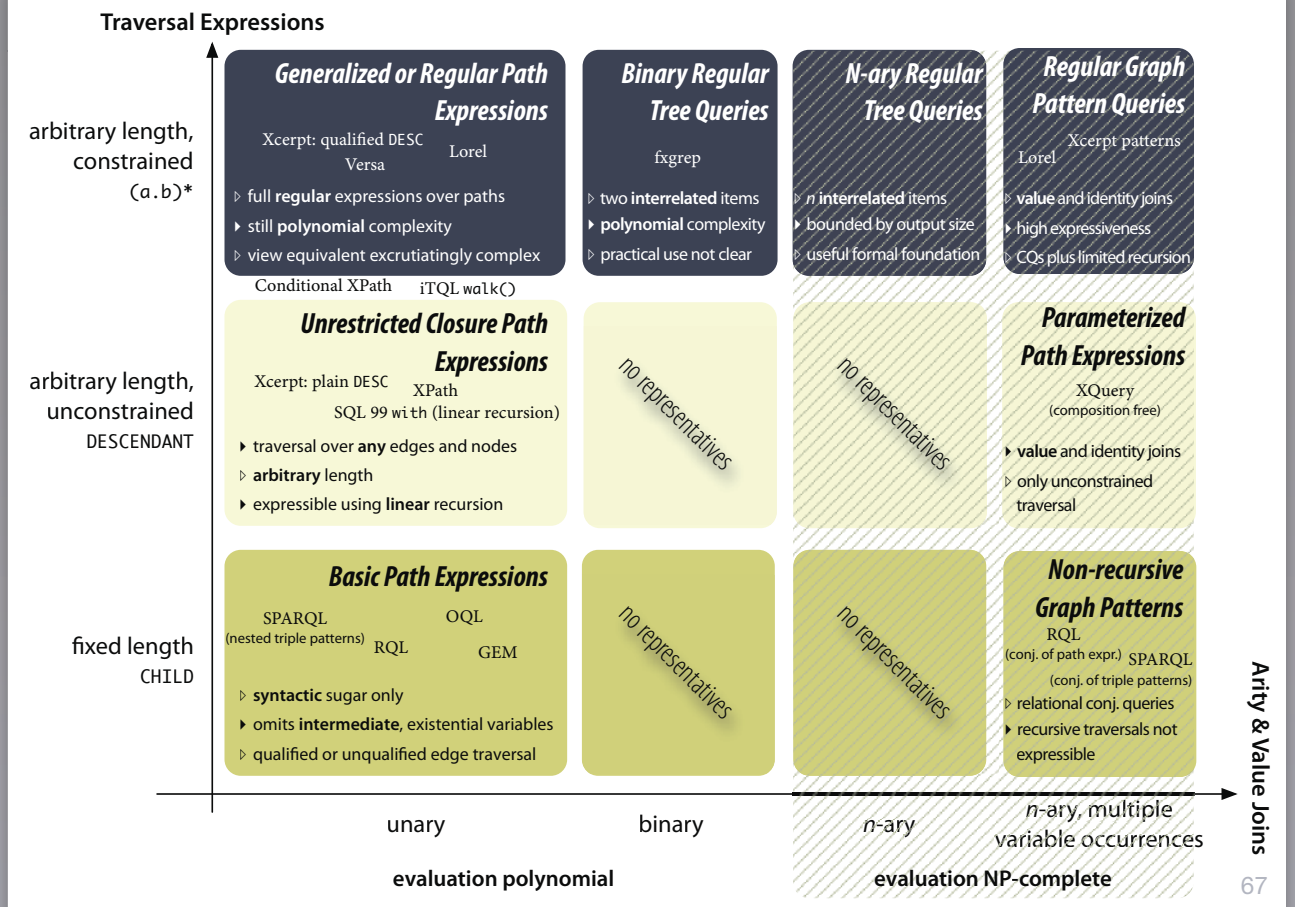
MODEL	LEVEL	COMPLEXITY OF DATA	CONNECTIVITY	TYPE OF DATA
Semantic	User	Simple / medium	High	Homogeneous
Semistructured	Logical	Medium	Medium	Heterogeneous
Graph	Logical	Medium	High	Heterogeneous
Semistructured	Logical	Medium	Medium	Heterogeneous
Graph	Logical	Medium	High	Heterogeneous

Graph notion	Real-life Application Query
Paths	**Are suspects A and B related?** (relevant paths in a police database)
	What is the shortest route between city A and city B? (Shortest path in a database of roads)
	What is the influence of article D? (transitive closure in database of bibliographic cites)
Distance	**What is the Erdös distance between author X and author Y?** (distance between nodes in a collaboration graph)
Pattern Matching	**Where and how much a motif (pattern) appears?** (Pattern matching in genome data)

RDF **Graph** Queries

Graph notion	Real-life Application Query
Adjacency	**All relatives of degree one of Alice** (adjacent nodes in a genealogy database)
	What chemical composes does a given chemical reaction produce? (Adjacent edges in chemical information)
	What cities are near Athens? (neighborhood in a tourism system graph)
Degree of a node	**What is/are the most cited paper/s?** (searching node/s with maximum in-degree in a database of bibliographic cites)

RDF **Graph** Queries



PROPERTY		Adjacent Nodes	Adjacent Edges	Degree of a Node	Path	Fixed-length path	Distance	Diameter
RDF Query Language	RQL							
	SeRQL							
	RDQL							
	Triple							
	N3							
	Versa							
	RxPath							

PROPERTY	RQL	SeRQL	RDQL	Triple	N3	Versa	RxPath
Adjacent nodes	+	+	±	±	+	+	–
Adjacent edges	+	+	±	±	+	–	–
Degree of a node	–	–	–	–	–	–	–
Path	–	–	–	–	–	–	–
Fixed-length Path	±	±	±	±	±	–	–
Distance between two nodes	–	–	–	–	–	–	–
Diameter	–	–	–	–	–	–	–

RDF **Graph** Queries

F + Rules

- We import all triples in a predicate `triple(Subj,Pred,Object,Graph)` which carries an additional argument for the dataset.
- For the import, we use the `rdf[URL](S,P,O)` built-in.

"select persons and their names"

```
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . }
```

```
triple(S,P,O,def) :- rdf["http://ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["http://alice.org"](S,P,O).
answer1(X,Y,def)  :- triple(X,"rdf:type","foaf:Person",def),
                     triple(X,"foaf:name",Y,def).

?- answer1(X,Y,def).
```

SPARQL to Logic Programming

- We import all triples in a predicate `triple(Subj,Pred,Object,Graph)` which carries an additional argument for the dataset.
- For the import, we use the `rdf[URL](S,P,O)` built-in.

"select persons and their names"

```
SELECT ?X ?Y
FROM <http://alice.org>
FROM <http://ex.org/bob>
WHERE { ?X a foaf:Person . ?X foaf:name ?Y . }
```

```
triple(S,P,O,def) :- rdf["http://ex.org/bob"](S,P,O).
triple(S,P,O,def) :- rdf["http://alice.org"](S,P,O).
answer1(X,Y,def)  :- triple(X,"rdf:type","foaf:Person",def),
                     triple(X,"foaf:name",Y,def).

?- answer1(X,Y,def).
```

SPARQL to Logic Programming

"select creators of graphs and the persons they know"

```
SELECT ?X ?Y
FROM <alice.org>
FROM NAMED <alice.org>
FROM NAMED <ex.org/bob>
WHERE { ?G foaf:maker ?X .
        GRAPH ?G { ?X foaf:knows ?Y. } }
```

```
triple(S,P,O,def) :- rdf["alice.org"](S,P,O).
triple(S,P,O,"alice.org") :- rdf["alice.org"](S,P,O).
triple(S,P,O,"ex.org/bob") :- rdf["ex.org/bob"](S,P,O).
answer1(X,Y,def) :- triple(G,"foaf:maker",X,def),
                    triple(X,"foaf:knows",Y,G).
```

SPARQL to Logic Programming

What if variables of the of constituent patterns don't coincide?
Slightly different than in SQL!
We emulate this by special `null` values!

```
SELECT ?X ?Y ?Z
FROM ...
WHERE { { ?X foaf:name ?Y . }
        UNION { ?X foaf:nick ?Z .} }
```

Data:

```
<alice.org#me> foaf:name "Alice".
<ex.org/bob#me> foaf:name "Bob"; foaf:nick "Bobby".
```

Result:

?X	?Y	?Z
<alice.org#me>	"Alice"	null
<ex.org/bob#me>	"Bob"	null
<ex.org/bob#me>	null	"Bobby"

$$\forall \bar{x}_1 \exists \bar{y}_1 \ldots \forall \bar{x}_n \exists \bar{y}_n \left(body(\bar{x}) \rightarrow head(\bar{x}, \bar{y}) \right)$$

$$P = \{ \forall sem \exists rm \forall stu \big((stu, \text{uni:attends}, sem) \rightarrow (sem, \text{uni:located\_in}, rm) \wedge (stu, \text{uni:knows}, rm) \big), \quad (1)$$

$$true \rightarrow (\text{uni:julie}, \text{uni:attends}, \text{'Logic'}) \wedge (\text{uni:john}, \text{uni:attends}, \text{uni:RDF}) \} \quad (2)$$

$$[\![P]\!] \ni \{ (\text{\_:b3}, \text{uni:located\_in}, \text{\_:b1}), (\text{uni:julie}, \text{uni:knows}, \text{\_:b1}), (\text{uni:RDF}, \text{uni:located\_in}, \text{\_:b2}), (\text{uni:john}, \text{uni:knows}, \text{\_:b2}), (\text{uni:julie}, \text{uni:attends}, \text{'Logic'}), (\text{uni:julie}, \text{uni:attends}, \text{\_:b3}), (\text{uni:john}, \text{uni:attends}, \text{uni:RDF}) \}$$

$$[\![P]\!] := \{ \mathsf{g} \in \mathsf{RDF} \mid \forall \mathsf{h} \in \mathsf{RDF} \, (P \models \varphi_{\mathsf{h}} \ \textit{iff} \ \varphi_{\mathsf{g}} \models \varphi_{\mathsf{h}}) \}$$

RDFLog: Taming Existence

$$s(P) = \{\forall sem \forall stu\big((stu, \text{uni:attends}, sem) \rightarrow (sem, \text{uni:located\_in}, s_{rm}(sem)) \wedge (stu, \text{uni:knows}, s_{rm}(sem))\big),$$
$$true \rightarrow (\text{uni:julie}, \text{uni:attends}, \text{'Logic'}) \wedge \text{uni:john}, \text{uni:attends}, \text{uni:RDF})\}$$

$$\varphi_{M_{s(P)}} = (\text{'Logic'}, \text{uni:located\_in}, s_{rm}(\text{'Logic'})) \wedge (\text{uni:julie}, \text{uni:knows}, s_{rm}(\text{'Logic'}))$$
$$\wedge (\text{uni:RDF}, \text{uni:located\_in}, s_{rm}(\text{uni:RDF})) \wedge (\text{uni:john}, \text{uni:knows}, s_{rm}(\text{uni:RDF}))$$
$$\wedge (\text{uni:julie}, \text{uni:attends}, \text{'Logic'}) \wedge (\text{uni:john}, \text{uni:attends}, \text{uni:RDF})$$

full RDFLog: already **Turing complete**

RDFLog: **Skolemization**

but: needs **un-Skolemization & Normalization** as post-processing operations

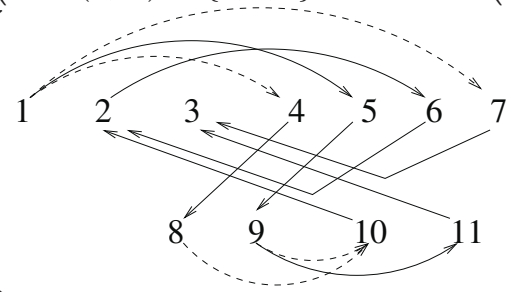
Fig. 1 Super-weakly acyclic program P and its dependency graph $\mathcal{G}(P)$

$$P = \begin{cases} \forall lec \exists crs \forall stu \ \big(\ (lec, \text{rdf:type}, \text{uni:lecture}) \wedge (stu \text{uni:attends}, lec) \\ \qquad \rightarrow (crs, \text{uni:practices}, lec) \wedge (stu, \text{uni:attends}, crs) \ \big) \\ \forall lec \forall crs \exists stu \ \big(\ (crs, \text{uni:practices}, lec) \rightarrow (stu, \text{uni:attends}, lec) \ \big) \end{cases}$$

$$s(P) = \begin{cases} (\underset{1}{lec}, \text{rdf:type}, \text{uni:lecture}) \wedge (\underset{2}{stu}, \text{uni:attends}, \underset{3}{lec}) \\ \qquad \rightarrow (\underset{4}{s_{crs}(lec)}, \text{uni:practices}, \underset{5}{lec}) \wedge (\underset{6}{stu}, \text{uni:attends}, \underset{7}{s_{crs}(lec)}) \\ (\underset{8}{crs}, \text{uni:practices}, \underset{9}{lec}) \rightarrow (\underset{10}{s_{stu}(crs, lec)}, \text{uni:attends}, \underset{11}{lec}) \end{cases}$$

$$\begin{cases} \text{LPos}(P) = \{1, 2, 3, 8, 9\} \\ \text{RPos}(P) = \{4, 5, 6, 7, 10, 11\} \\ \text{LPos}(5, P) = \{1, 2, 3\} \end{cases} \qquad \begin{cases} \text{term}(2) = stu \\ \text{term}(10) = s_{stu}(crs, lec) \\ \text{atom}(7) = T(stu, \text{uni:attends}, s_{crs}(lec)) \end{cases}$$

$\mathcal{G}(P) =$



Plain arrows:
- normal-edges and
- unification-edges

Dashed arrows:
- special-edges

$$F_{\forall\exists}(\phi) = \begin{cases} \phi & if\ \phi\ is\ in\ \forall\exists\ form \\ R^{\phi}_{proj} \wedge R^{\phi}_{gen} \wedge R^{\phi}_{join} & otherwise \end{cases} \quad where$$

$$R^{\phi}_{proj} = \forall \bar{x}, \bar{y}, \bar{z}\,(\psi(\bar{x}, \bar{y}, \bar{z}) \rightarrow \mathit{Proj}(\bar{x}))$$

$$R^{\phi}_{gen} = \forall \bar{x} \exists \bar{y}\,(\mathit{Proj}(\bar{x}) \rightarrow \mathit{Gen}(\bar{x}, \bar{y}))$$

$$R^{\phi}_{join} = \forall \bar{x}, \bar{y} Q \bar{z}\,(\psi(\bar{x}, \bar{y}, \bar{z}) \wedge \mathit{Gen}(\bar{x}, \bar{y}) \rightarrow \chi(\bar{x}, \bar{y}, \bar{z}))$$

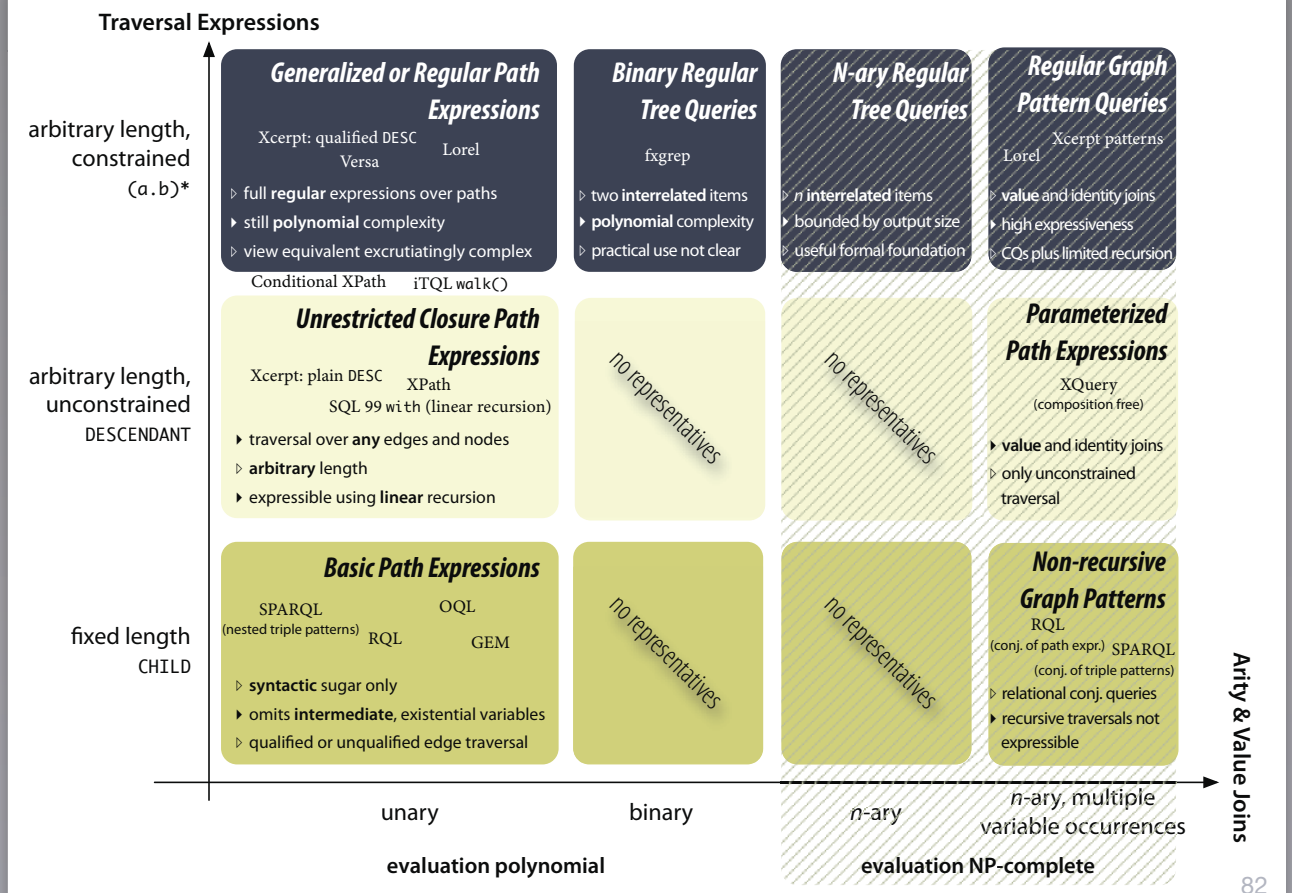
arbitrary quantifier alternation ≡ ∀∃ fragment

RDFLog Normalform

G
Fazit

RDF Query Languages
Summary I

- **SPARQL** by now the yardstick of RDF query languages
  - proper handling of optional and blank nodes
  - but: very limited **construction**
  - but: unnecessarily “hidden” **negation**
  - only equivalent to relational algebra: no graph queries or reasoning
- SPARQL + rules
  - several competing proposals
  - treatment of RDF specialities often limited
    - blank nodes treated fully only in **RDFLog**


Traversal Expressions
arbitrary length, constrained (a.b)*
arbitrary length, unconstrained DESCENDANT
fixed length CHILD
Generalized or Regular Path Expressions
Xcerpt: qualified DESC
Versa
Lorel
full regular expressions over paths
still polynomial complexity
view equivalent excrutiatingly complex
Conditional XPath
iTQL walk()
Binary Regular Tree Queries
fxgrep
two interrelated items
polynomial complexity
practical use not clear
N-ary Regular Tree Queries
n interrelated items
bounded by output size
useful formal foundation
Regular Graph Pattern Queries
Xcerpt patterns
Lorel
value and identity joins
high expressiveness
CQs plus limited recursion
Unrestricted Closure Path Expressions
Xcerpt: plain DESC
XPath
SQL 99 with (linear recursion)
traversal over any edges and nodes
arbitrary length
expressible using linear recursion
no representatives
no representatives
Parameterized Path Expressions
XQuery
(composition free)
value and identity joins
only unconstrained traversal
Basic Path Expressions
SPARQL
(nested triple patterns)
RQL
OQL
GEM
syntactic sugar only
omits intermediate, existential variables
qualified or unqualified edge traversal
no representatives
no representatives
Non-recursive Graph Patterns
RQL
(conj. of path expr.)
SPARQL
(conj. of triple patterns)
relational conj. queries
recursive traversals not expressible
Arity & Value Joins
unary
binary
n-ary
n-ary, multiple variable occurrences
evaluation polynomial
evaluation NP-complete

H

Fragen