



I4-D12

Xcerpt 2.0

Specification of the (Core) Language Syntax

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/I4-D12/D/PU/a1
Responsible editors:	Tim Furche
Reviewers:	Antonius Weinzierl and Jakob Henriksson
Contributing participants:	Munich
Contributing workpackages:	I4
Contractual date of deliverable:	28 February 2007
Actual submission date:	17 April 2007

Abstract

This article defines a revised syntax for the I4 query language, Xcerpt. Indeed, not only a single syntax, but rather three syntactical forms of Xcerpt are introduced: (1) the term syntax, a non-standard syntax that allows the succinct formulation of queries and is intended mostly for human authors; (2) the XML syntax provides a fine granular language markup in XML, ideal for processing through XML-based tools and for automated query generation or reasoning about query programs; (3) the compact XML syntax is a hybrid syntax of (1) and (2). The concepts are introduced UML. In addition to the formal syntax specification, principles of the syntax design are discussed. Furthermore, for a number of advanced constructs the reasoning supporting the design choice, as well as alternative solutions are illustrated. An impression of how the introduced language constructs allow to write and understand complex queries is given by numerous examples interspersed among the construct specifications.

We discuss several major revisions to Xcerpt's syntax from Deliverable I4-D6 in a second part: identity, grouping, and functions. We conclude with a report on ongoing work about modules and extended RDF support for Xcerpt. In an appendix we provide an ANTLR grammar for most of the core language used in the currently under-development Xcerpt 2.0 prototype.

Keyword List

reasoning, query language, Semantic Web, concepts, grammar, syntax, terms, XML

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

© REWERSE 2007.

Xcerpt 2.0

Specification of the (Core) Language Syntax

Tim Furche¹, François Bry¹, Sebastian Schaffert²

¹ Institute for Informatics, University of Munich, Germany
Email: {Francois.Bry,Tim.Furche}@ifi.lmu.de

² Salzburg Research, Salzburg, Austria
Email: {Sebastian.Schaffert@salzburgresearch.at}

17 April 2007

Abstract

This article defines a revised syntax for the I4 query language, Xcerpt. Indeed, not only a single syntax, but rather three syntactical forms of Xcerpt are introduced: (1) the term syntax, a non-standard syntax that allows the succinct formulation of queries and is intended mostly for human authors; (2) the XML syntax provides a fine granular language markup in XML, ideal for processing through XML-based tools and for automated query generation or reasoning about query programs; (3) the compact XML syntax is a hybrid syntax of (1) and (2). The concepts are introduced UML. In addition to the formal syntax specification, principles of the syntax design are discussed. Furthermore, for a number of advanced constructs the reasoning supporting the design choice, as well as alternative solutions are illustrated. An impression of how the introduced language constructs allow to write and understand complex queries is given by numerous examples interspersed among the construct specifications.

We discuss several major revisions to Xcerpt's syntax from Deliverable I4-D6 in a second part: identity, grouping, and functions. We conclude with a report on ongoing work about modules and extended RDF support for Xcerpt. In an appendix we provide an ANTLR grammar for most of the core language used in the currently under-development Xcerpt 2.0 prototype.

Keyword List

reasoning, query language, Semantic Web, concepts, grammar, syntax, terms, XML

Contents

1	Introduction	1
2	Meta-Syntax Notations for Abstract and Concrete Syntax	3
2.1	Abstract Syntax: UML Diagrams	3
2.2	Concrete Syntax: EBNF	4
2.3	Concrete Syntax: Relax NG	5
I	Definition of the Core Language	7
3	Xcerpt: A Versatile Web Query Language	9
3.1	Data Model	10
3.1.1	Terms for Representing Data and Queries	11
3.2	A Textual Non-XML Term Syntax for Xcerpt	11
3.2.1	Lexical Structures	12
3.2.2	Reserved Names	13
3.2.3	Whitespace and Comments	13
3.3	Hybrid XML-style Term Syntax	14
3.4	Pure XML Syntax	14
4	Specifying Semi-structured Data: Xcerpt's Data Terms	15
4.1	Defining Data Terms	15
4.1.1	Textual Term Syntax: Basic Data Terms	16
4.1.2	XML-style Term Syntax: Basic Data Terms	18
4.1.3	Pure XML Syntax: Basic Data Terms	18
4.2	Content Data Terms	20
4.2.1	Textual Term Syntax: Content Data Terms	21
4.2.2	XML-style Term Syntax: Content Data Terms	21
4.2.3	Pure XML Syntax: Content Data Terms	21
4.3	Structured Data Terms	22
4.3.1	Textual Term Syntax: Structured Data Terms	23
4.3.2	XML-style Term Syntax: Structured Data Terms	24
4.3.3	Pure XML Syntax: Structured Data Terms	25
4.4	Top-level Data Terms	27
4.4.1	Textual Term Syntax: Top-Level Data Terms	29
4.4.2	XML-style Term Syntax: Top-Level Data Terms	29

4.4.3	Pure XML Syntax: Top-Level Data Terms	29
4.5	Exemplary Data Term	29
4.6	XML Documents as Data Terms	30
5	How to specify queries?	
	Part 1: Construction	37
5.1	An Aside: A Parameterized Model for Terms	38
5.2	Specifying New Data: Construct Terms	40
5.2.1	Substitutions and Substitution Sets	40
5.3	The Shape of Construct Term	42
5.3.1	Textual Term Syntax	42
5.3.2	XML-style Term Syntax	46
5.3.3	Pure XML Syntax	46
5.4	Grouping in Construct Terms	47
5.4.1	Textual Term Syntax	48
5.4.2	XML-style Term Syntax	50
5.4.3	Pure XML Syntax	50
5.5	Optional Construct Terms	52
5.5.1	Textual Term Syntax	53
5.5.2	XML-style Term Syntax	53
5.5.3	Pure XML Syntax	53
5.6	Instantiating a Construct Term	54
6	How to specify queries?	
	Part 2: Selection	59
6.1	Specifying Query Patterns: Query Terms	59
6.1.1	Textual Term Syntax	61
6.1.2	XML-style Term Syntax	63
6.1.3	Pure XML Syntax	63
6.2	Variables in Query Terms	65
6.2.1	Textual Term Syntax	69
6.2.2	XML-style Term Syntax	70
6.2.3	Pure XML Syntax	70
6.3	Incomplete Patterns	71
6.3.1	Textual Term Syntax	75
6.3.2	XML-style Term Syntax	75
6.3.3	Pure XML Syntax	76
6.4	Top-level Query Terms	76
6.4.1	Term Formulas	76
6.4.2	Document Specifications	77
6.4.3	Textual Term Syntax	78
6.4.4	XML-style Term Syntax	80
6.4.5	Pure XML Syntax	80
6.5	Summary: Modifiers and Where they Occur	81

7	Programming in Xcerpt: Programs, Goals, and Rules	83
7.1	Xcerpt Programs	83
7.1.1	Textual Term Syntax	85
7.1.2	XML-style Term Syntax	85
7.1.3	Pure XML Syntax	86
7.2	Semantic Restrictions on Xcerpt Programs	88
7.2.1	Range Restrictedness	88
7.2.1.1	Polarity of Subterms	88
7.2.2	Negation and Grouping Stratification	90
II	Language Extensions and Open Issues	93
8	Node Identity in Xcerpt	95
8.1	Introduction	95
8.2	Object Identity in Data Management	97
8.3	“Do we really need object identity?”	99
8.3.1	Regular Infinite Trees	99
8.3.2	Object Identity: Updates, Sharing,	100
8.4	Aims of the Proposal	102
8.5	Proposal 1: Infinite Regular Trees	102
8.6	Proposal 2: Hidden Identity	103
8.7	Conclusion and Future Work	104
9	Grouping and Aggregation in Xcerpt	105
9.1	Introduction	105
9.2	Multirelations, Bags, and Sequences	105
9.3	Solution Proposal	107
10	Modules in Xcerpt 2.0—Reuseware Integration	109
10.1	Introduction	109
10.2	Rule Languages	109
10.3	Framework for rule language module systems	110
10.4	Module system operators	111
10.4.1	Module Definition Operators	111
10.4.2	Module Access Operators	112
10.5	Extending Xcerpt for Module Support	114
10.5.1	Abstract Syntax	115
10.5.2	Term Syntax	116
11	RDF Access in Xcerpt 2.0—An Outlook	119
11.1	Introduction	119
11.2	Challenges Related to the Data Model	120
11.2.1	Graph Data Model and References	120
11.2.2	Labeled Edges	121
11.2.3	Incomplete and Unbounded Data	121
11.2.4	RDF Graphs as Xcerpt Data Terms	122
11.2.5	Order of Sub-Terms	122

11.3	An Intuitive Syntax for Versatile Web Query Languages	122
11.4	Common Query Constructs for the Web and the Semantic Web	123
11.4.1	Query Patterns and Answer Closedness	123
11.4.2	Injectivity and Querying RDF Sequences	125
11.4.3	Blank Node Treatment	126
11.4.4	Negation and Breadth-Complete Queries	126
11.4.5	Optional Sub-Terms	127
11.5	From Queries to Transformations	128
11.5.1	Construct-Query-Rules and User Defined Reasoning	128
11.5.2	Grouping Constructs	128
11.5.3	Versatile access to XML and RDF	129
11.6	Conclusion and Outlook	130
12	Open Issues: Language Constructs	131
12.1	General Issues	131
12.1.1	Defaults and Default Modes	131
12.2	Construct Specific Issues	132
12.2.1	Conditional Construction and optional Terms	132
12.2.2	Query Formulas as Subterms	133
12.2.2.1	without s as Direct Siblings	133
12.2.3	Functions and Libraries: Built-In and User-defined	134
12.2.4	Variables	134
12.2.5	Varia	134
12.3	Querying the Type of Data, Typed Accessors	135
12.4	Collapsing Text Nodes	135
13	Open Issues: Specific to Data Representation Format	137
13.1	Serializing to XML and from XML	137
14	Open Issues: Specific to Concrete Syntax	139
14.1	Non-XML Term Syntax	139
14.2	XML-style Term Syntax	139
14.3	Pure XML Syntax	139
III	Full Language Grammars	141
A	Grammar for Non-XML Term Syntax	143
A.1	Literal Structures	143
A.2	Data Terms	144
A.3	Construct Terms	145
A.4	Query Terms	147
A.5	Programs	150
B	Grammar for XML-style Term Syntax	151
B.1	Literal Structures	151
B.2	Data Terms	151
B.3	Construct Terms	152
B.4	Query Terms	154

B.5	Programs	157
C	Relax NG Schema for XML Syntax	159
C.1	Parameterized Grammars: Terms, Declarations, Modifiers, etc.	159
C.1.1	Declarations	159
C.1.2	Conditions	160
C.1.3	Formulas	160
C.1.4	Modifiers	160
C.1.5	Term	161
C.2	Grammar for Xcerpt Programs	163
C.3	Exemplary Data Term	169
D	ANTLR Grammar for Xcerpt 2.0	181

Chapter 1

Introduction

Xcerpt is *semi-structured query language*, but very much unique among the exemplars of that type of query languages (for an overview see [7]):

1. In its use of a **graph data model**, it stands more closely to early semi-structured query languages such as Lorel [2, 45] than to current mainstream XML query languages.
2. In its aim to address all specificities of **XML with great care**, it resembles more current mainstream XML query languages such as XSLT [16] or XQuery [9]. Xcerpt is tailored to XML in numerous ways, e.g., by proper support for attributes and namespaces [10]. This is achieved without sacrificing the conceptual simplicity and syntactical conciseness of the language. Some aspects of XML are treated differently than in mainstream XML query languages, e.g., the transparent resolution of non-hierarchical relations expressed using ID/IDREF, XLink [24], etc.
3. In using (slightly enriched) **patterns** (or templates or examples) of the sought-for data for querying, it resembles more the “query-by-example” paradigm [54] and XML query languages such as XML-QL [25]. In contrast, current mainstream XML query languages use navigational access to XML data.
4. In offering a **consistent extension of XML** to overcome certain restrictions of XML, that seem arbitrary in the context of Web querying and Xcerpt in particular, it is ready to incorporate access to data represented in richer data representation formats. Instances of such features are element content, where the *order is irrelevant* (and can not be queried) and labels that contain “reserved” XML characters.
5. In providing (syntactical) extensions for querying, among others, RDF, Xcerpt becomes a **versatile query language** (as defined in [13]).
6. In a strict separation of querying and construction and in its use of logical variables and deductive rules, it resembles more logic programming languages or Datalog. In contrast, SQL, e.g., mixes construction and querying (nested queries) and uses explicit references to views rather than rule chaining.

These unique characteristics of Xcerpt motivate many of the language concepts introduced in the remainder of this chapter, a more complete discussion of the guiding design principles for (versatile) Web query languages as exemplified in Xcerpt can be found in [13].

Xcerpt exhibits not just one concrete syntax, but rather three, each focused on providing a unique set of strengths.

1. The first, **non-XML syntax** is based around the idea of representing **terms as in logic-programming** and the following principles:
 - (a) Terms are represented **similar to logic-programming**: prefix notation with bracketed argument lists for the children of the term. Special provisions are made to adapt this basic principle to handle the specificities of XML and other Web formats.
 - (b) Different types of brackets encode term properties and distinguish language constructs from data.
 - (c) The syntax strives to be *concise*, but still *easy to read*. The latter objective is supported, e.g., by the preference for explicit full-word keywords (e.g., **optional**) to represent language constructs instead of shorthand notations (such as **@** for attribute in XPath).
 - (d) The non-XML term syntax emphasizes that Xcerpts data model and language features are not specific for *one* representation formalism such as XML and RDF, but rather allow different ones to be handled with the same concepts by mapping them to Xcerpt terms, still providing for all the specificities of the supported representation formalisms.
2. The **hybrid XML-style term syntax** is a rather recent development. It aims at
 - (a) providing a syntax that is (almost) **immediately accessible** to persons accustomed with XML;
 - (b) **very explicit**, i.e., uses in addition to the XML syntax uses only English words to represent language features;

This makes the syntax both easier to read and harder to write, as it is slightly less compact than the term syntax but therefore often uses English words to represent language features instead of special symbols.

The mixing of XML-style syntax for terms and keywords as in the non-XML syntax raises a number of potential clashes. Most notably, *character data still has to be quoted* in contrast to XML to avoid having to escape non-XML parts of the syntax.

3. The previous two syntaxes are mainly intended for human use. Like Relax NG, Xcerpt also exhibits a **pure XML syntax** that, though harder to author and read for humans, is easy to process with XML tools. The guiding principle is a form of *markup reification*, i.e., elements and attributes are explicitly represented by XML elements named `element` and `attribute` (similar to XSLT's `xsl:element` and `xsl:attribute` elements for construction of elements with computed names).

As in the other concrete syntaxes, the Xcerpt namespace `http://xcerpt.org/ns/core/1.0` is reserved to indicate language constructs. Indeed, in the XML syntax *all* language constructs are expressed through elements in the Xcerpt namespace.

Chapter 2

Meta-Syntax Notations for Abstract and Concrete Syntax

This article presents the syntax of Xcerpt from three different perspectives: an abstract syntax focusing on the language concepts, a concrete syntax that represents terms in a compact style familiar from, e.g., logic programming, and a concrete XML syntax that represents a basis for Xcerpt tools and machine processing of Xcerpt programs.

To define each of these syntaxes appropriate meta-languages are chosen: For the abstract syntax of Xcerpt—in other words, its information model, what form of information is needed for which feature of the language—the OMG's Unified Modelling Language (UML) is used. The concrete term syntax is defined using EBNF grammars and/or railroad syntax diagrams. Finally, the concrete XML syntax is specified by means of Relax NG schemata.

The remainder of this chapter serves (1) as a (very brief) introduction in the notions of these meta-languages used in this article (2) to define, where necessary, the precise variant of the meta-languages referred to in the following chapters, and (3) to point to authoritative specifications of the meta-languages.

2.1 Abstract Syntax: UML Diagrams or What is the Information conveyed in an Xcerpt Program?

UML models are shown here in the notation from [47], the OMG UML 2.0 Superstructure specification. However, only a small subset of UML's diagrams and notions is needed for the purpose of this article: the abstract syntax is defined using *(static) class diagrams*. Moreover, neither attributes nor methods are present in the diagrams, thus allowing the diagrams to be considered as merely concepts and relations.

In particular, *generalizations* (solid line, with an closed, but unfilled arrow head at the end connected to the more general concept) are used to express different variants of a concept, e.g., the different kinds of data terms, each representing a more specialized variant of the general concept “data term”. Generalizations can be decorated with constraints (attached to the line in braces). In the following, only the complete and the disjoint constraint are used. The first indicates that no instances of the more general concept exist, that are not also instance of (at least) one of the specialized concepts, the latter that the instance sets of the specialized concepts are disjoint.

ISO EBNF Construct	Operator Type	Meaning
unquoted word		non-terminal symbol
"..."		terminal symbol
'...'		terminal symbol
(...)		grouping override precedence)
[...]		optional symbols
{...}		zero or more repetition of symbols
{...}-		one or more repetition of symbols
=	in	symbol definition
;	post	rule terminator
	in	alternative
,	in	symbol concatenation
-	in	except
*	in	<i>n</i> occurrences of symbols
(*...*)		comment
?...?		special sequences (extensible)

Table 2.1: ISO Standard Extended BNF Notation

Aggregations (solid line with an unfilled rhombus at the end connected to the “whole” concept) are used for “part-whole” relations, e.g., to express that data term-level declarations contain data terms. Roles and multiplicities can be used to further annotate aggregations (and other relations).

One advanced concept from UML is used to highlight the differences and commonalities among the three kinds of Xcerpt terms: “*parameterized collaborations*”. UML uses parameterized collaborations to describe what is often referred to as (software) *patterns* (not to be confused with Xcerpt’s patterns), i.e., collections of concepts and relations among concepts that occur in different contexts. They are “parameterized”, as a number of the concepts in the parameterized collaboration are “exported” as parameters and must be related to concrete concepts when using the pattern. See Section 5.1 for a description of a model for Xcerpt terms based on parameterized collaborations.

For more information on UML (including tutorials) see the OMG UML homepage <http://www.uml.org/>.

2.2 Concrete Syntax: Regular Expression-Style EBNF or Defining the Textual Term Syntax for Xcerpt

A common choice to define the textual syntax of a programming or query language is a variant of the “Backus Naur Form” introduced for the specification of ALGOL [6].

Several extensions, then referred to as Extended BNF or EBNF, to the basic BNF notation have been suggested, mostly adding some form of repetition and optionality to the original language (that only provided constructs for terminals, non-terminals, definition of non-terminals, and alternative).

Indeed, several standardization bodies have recently defined “standard”¹ variants of BNF, most notably the ISO EBNF international standard [36], the IETF [22] internet standard, and the W3C-style EBNF notation defined in Section 6 of the XML specification [11]. Table 2.1 shows the constructs of

¹Refined from the dozens of variants, cf. <http://www.cs.man.ac.uk/~pjj/bnf/ebnf.html>.

W3C EBNF Construct	Operator Type	Meaning
<u>unquoted word</u>		non-terminal symbol
"..."		terminal symbol
'...'		terminal symbol
[...]		character groups as in regular expressions
(...)		grouping (to override precedence)
...?	post	optional symbols
...*	post	zero or more repetition of symbols
...+	post	one or more repetition of symbols
::=	in	symbol definition
	in	alternative
-	in	except
/*...*/		comment

Table 2.2: W3C Extended BNF Notation

the ISO EBNF standard in contrast to the constructs of the W3C-style EBNF given in Table 2.2.

For this article, the W3C-style notation is used, since it is reminiscent of regular expressions as also used in Xcerpt and is likely to be most familiar for readers accustomed to W3C standards.

Like [9], we slightly deviate from the syntax in [11] to address the differences in print vs. online publications:

1. Instead of marking non-terminal symbols with links (commonly displayed using underline and blue color), non-terminal symbols are enclosed in typographic angle brackets ($\langle \cdot \rangle$) and set in italics.
2. To further strengthen the difference between meta-language and language constructs, teletype is reserved for terminal symbols, all non-terminals and meta-language constructs are formatted as usual text. In particular, typographic single quotes are used for quoting non-terminals.
3. Whitespace handling is left out of the grammar: By default, additional whitespace may occur anywhere between non-terminals. The few exceptions (names, IRIs, and strings) are noted in the natural language description of the grammar rules.

Finally, to reference non-terminals defined in other specifications a notation similar to the one proposed in [9] is adopted: ($\langle \textit{http://www.w3.org/TR/REC-xml-names/\#Digit} \rangle$ references the non-terminal $\langle \textit{Digit} \rangle$ from the XML specification identified through its canonical URI.

EBNF rules are often visualized using railroad syntax diagrams similar to [12]: terminals and non-terminals are drawn on a line to specify concatenation. Alternatives are represented by stacked lines fanning out at the decision point and repetition is indicated using loops (optionality comes for free as alternative where one of the stacked lines does not contain terminals or non-terminals).

2.3 Concrete Syntax: Relax NG or A Schema for Xcerpt Programs in XML

Xcerpt's XML syntax is specified in form of an Relax NG grammar. Relax NG [18] is a schema language for XML. It has been chosen for the specification of Xcerpt's syntax as it (a) has, in contrast to XML

Schema [28], a compact, easy to read syntax, (b) supports, in contrast to DTDs namespaces, and (c) has support for parameterized grammar rules. The latter point makes it possible to drastically reduce the size of the language specification by reusing the definition of, e.g., a term over all three term kinds of Xcerpt, parameterizing where necessary. It also allows a close alignment with the abstract syntax.

Relax NG has, like Xcerpt, both a compact textual non-XML syntax [17] and a more verbose pure XML syntax. In the following, the compact non-XML syntax for Relax NG is used.

A Relax NG grammar consists in a single **start** production and a set of normal productions, each defining one non-terminal (called *named pattern*). Element content can be defined using connectives like in regular expressions or DTDs: sequence (,), choice (|), repetition (* and +), and optionality (?). Elements and attributes are treated symmetrically, specified using **element**, resp. **attribute** followed by the name of the element or attribute. Literal content is either **text** or content typed according to the XML Schema Datatypes [8].

Non-terminals (named patterns) may be defined by multiple productions, if each production is marked with the how to *combine* the alternatives: |= instead of the usual = indicates that the alternative productions form a choice. I.e., if $S \mid= P$ and $S \mid= Q$ then $S = P \mid Q$. This allows the introduction of additional choices, e.g., when including an existing grammar and is used extensively to define construct and query terms as extensions of the productions for data terms, cf. Chapter 5, and Chapter 6.

Relax NG is a particular convenient choice for defining Xcerpt, as it allows (to some extent) to express *parameterized concepts* (as discussed above): In Relax NG grammars (i.e., sets of productions or (non-terminal) definitions) may be *merged* or included into each other. When including one grammar in another one (using the **include** keyword and a reference to the file in which the grammar to be included is contained), definitions for non-terminals may be replaced or combined with new ones. Such replaced non-terminals can be seen like parameters of the grammar. Grammars can furthermore be nested to “hide” away all productions except the start symbol of the grammar.

Namespaces can be attached to all elements of a grammar or to individual elements. There only the first means are used, cf. Section 4.1.

More information about Relax NG and its compact syntax can be found at the OASIS Relax NG site <http://www.relaxng.org/>.

Part I

Definition of the Core Language

Chapter 3

Xcerpt: A Versatile Web Query Language

Xcerpt is *semi-structured query language*, but very much unique among the exemplars of that type of query languages (for an overview see [7]):

1. In its use of a **graph data model**, it stands more closely to early semi-structured query languages such as Lorel [2, 45] than to current mainstream XML query languages.
2. In its aim to address all specificities of **XML with great care**, it resembles more current mainstream XML query languages such as XSLT [16] or XQuery [9]. Xcerpt is tailored to XML in numerous ways, e.g., by proper support for attributes and namespaces [10]. This is achieved without sacrificing the conceptual simplicity and syntactical conciseness of the language. Some aspects of XML are treated differently than in mainstream XML query languages, e.g., the transparent resolution of non-hierarchical relations expressed using ID/IDREF, XLink [24], etc.
3. In using (slightly enriched) **patterns** (or templates or examples) of the sought-for data for querying, it resembles more the “query-by-example” paradigm [54] and XML query languages such as XML-QL [25]. In contrast, current mainstream XML query languages use navigational access to XML data.
4. In offering a **consistent extension of XML** to overcome certain restrictions of XML, that seem arbitrary in the context of Web querying and Xcerpt in particular, it is ready to incorporate access to data represented in richer data representation formats. Instances of such features are element content, where the *order is irrelevant* (and can not be queried) and labels that contain “reserved” XML characters.
5. In providing (syntactical) extensions for querying, among others, RDE, Xcerpt becomes a **versatile query language** (as defined in [13]).
6. In a strict separation of querying and construction and in its use of logical variables and deductive rules, it resembles more logic programming languages or Datalog. In contrast, SQL, e.g., mixes construction and querying (nested queries) and uses explicit references to views rather than rule chaining.

These unique characteristics of Xcerpt motivate many of the language concepts introduced in the remainder of this chapter, a more complete discussion of the guiding design principles for (versatile) Web query languages as exemplified in Xcerpt can be found in [13].

3.1 Data Model

As stated above, Xcerpt uses a **graph data model**. More precisely, Xcerpt provides access to *one or more* data graphs (that are usually stored in data units called “documents” identified by IRIs [27]). Each data graph is a *rooted, directed, node-labeled, ordered, unranked* graph with two types of nodes:

§1 Element Nodes

Element (or structural) nodes represent XML elements or similar **structured** data items (e.g., resources in RDF).

Each element node is decorated further with a dictionary (or associative list) of (XML-style) **attributes**. Some attributes are predefined and exist at all nodes, viz. the label and namespace (cf. [10]), others are specified in the data, e.g., as XML attributes. Just like in XML, attributes are single valued¹ and unordered, i.e., for each attribute name (dictionary key) a single value exists and the order of the key-value pairs is not significant and can not be queried.

Currently, element nodes in Xcerpt do not have an explicit object or node identity, i.e., two element nodes with the same attributes and children can not be distinguished from each other. Though, explicit node identity can be simulated with the current approach, direct support of explicit node identity is under consideration, cf. Issue ??.

Element nodes closely resemble **element information items** from [21] with two minor deviations: Xcerpt does, at least at the time of writing, not provide access to in-scope namespaces (cf. Issue 19) and for the base URI according to the XML Base recommendation [43] (cf. Issue 18). The handling of attributes, however, deviates notably from the XML information set to emphasize the distinction of elements and attributes: attributes are simple key-value pair, where the key is an XML name (and thus may consist in prefix and local name) and the value is an arbitrary string. No further information can be attached to attributes.

Each element nodes has zero or more edges to other nodes, called its **children**. These edges are always *ordered*. However, in contrast to pure XML, one can specify whether this order is significant, i.e., whether it has to be preserved during storage or transformation and can be queried. All element nodes originating from XML documents are by default ordered (cf. Section 4.6). Element nodes where the order is significant are called *ordered*, element nodes where the order is insignificant *unordered*. There are no further restrictions on the edges, i.e., the graph may be cyclic, may have loops, the same two nodes may be connected by several nodes, e.g., if a node is the 2nd, 4th, and 12th children of another one.

§2 Content Nodes

Content (or atomic) nodes represent data items that are considered **unstructured** in the context of Xcerpt.

Content nodes can be further distinguished into

1. **Text nodes** that represent the textual content of element nodes. The only attribute of a text node is the string it represents. The same restrictions as for text nodes in XSLT [16], XQuery, and

¹See Issue ?? for a discussion on list-valued attributes (such as attributes of type IDREFS in XML).

XPath [29] apply, i.e., (1) text nodes *never* represent an *empty string*, (2) two text nodes can *never be direct siblings* of each other. Two nodes are direct siblings, if either they are children of the same ordered element node and are consecutive in the children order or they are children of the same unordered element node. Thus, an unordered element node may not have more than one text node child (cf. Issue 15). If two text nodes are constructed as direct siblings they are *collapsed*.

2. **Comment nodes** that represent comments, i.e., annotations on the actual data that are not meant for machine processing. As text nodes, they have only one attribute: the content of the comment. However, in contrast to text nodes no further restrictions are placed on comment nodes.
3. **Processing instruction nodes** that represent processing instructions, i.e., annotations on the actual data that are meant for processing by specific “target” services. They carry two attributes, the content of the processing instruction (usually some form of instructions for the “target” service) and the name of the “target” service.

3.1.1 Terms for Representing Data and Queries

Inspired by logic programming languages, Xcerpt chooses the concept of **terms** for representing, constructing, and querying complex (or structured) data: Xcerpt uses three forms of terms:

1. **data terms** for representing semi-structured data, i.e., all node types from the data model are represented as terms,
2. **construct terms** for specifying “forms” or “templates” of data to be constructed with variables to indicate where data obtained from the (separated) query part is to be “filled” in, and finally
3. **query terms** for specifying “patterns” or “examples” of the data to be matched by a query again with variables to indicate where data is to be extracted from the matches.

(XML) element nodes represented as terms are the **only complex data structure** in Xcerpt. In particular, variables can only be of type term (which includes literal content such as strings as atoms). Other complex data structure such as lists (or sequences), homogeneous or heterogeneous records, sets, and dictionaries (or associative lists) can be simulated as terms, but no specific constructs are offered. Instead Xcerpt avoids to burden the query author with the selection of the appropriate data structures and leaves this to the query processor. The query processor can choose appropriate storage and access methods, if a term is restricted, e.g., by means of a schema (i.e., type information, see Section 12.3). E.g., a term’s children may be stored and accessed using algorithms for dictionaries if it is known that the labels of all children are mutually distinct. Or duplicate elimination may be skipped during query evaluation if the children of a term are indeed restricted to a proper set.

The remainder of this chapter introduces the three concrete syntaxes for Xcerpt discussed in this article: the textual non-XML term syntax, the pure XML syntax, and the hybrid XML-style term syntax.

3.2 A Textual Non-XML Term Syntax for Xcerpt

Xcerpt exhibits not just one concrete syntax, but rather three, each focused on providing a unique set of strengths. The first, non-XML syntax is based around the idea of representing **terms as in logic-programming** and the following principles:

1. Terms are represented **similar to logic-programming**: prefix notation with bracketed argument lists for the children of the term. Special provisions are made to adapt this basic principle to handle the specificities of XML and other Web formats.
2. Different types of brackets encode term properties and distinguish language constructs from data.
3. The syntax strives to be *concise*, but still *easy to read*. The latter objective is supported, e.g., by the preference for explicit full-word keywords (e.g., **optional**) to represent language constructs instead of shorthand notations (such as @ for attribute in XPath).
4. The non-XML term syntax emphasizes that Xcerpt's data model and language features are not specific for *one* representation formalism such as XML and RDF, but rather allow different ones to be handled with the same concepts by mapping them to Xcerpt terms, still providing for all the specificities of the supported representation formalisms.

The actual syntax is introduced in each chapter along the abstract and the other concrete syntaxes. The following preliminary remarks set the basis for the discussion of the non-XML term syntax in the rest of this article.

3.2.1 Lexical Structures

The textual non-XML term syntax makes use of the following five lexical structures:

1. **Names:** For, among others, element labels and variables, Xcerpt uses the namespace-aware identifiers, that must adhere to the definition for $\langle NCName \rangle$ in the W3C XML Namespace recommendation [10]. Notice, that this allows for slightly different identifiers than allowed by the definition of an XML (1.0) $\langle Name \rangle$ in [11]. The difference is that in namespace-aware identifiers as used in Xcerpt the double colon characters is *not* allowed.
2. **IRIs:** For namespaces and as a pool for unique identifiers, Internationalized Resource Identifiers (short IRIs) may be used in Xcerpt. Internationalized Resource Identifiers are defined in RFC 3987 [27]. Like strings, IRIs are always enclosed in straight double quotes in Xcerpt.
3. **Strings:** Literal content is represented as strings of characters. However, to avoid the introduction of character entities into Xcerpt's non-XML syntax, Java strings (as of §3.10.5 of [32]) are chosen and not, e.g., XML character data. Since Xcerpt's syntax is not line-oriented, there is no need to escape linefeed or carriage return. Thus, an Xcerpt string is an arbitrary sequence of Unicode characters with straight double quotes and backslashes backslash-escaped. For convenience, Java escape sequence (e.g., \t for a tabulator) and Unicode escapes (e.g., \u000d for a carriage return) are also allowed. Strings in Xcerpt are *always* enclosed in straight double quotes ("), never in single straight quotes.
4. **Numbers:** Some Xcerpt constructs have parameters that are natural numbers. Here, we use again the definition from [11].
5. **Regular Expressions:** In query terms (cf. Chapter 6), Xcerpt uses POSIX.1 regular expressions as defined in [33], Base Definitions Volume (XBD), ch. 9, but extends these regular expressions with variable bindings.

This results in the following grammar for the lexical structures used in Xcerpt's non-XML term syntax (lexical structures are distinguished from other non-terminals by an uppercase first letter):

```

<NCName>           ::= <http://www.w3.org/TR/REC-xml-names/#NCName>
<IRI>              ::= "' <http://www.ietf.org/rfc/rfc3987.txt#IRI> '"
<String>          ::= "' <StringCharacter>* '"
<StringCharacter> ::= <http://java.sun.com/docs/books/jls#StringCharacter> | <Line-feed> | <Carriage-return>
<Line-feed>       ::= '\{\}u000a'
```

```

<Carriage-Return> ::= '\{u000d'
<Int> ::= <http://www.w3.org/TR/REC-xml-names/#Digit>*
<Regexp> ::= '/' <{http://www.unix.org/version3/ieee_std.html#}extended_reg_exp> '/'
<http://www.unix.org/version3/ieee_std.html#ERE_expression> ::=
    <http://www.unix.org/version3/ieee_std.html#one_char_or_coll_elem_ERE>
    | '^'
    | '$'
    | '(' <http://www.unix.org/version3/ieee_std.html#extended_reg_exp> ')'
    | '(' <variable> '->'
    <http://www.unix.org/version3/ieee_std.html#extended_reg_exp> ')'
    | <http://www.unix.org/version3/ieee_std.html#ERE_expression>
    <{http://www.unix.org/version3/ieee_std.html#}ERE_dupl_symbol>

```

Notice, how the *<ERE_expression>* from the POSIX standard is overwritten by the production specified here that includes (in line 6 of the production) the added syntax for binding Xcerpt variables in regular expressions. For *<variable>* production is defined in Chapter 6.

3.2.2 Reserved Names

Xcerpt's non-XML term syntax does not reserve any names as language keywords, as the structured of the language allows a disambiguation between keywords and names.

However, the Xcerpt namespace identified by the IRI `http://xcerpt.org/ns/core/1.0` is reserved for language constructs and can not be used for other purposes.

Some implementations may additionally want to restrict the occurrence of keywords in identifiers, i.e., they may want to choose the following restricted definition of *<NCName>*:

```

<NCName> ::= <http://www.w3.org/TR/REC-xml-names/#NCName> - <ReservedNames>
<ReservedNames> ::= 'all' | 'and' | 'desc' | 'descendant' | 'except' | 'first' | 'not' | 'optional' |
    'or' | 'position' | 'some' | 'without'

```

3.2.3 Whitespace and Comments

In the grammars for the non-XML term syntax, whitespace is not explicitly included. Rather Xcerpt uses the following whitespace handling rules (similar to, e.g., XQuery [9]):

1. Arbitrary sequence of whitespace characters (as defined by the character class *<S>* in the XML specification [11]) and Xcerpt comments may occur in between any two terminals and must occur where otherwise two sequential terminals are recognized as one. It can be safely "normalized" to a single whitespace character.
2. Strings, names, and other literal structures are considered a single terminal for the purpose of this rule. In other words, in strings, names, and other literals whitespace is significant and may not be ignored. E.g., the string " a " differs from the string "a".

Xcerpt's non-XML term syntax allows both end-of-line and block **comments** to occur in place of whitespace. The following rules define whitespace and comments for Xcerpt's non-XML term syntax.

```

<Whitespace> ::= (<http://www.w3.org/TR/REC-xml/#S> | <End-of-line-comment> |
    <Block-comment>)*
<Comment-char> ::= <http://www.w3.org/TR/REC-xml/#NT-Char>
<End-of-line> ::= <Line-feed> | <Carriage-return> (<Line-feed>)?

```

```

<End-of-line-comment> ::= '#' ((Comment-char)* –
                          ((Comment-char)* <End-of-line> (Comment-char)*)
                          <End-of-line>)

<Block-comment>      ::= '/#' ((Comment-char)* –
                          ((Comment-char)* ('/ #' | '/#') (Comment-char)*)
                          '/#')

```

Notice, that block comments can not be nested, cf. Issue

3.3 Hybrid XML-style Term Syntax

The hybrid XML-style term syntax is a rather recent development. It aims at

1. providing a syntax that is (almost) **immediately accessible** to persons accustomed with XML;
2. **very explicit**, i.e., uses in addition to the XML syntax uses only English words to represent language features;

This makes the syntax both easier to read and harder to write, as it is slightly less compact than the term syntax but therefore often uses English words to represent language features instead of special symbols.

The mixing of XML-style syntax for terms and keywords as in the non-XML syntax raises a number of potential clashes. Most notably, *character data still has to be quoted* in contrast to XML to avoid having to escape non-XML parts of the syntax.

As the remainder of this article illustrates, that the hybrid XML-style term syntax can indeed be defined by very few deviations from the non-XML term syntax that only affect the representation of (structured) terms as XML-style elements instead of logic-programming style.

The same *lexical structures* and *reserved words* as in the non-XML syntax are used. In particular, character data must be quoted as in the non-XML syntax. Neither normal XML character data nor CDATA sections are allowed.

Note, that this is possible as the Xcerpt term syntax follows the same convention as XML for plain names.

The same syntax for end-of-line and block comments as in Xcerpt's term syntax is used. Note, that block comments in the XML syntax, i.e., using `<!--` and `-->` as delimiters, may also occur, but represent comments in the *data*, not in the query language (cf. Section 4.2).

3.4 Pure XML Syntax

The previous two syntaxes are mainly intended for human use. Like Relax NG, Xcerpt also exhibits a pure XML syntax that, though harder to author and read for humans, is easy to process with XML tools. The guiding principle is a form of *markup reification*, i.e., elements and attributes are explicitly represented by XML elements named `element` and `attribute` (similar to XSLT's `xsl:element` and `xsl:attribute` elements for construction of elements with computed names).

The *lexical structures* and *whitespace handling* rules used are, of course, those of XML, see [11]. In particular, character data must follow the XML restrictions, i.e., `<` and `&` must be escaped.

As in the other concrete syntaxes, the Xcerpt namespace `http://xcerpt.org/ns/core/1.0` is reserved to indicate language constructs. Indeed, in the XML syntax *all* language constructs are expressed through elements in the Xcerpt namespace.

Chapter 4

Specifying Semi-structured Data: Xcerpt's Data Terms

4.1 Defining Data Terms

Starting with this section, the remainder of this chapter discusses the three different kinds of terms used in Xcerpt starting with data terms, the most basic term kind.

§1 Data Terms

Data terms are (linear) representations of semi-structured *data* in Xcerpt.

Unsurprisingly, data terms are closely aligned with Xcerpt's data model as introduced in Section 3.1. Each of the node types in Xcerpt's data model are represented by a corresponding kind of data term. However, data terms differ in two notable aspects:

First, data terms are hierarchical (i.e., tree shaped). Thus, to obtain a linear representations of the Xcerpt data graph, **referable term identifiers** and **references** are introduced that allow to express non-hierarchical relations.

§2 Term Identifiers

Referable term identifiers are identifiers for (structured) terms (representing element nodes) that are *unique* at least within the current document and allow *references* to the identified terms.

Term identifiers are only required to be unique within the current document (cf. Issue ??). This allows for easier authoring and validation but excludes out- or cross-document links. Such links must be explicitly traversed using a value join, e.g., in the case of (X)HTML between the fragment identifier in an href attribute and the id attributes in the target document.

§3 References

References are “links” in the linear term syntax for representing non-hierarchical relations. They are *transparently* resolved, i.e., the case of a term containing a reference to another term can not be distinguished from the case where the term contains the other term as a direct child. For all references in a document, there must be *exactly one* term with a term identifier identical to the value of the reference in the *same* document.

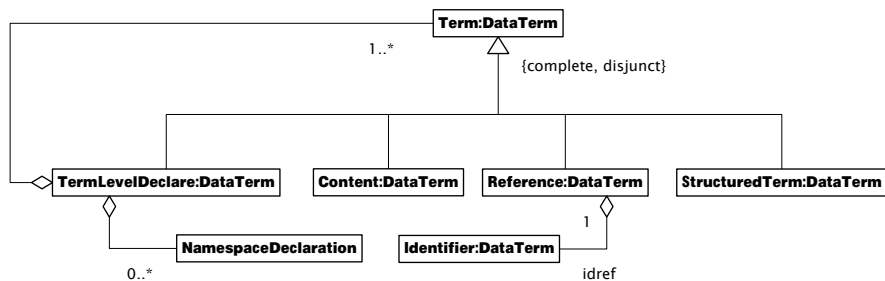


Figure 4.1: UML Model of Data Terms

At the time of writing, term identifiers and references are exclusively part of the linear term representations and *not* preserved in the data model (cf. Issue ?? on introducing explicit node identity in Xcerpt). As a consequence, *term identifiers can not be queried*.

Second, **namespace declarations** (and thus the concept of in-scope namespaces) are not considered as part of term specifications, but are declared in **declaration blocks** surrounding the terms where they may be used. This allows slightly more flexibility in the block structure for declarations and a consistent treatment of declarations on term and rule level (cf. Section 7). Declaration blocks are also used for other declarations (e.g., variable declarations in query and construct terms or type declarations), but at the time of writing namespace declarations are the only kind of declarations allowed in data term declaration blocks. A more detail description of declarations and declaration blocks is given in Section 7.

Figure 4.1 summarizes the kind of terms found as data terms in form of an UML model (cf. Section: A data term can be either

1. an **atomic or content** data term that represents a content node in the data model from Section 3.1,
2. a **structured** data term that represents an element node in the data model,
3. a **reference** to another (structured) data term expressed by a term identifier, or
4. a term-level **namespace declaration** surrounding any number of other data terms.

4.1.1 Textual Term Syntax: Basic Data Terms

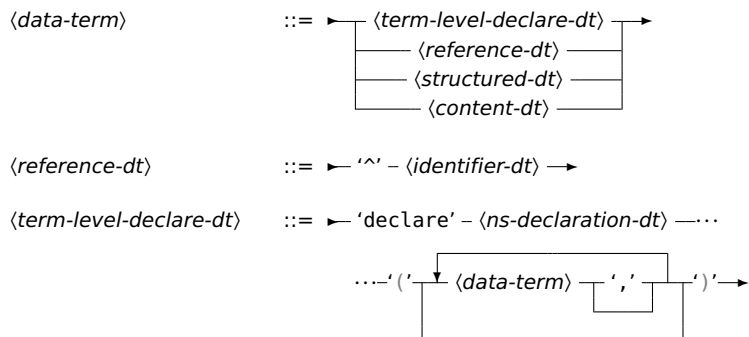
Data terms are defined in the textual term syntax just like in the abstract syntax: either term-level declarations, structured or content data terms, or a reference to another term.

The following productions define first *<data-term>*, a basic data-term, and then references (indicated by a \wedge symbolizing an \uparrow often used to indicate references) and term-level declarations (indicated by the **declare** keyword). Content and structured data terms are discussed in the next sections.

Notice, how the concrete syntax allows both the list of namespace declarations and the data terms in the scope of the declaration to be empty. The abstract syntax (cf. Figure 4.1) however demands that both lists are at least size 1. This is no contradiction: the concrete syntax is designed to be **open**, i.e., to *allow also constructs that are superfluous but not harmful* to ease, e.g., query refactoring and iterative query authoring. In the abstract syntax that presents the information model of an Xcerpt program these superfluous constructs are, however, not any more present. If in the concrete case of term-level declarations either list is empty,

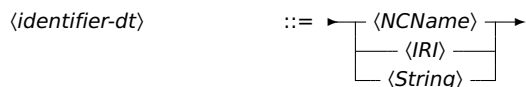
there either have been no namespaces declared or there are no affected data-terms. In both cases, the declaration is ineffectual and will be dropped.

Xcerpt has a considerable number of *parameterized concepts*, i.e., concepts that occur in different contexts with different parameters. E.g., declaration blocks may enclose data terms, top-level data terms, query terms, construct terms, and rules as body, but in each place where a declaration block may occur only one such enclosed construct type is allowed. This form of parameterized concepts can not be directly expressed in notation such as EBNF. Therefore non-terminals that indicate by a suffix the context in which, e.g., a declaration block occurs are used (e.g., $\langle term\text{-}level\text{-}declare\text{-}dt \rangle$ instead of just $\langle term\text{-}level\text{-}declare \rangle$).

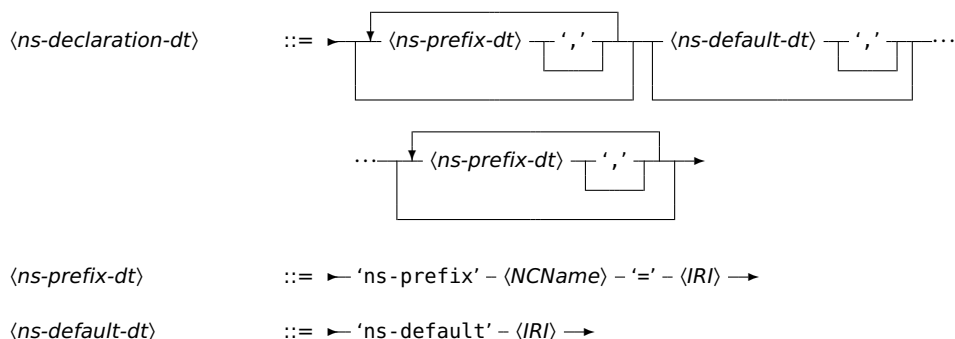


Note, that the parentheses used to enclose the in-scope data terms of the declaration are colored in 'gray'. In the following this is used as a short-hand to indicate that, if a construct that has a list of terms as parameter is applied to (a list of) exactly one, then the brackets can be omitted. I.e., $A \text{'(' } B^* \text{'')}$ where A and B are arbitrary right-hand side expressions in the EBNF syntax used in this article, is equivalent to $(A B) \mid A \text{'(' } B^+ \text{'')}$.

Identifiers (in data terms) are introduced here, but also use in several other parts of a data term, cf. Section 4.3. Identifiers in the non-XML term syntax can be either namespace-aware XML names, IRIs, or strings. As discussed in Section 3.2.1, both IRIs and strings are always enclosed in straight double quotes, only namespace-aware XML names remain unquoted.



Namespace Declarations (in data terms) are the basic form of namespace declarations. In query and construct terms namespace declarations are slightly extended to also allow variables instead of prefixes or namespace URIs.



4.1.2 XML-style Term Syntax: Basic Data Terms

Here, the XML-style term syntax uses the exact same productions as the non-XML term syntax, differences occur only in later parts of data-terms, viz. when defining structured data terms.

4.1.3 Pure XML Syntax: Basic Data Terms

The pure XML syntax is, as mentioned above, defined using Relax NG schemata. To highlight the similarities between data, construct, and query terms and to ensure consistency, all three terms are defined based a common grammar for terms, that is parameterized where needed. In fact, this common grammar exactly captures data terms and is explained in the following.

The first excerpt defines syntax for the basic concepts introduced above: terms, references, and term-level declarations. Syntax for content and structured (data) terms is discussed in the following sections.

```
1 default namespace = "http://xcerpt.org/ns/core/1.0"
3 ## A generic Xcerpt term. Variants are data, construct, and query terms.
   term.class |=
5   reference | content-term | structured-term | term-level-declare
7 ## A declaration block on term level allows possibly (in data and construct terms) only namespace
   declarations.
   term-level-declare =
9   grammar {
     include "declare-block.rnc" {
11     content = parent term.class*
     var-declaration = empty
13   }
   }
15 ## The using occurrence of a reference, i.e. "^ id" in term syntax.
17 reference = element reference { identifier.class }
```

Notice, how term-level declarations are indeed defined by referencing an nested grammar and parameterizing some of its non-terminals, viz. what the content of a declaration is (here a term) and that no variable-declarations are used. Figure 4.2 shows a visualization of the definition of term-level-declare unfolding the nested grammar.

Surprisingly, Relax NG restricts the ability to parameterize grammars to inclusion of grammars in external files, here the file `declare-block.rnc`, whose content is the following Relax NG grammar:

```
1 default namespace = "http://xcerpt.org/ns/core/1.0"
   namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"
3
   start = declare-block
5
   ## A declare block with an empty content and both namespace and variable declarations.
7 declare-block =
   element declare { (ns-declaration | var-declaration)*, content }
9 ns-declaration =
   ns-prefix-declaration*,
11 element ns-default {
   element iri { iri.class }
13 }?,
```

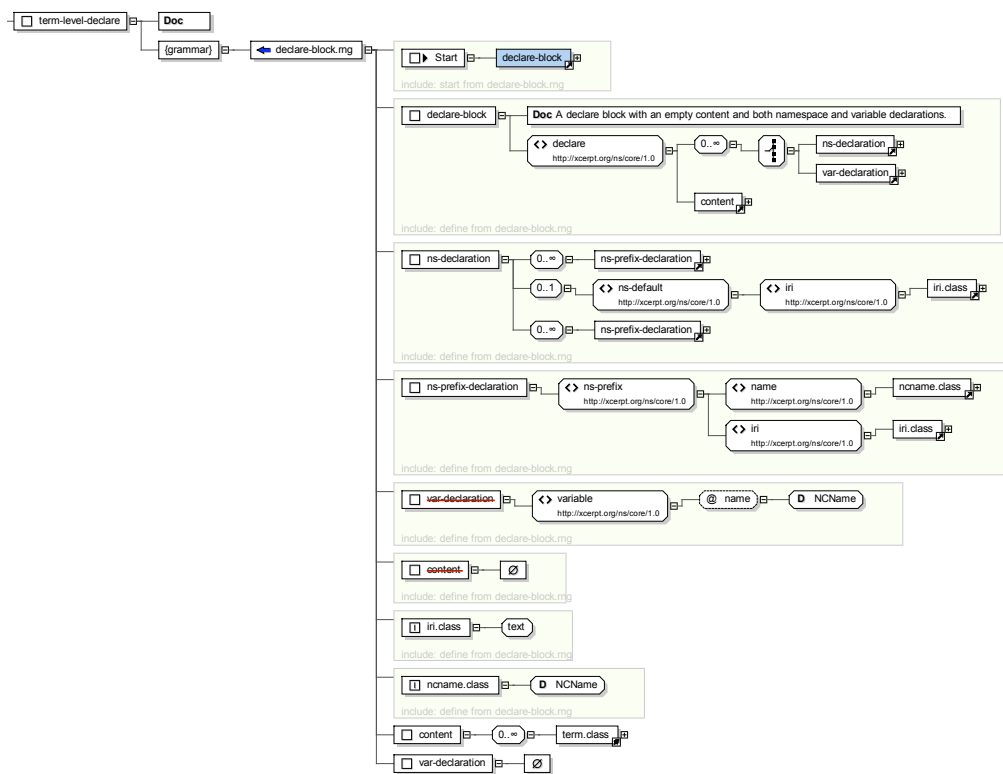


Figure 4.2: Relax NG Schema for Term-level Declarations in Data Terms

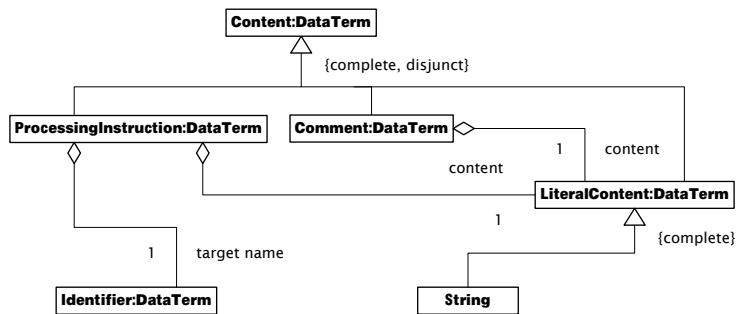


Figure 4.3: UML Model of Atomic Data Terms

```

ns-prefix-declaration*
15 ns-prefix-declaration =
  element ns-prefix {
17   element name { nname.class },
   element iri { iri.class }
19 }
var-declaration =
21 element variable {
   attribute name { xsd:NCName }
23 }
content = empty
25 iri.class |= text
nname.class |= xsd:NCName

```

Notice, the use of combinable definitions (indicated by |=) for uri.class and nname.class. This allows later the easy addition of more choices (viz. variables that can occur instead of the plain IRIs or names).

4.2 Content Data Terms

§4 Content Data Terms

The atomic form of data terms are terms that represent information that is considered *unstructured* in the context of Xcerpt, viz. literal (character) content as well as data annotations in form of comments for human consumption and in form of processing instructions for machine consumption.

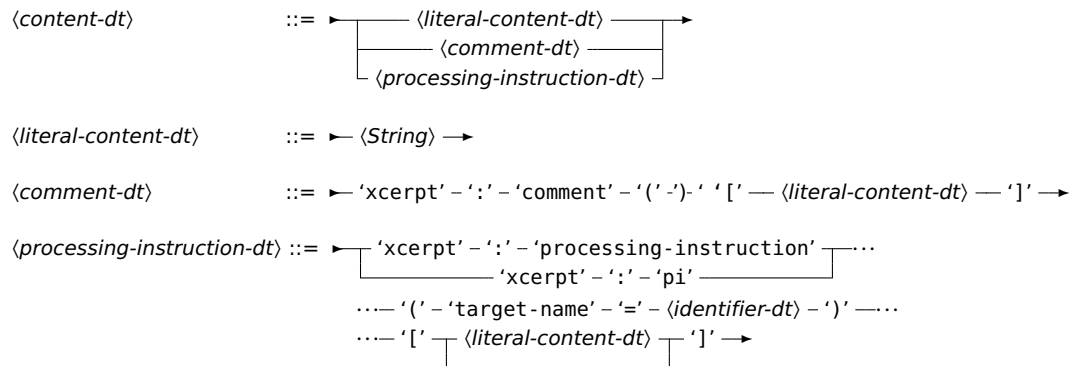
A formal model of content data terms is shown in Figure 4.3: for each of the content nodes in the data model introduced in Section 3.1 a corresponding data term exists. Notice, that in alignment with [11] neither processing instructions, comments, or literal (character) content can be nested.

XML restricts the literal content of comments (processing instructions, resp.) to disallow the character sequence ‘-’ (‘?>’, resp.). This is not the case in Xcerpt. However, when creating XML data these additional restrictions have to be considered, cf. Issue 16).

4.2.1 Textual Term Syntax: Content Data Terms

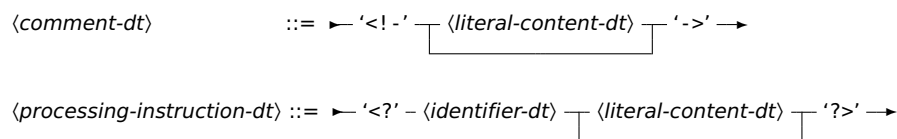
Literal content is represented in the non-XML term syntax by simple (Java-style Unicode) strings as defined in Section 3.2.1. Comments and processing-instructions use the same syntax as structured data terms introduced in the next section: label in prefix position followed by the list of attributes and children. Here, the first is enclosed in (round) parentheses, the second in (square) brackets. Structured terms may also use curly braces around the children list to indicate that the order is insignificant.

Since comments may have no attributes their attribute list is always empty.



4.2.2 XML-style Term Syntax: Content Data Terms

Again the XML-style term syntax uses the same productions as the non-XML term syntax, but differs on the representation of comments and processing instructions: Both are represented as in XML with the exception of their literal content. That is, as literal content of elements, quoted by straight double quotes (just like in the non-XML term syntax). Thus if an element in an XML document contains the character sequence a & b < c \ d or က this may be written as "a & b < c \\ d or \u1000" if it occurs in a Xcerpt data term in XML-style term syntax.



Also note, that XML disallows comments containing character sequence '-' (and no quoting mechanism is applicable, as entities are not expanded in comments) as well as processing instructions containing the character sequence '?>' (again no quoting mechanism applicable). These restrictions are not present in Xcerpt or one of its syntaxes.

4.2.3 Pure XML Syntax: Content Data Terms

Again, we show the case for representing general terms in the pure XML syntax:

```

## A content term represents literal or other non-nestable content.
2 content-term = literal-content.class | annotation-content

4 ## Content kinds that can be used to annotate elements.
annotation-content =
6 element comment { literal-content.class }
  | element processing-instruction {

```

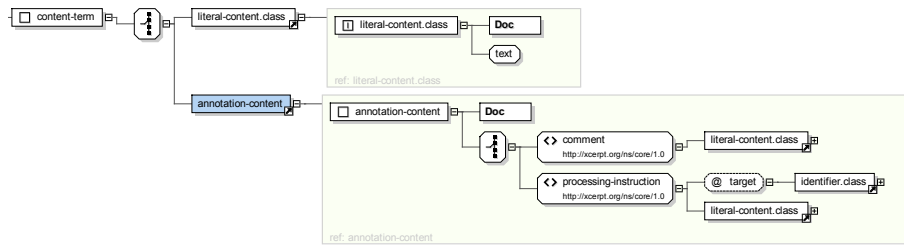


Figure 4.4: Relax NG Schema for Content Terms

```

8     attribute target { identifier.class },
      literal-content.class
10  }

12  ### Character data or other atomic content.
    literal-content.class |= text

```

Notice, how `literal-content.class` is “open” (indicated by `|=`), i.e., further choices for `literal-content.class` may be added separately. Figure 4.4 shows a visual representation of the Relax NG schema fragment with definitions inlined.

4.3 Structured Data Terms

§5 Structured Data Terms

Structured data terms correspond with element or structural nodes in the data model and can be nested.

Structured data terms can be distinguished in *ordered* and *unordered* data terms. As for element nodes, the distinction indicates

§6 Ordered and Unordered Terms

A (structured) term is called *ordered*, if the order of its children is significant. Otherwise it is called *unordered*. In the former case the order must be preserved during processing and storage and is accessible in queries, whereas in the latter it may change during storage or processing and can not be queried.

For consistency with query terms, structured data terms are further classified as *total*.

§7 Total Terms

A term is called *total*, if its list of children is complete, i.e., there can be no additional children.

In Xcerpt, all data terms are total, i.e., must have all their children specified (cf. Issue ??).

Each structured term consists in two parts: a specification of its attributes (called in the following “local” specification as attributes are the non-structural properties of that term) and its children as shown in Figure 4.5:

- The **children** of a structured data term is a sequence of zero or more (arbitrary) data terms. If

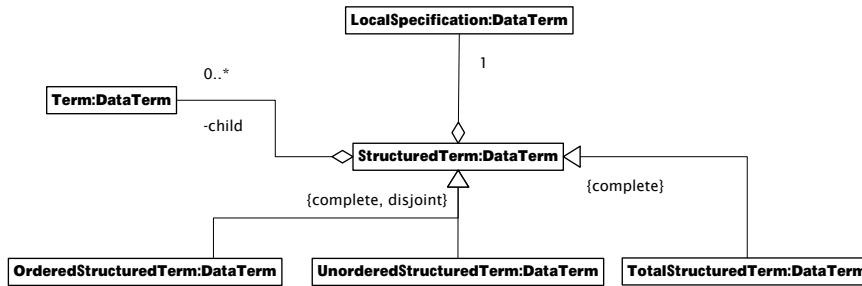


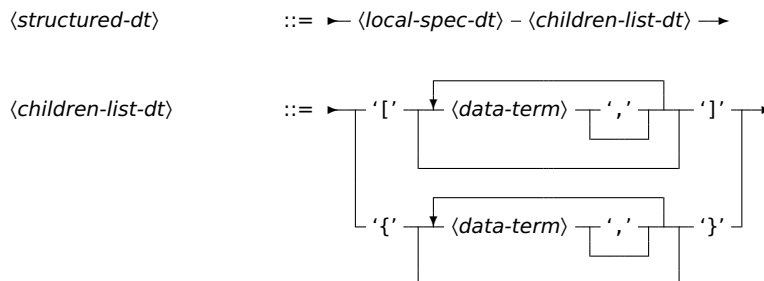
Figure 4.5: UML Model of Structured Data Terms

the children list is empty, the term itself is often referred to as an *empty* term. This children list corresponds to the child edges in the data model after reference resolution.

- The **local specification** of a structured data term is depicted in Figure 4.6. It allows the specification of
 1. An optional *term identifier* that can be used to reference the term, as discussed above. Term identifiers can be either XML Names¹ or IRIs.
 2. Exactly one name or *label* of the data term. The label itself consists in an optional namespace prefix and a mandatory local part. Again both can be either XML Names or IRIs.
 3. An *attribute term list*, that is (a) in data terms always total and (b) consists of one or more attribute terms, which in data terms are simple pairs of attribute names (or key) and values. The name has the same shape as data term labels, the value as literal content data terms.

4.3.1 Textual Term Syntax: Structured Data Terms

Structured data terms form the core of the non-XML term syntax: the “local”, non-structural properties are specified in prefix notation followed by the list of children. The list of children is enclosed in brackets, either square brackets (‘[]’) to indicate that the order in which the children are specified is significant or curly braces (‘{ }’) to indicate that the order is insignificant:



¹Or, more precisely, NCNames as defined in the W3C XML Namespace recommendation [10]. Notice, that these are not quite identical to the original XML (1.0) Names defined in [11], that do *not* treat the double colon as a special character.

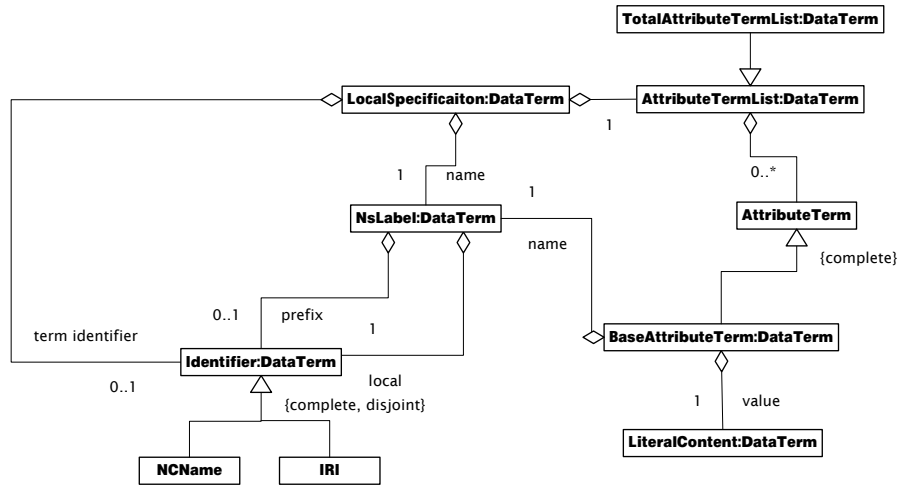


Figure 4.6: UML Model of the Local Specification for Structured Data Terms

Local properties of a structured term are the optional term identifier (that is preceding the remaining properties and separated from them by '@'), the label of the term, and its attributes. The label itself falls into two parts separated by ':', viz. the optional namespace and the local name. Term identifier, namespace, and local name are all identifiers as defined above in Section 4.1.

$\langle local-spec-dt \rangle ::= \langle term-identifier-dt \rangle \langle ns-label-dt \rangle \langle attr-term-list-dt \rangle$

$\langle term-identifier-dt \rangle ::= \langle identifier-dt \rangle \text{'@'}$

$\langle ns-label-dt \rangle ::= \langle identifier-dt \rangle \text{' ':'} \langle identifier-dt \rangle$

Attribute terms are specified in a attribute list term, enclosed in round parentheses. Each attribute is a pair, separated by '=', of the attribute label (itself, as an element label, a pair of namespace and local name separated by ':') and the attribute content.

$\langle attr-term-list-dt \rangle ::= \text{'('} \langle attr-term-dt \rangle \text{'}'$

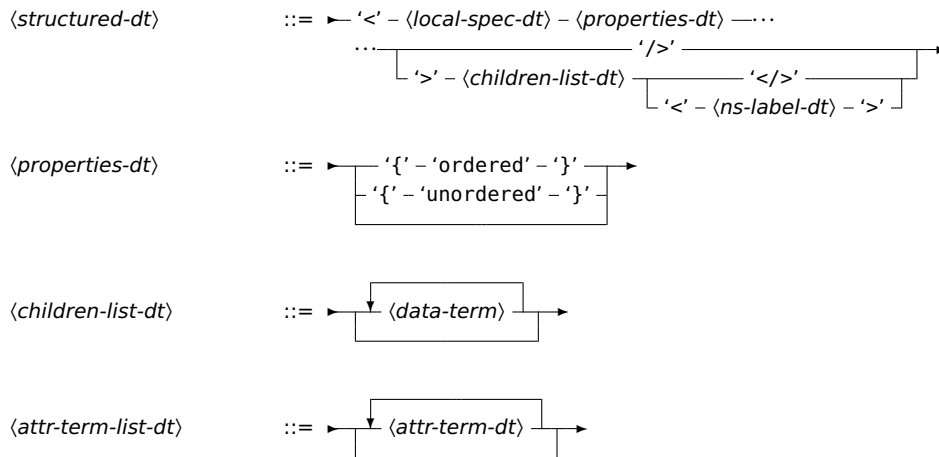
$\langle attr-term-dt \rangle ::= \langle base-attr-term-dt \rangle$

$\langle base-attr-term-dt \rangle ::= \langle ns-label-dt \rangle \text{'='} \langle literal-content-dt \rangle$

Notice, that both the list of children nor the attribute may be empty, but neither may be absent. I.e., neither a[] nor a() or a are structured terms, but, e.g., a() [] and a() {}, cf. Issue 1.

4.3.2 XML-style Term Syntax: Structured Data Terms

In the representation of structured terms lies the main difference between the non-XML and the XML-style term syntax: instead of prefix notation for terms as in logic-programming, XML-style element and attribute notation is used. To achieve this, the following four productions are changed or added:



There is an additional restriction on the first production: the (namespace, local name) pair used as label in the end element tag and the (namespace, local name) pair used in the start element tag (i.e., produced as part of *local-spec-dt*) must be (modulo whitespace) component wise equivalent character sequences.

Observe, how the first production encloses the entire local spec (including, e.g., the term identifier) in the start element tag. Just like in XML this makes all the attributes part of the start element tag. Instead of using different brackets, the significance of the order is indicated here using special term properties '{ordered}' and '{unordered}'. If neither is given the term is assumed to be ordered as in standard XML.

4.3.3 Pure XML Syntax: Structured Data Terms

Structured terms (and as such structured data terms) are represented in the pure XML syntax by an XML element element with sub-elements for its identifier (optional), its label, its list of children, and its list of attributes.

E.g., the following data term in non-XML term syntax

```
1 a(b = "c & a \ b"){ o1 @ d()[ e() [ ] ], d()[ ] }
```

is represented in pure XML syntax as:

```
1 <element> <!-- a(b = "c & a \ b"){ o1 @ d()[ e() [ ] ], d()[ ] } -->
  <label>a</label>
3 <attributes>
  <attribute total="true">
5     <label>b</label>
     <value>c & a \ b</value>
7 </attribute>
  </attributes>
9 <children ordered="false" total="true">
11 <element> <!-- o1 @ d()[ e() [ ] ] -->
   <identifier>o1</identifier>
13 <label>d</label>
   <attributes total="true" />
15 <children ordered="true" total="true">
   <element>
17 <label>e</label>
   <attributes total="true" />
```

```

19     <children ordered="true" total="true" />
      </element>
21    </children>
    </element>
23
    <element> <!-- d()[] -->
      <label>e</label>
      <attributes total="true" />
27     <children ordered="true" total="true" />
    </element>
29 </children>
</element>

```

Obviously, this is vastly more verbose than either the non-XML or the XML-style term syntax. However, it has the virtue that (with the exceptions of regular and qualified descendant expressions, cf. Chapter 6) all constructs of Xcerpt are explicitly represented as either XML elements or attributes. No non-XML “sub-languages” remain that require special consideration, such as XPath in XSLT. This makes the syntax very easy to process with XML tools.

The following gives the full grammar for structured terms in Relax NG compact syntax:

```

## A structured term is a term that may have children and
2 ## attributes. It contrasts with literal content.
structured-term =
4  element element { term-local-spec, term-children, term-condition? }

6 ## Some terms may have additional constraints attached to them.
term-condition = empty
8
## The children of a term can be ordered or unordered, total or partial.
10 term-children =
  element children {
12   attribute ordered { "true" | "false" },
  attribute total { total.class },
14   term.class*
  }
16
## The specification of the 'local' properties of a term: identifier, label, namespace, and attributes.
18 term-local-spec = term-identifier?, ns-label, attr-term-list

20 ## The defining occurrence of a reference, i.e. "id @" in term syntax.
term-identifier = element identifier { identifier.class }
22
## Label and namespace of an Xcerpt term or attribute.
24 ns-label =
  element label {
26   element ns { identifier.class }?,
  identifier.class
28  }

30 ## A term specifying the attributes of an element.
attr-term-list =
32  element attributes {

```

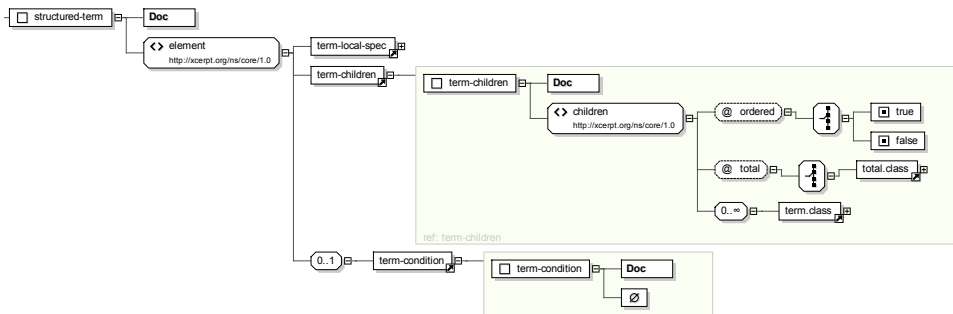


Figure 4.7: Relax NG Schema for Structured Terms

```

attribute total { total.class },
34 attribute-term.class*
  }
36
  ## Class of values for attributes specifying totality or
38 ## partiality of a term's children or attribute list.
  total.class |= "true"
40
  ## A attribute term is an attribute possibly modified with respect to location, modality, and selection.
42 attribute-term.class |= base-attribute

44 ## An attribute consists of a label and an attribute content.
  base-attribute =
46 element attribute {
    ns-label,
48 element value { literal-content.class }
  }

```

In the grammar (and, more easily recognizable, in Figure 4.7 where the productions are inlined) term conditions are provided but as an empty production. This is a sort of “hook” where in query terms actual term conditions can be plugged in. Similarly, the attribute total (cf. also Figure 4.8 showing the local part of a term specification) might strike as peculiar, since it is fixed to the value ‘true’. However, the possible values are defined in the non-terminal total.class for which here only a single production is given, but others might be added, e.g., when defining query terms leading to a choice of attributes.

4.4 Top-level Data Terms

To conclude the discussion of data terms, it should be noted, that data terms on top-level are slightly restricted in comparison to data terms at any other level as discussed so far: Only structured data terms and declare blocks are allowed, the later again being restricted to contain only a *single* top-level data term instead of one or more arbitrary data terms. Figure 4.9 shows the full model of top-level data terms.

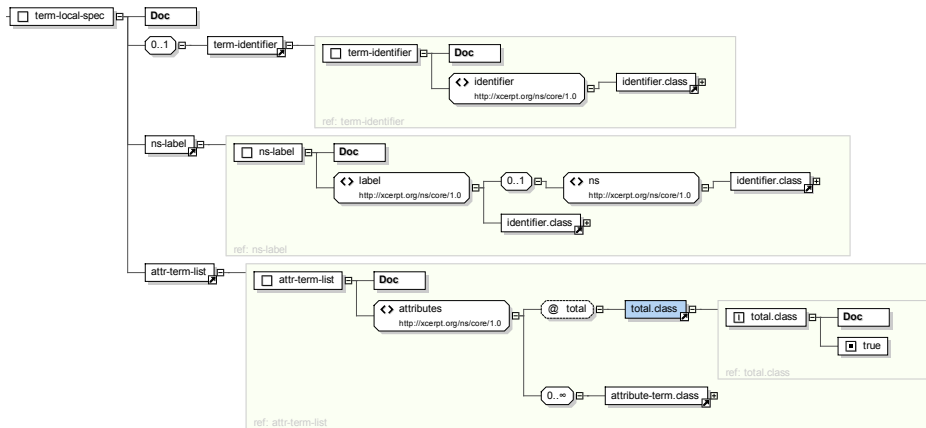


Figure 4.8: Relax NG Schema for Local Specifications of Structured Terms

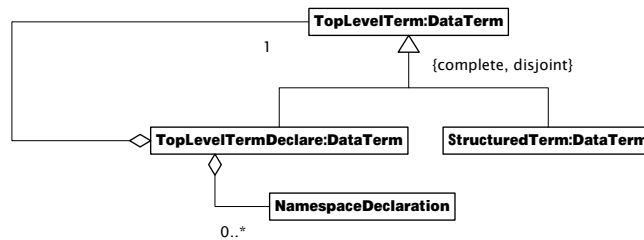


Figure 4.9: UML Model of Top Level Data Terms

4.4.1 Textual Term Syntax: Top-Level Data Terms

Due to the different declaration blocks (i.e., basic data terms may be declaration blocks that contain all forms of basic terms, top-level data terms may be declaration blocks that contain only top-level terms), a top-level data term can not be specified as a basic data term with some exceptions. Rather a separate production is needed:

$$\langle \text{top-level-data-term} \rangle ::= \blacktriangleright \langle \text{top-term-level-declare-dt} \rangle - \langle \text{structured-dt} \rangle \longrightarrow$$
$$\langle \text{top-term-level-declare-dt} \rangle ::= \blacktriangleright \text{'declare'} - \langle \text{ns-declaration-dt} \rangle - \text{'('} - \langle \text{top-level-data-term} \rangle - \text{'\text{'}} \longrightarrow$$

4.4.2 XML-style Term Syntax: Top-Level Data Terms

Again, the productions for the XML-style term syntax do not differ from the non-XML term syntax.

4.4.3 Pure XML Syntax: Top-Level Data Terms

The treatment of top-level terms concludes the definition of terms (which are, as stated above, exactly data terms). Terms are defined in a separate grammar and will be included in the full Xcerpt syntax three times, once for each type of term. For data terms, no parameterization occurs at inclusion:

```
1 # Data Terms
  data-term =
3  grammar {
    include "term.rnc"
5 }
```

term.rnc contains the grammar for terms including all productions discussed so far and the following fragment for top-level terms:

```
1 default namespace = "http://xcerpt.org/ns/core/1.0"
3 start = top-level-term.class
5 ## A term that may occur at top-level. Slightly more
  ## restricted than a basic term.
7 top-level-term.class =
  structured-term
9 | grammar {
    include "declare-block.rnc" {
11     content = parent top-level-term.class*
      var-declaration = empty
13   }
 }
```

Notice, how the start symbol of the grammar is set to top-level-term.class. The above inclusion of the grammar thus makes data-term an alias for top-level-term.class from term.rnc.

4.5 Exemplary Data Term

The following data term will be used as running example for the remainder of this article (e.g., as basis for query and construct term examples). It is drawn from the domain of bibliography management:

Mixing typical bibliographic records (similar to Bibtex or DBLP) with actual content (represented as XHTML or in a Docbook-style format) it combines

- so-called document-oriented with data-oriented XML, i.e., data with flexible, recursive structure and data with rather rigid and flat structure. Recursive structure is used, e.g., for the content of articles in Docbook-style format.
- normalized with de-normalized representation of data (e.g., author information is duplicated for each authored paper, whereas the information about the journal is represented once and referenced in other parts),
- hierarchical with delimiter-based structuring of data (e.g., (X)HTML style sections delimited by consecutive *h*n elements vs. nested sections as, e.g., in DocBook),
- resolved and unresolved links. Where links are used to normalize the data (e.g., in case of journal information of an article), these links are resolved to Xcerpt references. Other links (e.g., the link to another section in the content of an article) are left unresolved as they must be distinguished from “normal” nesting. E.g., links to other sections in the content of an article (like “cf.Section 10”) part-of relations).

Figures 4.10 and 4.11 show fictional journal, proceeding, and article information in non-XML and XML-style term syntax. Notice, the close relation of these syntactical variants: each line of the non-XML term syntax has exactly one corresponding line in the XML-style term syntax.

Unsurprisingly, in pure XML syntax the sample data is considerably longer than in either of the term syntaxes (715 lines vs. 130 lines, i.e., a 5.5-fold increase): Essentially, the entire syntax tree is represented explicitly as XML elements. The full sample data is shown in Appendix C.3, an excerpt (the full journal entry) in Figure 4.12.

4.6 XML Documents as Data Terms

Before taking a look at how queries—that is selection of existing data and construction of new data—are expressed in Xcerpt, the discussion of data terms in Xcerpt is concluded by a look at how XML documents are transformed into data terms when queried with Xcerpt.

An XML document contains (leaving aside for the moment prolog and epilog) a single document element that can be accessed in Xcerpt as a single data term. Section 6.4.2 introduces document specifications that also allow access to prolog and epilog of an XML document. Here, we take a look at how the document element of an XML document is interpreted as an Xcerpt data term. For the most part that is very straightforward, i.e., elements are mapped to structured terms, character data, comments, and processing instructions to their respective form of content terms. However, three issues demand a closer look:

- **Transparent Reference Resolution:** One of the strengths of Xcerpt is the transparent resolution of references. However, when reading XML documents one must consider
 1. How is the *identifier* of an element (represented by a structured data term) specified? Following, [11] and [44], Xcerpt aims at support for the two standard mechanisms for defining element ID’s: attributes of type ID (declared in a DTD or similar schema) and `xml:id` attributes. Currently, the first one is not available, as Xcerpt does not yet provide access to type information from a schema, cf. Issue 12.3.


```

bib(){
2  journal.adm @ journal(){
   title()[ "Applied Data Management" ]
4  editors(){
   editor-in-chief()[ "Titus Pomponius Atticus" ]
6  editor(region="Africa")[ "Marcus Aemilius Aemilianus" ]
   editor(region="Gaul")[ "Aulus Hirtius"
8     affiliation()[ "Governor, Transalpine Gaul" ] ]
   editor(region="Cilicia")[ "Marcus Tullius Cicero"
10    affiliation()[ "Governor, Cilicia" ] ]
  }
12 publisher()[ "Titus Pomponius Atticus" ]
  volumes(){
14   journal.adm.v10 @ volume(){
     journal.adm.v10.n1 @ number(type="special-issue"){
16      title()[ "Data Processing Challenges in the Age of Wax Tablets" ]
     editorial()[ ^ articles.66.cicero.wax ]
18      year()[ "60" ]
     month()[ "july" ]
20     }
     journal.adm.v10.n2 @ number(){
22      year()[ "60" ]
     month()[ "november" ]
24     }
  }
26 }
}

29 conf.dmmc @ proceedings(){
  editors(){
31   editor()[ "Marcus Aemilius Lepidus"
     affiliation [ "Consul, SPQR" ] ]
33   editor()[ "Gaius Julius Caesar Octavianus" ]
   editor()[ "Marcus Antonius" ]
35  }
  title()[
37   "Advancements in Data Management for Military and Civil Application"
  ]
39  invited-papers(){
    ^inproc.44.brutus
41   ^article.66.scaurus.qumran
  }
43  abbrev()[ "DMMC" ]
  year()[ "44" ]
45  month()[ "july" ]
  location()[ "Mutina" ]
47  publisher()[ "SPQR" ]
}

article.66.scaurus.qumran @ article(){
51  author()[ "Marcus Aemilius Scaurus"
     affiliation()[ "Tribun, Gnaeus Pompeius Magnus" ] ]
53  title()[ "From Wax Tablets to Papyri: The Qumran Case Study" ]
  in(scrolls="102-112")[ ^ journal.adm.v10.n1 ]
55  citations [
    cite(ref="article.66.cicero.wax") [ ]
57    cite(type="formatted")[ "M. Aemilius Scaurus (104): A Case for Permanent
      Storage of Senate Proceedings. In: M. Aemilius Scaurus, ed. (104): "
59    i()[ "Princeps Senatus: Honor and Responsibility" ]
      ", Chapter 2, 14-88." ]
61  ]
}

```

```

1 <bib {unordered}>
  <journal.adm @ journal {unordered}>
3   <title>"Applied Data Management"</title>
  <editors>
5     <editor-in-chief>"Titus Pomponius Atticus"</editor-in-chief>
     <editor region="Africa">"Marcus Aemilius Aemilianus"</editor>
7     <editor region="Gaul">"Aulus Hirtius"
     <affiliation>"Governor, Transalpine Gaul"</affiliation></editor>
9     <editor region="Cilicia">"Marcus Tullius Cicero"
     <affiliation>"Governor, Cilicia"</affiliation></editor>
11    </editors>
  <publisher>"Titus Pomponius Atticus"</publisher>
13  <volumes>
    <journal.adm.v10 @ volume>
15     <journal.adm.v10.n1 @ number type="special-issue" {unordered}>
     <title>"Data Processing Challenges in the Age of Wax Tablets"</title>
17     <editorial>^ articles.66.cicero.wax</editorial>
     <year>"60"</year>
19     <month>"july"</month>
     </number>
21     <journal.adm.v10.n2 @ number {unordered}>
     <year>"60"</year>
23     <month>"november"</month>
     </number>
25     </volume>
  </volumes>
27 </journal>

<conf.dmmc @ proceedings {unordered}>
30 <editors>
  <editor>"Marcus Aemilius Lepidus"
32   <affiliation>"Consul, SPQR"</affiliation></editor>
  <editor>"Gaius Julius Caesar Octavianus"</editor>
34  <editor>"Marcus Antonius"</editor>
</editors>
36 <title>
  "Advancements in Data Management for Military and Civil Application"
38 </title>
<invited-papers>
40   ^inproc.44.brutus
  ^article.66.scaurus.qumran
42 </invited-papers>
<abbrev>"DMMC"</abbrev>
44 <year>"44"</year>
  <month>"july"</month>
46 <location>"Mutina"</location>
  <publisher>"SPQR"</publisher>
48 </proceedings>

<article.66.scaurus.qumran @ article {unordered}>
51 <author>"Marcus Aemilius Scaurus"
  <affiliation>"Tribun, Gnaeus Pompeius Magnus"</affiliation></author>
53 <title>"From Wax Tablets to Papyri: The Qumran Case Study"</title>
  <in scrolls="102-112">^ journal.adm.v10.n1</in>
55 <citations>
  <cite ref="article.66.cicero.wax" />
57  <cite type="formatted">"M. Aemilius Scaurus (104): A Case for Permanent
    Storage of Senate Proceedings. In: M. Aemilius Scaurus, ed. (104): "
59  <i>"Princeps Senatus: Honor and Responsibility"
    ", Chapter 2, 14-88."</cite>
61 </citations>
</article>

```

Figure 4.10: Exemplary Data Term, Part I: Non-XML and XML-style Term Syntax

```

65 article.66.cicero.wax @ article(){
66   authors(){
67     author()[ "Marcus Tullius Cicero"
68     affiliation()[ "Governor, Cicilia" ] ]
69     author()[ "Marcus Aemilius Lepidus"
70     affiliation()[ "Gens Aemilia" ] ]
71     author()[ "Marcus Tullius Tiro"
72     affiliation()[ "Secretary, M. T. Cicero" ] ]
73   }
74   title()[ "Space- and Time-Optimal Data Storage on Wax Tablets" ]
75   in(scrolls="1-94")[ ^ journal.adm ]
76   content(type="xhtml"){
77     declare ns-default "http://www.w3.org/1999/xhtml"
78     body(){
79       xcerpt:comment[ "incomplete due to melted letters on some tablets" ]
80       h1(id="contributions")[ "Contributions" ]
81       h1()[ "A History of Data Storage: From Stone to Parchment" ]
82       p()[ "Despite " cite()[ ^ article.66.scaurus.qumran ] ... ]
83       ol(){
84         li()[ em()[ strong()[ "Homeric" ] "Age:" ] ... ]
85         li()[ em()[ "Age of the " strong()[ "Kings" ] ":" ] ... ]
86       ]
87       h1(id="tiro")[ "Notae Tironianae" ]
88       img(title="Tironian et" src=...)[ ]
89       p()[ "As discussed in " a(href="#contributions")[ ... ] ]
90       h1(id="tachygraphy")[ "Challenges for Tachygraphy on Wax" ]
91       p()[ "Though conditions for writing on wax tablets are adverse
92       to tachygraphy, systems as described in " a(href="#tiro")[ ... ] ]
93     }
94   }
95 }

65 <article.66.cicero.wax @ article {unordered}>
66   <authors>
67     <author>"Marcus Tullius Cicero"
68     <affiliation>"Governor, Cicilia"</affiliation></author>
69     <author>"Marcus Aemilius Lepidus"
70     <affiliation>"Gens Aemilia"</affiliation></author>
71     <author>"Marcus Tullius Tiro"
72     <affiliation>"Secretary, M. T. Cicero"</affiliation></author>
73   </authors>
74   <title>"Space- and Time-Optimal Data Storage on Wax Tablets"</title>
75   <in scrolls="1-94">^ journal.adm.v10.n1</in>
76   <content type="xhtml">
77     <declare ns-default "http://www.w3.org/1999/xhtml">
78     <body>
79       <!-- "incomplete due to melted letters on some tablets" -->
80       <h1 id="contributions">"Contributions"</h2>
81       <h1>"A History of Data Storage: From Stone to Parchment"</h1>
82       <p>"Despite "<cite>^ article.66.scaurus.qumran</cite> ...</p>
83       <ol>
84         <li><em><strong>"Homeric"</strong>" Age:"</em>...</li>
85         <li><em>"Age of the "<strong>"Kings"</strong>:"</em>...</li>
86       </ol>
87       <h1 id="tiro">"Notae Tironianae"</h1>
88       <img title="Tironian et" src=... />
89       <p>"As discussed in "<a href="#contributions">...</a></p>
90       <h1 id="tachygraphy">"Challenges for Tachygraphy on Wax"</h1>
91       <p>"Though conditions for writing on wax tablets are adverse
92       to tachygraphy, systems as described in "<a href="#tiro">...</a></p>
93     </body>
94   </content>
95 </article>

96 <inproc.44.brutus @ inproceedings {unordered}>
97   <authors>
98     <author>"Marcus Antonius"
99     <affiliation>"Consul, SPQR"</affiliation></author>
100    <author>"Decimus Junius Brutus"
101    <affiliation>"Governor, Cisalpine Gaul"</affiliation></author>
102  </authors>
103  <title>"Efficient Management of Rapidly Changing Personal Records"</title>
104  <in scrolls="24-48">^ conf.dmmc</in>
105  <content type="docbook">
106    <declare ns-default "http://example.org/ns/docbook/simplified/1.0">
107    <section><info><title>"Introduction"</title></info>
108    <section><info><title>"Contributions"</title></info>
109    <para>"The most notable contributions of this article include:"
110    <list type="ordered">
111      <item>
112        <para>"A new " em()[ "methodology" ] " to ..., cf. "
113        <pageref idref="inproc.44.brutus.s1">[ ... ] ]
114        <figure><title>"Chart of Desertions"
115        <img ... /></figure>
116        <para>"As " cite()[ ^ article.66.cicero.wax ] ... ]
117      </item>
118    </list>
119    </para>
120  </section>
121 </section>
122 <inproc.44.brutus.s1 @ section>
123 <info><title>"Acknowledgements" </title></info>
124 <para>"We would like to thank the editors of "
125 <cite>^ journal.adm.v10.n1 </cite> ... ] ]
126 </section>
127 </content>
128 </inproceedings>
129 </bib>

```

Figure 4.11: Exemplary Data Term, Part II: Non-XML and XML-style Term Syntax

```

6 <element>
8 <identifier>journal.adm</identifier>
10 <label>journal</label>
12 <attributes total="true" />
14 <children ordered="false" total="true">
16 <element>
18 <label>title</label>
20 <attributes total="true" />
22 <children ordered="true" total="true">
24 >Applied Data Management</children>
26 </element>
28 </element>
30 <label>editors</label>
32 <attributes total="true" />
34 <children ordered="true" total="true">
36 <element>
38 <label>editor-in-chief</label>
40 <attributes total="true" />
42 <children ordered="true" total="true">
44 >Titus Pomponius Atticus</children>
46 </element>
48 </element>
50 <label>editor</label>
52 <attributes total="true" />
54 <children ordered="true" total="true">
56 <label>region</label><value>Africa</value>
58 </children>
60 </element>
62 </element>
64 <label>editor</label>
66 <attributes total="true" />
68 <children ordered="true" total="true">
70 <label>regions</label><value>Gaul</value>
72 </children>
74 </element>
76 </element>
78 <label>editor</label>
80 <attributes total="true" />
82 <children ordered="true" total="true">
84 <label>affiliation</label>
86 <attributes total="true" />
88 <children ordered="true" total="true">
90 >Governor, Transalpine Gaul</children>
92 </element>
94 </element>
96 </children>
98 </element>
100 </element>
102 <label>editor</label>
104 <attributes total="true" />
106 <children ordered="true" total="true">
108 <label>region</label><value>Cilicia</value>
110 </children>
112 </element>
114 </element>
116 </children>
118 </element>
120 </element>
122 <label>publisher</label>
124 <attributes total="true" />
126 <children ordered="true" total="true">
128 >Titus Pomponius Atticus</children>
130 </element>
132 </element>
134 </children>
136 </element>
138 </element>
140 </element>
142 </element>

```

Figure 4.12: Exemplary Data Term, Excerpt: pure XML syntax

All `xml:id` attributes in XML documents are translated to identifiers for the appropriate structured term, if the document does not contain `xml:id` errors (cf.[44], Section 2), in which case `xml:id` attributes are handled as normal attributes.

2. How is a *reference* to a (defined) identifier specified? Again, Xcerpt aims to support the standard mechanism, i.e., attributes of type IDREF or IDREFS. However, there are other (internal) links, e.g., HTML-style href attributes. Furthermore, not all such links are to be transparently resolved, as discussed above.

In all cases, the query author can specify a view that resolves the references and then formulate the rest of the query on top of this view. However, such a view requires a recursive descent through the document structure and is not trivial to program. Thus, Xcerpt introduces a processing instruction as a convenience that address the most common cases where transparent references are needed:

The syntax of the processing instruction follows [15]: the target name is `xcerpt-resolve-reference` and the content of the processing instruction is a list of “pseudo-attributes” (again following syntax and notion from [15]). The following pseudo-attributes are supported:

attribute specifies the name of the referencing attribute, i.e., the attribute that contains the actual reference(s).

on specifies the (local) name of the element whose attributes contain the reference(s) to be resolved.

ns specifies the namespace of the element. May be omitted in which case only references on elements in the empty namespace are resolved.

type specifies the type of reference. Currently, the values IDREF, IDREFS, and fragment are supported, indicating that the referencing attribute contains a single ID reference, multiple ID references, or a single HTML-style fragment indicator (e.g., `#t i r o`) respectively.

replace specifies whether the element carrying the referencing attribute is merely a placeholder for the referenced element and thus is to be replaced by the reference. Possible values are true and false, with false as default value.

E.g., the processing instruction

```
<?xcerpt-resolve-reference "attribute='idref' on='cite' type='IDREF'  
2 ns='http://example.org/ns/docbook/simplified/1.0'?">
```

specifies that all values in `idref` attributes on `cite` elements in the specified namespace are to be considered as IDREF links and transparently resolved when loading the document.

- **Unordered Content:** Though the children of structured terms are always ordered, Xcerpt allows the specification whether this order is significant and must be preserved. In XML documents this distinction can be made by annotating elements with the `ordered` attribute from the Xcerpt namespace (`http://xcerpt.org/ns/core/1.0`). The possible values are true or false, as in the pure XML syntax, indicating significant and insignificant order.
- **In-scope Namespaces:** XML documents provide no means to separate the scope of namespaces from the scope of individual elements. E.g., the content element in our sample data may contain elements from the XHTML namespace or from the namespace for our simplified Docbook version. However, the namespace declaration must be attached to individual elements, thus

requiring either a wrapper element (the body element in line 77 in Figure 4.11 and line 89 in Figure 4.13) or separate namespace declarations on all sub-elements of content, cf. line 119 and 135 in Figure 4.13.

Figure 4.13 shows an XML document with the appropriate processing instructions and IDs to result in the sample data term, when loaded in Xcerpt.

```

<?xml version="1.0" standalone="yes"?>
2 <?xcerpt-resolve-reference
  "attribute='idref' on='cite' ns='http://example.org/ns/docbook/simplified/1.0'
4   type='IDREF'?>
<?xcerpt-resolve-reference "attribute='idref' on='in' type='IDREF'?>
6 <?xcerpt-resolve-reference "attribute='idref' on='editorial' type='IDREF'?>
<?xcerpt-resolve-reference "attribute='idref' on='ref' replace='true'
   type='IDREF'?>
8 <bib xmlns:xc="http://xcerpt.org/ns/core/1.0" xc:ordered="false">
  <journal xml:id="journal.adm" xc:ordered="false">
10    <title>Applied Data Management</title>
    <editors>
12      <editor-in-chief>Titus Pomponius Atticus</editor-in-chief>
      <editor region="Africa">Marcus Aemilius Aemilianus</editor>
14      <editor region="Gaul">Aulus Hirtius
        <affiliation>Governor, Transalpine Gaul</affiliation>
16      </editor>
      <editor region="Cilicia">Marcus Tullius Cicero
18      <affiliation>Governor, Cilicia</affiliation></editor>
    </editors>
    <publisher>Titus Pomponius Atticus</publisher>
    <volumes>
22      <volume xml:id="journal.adm.v10">
        <number xml:id="journal.adm.v10.n1" type="special-issue"
24          xc:ordered="false">
          <title>Data Processing Challenges in the Age of Wax
26          Tablets</title>
          <editorial idref="articles.66.cicero.wax"></editorial>
          <year>60</year>
          <month>july</month>
30        </number>
        <number xml:id="journal.adm.v10.n2" xc:ordered="false">
32          <year>60</year>
          <month>november</month>
34        </number>
        </volume>
    </volumes>
  </journal>
38 <proceedings xml:id="conf.dmmc" xc:ordered="false">
  <editors>
40    <editor>Marcus Aemilius Lepidus
      <affiliation>Consul, SPQR</affiliation></editor>
42    <editor>Gaius Julius Caesar Octavianus</editor>
    <editor>Marcus Antonius</editor>
44  </editors>
  <title>Advancements in Data Management for Military and Civil
46  Application</title>
  <invited-papers>
48    <ref idref="inproc.44.brutus" />
    <ref idref="article.66.scaurus.qumran" />
50  </invited-papers>
  <abbrev>DMMC</abbrev>
52  <year>44</year>
  <month>july</month>
54  <location>Mutina</location>
  <publisher>SPQR</publisher>
56  </proceedings>
  <article xml:id="article.66.scaurus.qumran" xc:ordered="false">
58    <author>Marcus Aemilius Scaurus
      <affiliation>Tribun, Gnaeus Pompeius Magnus</affiliation>
60    </author>
    <title>From Wax Tablets to Papyrus: The Qumran Case Study</title>
    <in scrolls="102-112" idref="journal.adm.v10.n1" />
    <citations>
62    <cite ref="article.66.cicero.wax" />
    <cite type="formatted">M. Aemilius Scaurus (104): A Case for
64    Permanent Storage of Senate Proceedings. In: M. Aemilius
    Scaurus, ed. (104): <i>Princeps Senatus: Honor
66
68    and Responsibility</i>, Chapter 2, 14-88.</cite>
    </citations>
70  </article>
  <article xml:id="article.66.cicero.wax" xc:ordered="true">
72    <authors>
    <author>Marcus Tullius Cicero
74      <affiliation>Governor, Cilicia</affiliation></author>
    <author>Marcus Aemilius Lepidus
76      <affiliation>Gens Aemilia</affiliation></author>
    <author>Marcus Tullius Tiro
78      <affiliation>Secretary, M. T. Cicero</affiliation>
    </authors>
    <title>Space- and Time-Optimal Data Storage on Wax Tablets</title>
    <in scrolls="1-94" idref="journal.adm" />
    <content type="xhtml">
    <body xmlns="http://www.w3.org/1999/xhtml">
84      <!-- incomplete due to melted letters on some tablets -->
      <h1 id="contributions">Contributions</h2>
86      <h1>A History of Data Storage: From Stone to Parchment</h1>
      <p>Despite recent evidence ...</p>
88      <ol>
        <li><em><strong>Homeric</strong> Age:</em>...</li>
90      <li><em>Age of the <strong>Kings</strong></em>...</li>
      </ol>
92      <h1 id="tiro">Notae Tironianae</h1>
      <img title="Tironian et" src=... />
94      <p>As discussed in <a href="#contributions">...</a></p>
      <h1 id="tachygraphy">Challenges for Tachygraphy on Wax</h1>
96      <p>Though conditions for writing on wax tablets are adverse to
      tachygraphy, systems as described in <a href="#tiro">...</a></p>
98    </body>
    </content>
100  </article>
  <inproceedings xml:id="inproc.44.brutus" xc:ordered="false">
102    <authors>
    <author>Marcus Antonius<affiliation>Consul, SPQR</affiliation>
104    </author>
    <author>Decimus Junius Brutus<affiliation
106    >Governor, Cisalpine Gaul</affiliation></author>
    </authors>
108    <title>Efficient Management of Rapidly Changing Personal Records</title>
    <in scrolls="24-48" idref="conf.dmmc" />
110    <content type="docbook">
    <section xmlns="http://example.org/ns/docbook/simplified/1.0">
112      <info><title>Introduction</title></info>
      <section><info><title>Contributions</title></info>
114      <para>The most notable contributions of this article include:
      <list type="ordered">
116        <item>
          <para>A new <em>methodology</em> to ..., cf.
118          <pageref idref="inproc.44.brutus.s1" /> ...</para>
          <figure><title>Chart of Desertions</title>
120          <img ... /></figure>
          <para>As <cite idref="article.66.cicero.wax" />...</para>
122        </item>
      </list>
    </section>
124    </section>
    <section xml:id="inproc.44.brutus.s1"
126      xmlns="http://example.org/ns/docbook/simplified/1.0">
      <info><title>Acknowledgements</title></info>
      <para>We would like to thank the editors of
128      <cite idref="journal.adm.v10.n1" /> ...</para>
    </section>
130  </content>
132  </inproceedings>
134 </bib>

```

Figure 4.13: Exemplary Data Term: From an XML Document

Chapter 5

How to specify queries?

Part 1: Construction

As briefly mentioned above, Xcerpt uses very much similar concepts and syntax for data and queries. Queries in Xcerpt are guided by a small number of principles:

- **Queries as Patterns.** Instead of using separate concepts and syntax for queries (as in navigational query languages such as XQuery [9]), Xcerpt uses terms for representing both data and queries. All data terms are also query terms, but there are some additional constructs in data terms, that allow (a) the extraction of data by using logical variables, (b) the specification of queries that are only incomplete patterns of the data, i.e., where more nodes may occur in the data than specified in the query, and (c) the specification of formulas in terms, i.e., conjunction, disjunction, negation, optionality etc.
- **Logical Variables.** In query terms, logical variables are used to indicate which data is to be selected and to join data (indicated by multiple occurrences of the same variable as in logic programming languages). The result of a query is conceptually a set of tuples each representing a combination of bindings (or matches) for all the variables occurring in the query term. For each tuple, a data term must exist that matches the query where all the variables are substituted by the bindings of the tuple.
- **Separation of Querying and Construction.** In contrast to query languages such as SQL or XQuery, construction and querying are strictly separated in Xcerpt, in particular there are no nested queries in Xcerpt (rather rules and rule chaining is used, cf. Section 7). The data constructed by a rule is specified in construct terms, that contain variables from the corresponding query terms acting as placeholders for selected data. Additionally construct terms make use of grouping constructs to return all or some of the alternative bindings of a variable.
- **Incomplete Patterns.** In most cases, queries specify just enough restrictions on the data to be returned, as required by the query intent, rather than specifying full or “total” patterns of the data. Xcerpt supports such queries by providing constructs to express that a pattern is incomplete in breadth (i.e., there can be more children than specified), depth (i.e., there can be additional nodes and edges between the matched nodes) etc.

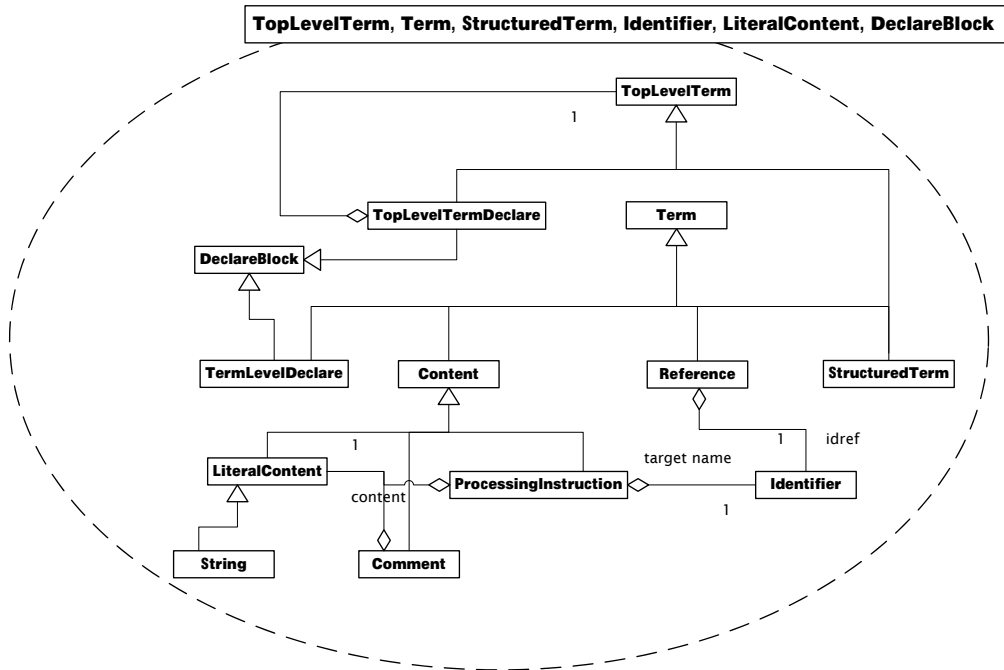


Figure 5.1: UML Model for Terms as Parameterized Collaboration

- **Terms as Formulas.** Query terms are not only augmented by variables, but also by constructs for expressing negation, disjunction, conjunction, and optionality.

In the remainder of this part, first construct terms and then query terms are introduced in detail and compared to data terms. To facilitate a better understanding and description of the differences between data terms, construct terms, and query terms, a short aside introduces a parameterized model for terms, that precisely specifies in what aspects the three kinds of terms may differ from each other.

5.1 An Aside: A Parameterized Model for Terms

UML uses the notion of “**parameterized collaborations**” to describe what is otherwise known as (software) *patterns* (not to be confused with Xcerpt’s patterns), i.e., collections of concepts and relations among concepts that occur in different contexts. They are “parameterized”, as a number of the concepts in the parameterized collaboration are “exported” as parameters and must be related to concrete concepts when using the pattern.

Figure 5.1 shows an example for the notation adopted in UML for defining such parameterized collaborations: concepts and relations are drawn as usual, but a *dashed ellipsis* is drawn around the concepts that are part of the definition. The parameter concepts are depicted in a *box* at the top of the ellipsis.

Indeed, Figure 5.1 shows an Xcerpt term as a parameterized collaboration: all the relations and concepts depicted are common to all three kinds of Xcerpt terms, they only vary in the six parameters

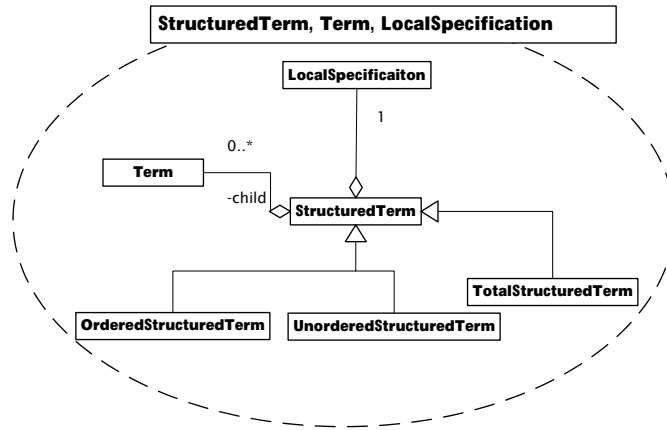


Figure 5.2: UML Model for Structured Terms as Parameterized Collaboration

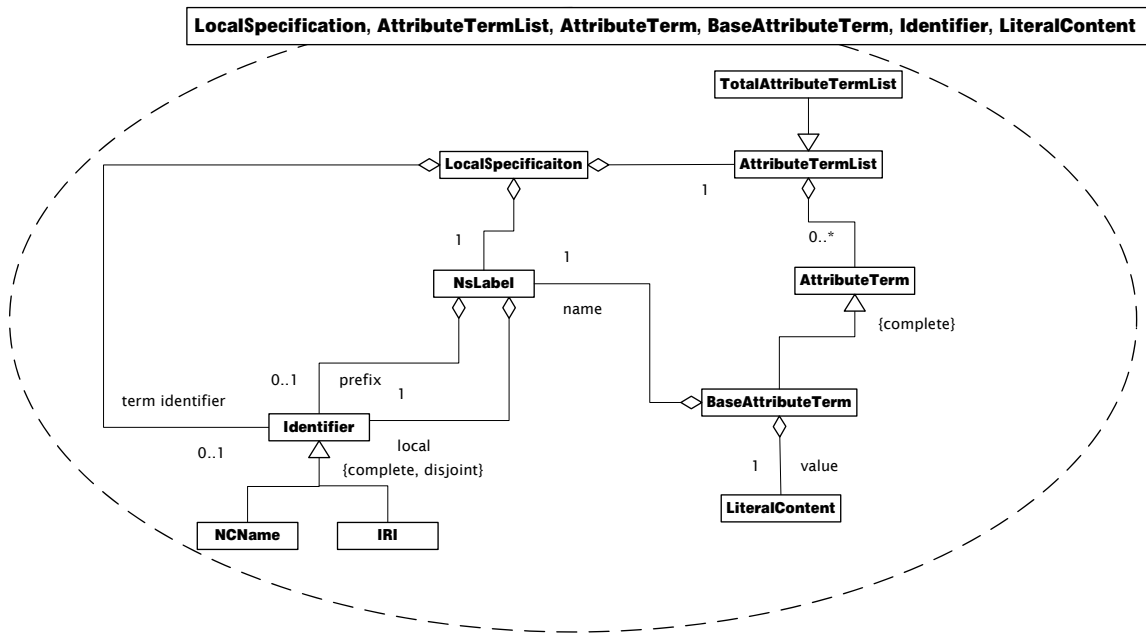


Figure 5.3: UML Model for Local Term Specifications

given in the top corner: (1) what is a top-level term, (2) what is a basic term, (3) what is a structured term, (4) what is an identifier, (5) what is a literal content, and (5) what is a declaration block.

Figures 5.2 and 5.3 complete the definition of a term by defining parameterized collaboration for structured terms and for local descriptions of terms (these could be part of a single pattern as they are never used separately, but for readability they have been split over three diagrams). The parameterized collaboration for structured terms shares the second and third parameter of the parameterized collaboration for terms and adds an additional parameter, the local specification, to link to the parameterized collaboration for local term specifications shown in Figure 5.3. The parameterized collaboration for a local term specification has additional parameters for (1) attribute terms, (2) attributes, and (3) basic (or literal) attributes.

Notice, how similar these patterns are to data terms. This is due to the fact, that all data terms are also valid terms in the other two term kinds.

Given these parameterized collaborations, data terms can be defined as shown in Figure 5.4: all parameters for the three parameterized collaborations are simply “instantiated” with concrete concepts for data terms without adding *any* additional concepts or relations.

5.2 Specifying New Data: Construct Terms

As mentioned above, conceptually the result of a query is a *multi-set of mappings* each representing one combination (or *substitution*) of bindings for all variables occurring in the query term. For each tuple, an (extensional or intensional) data term must exist that matches the query where all the variables are substituted by the bindings of the tuple.

5.2.1 Substitutions and Substitution Sets

A *substitution* is a mapping from the set of (all) variables to the set of (all) construct terms. As usual, a substitution is a mapping of infinite sets. Of course, finite representations are usually used, as the number of variables occurring in a term is finite. Substitutions are often conveniently denoted as sets of variable assignments instead of as functions. For example, we write $\{X \mapsto a, Y \mapsto b\}$ to denote a substitution that maps the variable X to a and the variable Y to b , and any other variable to arbitrary values. In general, a substitution provides assignments for all variables, but “irrelevant” variables are not given in the description of substitutions.

A *substitution multi-set* is simply a multi-set containing substitutions. Often the substitutions in a substitution multi-set have very similar sets of “relevant” variables, differing only, e.g., in optional variables. Thus a substitution multi-set can also be denoted as an n -ary multi-set relation over the set of all construct terms where n is the size of the maximum set of variables “relevant” for any substitution in the multi-set. Substitutions become tuples in this relation with “irrelevant” variables marked as **null** values. E.g., the following table is a representation for a substitution multi-set with three substitutions using “relevant” variables X , Y , and Z . The first tuple represents the substitution $\{X \mapsto a, Y \mapsto b, Z \mapsto c\}$, the second $\{X \mapsto c, Y \mapsto b, Z \mapsto b\}$, and the third $\{X \mapsto c, Y \mapsto a\}$.

X	Y	Z
a	b	c
c	b	b
c	a	null

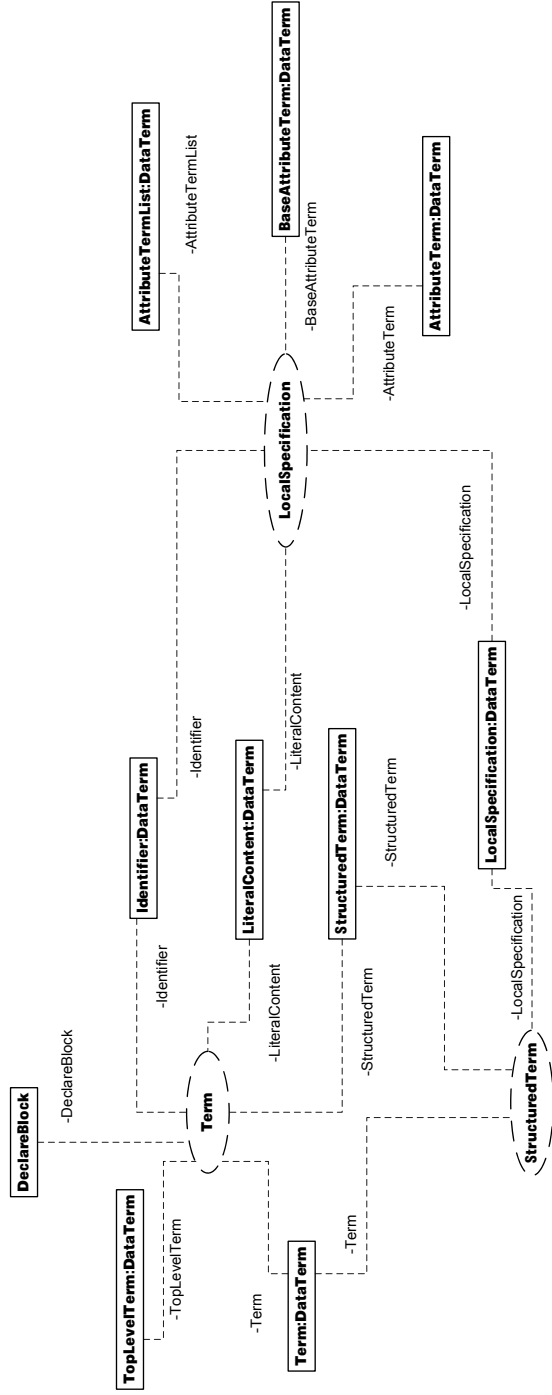


Figure 5.4: UML Model for Data Terms using Parameterized Collaborations, cf. Figure 5.1–5.3

More details on substitutions can be found in [49, 50]. Notice, that there substitution sets are used to simplify definitions and proofs (cf. Issue 9).

5.3 The Shape of Construct Term

§1 Construct term

Construct terms specify *the shape of data that is constructed (or derived)* for each match of the corresponding query term. In that, they are comparable to clause heads in Datalog.

If a construct term contains no variables, it becomes equivalent to a data term: The full shape of the derived data is specified. But construct terms may also contain *variables*, viz. in place of sub-terms (term variables) and in place of literal content or identifiers such as element labels and namespaces, attribute labels, etc. (literal variables).

However, there is one major difference between Xcerpt (and similar query languages for structured data) when comparing to Datalog on flat tuples: E.g., given bindings for authors and titles one would like to create author elements that contain for each author all corresponding titles. Explicit support for *grouping constructs* in the scope of other data is needed to express that form of construction.

Figure 5.5 shows an UML model for construct terms using the parameterized collaborations for general terms introduced in the aside of Section 5.1. The figure highlights the exact differences between data and construct terms:

1. **Variables** can occur instead of (a) (structured or attribute) terms or instead of (b) identifiers and literal content.
2. **Modifiers** specify (a) the grouping of sub-terms by one or more variables, i.e., the repetition of parts of a construct term for all or some of the alternative bindings of one or more variables, and (b) the optionality of sub-terms, i.e., the omission of a part of a construct term based on the bindings of one or more variables.

Notice, that the functionality of these modifiers is almost a corollary of adding variables: Once variables that may have more than one binding are allowed in construct terms, it is necessary to handle the case of bindings for one variable included in construct terms for bindings of another one (grouping). In the same way, once variables may have no bindings at all, it is necessary to define which part of a construct term is to be left out if there is no bindings (optionality).

Figures 5.6 and 5.7 detail modifiers for structured and attribute terms: All modifiers “modify” construct terms to indicate that the modified term is to be handled differently from its unmodified form. The construct terms modified by a modifier are the *scope of the modifier*. In construct terms, all modifiers have a scope of *one or more* construct terms except the grouping modifier for attribute terms. The latter one has a scope of a single attribute construct terms and deviates from the general rule, as attributes are unordered and single-valued (i.e., there may be no repeated attribute names) and thus grouping over sequences of terms is not useful.

5.3.1 Textual Term Syntax

Although the syntactic differences between data and construct terms are from a conceptual perspective few, the EBNF specification of the non-XML (as well as of the XML-style) term syntax share only the productions for lexical structures. This is due to the inability of the EBNF notation to express parameterized productions or grammars. E.g., declaration blocks are identical except that in data terms they contain data terms and

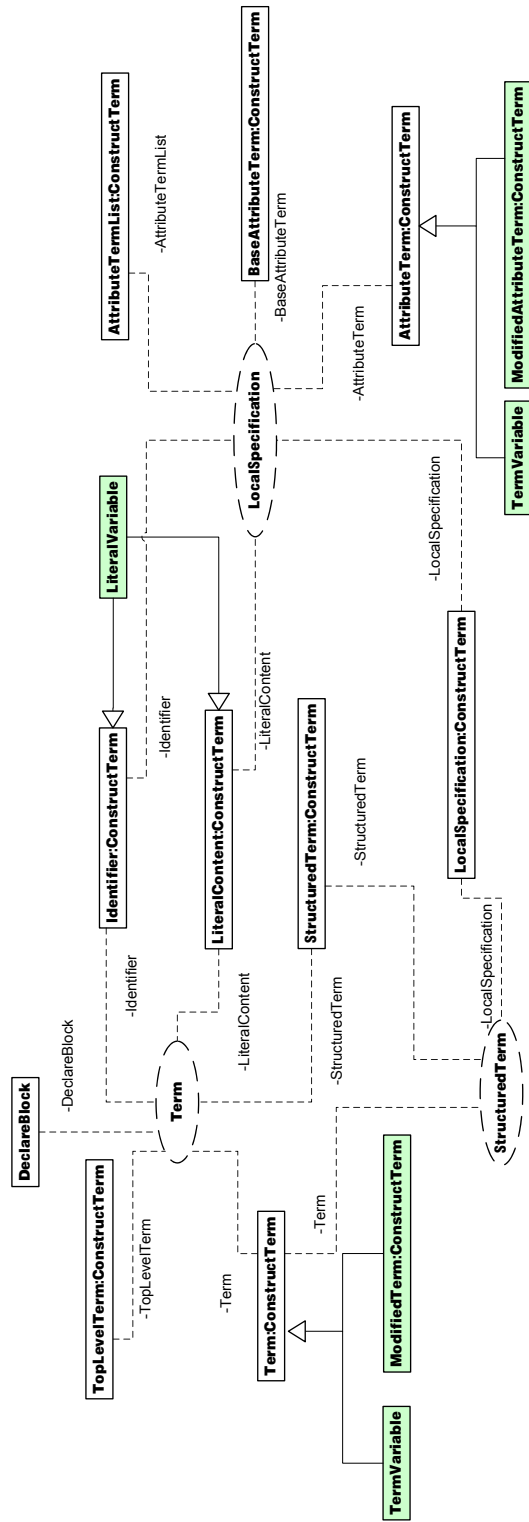


Figure 5.5: UML Model for Construct Terms using Parameterized Collaborations

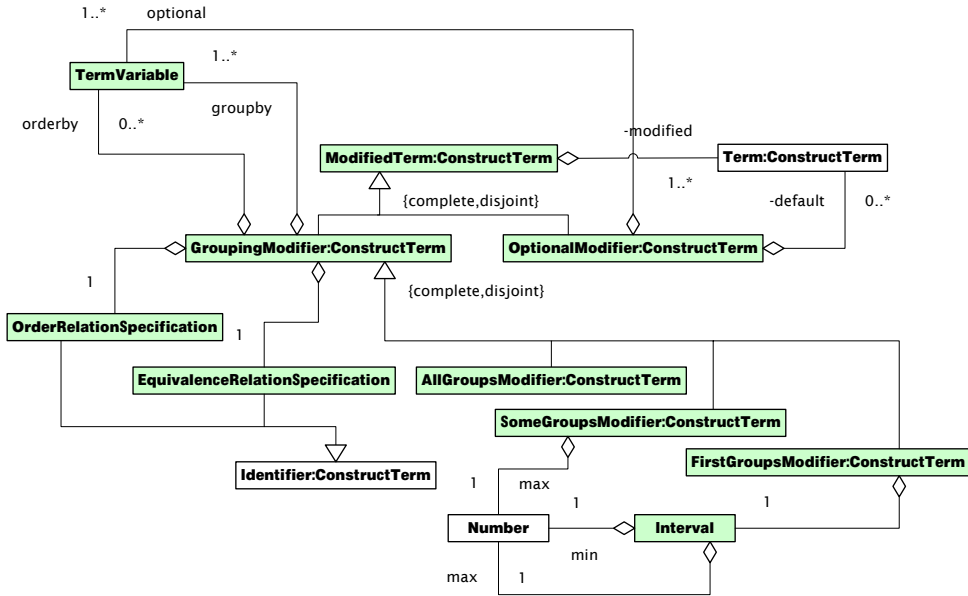


Figure 5.6: UML Model for Modified Structured Construct Terms

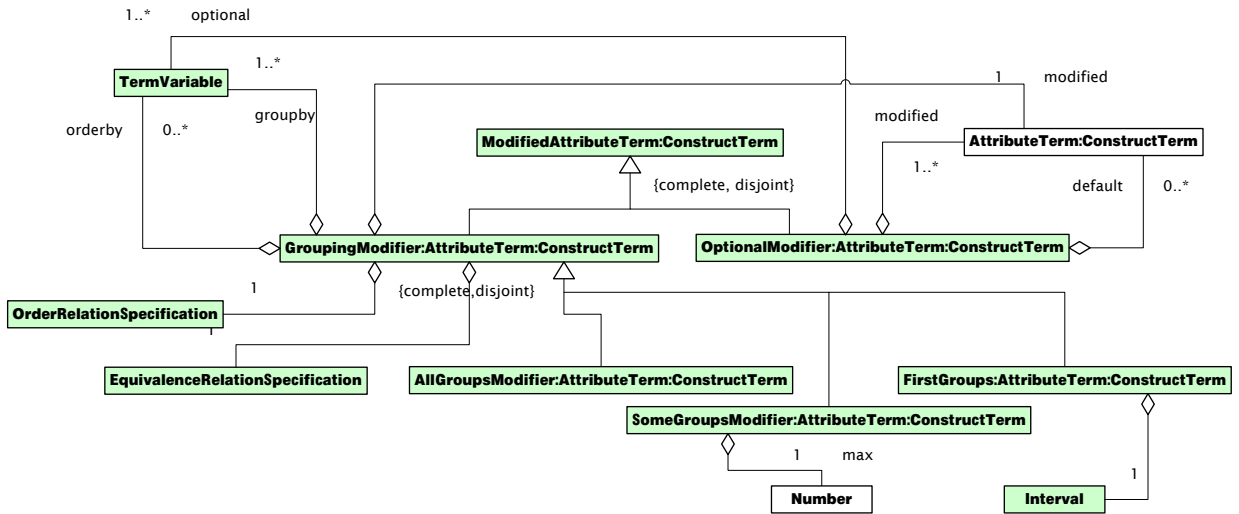
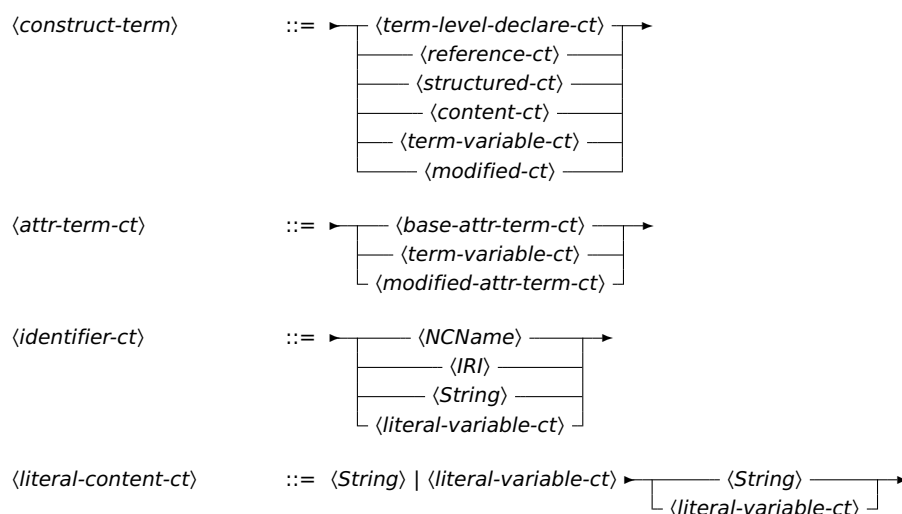


Figure 5.7: UML Model for Modified Attribute Construct Terms

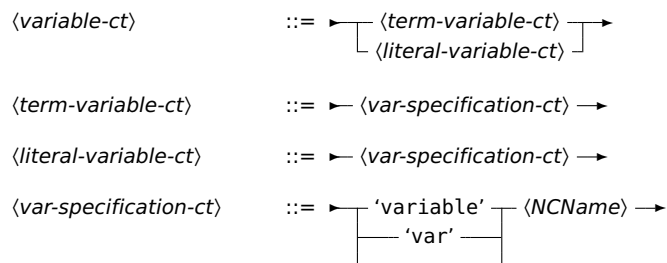
in construct terms they contain construct terms. In EBNF two separate non-terminals (and productions) are needed, since it is not possible to parameterize the production of declaration blocks with respect to the kind of contained terms.

Aside of this minor nuisance, the productions are very similar. In fact, except for the following four productions and the addition of variables and modifiers discussed below they are identical and will not be repeated in detail here. With the exception of the following four non-terminals and their productions all productions are copied from the data term syntax replacing the -dt prefix in all non-terminals by -ct. The full syntax can be found in Appendix A.3.

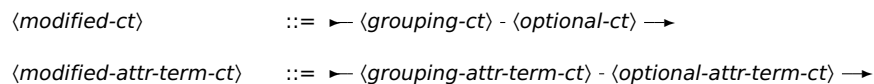
The deviation lies in the productions for the non-terminals $\langle \text{construct-term} \rangle$ and $\langle \text{attr-term-ct} \rangle$, where term variables and modified terms (for grouping and optionality) are added, as well as for $\langle \text{identifier-ct} \rangle$ and $\langle \text{literal-content-ct} \rangle$, where literal variables are added:



Though conceptually, literal and term variables are different and not always interchangeable they are not separated syntactically, as this is left to as-of-now unfinished type system. Syntactically variables are represented in the term syntax by names preceded with the keyword **variable** (or its shorthand var). A variable may occur without the leading keyword, if it is in the scope of a variable declaration that reserves the name of the variable, cf. Section 6 and 7.



Finally, modifiers occurring in construct terms can be distinguished in grouping and optional modifiers, the detailed syntax of which is discussed in the following sections.



Notice, that the grammar does not specify any order among the modifiers, i.e., both a grouping modifier in the scope of an optional modifier and vice versa are allowed.

5.3.2 XML-style Term Syntax

The XML-style term syntax once again uses the same productions as the non-XML term syntax, deviating from data terms (in XML-style term syntax) in exactly the same ways. The full grammar is shown in Appendix B.3.

5.3.3 Pure XML Syntax

In contrast to EBNF notation, Relax NG provides some means for parameterized grammars, as used above for defining declaration blocks. Indeed, construct terms can be defined using the same grammar as for data terms, but parameterizing `term.class` and `attribute.term.class`, as well as `identifier.class` and `literal-content.class`. In all cases the parameterization happens by adding additional choices to the existing ones: variables and modified terms or attributes.

Then definitions for variables as well as modified terms and attributes needs to be added to the grammar. Modified terms and attributes are defined using another parameterized grammar, this time the parameters are the content of the modifiers (once structured construct terms, once attribute construct terms) and what represents a variable. The details of that grammar are discussed in the following section.

```
1 construct-term =
  grammar {
3   variable-ct = parent variable-ct
   # Add grouping and optional for attributes
5   modified-attribute =
     grammar {
7     include "modifiers.rnc" {
       start = grouping
9     content = parent attribute-term.class
       variable = parent variable-ct
11    }
   }
13  | grammar {
     include "modifiers.rnc" {
15     start = optional
       content = parent attribute-term.class*
17     variable = parent variable-ct
     }
19  }
   # Add grouping and optional for elements
21 modified-term =
   grammar {
23   include "modifiers.rnc" {
     start = grouping
25   content = parent term.class*
     variable = parent variable-ct
27   }
   }
29  | grammar {
     include "modifiers.rnc" {
31     start = optional
       content = parent term.class*
33     variable = parent variable-ct
     }
35  }
```



```

37  ## Construct terms may also be variables or modified by
    ## grouping and optional modifiers.
39  term.class |= variable-ct | modified-term

41  ## Construct attribute terms may also be variables or modified by
    ## grouping and optional modifiers.
43  attribute-term.class |= variable-ct | modified-attribute
    # Add variables to identifiers and literal content
45  identifier.class |= variable-ct
    literal-content.class |= variable-ct
47  include "term.rnc"
}

```

5.4 Grouping in Construct Terms

§2 Grouping modifier

A grouping modifier expresses a grouping over the bindings of all its *grouping variables*. For alternative substitutions of the grouping variables, the construct in the scope of the grouping modifier are *repeated* once with the occurrences of the variables substituted accordingly.

A grouping modifier specifies four aspects of a grouping:

1. **Scope: What is to be repeated?** The scope of grouping modifiers for structured terms is a list of construct terms that is to be repeated. This allows, e.g., the bracketing of grouped terms or the creation of structures such as sections in HTML that are expressed through element delimiters instead of nesting. Grouping modifiers for attributes, however, only apply to a single attribute term, e.g., a variable or an attribute specification containing a variable for the name of the attribute. Lists of terms are not useful in this case, as attributes are always unordered and no two attributes of the same element node may have the same name.
2. **Groups: How to form groups?** An essential part of grouping is the determination of the actual groups: i.e., to specify when two substitutions of the associated grouping variables are considered **equivalent** and thus part of the same “group” (i.e., equivalence class).

Commonly, query languages use a *single, pre-defined equivalence relation* for grouping, e.g., SQL uses equivalence based on the *typed value* of the grouping attributes, i.e., all tuples with the same typed value for the grouping attributes are considered as one group. In object-oriented or semi-structured query languages, one as finds equivalence based on object or *node identity*, i.e., substitutions for the grouping variables are considered equivalent only if they have for each grouping variable the very same nodes as substitution.

In Xcerpt, the default equivalence relation is *structural equivalence*, i.e., two bindings are considered equivalent if their label, children, and/or content is equal (formally, structural equivalence in Xcerpt uses the notion of simulation as defined in [49, 14], cf. Section 6.1). Beyond this default equivalence, Xcerpt’s grouping modifiers may also explicit specify an **equivalence relation** that relates equivalent substitutions for the grouping variables. It must adhere to the usual definition of an equivalence relation, i.e., it must be a reflexive, symmetric, and transitive relation over the domain

3. **How to order the repeated terms?** An order among the groups established in point (2) is needed not only in the case of **first**-selecting grouping terms, but also if the grouping term is contained in a (structured) term where the order of the children is significant. In both cases, the order among the groups is defined by (a) a list of *ordering variables* and (b) a *total order relation* (i.e., a reflexive, antisymmetric, transitive, and comparable relation) on the substitutions for ordering variables. Note, that the ordering variables must be a subset of the grouping variables. Also note, that the order relation must be consistent with the equivalence relation, i.e., whenever $b_1 \leq b_2$ and $b_2 \leq b_1$ for the order relation \leq use in a grouping term G and bindings b_1, b_2 , then $b_1 \sim b_2$ for the equivalence relation \sim used in G .
4. **Group Selection: Which of the groups to consider?** In many cases, the grouping should only iterate over certain of the groups. Xcerpt addresses the selection of relevant groups by providing three grouping modifiers:
 - The **All**-Groups modifier uses *all* of the groups established as explained in point (2).
 - The **Some**-Groups modifier uses *some* of the groups: At most m groups are selected *arbitrarily* and possibly non-deterministically. *At most m* , as there may be less than m groups, in which case, all groups are selected.
 - The **First**-Groups modifier uses *some* of the groups, but the selection is determined by the order of the groups: An interval $n - m$ specifies that the n^{th} to m^{th} group are to be used. Again, there may be less than m (in which case all groups after and including the n^{th} group are used) and even less than n groups (in which case no group is used). The order of the groups is defined by the order relation described in the previous point.

Grouping modifiers may be *nested* leading to the expected behavior: say a grouping over authors of books contains another grouping over titles of books. In the constructed data, the terms constructed by the grouping over titles are contained in the terms constructed by grouping over authors based on the author-title combinations found in the substitutions. Intuitively, nested grouping constructs are similar to nested **for**-loops in imperative programming languages.

To summarize Xcerpt's grouping modifiers allow the repetition of subterms based on substitutions for grouping variables. They allow extensive customization of what defines a group and how to order the repetitions without sacrificing simplicity in common cases.

Like in all query languages, where the result of a query can have a complex (structured) shape, *grouping is not only essential in combination with aggregation* (as in relational query languages), but also to define how the nesting of the result is constructed based on the relations of data items selected by a query in variable bindings. The nature of data with complex shape also requires the support of nested grouping, i.e., repetition within repetition.

5.4.1 Textual Term Syntax

The non-XML term syntax for grouping modifiers in construct terms closely reflects the four aspects of the abstract syntax:

1. *Group selection* is indicated using the three different keywords **all**, **some**, **first**. **some** is followed by a number (or a variable) that indicates the number of groups to select. **first** is followed by an interval specification, i.e., two numbers (or variables) separated by a $-$. Two shorthands for intervals are provided: $n-$ to select all groups starting with the n^{th} and $+$ as abbreviation for $1-$. Thus **first** $1-$ is equivalent to **all**.

2. The *scope* of the modifiers are the construct terms included in parentheses after the modifier. As in declaration blocks the parentheses may be omitted, if the scope is exactly one construct term.
3. *Groups* are formed using the optional equivalence relation on the bindings of the grouping variables. *Grouping variables* are either implicit or explicit. Implicit grouping variables are all free variables in the scope of the grouping modifier, i.e., all variables that occur in the scope of the grouping modifier but not in the scope of another nested grouping modifier. Explicit grouping variables are specified in a list enclosed by parentheses after the **order-by** keyword. Again the parentheses may be omitted if the list is a singleton.
4. The *order* of the groups is determined by the order variables (a subset of the grouping variables) and the order relation. The order variables are specified in a list (enclosed by parentheses) after the keyword **order-by**. Again the parentheses may be omitted if the list is a singleton. Notice, that if both are present **order-by** follows **group-by**.

For attributes the specification is similar, but only a single attribute construct term may in the scope of a grouping modifier. This prevents the repetition of same-name attributes (recall, that attributes are essentially (key, value) pairs in a dictionary associated with their structured term and duplicate keys are forbidden in accordance to XML).

$\langle \text{grouping-ct} \rangle ::= \text{grouping-modifier} \text{'('} \langle \text{construct-term} \rangle \text{' , ' } \langle \text{construct-term} \rangle \text{' , ' } \dots$
 $\dots \langle \text{groupby} \rangle \langle \text{orderby} \rangle$

$\langle \text{grouping-attr-term-ct} \rangle ::= \text{grouping-modifier} \text{'('} \langle \text{attr-term-ct} \rangle \text{')' } \dots$
 $\dots \langle \text{groupby} \rangle \langle \text{orderby} \rangle$

$\langle \text{grouping-modifier} \rangle ::= \text{'all'}$
 $\text{'some' - } \langle \text{number-ct} \rangle$
 $\text{'first' - } \langle \text{interval-ct} \rangle$

$\langle \text{orderby} \rangle ::= \text{'order-by' ' (} \langle \text{optional-variable} \rangle \text{' , ' } \langle \text{variable-ct} \rangle \text{' , ' } \dots$
 $\dots \langle \text{order-relation} \rangle$

$\langle \text{order-relation} \rangle ::= \text{'ascending'}$
 'descending'
 $\langle \text{NCName} \rangle$

$\langle \text{groupby} \rangle ::= \text{'group-by' ' (} \langle \text{optional-variable} \rangle \text{' , ' } \langle \text{variable-ct} \rangle \text{' , ' } \dots$
 $\dots \langle \text{equivalence-relation} \rangle$

$\langle \text{equivalence-relation} \rangle ::= \blacktriangleleft \langle \text{NCName} \rangle \longrightarrow$
 $\langle \text{optional-variable} \rangle ::= \blacktriangleleft \langle \text{optional-modifier} \rangle - \langle \text{variable-ct} \rangle \longrightarrow$
 $\langle \text{interval-ct} \rangle ::= \blacktriangleleft \langle \text{number-ct} \rangle - '-' - \langle \text{number-ct} \rangle - \langle \text{number-ct} \rangle - '-' - '+' \longrightarrow$
 $\langle \text{number-ct} \rangle ::= \blacktriangleleft \begin{array}{c} \langle \text{Int} \rangle \\ \lrcorner \langle \text{literal-variable-ct} \rangle \lrcorner \end{array} \longrightarrow$

5.4.2 XML-style Term Syntax

The same productions as for the non-XML term syntax can be used for the XML-style term syntax. The full grammar is given in Appendix B.3.

5.4.3 Pure XML Syntax

As seen above, the pure XML syntax can utilize parameterized grammars not just for construct terms in general, but also for modifiers itself. Figure 5.8 shows the Relax NG schema for that grammar. As in the grammar for declaration blocks, the content pattern is to be overwritten when importing this grammar. Additionally also the variable pattern can be replaced to specify the shape of variable occurrences.

The following listing gives the textual grammar in Relax NG's compact syntax:

```

default namespace = "http://xcerpt.org/ns/core/1.0"
2 namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"

4 start = grouping
  content = empty
6 grouping =
  element all { content, order-by?, group-by? }
8 | element some { number, content, order-by?, group-by? }
  | element first { interval, content, order-by?, group-by? }
10 order-by =
  element order-by {
12   attribute order-relation { text }?,
    optional-variable+
14 }
  group-by =
16 element group-by {
  attribute equivalence-relation { text }?,
18   optional-variable+
  }
20 optional-variable =
  element optional { variable }
22 | variable
  variable = empty
24 interval =
  element interval {
26   element min { number-literal.class },
    element max { number-literal.class }
28 }
  number = element number { number-literal.class }
30 number-literal.class = xsd:int | variable

```

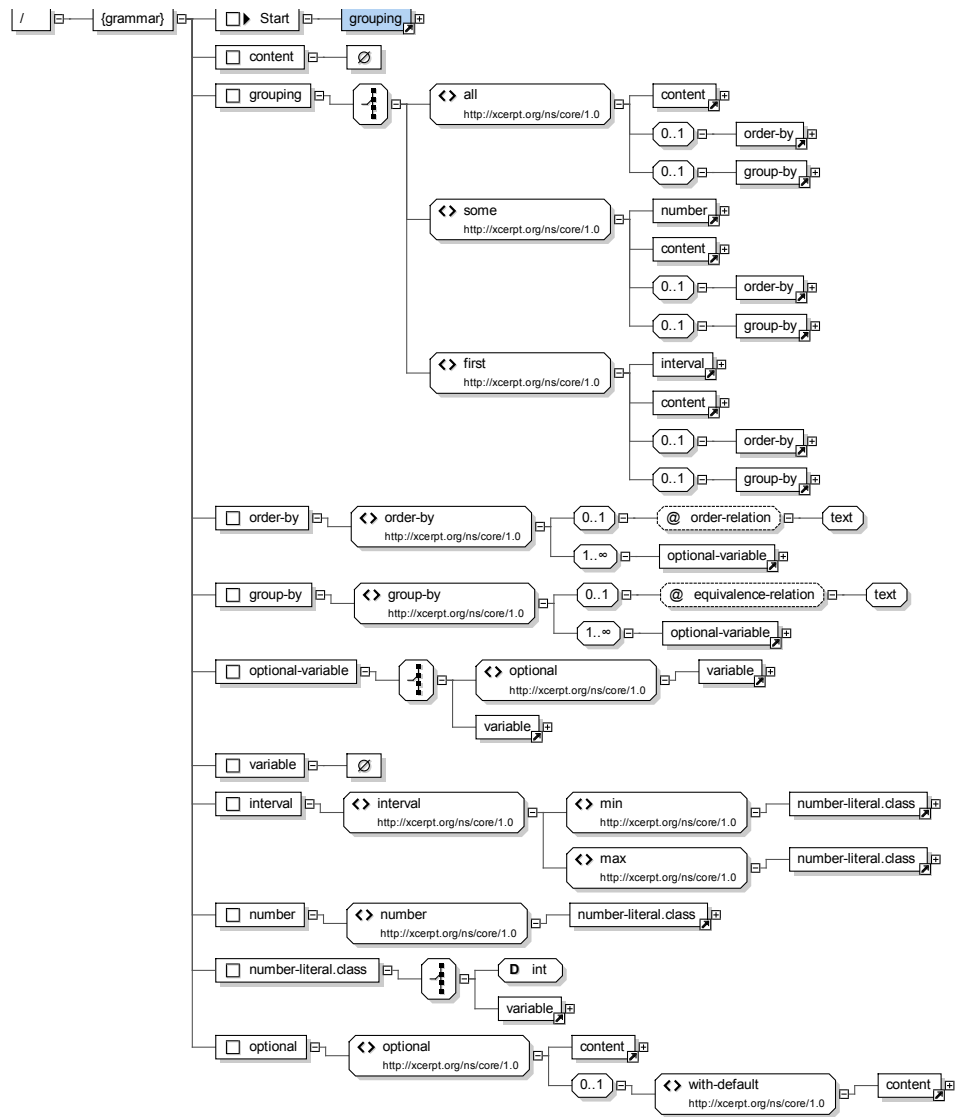


Figure 5.8: Relax NG Grammar for Modifiers in Construct Terms

```

optional =
32  element optional {
    content,
34  element with-default { content }?
}

```

As in the term syntax, grouping variables can be specified explicitly or implicitly, i.e., in the group-by subelement or by occurring as free variables inside the scope of the grouping modifier.

5.5 Optional Construct Terms

Aside of grouping to collect alternative bindings for a variable, Xcerpt's construct terms add one more modifier to handle the case where a variable may have no bindings in a given substitution: the optional construct term.

§3 Optional Modifier

An optional modifier in construct terms specifies a form of *conditional construction*: some of the variables reported as result of a query may not have any bindings in some substitutions, as they occur only in optional parts of a query (cf. Section 6.1). In this case, an optional modifier must be used in the construct term to mark the part of the construct term that depends on the existence of bindings for the optional variables associated with the optional modifier.

Recall, Figures 5.6 and 5.7 for the precise model of optional construct terms. An optional modifier in construct terms needs three parameters, the first two are similar to those for grouping modifiers:

1. **Scope: Which part of the construct term is optional?** The scope of an optional modifier are the modified construct terms. Again, a list of construct terms is allowed to facilitate optional parts that cover several siblings, e.g., for bracketing. In contrast to grouping modifiers for attribute terms, optional modifiers have also in the case of attribute terms a *list* of (attribute) construct terms as scope, since the problem with repeated attribute names does not occur in this case (since there is no repetition).

The construct terms in the scope of an *optional modifier are the only place, where optional variables may occur* in construct terms. More precisely, only those optional variables that an optional modifier O (or an optional modifier that O is part of) depends on, may occur in O .

2. **Condition: On bindings for which variables depends the conditional construction?** Optional modifiers specify a set of optional variables that are used to determine whether the construct terms in the scope of the optional modifier are part of the result or not: They are included in the result for a substitution σ only if bindings for all the specified optional variables exist in σ . Note, that all variables that an optional modifier depends on *must be optional* in the query as well (cf. Section 6.1).
3. **Default value:** Finally, an optional modifier also specifies a default value in the form of another list of construct terms. The default value is used in the result for a substitution σ if for one of the optional variables no binding exist in σ . Note, that the default value *must not contain any of the optional variables*, as they may have no binding.

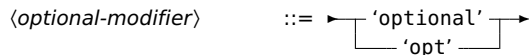
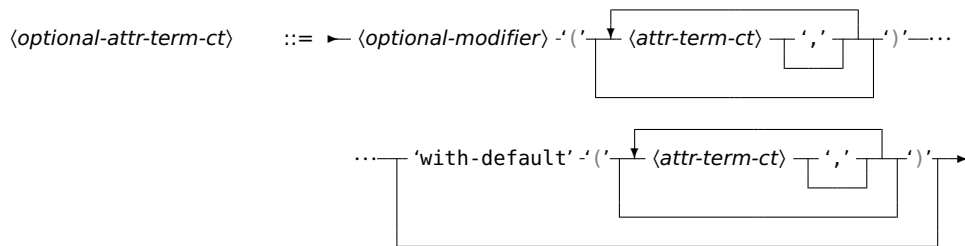
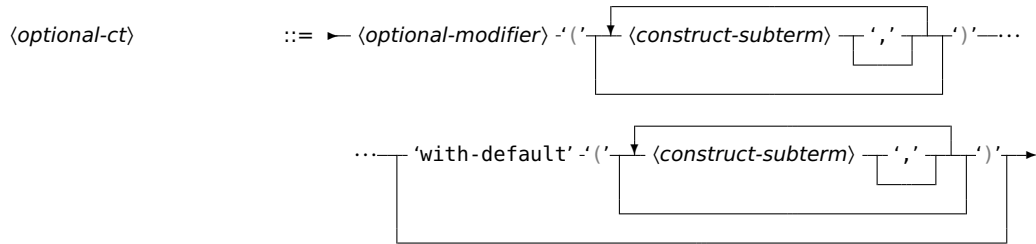
Notice how an optional construct term resembles a conditional expression of the form **if** $\langle condition \rangle$ **then** $\langle modified\ construct\ terms \rangle$ **else** $\langle default\ construct\ terms \rangle$.

The condition is, however, always fixed to the existence for bindings for all optional variables in a substitution (cf. Issue 2 for a discussion on general conditional expressions in Xcerpt construct terms).

5.5.1 Textual Term Syntax

Optional modifiers follow very much the syntax of grouping modifiers: the in-scope construct terms follow the **optional** keyword (or its shorthand **opt**) enclosed in parentheses. The parentheses may be omitted if the scope is a single construct term, as usual. The list of in-scope terms is followed by an optional specification of the default terms, i.e., those terms that are used in the construction, if for one of the optional variables no binding exists. The specification of the default terms is precluded by the **with-default** keyword and, as usual, enclosed in parentheses that may be omitted if there is a single default term.

The optional variables can *only* be specified implicitly (cf. Issue 2), i.e., all free variables inside the **optional** modifier form the list of optional variables for which bindings must exist so that the in-scope terms are constructed.



5.5.2 XML-style Term Syntax

Again there are no deviations in the productions and non-terminals for the XML-style term syntax. A full grammar is given in Appendix B.3.

5.5.3 Pure XML Syntax

Optional modifiers follow the general syntax for modifiers as introduced in Section 5.4. As in the term syntax, optional variables can only be specified implicitly.

```

fun instantiate( $\mathcal{C}$ : construct term list,  $B$ : substitution multi-set,
                $V$ : variables,  $\sim$ : equivalence relation)
   $R \leftarrow$  initially empty list of resulting construct terms
  for all  $t$  in  $\pi_{\sim}^{-1}(B)$  do
    for all  $C \in \mathcal{C}$  do
       $B^- \leftarrow B \setminus \pi_V(B)$ 
      replace  $V$  in  $C$  by bindings for  $V$  in  $t$ 
      for all  $G$  such that  $G$  is a grouping modifier directly in  $C$  do
         $T_G \leftarrow$  the list of modified construct terms of  $G$ 
         $V_G \leftarrow$  grouping variables in  $G$ 
         $\sim_G \leftarrow$  equivalence relation used in  $G$ 
        replace  $G$  by instantiate( $T_G, B^-, V_G, \sim_G$ )
      end for
      for all  $O$  such that  $O$  is an optional modifier directly in  $C$  do
         $V_O \leftarrow$  optional variables in  $O$ 
        if  $\forall v \in V_O : \exists b \in B : b$  is a binding for  $v$  then
           $T_+ \leftarrow$  the list of modified construct terms of  $O$ 
          replace  $O$  by sequence of instantiate( $T_+, B, V, \sim$ )
        else
           $T_- \leftarrow$  the list of default construct terms of  $O$ 
          replace  $O$  by sequence of instantiate( $T_-, B, V, \sim$ )
        end if
      end for
       $R \leftarrow$  append  $C$  to  $R$ 
    end for
  end for
  return  $R$ 
end fun

```

Algorithm 5.1: Instantiating a Construct Term (with π_{\sim}^{-1} understood as the projection for tuples using \sim to remove duplicates and “directly in a term T ” understood as “occurs in the scope of T but not within the scope of a nested grouping or optional construct term”)

5.6 Instantiating a Construct Term

Summarizing, a construct term C can be “instantiated” by a substitution multi-set B . If each substitution in the substitution multi-set maps all non-optional variables occurring in the construct term to data terms, the result of the instantiation is a data term. The details of the instantiation are described in [50]. The instantiate function shown in Algorithm 5.1 provides an instantiation of a construct term, if called with parameters C , B , the *free* variables in the construct term, i.e., all variables that occur (also) outside of any grouping or optional term, and the empty relation \emptyset as last parameter (i.e., an equivalence relation such that each binding tuple is equivalent to itself only).

In other words, the resulting data terms are obtained by

1. replacing all *free* variables in the construct term, i.e., all variables that occur (also) outside of any grouping or optional modifier, by bindings from each tuple. If there are no free variable a single resulting data term is constructed.

2. replacing each grouping modifier by repeating the in-scope construct terms for each group of grouping variables, each time replacing all occurrences of the grouping variables by the particular group's bindings. This is done recursively for all grouping modifiers.
3. replacing each optional modifier by the in-scope construct terms, if there exists a combination of bindings for all optional variables.

Example 5.1 (Construct Terms). The following substitutions for the variables Author, Title, and Publication are given as result of a query:

Author	Title	Publication
"Cicero"	"Data Processing ..."	null
"Cicero"	"Space and ..."	journal ²
"Antonius"	"Advancements ..."	null
"Antonius"	"Efficient Manage..."	proceedings ²⁹
"Tiro"	"Space and ..."	journal ²

Notice, that Publication is an optional variable.

Then the following construct term in XML-style term syntax

```

1 <result>
  all <author> var Author </author>
3 </result>

```

results in the single data term

```

1 <result>
  <author>"Cicero"</author>
3  <author>"Antonius"</author>
  <author>"Tiro"</author>
5 </result>

```

Notice, how Xcerpt defaults to grouping by structural equivalence and thus treats the two substitutions with author "Cicero" as one group, constructing only a single result data term for them.

If we add Title as free variable in the scope of the grouping modifier, the grouping variables and thus the groups change:

```

1 result()[
  all author()[
3     var Author
     title()[ var Title ]
5   ]
]

```

Leading to the result:

```

result()[
2  author()[ "Cicero"
     title()[ "Data Processing ..." ] ]
4  author()[ "Cicero"
     title()[ "Space and ..." ] ]
6  author()[ "Antonius"

```

```

    title()[ "Advancements ..." ] ]
8  author()[ "Antonius"
    title()[ "Efficient Manage..." ] ]
10 author()[ "Tiro"
    title()[ "Space and ..." ] ]
12 ]

```

Now the substitutions for author and title are both considered for forming a group, leading to more groups!

Nesting grouping modifiers also affects the free variables, e.g., in the following construct term Title is no longer free for the out **all** only for the inner.

```

result()[
2  all author()[
    var Author
4    all title()[ var Title ]
    ]
6 ]

```

Thus the result on the sample substitutions is:

```

result()[
2  author()[ "Cicero"
    title()[ "Data Processing ..." ]
4    title()[ "Space and ..." ]
    ]
6  author()[ "Antonius"
    title()[ "Advancements ..." ]
8    title()[ "Efficient Manage..." ]
    ]
10 author()[ "Tiro"
    title()[ "Space and ..." ] ]
12 ]

```

Combining grouping an optional modifiers can lead to surprisingly expressive constructs:

```

result()[
2  all author()[
    var Author
4    all (title()[ var Title ]
        optional var Publication
6        with-default standalone()[ ])
    ]
8 ]

```

Results in the following data term:

```

result()[
2  author()[ "Cicero"
    title()[ "Data Processing ..." ]
4    standalone()[ ]
    ]
6  author()[ "Cicero"
    title()[ "Space and ..." ]
8  journal.adm @ journal() [ ... ]

```

```
    ]
10  author()[ "Antonius"
      title()[ "Advancements ..." ]
12    standalone()[ ]
      ]
14  author()[ "Antonius"
      title()[ "Efficient Manage..." ]
16    conf.dmmc @ proceedings()[ ... ]
      ]
18  author()[ "Tiro"
      title()[ "Space and ..." ]
20    journal.adm @ journal()[ ... ]
      ]
22 ]
```


Chapter 6

How to specify queries? Part 2: Selection

6.1 Specifying Query Patterns: Query Terms

As introduced above, query terms are the second part of expressing the derivation of new data in Xcerpt: where construct terms dictate the shape of the new data, query terms specify (possibly incomplete) patterns for data that is to be found, e.g., in Web resources such as XML pages or RDF resource descriptions. As construct terms, query terms enrich basic data terms by variables, but here variables serve to identify data that is to be extracted by the query in form of variable bindings.

Query terms are “matched” with data or construct terms by a non-standard unification method called *simulation unification* that is based on a relation called *simulation* (for details see [14]). In contrast to Robinson’s unification (as e.g. used in Prolog), simulation unification is capable of determining substitutions also for incomplete and unordered query terms. Since incompleteness usually allows many different alternative bindings for the variables, the result of simulation unification is not only a single substitution, but a (finite) *multi-set of substitutions*, each of which yielding ground instances of the unified terms such that the one ground term matches with the other.

§1 Query term

Query terms specify *structure and values of data that is to be matched* and which parts of the matched data are to be extracted. In that, they are comparable to clause bodies in Datalog or **FROM** and **WHERE** clauses in SQL.

Query terms differ more notably from data terms than construct terms do, as they add additional features beyond variables that are essential to express patterns for data, when the data may vary or only limited knowledge about the (shape of the) data is available: In detail, query terms deviate from basic data terms in essentially three aspects (cf. Figure 6.1): the addition of variables, the support for incomplete patterns, and the use of term formulas to express conjunctions, disjunctions, and negations.

To better understand these extensions, an intuition of the answer notion in Xcerpt is needed. The questions, which data and construct terms match with a query term, and what the answer (i.e., the substitution multi-set) for a query term is, are formally addressed in [49, 50]. At the root of Xcerpt’s answer notion stands an extended form of rooted graph simulation (cf. [46, 35] and [31, 26] for more

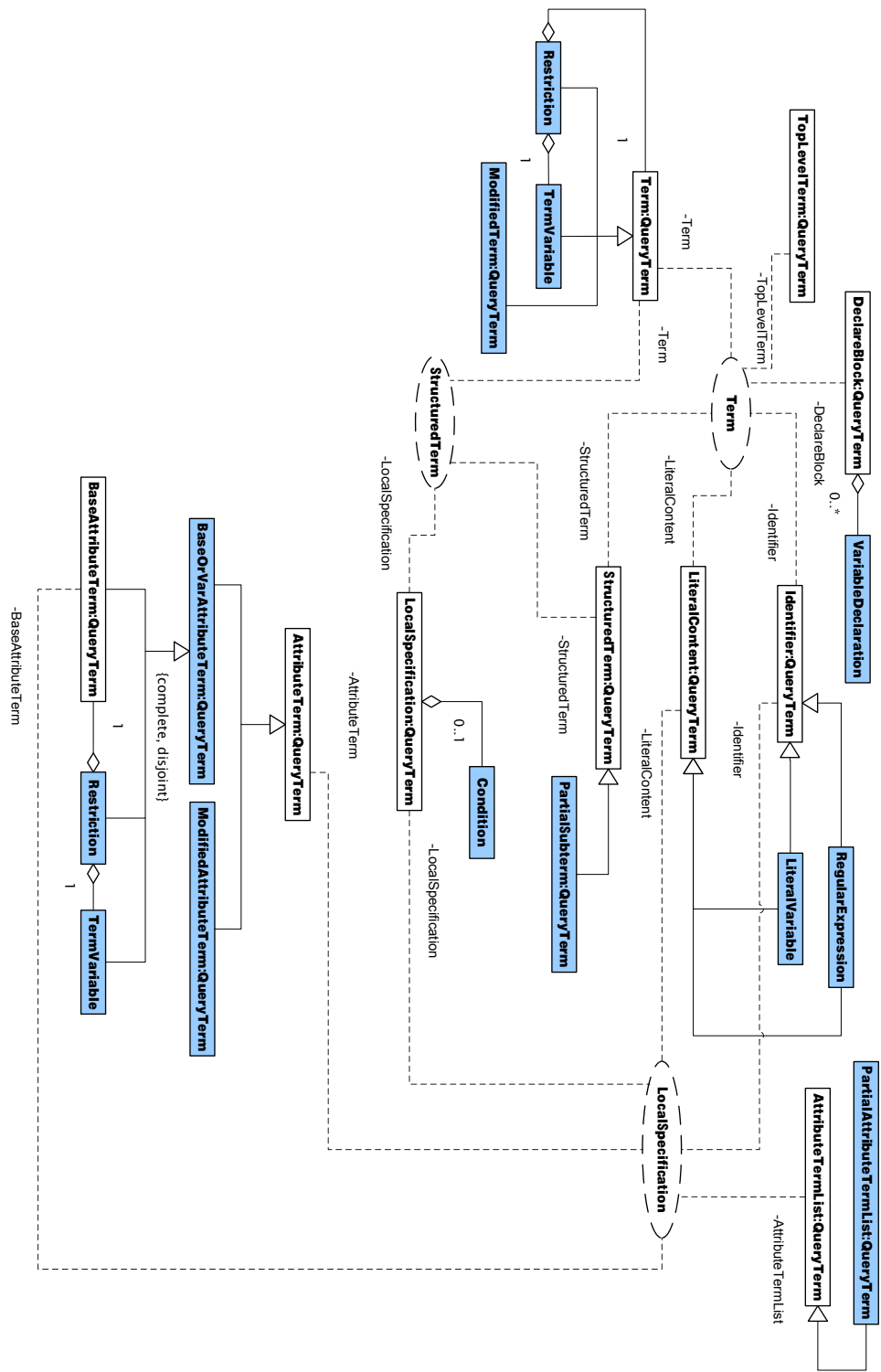


Figure 6.1: UML Model for Query Terms using Parameterized Collaborations

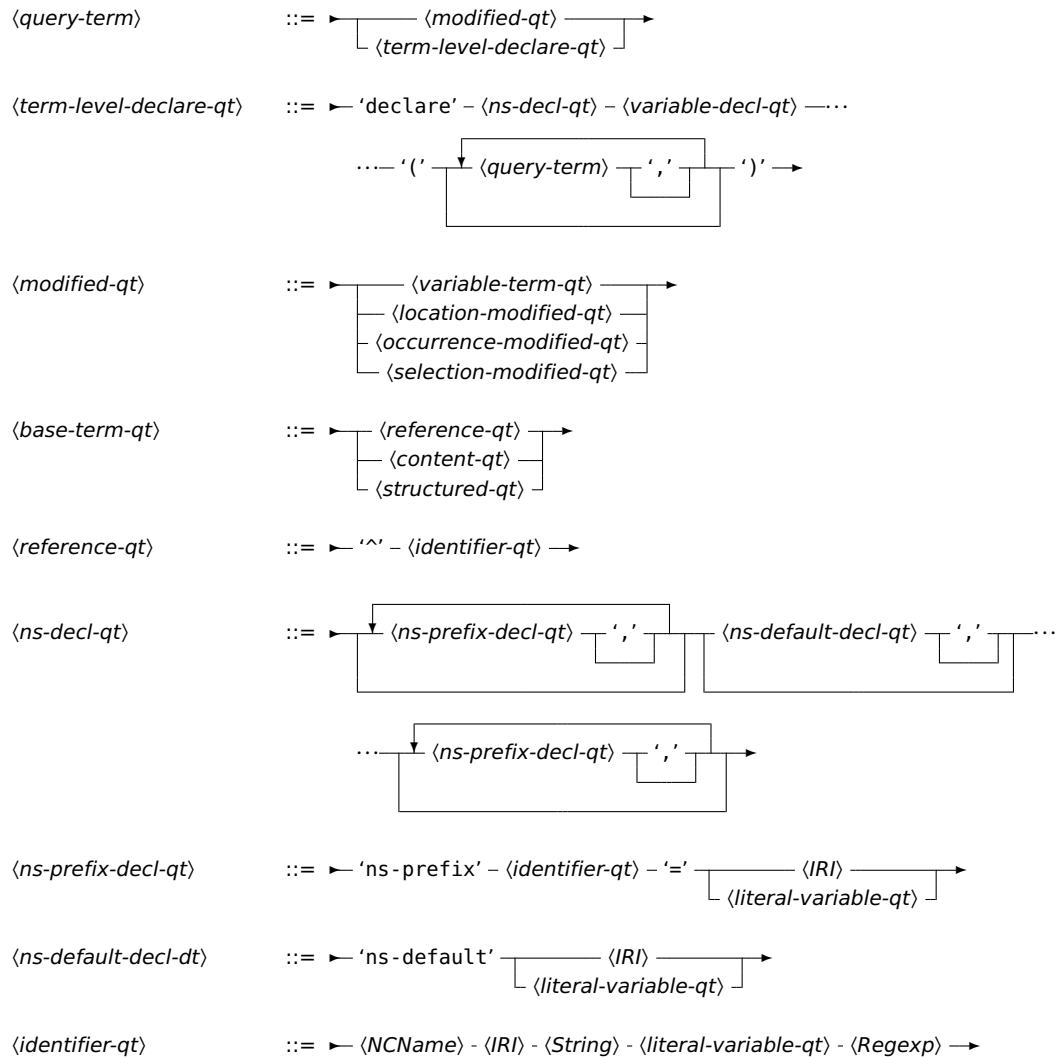
recent work on efficient algorithms for computing simulation and bisimulation). This extension of the classical notion is necessary to accommodate incomplete patterns as discussed below in Section 6.3.

Intuitively, a query term without any of the extensions discussed in the following matches only with a data term that has *exactly the same shape modulo reordering of direct sub-terms in unordered structured terms and of attributes in any terms*. In the following it is noted, how each of the extensions affect the matching of query terms, but the full details are left to [50].

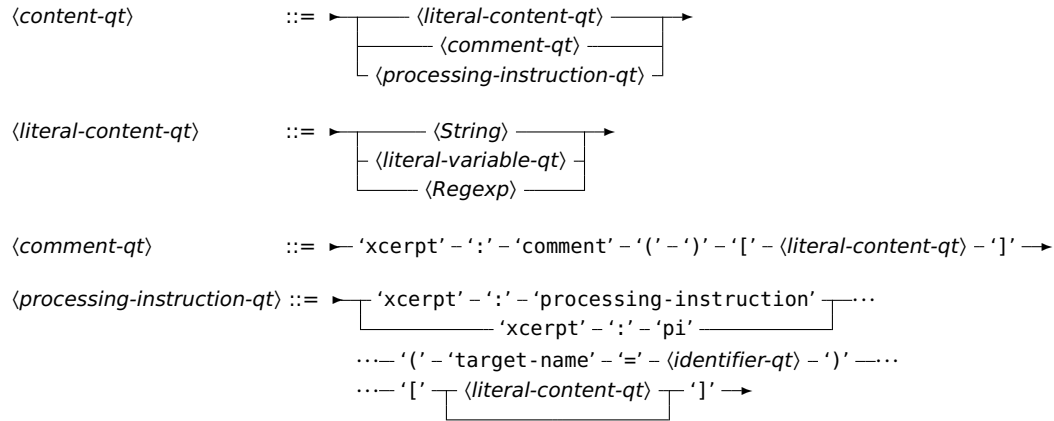
6.1.1 Textual Term Syntax

The following grammar defines the basic non-terminals for query terms. Notice, the added non-terminal $\langle \text{modified-qt} \rangle$. It represents query terms that are possibly modified by variables or operators discussed in the remainder of this chapter.

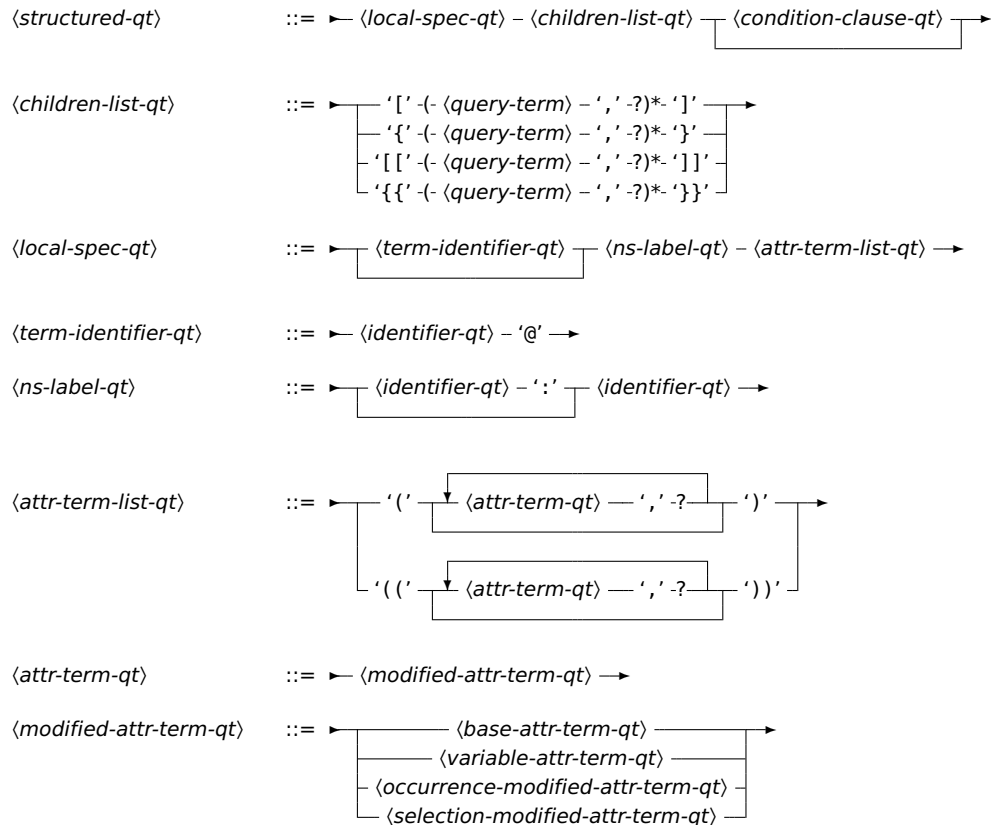
Declaration blocks in query terms may also contain variable declarations the details of which are discussed in the next section.



Content Query Terms Though the productions for content query terms remain unchanged, the introduction of variables into literal content and identifiers in the next sections indirectly also affect content query terms.



Structured Query Terms Aside of the introduction of variables and their new found ability to occur inside modifiers, structured query terms are nearly identical to structured data terms. Additionally, query terms may be incomplete indicated by double braces or brackets enclosing the list of children. Also query terms can be annotated with condition clauses as described in the next section.



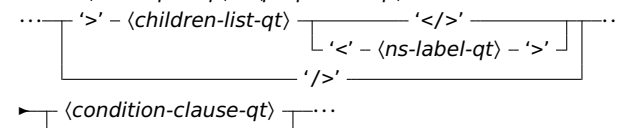
$\langle \text{base-attr-term-qt} \rangle ::= \blacktriangleright \langle \text{ns-label-qt} \rangle - '=' - \langle \text{literal-content-qt} \rangle \rightarrow$

6.1.2 XML-style Term Syntax

Once more, the XML-style term syntax is closely aligned with the non-XML term syntax, but differs in the representation of comments, processing-instructions, and properties of structured terms. Here, incomplete term specifications are indicated with '{partial}' (and complete or total terms specifications with '{total}'). The same applies for incomplete attribute term lists. As in data and construct terms, query terms are syntactically similar to XML elements, but may contain Xcerpt constructs additionally.

$\langle \text{comment-qt} \rangle ::= \blacktriangleright ' < ! - ' - \langle \text{literal-content-qt} \rangle - ' - > ' \rightarrow$

$\langle \text{processing-instruction-qt} \rangle ::= \blacktriangleright ' < ? ' - \langle \text{identifier-qt} \rangle - \langle \text{literal-content-qt} \rangle - ' ? > ' \rightarrow$

$\langle \text{structured-qt} \rangle ::= \dots - ' < ' - \langle \text{local-spec-qt} \rangle - \langle \text{properties-qt} \rangle \rightarrow$
 $\dots - ' > ' - \langle \text{children-list-qt} \rangle \dots$

 $\blacktriangleright \langle \text{condition-clause-qt} \rangle \dots$

$\langle \text{properties-qt} \rangle ::= \dots - \{ ' - \text{'ordered' - '}' \} - \{ ' - \text{'unordered' - '}' \} - \{ ' - \text{'total' - '}' \} - \{ ' - \text{'partial' - '}' \} \rightarrow$
 $\blacktriangleright \{ ' - \text{'total attributes' - '}' \} - \{ ' - \text{'partial attributes' - '}' \} \dots$

$\langle \text{children-list-qt} \rangle ::= \blacktriangleright \langle \text{query-term} \rangle \rightarrow$

$\langle \text{attr-term-list-qt} \rangle ::= \blacktriangleright \langle \text{attr-term-qt} \rangle \rightarrow$

6.1.3 Pure XML Syntax

Once more, the pure XML syntax for query terms relies heavily on the parameterizable Relax NG grammar for terms introduced in Section 4.1. Evidently, query terms can become the most complex of the three term kinds in Xcerpt, cf. Figure 6.2. As construct terms they add variables to data terms. But they also provide means for expressing incompleteness: partial terms, **descendant** and **position** location modifiers, etc.

Query terms deviate from the basic term case in (a) extended top-level terms, (b) the introduction of modified-terms, (c) the use of condition formulas attached to each term, and (d) the use of variables and regular-expressions in identifiers and literal content.

```

## A POSIX.1 regular expression annotated with variables may occur in
2 ## query terms at the position of identifiers or literal content.
regular-expression =
4 element regexp {
    attribute value { text }
6 }

8 query-term =

```

```

grammar {
10
    include "term.rnc" {
12      # Redefine the top-level term for query terms: add variables to
      # declare blocks and allow optional, descendant, variable restriction.
14      # Add document specifications
      # Add query term formula

16
      ## A term that may occur at top-level. Slightly more
18      ## restricted than a basic term.
      top-level-term.class =
20      optional-top-level-term
      | term-formula
22      | document-specification
      | grammar {
24          include "declare-block.rnc" {
              content = parent top-level-term.class*
26          }
      }

28      # Redefine terms: only modified terms, which can in fact be
      # unmodified :-) Term-level declare blocks may also contain variable
30      # declarations

32      ## A generic Xcerpt term. Variants are data, construct, and query terms.
      term.class = modified-term | term-level-declare

34
      ## A declaration block on term level allows possibly (in data and construct terms) only namespace
      declarations.
36      term-level-declare =
          grammar {
38          include "declare-block.rnc" {
              content = parent term.class*
40          }
          }

42
      ## An attribute term is an attribute possibly modified with respect to location, modality, and
      selection.
44      attribute-term.class = modified-attr-term
      # Allow conditions on arbitrary query terms
46      term-condition = condition-clause

48      }

50      # Add variables and regular expressions to identifiers and literal
      # content
52      identifier.class |= variable | parent regular-expression
      literal-content.class |= variable | parent regular-expression

54
      ## Variables for query terms.
56      variable =
          element variable {

```

```

58     attribute anonymous { "true" }
        | attribute name { xsd:NCName }
60     }

62     # #1# TOP-LEVEL QUERY TERM
        # see below

64     # #2# CONDITION CLAUSES
        # see below

66     # #3# MODIFIED TERMS AND ATTRIBUTE TERMS
        # see below

70     }

```

6.2 Variables in Query Terms

Variables in query terms are used for three purposes: (a) to specify *which parts of a matched (construct or data) term are “selected”* by the query and can be used in the corresponding construct term, (b) to specify *joins*, i.e., multiple occurrences of the same data term or literal value (usually unknown at time of query authoring), and (c) to specify arithmetic or other *conditions* involving (literal) values of variables.

Like in construct terms variables may be used in query terms in place of (a) (structured or attribute) terms or in place of (b) identifiers and literal content. In either case, a variable matches (unless further restricted as discussed below) any sub-term or literal that may occur at that position, i.e., regardless of the shape of the sub-term or literal.

Variables are *named* so that they can be referred to in other parts of the query term (forming a join) or in the corresponding construct term. Xcerpt allows in addition to named variables also contain **anonymous variables** (like in Prolog). As unrestricted named variables, an anonymous variables matches arbitrary terms or literals that may occur at the position of the variable. However, bindings for anonymous variables are not recorded and different occurrences of the same anonymous variables are treated like different named variables. Thus, anonymous variables can neither be used for joins, nor be restricted through variable restrictions or conditions, nor occur in construct terms. Their sole purpose is to act as a wildcard construct.

Additionally, query terms may contain so-called **variable restrictions**, where a variable does not just replace some sub-term (and thus is bound to all sub-terms in a matching data term that can occur at that point), but the sub-terms that may be bound to the variable are further restricted by specifying a arbitrary query term.

§2 Variable Restrictions

A variable restriction places a *constraint on the structure* of (data or construct) terms that can be bound to the restricted variable by specifying the possible shapes of such (data or construct) terms as a query term.

Variable restrictions may only occur in place of structured and attribute query terms, not in place of identifiers or literal content (cf. Issue 8). Figure 6.3 shows variable restrictions in the context of basic query terms: Each variable restriction restricts one (term) variable to a (basic) query term (the scope of that variable restriction). Variable restrictions for attribute terms are analogous.

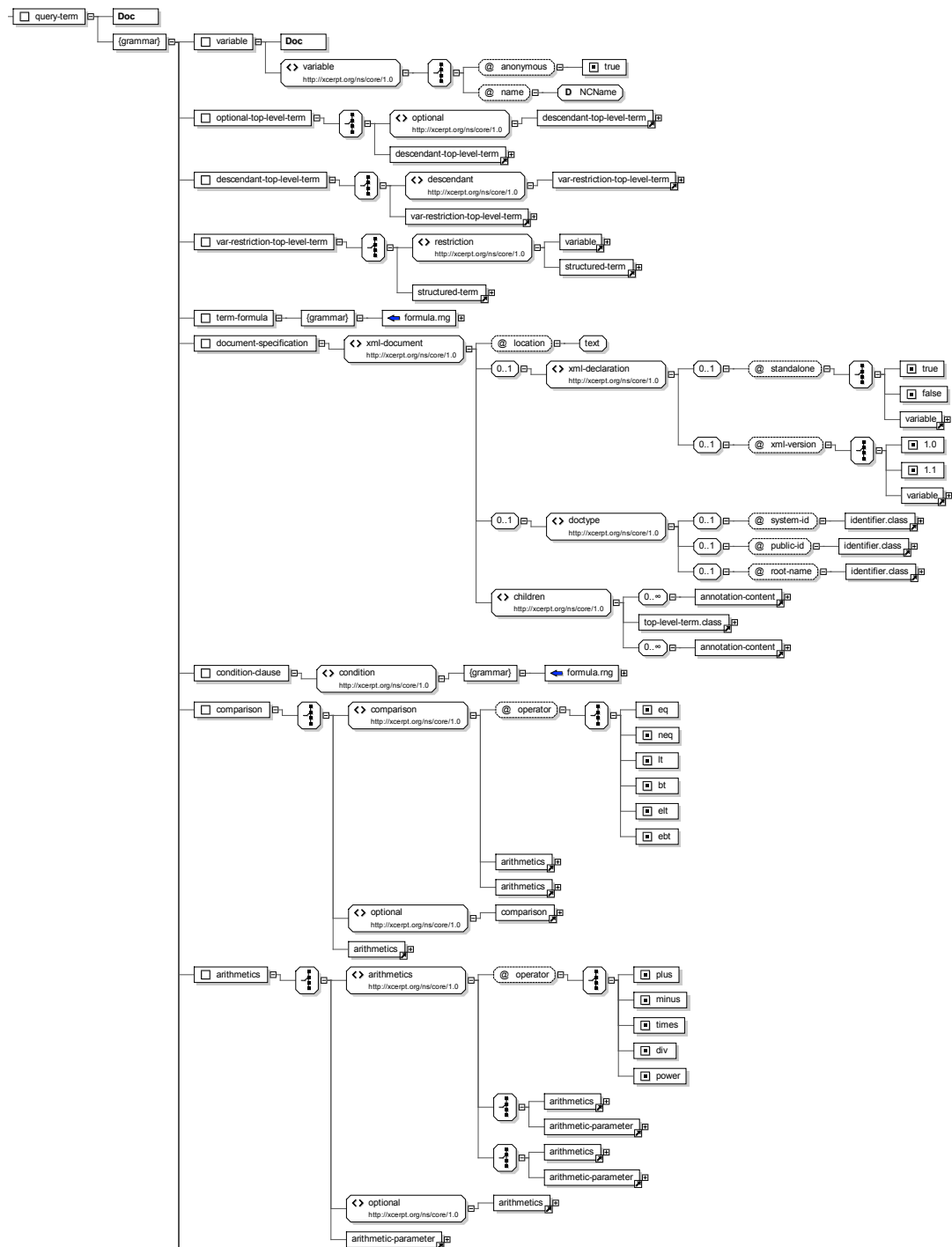


Figure 6.2: Relax NG Grammar for Query Terms (Excerpt)

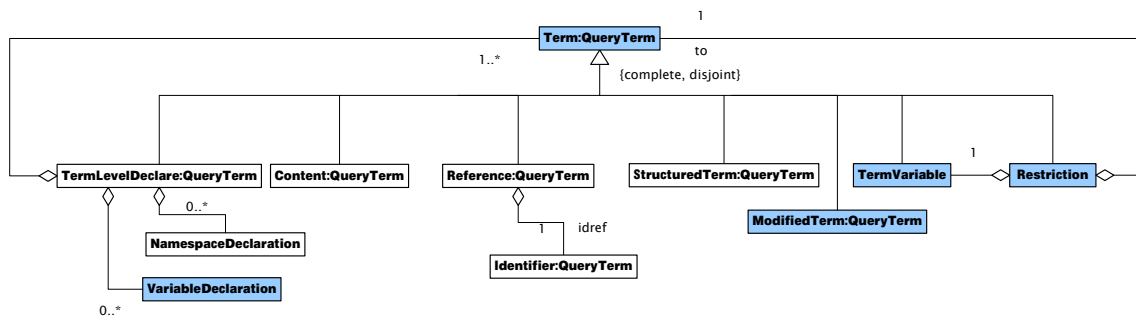


Figure 6.3: UML Model for (basic) Query Terms

Another means to restrict the bindings of a variable are **conditions**. Where variable restrictions restrict the terms a variable can be bound to, conditions restrict the *values*:

§3 Conditions

Conditions restrict the possible (*literal*) *values* a variable can be bound to and thus are mostly used to restrict literal variables, i.e., variables that occur in place of identifiers or literal content.

Conditions consist in arbitrary expressions (though currently, only arithmetic and comparison expressions on (floating point) numbers are defined, cf. Section 12.2.3) formed with identifiers or literal content as atoms. Also conditions may use boolean connectives to form condition formulas. A more detailed description of conditions (and functions in general) is under development, cf. Issue 7.

Rather than restricting the terms or literal values that a variable can be bound to, term variables may also be *bound to only parts of a matched sub-term*:

§4 Except Binding Modifier

A binding modifiers changes the binding a variable without affecting the match of a query. The only kind of binding modifier in Xcerpt is the **except binding modifier**. Using **except** a part of a sub-term can be omitted from the bindings.

The *scope* of an **except** modifier is a single (possibly modified basic or attribute, resp.) query term as seen in Figures 6.4 and 6.5 (cf. Issue ??).

Since **except** changes variable bindings it is only useful in *the scope of a variable restriction*. Occurrences outside of any variable restriction are ignored (cf. Issue ??).

Notice, that **except** does not change the original matched term, but only the variable bindings. It also does not affect the matching of a query term: The query term obtained by replacing all **except**'s in a query term by their in-scope query terms matches the same data or construct terms as the original one. The only differences is that some variable bindings might have certain parts removed.

Variables may be *declared* in term-level declare blocks just like namespaces.

§5 Variable Declarations

A **variable declaration** is, at the time of writing, only reserving a certain identifier for use as a variable in the scope of the declaration block.

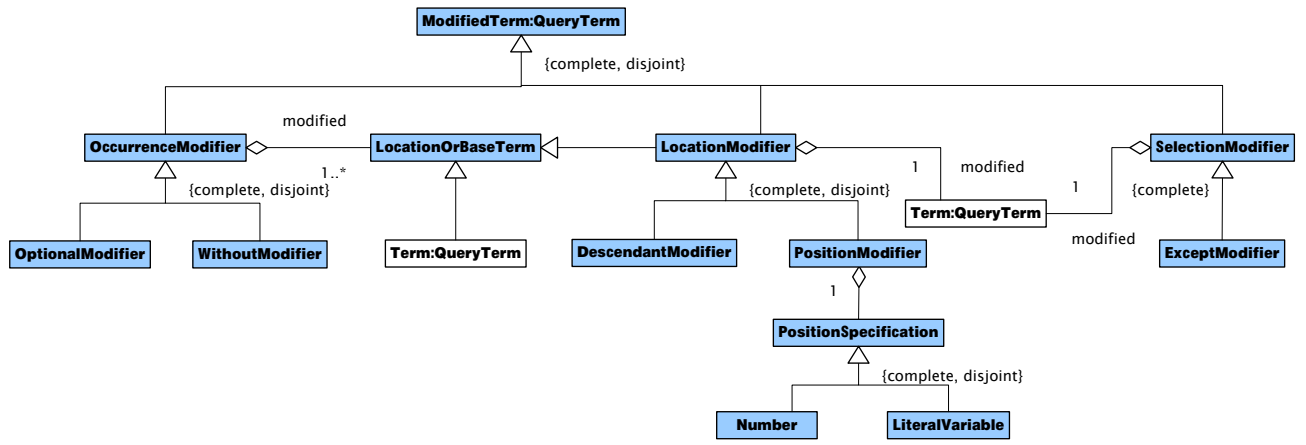


Figure 6.4: UML Model for Modified Structured Query Terms

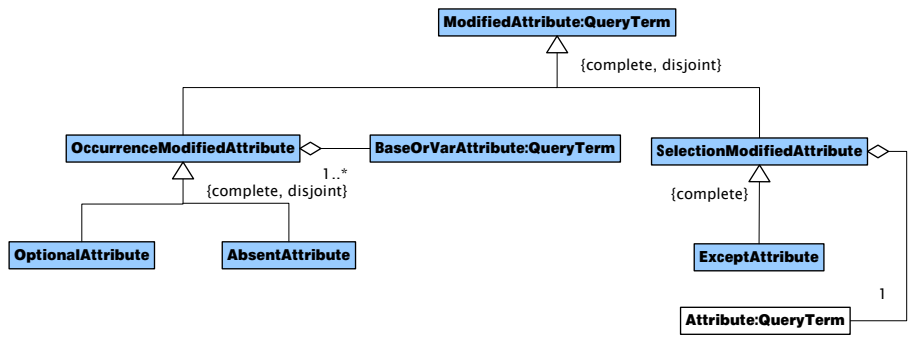


Figure 6.5: UML Model for Modified Attribute Query Terms

However, it is envisioned that in the same way variables might be typed, e.g., to restrict a certain variable to literal values or to structured values only (cf. Issue 12).

6.2.1 Textual Term Syntax

$\langle\text{condition-clause-qt}\rangle ::= \text{'where' ' (' } \langle\text{condition-qt}\rangle \text{')'}$

$\langle\text{condition-qt}\rangle ::= \begin{array}{l} \langle\text{c-parameter}\rangle - \langle\text{comparison-op}\rangle - \langle\text{c-parameter}\rangle \\ \langle\text{comparison-op}\rangle - \text{'('} - \langle\text{c-parameter}\rangle \text{' , ' } \langle\text{c-parameter}\rangle \text{' - ')} \\ \text{'and' - ' ('} - \langle\text{condition-qt}\rangle \text{' , ' } \langle\text{condition-qt}\rangle \text{' , ' } \langle\text{condition-qt}\rangle \text{' - ')} \\ \text{'or' - ' ('} - \langle\text{condition-qt}\rangle \text{' , ' } \langle\text{condition-qt}\rangle \text{' , ' } \langle\text{condition-qt}\rangle \text{' - ')} \\ \text{'not' - ' ('} - \langle\text{condition-qt}\rangle \text{' - ')} \\ \langle\text{c-parameter}\rangle \end{array}$

$\langle\text{condition-op}\rangle ::= \begin{array}{l} \text{'=='} \\ \text{'!='} \\ \text{'<'} \\ \text{'>'} \\ \text{'<='} \\ \text{'>='} \end{array}$

$\langle\text{arithmetic-op}\rangle ::= \begin{array}{l} \text{'+'} \\ \text{'-'} \\ \text{'*'} \\ \text{'/'} \\ \text{'^'}$

$\langle\text{c-parameter}\rangle ::= \begin{array}{l} \langle\text{optional-variable-qt}\rangle \\ \langle\text{variable-qt}\rangle \\ \langle\text{String}\rangle \\ \langle\text{Int}\rangle \\ \langle\text{c-parameter}\rangle - \langle\text{arithmetic-op}\rangle - \langle\text{c-parameter}\rangle \\ \langle\text{arithmetic-op}\rangle - \text{'('} - \langle\text{c-parameter}\rangle - \langle\text{c-parameter}\rangle \text{' - ')} \end{array}$

$\langle\text{optional-variable-qt}\rangle ::= \langle\text{optional-modifier}\rangle - \langle\text{variable-qt}\rangle \rightarrow$

$\langle\text{variable-decl-qt}\rangle ::= \begin{array}{l} \text{'variable' } \langle\text{NCName}\rangle \text{' , ' } \\ \text{'var' } \langle\text{NCName}\rangle \text{' , ' } \end{array}$

$\langle\text{variable-term-qt}\rangle ::= \begin{array}{l} \langle\text{base-term-qt}\rangle \\ \langle\text{term-variable-qt}\rangle - \text{'->'} - \langle\text{base-term-qt}\rangle \end{array}$

$\langle\text{variable-qt}\rangle ::= \begin{array}{l} \langle\text{term-variable-qt}\rangle \\ \langle\text{literal-variable-qt}\rangle \end{array}$

$\langle \text{variable-attr-term-qt} \rangle ::= \langle \text{term-variable} \rangle \text{ '-' } \langle \text{base-attr-term-qt} \rangle$

$\langle \text{term-variable-qt} \rangle ::= \langle \text{var-specification-qt} \rangle$

$\langle \text{literal-variable-qt} \rangle ::= \langle \text{var-specification-qt} \rangle$

$\langle \text{var-specification-qt} \rangle ::= \langle \text{'variable'} \rangle \langle \text{NCName} \rangle$
 $\quad \quad \quad \langle \text{'var'} \rangle$
 $\quad \quad \quad \langle \text{anonymous-variable} \rangle$

$\langle \text{anonymous-variable} \rangle ::= \text{'_'}$

6.2.2 XML-style Term Syntax

Once more, the XML-style term syntax uses productions identical to the ones for the non-XML term syntax. The full grammar is given in Appendix B.4.

6.2.3 Pure XML Syntax

The following parameterized grammar for formulas is used not only for defining condition clauses at term-level, but also for term formulas only occurring at top-level in query terms.

```

1 default namespace = "http://xcerpt.org/ns/core/1.0"
  namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"
3
4 start = formula
5 formula =
  element and { formula, formula+, condition? }
7 | element or { formula, formula+, condition? }
  | element not { formula }
9 | content
  condition = empty
11 content = empty

```

Condition clauses consist in such formulas, where the content can be comparisons and arithmetic expressions over variables.

```

1 # #2# CONDITION CLAUSES ##
  condition-clause =
3 element condition {
  grammar {
5 include "formula.rnc" {
  content = parent comparison*
7 }
  }
9 }
  comparison =
11 element comparison {
  attribute operator {
13 "eq" | "neq" | "lt" | "bt" | "elt" | "ebt"
  },

```



```

15     arithmetics,
16     arithmetics
17   }
18   | element optional { comparison }
19   | arithmetics
arithmetics =
21   element arithmetics {
22     attribute operator {
23       "plus" | "minus" | "times" | "div" | "power"
24     },
25     (arithmetics | arithmetic-parameter),
26     (arithmetics | arithmetic-parameter)
27   }
28   | element optional { arithmetics }
29   | arithmetic-parameter
arithmetic-parameter =
31   variable
32   | element value { text }

```

6.3 Incomplete Patterns

As discussed above, query terms are meant to be examples or patterns of the sought-for data. So far, however, query terms always have to specify all parts of matched data, even if only a small part is sufficient to distinguish relevant from irrelevant data. Obviously, this is unacceptable for many queries. Therefore, Xcerpt introduces a number of concepts to allow query terms to be *incomplete patterns* of the sought-for data, that may specify only what is needed to distinguish relevant from irrelevant data. In contrast to many other query languages (such as XQuery and SQL) that assume that a query specifies only parts of the sought-for data and make it difficult to specify queries where no additional data may occur, Xcerpt patterns make it *obvious* where a query term is incomplete and where not. This is a property that is particularly welcome in the context of semi-structured data as here the schema of the data is often unknown or variable, allowing, e.g., optional or repeated children.

Query terms (i.e., patterns) can be incomplete with respect to

1. **breadth**, i.e., only a subset of the actual children of a term is specified.

§6 Partial Terms

A term is called *partial*, if only some of the children of the term are specified. When matching partial terms, additional children may occur without affecting the match. However, all specified children must occur and, in case of ordered terms, must occur in the same order.

If a term is not *partial*, it is called *total* as discussed above. Notice, that neither data nor construct terms may contain partial terms, as data is always presumed to be “complete” (cf. Issue ??).

Partial terms are obviously essential for dealing with semi-structured data, where the schema of the data may allow for repetition or omissions of data (both in breadth and depth). Thus, queries can not specify total (or complete) patterns for the data.

However, partial terms also introduce a number of new challenges in a pattern language such as Xcerpt—that do not occur, e.g., in logic programming languages such as Prolog, where term arity and children order of two matching terms are always the same.

First, assume an *ordered* query term q that specifies only a partial list of children. Then the position (among its siblings) of a match m' for a child of m may (and will in most cases) differ from the position of m among its siblings (i.e., among q 's children). However, in many cases access to the (sibling) position is needed, e.g., to obtain the first child or immediate following sibling of a matched term. Therefore, Xcerpt provides access to the *sibling position* through the position modifier:

§7 Position Modifier

The **position modifier** allows the specification of the position of a child among its siblings, i.e., in the list of children of its parent. The *position* may be specified as an arithmetic expression composed of the usual arithmetic operators for natural numbers and natural numbers as well as (literal) variables as atomic expressions. The *scope* of a **position** modifier is a single term.

Obviously, a **position** modifier can only occur inside other (structured) terms and not at the top-level of a query term.

Using variables one can (a) query the position, (b) express positional joins, e.g., to find immediate siblings, and (c) correlate data and position. Variables occurring in position specifications may only be bound to natural numbers, otherwise a (run-time) exception occurs (cf. Issue 12). The following example shows the use of positional variables to find the immediate two following siblings of a term:

Example 6.1. immediate following two siblings

Second, though one may not be able to specify how all of the children of a sought-for term ought to be shaped like, one might be able to specify how they ought *not* to be shaped. Again, this makes sense only in a partial term, as in a total term the shape of all children must be explicitly stated. Xcerpt uses the **without** modifier to express this *subterm negation*:

§8 Without Modifier

The **without modifier** expresses that *not all sub-terms in its scope* may match with a sub-term of a match for its parent term. The scope of a **without** modifier is a list of one or more sub-terms.

Sub-terms in the scope of a **without** modifier are often referred to as *negative*, those outside as *positive*.

Notice, that the definition does not specify that *none* of the sub-terms may occur in a match for the parent: If the scope of a **without** modifier is a list of sub-terms, then the parent term t matches another term t' , if there is a mapping of the positive sub-terms of t to t' , that can not be extended to also cover all negative sub-terms. Thus, subterm negation is *existential*.

Example 6.2. examples with a list and a single without

Like **position** modifiers, **without** modifiers may only occur in sub-terms. Note, also that Xcerpt places some limitations on variables used in **without** modifiers, cf. Section 7.2.

Currently, **without** modifiers may not be either arbitrary siblings of each other in an unordered parent term or immediate siblings of each other in an ordered parent term (cf. Issue 5).

Figure 6.6: UML Model for Qualified Descendant

2. **depth**: Semi-structured data may not only vary in the number, order, and repetition of children, but also in how elements are nested. E.g., in (X)HTML most inline elements such as `em` may occur in most block-level elements such as `p` or `div`, but may also be nested inside each other. Thus selecting all `em` elements in an (X)HTML document in a pattern requires a means to specify patterns that are incomplete in depth, i.e., that contain sub-terms that are not direct sub-terms of their parent but stand in another structural relation to it, e.g., occurring at any depth under their parent or occurring at depth 5 under their parent.

To express such incompleteness in depth Xcerpt provides the **descendant** modifier, similar in its basic form to the **descendant** axis in XPath. In contrast to XPath (and thus XSLT and XQuery), Xcerpt also provides a more expressive variant of the **descendant** modifier that allows direct expression of constraints such as “occurs at depth 5 under its parent” or “occurs at any depth under its parent but with only `div` elements in between its parent and itself”. The latter variant of the **descendant** modifier is referred to as *qualified*, the basic case as *unqualified*.

§9 (Unqualified) Descendant Modifier

The unqualified variant of the **descendant modifier** specifies that the *single sub-term* in its scope may occur at any depth under the parent term (rather than as an immediate child).

Notice, that the **position** and **descendant** modifier can not be mixed, as the former to the position among the immediate children of the parent term, and the latter specifies that the sub-term may also be nested more deeply inside the parent term.

In contrast to the **position** modifier the **descendant** modifier may occur at top-level, thus specifying that the contained term may occur at any level in the document.

§10 Qualified Descendant Modifier

The qualified variant of the **descendant modifier** specifies a more involved relation between the parent term and the single in-scope sub-term: The in-scope sub-term occurs inside the parent term, but the path in between is restricted by a qualifying expression that consists in a selection and a repetition part.

A detail model of the qualified descendant is given in Figure 6.6: The *selection* part is a sequence of (one or more) element label and optional attribute term specifications, both possibly containing variables. The *repetition* part is an interval $[i, j]$ with $i \leq j$ and $i, j \in \mathbb{N}_0 \cup \{\infty\}$. The interval boundaries may also be literal variables.

Thus, the qualified descendant restricts the in-scope sub-term to matchings that are reached from a match of the parent term via a path that matches the selection part repeated between i and j times.

Notice, that variable occurrences in the selection and repetition part of a qualified descendant are non-binding, i.e., all such variables must be bound in another part of the query term (cf. Section 7.2).

Example 6.3.

3. **optional parts:** One of the most distinguishing features of semi-structured data in contrast to, e.g., relational data aside is the allowance for optional information, i.e., information that occurs in some elements of a certain type but is missing in others of the same type. Though *testing* for the existence or absence of such optional information has been a focus in many semi-structured and XML query languages (most notably structural predicates in XPath), *selecting* of or *construction* based on optional information has been far less closely investigated. Xcerpt provides query authors with a unified concept for handling optional information in the context of testing, selection, and construction, quite in contrast to mainstream XML query languages such as XQuery and XSLT.

Just like in construct terms, the optional modifier is used in query terms to indicate which parts of a query may be missing without affecting the matching of the remainder of the query.

§11 Optional Modifier

The **optional modifier** in query terms indicates that its in-scope terms may be missing in a matching term, but have to be considered, if they are existent. The scope of an optional modifier is a list of terms.

This way **optional** modifiers serve to bind variables to part of the data that may be absent, but that must be included in the substitutions resulting from a query, if present. Obviously, only bindings for variables that do not occur (positively, cf. Section 7.2) also outside of the scope of any **optional** are effect by the presence or absence of the optional part, as in the other case their bindings are already established by the outside occurrence. Therefore, an **optional** modifier with no such variable in scope does not affect either matching or the resulting substitutions and can thus be safely removed from the query.

4. **order:** As discussed above, data and construct terms may already be distinguished in ordered and unordered terms. Often, however, one might not care about the order in which matches for the sub-terms in a query occur in the data, even if the data itself is ordered. Xcerpt acknowledges this fact by allowing query terms that are unordered to match with ordered terms, but not the other way around. I.e., if the query specifies the order is significant then only data where the order is significant as well can match with that query; if the query however indicates that the order may be ignored, then also data is considered that is ordered, however the sub-terms of the query are matched in any order with the sub-terms of the data.
5. **literal specification:** Finally, like in the relational case, queries often may not be able to specify literal content or identifiers completely, but rather query for data where the literal content or the identifiers falls into some class, specified in Xcerpt by means of POSIX.1 regular expressions enhanced with variable bindings: additionally to using POSIX's numeric backreferences, Xcerpt allows subexpressions to be bound to Xcerpt literal variables. This allows the extraction and insertion of data from the rest of the Xcerpt query into the regular expression.

As stated, regular expressions may occur anywhere in a query term where literal content or identifiers may occur, except where only natural numbers are allowed, as in repetition and position specifications.

Notice, that for the “wildcard” regular expression `.*` anonymous variables may be used and are often more convenient.

6.3.1 Textual Term Syntax

$\langle \text{selection-modified-qt} \rangle ::= \text{selection-modifier} - ' (\langle \text{modified-qt} \rangle , ' \rightarrow$

$\langle \text{selection-modified-attr-term-qt} \rangle ::= \text{selection-modifier} - ' (\langle \text{modified-attr-term-qt} \rangle , ' \rightarrow$

$\langle \text{selection-modifier} \rangle ::= \text{'except'} \rightarrow$

$\langle \text{occurrence-modified-qt} \rangle ::= \text{occurrence-modifier} - ' (\langle \text{modified-qt} \rangle , ' \rightarrow$

$\langle \text{occurrence-modified-attr-term-qt} \rangle ::= \text{occurrence-modifier} - ' (\langle \text{modified-attr-term-qt} \rangle , ' \dots$

$\langle \text{occurrence-modifier} \rangle ::= \text{optional-modifier} \text{ or } \text{'without'} \rightarrow$

$\langle \text{location-modified-qt} \rangle ::= \text{location-modifier} - ' (\langle \text{term-variable-qt} \rangle , ' \rightarrow$

$\langle \text{location-modifier} \rangle ::= \text{descendant-modifier} \text{ or } \text{position-modifier} \rightarrow$

$\langle \text{descendant-modifier} \rangle ::= \text{'descendant'} \text{ or } \text{'desc'} \rightarrow$

$\langle \text{position-modifier} \rangle ::= \text{'position'} \text{ or } \text{'pos'} \langle \text{number-qt} \rangle \rightarrow$

$\langle \text{number-qt} \rangle ::= \langle \text{Int} \rangle \text{ or } \langle \text{literal-variable-ct} \rangle \rightarrow$

6.3.2 XML-style Term Syntax

Once more, the XML-style term syntax uses productions identical to the ones for the non-XML term syntax. The full grammar is given in Appendix B.4.

6.3.3 Pure XML Syntax

Like in the term syntax, the XML syntax enforces a hierarchy of modified terms with occurrence and selection modified terms at the top followed by occurrence modified terms, variable restrictions and finally base terms.

```
# #3# MODIFIED TERMS ##
2 modified-term =
  variable-term | location-term | occurrence-term | selection-term
4 base-term = reference | content-term | structured-term
variable-term =
6   base-term
  | variable
8   | element restriction { variable, base-term }
location-term =
10  element descendant { variable-term }
  | element position {
12    element number { variable | xsd:int },
  variable-term
14  }
selection-term = element except { modified-term }
16 occurrence-term =
  element without { modified-term }
18  | element optional { modified-term }
# #4# MODIFIED ATTRIBUTE TERMS ##
20 modified-attr-term =
  base-attribute,
22  variable-attr-term,
  occurrence-modified-attr-term,
24  selection-modified-attr-term
variable-attr-term =
26  variable
  | element restriction { variable, base-attribute }
28 occurrence-modified-attr-term =
  element without { modified-attr-term }
30  | element optional { modified-attr-term }
selection-modified-attr-term = element except { modified-attr-term }
```

6.4 Top-level Query Terms

As mentioned in the discussion of some of the query term modifiers above, certain modifiers are only allowed at sub-term level, but not at the top-level. On the other hand, there are some constructs that may only occur in top-level query terms, viz. term formulas and document specifications. Figure 6.7 shows a detailed model of top-level query terms. Notice that from all modifiers in query terms only **optional** and **descendant** modifiers are allowed at top-level.

6.4.1 Term Formulas

(Top-level) query terms can be connected by the usual boolean connectives to form so-called query *term formulas*.

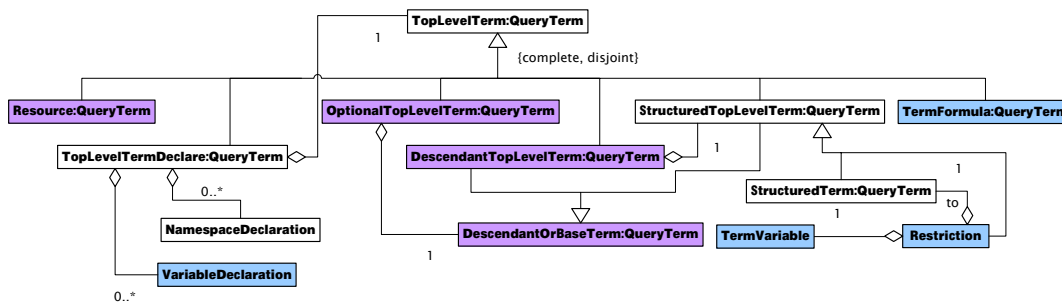


Figure 6.7: UML Model for Top-level Query Terms

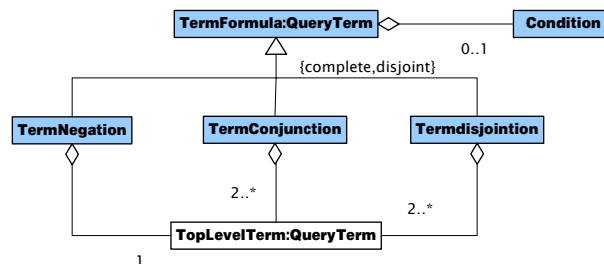


Figure 6.8: UML Model for Query Formula

§12 Term Formulas

A **term formula** is an expression formed from boolean connectives (**and**, **or**, and **not**) over top-level query terms. Intuitively, **or** merely merges the resulting multi-sets of substitutions resulting from the contained queries (similar to **union** in SQL database systems), whereas **and** creates the joins the individual substitutions (if none of the queries contains a negation).

Besides the sub-term negation introduced in Section 6.3 above (**without** modifier), Xcerpt also supports *query negation*, denoted using **not**. The query negation used in Xcerpt is negation as (finite or infinite) failure like in logic programming, i.e., a negated query **not**(*Q*) succeeds if the query *Q* fails. Like in negated sub-terms, variables occurring in a negated query do not yield bindings, i.e., they have to appear elsewhere in the query outside the scope of a negation construct (cf. Section 7.2).

Notice, that query negation is universal quantified, i.e., there may be no term that matches the query, whereas sub-term negation filters out those parent terms that contain the negated sub-term, and thus is effectively existential quantified if the parent term is not bound to a variable.

6.4.2 Document Specifications

So far, query terms have not specified what document the data they are matched against comes from. In this case, data and construct terms that are part of the program (or set-up by some other environment specific method) are considered. If data stored in external sources is to be accessed, a *document specification* is needed to specify the needed information about that external data source.

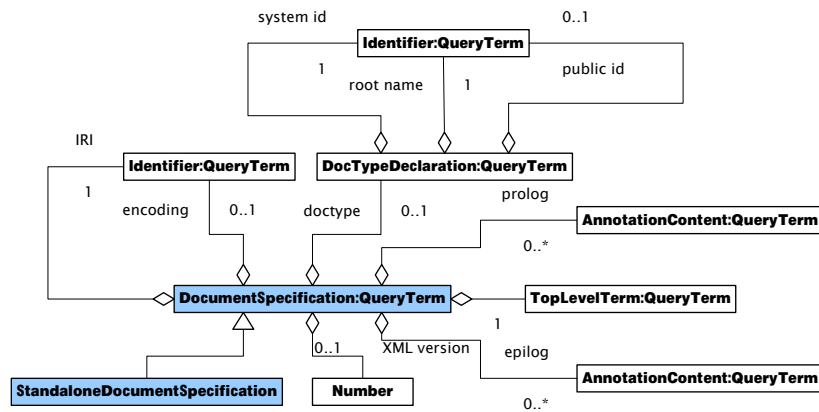


Figure 6.9: UML Model for Document Specifications

§13 Document Specification

A **document specification** describes an external data source such as an XML document. Typically document specifications contain at least *access parameters*, e.g., the IRI (Internationalized Resource Identifier [27]) of an XML document.

At the time of writing, the only form of document specification are XML document specifications.

§14 XML Document Specification

An **XML document specification** is a document specification to specify access to XML documents. Aside of an IRI identifying the document to be accessed, an XML document specification may contain most of the information present in the document and document type declaration information items from [21]: XML version, standalone status flag, root name (i.e., the tag name of the document element), system, and public identifier of the document type declaration if any, as well as the document element (a top-level term) and two lists of annotation content (i.e., processing instruction or comment terms) for document prolog and epilog.

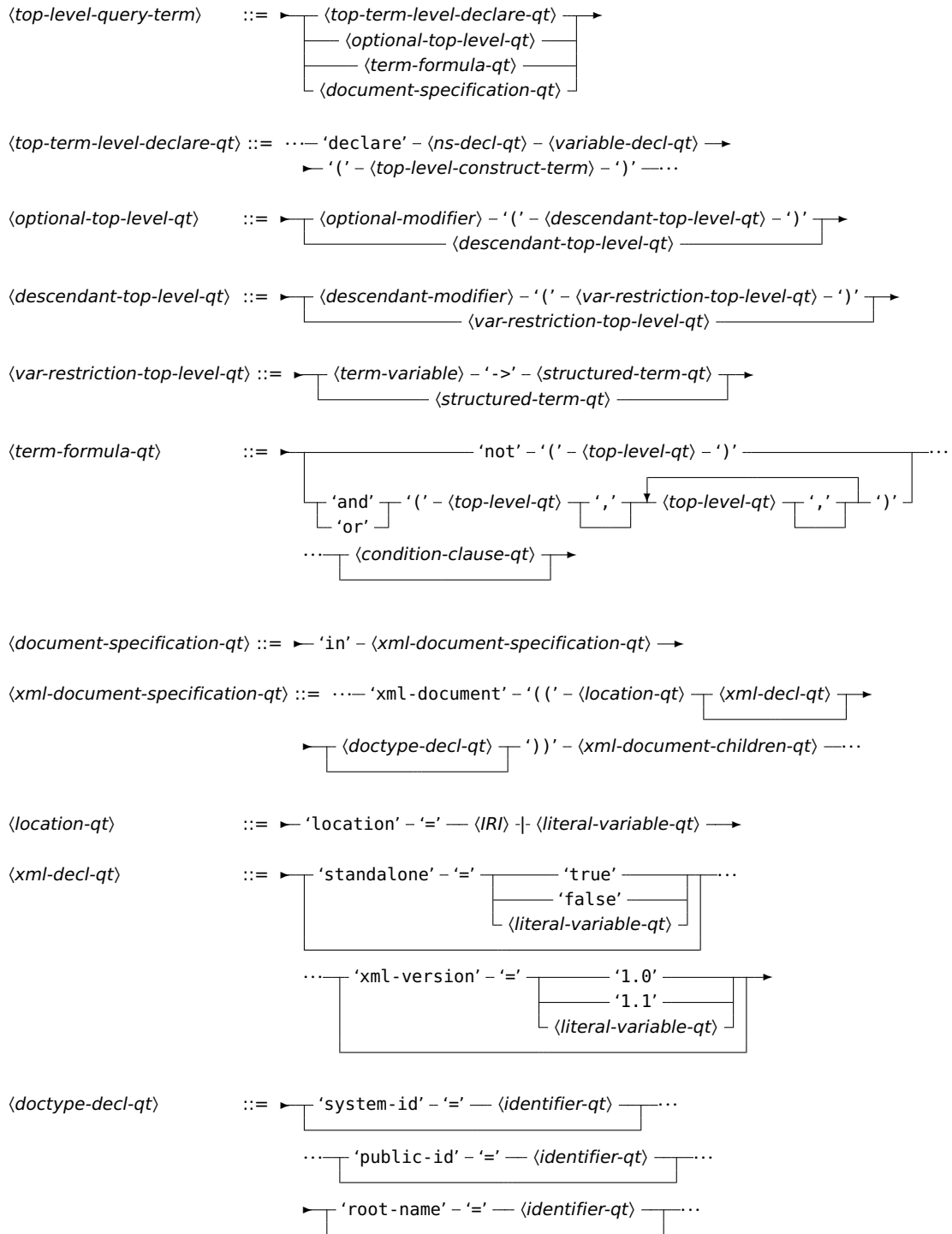
Note, that there is currently no mechanism in Xcerpt to enable or disable validation in presence of a document type declaration. In absence of such a mechanism, Xcerpt implementations are expected to validate all documents with document type declaration. If such a document is not valid, an error is generated. Thus, if a root name is present, it will always be the same as the label of the document element.

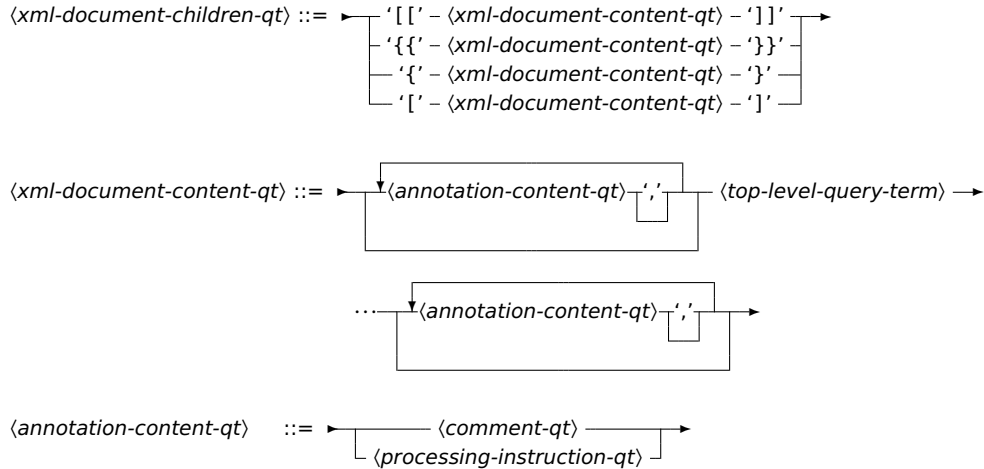
For querying, document specifications are treated just like any other form of data, i.e., one can, e.g., query all documents conforming to a DTD identified by a specific public or system ID.

Document specifications are also used for specifying properties of output documents, cf. Section 7.

6.4.3 Textual Term Syntax

As noted, top-level query terms may, in addition to structured query terms and top-level declare blocks as in data and construct terms, may also be modified through **optional**, **descendant**, or a variable restriction and may be formulas of query terms as well as document specifications, i.e., expression for accessing specific resources on the Web.





6.4.4 XML-style Term Syntax

Once more, the XML-style term syntax uses productions identical to the ones for the non-XML term syntax. The full grammar is given in Appendix B.4.

6.4.5 Pure XML Syntax

Top-level query terms differ from general top-level terms in the possible use of a **optional** or **descendant** modifier, the possible use of a variable restriction and, most notably, in the addition of term formulas and document specifications, both occurring exclusively at the top-level of query terms. Notice the reference to the generic grammar for formulas using $\langle optional\text{-}top\text{-}level\text{-}term \rangle$ as basic content type.

```

# #1# TOP-LEVEL QUERY TERM ##
2 optional-top-level-term =
  element optional { descendant-top-level-term }
4   | descendant-top-level-term
descendant-top-level-term =
6   element descendant { var-restriction-top-level-term }
   | var-restriction-top-level-term
8 var-restriction-top-level-term =
  element restriction { variable, structured-term }
10  | structured-term
term-formula =
12  grammar {
    include "formula.rnc" {
14     content = parent optional-top-level-term
    condition = parent condition-clause
16   }
  }
18 document-specification =
  element xml-document {
20   attribute location { text },
  element xml-declaration {

```

```

22     attribute standalone { "true" | "false" | variable }?,
    attribute xml-version { "1.0" | "1.1" | variable }?
24 }?,
    element doctype {
26     attribute system-id { identifier.class }?,
    attribute public-id { identifier.class }?,
28     attribute root-name { identifier.class }?
    }?,
30 element children {
    annotation-content*, top-level-term.class, annotation-content*
32 }
}

```

6.5 Summary: Modifiers and Where they Occur

To address the needs of querying and construction, Xcerpt provides quite a number of modifiers that affect the way their in-scope terms are handled.

Most of these modifiers may occur either in construct or in query terms. The single exception from this rule is the **optional** modifier: it marks both parts of construct terms that may or may not occur depending on the result of the query term and parts of query terms that may or may not yield bindings depending on the data the query term is matched against.

Table 6.1 summarizes the eight modifiers (and three boolean connectives for term formulas) and gives at a glance which modifier may occur where and with what scope.

As discussed above, the following additional constraints hold:

1. In construct terms grouping and occurrence (**optional**) modifiers may be arbitrarily mixed. Grouping modifiers on attributes are limited to a single term as scope to avoid the repetition of attributes with the same name.
2. In query terms location modifiers “stick closely to the term modified by them”, i.e., occurrence and selection modifiers may contain location modifiers, but not vice versa. Therefore, location modifiers also always affect only a single term.
3. Finally, for top-level query terms only **optional** and **descendant** are allowed (the latter may occur inside the former), each with a single in-scope term.

	Data Terms		Construct Terms		Query Terms		
			Subterms	Attributes	Subterms	Attributes	Top-level
Grouping Modifiers							
all (all bindings)	—	*	•	—	—	—	—
some (some m bindings)	—	*	•	—	—	—	—
first ($n^{\text{th}} - m^{\text{th}}$ bindings)	—	*	•	—	—	—	—
Selection Modifiers							
except (omit from binding)	—	—	—	*	*	—	—
Occurrence Modifiers							
optional (may occur)	—	*	*	*	*	•	—
without (must <i>not</i> occur)	—	—	—	*	*	—	—
Location Modifiers							
descendant (at any depth)	—	—	—	•	—	•	—
position (as n^{th} child)	—	—	—	•	—	—	—
Term formulas							
and, or	—	—	—	—	—	—	2..*
not	—	—	—	—	—	—	•

Table 6.1: Occurrence of Modifiers (* indicates that the modifier has a scope of one to many terms, • exactly one term; — indicates that the modifier may not occur in that context)

Chapter 7

Programming in Xcerpt: Programs, Goals, and Rules

7.1 Xcerpt Programs

§1 Program

An Xcerpt program consists of at least one *goal* and some (possibly zero) *construct-query rules*.

Figure 7.1 shows an UML model for Xcerpt programs. For convenience, some of the input data of a program may be specified as part of the program using *data blocks* (similar to facts from logic programming, i.e., rules with an always successful query part).

§2 Goal

An Xcerpt goal specifies output of an Xcerpt program including an optional specification of where the output is to be stored.

Implementations must define default behavior for goals without output specification. More refined specifications for (XML and other) document properties as part of output specifications are under investigation, cf. Issue 16.

A program may contain multiple goals, allowing result to be stored in different files or at different Web locations. Notice, that the order of goals is currently undefined, cf. Issue 10, thus multiple goals with the same output target should be avoided for the time being.

§3 Rule

An Xcerpt *construct-query rule* (short: *rule*) relates a construct term to a top-level query term.

Rules can be seen as “views” specifying how to obtain documents shaped in the form of the construct term by evaluating the query against Web resources (e.g. an XML document or a database).

Recursive chaining of rules is possible (but note certain restrictions on recursion, cf. Section 7.2). In contrast to the inherent structural recursion used e.g. in XSLT, which is essentially limited to the tree structure of the input document, recursion in Xcerpt is always explicit and free in the sense that any kind of recursion can be implemented. Applications of recursion on the Web are manifold:

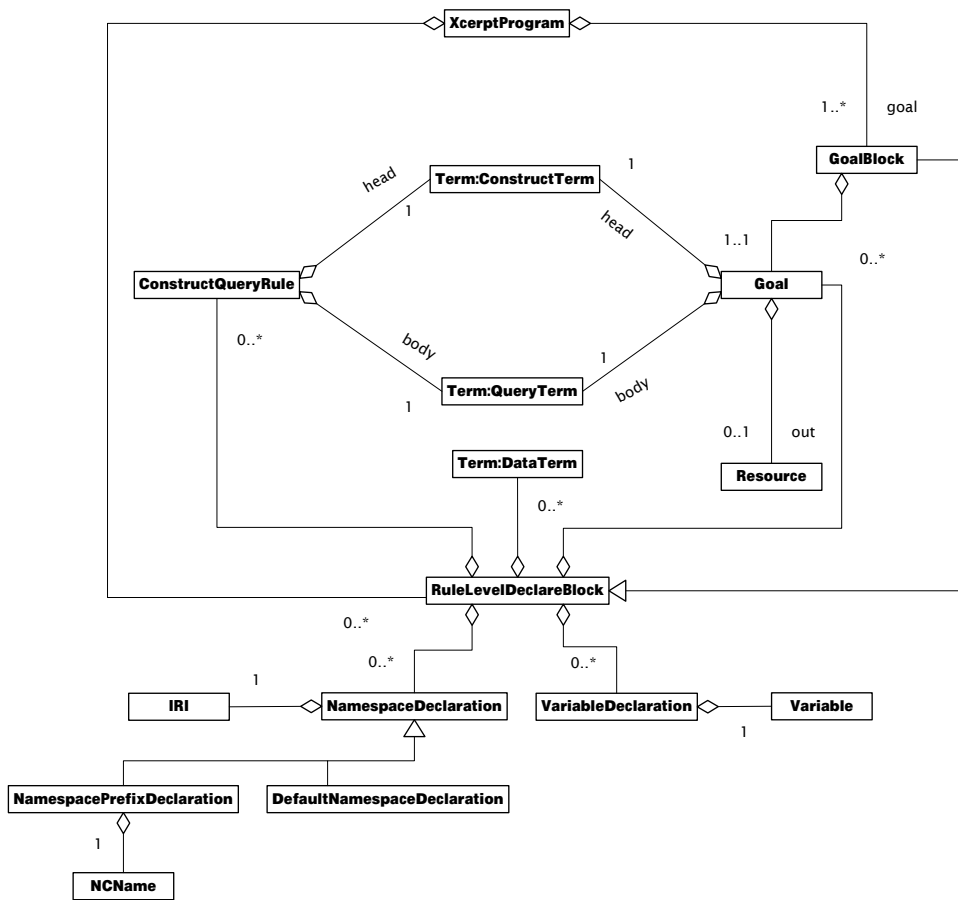


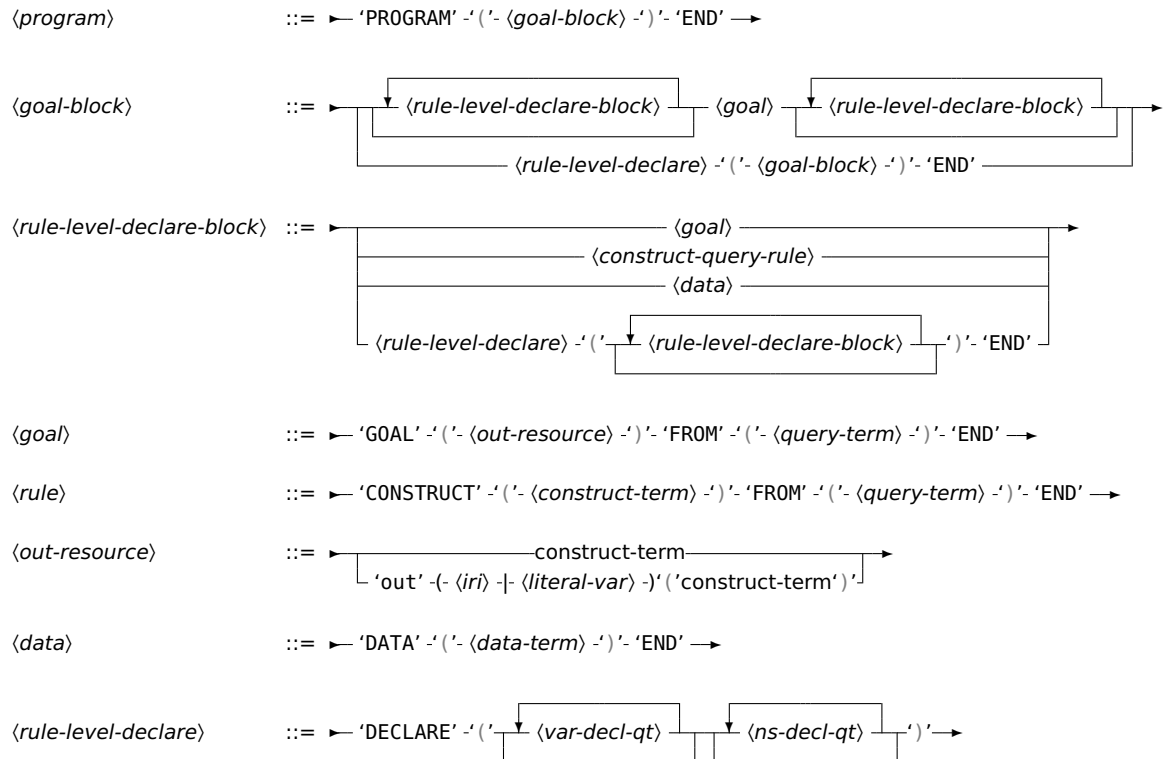
Figure 7.1: UML Model for Xcerpt Programs

- 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 `em` elements in HTML documents by `strong` 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 Web crawlers that recursively visit Web pages.

In addition to the syntactic constraints discussed so far, semantic constraints are imposed on the variables in rules, cf. Section 7.2.

7.1.1 Textual Term Syntax

The textual syntax for rules deviates from the term syntax by using uppercase keywords for block structures. This provides an easier visual distinction of rule- and term-level constructs.



7.1.2 XML-style Term Syntax

The XML-style term syntax follows the same grammar as the non-XML term syntax. The full grammar is given in Section B.5.

7.1.3 Pure XML Syntax

In the pure XML syntax programs, goals, rules, and data blocks are very much straight forward and make extensive use of Relax NG's parameterizable grammars for declaration blocks. Figure 7.2 gives a graphical representation of the following Relax NG grammar.

```
1  ## An Xcerpt program is a set of (one or more) goals, as well as (any number of) rules and inline data
   terms (like facts in Prolog). Rules and data terms may be surrounded by declaration blocks.
   program = element program { goal-block }
3  goal-block =
   rule-level-block*
5  | goal
   | rule-level-block*
7  | grammar {
   include "declare-block.rnc" {
9     content = parent goal-block
   }
11 }

13 ## Rule-level blocks form the basic block structure of an Xcerpt programs: goals, rules, and inline data
   terms form the basic block structures. They can be included into declaration blocks that define the
   scope of variable and namespace declarations.
   rule-level-block =
15  goal
   | rule
17  | data
   |
19  ## A declaration block on rule level allows both variable and namespace declarations.
   grammar {
21     include "declare-block.rnc" {
   content = parent rule-level-block*
23     }
   }
25

   ## A rule specifies how from data matched by the query term new data is constructed according to a
   construct term.
27  rule =
   element rule {
29     element construct { construct-term },
   element from { query-term }
31  }

33 ## A goal is a rule, where the resulting data is written to a specified resource. Hence, goals are not
   chained.
   goal =
35  element goal {
   element out {
37     (variable-ct
   | attribute value {
39     text
   >> a:documentation [
```

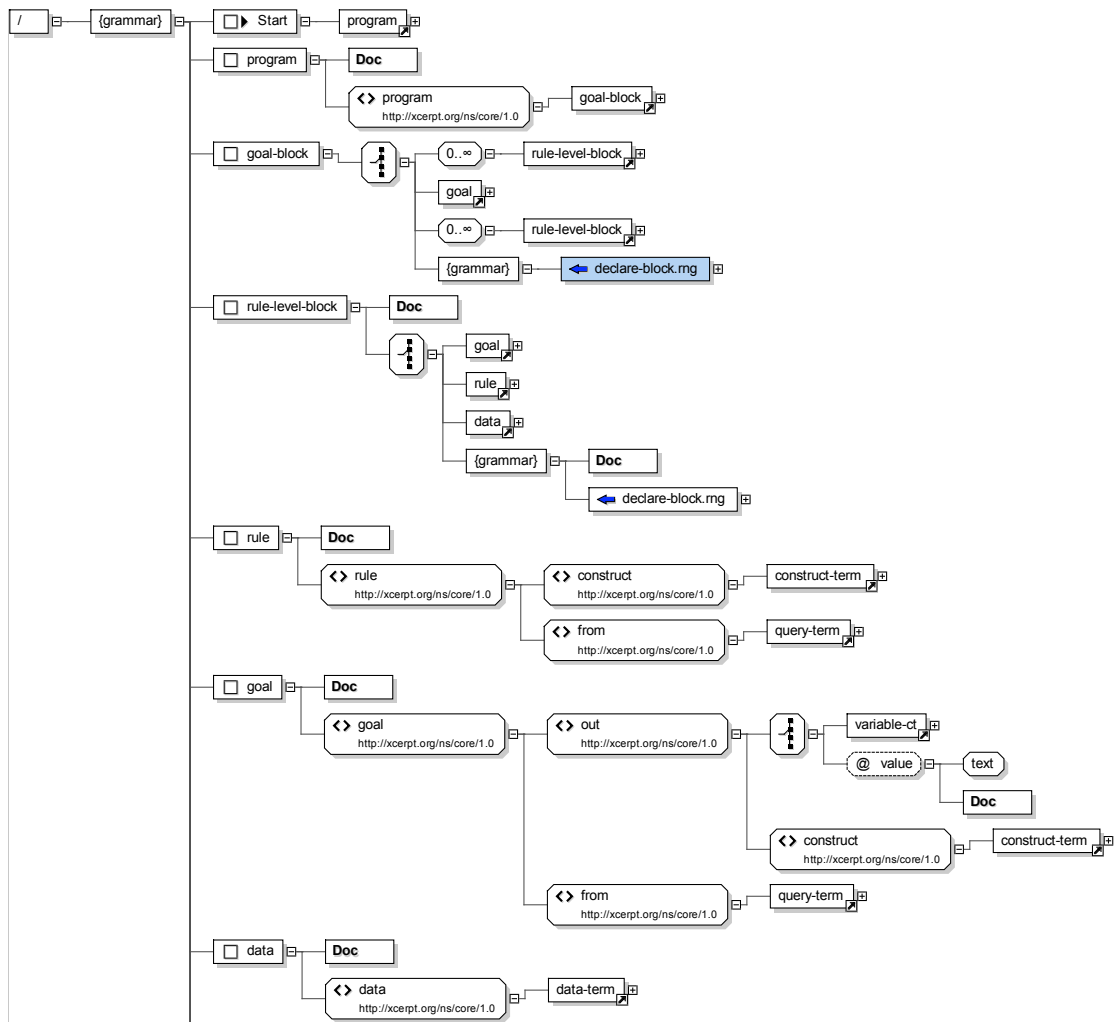



Figure 7.2: Relax NG Grammar for Xcerpt Programs

```

41         "This should in–fact be a IRI as by RFC 3987. Since XML Schema datatypes only provides the
           anyURI datatype for URIs conforming the older RFC 2396, arbitrary text is allowed."
           ]
43     }),
     element construct { construct-term }
45 },
     element from { query-term }
47 }

49 ## An inline data term allows the specification of data terms inside Xcerpt programs similar to facts in
     Prolog.
     data = element data { data-term }

```

7.2 Semantic Restrictions on Xcerpt Programs

Xcerpt imposes two major semantic restrictions on valid programs: range restrictedness and negation/grouping stratification. Where range restrictedness is a “local” property of a rule, negation and grouping stratification is a (global) property of an entire program.

Intuitively, range restrictedness ensures that all variables used in the construction part of a rule are properly bound in the query part. Negation and grouping stratification, on the other hand, disallow programs with a recursion over negated queries or grouping constructs, thus allowing an easier declarative semantics for Xcerpt programs.

optional or except variable only inside optional in head ERR optional should contain at least one variable WAR

or where the arity of the variable sets of the parts is not the same

7.2.1 Range Restrictedness

Intuitively, range restrictedness means that a variable occurring in a rule head also must occur at least once in the rule body. This requirement simplifies the definition of the formal semantics of Xcerpt, as it allows to assume that all query terms are unified with data terms instead of construct terms (i.e., variable free and collection free terms). Without this restriction, it is necessary to consider undefined or infinite sets of variable bindings, which would be a difficult obstacle for a forward chaining evaluation. Besides this formal reason, range restricted programs are also usually more intuitive, as they disallow variables in the head that are not justified somewhere in the body.

The following sections give a formal, syntactic criterion for range restrictedness, which considers negated queries and optional subterms as well as disjunctions in rule bodies.

7.2.1.1 Polarity of Subterms

So as to determine whether a rule is range restricted, variable occurrences in query and construct terms are associated with the polarities *positive* (+) or *negative* (–), and the attributes *optional* (?) or *not optional* (!) for such variables that are contained within an optional subtree and thus are not bound in all valid matchings. Intuitively, a *negative* variable occurrence is a *defining* occurrence, whereas a *positive* variable occurrence is a *consuming* occurrence. Since most terms are considered to be not optional, the attribute ! is omitted in most examples.

The polarity of variable occurrences in a term can be determined by recursively attributing all subterms of a term.

Polarity of Subterms

1. Let t be a query term with polarity p and optionality o .
 - if t is of the form **without** t' , then t' is of polarity $+$ (regardless of p) and optionality o
 - if t is of the form **optional** t' , then t' is of polarity p and optionality $?$.
 - if t is of the form $\langle modifier \rangle t'$, where $\langle modifier \rangle$ is not an occurrence modifier (i.e., neither **optional** nor **without**), then t' is of polarity p and optionality o (*unchanged*).
 - if t is of the form $var X \rightarrow t'$ then t' is of polarity p and optionality o (*unchanged*).
 - if t is of one of the forms $l\{t'_1, \dots, t'_n\}$, $l\{t'_1, \dots, t'_n\}$, $l[[t'_1, \dots, t'_n]]$ or $l[t'_1, \dots, t'_n]$ ($n \geq 0$), then t'_1, \dots, t'_n are of polarity p and optionality o (*unchanged*).
 - if t is of the form t' **where** c with c a condition formula, then t' is of polarity p and optionality o , c is of polarity $+$ and optionality o (*conditions are always consuming*).
 - if c is a condition formula of the form **optionl** c' with c' a condition formula, then c' is of polarity p and optionality $?$.

The rules for attribute terms are analogously, i.e., **optional** forces the optionality to $?$, **without** the polarity to $+$, all other modifiers leave them unchanged.

2. Let t be a construct or data term with polarity p and optionality o .
 - if t is of the form **optional** t' , then t' is of polarity p and optionality $?$.
 - if t is of the form **optional** t' **with-default** t'' , then t' is of polarity p and optionality $?$ and t'' is of polarity p and optionality o .
 - if t is of one of the forms $f\{t'_1, \dots, t'_n\}$ or $f[t'_1, \dots, t'_n]$ ($n \geq 0$), then t'_1, \dots, t'_n are of polarity p and optionality o .
 - if t is of the forms **all** t' or **some** t' , then t' is of polarity p and optionality o .
 - if t is of the form $op(t'_1, \dots, t'_n)$, with op a function or aggregation identifier, then t'_1, \dots, t'_n are of polarity p and optionality o .

The root of a query term is usually of negative polarity (and thus define variable bindings), as query terms usually occur in rule bodies. The root of a construct or data term is usually of positive polarity.

In a rule, the construct term in the head always has positive polarity and the query part has negative polarity and both are, by default, not optional. If negation constructs occur, the polarity changes accordingly. Furthermore, if parts of a query are negated by **not**, the polarity of these parts is again positive:

Polarity in Rules

1. If $R = t^c \leftarrow Q$ is a rule or goal with t^c a construct term and Q a query part, then the polarity of t^c is $+$ and the polarity of Q is $-$.
2. Let Q be a query part with polarity p .

- if Q is of the form *not* Q' , then Q' is of polarity + (regardless of p)
- if Q is of the forms *and*(Q_1, \dots, Q_n) or *or*(Q_1, \dots, Q_n), then Q_1, \dots, Q_n are of polarity p
- if Q is a document specification with content C , then all variables occurring in the document specification are of polarity + and C is of polarity p .
- if Q is of the form t (a query term), then t is of polarity p .

Note that the polarity of negated subterms and queries is *always* positive, regardless of the level of nesting. The rationale behind this is that, since negation in Xcerpt is *negation as failure* and not the negation of classic logic, additional negations do not completely revert previous negations. Variable occurrences that are in the scope of at least one negation construct are always consuming occurrences, since negation as failure requires to perform auxiliary computations.

Returning to the definition of range restrictedness, it requires that in a rule, for each consuming occurrence of a variable, there exists at least one defining occurrence. Furthermore, a variable for which all defining occurrences are optional also needs to be optional on all consuming occurrences. This restriction is straightforward to understand, as it just requires that “*each variable in the head or in a negated query needs to be bound elsewhere*”.

This intuitive definition of range restrictedness is complicated by the possibility of disjunctions in the rule body, in which case a variable occurring positively in the rule head needs to occur negatively in *each* disjunct. Since disjunctions can also be nested, it is useful to define a *disjunctive rule normal form*, cf. [50].

Given a rule in such a disjunctive rule normal form, range restrictedness requires that each variable that occurs positively in one of the disjuncts occurs also negatively in the same disjuncts. Range restrictedness is formalised by the following definition:

Range Restrictedness Let R be a rule or goal and let $R' = t^c \leftarrow Q_1 \vee \dots \vee Q_n$ ($n \geq 0$) be the disjunctive rule normal form of R . R is said to be *range restricted*, iff

1. for each disjunct Q_i ($1 \leq i \leq n$) holds that each variable occurring with positive polarity in either t^c or Q_i also occurs at least once with negative polarity in Q_i .
2. each variable that occurs in at least one of the Q_i ($1 \leq i \leq n$) attributed as *optional* without another non-optional, negative occurrence in Q_i ¹ or that occurs in Q_i , but not in Q_j for some $1 \leq i, j \leq n$ is attributed as optional in all positive occurrences in Q_i (including condition formulas and document specifications) and t^c .

A program P is called *range restricted*, if all rules $R \in P$ are range restricted.

7.2.2 Negation and Grouping Stratification

Stratification is a technique first proposed by Apt, Blair, and Walker [3] to define a class of logic programs where non-monotonic features like Xcerpt’s grouping constructs or negation can be defined in a declarative manner. The principal idea of stratification is to disallow programs with a recursion over negated queries or grouping constructs and thereby precluding undesirable programs. While this requirement is very strict, its advantages are that it is straightforward to understand and can be verified by purely syntactical means without considering terms that are not part of the program.

¹Notice, that the optional occurrence must be of negative polarity in this case, as the query term would otherwise contradict rule (1).

Several refinements over stratification have been proposed, e.g. *local stratification* [48] that allow certain kinds of recursion, but these usually require more “knowledge” of the program or the queried resources.

Xcerpt programs in this thesis are considered to be stratifiable². Furthermore, the notion of stratification is not only used for proper treatment of negation, it also extends to rules with grouping constructs, because a recursion over grouping constructs usually defines undesirable behaviour. A detailed discussion of stratification in Xcerpt can be found in [50, 49].

Here, we only give the final definition of a fully stratified Xcerpt program:

Full Stratification of an Xcerpt Program Let $P = P_1 \uplus \dots \uplus P_n$ denote a *partitioning* of a set P into disjoint sets P_i . Given a program P consisting of rules/goals $\{R_1, \dots, R_m\}$ ($m \geq 1$).

1. Let $R = t^c \leftarrow Q$ and $R' = t^{c'} \leftarrow Q'$ be rules.
 - R *depends* on R' if there exists a (negated or non-negated) query term t^q in Q such that t^q simulation unifies in $t^{c'}$
 - R *depends positively* on R' if there exists a non-negated query term t^q in Q such that t^q simulation unifies in $t^{c'}$
 - R *depends negatively* on R' if there exists a negated query term *not* t^q in Q such that t^q simulation unifies in $t^{c'}$
2. P is called *fully stratifiable* (or simply *stratifiable*), if there exists a partitioning ($n \geq 1$)

$$P = P_1 \uplus \dots \uplus P_n$$

of P such that for every stratum P_i ($1 \leq i \leq n$) and for every rule $R \in P_i$ holds:

- if R depends negatively on a rule R' , or the head of R contains grouping constructs and R depends positively or negatively on R' , then $R' \in \bigcup_{j < i} P_j$, i.e. R' is in a strictly lower stratum than R
- if the head of R contains no grouping constructs and R depends positively on a rule R' then $R' \in \bigcup_{j \leq i} P_j$, i.e. R' is in the same or in a lower stratum than R

The partition $P = P_1 \uplus \dots \uplus P_n$ is called a *full stratification* of P , and the P_i are called *strata* of P .

²Rather than calling a program *stratified* as in the original definition, we call it *stratifiable* as it is not necessary to compute the stratification during (backward chaining) evaluation.

Part II

Language Extensions and Open Issues

Chapter 8

Node Identity in Xcerpt

8.1 Introduction

Managing structured or semi-structured data involves the determination of what defines the identity of a data item (be it a node in a tree, graph, or network, an object, a relational tuple, a term, or an XML element). Identity of data items is relevant for a variety of concepts in data management, most notably for joining, grouping and aggregation, as well as for the representation of cyclic structures.

“What constitutes the identity of a data item or entity?” is a question that has been answered, both in philosophy and in mathematics and computer science, essentially in two ways: based on the extension (or structure and value) of the entity or separate from it (and then represented through a surrogate).

Extensional Identity Extensional identity defines identity based on the extension (or structure and value) of an entity. Variants of extensional identity are Leibniz’s law¹ of the *identity of indiscernibles*, i.e., the principle that if two entities have the same properties and thus are indiscernible they must be one and the same. Another example of this view of identity is the *axiom of extensionality* in Zermelo-Fraenkel or Von Neumann-Bernays-Gödel set theory stating that a set is uniquely defined by its members.

Extensional identity has a number of desirable properties, most notably the compositional nature of identity, i.e., the identity of an entity is defined based on the identity of its components. However, it is insufficient to reason about identity of entities in the face of changes, as first pointed out by Heraclitus around 500 BC:

ποταμοῖσι τοῖσιν αὐτοῖσιν ἐμβαίνουσιν ἕτερα καὶ ἕτερα
ὔδατα ἐπιρρεῖ·

You cannot step twice into the same river; for fresh waters are flowing in upon you.
(HERACLEITUS, *Fragment 12*)

He postulates that the composition or extension of an object defines its identity and that the composition of any object changes in time. Thus, nothing retains its identity for any time at all, there are no persistent objects.

¹So named and extensively studied by Willard V. Quine.

This problem has been addressed both in philosophy and in mathematics and computer science by separating the extension of an object from its identity.

Surrogate Identity Surrogate identity defines the identity of an entity independent from its value as an external surrogate. In computer science surrogate identity is more often referred to as *object identity*. The use of identity separate from value has three implications (cf.[5] and [38])

- In a model with surrogate identity, naturally two notions of object *equivalence* exist: two entities can be identical (they are the same entity) or they can be equal (they have the same value).
- If identity is separate from value, identity is no longer necessarily compositional and it is possible that two distinct entities *share* the same (meaning identical, not just same value) properties or sub-entities.
- *Updates* or changes on the value of an entity are possible without changing its identity, thus allowing the tracking of changes over time.

In [5] value, structure, and location independence are identified as essential attributes of surrogate identity in data management. An identifier or identity surrogate is value and structure independent if the identity is not affected by changes of the value or internal structure of an entity. It is location independent if the identity is not affected by movement of objects among physical locations or address spaces.

Object identity in object-oriented data bases following the ODMG data model fulfill all three requirements. Identity management through primary keys as in relational databases violates value independence (leading to Codd's extension to the relation model [19] with separate surrogates for identity). Since object-oriented programming languages are usually not concerned with persistent data, their object identifiers often violate the location independence leading to anomalies if objects are moved (e.g., in Java's RMI approach).

Surrogate or object identity poses, among others, two challenges for query and programming languages based on a data model supporting this form of identity: First, where for extensional identity a single form of equality (viz.the value and structure of an entity) suffices, object identity induces at least two, often three flavors of *equality* (and thus three different *joins*): Two entities may be equal w.r.t. identity (i.e., their identity surrogates are equivalent) or value. If entities are complex, i.e., can be composed from other objects, one can further distinguish between "shallow" and "deep" value equality: Two entities are "shallow" equivalent if their value is equal and their components are the same objects (i.e., equal w.r.t. identity) and "deep" equivalent if their value is equal and the values of their components are equal. Evidently, "shallow" value-based equality can be defined on top of identity-based and "deep" value-based equality.

The same distinction also occurs when *constructing* new entities based on entities selected in a query: A selected entity may be linked as a component of a constructed entity (object sharing) or a "deep" or "shallow" copy may be used as component.

Summarizing, surrogate or object identity is the richer notation than extensional identity addressing in particular object sharing and updates, but conversely also requires a slightly more complex set of operators in query language and processor.

Following a short outlook at related work on object identity in data management, the advantages and challenges for introducing surrogate identity in Xcerpt, a versatile Web query language for XML and RDF, are investigated.

8.2 Object Identity in Data Management or “How do the others solve it?”

The need for surrogate identity in contrast to extensional identity as in early proposals for relational databases has been argued for [19], as early as 1979 by Codd himself. He acknowledges the need for unique and permanent identifiers for database entities and argues that user-defined, user-controlled primary keys as in the original relational model are not sufficient. Rather permanent *surrogates* are suggested to avoid anomalies resulting from user-defined primary keys with external semantics that is subject to change.

In [38] an extensive review of the implications of object identity in data management is presented. The need for object identity arises if it is desired to “distinguish objects from one another regardless of their content, location, or addressability” [38]. This desire might stem from the need for dynamic objects, i.e., objects whose properties change over time without losing their identity, or versioning as well as from object sharing.

[38] argues that identity should neither be based on address (-ability) as in imperative programming languages (variables) nor on data values (in the form of identifier keys) as in relational databases, but rather a separate concept maintained and guaranteed by the database management system.

Following [38], programming and query languages can be classified in two dimensions by their support for object identity: the first dimension represents to what degree the identity is managed by the system vs. the user, the second dimension represents to what degree identity is preserved over time and changes.

Problems of user defined identity keys as used in relational databases lie in the fact that they cannot be allowed to change, although they are user-defined descriptive data. This is especially a problem if the identifier carries some external semantics, such as social security numbers, ISBNs, etc. The second problem is that identifiers can not provide identity for some subsets of attributes.

The value of object identities (OIDs) as query language primitives is investigated in [1]. It is shown that OIDs are useful for

- object sharing and cycles in data,
- set operations,
- expressing any computable database query.

The data model proposed in [1] generalizes the relational data model, most complex-object data models, and the logical data model [40]. At the core of this data model stands a mapping from OIDs to so-called *o*-values, i.e., either constants or complex values containing constants or further OIDs. Repeated applications of the OID-mapping yield *pure values* that are regular infinite trees. Thus trees with OIDs can be considered finite representations of infinite structures.

The OID-mapping function is partial, i.e., there may be OIDs with no mapping for representing incomplete information.

It is shown that “a primitive for OID invention must be in the language . . . if unbounded structures are to be constructed” [1]. Unbounded structures include arbitrary sets, bags, and graph structures.

Lorel [2] represents a semi-structured query language that supports both extensional and object identity. Objects may be shared, but not all “data items” (e.g., paths and sets) are objects, and thus not all have identity. In Lorel construction defaults to object sharing and grouping defaults to duplicate elimination based on OIDs.

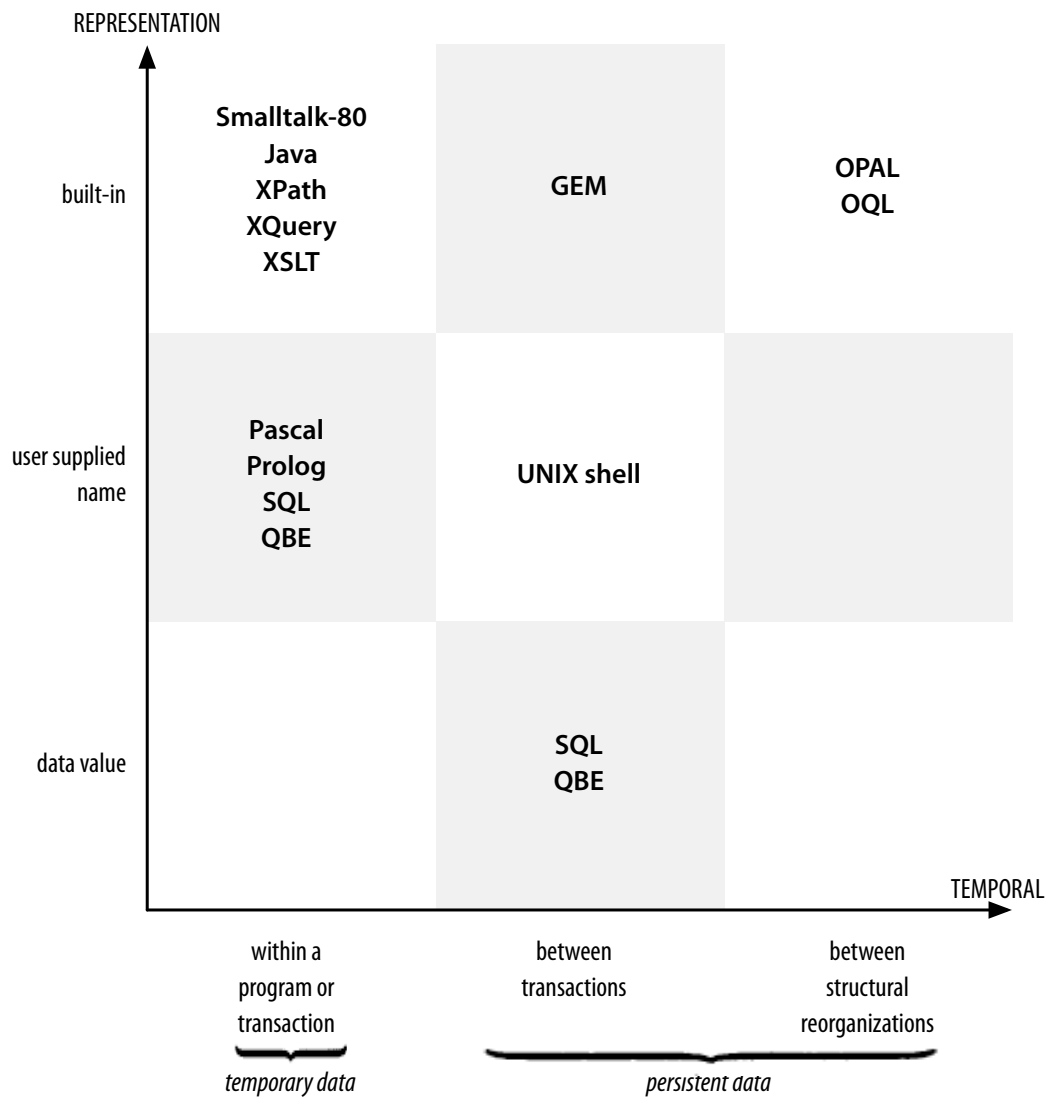


Figure 8.1: Identity in Programming and Query Languages

```

article [
2  sect [
    para [ "Wie froh ..." ]
4  title [ "First" ]
    intro @ sect [
6  title [ "First" ]
    sect [
8  title [ "First" ]
    ^intro
10 ]
]
12 ]
]

article [
2  sect [
    para [ "Wie froh ..." ]
4  title [ "First" ]
    intro @ sect [
6  title [ "First" ]
    sect [
8  title [ "First" ]
    sect [
10 title [ "First" ]
    sect [
12 title [ "First" ]
    ^intro
14 ] ] ] ] ]
] ] ] ] ]

```

Figure 8.2: Cyclic Xcerpt Data Terms

8.3 Issue Description or “Do we really need object identity?”

In the classification scheme established in the previous section, Xcerpt currently supports extensional identity exclusively. Before presenting arguments for surrogate identity, a closer look at the current Xcerpt data model is needed.

8.3.1 Regular Infinite Trees

In contrast to claims in prior descriptions of Xcerpt, the *data model* of Xcerpt is indeed not much so based on graphs, as on regular infinite trees. This is in part due to the lack of object sharing at the level of the data model (rather than only in the serialization of data) and thus very much related to the topic of this article. After [20] a (possibly infinite) tree t is called *regular*, if under structural equality the set of all its (possibly infinite) subtrees is finite. Acyclic Xcerpt data terms can be seen as finite regular trees, cyclic Xcerpt data terms as infinite regular trees, since their number of subtrees is finite under structural equality. Consider, e.g., the two data terms in Figure 8.2.

Both data terms have the same five subtrees, viz. the subtree rooted at `article`¹, `sect`², `para`³, `title`⁴, and `sect`⁵. Evidently, the number of explicit representations of the cycle in the data term does not affect the number of subtrees.

This result is unsurprising in the light of [1], where it is shown that a graph-shaped object-oriented data model (with object sharing) can be reduced to a regular infinite tree, if one ignores object identity replacing (recursively) each object reference with the value of the object. The informed reader will notice that the latter is the conceptual *modus operandi* of Xcerpt as defined so far.

The cause of this limitation is a desirable property of Xcerpt, viz. that parent-child and ID-IDREF links are indistinguishable in the data model and in queries. This means in other words, that Xcerpt does not distinguish between object copy and object reference or sharing.

In a pure *tree data model* this is indeed no limitation at all, since in a tree data model the position of an object (i.e., the position among its siblings and (recursively defined) the identity of its parent node) is sufficient for a unique identification of that object. However, *in graphs* this does not suffice due to

object sharing, i.e., the occurrence of the identical object at different positions. Positional identity would, e.g., in the data term $a\{x@b\}, \hat{x}$ result in two a children of the b with different identity. Considering the example $x@a\{\hat{x}, \hat{x}\}$ one sees how this “positional” identity leads to infinitely many objects, if the term contains cyclic references. “infinite trees”. This is also the root cause why there is no Xcerpt query that can distinguish between the data term $a\{x@b\}, \hat{x}$ and the data term $a\{b\}, b\}$. Neither is there a query that can distinguish the two data terms in Figure 8.2.

Indeed, positional identity is already used in Xcerpt to some extent, viz. in the index injectivity property of the simulation relation.

To sum up, currently Xcerpt uses a data model based on regular infinite trees but for serialization (and in-memory representation) these regular trees are represented as finite and, in general, cyclic graphs. The following section argues that there are at least three reasons, why a reconsideration of object identity and as an immediate consequence a proper graph data model is worthwhile.

8.3.2 Object Identity: Updates, Sharing,

Historically, there are two main incentives for object identity vs. extensional identity in data management systems, viz. object sharing and object updates. Both apply also for Xcerpt. Additionally, object identity is needed in Xcerpt to properly support an important class of queries, viz. occurrence queries.²

Object Updates: Under extensional identity object updates and transformations are indistinguishable. In particular, it is not possible to *track* changes to objects over time. Object identity, on the other hand, allows the *tracking* of object updates. E.g., event queries of the form “if a certain object is changed twice within 10 minutes” require some form of object identity. As argued above, it would be possible to simulate object identity through extensional identity, but this leads to a violation of the value independence and places the burden of managing the identity on the user.

Object Sharing: If object identity is provided in a data management system, object sharing is almost always desirable to make views, rules, or similar “procedural abstraction” mechanisms transparent. Without object sharing, such procedural abstractions have a side-effect, namely that all objects in their result get a new identity, even if they are extracted from queried data. This makes identity joins over rule borders infeasible (see XQuery).

Occurrence Queries: Possibly the strongest argument for object identity is, however, the need for occurrence queries. Occurrence queries ask for occurrences of certain data items in the data base. Examples of occurrence queries are, e.g., “*How many title elements occur in the first section?*”, “*List all occurrences of title in the first section!*”, or “*For each supplier count the average number of items per order (where an order may contain the same item several times)!*”.

In Xcerpt occurrence queries are only supported as “distinct” occurrence queries, i.e., in the form “*How many distinct title elements occur in the first section?*”. It is not possible to count the actual occurrences, if some of them share the same structure, as they are considered identical in this case. One might think that a simple solution here could be to use bags to gather these occurrences (instead of sets as in the current semantics of Xcerpt). This, however, suffices not as a solution: Consider, e.g., the following query term against the first data term of Figure 8.2:

```
article [[
2 position 1 sect {{
```

²Note, that one might think that identity is linked closely to duplicate elimination and thus to grouping and aggregation. In fact, this is not necessarily the case as grouping and aggregation constructs have to be based on bags or sequences (as in virtually any mainstream database query language, viz. SQL, QUEL, OQL, Lorel, XSLT, XQuery) instead of sets anyway. However, a stringent handling of object identity provides for an easier handling of grouping and aggregation.

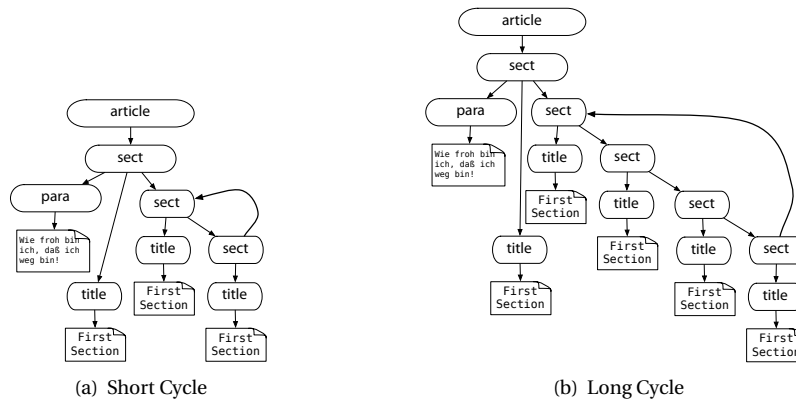


Figure 8.3: Structure-equivalent Data Terms with Different Cycle Length

```
var X →desc title {{ }}
4 }}
```

The query asks “Find all occurrences of title elements at any depth in the first section.”

How many bindings for X should this query return on the given data? If one assumes duplicate elimination based on extensional identity (structure and value of terms), a single binding for X is returned. If no duplicate elimination takes place, however, i.e., in the case of an occurrence query where simply all the occurrences are of interest, there are *infinite occurrences* under the section. This becomes more obvious if one considers the same data with a longer cycle as in the second data term of Figure 8.2. To better illustrate the point, Figure 8.3 shows a visual representation of the data.

Since references are considered transparent the answer to this query should be the same! Therefore not bound for the number of occurrences of title in the first section can be given. Indeed, this is the expected answer, if one assumes a data model of regular infinite trees as Xcerpt does (see above).

If one assumes duplicate elimination based on object identity, the query returns different answers on the two data terms, as they are no longer identical (they represent different graphs), viz. three bindings for the first data term and five for the second.

The following two tables summarize the differences between the result for the query term on the two data terms with different forms of duplicate elimination:

duplicate elimination	X
value-based	title["First"]
identity-based	t ⁴ , t ⁶ , t ⁸
none	t ⁴ , t ⁶ , t ⁸ , t ¹⁰ , t ¹² , t ¹⁴ , t ¹⁶ , ...

duplicate elimination	X
value-based	title["First"]
identity-based	t ⁴ , t ⁶ , t ⁸ , t ¹⁰ , t ¹²
none	t ⁴ , t ⁶ , t ⁸ , t ¹⁰ , t ¹² , t ¹⁴ , t ¹⁶ , ...

	Changes	Transparent Rules	Updates	Occurrence Queries	No User Manipulation
1:	few	++	-	-	-
2:	many	+	++	++	+

Table 8.1: Evaluation of the Proposals

To conclude, unrestricted occurrence queries are currently not expressible in Xcerpt (only distinct occurrence queries are). But unrestricted occurrence queries are an essential class of queries. Their introduction on a regular infinite tree data model is however at least problematic, at worst impossible. On a graph data model with object identity they are easy to handle, but an obvious consequence is the loss of Xcerpt’s property to handle reference and child links indistinguishably.

8.4 Aims of the Proposal

The following two sections present two proposals for addressing the issues raised above in this paper. They are evaluated by five possibly contradicting goals:

1. Keep the changes to Xcerpt small and easy to grasp.
2. Ensure Xcerpt’s procedural abstraction mechanism remains “transparent”, i.e., that it can be introduced without changing the query beyond the factoring out of the procedure.
3. Enable updates without identity change to allow update tracking.
4. Enable unrestricted occurrence queries.
5. Avoid the need for the user to manage identity. It should be possible to express identity joins and to create objects with new identity, but it should not be possible to explicitly manipulate the identity surrogates (e.g., identifiers).

Table 8.1 gives a brief overview of the two proposals with respect to these criteria. The details are discussed in the following two sections.

8.5 Proposal 1: Infinite Regular Trees

The first proposal is to make *no changes to the language*. However, as discussed above, currently Xcerpt’s data model resembles more an infinite regular tree than a graph. This fact should be made much clearer if this proposal is adopted. Also, the fact the simulation is *index* injective, not *node identity* injective must be clarified in this case.

The main advantage of this proposal is that Xcerpt retains the desirable property that reference links and parent-child links are indistinguishable and transparently handled. However, the price is that this precludes unrestricted occurrence queries and that the user still has to cope with the finite representations needed for serialization. Update tracking could be realized using user managed identity (e.g., keys) with all the known problems this involves.

8.6 Proposal 2: Hidden Identity

The second proposal is to move Xcerpt to a *proper graph data model* with *object or node identity*. This is obviously a major change to the language. However, it is a much needed change to address updates and occurrence queries.

On the level of the data model, a (total) surrogate function s associates each node in the domain with a unique identifier, i.e., the surrogate function is injective. The identifiers are assumed to be *globally* unique, however different documents may contain the same node (object sharing, see below). The actual shape of the identifiers is implementation dependent and *never revealed to the user*.

The query language is extended by the following construct:

- **Identity variables** are (temporary) representatives for the identity surrogate of a node. In query terms, they may exclusively occur instead of term identifiers, e.g., `desc idvar ID @ a {{ }}`. Though the position already clearly separates them from other variables in query terms, this is not true for construct terms. Therefore a different keyword, e.g., **idvar**, is used to emphasize the difference.
- As hitherto, in data terms only local “pseudo identifiers” that are resolved at parsing are allowed.
- Queries may contain **identity joins** expressed simply by repeated occurrence of the same identity variable. Cyclic queries using “pseudo identifiers” are no longer supported. This means that queries can define the length of the cycle, but arbitrary cycle length queries can still be expressed using **descendant** or qualified descendant. Such cyclic queries are not only easier to evaluate, but have also a very strict and immediate meaning.
- Joins on non-identity variables remain pure value joins. A shorthand for an unrestricted identity join might be worth considering, e.g., by allowing **idvar** outside of term identifiers in query terms.
- In construct terms, **object sharing** is achieved by using identity variables instead of normal variables, indicating that a link to the object represented by the surrogate is to be constructed. A normal variable still indicates just a deep copy of the value. When creating unbounded structures, i.e., in the case of grouping, *surrogate invention* is needed: E.g., if the result should contain an a with itself as child for each binding of some variable the link between the two must be established using surrogate invention. A possible syntax could be:

```
CONSTRUCT
2 result [
   all new_id(invention_label) @ a [
4     ^last_id(invention_label)
   ] group-by var X
6 ]
FROM
8 var X →desc title {{ }}
END
```

- All nodes have identity including text nodes, comment nodes, and processing instruction nodes. This is necessary to properly support occurrence queries.

Notice, that no direct support for shallow copies or shallow-value joins is proposed as these can be simulated by a combination of deep-value and identity joins.

Cross-Document References: Using the object sharing mechanism discussed here it becomes possible to share nodes not only between different elements in the same document but also beyond document borders. If this is a desirable property and, at least in the context of RDF, this seems to be the case, a “serialization” format for these cross document references as well as possible automatic dereferencing should be addressed in further work.

Exposable Identity: Indeed, in the context of “serialization” of identity a closer look at exposing identity must be taken: Exposing identity might be necessary for update tracking, but also for handling of documents where identity is also used to reference sub-documents (as in (X)HTML). A principled correlation between “internal” and “external” identity must be established. This will be addressed in an upcoming discussion on serialization issues for XML and RDF, cf.[30], Chapter 9.

8.7 Conclusion and Future Work

This article summarizes the state-of-the-art on identity of data items, both for general data management as well as for Xcerpt. Based on this analysis, a number of problems of the current status in Xcerpt are discussed that are mostly equally valid for any language that uses extensional identity. Two proposals for addressing these problems are discussed and evaluated along a number of criteria.

The evaluation clearly shows that a stringent introduction of object identity in Xcerpt is possible, but that it is also a not trivial change to the language.

Chapter 9

Grouping and Aggregation in Xcerpt

9.1 Introduction

Currently, the Xcerpt semantics uses substitution sets, but the description of construct terms presented in Section 5 uses multi-sets. In the case of *sets*, duplicate substitutions are removed. Though this is useful in some queries, there are many cases where this is not necessary or even detrimental:

1. First, duplicate removal is a costly operation. Though (node identity-based) duplicates can be avoided at small additional cost for queries against trees (as shown for XPath evaluation), this is not true for general queries against graphs. In particular, the current duplicate elimination based on structural equivalence is unfavorable from a complexity point of view (structural equivalence between ordered terms resumes to graph isomorphism).
2. Second, for many types of queries duplicate removal is not wanted, e.g., in transformation and aggregation queries: It is currently not possible to express such simple and straightforward queries as “Count the number of title elements in a document” or “Replace all a labeled elements in the input tree with b labeled elements otherwise retaining the same structure” without resorting to complicated rules using **position**.

9.2 Multirelations, Bags, and Sequences

Multirelations (or relations with duplicate tuples) are usually ignored in relational model theory as relations as sets make many theoretical considerations, such as relational algebra and complexity theory, easier and more elegant. Nevertheless, the need for multirelations is evident when considering aggregation queries like “*what is the sum of the values of some column*”. In a set relational algebra the projection to the aggregation column collapses all same value tuples and thus makes this query impossible to express. The same argument can be made for any query that returns just a projection of the columns of queried relations, as the number of duplicates may be significant to the user (e.g., in “*what are the titles of all first and second level sections*”). Also, many practical systems support multirelations to save the cost of duplicate handling. Indeed, neither QUEL [52] nor SQL [4] are (set) relational query languages, but rather possesses features that are not expressible in (set) relational

algebra, viz. aggregation, grouping, and duplicate elimination. The semantic of these expressions assumes that the underlying data structure is a bag rather than a set.

Therefore, in practical query languages duplicate handling must be addressed. Based on the control over duplicate creation and elimination, one can distinguish relational query languages into *weak* and *strong* duplicate controlling languages. QUEL [52] and SQL [4] provide little control over duplicates (essentially just the **DISTINCT** operator in combination with **GROUP-BY** clauses) and thus fall into the first class. The only means is an explicit duplicate elimination. Similarly, Prolog's operational semantics [53] also contains a operations for explicit duplicate handling (e.g., **bagof** vs. **setof**).

In contrast, DAPLEX [51] is based on “iteration semantics” and gives precise control over the creation and elimination of duplicates. An example of a DAPLEX query is shown in the following listing:

```

FOR EACH Student
2 SUCH THAT FOR SOME Course(Student)
    Name(Dept(Course)) = "EE" AND
4 Rank(Instructor(Course)) = "ASSISTANT PROFESSOR"
PRINT Name(Student)

```

A first formal treatment of this “iteration semantics” for relational databases is found in [23], where a *generalization of the relational algebra* to multirelations is proposed. This extension is not trivial and raises a number of challenges for optimizations: joins are no longer idempotent, the position of projections and selections is less flexible, as $\pi_R(R \times S) \neq R$ and $\sigma_P(R) \uplus \sigma_Q(R) \neq \sigma_{P \vee Q}(R)$ due to duplicates in the first expression¹. Though this algebra provides a useful theoretical foundation, it does little to address the concerns regarding efficient processing of “iteration semantics” expressions.

[34] shows that (nested) relational algebra on multirelations (i.e., relations as bags) is strictly more expressive than on relations as sets. Unsurprisingly, the core difference lies in the “counting power” of the bag algebra. Similarly, [41] proposes a query language called \mathcal{BQL} over bags whose expressiveness amounts to that of a relational language with aggregates. This approach provides a formal treatment of aggregations and grouping as found, e.g., in SQL (**GROUP-BY** clause and aggregation operators such as **AVG**, **COUNT**, and **SUM**). \mathcal{BQL} is “seen as a rational reconstruction of SQL” that is fully amenable to formal treatment. [41] also considers extensions of \mathcal{BQL} with power operators, structural recursion, or loops and shows that the latter two extensions are equivalent.

[39] proposes a different view of multirelations as *incomplete information*: though conceptually “a relation represents a set of entities”, one tuple per entity, and thus does not contain duplicates, the concept of a multirelation allows a formal treatment of partial information about entities. A multirelation is a projection of a complete relation, i.e., it consists of a subset of columns within some relation (without duplicates). Thus it may contain duplicates in contrast to the relation. [39] considers multirelations only as output of queries not as first class data items in the database. Semantically, they can not exist independently of the base relation. No (base or derived) relation should contain duplicates.

¹Assuming π and σ to be the multiset generalizations of their relational counterparts. \uplus is understood here as *additive union*, i.e., t occurs n times in $R \cup S$, if t occurs i times in R and j times in S and $n = i + j$. [34] considers additionally maximal union (i.e., where $n = \max(i, j)$), which does not exhibit this particular anomaly.

9.3 Solution Proposal

The solution shown in Section 5.6 already uses multi-sets as foundation for construction. Syntactically, there are no further changes needed. A new specification of the semantics is currently under development as part of the effort to specify a query algebra for Xcerpt 2.0 and related query languages such as XQuery and XPath.

Chapter 10

Modules in Xcerpt 2.0—Reuseware Integration

10.1 Introduction

In this chapter, we first discuss a generic module extension framework for rule languages developed jointly between I3 and I4 and then show some consideration of a suitable extension employing that framework for Xcerpt. For details on the implementation see I4-D13.

10.2 Rule Languages

A relatively common way of expressing or accessing knowledge is by means of *rules*. Rules are considered a natural and easily understandable formalism for knowledge management. Rules can be considered a versatile formalism covering a wide range of application areas, including deductive querying, reactive data, data construction and schema specification. In particular, rules are currently playing a major role in the development of the Semantic Web with languages such as SWRL, RIF-Core, Xcerpt and XChange etc.

In this section we introduce the notion of rules in a general manner as possible, such as not to be weighed down with the details of one particular language. One of the advantages of rules is their declarative nature, which makes them easy to read and specify. As such rules can also be used by non-technical users, e.g. for a manager to clearly specify business strategies to be implemented in an organization.

Rules in a particular language are usually specified in a *rule-base*, a finite set of (possibly) related simple rules. A *simple rule* consists of a *head* and a *body* and has the following form:

$$\textit{head op body}$$

where *body* and *head* are sets of atomic formula of the rule language. The formula of each set are composed using some language specific logical connectives. The operator *op* relates the head and the body of the rule, and thus defines how the rule is to be understood. Depending on the particular rule language, different logical connectives and relation operators are selected.

Datalog A well-known rule language is the database query language Datalog. For Datalog, the operator *op* is instantiated with the implication operator \leftarrow . As such, rules are read "if the body holds,

then the head also holds". In Datalog, the *head* is usually considered to be a singleton set and the *body* a set of Datalog atoms connected via conjunction (\wedge). A Datalog atom is an n -ary predicate of the form $p(a_1, \dots, a_n)$, where $a_i (1 \leq i \leq n)$ are constant symbols or variables. As such, a Datalog rule may take the following form:

$$\text{uncle}(\text{john}, \text{steve}) \leftarrow \text{father}(\text{john}, \text{pa}), \text{brother}(\text{pa}, \text{steve})$$

to be understood as defining that *john* is the uncle of *steve* if *pa* is the father of *john* and *steve* is the brother of *pa*.

For other rule languages, the general rule form presented above can be instantiated differently and rules associated with specific semantics. Note that the semantics associated with rules are specific to each rule language and cannot be described in general terms. For the case of Datalog, semantics are given by definition of a least Herbrand model of a rule-set.

10.3 Framework for rule language module systems

In Section 10.2 we introduced rule languages and their form in general terms and described their usefulness. In the following section, we introduce the formal notions of our rule language module system framework. This is achieved by first establishing requirements for specific rule languages such that they fit the framework. That is, if the rule language fulfills the requirements then it is applicable to the framework. We then briefly introduce two such rule languages which fulfill the requirements, viz. Xcerpt and R_2G_2 . Xcerpt in particular we will treat in Section 10.5 where we use the notions of the framework to realize an Xcerpt module system.

The main assumption made on a rule language is that a rule language has a concept of "rule chaining", or "rule dependency", where one rule depends on another for proper processing.

More precisely, for a language to be amenable to the module extension framework presented here, it must fulfill the following requirements:

1. It must have some form of **program** concept. A program is understood as a (finite) *sequence of rules*. Note, that here and in the following we use sequences to allow the support for languages where order is relevant. Obviously, if rule order is irrelevant, an arbitrary order may be chosen.
2. A **rule** is understood as consisting in (exactly) one head and (exactly) one body, where each the body and the head are (finite) *sequences of rule parts*. Body rule parts are understood as a kind of condition, head rule parts as a kind of result, action, or other consequence of the rule.
3. **Rule parts**¹ are, for the purpose of this work, atomic.
4. Finally, and most importantly, we assume that a rule language has a concept of **accessibility**: With each program P , a relation $\aleph \subset \mathbb{N}^2 \times \mathbb{N}^2$ can be associated such that $(i_{r_1}, i_b, i_{r_2}, i_h) \in \aleph$ if results of the i_h -th head part of the i_{r_2} -th rule in P (such as derived data, actions taken, state changes, or triggered events) *are accessible* to (and thus may affect) the condition expressed by the i_b -th body part of rule i_{r_1} . Conversely, if a pair $(i_{r_1}, i_b, i_{r_2}, i_h) \notin \aleph$ the results of (i_{r_2}, i_h) *may not* be accessible to (i_{r_1}, i_b) . Intuitively, \aleph partitions the space of rule parts into allowed and forbidden pairs.

A rule language is required to provide means to express the allowed accessibility in an arbitrary \aleph relation on a given program. How this achieve is left to the language.

¹We refrain from calling rule parts literals, as they may be, e.g., entire formulas or other constructs such as actions that are not usually considered logical literals.

Notice, that the notion of accessibility is weaker than common notions of rule dependency or rule dependency graph. Furthermore, the module extension framework only requires the ability to express an arbitrary accessibility relation, however poses no restrictions on the shape of \aleph for a module-free program. Indeed, for any module-free rule program $P \mathbb{N}^4$ forms a perfectly acceptable \aleph relation on P .

However, more restricted versions of \aleph for module-free programs may help to better “localize” the effect of module imports and thus are usually beneficial. E.g., in case of Datalog predicate symbols are often partitioned into extensional and intensional ones and this can be exploited to only adapt \aleph for intensional body parts since only those can be affected by a module import. Essentially, whenever a body part is explicitly referring to external data or resources (i.e., data that can not be resulting from rule applications) the same argument applies. This is, e.g., the case if external resources are accessed in query languages such as Xcerpt. Xcerpt, e.g., surrounds body parts with `in` statements specifying the external resource to access.

To summarize, the main requirement on a rule language to be amenable to the proposed module extension framework is the ability to express (in some way) arbitrary accessibility relations. This requirement is fulfilled by most rule languages that allow some form of rule chaining, e.g., Datalog, SWRL, SQL, Xcerpt, R_2G_2 . However, it precludes rule languages such as CSS where all rules operate on the same input and no rule chaining is possible. CSS already provides its own module concept but without information hiding. Rules from all imported modules are merged into one sequence of rules and all applied to the input data.

10.4 Module system operators

As discussed above, the aim of the module extension framework is to allow a rule program to be divided into conceptually independent collection of rules with well-defined interfaces between these collections. For this purpose, we first in Section 10.4.1 define operators that define collections of rules (called “modules”) and, among such a collection of rules, which rules contribute to the (public) interface of the collection. These operators are complemented with module access operators in Section 10.4.2 which specify where the (public) interface of a previously defined module is accessible.

10.4.1 Module Definition Operators

We use *module identifiers* (from some alphabet Σ_{ID} as means to refer to modules, e.g., when importing modules. Some means of resolving module identifiers to modules (stored, e.g., in files or in a database) is assumed. See Section ?? for an example how this is achieved in the case of the Xcerpt module system.

Module Formally, we understand a module M as a quadruple $(R_1, R_2, \aleph, ID) \in \mathcal{R} \times \mathcal{R} \setminus \{\emptyset\} \times \mathbb{N}^4 \times \Sigma_{ID}$ where \mathcal{R} is the set of all finite sequences over the set of permissible rules for a given rule language. We call R_1 the private, R_2 the public rules of M , \aleph the accessibility relation for M , and ID its identifier (which is assumed to be unique among all modules). For the purpose of numbering rules, we consider $R = R_1 \diamond R_2$ the sequence of all rules in M .

In the following, we consider, where convenient, a program as a special case of a module where all rules are public.

There is only a single, unsurprisingly simple module construction operator:

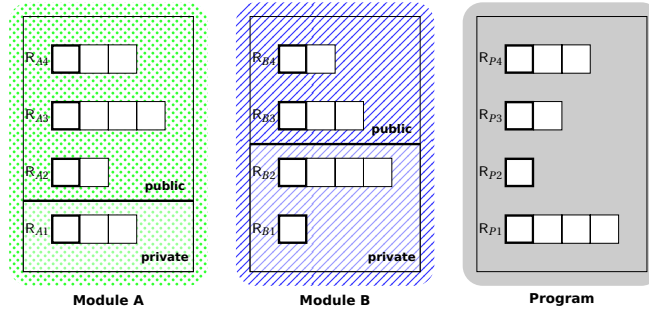


Figure 10.1: Program and two defined modules without imports

CREATE-MODULE Σ_{ID} be a set of module identifiers. Then, $\text{CREATE-MODULE} : \mathcal{R} \times \mathcal{R} \setminus \{\{\}\} \rightarrow \Sigma_{ID}$. $\text{CREATE-MODULE}(r_1, r_2)$ creates a module from the finite sequence of private rules r_1 and the finite, non-empty sequence of public rules r_2 and returns an identifier for the newly created module.

Notice, that only the sequence of private rules may be empty. Every module is expected to consist in at least one public rule. It is expected that the returned identifier is uniquely resolvable to the module created by this call of CREATE-MODULE . However, the details of module storage and resolution are left to each instance of the module extension framework.

Figure 10.1 shows an exemplary configuration of a program together with two modules A and B. Where the program consists in a single sequence of rules, the rules of each of the modules are partitioned into private and public rules. The allowed dependency relation \aleph is represented in the following way: All body parts in each of the areas \dots , \dots , and \dots are depending on all head parts in the same area and no other head parts. No access or import of modules takes place, thus no inter-module access exists in \aleph between rule parts from one of the modules with each other or with the (main) program.

Notice, that for the accessibility within a module the partitioning in private and public plays no role whatsoever. Body parts in private rules may access head parts from public rules and vice versa.

10.4.2 Module Access Operators

Module access or import operators allow the (principled, i.e., via their public interface) definition of inter-module access. Three types of module access and their corresponding operators are discussed here that differ mostly in the amount of information hiding. All three operators hide information resulting from private rules in a module, however the public information is made accessible in different ways by each of the operators: The private and public import operators differ only if modules are cascaded, i.e., a module A that imports another modules B is itself imported. In that case the public import operator makes the public interface of B part of the public interface of A , whereas the private import operator keeps the import of B hidden. Both operators make all information from the public interface of B accessible to all rules of A . In contrast, the third operator, called qualified import operator, makes the information from the public interface of B accessible only to explicitly marked rules.

Formally, the operators are defined as follows:

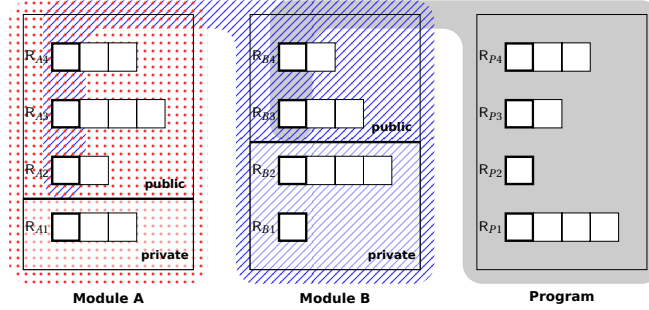


Figure 10.2: Private import of A into B and B into the main program

PRIVATE-IMPORT Given a module identifier ID' and a module $M = (R_1, R_2, \aleph, ID)$, $PRIVATE-IMPORT(ID', M)$ resolves the module identifier to obtain a module $M' = (R'_1, R'_2, \aleph', ID')$ that is to be imported into M and returns a new module $M'' = (R''_1 = R_1 \diamond R'_1 \diamond R'_2, R''_2 = R_2, \aleph'', ID)$. \aleph'' is the accessibility relation for the resulting module: $\aleph'' = I \cup I' \cup C$ where $I = \text{slide}(\aleph, [|R_1| + 1, |R_1 \diamond R_2|], [|R'_1| + 1, |R'_1 \diamond R'_2|])$, i.e., the accessibility relation of the importing module with R_2 slided after R'_1 and R'_2 , $I' = \text{slide}(\aleph', [|R_1| + 1, |R'_1|])$, i.e., the accessibility relation of the imported module slided after R_1 and $C = \{(i, j, k, l) \in \mathbb{N}^4 : \exists r, b, r', h : r \text{ is the } i\text{-th rule in } R_1 \diamond R_2 \wedge b \text{ is the } j\text{-th body part in } r \wedge r' \text{ is the } k\text{-th rule in } R_1 \diamond R'_1 \diamond R'_2 \wedge k > |R_1 \diamond R'_1| \wedge h \text{ is the } l\text{-th head part in } r'\}$ is the part of the accessibility relation that established that all body parts of the importing module may access all *public* head parts of the imported modules.

Figure 10.2 shows the effect of the private import operator on the configuration from Figure 10.1: Module B imports module A privately and the main program imports module B privately. In both cases, the immediate effect is the same: The body parts of B get access to the head parts in A 's public rules and the body parts of the main program get access to the head parts in B 's public rules. The import of A into B is hidden entirely from the main program. This contrasts to the case of the public import shown in Figure 10.3. There the main program's body parts also gain access to the head parts in A 's (and not only B 's) public rules.

PUBLIC-IMPORT Given a module identifier ID' and a module $M = (R_1, R_2, \aleph, ID)$, $PUBLIC-IMPORT(ID', M)$ resolves the module identifier to obtain a module $M' = (R'_1, R'_2, \aleph', ID')$ that is to be imported into M and returns a new module $M'' = (R''_1 = R_1 \diamond R'_1, R''_2 = R_2 \diamond R'_2, \aleph'', ID)$. \aleph'' is the accessibility relation for the resulting module: $\aleph'' = I \cup I' \cup C$ where $I = \text{slide}(\aleph, [|R_1| + 1, |R_1 \diamond R_2|], [|R'_1| + 1, |R'_1 \diamond R'_2|])$, i.e., the accessibility relation of the importing module with R_2 slided after R'_1 , $I' = \text{slide}(\text{slide}(\aleph', [|R_1| + 1, |R'_1 \diamond R'_2|]), [|R'_1 \diamond R_2| + 1, |R'_2|])$, $[1, |R'_1|]$, $[|R_1| + 1, |R'_1|]$, i.e., the accessibility relation of the imported module with R'_1 slided after R_1 and R'_2 slided after R_2 and $C = \{(i, j, k, l) \in \mathbb{N}^4 : \exists r, b, r', h : r \text{ is the } i\text{-th rule in } R_1 \diamond R_2 \wedge b \text{ is the } j\text{-th body part in } r \wedge r' \text{ is the } k\text{-th rule in } R_1 \diamond R'_1 \diamond R_2 \diamond R'_2 \wedge k > |R_1 \diamond R'_1 \diamond R_2| \wedge h \text{ is the } l\text{-th head part in } r'\}$ is the part of the accessibility relation that established that all body parts of the importing module may access all *public* head parts of the imported modules.

QUALIFIED-IMPORT Given a module identifier ID' and a module $M = (R_1, R_2, \aleph, ID)$, and a set P of pairs (i, j) identifying body parts in rules that explicitly access the imported module, $QUALIFIED-IMPORT(ID', M, P)$ resolves the module identifier to obtain a module $M' = (R'_1, R'_2, \aleph', ID')$ and returns a new module $M'' = (R''_1 = R_1 \diamond R'_1 \diamond R'_2, R''_2 = R_2, \aleph'', ID)$. \aleph'' is the accessibility relation for the resulting module:

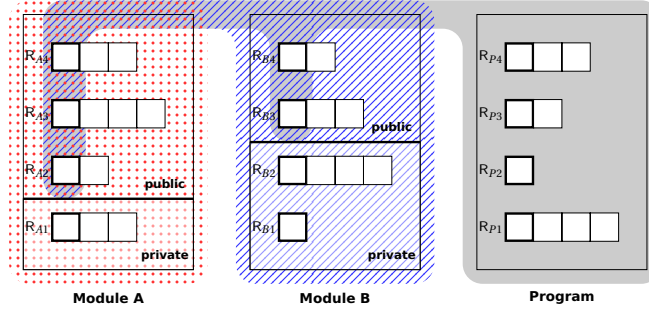


Figure 10.3: Public import of A into B and B into the main program

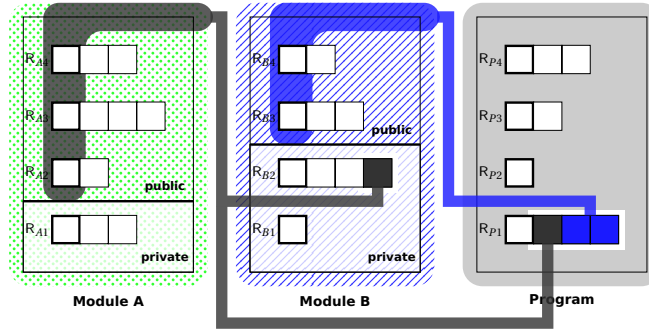


Figure 10.4: Qualified import of (1) module A into body part 3 of rule R_{B2} and into body part 1 of rule R_{P1} and (2) of module B into body part 2 and 3 of rule R_{P1} . into the main program

$\aleph'' = I \cup I' \cup C$ where $I = \text{slide}(\aleph, [|R_1| + 1, |R_1 \diamond R_2|], [|R'_1| + 1, |R''_1 \diamond R_2|])$, i.e., the accessibility relation of the importing module with R_2 slid after R'_1 and R'_2 , $I' = \text{slide}(\aleph', [1, |R'_1 \diamond R'_2|], [|R_1| + 1, |R''_1|])$, i.e., the accessibility relation of the imported module slid after R_1 and $C = \{(i, j, k, l) \in \mathbb{N}^4 : (i, j) \in P \wedge \exists r', h : r' \text{ is the } k\text{-th rule in } R_1 \diamond R'_1 \diamond R'_2 \wedge k > |R_1 \diamond R'_1| \wedge h \text{ is the } l\text{-th head part in } r'\}$ is the part of the accessibility relation that established that only the explicitly qualified body parts in P may access any of the *public* head parts of the imported module.

The effect of the qualified import operator is, expected, more focused than that of either the public or the private import operator. For the configuration from Figure 10.1, it is shown in Figure 10.4, assuming that (1) module A is imported into B qualifying only body part 3 of rule R_{B2} and that (2) module A is imported into the main program qualifying only body part 1 of rule R_{P1} for access and module B is imported into the main program qualifying only body parts 2 and 3 of rule R_{P1} .

10.5 Extending Xcerpt for Module Support

The following two sections give first extensions to the abstract of Xcerpt as introduced in the first part of this deliverable and second extensions to the concrete term syntax. Extensions for the other syntactical variants should be straightforward. Notice, that the discussed realisation of the module system uses a sophisticated *parametrized* module concept that allows arbitrary expressions of the module algebra from Section 10.3 to be used for module import in Xcerpt. E.g., one can import an

RDF reasoner parametrizing the actual RDF database, which parts of the RDF(S) entailment rules to use, whether to include OWL- reasoning, etc.

A more complete description of the module system is currently being specified in joint work between I3 and I4 and is expected to be published as extra REVERSE deliverable in 2007.

10.5.1 Abstract Syntax

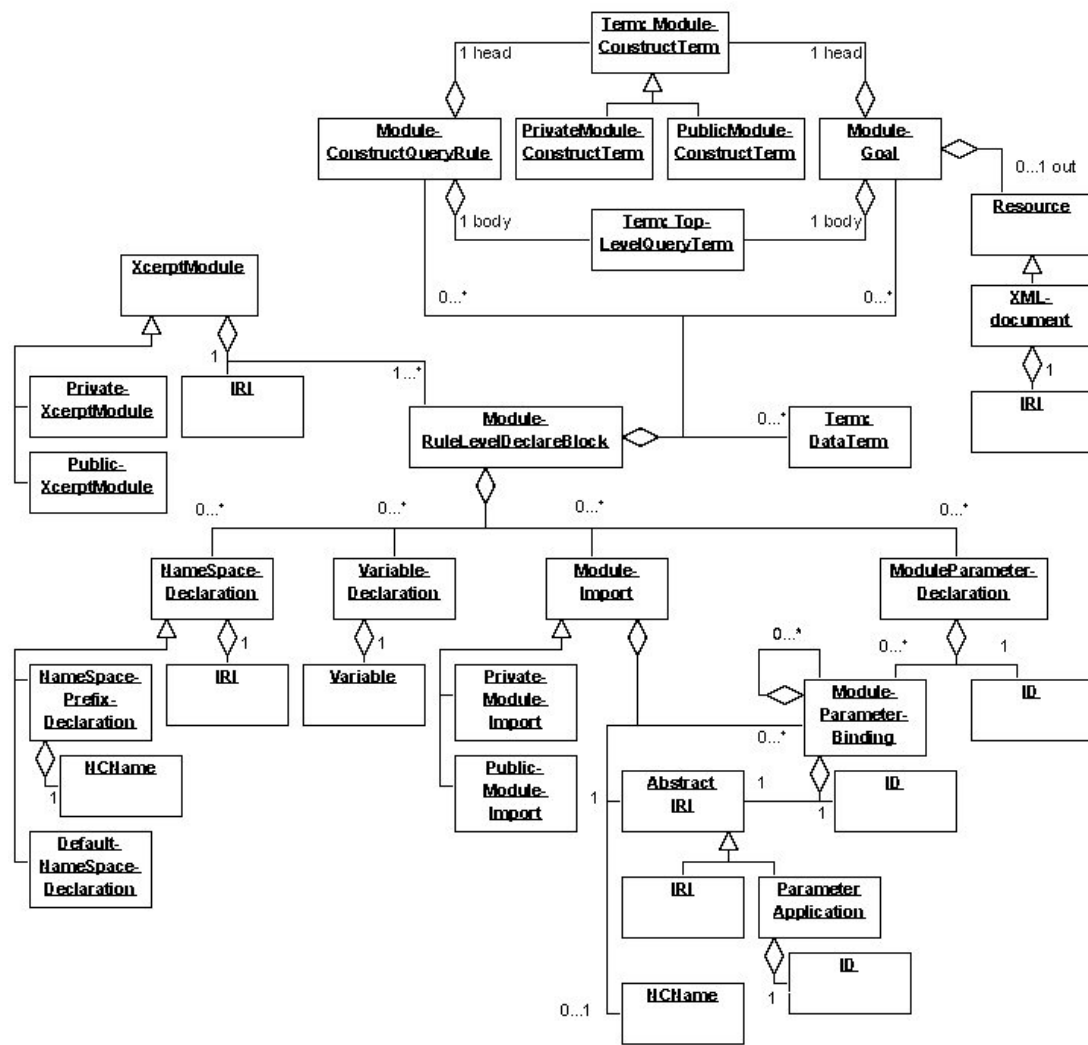


Figure 10.5: Xcerpt Module

10.5.2 Term Syntax

```
<module> ::= <visibility> 'MODULE' '(' <iri>
                                     <module-rule-level-declare-block>+
                                     ')' 'END'

<visibility> ::= ('public' | 'private')

<module-rule-level-declare-block> ::= <module-goal>
                                     | <module-construct-query-rule>
                                     | <data>
                                     | <module-rule-level-declare>
                                     <module-rule-level-declare-block>+

<module-goal> ::= 'GOAL' '(' <resource> ')'
                 'FROM' '(' <top-level-query-term> ')'
                 'END'

<resource> ::= <module-construct-term>
              | 'out' ( <iri> | <literal-var> ) '(' <module-construct-term> ')

<module-construct-term> ::= <visibility>? <construct-term>
                          | <NCName> <construct-term>

<module-construct-query-rule> ::= 'CONSTRUCT' '(' <module-construct-term> ')'
                                 'FROM' '(' <top-level-query-term> ')'
                                 'END'

<module-rule-level-declare> ::=
    'DECLARE' '(' <var-decl-qt>* <ns-decl-qt>* ')'
  | 'DECLARE PARAM' '(' ( <NCName>
                        ( 'AS' <iri>
                          | 'AS PARAM' <NCName>
                          | 'WITH DEFAULT' ( <iri> | 'PARAM' <NCName> )
                        )?
                        ','?
                      )+
    ')'
  | <visibility>? ( 'IMPORT' '(' <iri>
                  | 'IMPORT PARAM' '(' <NCName>
                  )
                  ( 'AS' <NCName> )?
                  <using-expressions>*
                  ')'


```

```
<using-expressions> ::= 'USING' ( <iri> | <NCName> )  
                        ( '(' <using-expressions> ')' ) *  
                        'AS PARAM' <NCName>
```


Chapter 11

RDF Access in Xcerpt 2.0—An Outlook

11.1 Introduction

Data on the web is increasingly enriched with semantic meta-data, linking it to the real world or to other information. While XML has already gained wide-spread acceptance, RDF is on the best way to do so. Query languages have established themselves as a valuable means for accessing both formats, and a considerable number of query languages for XML (such as XQuery[?], XPath[?], XSLT[16], Xcerpt[49, ?, ?]) and for Semantic Web data (e.g. SPARQL[?], RQL[?], Versa[?]) have been proposed and implemented, cf. [7] for a survey. XML query languages can be used to query XML serializations of RDF data. This, however, hardly yields a programmer-comfortable approach to RDF data. In fact, most of the above languages have not been specifically designed to cope with both worlds, and do not provide a uniform language and common constructs to query and transform data in the various formats. Moreover, most mainstream query languages lack a flexible data model that is powerful enough to naturally comprehend both Semantic Web data formats (especially RDF and Topic Maps) and XML.¹

This article highlights challenges related to the data model and convenient constructs for querying both standard Web and Semantic Web data with an emphasis on facilitating sophisticated reasoning. It is shown that Xcerpt's data model and querying constructs are well-suited also for the Semantic Web, but that some adjustments of Xcerpt's syntax would allow for even more effective and natural query authoring with respect to RDF.

The rest of this article is structured according to its contributions: Section 11.2 examines requirements related to the data model of versatile web query languages with focus on RDF and XML. Section 11.3 proposes an extended edge-labeled syntax for Xcerpt terms that can be straightforwardly mapped to usual Xcerpt data terms. Section 11.4 illustrates that Xcerpt's constructs for handling heterogeneity are beneficial to both XML and RDF querying. Section 11.5 underlines the importance of grouping constructs in the scope of the Semantic Web. Finally, Section 11.6 concludes this article and sheds light upon further research both with respect to the language itself and its efficient evaluation.

¹Exceptions are early query languages for semi-structured data such as XML-QL and Lorel.

11.2 Challenges Related to the Data Model

Figure 11.1 presents two possible representations of information about countries, their names and their border-countries in XML (on the left hand side) and RDF (on the right hand side). Nodes of the XML document tree are represented as grey rectangles containing the element name. Text nodes are distinguished by quotes and attribute-value pairs are displayed at the top right of the node they belong to. The namespace prefixes `rdf`, `rdfs` and `geo` are assumed to be bound to `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, `http://www.w3.org/2000/01/rdf-schema#` and `http://geo.org/#`, respectively, in the entire article. Nodes of the RDF graph on the right are either depicted as grey rectangles containing the URI or blank node name in the case of non-literals or as oval nodes in the case of literal values.

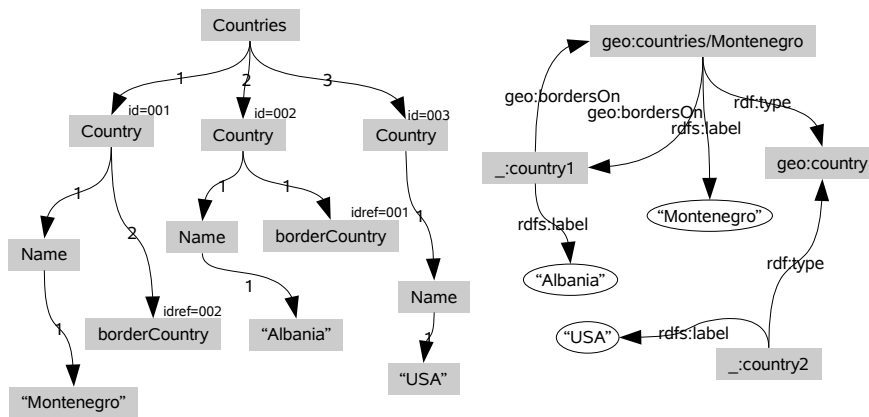


Figure 11.1: XML data versus RDF data

Figure 11.1 naturally exemplifies that XML semi-structured data and Semantic Web data differ in various ways, complicating the conversion of the formats in either direction and impeding the use of a query language specialized on only one of the formats for accessing both. On the one hand, XML data can only be unnaturally represented as RDF, because (1) the order of outgoing edges in RDF is irrelevant, (2) nodes are uniquely identified by URIs except for literals and blank nodes, (3) RDF does not support the concept of attributes. On the other hand, XML cannot naturally comprehend RDF data, in that (1) besides nodes also the edges of RDF graphs are labeled, (2) RDF is truly graph structured, and (3) RDF graphs need not be connected and are unrooted. In this section all of these differences are discussed and it is illustrated that although Xcerpt's term and graph oriented data model allows the representation also of RDF data in a very straight-forward way.

11.2.1 Graph Data Model and References

One of the most striking differences between Semantic Web data and XML is that XML does not allow multiple parent nodes for the same XML element and must be considered tree structured under this consideration. This is why most XML query languages such as XPath and XQuery provide a tree data

model. Taking the special attributes `id` and `idref` into account, XML may also be viewed as a graph structure. When querying XML it may sometimes even be useful to consider these XML references as true parent-child relationships. In contrast, Semantic Web data is truly graph structured, in that predicates are the only way of specifying relationships amongst resources, and nodes of an RDF graph may very well have multiple incoming edges. RDF graphs are usually represented by triples without any explicit references. Nevertheless, a graph structure is implied by these triples, because RDF references are implicit in that they exploit the fact that RDF resources are uniquely identified by URIs.

Whereas for XML query languages, such as XPath, XQuery and XSLT a tree data model is a natural choice, versatile query languages that incorporate also Semantic Web data must adopt a graph data model.

From the beginning Xcerpt was designed to not only handle XML data, but also semi-structured graph data, which means that it can be adapted to natively handle Semantic Web data easier than other XML query languages.

11.2.2 Labeled Edges

Put simply, the XML data model is a node-labeled tree. In contrast, RDF graphs are not only node-labeled, but also edge-labeled. In XML serializations of RDF graphs such as RDF/XML, this difference is overcome by “striped” XML, which means that element nodes representing RDF nodes and edges alternate in the nested XML serialization. The *Syntactic Web Approach* suggests querying RDF serializations with XML query languages. This solution is unsatisfactory in various ways: (1) It is not coherent with the visual and intuitive representation of RDF data as graphs, and is thus more difficult to grasp. (2) It does not pay tribute to the different roles assumed by subjects, objects and predicates of the RDF graph, which complicates e.g. the determination of the set of all predicates of an RDF graph. (3) Many XML serializations (such as RDF/XML and RDF/A) offer a great amount of variability and syntactic sugar for representing RDF graphs, which makes the formulation of queries against such serializations in XML query languages cumbersome.

As a result, a truly versatile query language for the Web must offer a data model that comprehends both: node- and edge-labeled graphs as well as purely node-labeled graphs. As has been mentioned before, node- and edge-labeled graphs can be transformed into graphs without edge labels in a straightforward manner. Nevertheless, the user must be provided with a syntax (see Section 11.3) that clearly distinguishes between edge- and node-labels both in query constructs and in the data.

11.2.3 Incomplete and Unbounded Data

In the Semantic Web, resources are uniquely identifiable, and thus anybody is free to make statements about resources by simply referencing the unique URI as subject, predicate or object within one’s own statements. A consequence of this ability for everyone to make statements about arbitrary resources is that one may never be sure to be aware of all statements made about a given resource (this is why RDF data can be considered inherently *incomplete*). From a graph perspective on Semantic Web data, this means that collecting all existing outgoing edges of a resource is not possible, which is a fundamental difference to XML data, where the sequence of children of an element node is fixed and can be determined simply by looking at the document containing the node in question.

A possible solution (which also yields other benefits) to this problem is to restrict one’s attention to the contents of specific documents or groups of statements, which are often referred to as *Named*

Graphs. “Named graphs is the idea that having multiple RDF graphs in a single document/repository and naming them with URIs provides useful additional functionality built on top of the RDF Recommendations.”² In fact, RDF query languages such as SPARQL and TriQL provide constructs for handling and constructing named graphs.

The above observations show that the data model for a Semantic Web query language must be able to express both complete (in form of named graphs or documents) and incomplete data (information that does not belong to any graph). While conventional Xcerpt query terms may already be complete and incomplete in breadth, data terms have always been considered to be complete. As shown in section 11.4 data terms can be naturally extended to include incomplete data, and an extended operational semantics that takes this extension of the data model into account is being considered.

11.2.4 RDF Graphs as Xcerpt Data Terms

While in semi-structured data, there is always a distinguished top level term, the root, Semantic Web data does not have the concept of top level terms. Furthermore, it may not even be possible to single out a resource from which all other resources are reachable over edges in the graph, because RDF graphs may consist of disconnected subgraphs. It is, however, possible to determine a set of resources, such that each resource in the graph is reachable from at least one of them. Choosing these resources as top level nodes, RDF graphs are very conveniently represented by sets of Xcerpt data terms.

11.2.5 Order of Sub-Terms

Another difference between RDF and XML data illustrated in Figure 11.1 is that RDF data usually does not impose an order on outgoing edges of a node. To be more precise, RDF data is always unordered unless otherwise specified by the use of an `rdf:Seq` sequence container. Hence, the data model must be able to represent both ordered and unordered information. The distinction between ordered and unordered data is especially useful in the scope of positional queries against semi-structured data as exemplified in Section 11.4.

Xcerpt data terms have been conceived to not only represent XML data, but also semi-structured data in general. Therefore Xcerpt already supports the concept of unordered sets of children unlike most other XML query languages and does not need to be adapted to the Semantic Web in this respect.

Summing up the particularities of XML and RDF data, the data model must support possibly cyclic and disconnected graphs with labeled and unlabeled edges, complete and incomplete data specifications, ordered and unordered child elements, implicit and explicit references, and finally multiple roots.

11.3 An Intuitive Syntax for Versatile Web Query Languages

In previous work [?], we have shown that due to its versatility gained from construct-query-rules and constructs for treating heterogeneous data, Xcerpt is particularly well-suited to handle XML serializations for the Semantic Web data formats RDF and Topic Maps such as RDF/A, RDF/XML and XTM. An obvious alternative to processing XML serializations of Semantic Web formats is their direct treatment. In fact, for Xcerpt’s users it may be more convenient to use a syntax that better distinguishes between edges and nodes within an RDF graph. In this section, we propose a possible

²<http://www.w3.org/2004/03/trix/>

syntax derived from the syntax of Xcerpt data terms that represents RDF data in a very similar way to XML data.

Listing 11.1: The RDF Graph of Figure 11.1 represented as an Xcerpt data term

```
geo:countries/Montenegro{
2  <geo:bordersOn> _:country1{
    <geo:bordersOn> geo:countries/Montenegro,
4    <rdfs:label> literal('Albania'),
    }
6  <rdfs:label> literal('Montenegro'),
  <rdf:type> geo:country,
8 }

10 _:country2 {
  <rdfs:label> literal('USA'),
12 <rdf:type> geo:country
}
```

In listing 11.1 edges (predicates) of the RDF graph in figure 11.1 are enclosed by angle braces and appear in between the elements (subjects and objects) that stand for the nodes of the graph. This syntax eases the authoring and understanding of queries considerably, because subjects, predicates and objects are much more easily distinguished.

As has been mentioned above, data with labeled edges may be transformed to graph structured data with unlabeled edges by the introduction of an additional node for each edge. This approach has already been used to query Semantic Web data with Xcerpt in [?]. A graph data model with labeled edges can be offered to the user by the internal and automatic transformation of both RDF query and data graphs to graph data with unlabeled edges, which can already be handled by Xcerpt. In this article it is argued that the user of a versatile query language should be unconscious of and not be confronted with this transformation.

11.4 Common Query Constructs for the Web and the Semantic Web

Schema information often being unavailable, data on the Web is very heterogeneous. But even if schema information is present, it usually leaves room for variability. In contrast to relational database query languages, Web query languages must therefore provide constructs for handling this heterogeneity.

Besides querying Semantic Web data, programmers are also interested in transforming it. An example scenario for one such transformation is the collection of data from different sources, and its rearrangement according to a joint schema.

Xcerpt has been designed as a declarative language rooted in logic programming. This section shows that Xcerpt's approach to querying, transforming and reasoning is well-suited not only for ordinary semi-structured data, but also for the Semantic Web.

11.4.1 Query Patterns and Answer Closedness

One of the design principles of SPARQL and Xcerpt is answer closedness. This principle dictates that all answers to queries may themselves be used as queries. By ensuring similar syntaxes for

both the formulation of queries and the representation of data, answer closedness eases program understanding.

Using data terms as queries, it is just possible to check whether an RDF graph is entailed³ by the queried data, or whether a particular XML fragment is contained within a document. In order to extract parts of the data, queries must contain logical variables. Xcerpt query terms are data terms enriched by variables and a series of constructs for handling heterogeneous data. These constructs are just as useful in the Semantic Web as for ordinary XML data. Constructs for handling heterogeneity in Xcerpt include optional term selection, double braces for incompleteness in breadth and arbitrary length traversal path expressions.

One might be interested in all resources that represent countries directly or transitively bordering on Montenegro and their names. Assuming data of a similar form as in Figure 11.1, the following Xcerpt query in edge-labeled notation helps out:

Listing 11.2: An Xcerpt query term with constructs for handling heterogeneity

```
1 var Country →./.*/{  
  <rdf:type> geo:country{{ }},  
3 desc(<geo:borders0n> ./.*)*  
  <geo:borders0n>  
5   geo:countries/Montenegro{{ }},  
  optional <rdfs:label> var Name →literal(/.*/)  
7 }}
```

There are several noteworthy constructs in the above query term:

- *Variable Constraints.* In Line 1, the bindings for the variable `Country` is constrained to graphs matching the pattern following `→`.
- *Incompleteness in breadth.* The schema of data on the web is in many cases unknown. Therefore one might not know or even not care about the set of outgoing edges of an RDF node. Double curly braces are used in Xcerpt to indicate that the matched data may also contain additional siblings other than those specified by the query term.
- *Regular expressions for labels.* The logical variable `Country` in Listing 11.2 is supposed to be bound to all kinds of nodes within the queried RDF graph, no matter whether it is a blank node or a resource. The regular expression `./.*` matches arbitrary URIs and b-nodes. In order to match just blank nodes or resources, the keywords `b-node` and `resource` can be used.
- *Incompleteness in depth.* The resource r_1 matching with variable `Country` shall be directly or transitively connected over `geo:borders0n`-predicates with the resource `geo:countries/Montenegro`, which stands for Montenegro. The RDF nodes in between r_1 and `geo:countries/Montenegro` are of no interest, and therefore an *arbitrary length traversal path expression* containing a wild-card regular expression for the resources of the intermediate nodes is used in line three.
- *Optional sub-terms.* Labels for the resources r_1 are to be retrieved if present. In the absence of such a label the query is not intended to fail, but to simply restitute no binding for the variable `Name`. Making use of the keyword `literal` ensures that `Name` is only bound to literals, never to URIs.

³for a definition of RDF entailment see <http://www.w3.org/TR/rdf-mt/>, Section 3.2

Solutions to Xcerpt queries are given in the form of substitution sets, which are sets of mappings from the logical variables in the query to subgraphs of the data. The query in Listing 11.2 applied to the RDF graph in Figure 11.1 yields the following substitution set:

$$\{ \{ \text{Country} \mapsto _:\text{country1}\{ \dots \}, \text{Name} \mapsto \text{'Albania'} \} \}$$

The fact that the variable `Country` is bound to the entire subgraph rooted at the resource it matches differentiates Xcerpt from other query languages such as SPARQL and RQL. Since in densely connected RDF graphs, the bindings of variables may contain large sub-graphs of the data or even the whole data graph, Xcerpt provides a second kind of variables called *label variables* which are not bound to entire subgraphs but only to the nodes they match with. The usage of a label variable in Listing 11.2 would be syntactically indicated by directly prefixing the double curly braces in Line 1 by the variable `var Country`.

Note that also SPARQL provides a way to return more information (entire subgraphs) about resources than just their URIs through the keyword `describe`. The exact nature of such descriptions is left unspecified by the SPARQL working draft, but the Concise Bounded Description⁴ proposed by Nokia is mentioned as an example.

The semantics of the query in Listing 11.2 is implicitly defined by mapping the node-and-edge-labeled syntax of the query to purely node-labeled query terms.

11.4.2 Injectivity and Querying RDF Sequences

When specifying a query term to be matched with semi-structured data, the semantics intended by the query author is usually that sibling nodes shall *not* match with the same node of the queried graph. *Matching* a query term q and its children q_1, \dots, q_n with a data term d and its children d_1, \dots, d_m can be formalized by a function $m: \{q_1, \dots, q_n\} \rightarrow \{d_1, \dots, d_m\}$. We demand m to be *injective* to reflect the authors intention. Listing 11.3 shows a query selecting all pairs of countries bordering on Montenegro, and Xcerpt's semantics⁵ ensures that the variables `Country1` and `Country2` are not bound to the same node. Note that formulating a query that allows the bindings for `Country1` and `Country2` to be the same can be easily expressed using Xcerpt's and connective for queries.

Listing 11.3: A query selecting all pairs of countries bordering to Montenegro

```
geo:countries/Montenegro{{
  <geo:borders0n> var Country1 →./.*/{ { } },
  <geo:borders0n> var Country2 →./.*/{ { } }
}}
```

As has been mentioned in Section 11.2.5 Semantic Web data can both be ordered and unordered. Xcerpt's positional approach to querying allows to match data dependent on the order of sub-terms. Figure 11.2 contains a possible representation of information about spoken languages in countries using an RDF sequence container.

The query in Listing 11.4 selects all countries in which Serbian is more common than Albanian assuming a schema as in Figure 11.2. The use of square brackets instead of curly braces indicates that the order of occurrence within the RDF sequence is relevant.

Listing 11.4: An Xcerpt query taking into account the order of subterms within an RDF-Container

```
var Country →./.*/{ {
```

⁴<http://swdev.nokia.com/uriqa/CBD.html>

⁵formally defined in [49, Chapter 8] at the aid of functions similar to m above

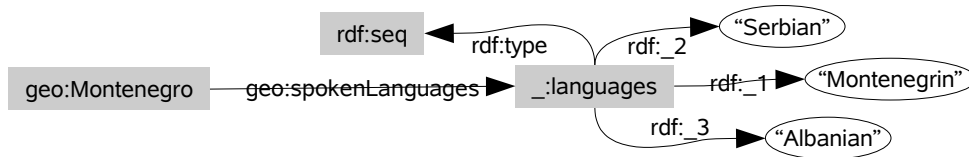


Figure 11.2: An RDF sequence containing the languages in the order of their diffusion in Montenegro

```
<geo:spokenLanguage> /*/[[
  <./.> literal('Serbian'),
  <./.> literal('Albanian')
]]
}}
```

11.4.3 Blank Node Treatment

Blank nodes (also called b-nodes) in RDF graphs are used to assert that a resource r_1 exists that is related with other resources in a certain way without associating a URI to r_1 . One unresolved issue related to querying RDF data containing b-nodes concerns the redundancy of answer sets. To see this reconsider the RDF graph from Figure 11.1.

Listing 11.5: A query selecting all resources of type `geo:country`

```
var Country → /*/{<rdf:type> geo:country}}
```

Selecting all resources of the graph that are of type `geo:country`, the query in Listing 11.5 cannot determine whether `_:country2` and `geo:countries/Montenegro` are meant to be the same concepts. Hence, the question arises, whether both the blank node `_:country2` and the resource identified by `geo:countries/Montenegro` should be returned or only the URI. The solution considered to be the most convincing by the authors is to exclude such query solutions that are entailed by other solutions, but to keep all others. The query in Listing 11.5 would therefore return both resources. In the case that the triple `(_:country2, rdfs:label, 'USA')` were not present, returning the blank node of the graph would be redundant.

11.4.4 Negation and Breadth-Complete Queries

As has been discussed in Section 11.2.3, Semantic Web data must be considered as inherently incomplete and unbounded in comparison to XML. Additionally taking into account that RDF statements are always positive assertions, the only sensible form of negation is scoped negation as failure, which has already been proven useful in the context of the Semantic Web[?, ?].

An approach that goes even beyond scoped negation as failure by providing explicit negative information to additionally enable strong negation is suggested in [?]. Although strong negation would certainly be helpful for Semantic Web Reasoning, it is not yet supported by Xcerpt.

While some Semantic Web query languages including the SPARQL family do not provide negation, XML query languages including Xcerpt usually do. To underline the importance of scoped negation in

the Semantic Web consider the following query issued against the resource `http://countries.org/country-information`.

Listing 11.6: Scoped negation as failure in Xcerpt query terms

```
1 in{ resource{ 'http://countries.org/country_information' },
  /.*/{{
3   <rdf:type> geo:country{{ }},
   <geo:bordersOn> geo:countries/Montenegro{{ }},
5   <rdfs:label> var Name →literal(/./),
   not(<geo:bordersOn> /.*/{{ <rdfs:label> literal('Albania') }})
7 }}},
}
```

Listing 11.6 queries the names of all countries bordering to Montenegro but not to Albania. Matching a term with both positive and negated sub-terms with a data term is carried out as follows: At first, it is tested, whether each of the positive query sub-terms can be associated with a matching sub-term of the data respecting the injectivity requirement mentioned in Section 11.4.2. If this matching succeeds, it is searched for a matching sub-term of the data for the negated sub-terms. If any of the negated sub-terms can be matched, the entire matching fails. If all positive sub-terms can be matched with the data, but none of the negated ones, the entire matching succeeds. The query semantics for node- and edge-labeled query and data terms as needed for RDF data is ascribed to the semantics of purely node-labeled terms as described in [49, Section 8.2] by straight-forward normalization rules transforming edge-labeled terms to purely node-labeled terms.

Breadth-complete queries are an issue which is closely related to negated sub-terms, because they can be rewritten as breadth-incomplete queries using the `without`-construct. They are indicated by single curly braces or brackets instead of double ones and can only be matched with data that does not contain any additional sub-terms besides those specified in the query term. To find countries that only border to Italy (and do not appear as subject in any other statement of the RDF graph), the query in Listing 11.7 could be used.

Listing 11.7: Breadth-complete queries against RDF data

```
in{ resource{ 'http://countries.org/country_information' },
2  var Country →././{{
   <geo:bordersOn> geo:countries/Italy{{ }},
4  },
}
```

In the same way as queries with negated sub-terms, breadth-complete queries must be scoped to a single or a set of named graphs.

11.4.5 Optional Sub-Terms

As exemplified in Listing 11.2, optional constructs are of great help for Semantic Web queries in that they allow to extract certain parts of the queried data only if they are present. Closer examination of the `optional` construct reveals that it is only syntactic sugar for a disjunction of queries. The query in Listing 11.2 could also be written using the Xcerpt `or` construct:

Listing 11.8: The same query as in Listing 11.2 without the `optional` construct

```
1 or (
  var Country →././{{
```

```

3   <rdf:type> geo:country{{ }},
   desc(<geo:borders0n> /.*/)* <geo:borders0n> geo:countries/Montenegro{{ }},
5   <rdfs:label> var Name →literal(/.*/)
   }},
7   var Country →/.*/{{
   <rdf:type> geo:country{{ }},
9   desc(<geo:borders0n> /.*/)* <geo:borders0n> geo:countries/Montenegro{{ }},
   without <rdfs:label> var Name →literal(/.*/)
11  }}
   )

```

As in SPARQL, multiple optional sub-terms may occur as siblings, or may even be nested. The semantics of such graph patterns seems to be straightforward at first glance: For each optional sub-term that succeeds to match, the bindings of its variables are included in the substitution set returned by the overall graph pattern. The failed matching of an optional sub-term does not prevent the overall graph pattern from returning a substitution set, which simply does not contain bindings for the variables in the unmatched optional sub-terms. Since variables may – and often do – occur multiple times in a query pattern, they may also be shared among multiple optional sub-terms, causing interdependencies among them. In particular, it may happen that only one of two optional sub-terms may be matched, but not both. While the SPARQL working draft does not define which of the sub-terms is to be picked, Xcerpt adopts the following convention: If multiple optional sub-terms impede each other from matching, all selections of these sub-terms are chosen that maximize the number of variable bindings.

11.5 From Queries to Transformations

While most Semantic Web query languages are limited to querying and returning sets of mappings of their variables to resources, Xcerpt – and to some extent also SPARQL – are designed to do more: by providing construct terms (in SPARQL they are called graph templates) to be filled with the variable bindings gained from the evaluation of queries, they allow the construction of results having an entirely different schema. This combination of querying and construction in so-called *construct-query-rules* (see Section 11.5.1 for details) gives rise to the possibility of complex transformations.

11.5.1 Construct-Query-Rules and User Defined Reasoning

The evaluation of Xcerpt query terms and SPARQL graph patterns against RDF data yields substitution sets. Xcerpt construct terms are Xcerpt data terms enriched by variables as place holders and grouping constructs like `all` and `some`. Substitutions are applied to construct terms by replacing the variables in the construct term by their bindings in the substitution set (for the detailed semantics see [49, Section 7.3.3]). Query and construct terms are combined by so-called *construct-query-rules*, which allow sophisticated user-defined reasoning which goes beyond the predefined rules of RDFS and OWL.

11.5.2 Grouping Constructs

A major difference between SPARQL graph templates and Xcerpt construct terms is that only the latter allow merging of substitution sets (called result sets in SPARQL) by using grouping constructs. Merging substitution sets is necessary because often the need arises to collect variable bindings from

different matches of the query pattern with the data. In contrast, a query result form within a SPARQL query is always filled exactly as often as the graph pattern in the WHERE clause matches with the queried RDF graph.

Reconsidering the information about countries and languages as exemplified in Figure 11.2, one might wish to construct an RDF graph that groups countries according to the languages which are spoken in them. To be more precise, for each language a blank node shall be constructed carrying an `rdfs:label` such as “Albanian”, “Serbian”, etc. Moreover the blank node must feature outgoing `geo:spokenIn` edges for each country that the language is spoken in.

Listing 11.9: Grouping countries according to languages

```
CONSTRUCT
2  _:language{
    <rdfs:label> var Language,
4  all <geo:spokenIn> var Country }
FROM
6  var Country →/.*/{{
    <geo:spokenLanguage> /.*/{{ </.*> var Language }} }}
8 END
```

Using the grouping construct `all` (line 4), the query in Listing 11.9 collects all bindings for the variable `Country` that are contained within a substitution set for a fixed binding of variable `Language`. An important issue to note is that – just as in SPARQL – although the name `_:language` of the blank node in Line 2 is constant, a new blank node is constructed for each binding of the variable `Language`.

11.5.3 Versatile access to XML and RDF

Integrated access to different data formats includes the requirement that data should be easily transformed from one format to the other, and that different formats are queried simultaneously. As an exemplary use-case imagine that information about bordering countries is available in XML format structured similarly to that in the left part of Figure 11.1, and that information about languages spoken in these countries is only available in RDF format as in Figure 11.2.

The query in Listing 11.10 extracts all those pairs of border-countries whose citizens understand each other, because they speak the same language. The query part of the rule is a conjunction of two query terms, the first one querying the XML resource, and the second one drawing information from an RDF file. The names of countries sharing a common border are found by comparing the values of the `id` and `idref` attributes with a value join over the variable `ID` (in Xcerpt, XML attributes are enclosed in parentheses; double parentheses indicate that there may be additional unspecified attributes). Similarly, pairs of countries which have the same most common language are selected by a join over the variable `Language`.

The query uses both constructs that are peculiar to either RDF or XML – such as variables for XML attribute values and edge-labeled query terms – and constructs that are applicable to both – such as complete and incomplete query term specifications. Notice that the variables `Name1` and `Name2` are shared among both conjuncts, which would be cumbersome to implement with two specialized languages for RDF and XML.

Listing 11.10: Versatile access to Web data Formats in Xcerpt

```

CONSTRUCT
2  result[ all understanding-neighbors[ var Name1, var Name2 ] ]
FROM
4  and (
      in{ resource{ 'http://geo.org/countries.xml' },
6      Countries {{
          Country((var ID →id)){ Name{ var Name1 } }},
8      Country{{
          borderCountry((var ID →idref)),
10     Name{ var Name2 } } } }},
      in{ resource{ 'http://geo.org/languages.rdf' },
12     /.*/{{
          <rdfs:label> var Name1,
14     <geo:spokenLanguage> /.*/{{ <rdf:_1> var Language } } }},
          /.*/{{
16     <rdfs:label> var Name2,
          <geo:spokenLanguage> /.*/{{ <rdf:_1> var Language } } } } }
18 )
END

```

11.6 Conclusion and Outlook

Due to its graph data model, its rule-based nature and its convenient constructs for handling heterogeneity, Xcerpt turns out to be very well-suited not only for XML, but also for Semantic Web querying, transformations and reasoning. RDF data being increasingly made available as descriptive meta-data for HTML and XML documents, versatile access to both meta-data and XML in the same query program becomes ever more important for the next generation of web applications such as specialized search engines, and online booking and library systems. Developing such applications can be strongly eased by providing a query language that does not restrict itself to one of the formats, but provides integrated access to all of them, freeing the programmer from the burden of learning and combining multiple languages.

Besides laying the foundation for effective query authoring, a versatile query and reasoning language must process query programs efficiently in order to gain strong acceptance throughout the Web community. Several challenges are related to efficient query processing, demanding future work in the domain of Xcerpt.

- Efficient parsing of semi-structured data from various serializations and efficient construction of in-memory graph representations of the data. Besides parsing documents, in-memory graph representations must also be efficiently constructed from relational RDF stores.
- Efficient simulation unification of query patterns with graph data and construct terms. A large amount of research has been carried out in this direction concerning primarily tree queries, but also graph queries [?].
- Efficient backward chaining evaluation of programs. A forward chaining evaluation of Xcerpt programs is less reasonable because (a) the set of facts of an Xcerpt program can be very large, (b) the major part of derived facts may be irrelevant to the query, and (c) Xcerpt programs may have infinite fixpoints if they contain recursive rules.

Chapter 12

Open Issues: Language Constructs

The previous sections define the syntax for the language Xcerpt separated in (a) literal structures, (b) data, construct, and query terms, and (c) programs, goals, rules, and data blocks. Where appropriate, open issues are referenced in these sections. The remaining sections list those open issues and give a brief discussion. It is expected that these open issues are gradually resolved and their resolution integrated into the language Xcerpt.

Note, that these open issues do, by and large, not affect the core of the language but rather present several “convenience” or extensions of the basic language.

This chapter is structured into issues related to general language constructs (Section 12.1), into issues on specific data representation formats such as XML or RDF (Section 13), into issues specific to one of the syntaxes for Xcerpt introduced above (Section 14), and finally into language extensions (Section ??).

12.1 General Issues

12.1.1 Defaults and Default Modes

Issue 1. Absent attribute and children lists and other parts of a term specification _____

A number of parts in term specification are optional, i.e., may or may not occur or may or may not be empty. These include:

- the lists of attributes and children,
- the namespace and identifier of a term,
- various parts of a XML document specification, and
- comments and processing instructions.

Currently, attribute and children lists may not be omitted, even if they are empty. Namespaces and term identifiers may be omitted, but where no namespace in a query is equivalent to an empty namespace, missing term identifiers do not affect matching. No special consideration is given to

comments and processing instructions, i.e., e.g., a total query term requires that a matching data term contains not only the same children as the total query term but also the same comments and processing instructions.

Solution Proposal(s): A more “unified” policy for defaults should be enacted. In particular, it might be worth considering “default modes” that the programmer can control. Similar to (rule-level) declaration blocks, a programmer could specify that in the scope of the mode declaration, e.g., omitted attribute lists are to be considered as `()` or as `(())`.

Resolution: This issue is merely related to the language syntax and the Xcerpt parser/compiler. The implementation of the language’s abstract machine is not affected by this choice. Therefore, we can afford to leave this issue open for the moment.

12.2 Construct Specific Issues

12.2.1 Conditional Construction and optional Terms

Issue 2. Conditional Construction

Optional construct terms allow one form of conditional construction, viz. where the condition is that bindings for all optional variables of the optional construct term exist. The **with-default** clause plays a role similar to the `else` clause in `if ... then ... else` expressions.

However, this is a very limited form of conditional construction that also exhibits some anomalies:

1. **optional** is the only modifier allowed in query and construct terms, though with a slightly different meaning. E.g., an optional ground term in a query term may or may not occur in the data, but an optional ground term in a construct term always occurs in the result.
2. It allows only implicit specification of optional variables (in contrast to grouping), i.e., it is not possible to make the optional part dependent on a variable not occurring in the result. This makes it difficult to express the following query:

```

1 for $x in doc("books.xml")/bookstore/book
  return if ($x/@category="CHILDREN")
3     then <child>{data($x/title)}</child>
     else <adult>{data($x/title)}</adult>

```

3. When the choice depends on the value of an optional variable this can be expressed in a condition box, but only in the case of exactly two choices, not if there are more alternatives.

CONSTRUCT

```

2 result [
   optional ( child(opt from-year=var Year)[ var Title ] ) with-default ( adult [
     var Title ] )
4 ]

```

FROM

```

6 bookstore [[
   book (from-year=opt var Year)[[ var Title →title [[]]]
8 ]] where opt (var Year > 25)

```

END

Solution Proposal(s): A proposed solution is the introduction of a `if ... then ... else` or case expression into construct terms, that allows for conditions as in **where** clauses. The requirement that matching is never affected by the construct part of a rule should be upheld.

Resolution: For Xcerpt 2.0's implementation the above proposal is adopted. On the syntax level, we continue to experiment with different approaches.

Issue 3. Optional for Non-Term Variables

Occurrence modifiers (**without** and **optional**) are currently only allowed on entire term expressions. However, namespaces and term identifier are also non-mandatory parts of a term, that might be optionally queried. It is currently not possible to query, e.g., for all nodes without term identifiers.

Solution Proposal(s): For namespaces this issue could be resolved by referring to the empty namespace (represented by the empty string). However, this introduces another form of “optionality” into the language.

Alternatively, **optional** and **without** could also be allowed on non-term expressions (labels, namespaces, etc.). Then a careful consideration where is needed (e.g., document specifications).

This issue is related to Issue 1.

Resolution: Not yet reached.

12.2.2 Query Formulas as Subterms

Issue 4. Query Formulas as Subterms

Currently, query formulas (expressions using **and**, **or**, and **not**) are only allowed at the top-level of query terms. However, programs become both more compact and easier to read, as well as easier to efficiently evaluate, if formulas are allowed at sub-term level.

However, this introduces a number of questions:

- What becomes then the different between **not** and **without**? If **not** is allowed at sub-term level, **without** seems to be superfluous.
- What is then the difference between $t[[a,b]]$ and $t[[and(a,b)]]$? Only that the injectivity constraint between a and b is lifted? Essentially, children are already connected by implicit “**and-injective**” or “**and-ordered**”.

Solution Proposal(s): *A detailed proposal is needed.*

Resolution: The implementation of Xcerpt 2.0 allows arbitrary nesting of boolean formulas in Xcerpt terms with configurable semantics. On the syntax level, experimentation continues.

12.2.2.1 withouts as Direct Siblings

Issue 5. Without Siblings

If several direct siblings in an ordered term or arbitrary siblings in an unordered term are modified by **without**, the semantics of the expression becomes unclear: E.g., matches $f[[a, \text{without } b, \text{without } c, d]]$ with $f[a, c, b, d]$ or not? In other words how are order and injectivity constraints enforced between **without**-modified terms.

Solution Proposal(s): We allow multiple directly consecutive **without**'s with “or” semantics. If sequence semantics (i.e., without a b followed by a c) is intended a single **without** may include multiple query terms.

Resolution: Accepted, however implementation of this resolution in grammar and translation is still open.

12.2.3 Functions and Libraries: Built-In and User-defined

Issue 6. Relational vs. functional Operators

Xcerpt does not have an extensive function library as of now. Introducing such a library requires great care, in particular when considering not only functional operators (such as compare, concat, or arithmetic operators), but also relational (sequence-valued) operators (such as tokenizers).

Solution Proposal(s): The latter are more similar to predicates (rules) where certain parameters are consuming, others are defining. The addition of rules where parameters are specifically marked as consuming only would solve these cases. Functional operators could also be handled this way, but differ in where they may be used. Functional operators are also useful in construct terms and in condition expressions.

Considering the sort of functions and operators to support, the XQuery and XPath function and operator library [42] is certainly a good starting point.

Resolution: A more detailed proposals on functions and a basic function library is under development, see Deliverable I4-D13, which describes also its implementation for Xcerpt 2.0

Issue 7. Expressions in Conditions

Optional modifiers are currently only allowed on variables in conditions, but should actually range over entire expressions.

Resolution: Accepted.

12.2.4 Variables

Issue 8. Variable Restrictions for Identifiers and Literal Content

Following [49], the text currently allows variable restrictions only on structured terms and attribute terms. This disallows in particular variable restrictions on label variables. The latter ones are useful just as variable restrictions on literal content, if regular expressions are used to restrict the set of labels or content nodes (or namespaces or any other place where an identifier may occur).

Solution Proposal(s): Note, that they are not strictly necessary due to variables in regular expressions, i.e., instead of writing *var X →/⟨reg-exp⟩/* one may use */⟨var X →⟨reg-exp⟩⟩/*.

Resolution: Not implemented in Xcerpt 2.0. Might be added later as convenience.

12.2.5 Varia

Issue 9. Explicit Variable Specifications for Except

Except is defined to affect the bindings of all variables in restrictions for which it occurs. In the following query term bindings for both X and Y are affected (i.e., the c sub-term is excluded from bindings for both):

```
root [[ var X →a [[ var Y →b [[ except c [[ ]] ] ] ] ] ] ]
```

Against the data term *root[a[b[c]]]* this results in one binding for X, viz. *a [b []]* and one binding for Y, viz. *b []*.

This is problematic as it makes impossible nested variable restrictions where one excludes some sub-term and another does not. In the example, it is not possible to affect the bindings to Y without affecting the bindings for X.

Solution Proposal(s): Introducing explicit “variable specifications” for **except**, i.e., an explicit list of variables in whose bindings the sub-terms in the scope of the **except** modifier are removed.

Bindings of all other variables remain unaffected and must contain all sub-terms in the scope of the **except** (i.e., for them the **except** is simply ignored).

Resolution: Xcerpt 2.0 currently does not implement **except** due to its severe effects on the complexity of query evaluation.

Issue 10. _____

Are goals evaluated in a particular order? What if one goal modifies data used in another? What if two goals write to the same output?

Resolution: Xcerpt 2.0 uses order in file and creates output for all goals at the end of the evaluation.

Issue 11. Document Specifications _____

Currently, document specifications are specifically treated in the grammar, which makes the grammar rather bloated. It could be better to consider them as a canonical “transformation” to terms. However, this would make the enforcement of constraints such as all variables in document specifications are consuming only more difficult.

Resolution: None.

12.3 Querying the Type of Data, Typed Accessors

There are a number of issues related to an upcoming Xcerpt type system.

Issue 12. Typed Accessors and Coercion _____

For terms, it should be possible to access (a) the actual structure, (b) the typed value of the term, if it has any, and (c) the string value of the term (defined, e.g., as in XPath).

Resolution: An extension for typed Xcerpt is under development. Details are going to be published in co-operation with working group I3 in 2007.

Issue 13. Typing Data Terms _____

It should be possible to explicitly type data terms, e.g., to distinguish plain strings from strings representing date.

Resolution: See above.

Issue 14. Querying Typed Data _____

It should be possible to query data based on its type. This is particularly helpful, if the type is complex, as it avoids the need for complex (and difficult to evaluate) patterns in these cases.

Resolution: See above.

12.4 Collapsing Text Nodes

Issue 15. Consecutive Text Nodes Collapse _____

Currently, consecutive text nodes are (silently) collapsed at construction. This is in accordance to the XML data model. Is this the desired behavior? Should we allow consecutive text nodes and provide an explicit concatenation?

Resolution: The implementation of Xcerpt 2.0 allows multiple consecutive text nodes. A mode (see Issue 12.1.1) is under consideration for switching between collapsing and non-collapsing behavior.

Chapter 13

Open Issues: Specific to Data Representation Format

13.1 Serializing to XML and from XML

Issue 16. Serializing to XML

Similar to the XML document specification in query terms, there should be a document serialization specification in goals describing how to serialize the resulting term. It might be worth considering the adoption of the XQuery/XSLT Serialization Recommendation [37]. Additionally, a canonical representation of graph structures, non XML labels, contents of comments or processing-instructions that are not XML conform, unordered elements, adjacent text nodes (if allowed, cf. Issue 15) must be defined.

Resolution: We adopt the mechanisms discussed in XQuery/XSLT Serialization Recommendation [37].

Issue 17. Accessible Encoding of XML Documents

Currently, the XML document specification does not allow access to the encoding of the file in accordance to the XML Information Set recommendation.

Resolution: Not accessible in Xcerpt 2.0.

Issue 18. XML Base

Xcerpt does not support the XML Base specification [43], i.e., no base URL at elements as in the XML Information Set.

Consider the extraction of some element at a non-root level. Now relative references are not any longer resolvable as the connection to the root-level `xml:base` attribute is lost.

Resolution: Not supported in Xcerpt 2.0, but query authors can write rules that “materialize” XML base attributes according to [43].

Issue 19. In-scope Namespaces

There are a number of issues related to namespaces in XML in general and Xcerpt in particular:

1. exactly what strings are permitted as namespace URIs? Different W3C specifications differ on this point.

2. Are namespace declarations information-bearing? For example, are the two documents below equivalent: (1) `<a xmlns:x="x">` (2) `<a xmlns:x="x"><b xmlns:x="x"/>`
3. Are in-scope namespaces that are not referenced information-bearing? For example, are either of the above documents equivalent to:
 - (3) `<a>`?

The main problem here, of course, is "QNames in content": the use of namespace-sensitive element and attribute values. But there are also applications that use the mere presence of a namespace declaration as a flag or marker.
4. Are prefixes information-bearing? That is, is document (1) equivalent to: (4) `<a xmlns:y="x">`
Again, the main problem is "QNames in content".
5. In the light of the above, how should namespaces be handled by applications that allow a document to be modified? For example, if an element is deep-copied from one place to another, should it take all its in-scope namespace declarations with it?

Indeed the Canonical XML recommendation *dropped even namespace rewriting* for precisely these reasons: "The C14N-20000119 Canonical XML draft described a method for rewriting namespace prefixes such that two documents having logically equivalent namespace declarations would also have identical namespace prefixes. The goal was to eliminate dependence on the particular namespace prefixes in a document when testing for logical equivalence. However, there now exist a number of contexts in which namespace prefixes can impart information value in an XML document. For example, an XPath expression in an attribute value or element content can reference a namespace prefix. Thus, rewriting the namespace prefixes would damage such a document by changing its meaning (and it cannot be logically equivalent if its meaning has changed)."

Notice, that currently Xcerpt supports only namespace URIs in patterns. This is sufficient, as namespace prefixes are considered not information carrying. However, since the introduction of QNames in numerous W3C recommendations for XML applications (XML Schema, XSLT, etc.) in-scope namespace prefixes **are** information carrying, as they are needed to find the namespace of a QName.

Resolution: Namespaces *and* in-scope namespaces are considered information bearing and thus preserved in the current implementation of Xcerpt 2.0. However, multiple declarations of the same prefix in the same scope are (in accordance to Canonical XML and XQuery) not preserved.

Chapter 14

Open Issues: Specific to Concrete Syntax

14.1 Non-XML Term Syntax

Issue 20. Nested Comments in Non-XML and XML-style Term Syntax

Currently, neither syntax allows nested comments. Though nested comments are commonly shunned in programming languages, there are some recent languages (REXX, Haskell, XQuery) that use nested comments. For a comparison of comments in programming languages cf. <http://www.gavilan.edu/csis/languages/comments.html>.

Resolution: In Xcerpt 2.0, nested comments are not allowed. (No change resolution)

Issue 21. IRIs in Recommended IETF Angle Bracket Notation

Currently, IRIs are not syntactically separated from strings. The IETF recommends URIs to be denoted in angle brackets, a notation used in many RDF formats. This clashes however with angle brackets in XML.

Resolution: In Xcerpt 2.0, IRIs are not syntactically separate from strings. (No change resolution)

14.2 XML-style Term Syntax

Issue 22. Quoting in XML-style Term Syntax

Currently, strings are quoted in the XML-style term syntax. However, there is a strong motivation to make XML documents cut-and-pastable, thus requiring the adoption of the XML character encoding rules into the XML-style Term syntax. In this case, the issue of entities must be considered. Also, keywords must be quoted in this case, possibly using character sequences illegal in XML.

Resolution: The XML-style term syntax is not yet supported in the current Xcerpt 2.0 implementation. (Unresolved)

14.3 Pure XML Syntax

No issues with the pure XML syntax have been identified so far.

Acknowledgements.

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REVERSE number 506779 (cf.<http://reverse.net>).

Part III

Full Language Grammars

Appendix A

Grammar for Non-XML Term Syntax

A.1 Literal Structures

<i><NCName></i>	::= <i><http://www.w3.org/TR/REC-xml-names/#NCName></i>
<i><IRI></i>	::= <i>'" <http://www.ietf.org/rfc/rfc3987.txt#IRI> "'</i>
<i><String></i>	::= <i>'" <StringCharacter>* "'</i>
<i><StringCharacter></i>	::= <i><http://java.sun.com/docs/books/jls#StringCharacter> <Line-feed> <Carriage-return></i>
<i><Line-feed></i>	::= <i>'\{ }u000a'</i>
<i><Carriage-Return></i>	::= <i>'\{ }u000d'</i>
<i><Number></i>	::= <i><int></i>
<i><Int></i>	::= <i><http://www.w3.org/TR/REC-xml-names/#Digit>*</i>
<i><Regex></i>	::= <i>'/ { http://www.unix.org/version3/ieee_std.html#extended_reg_exp } /'</i>
<i><ERE_expression></i>	::= <i><http://www.unix.org/version3/ieee_std.html#one_char_or_coll_elem_ERE></i> <i> '^'</i> <i> '\$'</i> <i> '(' <http://www.unix.org/version3/ieee_std.html#extended_reg_exp> ')'</i> <i> '(' <variable> '->' <http://www.unix.org/version3/ieee_std.html#extended_reg_exp> ')'</i> <i> <http://www.unix.org/version3/ieee_std.html#ERE_expression></i> <i><{ http://www.unix.org/version3/ieee_std.html#ERE_dupl_symbol }</i>
<i><Whitespace></i>	::= <i>(<http://www.w3.org/TR/REC-xml#S> <End-of-line-comment> <Block-comment>)*</i>
<i><Comment-char></i>	::= <i><http://www.w3.org/TR/REC-xml#NT-Char></i>

<End-of-line> ::= <Line-feed> | <Carriage-return> (<Line-feed>)?
 <End-of-line-comment> ::= '#' (<Comment-char>* - (<Comment-char>* <End-of-line> <Comment-char>*)
 <End-of-line>
 <Block-comment> ::= '/#' (<Comment-char>* - (<Comment-char>* ('/' | '#') <Comment-char>*)
 '/'#

A.2 Data Terms

<data-term> ::= <term-level-declare-dt> | <reference-dt> | <structured-dt> | <content-dt>
 <reference-dt> ::= '^' <identifier-dt>
 <term-level-declare-dt> ::= 'declare' <ns-declaration-dt> '(' (<data-term> ',')* ')'
 <identifier-dt> ::= <NCName> | <IRI> | <String>
 <ns-declaration-dt> ::= (<ns-prefix-declaration-dt> ',')* (<ns-default-declaration-dt> ',')?
 (<ns-prefix-declaration-dt> ',')*
 <ns-prefix-declaration-dt> ::= 'ns-prefix' <identifier-dt> '=' <IRI>
 <ns-default-declaration-dt> ::= 'ns-default' <IRI>
 <content-dt> ::= <literal-content-dt> | <comment-dt> | <processing-instruction-dt>
 <literal-content-dt> ::= <String>
 <comment-dt> ::= 'xcerpt' ':' 'comment' '(' ')' '[' <literal-content-dt> ']'
 <processing-instruction-dt> ::= ('xcerpt' ':' 'processing-instruction' | 'xcerpt' ':' 'pi')
 '(' 'target-name' '=' <identifier-dt> ')' '[' <literal-content-dt> ']'
 <structured-dt> ::= <local-spec-dt> <children-list-dt>
 <children-list-dt> ::= '[' (<data-term> ',')* ']'
 | '{' (<data-term> ',')* '}'
 <local-spec-dt> ::= <term-identifier-dt>? <ns-label-dt> <attr-term-list-dt>
 <term-identifier-dt> ::= <identifier-dt> '@'
 <ns-label-dt> ::= (<identifier-dt> ':')? <identifier-dt>
 <attr-term-list-dt> ::= '(' (<attr-term-dt> ',')* ')'
 <attr-term-dt> ::= <base-attr-term-dt>

$\langle \text{base-attr-term-dt} \rangle ::= \langle \text{ns-label-dt} \rangle '=' \langle \text{literal-content-dt} \rangle$
 $\langle \text{top-level-data-term} \rangle ::= \langle \text{top-term-level-declare-dt} \rangle | \langle \text{structured-dt} \rangle$
 $\langle \text{top-term-level-declare-dt} \rangle ::= 'declare' \langle \text{ns-declaration-dt} \rangle '(' \langle \text{top-level-data-term} \rangle ')'$

A.3 Construct Terms

$\langle \text{construct-term} \rangle ::= \langle \text{term-level-declare-ct} \rangle | \langle \text{reference-ct} \rangle | \langle \text{structured-ct} \rangle | \langle \text{content-ct} \rangle$
 $\quad | \langle \text{term-variable-ct} \rangle$
 $\quad | \langle \text{modified-ct} \rangle$
 $\langle \text{reference-ct} \rangle ::= '^' \langle \text{identifier-ct} \rangle$
 $\langle \text{term-level-declare-ct} \rangle ::= 'declare' \langle \text{ns-declaration-ct} \rangle '(' (\langle \text{construct-term} \rangle ',?)* ')'$
 $\langle \text{identifier-ct} \rangle ::= \langle \text{NCName} \rangle | \langle \text{IRI} \rangle | \langle \text{String} \rangle | \langle \text{literal-variable-ct} \rangle$
 $\langle \text{ns-declaration-ct} \rangle ::= (\langle \text{ns-prefix-declaration-ct} \rangle ',?)* (\langle \text{ns-default-declaration-ct} \rangle ',?)*$
 $\quad (\langle \text{ns-prefix-declaration-ct} \rangle ',?)*$
 $\langle \text{ns-prefix-declaration-ct} \rangle ::= 'ns-prefix' \langle \text{identifier-ct} \rangle '=' (\langle \text{IRI} \rangle | \langle \text{literal-variable-ct} \rangle)$
 $\langle \text{ns-default-declaration-dt} \rangle ::= 'ns-default' (\langle \text{IRI} \rangle | \langle \text{literal-variable-ct} \rangle)$
 $\langle \text{content-ct} \rangle ::= \langle \text{literal-content-ct} \rangle | \langle \text{comment-ct} \rangle | \langle \text{processing-instruction-ct} \rangle$
 $\langle \text{literal-content-ct} \rangle ::= \langle \text{String} \rangle | \langle \text{literal-variable-ct} \rangle$
 $\langle \text{comment-ct} \rangle ::= 'xcerpt' ':' 'comment' '(' ')' '[' \langle \text{literal-content-ct} \rangle ']'$
 $\langle \text{processing-instruction-ct} \rangle ::= ('xcerpt' ':' 'processing-instruction' | 'xcerpt' ':' 'pi')$
 $\quad '(' 'target-name' '=' \langle \text{identifier-ct} \rangle ')' '[' \langle \text{literal-content-ct} \rangle ']'$
 $\langle \text{structured-ct} \rangle ::= \langle \text{local-spec-ct} \rangle \langle \text{children-list-ct} \rangle$
 $\langle \text{children-list-ct} \rangle ::= '[' (\langle \text{construct-term} \rangle ',?)* ']'$
 $\quad | '{' (\langle \text{construct-term} \rangle ',?)* '}'$
 $\langle \text{local-spec-ct} \rangle ::= \langle \text{term-identifier-ct} \rangle? \langle \text{ns-label-ct} \rangle \langle \text{attr-term-list-ct} \rangle$
 $\langle \text{term-identifier-ct} \rangle ::= \langle \text{identifier-ct} \rangle '@'$
 $\langle \text{ns-label-ct} \rangle ::= (\langle \text{identifier-ct} \rangle ':')? \langle \text{identifier-ct} \rangle$
 $\langle \text{attr-term-list-ct} \rangle ::= '(' (\langle \text{attr-term-ct} \rangle ',?)* ')'$
 $\langle \text{attr-term-ct} \rangle ::= \langle \text{base-attr-term-ct} \rangle$
 $\quad | \langle \text{term-variable-ct} \rangle$
 $\quad | \langle \text{modified-attr-term-ct} \rangle$

$\langle \text{base-attr-term-ct} \rangle ::= \langle \text{ns-label-ct} \rangle '=' \langle \text{literal-content-ct} \rangle$
 $\langle \text{top-level-construct-term} \rangle ::= \langle \text{top-term-level-declare-ct} \rangle | \langle \text{structured-ct} \rangle$
 $\langle \text{top-term-level-declare-ct} \rangle ::= 'declare' \langle \text{ns-declaration-ct} \rangle '(' \langle \text{top-level-construct-term} \rangle ')'$
 $\langle \text{variable-ct} \rangle ::= \langle \text{term-variable-ct} \rangle | \langle \text{literal-variable-ct} \rangle$
 $\langle \text{term-variable-ct} \rangle ::= \langle \text{var-specification-ct} \rangle$
 $\langle \text{literal-variable-ct} \rangle ::= \langle \text{var-specification-ct} \rangle$
 $\langle \text{var-specification-ct} \rangle ::= ('variable' | 'var')? \langle \text{NCName} \rangle$
 $\langle \text{modified-ct} \rangle ::= \langle \text{grouping-ct} \rangle | \langle \text{optional-ct} \rangle$
 $\langle \text{modified-attr-term-ct} \rangle ::= \langle \text{grouping-attr-term-ct} \rangle | \langle \text{optional-attr-term-ct} \rangle$
 $\langle \text{grouping-ct} \rangle ::= \langle \text{grouping-modifier} \rangle '(' ((\langle \text{construct-term} \rangle ',')^* ') \langle \text{groupby} \rangle? \langle \text{orderby} \rangle?$
 $\langle \text{grouping-attr-term-ct} \rangle ::= \langle \text{grouping-modifier} \rangle '(' \langle \text{attr-term-ct} \rangle? ')' \langle \text{groupby} \rangle? \langle \text{orderby} \rangle?$
 $\langle \text{grouping-modifier} \rangle ::= 'all'$
 $\quad | 'some' \langle \text{number-ct} \rangle$
 $\quad | 'first' \langle \text{interval-ct} \rangle$
 $\langle \text{orderby} \rangle ::= 'order-by' '(' (((\langle \text{optional-variable} \rangle | \langle \text{variable-ct} \rangle) ',')^* ') \langle \text{order-relation} \rangle?$
 $\langle \text{order-relation} \rangle ::= 'ascending' | 'descending' | \langle \text{NCName} \rangle$
 $\langle \text{groupby} \rangle ::= 'group-by' '(' (((\langle \text{optional-variable} \rangle | \langle \text{variable-ct} \rangle) ',')^* ') \langle \text{equivalence-relation} \rangle?$
 $\langle \text{equivalence-relation} \rangle ::= \langle \text{NCName} \rangle$
 $\langle \text{optional-variable} \rangle ::= \langle \text{optional-modifier} \rangle \langle \text{variable-ct} \rangle$
 $\langle \text{interval-ct} \rangle ::= \langle \text{number-ct} \rangle '-' \langle \text{number-ct} \rangle$
 $\quad | \langle \text{number-ct} \rangle '-'$
 $\quad | '+'$
 $\langle \text{number-ct} \rangle ::= \langle \text{Int} \rangle | \langle \text{literal-variable-ct} \rangle$
 $\langle \text{optional-ct} \rangle ::= \langle \text{optional-modifier} \rangle '(' ((\langle \text{construct-subterm} \rangle ',')^* ')$
 $\quad ('with-default' '(' ((\langle \text{construct-subterm} \rangle ',')^* ')')?'$
 $\langle \text{optional-attr-term-ct} \rangle ::= \langle \text{optional-modifier} \rangle '(' ((\langle \text{attr-term-ct} \rangle ',')^* ')$
 $\quad ('with-default' '(' ((\langle \text{attr-term-ct} \rangle ',')^* ')')?'$
 $\langle \text{optional-modifier} \rangle ::= 'optional' | 'opt'$

A.4 Query Terms

$\langle \text{query-term} \rangle$::= $\langle \text{modified-qt} \rangle$ $\langle \text{term-level-declare-qt} \rangle$
$\langle \text{term-level-declare-qt} \rangle$::= 'declare' $\langle \text{ns-decl-qt} \rangle$ $\langle \text{variable-decl-qt} \rangle$ '(' $\langle \text{query-term} \rangle$ ','?)* ''
$\langle \text{modified-qt} \rangle$::= $\langle \text{variable-term-qt} \rangle$ $\langle \text{location-modified-qt} \rangle$ $\langle \text{occurrence-modified-qt} \rangle$ $\langle \text{selection-modified-qt} \rangle$
$\langle \text{base-term-qt} \rangle$::= $\langle \text{reference-qt} \rangle$ $\langle \text{content-qt} \rangle$ $\langle \text{structured-qt} \rangle$
$\langle \text{reference-qt} \rangle$::= '^' $\langle \text{identifier-qt} \rangle$
$\langle \text{identifier-qt} \rangle$::= $\langle \text{NCName} \rangle$ $\langle \text{IRI} \rangle$ $\langle \text{String} \rangle$ $\langle \text{literal-variable-qt} \rangle$ $\langle \text{Regexp} \rangle$
$\langle \text{ns-decl-qt} \rangle$::= ($\langle \text{ns-prefix-decl-qt} \rangle$ ','?)* ($\langle \text{ns-default-decl-qt} \rangle$ ','?)? ($\langle \text{ns-prefix-decl-qt} \rangle$ ','?)*
$\langle \text{ns-prefix-decl-qt} \rangle$::= 'ns-prefix' $\langle \text{identifier-qt} \rangle$ '=' ($\langle \text{IRI} \rangle$ $\langle \text{literal-variable-qt} \rangle$)
$\langle \text{ns-default-decl-qt} \rangle$::= 'ns-default' ($\langle \text{IRI} \rangle$ $\langle \text{literal-variable-qt} \rangle$)
$\langle \text{variable-decl-qt} \rangle$::= (('variable' 'var') $\langle \text{NCName} \rangle$ ','?)*
$\langle \text{content-qt} \rangle$::= $\langle \text{literal-content-qt} \rangle$ $\langle \text{comment-qt} \rangle$ $\langle \text{processing-instruction-qt} \rangle$
$\langle \text{literal-content-qt} \rangle$::= $\langle \text{String} \rangle$ $\langle \text{literal-variable-qt} \rangle$ $\langle \text{Regexp} \rangle$
$\langle \text{comment-qt} \rangle$::= 'xcerpt' ':' 'comment' '(' '[' $\langle \text{literal-content-qt} \rangle$ ']'
$\langle \text{processing-instruction-qt} \rangle$::= ('xcerpt' ':' 'processing-instruction' 'xcerpt' ':' 'pi') '(' 'target-name' '=' $\langle \text{identifier-qt} \rangle$ ')' '[' $\langle \text{literal-content-qt} \rangle$ ']'
$\langle \text{structured-qt} \rangle$::= $\langle \text{local-spec-qt} \rangle$ $\langle \text{children-list-qt} \rangle$ $\langle \text{condition-clause-qt} \rangle$?
$\langle \text{children-list-qt} \rangle$::= '[' ($\langle \text{query-term} \rangle$ ','?)* ']' '{' ($\langle \text{query-term} \rangle$ ','?)* '}' '[' [$\langle \text{query-term} \rangle$ ','?)* ']' '{' { ($\langle \text{query-term} \rangle$ ','?)* '}'
$\langle \text{condition-clause-qt} \rangle$::= 'where' '(' $\langle \text{condition-qt} \rangle$ ')'

$\langle \text{condition-qt} \rangle ::= \langle \text{c-parameter} \rangle \langle \text{comparison-op} \rangle \langle \text{c-parameter} \rangle$
 $\quad | \langle \text{comparison-op} \rangle ' (\langle \text{c-parameter} \rangle \langle \text{c-parameter} \rangle) '$
 $\quad | \text{'and' } ' (\langle \text{condition-qt} \rangle \langle \text{condition-qt} \rangle +) '$
 $\quad | \text{'or' } ' (\langle \text{condition-qt} \rangle \langle \text{condition-qt} \rangle +) '$
 $\quad | \text{'not' } ' (\langle \text{condition-qt} \rangle) '$
 $\quad | \langle \text{c-parameter} \rangle$

$\langle \text{condition-op} \rangle ::= \text{'==' | '!=' | '<' | '>' | '<=' | '>='}$

$\langle \text{arithmetic-op} \rangle ::= \text{'+' | '-' | '*' | '/' | '^'}$

$\langle \text{c-parameter} \rangle ::= \langle \text{optional-variable-qt} \rangle | \langle \text{variable-qt} \rangle$
 $\quad | \langle \text{String} \rangle | \langle \text{Int} \rangle$
 $\quad | \langle \text{c-parameter} \rangle \langle \text{arithmetic-op} \rangle \langle \text{c-parameter} \rangle$
 $\quad | \langle \text{arithmetic-op} \rangle ' (\langle \text{c-parameter} \rangle \langle \text{c-parameter} \rangle) '$

$\langle \text{optional-variable-qt} \rangle ::= \langle \text{optional-modifier} \rangle \langle \text{variable-qt} \rangle$

$\langle \text{local-spec-qt} \rangle ::= \langle \text{term-identifier-qt} \rangle ? \langle \text{ns-label-qt} \rangle \langle \text{attr-term-list-qt} \rangle$

$\langle \text{term-identifier-qt} \rangle ::= \langle \text{identifier-qt} \rangle '@'$

$\langle \text{ns-label-qt} \rangle ::= (\langle \text{identifier-qt} \rangle ':') ? \langle \text{identifier-qt} \rangle$

$\langle \text{attr-term-list-qt} \rangle ::= ' ((\langle \text{attr-term-qt} \rangle ', ?) *) '$
 $\quad | ' (((\langle \text{attr-term-qt} \rangle ', ?) *)) '$

$\langle \text{attr-term-qt} \rangle ::= \langle \text{modified-attr-term-qt} \rangle$

$\langle \text{modified-attr-term-qt} \rangle ::= \langle \text{base-attr-term-qt} \rangle$
 $\quad | \langle \text{variable-attr-term-qt} \rangle$
 $\quad | \langle \text{occurrence-modified-attr-term-qt} \rangle$
 $\quad | \langle \text{selection-modified-attr-term-qt} \rangle$

$\langle \text{base-attr-term-qt} \rangle ::= \langle \text{ns-label-qt} \rangle \text{'=' } \langle \text{literal-content-qt} \rangle$

$\langle \text{variable-term-qt} \rangle ::= \langle \text{base-term-qt} \rangle$
 $\quad | \langle \text{term-variable-qt} \rangle \text{'->' } \langle \text{base-term-qt} \rangle ?$

$\langle \text{variable-attr-term-qt} \rangle ::= \langle \text{term-variable} \rangle \text{'->' } \langle \text{base-attr-term-qt} \rangle ?$

$\langle \text{variable-qt} \rangle ::= \langle \text{term-variable-qt} \rangle | \langle \text{literal-variable-qt} \rangle$

$\langle \text{term-variable-qt} \rangle ::= \langle \text{var-specification-qt} \rangle$

$\langle \text{literal-variable-qt} \rangle ::= \langle \text{var-specification-qt} \rangle$

$\langle \text{var-specification-qt} \rangle ::= \text{'variable' | 'var'} ? \langle \text{NCName} \rangle$
 $\quad | \langle \text{anonymous-variable} \rangle$

$\langle \text{anonymous-variable} \rangle ::= \text{'_}'$
 $\langle \text{selection-modified-qt} \rangle ::= \langle \text{selection-modifier} \rangle \text{'('} (\langle \text{modified-qt} \rangle \text{' ,? '* '})$
 $\langle \text{selection-modified-attr-term-qt} \rangle ::= \langle \text{selection-modifier} \rangle \text{'('} (\langle \text{modified-attr-term-qt} \rangle \text{' ,? '* '})$
 $\langle \text{selection-modifier} \rangle ::= \text{'except'}$
 $\langle \text{occurrence-modified-qt} \rangle ::= \langle \text{occurrence-modifier} \rangle \text{'('} (\langle \text{modified-qt} \rangle \text{' ,? '* '})$
 $\langle \text{occurrence-modified-attr-term-qt} \rangle ::= \langle \text{occurrence-modifier} \rangle \text{'('} (\langle \text{modified-attr-term-qt} \rangle \text{' ,? '* '})$
 $\langle \text{occurrence-modifier} \rangle ::= \langle \text{optional-modifier} \rangle | \text{'without'}$
 $\langle \text{location-modified-qt} \rangle ::= \langle \text{location-modifier} \rangle \text{'('} \langle \text{term-variable-qt} \rangle \text{' ,? '* '})$
 $\langle \text{location-modifier} \rangle ::= \langle \text{descendant-modifier} \rangle | \langle \text{position-modifier} \rangle$
 $\langle \text{descendant-modifier} \rangle ::= \text{'descendant'} | \text{'desc'}$
 $\langle \text{position-modifier} \rangle ::= (\text{'position'} | \text{'pos'}) \langle \text{number-qt} \rangle$
 $\langle \text{number-qt} \rangle ::= \langle \text{Int} \rangle | \langle \text{literal-variable-ct} \rangle$
 $\langle \text{top-level-query-term} \rangle ::= \langle \text{top-term-level-declare-qt} \rangle$
 $\quad | \langle \text{optional-top-level-qt} \rangle$
 $\quad | \langle \text{term-formula-qt} \rangle$
 $\quad | \langle \text{document-specification-qt} \rangle$
 $\langle \text{top-term-level-declare-qt} \rangle ::= \text{'declare'} \langle \text{ns-decl-qt} \rangle \langle \text{variable-decl-qt} \rangle$
 $\quad \text{'('} \langle \text{top-level-construct-term} \rangle \text{')'}$
 $\langle \text{optional-top-level-qt} \rangle ::= \langle \text{optional-modifier} \rangle \text{'('} \langle \text{descendant-top-level-qt} \rangle \text{')'}$
 $\quad | \langle \text{descendant-top-level-qt} \rangle$
 $\langle \text{descendant-top-level-qt} \rangle ::= \langle \text{descendant-modifier} \rangle \text{'('} \langle \text{var-restriction-top-level-qt} \rangle \text{')'}$
 $\quad | \langle \text{var-restriction-top-level-qt} \rangle$
 $\langle \text{var-restriction-top-level-qt} \rangle ::= \langle \text{term-variable} \rangle \text{'->'} \langle \text{structured-term-qt} \rangle$
 $\quad | \langle \text{structured-term-qt} \rangle$
 $\langle \text{term-formula-qt} \rangle ::= \text{'not'} \text{'('} \langle \text{top-level-query-term} \rangle \text{')' } \langle \text{condition-clause-qt} \rangle?$
 $\quad | \text{'and'} | \text{'or'} \text{'('} \langle \text{top-level-query-term} \rangle \text{' ,? ' } (\langle \text{top-level-query-term} \rangle \text{' ,? '})^+$
 $\quad \text{')' } \langle \text{condition-clause-qt} \rangle?$
 $\langle \text{document-specification-qt} \rangle ::= \text{'in'} \langle \text{xml-document-specification-qt} \rangle$
 $\langle \text{xml-document-specification-qt} \rangle ::= \text{'xml-document'} \text{'('} (\langle \text{location-qt} \rangle$
 $\quad \langle \text{xml-decl-qt} \rangle? \langle \text{doctype-decl-qt} \rangle? \text{')' } \langle \text{xml-document-children-qt} \rangle$

$\langle location-qt \rangle ::= 'location' '=' (\langle IRI \rangle | \langle literal-variable-qt \rangle)$
 $\langle xml-decl-qt \rangle ::= ('standalone' '=' ('true' | 'false' | \langle literal-variable-qt \rangle))? ('xml-version' '=' ('1.0' | '1.1' | \langle literal-variable-qt \rangle))?$
 $\langle doctype-decl-qt \rangle ::= ('system-id' '=' \langle identifier-qt \rangle)? ('public-id' '=' \langle identifier-qt \rangle)? ('root-name' '=' \langle identifier-qt \rangle)?$
 $\langle xml-document-children-qt \rangle ::= '[[\langle xml-document-content-qt \rangle]]'$
 $\quad | '\{\{ \langle xml-document-content-qt \rangle \}\}'$
 $\quad | '\{ \langle xml-document-content-qt \rangle \}'$
 $\quad | '[' \langle xml-document-content-qt \rangle ']'$
 $\langle xml-document-content-qt \rangle ::= (\langle annotation-content-qt \rangle ',')^* \langle top-level-query-term \rangle$
 $\quad (\langle annotation-content-qt \rangle ',')^*$
 $\langle annotation-content-qt \rangle ::= \langle comment-qt \rangle | \langle processing-instruction-qt \rangle$

A.5 Programs

$\langle program \rangle ::= 'PROGRAM' '(' \langle goal-block \rangle ')' 'END'$
 $\langle goal-block \rangle ::= \langle rule-level-declare-block \rangle^* \langle goal \rangle \langle rule-level-declare-block \rangle^*$
 $\quad | \langle rule-level-declare \rangle '(' \langle goal-block \rangle ')' 'END'$
 $\langle rule-level-declare-block \rangle ::= \langle goal \rangle | \langle construct-query-rule \rangle | \langle data \rangle$
 $\quad | \langle rule-level-declare \rangle '(' \langle rule-level-declare-block \rangle^* ')' 'END'$
 $\langle goal \rangle ::= 'GOAL' '(' \langle out-resource \rangle ')' 'FROM' '(' \langle query-term \rangle ')' 'END'$
 $\langle rule \rangle ::= 'CONSTRUCT' '(' \langle construct-term \rangle ')' 'FROM' '(' \langle query-term \rangle ')' 'END'$
 $\langle out-resource \rangle ::= \text{construct-term}$
 $\quad | 'out' (\langle iri \rangle | \langle literal-var \rangle) '(' \text{construct-term} ')'$
 $\langle data \rangle ::= 'DATA' '(' \langle data-term \rangle ')' 'END'$
 $\langle rule-level-declare \rangle ::= 'DECLARE' '(' \langle var-decl-qt \rangle^* \langle ns-decl-qt \rangle^* ')' 'END'$

Appendix B

Grammar for XML-style Term Syntax

B.1 Literal Structures

The literal structures for the XML-style term syntax are identical to the literal structures for non-XML term syntax, i.e., as given in Section A.1.

B.2 Data Terms

$\langle data-term \rangle ::= \langle term-level-declare-dt \rangle \mid \langle reference-dt \rangle \mid \langle structured-dt \rangle \mid \langle content-dt \rangle$

$\langle reference-dt \rangle ::= \text{'^'} \langle identifier-dt \rangle$

$\langle term-level-declare-dt \rangle ::= \text{'declare'} \langle ns-declaration-dt \rangle \text{'('} (\langle data-term \rangle \text{' , '?' })^* \text{')'}$

$\langle identifier-dt \rangle ::= \langle NCName \rangle \mid \langle IRI \rangle \mid \langle String \rangle$

$\langle ns-declaration-dt \rangle ::= (\langle ns-prefix-declaration-dt \rangle \text{' , '?' })^* (\langle ns-default-declaration-dt \rangle \text{' , '?' })? (\langle ns-prefix-declaration-dt \rangle \text{' , '?' })^*$

$\langle ns-prefix-declaration-dt \rangle ::= \text{'ns-prefix'} \langle identifier-dt \rangle \text{'='} \langle IRI \rangle$

$\langle ns-default-declaration-dt \rangle ::= \text{'ns-default'} \langle IRI \rangle$

$\langle content-dt \rangle ::= \langle literal-content-dt \rangle \mid \langle comment-dt \rangle \mid \langle processing-instruction-dt \rangle$

$\langle literal-content-dt \rangle ::= \langle String \rangle$

$\langle comment-dt \rangle ::= \text{'<!-' } \langle literal-content-dt \rangle \text{'->'}$

$\langle processing-instruction-dt \rangle ::= \text{'<?' } \langle identifier-dt \rangle \langle literal-content-dt \rangle \text{'?>'}$

$\langle structured-dt \rangle$::= '<' $\langle local-spec-dt \rangle$ $\langle properties-dt \rangle$ ('>' $\langle children-list-dt \rangle$ ('</>' '<' $\langle ns-label-dt \rangle$ '>') '/>')
$\langle properties-dt \rangle$::= ('{' 'ordered' '}')? ('{' 'unordered' '}')
$\langle children-list-dt \rangle$::= $\langle data-term \rangle^*$
$\langle local-spec-dt \rangle$::= $\langle term-identifier-dt \rangle?$ $\langle ns-label-dt \rangle$ $\langle attr-term-list-dt \rangle$
$\langle term-identifier-dt \rangle$::= $\langle identifier-dt \rangle$ '@'
$\langle ns-label-dt \rangle$::= (($\langle identifier-dt \rangle$ ':')? $\langle identifier-dt \rangle$)
$\langle attr-term-list-dt \rangle$::= $\langle attr-term-dt \rangle^*$
$\langle attr-term-dt \rangle$::= $\langle base-attr-term-dt \rangle$
$\langle base-attr-term-dt \rangle$::= $\langle ns-label-dt \rangle$ '=' $\langle literal-content-dt \rangle$
$\langle top-level-data-term \rangle$::= $\langle top-term-level-declare-dt \rangle$ $\langle structured-dt \rangle$
$\langle top-term-level-declare-dt \rangle$::= 'declare' $\langle ns-declaration-dt \rangle$ '(' $\langle top-level-data-term \rangle$ ')'

- There is an additional restriction on the production for $\langle structured-dt \rangle$: the (namespace, local name) pair used as label in the end element tag and the (namespace, local name) pair used in the start element tag (i.e., produced as part of $\langle local-spec-dt \rangle$) must be (modulo whitespace) component wise equivalent character sequences.

B.3 Construct Terms

$\langle construct-term \rangle$::= $\langle term-level-declare-ct \rangle$ $\langle reference-ct \rangle$ $\langle structured-ct \rangle$ $\langle content-ct \rangle$ $\langle term-variable-ct \rangle$ $\langle modified-ct \rangle$
$\langle reference-ct \rangle$::= '^' $\langle identifier-ct \rangle$
$\langle term-level-declare-ct \rangle$::= 'declare' $\langle ns-declaration-ct \rangle$ '(' ($\langle construct-term \rangle$ ',')* ')'
$\langle identifier-ct \rangle$::= $\langle NCName \rangle$ $\langle IRI \rangle$ $\langle String \rangle$ $\langle literal-variable-ct \rangle$
$\langle ns-declaration-ct \rangle$::= (($\langle ns-prefix-declaration-ct \rangle$ ',')* ($\langle ns-default-declaration-ct \rangle$ ',')? ($\langle ns-prefix-declaration-ct \rangle$ ',')*)
$\langle ns-prefix-declaration-ct \rangle$::= 'ns-prefix' $\langle identifier-ct \rangle$ '=' (($\langle IRI \rangle$ $\langle literal-variable-ct \rangle$)
$\langle ns-default-declaration-ct \rangle$::= 'ns-default' (($\langle IRI \rangle$ $\langle literal-variable-ct \rangle$)
$\langle content-ct \rangle$::= $\langle literal-content-ct \rangle$ $\langle comment-ct \rangle$ $\langle processing-instruction-ct \rangle$

<literal-content-ct> ::= <String> | <literal-variable-ct>
 <comment-dt> ::= '<!-' <literal-content-ct> '->'

<processing-instruction-dt> ::= '<?' <identifier-ct> <literal-content-ct> '?>'

<structured-ct> ::= '<' <local-spec-ct> <properties-ct> (>' <children-list-ct> (</>' | '<' <ns-label-ct> '>') | '/>')

<properties-ct> ::= ('{' 'ordered' '}')? | ('{' 'unordered' '}')

<children-list-ct> ::= <construct-term>*

<local-spec-ct> ::= <term-identifier-ct>? <ns-label-ct> <attr-term-list-ct>

<term-identifier-ct> ::= <identifier-ct> '@'

<ns-label-ct> ::= (<identifier-ct> ':')? <identifier-ct>

<attr-term-list-ct> ::= <attr-term-ct>*

<attr-term-ct> ::= <base-attr-term-ct>
 | <term-variable-ct>
 | <modified-attr-term-ct>

<base-attr-term-ct> ::= <ns-label-ct> '=' <literal-content-ct>

<top-level-construct-term> ::= <top-term-level-declare-ct> | <structured-ct>

<top-term-level-declare-ct> ::= 'declare' <ns-declaration-ct> '(' <top-level-construct-term> ')

<variable-ct> ::= <term-variable-ct> | <literal-variable-ct>

<term-variable-ct> ::= <var-specification-ct>

<literal-variable-ct> ::= <var-specification-ct>

<var-specification-ct> ::= ('variable' | 'var')? <NCName>

<modified-ct> ::= <grouping-ct> | <optional-ct>

<modified-attr-term-ct> ::= <grouping-attr-term-ct> | <optional-attr-term-ct>

<grouping-ct> ::= <grouping-modifier> '(' ((<construct-term> ',')* '>')? <groupby>? <orderby>?

<grouping-attr-term-ct> ::= <grouping-modifier> '(' <attr-term-ct>? '>' <groupby>? <orderby>?

<grouping-modifier> ::= 'all'
 | 'some' <number-ct>
 | 'first' <interval-ct>

<i><orderby></i>	::= 'order-by' '(' (((<i><optional-variable></i>) <i><variable-ct></i>) ',')* ')' <i><order-relation></i> ?
<i><order-relation></i>	::= 'ascending' 'descending' <i><NCName></i>
<i><groupby></i>	::= 'group-by' '(' (((<i><optional-variable></i>) <i><variable-ct></i>) ',')* ')' <i><equivalence-relation></i> ?
<i><equivalence-relation></i>	::= <i><NCName></i>
<i><optional-variable></i>	::= <i><optional-modifier></i> <i><variable-ct></i>
<i><interval-ct></i>	::= <i><number-ct></i> '-' <i><number-ct></i> <i><number-ct></i> '-' '+'
<i><number-ct></i>	::= <i><Int></i> <i><literal-variable-ct></i>
<i><optional-ct></i>	::= <i><optional-modifier></i> '(' ((<i><construct-subterm></i>) ',')* ')' ('with-default' '(' ((<i><construct-subterm></i>) ',')* ')')?
<i><optional-attr-term-ct></i>	::= <i><optional-modifier></i> '(' ((<i><attr-term-ct></i>) ',')* ')' ('with-default' '(' ((<i><attr-term-ct></i>) ',')* ')')?
<i><optional-modifier></i>	::= 'optional' 'opt'

B.4 Query Terms

<i><query-term></i>	::= <i><modified-qt></i> <i><term-level-declare-qt></i>
<i><term-level-declare-qt></i>	::= 'declare' <i><ns-decl-qt></i> <i><variable-decl-qt></i> '('(<i><query-term></i>) ',')* ')'
<i><modified-qt></i>	::= <i><variable-term-qt></i> <i><location-modified-qt></i> <i><occurrence-modified-qt></i> <i><selection-modified-qt></i>
<i><base-term-qt></i>	::= <i><reference-qt></i> <i><content-qt></i> <i><structured-qt></i>
<i><reference-qt></i>	::= '^' <i><identifier-qt></i>
<i><identifier-qt></i>	::= <i><NCName></i> <i><IRI></i> <i><String></i> <i><literal-variable-qt></i> <i><Regexp></i>
<i><ns-decl-qt></i>	::= ((<i><ns-prefix-decl-qt></i>) ',')* ((<i><ns-default-decl-qt></i>) ',')? (<i><ns-prefix-decl-qt></i>) ',')*
<i><ns-prefix-decl-qt></i>	::= 'ns-prefix' <i><identifier-qt></i> '=' ((<i><IRI></i>) <i><literal-variable-qt></i>)

<code><ns-default-decl-dt></code>	::= 'ns-default' (<IRI> <literal-variable-qt>)
<code><variable-decl-qt></code>	::= (('variable' 'var') <NCName> ',')*
<code><content-qt></code>	::= <literal-content-qt> <comment-qt> <processing-instruction-qt>
<code><literal-content-qt></code>	::= <String> <literal-variable-qt> <Regex>
<code><comment-qt></code>	::= '<!--' <literal-content-qt> '->'
<code><processing-instruction-qt></code>	::= '<?' <identifier-qt> <literal-content-qt> '?>'
<code><structured-qt></code>	::= '<' <local-spec-qt> <properties-qt> ('>' <children-list-qt> ('</>' '<' <ns-label-qt> '>') '/>') <condition-clause-qt>?
<code><properties-qt></code>	::= ('{' 'ordered' '}')? ('{' 'unordered' '}') (('{' 'total' '}')? {' 'partial' '}') (('{' 'total attributes' '}')? {' 'partial attributes' '})
<code><children-list-qt></code>	::= <query-term>*
<code><condition-clause-qt></code>	::= 'where' '(' <condition-qt> ')'
<code><condition-qt></code>	::= <c-parameter> <comparison-op> <c-parameter> <comparison-op> '(' <c-parameter> <c-parameter> ')' 'and' '(' <condition-qt> <condition-qt>+ ')' 'or' '(' <condition-qt> <condition-qt>+ ')' 'not' '(' <condition-qt> ')' <c-parameter>
<code><condition-op></code>	::= '==' '!=' '<' '>' '<=' '>='
<code><arithmetic-op></code>	::= '+' '-' '*' '/' '^'
<code><c-parameter></code>	::= <optional-variable-qt> <variable-qt> <String> <Int> <c-parameter> <arithmetic-op> <c-parameter> <arithmetic-op> '(' <c-parameter> <c-parameter> ')'
<code><optional-variable-qt></code>	::= <optional-modifier> <variable-qt>
<code><local-spec-qt></code>	::= <term-identifier-qt>? <ns-label-qt> <attr-term-list-qt>
<code><term-identifier-qt></code>	::= <identifier-qt> '@'
<code><ns-label-qt></code>	::= (<identifier-qt> ':')? <identifier-qt>
<code><attr-term-list-qt></code>	::= <attr-term-qt>*
<code><attr-term-qt></code>	::= <modified-attr-term-qt>

$\langle \text{modified-attr-term-qt} \rangle ::= \langle \text{base-attr-term-qt} \rangle$
 $\quad | \langle \text{variable-attr-term-qt} \rangle$
 $\quad | \langle \text{occurrence-modified-attr-term-qt} \rangle$
 $\quad | \langle \text{selection-modified-attr-term-qt} \rangle$

$\langle \text{base-attr-term-qt} \rangle ::= \langle \text{ns-label-qt} \rangle '=' \langle \text{literal-content-qt} \rangle$

$\langle \text{variable-term-qt} \rangle ::= \langle \text{base-term-qt} \rangle$
 $\quad | \langle \text{term-variable-qt} \rangle ('->' \langle \text{base-term-qt} \rangle)?$

$\langle \text{variable-attr-term-qt} \rangle ::= \langle \text{term-variable} \rangle ('->' \langle \text{base-attr-term-qt} \rangle)?$

$\langle \text{variable-qt} \rangle ::= \langle \text{term-variable-qt} \rangle | \langle \text{literal-variable-qt} \rangle$

$\langle \text{term-variable-qt} \rangle ::= \langle \text{var-specification-qt} \rangle$

$\langle \text{literal-variable-qt} \rangle ::= \langle \text{var-specification-qt} \rangle$

$\langle \text{var-specification-qt} \rangle ::= ('variable' | 'var')? \langle \text{NCName} \rangle$
 $\quad | \langle \text{anonymous-variable} \rangle$

$\langle \text{anonymous-variable} \rangle ::= '_'$

$\langle \text{selection-modified-qt} \rangle ::= \langle \text{selection-modifier} \rangle '(' (\langle \text{modified-qt} \rangle ',')^* ')'$

$\langle \text{selection-modified-attr-term-qt} \rangle ::= \langle \text{selection-modifier} \rangle '(' (\langle \text{modified-attr-term-qt} \rangle ',')^* ')'$

$\langle \text{selection-modifier} \rangle ::= \text{'except'}$

$\langle \text{occurrence-modified-qt} \rangle ::= \langle \text{occurrence-modifier} \rangle '(' (\langle \text{modified-qt} \rangle ',')^* ')'$

$\langle \text{occurrence-modified-attr-term-qt} \rangle ::= \langle \text{occurrence-modifier} \rangle '(' (\langle \text{modified-attr-term-qt} \rangle ',')^* ')'$

$\langle \text{occurrence-modifier} \rangle ::= \langle \text{optional-modifier} \rangle | \text{'without'}$

$\langle \text{location-modified-qt} \rangle ::= \langle \text{location-modifier} \rangle '(' \langle \text{term-variable-qt} \rangle ',')^* ')'$

$\langle \text{location-modifier} \rangle ::= \langle \text{descendant-modifier} \rangle | \langle \text{position-modifier} \rangle$

$\langle \text{descendant-modifier} \rangle ::= \text{'descendant'} | \text{'desc'}$

$\langle \text{position-modifier} \rangle ::= (\text{'position'} | \text{'pos'}) \langle \text{number-qt} \rangle$

$\langle \text{number-qt} \rangle ::= \langle \text{Int} \rangle | \langle \text{literal-variable-ct} \rangle$

$\langle \text{top-level-query-term} \rangle ::= \langle \text{top-term-level-declare-qt} \rangle$
 $\quad | \langle \text{optional-top-level-qt} \rangle$
 $\quad | \langle \text{term-formula-qt} \rangle$
 $\quad | \langle \text{document-specification-qt} \rangle$

$\langle \text{top-term-level-declare-qt} \rangle ::= \text{'declare' } \langle \text{ns-decl-qt} \rangle \langle \text{variable-decl-qt} \rangle$
 $\quad \text{'(} \langle \text{top-level-construct-term} \rangle \text{'}$

$\langle \text{optional-top-level-qt} \rangle ::= \langle \text{optional-modifier} \rangle \text{'(} \langle \text{descendant-top-level-qt} \rangle \text{'}$
 $\quad | \langle \text{descendant-top-level-qt} \rangle$

$\langle \text{descendant-top-level-qt} \rangle ::= \langle \text{descendant-modifier} \rangle \text{'(} \langle \text{var-restriction-top-level-qt} \rangle \text{'}$
 $\quad | \langle \text{var-restriction-top-level-qt} \rangle$

$\langle \text{var-restriction-top-level-qt} \rangle ::= \langle \text{term-variable} \rangle \text{'->' } \langle \text{structured-term-qt} \rangle$
 $\quad | \langle \text{structured-term-qt} \rangle$

$\langle \text{term-formula-qt} \rangle ::= \text{'not' } \text{'(} \langle \text{top-level-query-term} \rangle \text{' } \langle \text{condition-clause-qt} \rangle ?$
 $\quad | \text{'and' } | \text{'or' } \text{'(} \langle \text{top-level-query-term} \rangle \text{' , } ? \langle \text{top-level-query-term} \rangle \text{' , } ? \rangle +$
 $\quad \text{')' } \langle \text{condition-clause-qt} \rangle ?$

$\langle \text{document-specification-qt} \rangle ::= \text{'in' } \langle \text{xml-document-specification-qt} \rangle$

$\langle \text{xml-document-specification-qt} \rangle ::= \text{'xml-document' } \text{'(} \langle \text{location-qt} \rangle$
 $\quad \langle \text{xml-decl-qt} \rangle ? \langle \text{doctype-decl-qt} \rangle ? \text{')' } \langle \text{xml-document-children-qt} \rangle$

$\langle \text{location-qt} \rangle ::= \text{'location' } \text{'=' } \langle \text{IRI} \rangle | \langle \text{literal-variable-qt} \rangle$

$\langle \text{xml-decl-qt} \rangle ::= \text{'standalone' } \text{'=' } \langle \text{true} \rangle | \langle \text{false} \rangle | \langle \text{literal-variable-qt} \rangle \rangle ? \text{'xml-version'}$
 $\quad \text{'=' } \langle \text{1.0} \rangle | \langle \text{1.1} \rangle | \langle \text{literal-variable-qt} \rangle \rangle ?$

$\langle \text{doctype-decl-qt} \rangle ::= \text{'system-id' } \text{'=' } \langle \text{String} \rangle | \langle \text{literal-variable-qt} \rangle \rangle ? \text{'public-id' } \text{'=' } \langle \text{String} \rangle$
 $\quad | \langle \text{literal-variable-qt} \rangle \rangle ? \text{'root-name' } \text{'=' } \langle \text{String} \rangle | \langle \text{literal-variable-qt} \rangle \rangle ?$

$\langle \text{xml-document-children-qt} \rangle ::= \text{'[} \langle \text{xml-document-content-qt} \rangle \text{']'}$
 $\quad | \text{'\{ \{ } \langle \text{xml-document-content-qt} \rangle \text{' \} \}'}$
 $\quad | \text{'\{ } \langle \text{xml-document-content-qt} \rangle \text{' \}'}$
 $\quad | \text{'[} \langle \text{xml-document-content-qt} \rangle \text{']'}$

$\langle \text{xml-document-content-qt} \rangle ::= \langle \text{annotation-content-qt} \rangle \text{' , ' } * \langle \text{top-level-query-term} \rangle$
 $\quad \langle \text{annotation-content-qt} \rangle \text{' , ' } *$

$\langle \text{annotation-content-qt} \rangle ::= \langle \text{comment-qt} \rangle | \langle \text{processing-instruction-qt} \rangle$

B.5 Programs

$\langle \text{program} \rangle ::= \text{'PROGRAM' } \text{'(} \langle \text{goal-block} \rangle \text{')' } \text{'END'}$

$\langle \text{goal-block} \rangle ::= \langle \text{rule-level-declare-block} \rangle * \langle \text{goal} \rangle \langle \text{rule-level-declare-block} \rangle *$
 $\quad | \langle \text{rule-level-declare} \rangle \text{'(} \langle \text{goal-block} \rangle \text{')' } \text{'END'}$

$\langle \text{rule-level-declare-block} \rangle ::= \langle \text{goal} \rangle | \langle \text{construct-query-rule} \rangle | \langle \text{data} \rangle$
 $\quad | \langle \text{rule-level-declare} \rangle \text{'(} \langle \text{rule-level-declare-block} \rangle * \text{')' } \text{'END'}$

$\langle \textit{goal} \rangle$::= 'GOAL' '(' $\langle \textit{out-resource} \rangle$ ')' 'FROM' '(' $\langle \textit{query-term} \rangle$ ')' 'END'
 $\langle \textit{rule} \rangle$::= 'CONSTRUCT' '(' $\langle \textit{construct-term} \rangle$ ')' 'FROM' '(' $\langle \textit{query-term} \rangle$ ')' 'END'
 $\langle \textit{out-resource} \rangle$::= $\textit{construct-term}$
| 'out' ((\textit{iri}) | $\langle \textit{literal-var} \rangle$) '(' $\textit{construct-term}$ ')'
 $\langle \textit{data} \rangle$::= 'DATA' '(' $\langle \textit{data-term} \rangle$ ')' 'END'
 $\langle \textit{rule-level-declare} \rangle$::= 'DECLARE' '(' $\langle \textit{var-decl-qt} \rangle^* \langle \textit{ns-decl-qt} \rangle^*$ ')'

Appendix C

Relax NG Schema for XML Syntax

C.1 Parameterized Grammars: Terms, Declarations, Modifiers, etc.

C.1.1 Declarations

```
default namespace = "http://xcerpt.org/ns/core/1.0"
2 namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"

4 start = declare-block

6 ## A declare block with an empty content and both namespace and variable declarations.
declare-block =
8 element declare { (ns-declaration | var-declaration)*, content }
ns-declaration =
10 ns-prefix-declaration*,
(element ns-default {
12 element iri { iri.class }
})
| ns-prefix-declaration),
ns-prefix-declaration*
16 ns-prefix-declaration =
element ns-prefix {
18 element name { ncname.class },
element iri { iri.class }
20 }
var-declaration =
22 element variable {
attribute name { xsd:NCName }
24 }
content = empty
26 iri.class |= text
ncname.class |= xsd:NCName
```

C.1.2 Conditions

```
1 namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"
3 start = condition
5 ## A condition is an opaque expression involving simple arithmetic and comparisons. This is preliminary
   syntax.
   condition = element condition { expression }
7 expression =
   element arithmetic-expression {
9     attribute operator { "+" | "-" | "*" | "/" | "^" },
       expression,
11    expression
   }
13 | element comparison-expression {
       attribute operator { "eq" | "neq" | "lt" | "gt" | "leq" | "geq" },
15    expression,
       expression
17    }
   | grammar {
19     include "formula.rnc" {
       content = parent expression
21    }
   }
23 | content
   content = empty
```

C.1.3 Formulas

```
default namespace = "http://xcerpt.org/ns/core/1.0"
2 namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"
4 start = formula
   formula =
6     element and { formula, formula+, condition? }
       | element or { formula, formula+, condition? }
8     | element not { formula }
       | content
10    condition = empty
       content = empty
```

C.1.4 Modifiers

```
1 default namespace = "http://xcerpt.org/ns/core/1.0"
   namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"
3
   start = grouping
5    content = empty
       grouping =
7     element all { content, order-by?, group-by? }
```

```

    | element some { number, content, order-by?, group-by? }
9 | element first { interval, content, order-by?, group-by? }
  order-by =
11 element order-by {
    attribute order-relation { text }?,
13   optional-variable+
    }
15 group-by =
    element group-by {
17   attribute equivalence-relation { text }?,
    optional-variable+
19   }
  optional-variable =
21 element optional { variable }
    | variable
23 variable = empty
  interval =
25 element interval {
    element min { number-literal.class },
27   element max { number-literal.class }
    }
29 number = element number { number-literal.class }
  number-literal.class = xsd:int | variable
31 optional =
    element optional {
33   content,
    element with-default { content }?
35   }

```

C.1.5 Term

```

1 default namespace = "http://xcerpt.org/ns/core/1.0"
  namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"
3
  start = top-level-term.class
5
  ## A term that may occur at top-level. Slightly more
7 ## restricted than a basic term.
  top-level-term.class =
9   structured-term
    | grammar {
11     include "declare-block.rnc" {
        content = parent top-level-term.class*
13       var-declaration = empty
    }
15   }

17 ## A generic Xcerpt term. Variants are data, construct, and query terms.
  term.class |=
19   reference | content-term | structured-term | term-level-declare

```

```

21 ## A declaration block on term level allows possibly (in data and construct terms) only namespace
    declarations.
    term-level-declare =
23   grammar {
        include "declare-block.rnc" {
25         content = parent term.class*
            var-declaration = empty
27     }
    }
29
    ## A structured term is a term that may have children and
31 ## attributes. It contrasts with literal content.
    structured-term =
33   element element { term-local-spec, term-children, term-condition? }

35 ## Some terms may have additional constraints attached to them.
    term-condition = empty
37
    ## The specification of the 'local' properties of a term: identifier, label, namespace, and attributes.
39 term-local-spec = term-identifier?, ns-label, attr-term-list

41 ## The defining occurrence of a reference, i.e. "id @" in term syntax.
    term-identifier = element identifier { identifier.class }
43
    ## Label and namespace of an Xcerpt term or attribute.
45 ns-label =
        element label {
47     element ns { identifier.class }?,
        identifier.class
49 }

51 ## A term specifying the attributes of an element.
    attr-term-list =
53   element attributes {
        attribute total { total.class },
55     attribute-term.class*
    }
57
    ## Class of values for attributes specifying totality or
59 ## partiality of a term's children or attribute list.
    total.class |= "true"
61
    ## A attribute term is an attribute possibly modified with respect to location, modality, and selection.
63 attribute-term.class |= base-attribute

65 ## An attribute consists of a label and an attribute content.
    base-attribute =
67   element attribute {
        ns-label,
69     element value { literal-content.class }
    }

```

```

71  ## An identifier such as a namespace or label.
73  identifier.class |= text
    content-term = literal-content.class | annotation-content
75  ## Content kinds that can be used to annotate elements.
77  annotation-content =
    element comment { literal-content.class }
79  | element processing-instruction {
    attribute target { identifier.class },
81  literal-content.class
    }
83  ## Character data or other atomic content.
85  literal-content.class |= text

87  ## The children of a term can be ordered or unordered, total or partial.
    term-children =
89  element children {
    attribute ordered { "true" | "false" },
91  attribute total { total.class },
    term.class*
93  }

95  ## The using occurrence of a reference, i.e. "^ id" in term syntax.
    reference = element reference { identifier.class }

```

C.2 Grammar for Xcerpt Programs

```

default namespace = "http://xcerpt.org/ns/core/1.0"
2  namespace a = "http://relaxng.org/ns/compatibility/annotations/1.0"

4  start = program
    # -----
    # -----
6  # Programs: Rules, Data, and Goals
8  # -----
    # -----
10  ## An Xcerpt program is a set of (one or more) goals, as well as (any number of) rules and inline data
    terms (like facts in Prolog). Rules and data terms may be surrounded by declaration blocks.
12  program = element program { goal-block }
    goal-block =
14  rule-level-block*
    | goal
16  | rule-level-block*
    | grammar {
18  include "declare-block.rnc" {
    content = parent goal-block

```

```

20     }
    }
22
    ## Rule-level blocks form the basic block structure of an Xcerpt programs: goals, rules, and inline data
    terms form the basic block structures. They can be included into declaration blocks that define the
    scope of variable and namespace declarations.
24 rule-level-block =
    goal
26 | rule
    | data
28 |
    ## A declaration block on rule level allows both variable and namespace declarations.
30 grammar {
    include "declare-block.rnc" {
32     content = parent rule-level-block*
    }
34 }

36 ## A rule specifies how from data matched by the query term new data is constructed according to a
    construct term.
    rule =
38 element rule {
    element construct { construct-term },
40 element from { query-term }
    }
42
    ## A goal is a rule, where the resulting data is written to a specified resource. Hence, goals are not
    chained.
44 goal =
    element goal {
46 element out {
    (variable-ct
48 | attribute value {
    text
50 >> a:documentation [
    "This should in-fact be a IRI as by RFC 3987. Since XML Schema datatypes only provides the
    anyURI datatype for URIs conforming the older RFC 2396, arbitrary text is allowed."
52 ]
    }),
54 element construct { construct-term }
    },
56 element from { query-term }
    }
58
    ## An inline data term allows the specification of data terms inside Xcerpt programs similar to facts in
    Prolog.
60 data = element data { data-term }
    # -----
    -----
62
    # Data Terms

```

```

64 # -----
66 data-term =
    grammar {
68     include "term.rnc"
    }
70 # -----

72 # Construct Terms

74 # -----

76 ## Variables for construct terms.
variable-ct =
78     element variable {
        attribute name { xsd:NCName }
80     }

82 ## A construct term differs from a data term in the addition of variables. As a corollary a few so-called
    modifiers are needed to indicate, e.g., how to group the variables or whether a variable may have no
    bindings.
construct-term =
84     grammar {
        variable-ct = parent variable-ct
86     # Add grouping and optional for attributes
        modified-attribute =
88         grammar {
            include "modifiers.rnc" {
90             start = grouping
            content = parent attribute-term.class
92             variable = parent variable-ct
            }
94         }
        | grammar {
96             include "modifiers.rnc" {
                start = optional
98             content = parent attribute-term.class*
            variable = parent variable-ct
100             }
            }
102     # Add grouping and optional for elements
    modified-term =
104     grammar {
        include "modifiers.rnc" {
106             start = grouping
            content = parent term.class*
108             variable = parent variable-ct
        }
    }

```

```

110     }
111     | grammar {
112         include "modifiers.rnc" {
113             start = optional
114             content = parent term.class*
115             variable = parent variable-ct
116         }
117     }
118
119     ## Construct terms may also be variables or modified by
120     ## grouping and optional modifiers.
121     term.class |= variable-ct | modified-term
122
123     ## Construct attribute terms may also be variables or modified by
124     ## grouping and optional modifiers.
125     attribute-term.class |= variable-ct | modified-attribute
126     # Add variables to identifiers and literal content
127     identifier.class |= variable-ct
128     literal-content.class |= variable-ct
129     include "term.rnc"
130 }
131 # -----
132
133 # Query Terms
134 # -----
135
136 ## A POSIX.1 regular expression annotated with variables may occur in query terms at the position of
137     identifiers or literal content.
138 regular-expression =
139     element regexp {
140         attribute value { text }
141     }
142
143     ## Query terms can evidently become the most complex of the three term kinds in Xcerpt. As construct
144     terms they add variables to data terms. But they also provide means for expressing incompleteness:
145     partial terms, desc and position location modifiers, etc. A construct term differs from a data term in
146     the addition of variables.
147 query-term =
148     grammar {
149
150         ## Variables for query terms.
151         variable =
152             element variable {
153                 attribute anonymous { "true" }
154                 | attribute name { xsd:NCName }
155             }
156         # #1# TOP-LEVEL QUERY TERM ##
157         optional-top-level-term =

```



```

    element optional { descendant-top-level-term }
156 | descendant-top-level-term
descendant-top-level-term =
158   element descendant { var-restriction-top-level-term }
    | var-restriction-top-level-term
160 var-restriction-top-level-term =
    element restriction { variable, structured-term }
162 | structured-term
term-formula =
164   grammar {
    include "formula.rnc" {
166     content = parent optional-top-level-term
    condition = parent condition-clause
168   }
  }
170 document-specification =
    element xml-document {
172     attribute location { text },
    element xml-declaration {
174     attribute standalone { "true" | "false" | variable }?,
    attribute xml-version { "1.0" | "1.1" | variable }?
176   }?,
    element doctype {
178     attribute system-id { identifier.class }?,
    attribute public-id { identifier.class }?,
180     attribute root-name { identifier.class }?
    }?,
182     element children {
    annotation-content*, top-level-term.class, annotation-content*
184   }
  }
186 ##2## CONDITION CLAUSES ##
condition-clause =
188   element condition {
    grammar {
190     include "formula.rnc" {
    content = parent comparison*
192   }
  }
194 }
comparison =
196   element comparison {
    attribute operator {
198     "eq" | "neq" | "lt" | "bt" | "elt" | "ebt"
    },
200     arithmetics,
    arithmetics
202   }
  | element optional { comparison }
204 | arithmetics
arithmetics =

```

```

206     element arithmetics {
      attribute operator {
208         "plus" | "minus" | "times" | "div" | "power"
      },
210     (arithmetics | arithmetic-parameter),
      (arithmetics | arithmetic-parameter)
212   }
    | element optional { arithmetics }
214   | arithmetic-parameter
arithmetic-parameter =
216   variable
    | element value { text }
218 # #3# Modified terms ##
modified-term =
220   variable-term | location-term | occurrence-term | selection-term
base-term = reference | content-term | structured-term
222 variable-term =
    base-term
224   | variable
    | element restriction { variable, base-term }
226 location-term =
    element descendant { variable-term }
228   | element position {
        element number { variable | xsd:int },
230     variable-term
    }
232 selection-term = element except { modified-term }
occurrence-term =
234   element without { modified-term }
    | element optional { modified-term }
236 # #4# Modified Attribute terms ##
modified-attr-term =
238   base-attribute,
    variable-attr-term,
240   occurrence-modified-attr-term,
    selection-modified-attr-term
242 variable-attr-term =
    variable
244   | element restriction { variable, base-attribute }
occurrence-modified-attr-term =
246   element without { modified-attr-term }
    | element optional { modified-attr-term }
248 selection-modified-attr-term = element except { modified-attr-term }
# #A1# BASICS ##
250
# Add variables and regular expressions to identifiers and literal
252 # content
identifier.class |= variable | parent regular-expression
254 literal-content.class |= variable | parent regular-expression
include "term.rnc" {
256   # Redefine the top-level term for query terms: add variables to

```

```

# declare blocks and allow optional, descendant, variable restriction.
258 # Add document specifications
# Add query term formula

260
## A term that may occur at top-level. Slightly more
262 ## restricted than a basic term.
top-level-term.class =
264   optional-top-level-term
    | term-formula
266   | document-specification
    | grammar {
268     include "declare-block.rnc" {
        content = parent top-level-term.class*
270     }
    }
272 # Redefine terms: only modified terms, which can in fact be
# unmodified :-) Term-level declare blocks may also contain variable
274 # declarations

276 ## A generic Xcerpt term. Variants are data, construct, and query terms.
term.class = modified-term | term-level-declare
278
## A declaration block on term level allows possibly (in data and construct terms) only namespace
    declarations.
280 term-level-declare =
    grammar {
282     include "declare-block.rnc" {
        content = parent term.class*
284     }
    }
286 # Redefine attributes as well, again to make modification possible

288 ## An attribute term is an attribute possibly modified with respect to location, modality, and
    selection.
attribute-term.class = modified-attr-term
290 # Allow conditions on arbitrary query terms
term-condition = condition-clause
292 }
}

```

C.3 Exemplary Data Term

```

1 <?xml version="1.0" encoding="UTF-8"?>
  <element xmlns="http://xcerpt.org/ns/core/1.0">
3   <label>bib</label>
  <attributes total="true" />
5   <children ordered="false" total="true">
    <element>
7     <identifier>journal.adm</identifier>
    <label>journal</label>
9     <attributes total="true" />
    <children ordered="false" total="true">
11    <element>
      <label>title</label>
    </element>
    </children>
  </element>
  </children>
</element>

```

```

13     <attributes total="true" />
14     <children ordered="true" total="true"
15     >Applied Data Management</children>
16 </element>
17 <element>
18   <label>editors</label>
19   <attributes total="true" />
20   <children ordered="true" total="true">
21     <element>
22       <label>editor-in-chief</label>
23       <attributes total="true" />
24       <children ordered="true" total="true"
25       >Titus Pomponius Atticus</children>
26     </element>
27     <element>
28       <label>editor</label>
29       <attributes total="true" >
30         <attribute><label>region</label><value>Africa</value>
31         </attribute>
32       </attributes>
33       <children ordered="true" total="true"
34       >Marcus Aemilius Aemilianus</children>
35     </element>
36     <element>
37       <label>editor</label>
38       <attributes total="true" >
39         <attribute><label>region</label><value>Gaul</value>
40         </attribute>
41       </attributes>
42       <children ordered="true" total="true"
43       >Aulus Hirtius<!-- -->
44         <element>
45           <label>affiliation</label>
46           <attributes total="true" />
47           <children ordered="true" total="true"
48           >Governor, Transalpine Gaul</children>
49         </element>
50       </children>
51     </element>
52     <element>
53       <label>editor</label>
54       <attributes total="true" >
55         <attribute><label>region</label><value>Cilicia</value>
56         </attribute>
57       </attributes>
58       <children ordered="true" total="true"
59       >Marcus Tullius Cicero<!-- -->
60         <element>
61           <label>affiliation</label>
62           <attributes total="true" />
63           <children ordered="true" total="true">Governor, Cilicia</children>
64         </element>
65       </children>
66     </element>
67 </children>
68 </element>
69 <element>
70   <label>publisher</label>
71   <attributes total="true" />
72   <children ordered="true" total="true"
73   >Titus Pomponius Atticus</children>
74 </element>
75 <element>
76   <label>volumes</label>
77   <attributes total="true" />
78   <children ordered="true" total="true">
79     <element>
80       <identifier>journal.adm.v10</identifier>
81       <label>volume</label>
82       <attributes total="true" />

```

```

83     <children ordered="true" total="true">
84         <element>
85             <identifier>journal.adm.v10.n1</identifier>
86             <label>number</label>
87             <attributes total="true">
88                 <attribute><label>type</label><value>special-issue</value>
89             </attribute>
90         </attributes>
91     <children ordered="false" total="true">
92         <element>
93             <label>title</label>
94             <attributes total="true" />
95             <children ordered="true" total="true">
96                 >Data Processing Challenges in the Age of Wax Tablets</children>
97         </element>
98     </element>
99     <label>editorial</label>
100    <attributes total="true" />
101    <children ordered="true" total="true">
102        ><reference>articles.66.cicero.wax</reference></children>
103    </element>
104 </element>
105 <label>year</label>
106 <attributes total="true" />
107 <children ordered="true" total="true">
108     >60</children>
109 </element>
110 </element>
111 <label>month</label>
112 <attributes total="true" />
113 <children ordered="true" total="true">
114     >july</children>
115 </element>
116 </children>
117 </element>
118 </element>
119 <identifier>journal.adm.v10.n2</identifier>
120 <label>number</label>
121 <attributes total="true" />
122 <children ordered="false" total="true">
123     <element>
124         <label>year</label>
125         <attributes total="true" />
126         <children ordered="true" total="true">
127             >60</children>
128         </element>
129     </element>
130     <label>month</label>
131     <attributes total="true" />
132     <children ordered="true" total="true">
133         >november</children>
134     </element>
135 </children>
136 </element> <!-- number -->
137 </children>
138 </element> <!-- volume -->
139 </children>
140 </element> <!-- volumes -->
141 </children>
142 </element> <!-- journal -->
143
144 <element>
145     <identifier>conf.dmmc</identifier>
146     <label>proceedings</label>
147     <attributes total="true" />
148     <children ordered="false" total="true">
149         <element>
150             <label>title</label>
151             <attributes total="true" />
152             <children ordered="true" total="true">

```

```

153     >Advancements in Data Management for Military and Civil Application</children>
154 </element>
155 <element>
156   <label>editors</label>
157   <attributes total="true" />
158   <children ordered="true" total="true">
159     <element>
160       <label>editor</label>
161       <attributes total="true" />
162       <children ordered="true" total="true"
163         >Marcus Aemilius Lepidus<!-- -->
164         <element>
165           <label>affiliation</label>
166           <attributes total="true" />
167           <children ordered="true" total="true"
168             >Consul, SPQR</children>
169         </element></children>
170       </element>
171     <element>
172       <label>editor</label>
173       <attributes total="true" />
174       <children ordered="true" total="true"
175         >Gaius Julius Caesar Octavianus</children>
176     </element>
177     <element>
178       <label>editor</label>
179       <attributes total="true" />
180       <children ordered="true" total="true"
181         >Marcus Antonius</children>
182     </element>
183   </children>
184 </element>
185 <element>
186   <label>publisher</label>
187   <attributes total="true" />
188   <children ordered="true" total="true"
189     >SPQR</children>
190 </element>
191 <element>
192   <label>abbrev</label>
193   <attributes total="true" />
194   <children ordered="true" total="true"
195     >DMMC</children>
196 </element>
197 <element>
198   <label>year</label>
199   <attributes total="true" />
200   <children ordered="true" total="true"
201     >44</children>
202 </element>
203 <element>
204   <label>month</label>
205   <attributes total="true" />
206   <children ordered="true" total="true"
207     >july</children>
208 </element>
209 <element>
210   <label>location</label>
211   <attributes total="true" />
212   <children ordered="true" total="true"
213     >Mutina</children>
214 </element>
215 <element>
216   <label>invited-papers</label>
217   <attributes total="true" />
218   <children ordered="true" total="true">
219     <reference>inproc.44.brutus</reference>
220     <reference>article.66.scaurus.qumran</reference>
221   </children>
222 </element> <!-- invited papers -->

```

```

223     </children>
224 </element> <!-- proceedings -->
225
226 <!-- //////////////////////////////////////////////////////////////////// -->
227 <element>
228   <identifier>article.66.scaurus.qumran</identifier>
229   <label>article</label>
230   <attributes total="true" />
231   <children ordered="false" total="true">
232     <element>
233       <label>title</label>
234       <attributes total="true" />
235       <children ordered="true" total="true">
236         >From Wax Tablets to Papyri: The Qumran Case Study</children>
237     </element>
238     <element>
239       <label>author</label>
240       <attributes total="true" />
241       <children ordered="true" total="true">
242         >Marcus Aemilius Scaurus<!-- -->
243       </children>
244       <element>
245         <label>affiliation</label>
246         <attributes total="true" />
247         <children ordered="true" total="true">
248           >Tribun, Gnaeus Pompeius Magnus</children>
249       </element>
250     </children>
251   </element>
252   <label>in</label>
253   <attributes total="true">
254     <attribute><label>scrolls</label><value>102-112</value>
255   </attribute>
256 </attributes>
257 <children ordered="true" total="true">
258   <reference>journal.adm.v10.n1</reference>
259 </children>
260 </element>
261 <element>
262   <label>citations</label>
263   <attributes total="true" />
264   <children ordered="true" total="true">
265     <element>
266       <label>cite</label>
267       <attributes total="true">
268         <attribute><label>ref</label><value>article.66.cicero.wax</value>
269       </attribute>
270     </attributes>
271     <children ordered="true" total="true" />
272   </element>
273   <element>
274     <label>cite</label>
275     <attributes total="true">
276       <attribute><label>type</label><value>formatted</value>
277     </attribute>
278   </attributes>
279   <children ordered="true" total="true">
280     >M. Aemilius Scaurus (104): A Case for Permanent Storage of
281     Senate Proceedings. In: M. Aemilius Scaurus, ed. (104):
282     <element>
283       <label>i</label>
284       <attributes total="true" />
285       <children ordered="true" total="true">
286         >Princeps Senatus: Honor and Responsibility</children>
287     </element>, Chapter 2, 14-88.</children>
288   </children> <!-- cite -->
289 </children>
290 </element> <!-- citations -->
291 </children>
</element> <!-- article-->

```

```

293 <element>
295 <identifier>article.66.cicero.wax</identifier>
297 <label>article</label>
297 <attributes total="true" />
297 <children ordered="false" total="true">
299 <element>
301 <label>title</label>
301 <attributes total="true" />
301 <children ordered="true" total="true">
303 >Space- and Time-Optimal Data Storage on Wax Tablets</children>
305 </element>
305 <element>
307 <label>authors</label>
307 <attributes total="true" />
307 <children ordered="true" total="true">
309 <element>
309 <label>author</label>
309 <attributes total="true" />
309 <children ordered="true" total="true">
311 >Marcus Tullius Cicero<!-- -->
313 </element>
313 <label>affiliation</label>
313 <attributes total="true" />
313 <children ordered="true" total="true">
315 >Governor, Cicilia</children>
317 </element>
319 </children>
321 </element>
321 <element>
323 <label>author</label>
323 <attributes total="true" />
323 <children ordered="true" total="true">
325 >Marcus Aemilius Lepidus<!-- -->
327 </element>
327 <label>affiliation</label>
327 <attributes total="true" />
327 <children ordered="true" total="true">
329 >Gens Aemilia</children>
331 </element>
333 </children>
335 </element>
335 <element>
337 <label>author</label>
337 <attributes total="true" />
337 <children ordered="true" total="true">
339 >Marcus Tullius Tiro<!-- -->
341 </element>
341 <label>affiliation</label>
341 <attributes total="true" />
341 <children ordered="true" total="true">
343 >Secretary, M. T. Cicero</children>
345 </element>
347 </children>
349 </element> <!-- authors -->
349 <element>
351 <label>in</label>
351 <attributes total="true">
353 <attribute><label>scrolls</label><value>1-94</value>
355 </attribute>
355 </attributes>
355 <children ordered="true" total="true">
357 <reference>journal.adm.v10.n1</reference>
359 </children>
359 </element>
361 <element>
361 <label>content</label>
361 <attributes total="true">

```



```

363     <attribute><label>type</label><value>xhtml</value>
364     </attribute>
365 </attributes>
366 <children ordered="true" total="true">
367   <declare>
368     <ns-default><iri>http://www.w3.org/1999/xhtml</iri></ns-default>
369   <element>
370     <label>body</label>
371     <attributes total="true" />
372     <children ordered="true" total="true">
373       <comment>incomplete due to melted letters on some tablets</comment>
374       <element>
375         <label>h1</label>
376         <attributes total="true">
377           <attribute><label>id</label><value>contributions</value></attribute>
378         </attributes>
379         <children ordered="true" total="true">Contributions</children>
380       </element>
381       <element>
382         <label>h1</label>
383         <attributes total="true" />
384         <children ordered="true" total="true">
385           >A History of Data Storage: From Stone to Parchment</children>
386         </element>
387       <element>
388         <label>p</label>
389         <attributes total="true" />
390         <children ordered="true" total="true">
391           >Despite recent evidence <element>
392             <label>cite</label>
393             <attributes total="true" />
394             <children ordered="true" total="true">
395               ><reference>article.66.scaurus.qumran</reference></children>
396             </element> ... </children>
397         </element> <!-- p -->
398       <element>
399         <label>ol</label>
400         <attributes total="true" />
401         <children ordered="true" total="true">
402           <element>
403             <label>li</label>
404             <attributes total="true" />
405             <children ordered="true" total="true">
406               <element>
407                 <label>em</label>
408                 <attributes total="true" />
409                 <children ordered="true" total="true">
410                   <element>
411                     <label>strong</label>
412                     <attributes total="true" />
413                     <children ordered="true" total="true">Homeric</children>
414                   </element> Age:</children>
415                 </element><!-- em -->
416               ...
417             </children>
418           </element> <!-- li -->
419         <element>
420           <label>li</label>
421           <attributes total="true" />
422           <children ordered="true" total="true">
423             <element>
424               <label>em</label>
425               <attributes total="true" />
426               <children ordered="true" total="true">
427                 >Age pf the <element>
428                   <label>strong</label>
429                   <attributes total="true" />
430                   <children ordered="true" total="true">Kings</children>
431                 </element></children>
432             </element><!-- em -->

```

```

433         ...
434         </children>
435     </element> <!-- li --->
436 </children>
437 </element> <!-- ol --->
438 <element>
439     <label>h1</label>
440     <attributes total="true">
441         <attribute><label>id</label><value>tiro</value></attribute>
442     </attributes>
443     <children ordered="true" total="true">Notae Tironianae</children>
444 </element> <!-- hi --->
445 <element>
446     <label>img</label>
447     <attributes total="true">
448         <attribute><label>title</label><value>Tironian et</value></attribute>
449         <attribute><label>src</label><value>...</value></attribute>
450     </attributes>
451     <children ordered="true" total="true" />
452 </element> <!-- img --->
453 <element>
454     <label>p</label>
455     <attributes total="true" />
456     <children ordered="true" total="true"
457     >As discussed in <element>
458         <label>a</label>
459         <attributes total="true">
460             <attribute><label>href</label><value>#contributions</value></attribute>
461         </attributes>
462         <children ordered="true" total="true"> ... </children>
463     </element></children>
464 </element> <!-- p --->
465 <element>
466     <label>h1</label>
467     <attributes total="true">
468         <attribute><label>id</label><value>tachygraphy</value></attribute>
469     </attributes>
470     <children ordered="true" total="true">Challenges for Tachygraphy on Wax</children>
471 </element> <!-- hi --->
472 <element>
473     <label>p</label>
474     <attributes total="true" />
475     <children ordered="true" total="true"
476     >Though conditions for writing on wax tablets are adverse
477     to tachygraphy, systems as described in <element>
478         <label>a</label>
479         <attributes total="true">
480             <attribute><label>href</label><value>#tiro</value></attribute>
481         </attributes>
482         <children ordered="true" total="true"> ... </children>
483     </element></children>
484 </element> <!-- p --->
485 </children>
486 </element> <!-- html --->
487 </declare>
488 </children>
489 </element> <!-- content --->
490 </children>
491 </element> <!-- article --->
492
493 <element>
494     <identifier>inproc.44.brutus</identifier>
495     <label>inproceedings</label>
496     <attributes total="true" />
497     <children ordered="false" total="true">
498         <element>
499             <label>title</label>
500             <attributes total="true" />
501             <children ordered="true" total="true"
502             >Efficient Management of Rapidly Changing Personal Records</children>

```

```

503 </element>
504 <element>
505 <label>authors</label>
506 <attributes total="true" />
507 <children ordered="true" total="true">
508 <element>
509 <label>author</label>
510 <attributes total="true" />
511 <children ordered="true" total="true">
512 >Marcus Antonius<!-- -->
513 <element>
514 <label>affiliation</label>
515 <attributes total="true" />
516 <children ordered="true" total="true">
517 >Consul, SPQR</children>
518 </element>
519 </children>
520 </element>
521 <element>
522 <label>author</label>
523 <attributes total="true" />
524 <children ordered="true" total="true">
525 >Decimus Junius Brutus<!-- -->
526 <element>
527 <label>affiliation</label>
528 <attributes total="true" />
529 <children ordered="true" total="true">
530 >Governor, Cisalpine Gaul</children>
531 </element>
532 </children>
533 </element>
534 </children>
535 </element> <!-- authors -->
536 <element>
537 <label>in</label>
538 <attributes total="true">
539 <attribute<label>scrolls</label><value>24-48</value>
540 </attribute>
541 </attributes>
542 <children ordered="true" total="true">
543 <reference>conf.dmmc</reference>
544 </children>
545 </element>
546 <element>
547 <label>content</label>
548 <attributes total="true">
549 <attribute<label>type</label><value>docbook</value>
550 </attribute>
551 </attributes>
552 <children ordered="true" total="true">
553 <declare>
554 <ns-default<iri>http://example.org/ns/docbook/simplified/1.0</iri></ns-default>
555 <element>
556 <label>section</label>
557 <attributes total="true" />
558 <children ordered="true" total="true">
559 <element>
560 <label>info</label>
561 <attributes total="true" />
562 <children ordered="true" total="true">
563 <element>
564 <label>title</label>
565 <attributes total="true" />
566 <children ordered="true" total="true">Introduction</children>
567 </element>
568 </children>
569 </element>
570 </children>
571 </element>
572 <label>section</label>
573 <attributes total="true" />

```

```

573 <children ordered="true" total="true">
574 <element>
575 <label>info</label>
576 <attributes total="true" />
577 <children ordered="true" total="true">
578 <element>
579 <label>title</label>
580 <attributes total="true" />
581 <children ordered="true" total="true">Contributions</children>
582 </element>
583 </children>
584 </element>
585 <element>
586 <label>para</label>
587 <attributes total="true" />
588 <children ordered="true" total="true">
589 >The most notable contributions of this article
590 include:<element>
591 <label>list</label>
592 <attributes total="true">
593 <attribute><label>type</label><value>ordered</value></attribute>
594 </attributes>
595 <children ordered="true" total="true">
596 <element>
597 <label>item</label>
598 <attributes total="true" />
599 <children ordered="true" total="true">
600 <element>
601 <label>para</label>
602 <attributes total="true" />
603 <children ordered="true" total="true">
604 <element>
605 <label>em</label>
606 <attributes total="true" />
607 <children ordered="true" total="true">Clear Evidence</children>
608 </element> of the need ...</children>
609 </element> <!-- para -->
610 </children>
611 </element> <!-- item -->
612 </element>
613 <label>item</label>
614 <attributes total="true" />
615 <children ordered="true" total="true">
616 <element>
617 <label>para</label>
618 <attributes total="true" />
619 <children ordered="true" total="true">A new
620 <element>
621 <label>em</label>
622 <attributes total="true" />
623 <children ordered="true" total="true">methodology</children>
624 </element> to ..., cf. <element>
625 <label>pageref</label>
626 <attributes total="true">
627 <attribute><label>idref</label><value>inproc.44.brutus.s1</value></attribute>
628 </attributes>
629 <children ordered="true" total="true" />
630 </element></children>
631 </element> <!-- para -->
632 </element>
633 <label>figure</label>
634 <attributes total="true" />
635 <children ordered="true" total="true">
636 <element>
637 <label>title</label>
638 <attributes total="true" />
639 <children ordered="true" total="true">Chart of Desertions</children>
640 </element>
641 </element>
</label>img</label>

```

```

643         <attributes total="true" />
        <children ordered="true" total="true"> ... </children>
645     </element>
</children>
647 </element> <!-- figure -->
<element>
649     <label>para</label>
    <attributes total="true" />
651     <children ordered="true" total="true"
        >As <element>
653         <label>cite</label>
        <attributes total="true" />
655         <children ordered="true" total="true">
            <reference>article.66.cicero.wax</reference>
657         </children>
        </element> of the need ...</children>
659     </element> <!-- para -->
</children>
661 </element> <!-- item -->
</children>
663 </element> <!-- list -->
</children>
665 </element> <!-- para -->
</children>
667 </element> <!-- section -->
</children>
669 </element> <!-- section -->
<element>
671     <identifier>inproc.44.brutus.s1</identifier>
    <label>section</label>
673     <attributes total="true" />
    <children ordered="true" total="true">
675         <element>
            <label>info</label>
677             <attributes total="true" />
            <children ordered="true" total="true">
679                 <element>
                    <label>title</label>
681                     <attributes total="true" />
                    <children ordered="true"
683                         total="true">Acknowledgements</children>
                    </element>
                </children>
685            </element> <!-- info -->
687        <element>
            <label>para</label>
689            <attributes total="true" />
            <children ordered="true" total="true"
691                >We would like to thank the editors of <element>
                <label>cite</label>
693                <attributes total="true" />
                <children ordered="true" total="true">
695                    <reference>journal.adm.v10.n1</reference>
                </children>
                </element> ...
697            </children>
699        </element> <!-- para -->
</children>
701 </element> <!-- section -->
</declare>
</children>
703 </element> <!-- content -->
705 </children>
</element> <!-- inproceedings-->
707 </children>
</element>

```


Appendix D

ANTLR Grammar for Xcerpt 2.0

The following listing shows the grammar for Xcerpt 2.0 as used in the implementation currently under development (see <http://amachos.com/>). It is a grammar for the ANTLR 3 (<http://www.antlr.org/>) parser generator.

```
grammar XcerptGrammar;
2
3 options {
4   output=AST;
5   backtrack=true;
6   memoize=true;
7 }
8
9 tokens {
10  PROGRAM;
11  GOAL_BLOCK;
12  RULE_LEVEL_DECLARE_BLOCK;
13  RULE_LEVEL_DECLARE_BLOCK_GOAL;
14  RULE_LEVEL_DECLARE_BLOCK_RULE;
15  RULE_LEVEL_DECLARE_BLOCK_DATA;
16  RULE_LEVEL_DECLARE_BLOCK_DECLARE;
17  GOAL;
18  RULE;
19  OUT_RESOURCE;
20  OUT_RESOURCE_CONSTRUCT_TERM;
21  OUT_RESOURCE_IRI_LITERAL_VARIABLE;
22  DATA;
23  RULE_LEVEL_DECLARE;
24
25  QUERY_TERM;
26  QUERY_TERM_MODIFIED;
27  QUERY_TERM_DECLARE;
28  TERM_LEVEL_DECLARE_QT;
29  MODIFIED_QT;
30  MODIFIED_QT_VARIABLE_TERM;
31  MODIFIED_QT_LOCATION_MODIFIED;
32  MODIFIED_QT_OCCURRENCE_MODIFIED;
33  MODIFIED_QT_SELECTION_MODIFIED;
34  BASE_TERM_QT;
35  BASE_TERM_QT_REFERENCE;
36  BASE_TERM_QT_STRUCTURED;
37  BASE_TERM_QT_CONTENT;
38  REFERENCE_QT;
39  IDENTIFIER_QT;
40  IDENTIFIER_QT_NCNAME;
41  IDENTIFIER_QT_IRI;
42  IDENTIFIER_QT_STRING;
43  IDENTIFIER_QT_LITERAL_VARIABLE;
```

```

44 IDENTIFIER_QT_REGEX;
   NS_DECL_QT;
46 NS_PREFIX_DECL_QT;
   NS_DEFAULT_DECL_QT;
48 VARIABLE_DECL_QT;
   CONTENT_QT;
50 CONTENT_QT_LITERAL_CONTENT;
   CONTENT_QT_COMMENT;
52 CONTENT_QT_PROCESSING_INSTRUCTION;
   LITERAL_CONTENT_QT;
54 LITERAL_CONTENT_QT_STRING;
   LITERAL_CONTENT_QT_LITERAL_VARIABLE;
56 LITERAL_CONTENT_QT_REGEX;
   COMMENT_QT;
58 PROCESSING_INSTRUCTION_QT;
   STRUCTURED_QT;
60 CHILDREN_LIST_QT;
   CHILDREN_LIST_QT_BRACKETS;
62 CHILDREN_LIST_QT_BRACES;
   CHILDREN_LIST_QT_DOUBLEBRACKETS;
64 CHILDREN_LIST_QT_DOUBLEBRACES;
   CONDITION_CLAUSE_QT;
66 CONDITION_QT;
   CONDITION_QT_CONDITION_OP;
68 CONDITION_QT_AND;
   CONDITION_QT_OR;
70 CONDITION_QT_NOT;
   CONDITION_QT_C_PARAMETER;
72 CONDITION_OP;
   CONDITION_OP_EQUALS;
74 CONDITION_OP_NOT_EQUALS;
   CONDITION_OP_LESS_THAN;
76 CONDITION_OP_GREATER_THAN;
   CONDITION_OP_LESS_OR_EQUALS;
78 CONDITION_OP_GREATER_OR_EQUALS;
   ARITHMETIC_OP;
80 ARITHMETIC_OP_PLUS;
   ARITHMETIC_OP_MINUS;
82 ARITHMETIC_OP_MULTIPLY;
   ARITHMETIC_OP_DIVIDE;
84 ARITHMETIC_OP_POWER;
   C_PARAMETER;
86 C_PARAMETER_OPTIONAL_VARIABLE;
   C_PARAMETER_VARIABLE;
88 C_PARAMETER_STRING;
   C_PARAMETER_XCERPT_INT;
90 C_PARAMETER_ARITHMETIC_OP;
   OPTIONAL_VARIABLE_QT;
92 LOCAL_SPEC_QT;
   TERM_IDENTIFIER_QT;
94 NS_LABEL_QT;
   ATTR_TERM_LIST_QT;
96 ATTR_TERM_LIST_QT_PAREN;
   ATTR_TERM_LIST_QT_DOUBLEPAREN;
98 ATTR_TERM_LIST_QT_NOPAREN;
   ATTR_TERM_QT;
100 MODIFIED_ATTR_TERM_QT;
   MODIFIED_ATTR_TERM_QT_BASE_ATTR_TERM;
102 MODIFIED_ATTR_TERM_QT_VARIABLE_ATTR_TERM;
   MODIFIED_ATTR_TERM_QT_OCCURRENCE_MODIFIED_ATTR_TERM;
104 MODIFIED_ATTR_TERM_QT_SELECTION_MODIFIED_ATTR_TERM;
   BASE_ATTR_TERM_QT;
106 VARIABLE_TERM_QT;
   VARIABLE_TERM_QT_BASE_TERM;
108 VARIABLE_TERM_QT_TERM_VARIABLE;
   VARIABLE_ATTR_TERM_QT;
110 VARIABLE_QT;
   VARIABLE_QT_TERM_VARIABLE;
112 VARIABLE_QT_LITERAL_VARIABLE;
   TERM_VARIABLE_QT;

```



```

114 LITERAL_VARIABLE_QT;
    VAR_SPECIFICATION_QT;
116 VAR_SPECIFICATION_QT_NCNAME;
    VAR_SPECIFICATION_QT_ANONYMOUS_VARIABLE;
118 ANONYMOUS_VARIABLE;
    SELECTION_MODIFIED_QT;
120 SELECTION_MODIFIED_ATTR_TERM_QT;
    SELECTION_MODIFIER;
122 OCCURRENCE_MODIFIED_QT;
    OCCURRENCE_MODIFIED_ATTR_TERM_QT;
124 OCCURRENCE_MODIFIER;
    OCCURRENCE_MODIFIER_WITHOUT;
126 OCCURRENCE_MODIFIER_OPTIONAL_MODIFIER;
    LOCATION_MODIFIED_QT;
128 LOCATION_MODIFIER;
    LOCATION_MODIFIER_DESCENDANT_MODIFIER;
130 LOCATION_MODIFIER_POSITION_MODIFIER;
    DESCENDANT_MODIFIER;
132 POSITION_MODIFIER;
    NUMBER_QT;
134 NUMBER_QT_XCERPT_INT;
    NUMBER_QT_LITERAL_VARIABLE;
136 TOP_LEVEL_QUERY_TERM;
    TOP_LEVEL_QUERY_TERM_TOP_TERM_LEVEL_DECLARE;
138 TOP_LEVEL_QUERY_TERM_OPTIONAL_TOP_LEVEL;
    TOP_LEVEL_QUERY_TERM_TERM_FORMULA;
140 TOP_LEVEL_QUERY_TERM_DOCUMENT_SPECIFICATION;
    TOP_TERM_LEVEL_DECLARE_QT;
142 OPTIONAL_TOP_LEVEL_QT;
    OPTIONAL_TOP_LEVEL_QT_OPTIONAL_MODIFIER;
144 OPTIONAL_TOP_LEVEL_QT_DESCENDANT_TOP_LEVEL;
    DESCENDANT_TOP_LEVEL_QT;
146 DESCENDANT_TOP_LEVEL_QT_DESCENDANT_MODIFIER;
    DESCENDANT_TOP_LEVEL_QT_VAR_RESTRICTION_TOP_LEVEL;
148 VAR_RESTRICTION_TOP_LEVEL_QT;
    VAR_RESTRICTION_TOP_LEVEL_QT_TERM_VARIABLE;
150 VAR_RESTRICTION_TOP_LEVEL_QT_STRUCTURED;
    TERM_FORMULA_QT;
152 TERM_FORMULA_QT_NOT;
    TERM_FORMULA_QT_AND;
154 TERM_FORMULA_QT_OR;
    DOCUMENT_SPECIFICATION_QT;
156 XML_DOCUMENT_SPECIFICATION_QT;
    LOCATION_QT;
158 LOCATION_QT_IRI;
    LOCATION_QT_LITERAL_VARIABLE;
160 XML_DECL_QT;
    XML_DECL_QT_STANDALONE_TRUE;
162 XML_DECL_QT_STANDALONE_FALSE;
    XML_DECL_QT_STANDALONE_LITERAL_VARIABLE;
164 XML_DECL_QT_VERSION_1_0;
    XML_DECL_QT_VERSION_1_1;
166 XML_DECL_QT_VERSION_LITERAL_VARIABLE;
    DOCTYPE_DECL_QT;
168 XML_DOCUMENT_CHILDREN_QT;
    XML_DOCUMENT_CHILDREN_QT_DOUBLEBRACKETS;
170 XML_DOCUMENT_CHILDREN_QT_DOUBLEBRACES;
    XML_DOCUMENT_CHILDREN_QT_SINGLEBRACES;
172 XML_DOCUMENT_CHILDREN_QT_SINGLEBRACKETS;
    XML_DOCUMENT_CONTENT_QT;
174 ANNOTATION_CONTENT_QT;
    ANNOTATION_CONTENT_QT_COMMENT;
176 ANNOTATION_CONTENT_QT_PROCESSING_INSTRUCTION;

178 CONSTRUCT_TERM;
    CONSTRUCT_TERM_TERM_LEVEL_DECLARE;
180 CONSTRUCT_TERM_REFERENCE;
    CONSTRUCT_TERM_MODIFIED;
182 CONSTRUCT_TERM_STRUCTURED;
    CONSTRUCT_TERM_CONTENT;

```

```

184    CONSTRUCT_TERM_TERM_VARIABLE;
      REFERENCE_CT;
186    TERM_LEVEL_DECLARE_CT;
      IDENTIFIER_CT;
188    IDENTIFIER_CT_NCNAME;
      IDENTIFIER_CT_IRI;
190    IDENTIFIER_CT_STRING;
      IDENTIFIER_CT_LITERAL_VARIABLE;
192    NS_DECLARATION_CT;
      NS_PREFIX_DECLARATION_CT;
194    NS_DEFAULT_DECLARATION_CT;
      NS_DEFAULT_DECLARATION_CT_LITERAL_CONTENT;
196    NS_DEFAULT_DECLARATION_CT_COMMENT;
      NS_DEFAULT_DECLARATION_CT_PROCESSING_INSTRUCTION;
198    CONTENT_CT;
      CONTENT_CT_LITERAL_CONTENT;
200    CONTENT_CT_COMMENT;
      CONTENT_CT_PROCESSING_INSTRUCTION;
202    LITERAL_CONTENT_CT;
      LITERAL_CONTENT_CT_STRING;
204    LITERAL_CONTENT_CT_LITERAL_VARIABLE;
      COMMENT_CT;
206    PROCESSING_INSTRUCTION_CT;
      STRUCTURED_CT;
208    CHILDREN_LIST_CT;
      CHILDREN_LIST_CT_BRACKETS;
210    CHILDREN_LIST_CT_BRACES;
      LOCAL_SPEC_CT;
212    TERM_IDENTIFIER_CT;
      NS_LABEL_CT;
214    ATTR_TERM_LIST_CT;
      ATTR_TERM_CT;
216    ATTR_TERM_CT_BASE_ATTR_TERM;
      ATTR_TERM_CT_TERM_VARIABLE;
218    ATTR_TERM_CT_MODIFIED_ATTR_TERM;
      BASE_ATTR_TERM_CT;
220    TOP_LEVEL_CONSTRUCT_TERM;
      TOP_LEVEL_CONSTRUCT_TERM_TOP_TERM_LEVEL_DECLARE;
222    TOP_LEVEL_CONSTRUCT_TERM_STRUCTURED;
      TOP_TERM_LEVEL_DECLARE_CT;
224    VARIABLE_CT;
      VARIABLE_CT_TERM_VARIABLE;
226    VARIABLE_CT_LITERAL_VARIABLE;
      TERM_VARIABLE_CT;
228    LITERAL_VARIABLE_CT;
      VAR_SPECIFICATION_CT;
230    MODIFIED_CT;
      MODIFIED_CT_GROUPING;
232    MODIFIED_CT_OPTIONAL;
      MODIFIED_ATTR_TERM_CT;
234    MODIFIED_ATTR_TERM_CT_GROUPING_ATTR_TERM;
      MODIFIED_ATTR_TERM_CT_OPTIONAL_ATTR_TERM;
236    GROUPING_CT;
      GROUPING_ATTR_TERM_CT;
238    GROUPING_MODIFIER;
      GROUPING_MODIFIER_ALL;
240    GROUPING_MODIFIER_SOME;
      GROUPING_MODIFIER_FIRST;
242    ORDERBY;
      ORDERBY_LIST_OPTIONAL_VARIABLE;
244    ORDERBY_LIST_VARIABLE;
      ORDER_RELATION;
246    ORDER_RELATION_ASCENDING;
      ORDER_RELATION_DESCENDING;
248    ORDER_RELATION_NCNAME;
      GROUPBY;
250    GROUPBY_LIST_OPTIONAL_VARIABLE;
      GROUPBY_LIST_VARIABLE;
252    EQUIVALENCE_RELATION;
      OPTIONAL_VARIABLE;

```

```

254     INTERVAL_CT;
        INTERVAL_CT_FROM_TO;
256     INTERVAL_CT_FROM;
        INTERVAL_CT_PLUS;
258     NUMBER_CT;
        NUMBER_CT_XCERPT_INT;
260     NUMBER_CT_LITERAL_VARIABLE;
        OPTIONAL_CT;
262     OPTIONAL_CT_DEFAULT;
        OPTIONAL_ATTR_TERM_CT;
264     OPTIONAL_ATTR_TERM_CT_DEFAULT;
        OPTIONAL_MODIFIER;

266     DATA_TERM;
268     DATA_TERM_TERM_LEVEL_DECLARE;
        DATA_TERM_REFERENCE;
270     DATA_TERM_STRUCTURED;
        DATA_TERM_CONTENT;
272     REFERENCE_DT;
        TERM_LEVEL_DECLARE_DT;
274     IDENTIFIER_DT;
        IDENTIFIER_DT_NCNAME;
276     IDENTIFIER_DT_IRI;
        IDENTIFIER_DT_STRING;
278     NS_DECLARATION_DT;
        NS_PREFIX_DECLARATION_DT;
280     NS_DEFAULT_DECLARATION_DT;
        CONTENT_DT;
282     CONTENT_DT_LITERAL_CONTENT;
        CONTENT_DT_COMMENT;
284     CONTENT_DT_PROCESSING_INSTRUCTION;
        LITERAL_CONTENT_DT;
286     COMMENT_DT;
        PROCESSING_INSTRUCTION_DT;
288     STRUCTURED_DT;
        STRUCTURED_DT_LOCAL_SPEC;
290     STRUCTURED_DT_CHILDREN_LIST;
        CHILDREN_LIST_DT;
292     CHILDREN_LIST_DT_BRACKETS;
        CHILDREN_LIST_DT_BRACES;
294     LOCAL_SPEC_DT;
        TERM_IDENTIFIER_DT;
296     NS_LABEL_DT;
        ATTR_TERM_LIST_DT;
298     ATTR_TERM_DT;
        BASE_ATTR_TERM_DT;
300     TOP_LEVEL_DATA_TERM;
        TOP_LEVEL_DATA_TERM_TOP_TERM_LEVEL_DECLARE;
302     TOP_LEVEL_DATA_TERM_STRUCTURED;
        TOP_TERM_LEVEL_DECLARE_DT;

304     NCNAME;
306     LITERAL_NCNAME;
        IRI;
308     STRING;
        STRING_CHARACTER;
310     LINE_FEED;
        CARRIAGE_RETURN;
312     NUMBER;
        XCERPT_INT;
314     REGEXP;
        ERE_EXPRESSION;
316     WHITESPACE;
        COMMENT_CHAR;
318     END_OF_LINE;
        END_OF_LINE_COMMENT;
320     BLOCK_COMMENT;
    }
322 @parser::header{

```

```

324 package org.xcerpt.generated;
325 }
326
327 @lexer::header {
328 package org.xcerpt.generated;
329 }
330
331 /**
332 NOTE: All lines indented only one TAB exist only for less paranthesised syntax. You savely should be able to comment them out.
333 Lesser paranthesis also cause massive backtracking sometimes.
334 */
335
336 /* PROGRAMS */
337 program :
338     ('PROGRAM' ( '(' goal_block '))
339     | (goal_block
340         ) 'END')
341 → ^(PROGRAM goal_block)
342     ;
343
344 /*
345 goal_block :
346     rule_level_declare_block+
347 → ^(GOAL_BLOCK rule_level_declare_block+)
348     ; // to be LL(k)
349
350 rule_level_declare_block :
351     goal
352 → ^(RULE_LEVEL_DECLARE_BLOCK_GOAL goal)
353     | rule
354 → ^(RULE_LEVEL_DECLARE_BLOCK_RULE rule)
355     | data
356 → ^(RULE_LEVEL_DECLARE_BLOCK_DATA data)
357     | (rule_level_declare ( '(' rule_level_declare_block* ')')
358         | (rule_level_declare_block?)
359         ) 'END')
360 → ^(RULE_LEVEL_DECLARE_BLOCK_DECLARE rule_level_declare rule_level_declare_block*)
361     ;
362
363 goal :
364     'GOAL' ( '(' out_resource ')')
365     | (out_resource
366         ) 'FROM' ( '(' query_term ')')
367     | (query_term
368         ) 'END'
369 → ^(GOAL out_resource query_term)
370     ;
371
372 rule :
373     'CONSTRUCT' ( '(' construct_term ')')
374     | (construct_term
375         ) 'FROM' ( '(' query_term ')')
376     | (query_term
377         ) 'END'
378 → ^(RULE construct_term query_term)
379     ;
380
381 out_resource :
382     construct_term
383 → ^(OUT_RESOURCE_CONSTRUCT_TERM construct_term)
384     | ('out' (iri | literal_variable_qt) ( '(' construct_term ')')
385         | (construct_term
386         ) )
387 → ^(OUT_RESOURCE_IRI_LITERAL_VARIABLE construct_term iri? literal_variable_qt?)
388     ;
389
390 data :
391     'DATA' ( '(' data_term ')')

```

```

394     | (data_term
396     ) 'END'
398 → ^(DATA data_term
398     ;
398 rule_level_declare :
400     'DECLARE' ( ((' variable_decl_qt* ns_decl_qt* ')
402     | (variable_decl_qt? ns_decl_qt?)
404     )
404     → ^(RULE_LEVEL_DECLARE variable_decl_qt* ns_decl_qt*)
404     ;

406 /* QUERY TERMS */
406 query_term :
408     modified_qt
408     → ^(QUERY_TERM_MODIFIED modified_qt
410     | term_level_declare_qt
412     → ^(QUERY_TERM_DECLARE term_level_declare_qt)
412     ;

414 term_level_declare_qt : 'declare' ns_decl_qt variable_decl_qt ( ((' (query_term ',?)* ')
416     | (query_term?)
418     )
418     → ^(TERM_LEVEL_DECLARE_QT ns_decl_qt variable_decl_qt query_term*)
418     ;

420 modified_qt :
422     variable_term_qt
422     → ^(MODIFIED_QT_VARIABLE_TERM variable_term_qt
424     | location_modified_qt
424     → ^(MODIFIED_QT_LOCATION_MODIFIED location_modified_qt)
426     | occurrence_modified_qt
426     → ^(MODIFIED_QT_OCCURRENCE_MODIFIED occurrence_modified_qt)
428     | selection_modified_qt
428     → ^(MODIFIED_QT_SELECTION_MODIFIED)
430     ;

430 base_term_qt :
432     reference_qt
432     → ^(BASE_TERM_QT_REFERENCE reference_qt)
434     | structured_qt
434     → ^(BASE_TERM_QT_STRUCTURED structured_qt)
436     | content_qt
436     → ^(BASE_TERM_QT_CONTENT content_qt)
438     ;

440 reference_qt :
442     '^' identifier_qt
442     → ^(REFERENCE_QT identifier_qt)
444     ;

444 identifier_qt :
446     ncname
446     → ^(IDENTIFIER_QT_NCNAME ncname)
448     | iri
448     → ^(IDENTIFIER_QT_IRI iri)
450     | string
450     → ^(IDENTIFIER_QT_STRING string)
452     | literal_variable_qt
452     → ^(IDENTIFIER_QT_LITERAL_VARIABLE literal_variable_qt)
454     | regexp
454     → ^(IDENTIFIER_QT_REGEXP regexp)
456     ;

458 ns_decl_qt :
460     ( ns_prefix_decl_qt ',?' )* ( ns_default_decl_qt ',?' )?
460     ( ns_prefix_decl_qt ',?' )*
462     → ^(NS_DECL_QT ns_default_decl_qt? ns_prefix_decl_qt*)
462     ;

```

```

464 ns_prefix_decl_qt :
      'ns-prefix' identifier_qt '=' ( iri | literal_variable_qt )
466 → ^(NS_PREFIX_DECL_QT identifier_qt iri? literal_variable_qt?)
      ;
468
468 ns_default_decl_qt :
      'ns_default' ( iri | literal_variable_qt )
470 → ^(NS_DEFAULT_DECL_QT iri? literal_variable_qt?)
472 ;
474
474 bughelper1 : ('variable'|'var');
variable_decl_qt :
476 → ^ (VARIABLE_DECL_QT ncname)
478 ;
480
480 content_qt :
      literal_content_qt
482 → ^(CONTENT_QT_LITERAL_CONTENT literal_content_qt)
      | comment_qt
484 → ^(CONTENT_QT_COMMENT comment_qt)
      | processing_instruction_qt
486 → ^(CONTENT_QT_PROCESSING_INSTRUCTION processing_instruction_qt)
      ;
488
488 literal_content_qt :
      string
490 → ^(LITERAL_CONTENT_QT_STRING string)
      | literal_variable_qt
492 → ^(LITERAL_CONTENT_QT_LITERAL_VARIABLE literal_variable_qt)
      | regexp
494 → ^(LITERAL_CONTENT_QT_REGEXP regexp)
496 ;
496 comment_qt : 'xcerpt' ':' 'comment' ('(' ')')
498 ?
      '[' literal_content_qt ']'
500 → ^(COMMENT_QT literal_content_qt)
      ;
502
502 processing_instruction_qt
504 : ( 'xcerpt' ':' 'processing-instruction' | 'xcerpt' ':' 'pi' )
      ( ('(' 'target-name' '=' identifier_qt ')')
506 | ('target-name' '=' identifier_qt)
      ) '[' literal_content_qt ']'
508 → ^(PROCESSING_INSTRUCTION_QT identifier_qt literal_content_qt)
      ;
510
510 structured_qt :
512 → ^(STRUCTURED_QT local_spec_qt children_list_qt condition_clause_qt?)
514 ;
516
516 children_list_qt :
      ('[' (query_term ';?)* ']')
518 → ^(CHILDREN_LIST_QT_BRACKETS query_term*)
      | ('{' (query_term ';?)* '}')
520 → ^(CHILDREN_LIST_QT_BRACES query_term*)
      | ('[[' (query_term ';?)* ']]')
522 → ^(CHILDREN_LIST_QT_DOUBLEBRACKETS query_term*)
      | ('{{{ (query_term ';?)* }}}')
524 → ^(CHILDREN_LIST_QT_DOUBLEBRACES query_term*)
      ;
526
526 condition_clause_qt :
528 → ^ (CONDITION_CLAUSE_QT 'where' ( ('(' condition_qt ')')
      | (condition_qt)
530 )
      )
532 → ^(CONDITION_CLAUSE_QT condition_qt)
      ;

```

```

534 condition_qt :
      (c_parameter condition_op c_parameter)
536 → ^(CONDITION_QT_CONDITION_OP condition_op c_parameter+)
      | (condition_op '(' c_parameter c_parameter ')')
538 → ^(CONDITION_QT_CONDITION_OP condition_op c_parameter+)
      | ('and' '(' condition_qt condition_qt+ ')')
540 → ^(CONDITION_QT_AND condition_qt+)
      | ('or' '(' condition_qt condition_qt+ ')')
542 → ^(CONDITION_QT_OR condition_qt+)
      | ('not' '(' condition_qt ')')
544 → ^(CONDITION_QT_NOT condition_qt)
      | c_parameter
546 → ^(CONDITION_QT_C_PARAMETER c_parameter)
      ;

548 condition_op :
550 → ^('==')
      | '^(')
552 → ^('!=')
      | '^(')
554 → ^('<')
      | '^(')
556 → ^('>')
      | '^(')
558 → ^('<=')
      | '^(')
560 → ^('>=')
      | '^(')
562 → ^('>=')
      ;

564 arithmetic_op :
566 → ^('+')
      | '^(')
568 → ^('-')
      | '^(')
570 → ^('*')
      | '^(')
572 → ^('/')
      | '^(')
574 → ^('^')
      | '^(')
576 → ^('^')
      ;

578 c_parameter :
580 → ^('optional_variable_qt')
      | '^(')
582 → ^('variable_qt')
      | '^(')
584 → ^('string')
      | '^(')
586 → ^('xcerpt_int')
      | '^(')
588 // rule below extended by parenthesis to avoid leftrecursiveness —not conforming to syntax definition
      | '^(' c_parameter arithmetic_op c_parameter ')')
590 → ^('arithmetic_op '(' c_parameter c_parameter ')')
      | '^(')
592 → ^('arithmetic_op arithmetic_op c_parameter+')
      ;

592 optional_variable_qt :
594 → ^('optional_modifier variable_qt')
      | '^(')
596 → ^('optional_modifier variable_qt')
      ;

598 local_spec_qt :
600 → ^('term_identifiaer_qt? ns_label_qt attr_term_list_qt')
      | '^(')
602 → ^('term_identifiaer_qt? ns_label_qt attr_term_list_qt term_identifiaer_qt?')
      ;

602 term_identifiaer_qt :

```

```

604         identifier_qt '@'
        → ^(TERM_IDENTIFIER_QT identifier_qt)
606         ;

608 ns_label_qt :
        (left=identifier_qt ':')? right=identifier_qt
610 → ^(NS_LABEL_QT $right $left?)
        ;

612

614 attr_term_list_qt : ( '(' (attr_term_qt ';?)* ')' )
        → ^(ATTR_TERM_LIST_QT_PAREN attr_term_qt*)
616         | ( '(' (attr_term_qt ';?)* ')' )
        → ^(ATTR_TERM_LIST_QT_DOUBLEPAREN attr_term_qt*)
618         | ( attr_term_qt? )
        → ^(ATTR_TERM_LIST_QT_NOPAREN attr_term_qt?)
620         ;

622 attr_term_qt :
        modified_attr_term_qt
624 → ^(ATTR_TERM_QT modified_attr_term_qt)
        ;

626 modified_attr_term_qt :
        base_attr_term_qt
628 → ^(MODIFIED_ATTR_TERM_QT_BASE_ATTR_TERM base_attr_term_qt)
        | variable_attr_term_qt
        → ^(MODIFIED_ATTR_TERM_QT_VARIABLE_ATTR_TERM variable_attr_term_qt)
630         | occurrence_modified_attr_term_qt
        → ^(MODIFIED_ATTR_TERM_QT_OCCURRENCE_MODIFIED_ATTR_TERM occurrence_modified_attr_term_qt)
632         | selection_modified_attr_term_qt
        → ^(MODIFIED_ATTR_TERM_QT_SELECTION_MODIFIED_ATTR_TERM selection_modified_attr_term_qt)
634         ;
636

638 base_attr_term_qt :
        ns_label_qt '=' literal_content_qt
640 → ^(BASE_ATTR_TERM_QT ns_label_qt literal_content_qt)
        ;

642 variable_term_qt :
        base_term_qt
644 → ^(VARIABLE_TERM_QT_BASE_TERM base_term_qt)
        | term_variable_qt ('-' base_term_qt)?
646 → ^(VARIABLE_TERM_QT_TERM_VARIABLE term_variable_qt base_term_qt?)
648         ;

650 variable_attr_term_qt :
        term_variable_qt ('-' base_attr_term_qt)?
652 → ^(VARIABLE_ATTR_TERM_QT term_variable_qt base_attr_term_qt?)
        ;

654 variable_qt :
        term_variable_qt
656 → ^(VARIABLE_QT_TERM_VARIABLE term_variable_qt)
        | literal_variable_qt
658 → ^(VARIABLE_QT_LITERAL_VARIABLE literal_variable_qt)
660         ;

662 term_variable_qt :
        var_specification_qt
664 → ^(TERM_VARIABLE_QT var_specification_qt)
        ;

666 literal_variable_qt :
        var_specification_qt
668 → ^(LITERAL_VARIABLE_QT var_specification_qt)
670         ;

672 bughelper2 : 'variable'|'var' ;
        var_specification_qt : ( bughelper2? nname)

```



```

674 → ^(VAR_SPECIFICATION_QT_NCNAME ncname
      | anonymous_variable
676 → ^(VAR_SPECIFICATION_QT_ANONYMOUS_VARIABLE)
      ;
678 anonymous_variable : '_'
680 → ^(ANONYMOUS_VARIABLE)
      ;
682 selection_modified_qt :
684     selection_modifier ( (' (modified_qt ',?)* ')
      | (modified_qt?)
686     )
688 → ^(SELECTION_MODIFIED_QT selection_modifier modified_qt*)
      ;

690 selection_modified_attr_term_qt
      : selection_modifier ( (' (modified_attr_term_qt ',?)* ')
692     | (modified_attr_term_qt?)
      )
694 → ^(SELECTION_MODIFIED_ATTR_TERM_QT selection_modifier modified_attr_term_qt*)
      ;
696 selection_modifier :
698     'except'
700 → ^(SELECTION_MODIFIER)
      ;

702 occurrence_modified_qt :
704     occurrence_modifier ( (' (modified_qt ',?)* ')
      | (modified_qt?)
706     )
708 → ^(OCCURRENCE_MODIFIED_QT occurrence_modifier modified_qt*)
      ;

710 occurrence_modified_attr_term_qt :
712     occurrence_modifier ( (' (modified_attr_term_qt ',?)* ')
      | (modified_attr_term_qt?)
714     )
716 → ^(OCCURRENCE_MODIFIED_ATTR_TERM_QT occurrence_modifier modified_attr_term_qt*)
      ;

716 occurrence_modifier :
718     'without'
720 → ^(OCCURRENCE_MODIFIER_WITHOUT)
      | optional_modifier
722 → ^(OCCURRENCE_MODIFIER_OPTIONAL_MODIFIER)
      ;

722 // Below was a TYPO in the syntax appendix
724 location_modified_qt : location_modifier ( (' (variable_term_qt ',?)* ')
726     | (variable_term_qt?)
728     )
730 → ^(LOCATION_MODIFIED_QT location_modifier variable_term_qt*)
      ;

730 location_modifier :
732     descendant_modifier
734 → ^(LOCATION_MODIFIER_DESCENDANT_MODIFIER descendant_modifier)
      | position_modifier
736 → ^(LOCATION_MODIFIER_POSITION_MODIFIER position_modifier)
      ;

736 descendant_modifier :
738     'descendant'
740 → ^(DESCENDANT_MODIFIER)
      | 'desc'
742 → ^(DESCENDANT_MODIFIER)
      ;

```

```

744 bughelper3 : 'position' | 'pos' ;
      position_modifier : bughelper3 number_qt
746 → ^(POSITION_MODIFIER number_qt)
      ;
748
      number_qt :
750           xcerpt_int
      → ^(NUMBER_QT_XCERPT_INT xcerpt_int)
752           | literal_variable_qt
      → ^(NUMBER_QT_LITERAL_VARIABLE literal_variable_qt)
754           ;

756 top_level_query_term :
      top_term_level_declare_qt
758 → ^(TOP_LEVEL_QUERY_TERM_TOP_TERM_LEVEL_DECLARE top_term_level_declare_qt)
      | optional_top_level_qt
760 → ^(TOP_LEVEL_QUERY_TERM_OPTIONAL_TOP_LEVEL optional_top_level_qt)
      | term_formula_qt
762 → ^(TOP_LEVEL_QUERY_TERM_TERM_FORMULA term_formula_qt)
      | document_specification_qt
764 → ^(TOP_LEVEL_QUERY_TERM_DOCUMENT_SPECIFICATION document_specification_qt)
      ;

766 top_term_level_declare_qt
768 : 'declare' ns_decl_qt variable_decl_qt ( '(' top_level_construct_term ')' )
      | (top_level_construct_term)
770 )
772 → ^(TOP_TERM_LEVEL_DECLARE_QT ns_decl_qt variable_decl_qt top_level_construct_term)
      ;

774 optional_top_level_qt :
      (optional_modifier ( '(' descendant_top_level_qt ')' )
776 | (descendant_top_level_qt)
      )
778 → ^(OPTIONAL_TOP_LEVEL_QT_OPTIONAL_MODIFIER optional_modifier descendant_top_level_qt)
      | descendant_top_level_qt
780 → ^(OPTIONAL_TOP_LEVEL_QT_DESCENDANT_TOP_LEVEL descendant_top_level_qt)
      ;

782 descendant_top_level_qt : (descendant_modifier ( '(' var_restriction_top_level_qt ')' )
784 | (var_restriction_top_level_qt)
      )
786 → ^(DESCENDANT_TOP_LEVEL_QT_DESCENDANT_MODIFIER descendant_modifier var_restriction_top_level_qt)
      | var_restriction_top_level_qt
788 → ^(DESCENDANT_TOP_LEVEL_QT_VAR_RESTRICTION_TOP_LEVEL var_restriction_top_level_qt)
      ;

790 var_restriction_top_level_qt :
792 (term_variable_qt '→' structured_qt )
      → ^(VAR_RESTRICTION_TOP_LEVEL_QT_TERM_VARIABLE term_variable_qt structured_qt)
794 | structured_qt
      → ^(VAR_RESTRICTION_TOP_LEVEL_QT_STRUCTURED structured_qt)
796 ;

798 term_formula_qt :
      ( 'not' ( '(' top_level_query_term ')' )
800 | (top_level_query_term)
      ) condition_clause_qt? )
802 → ^(TERM_FORMULA_QT_NOT top_level_query_term condition_clause_qt?)
      | ( 'and' ( '(' top_level_query_term ','? (top_level_query_term ','?)+ ')' )
804 | (top_level_query_term ','? top_level_query_term)
      ) condition_clause_qt? )
806 → ^(TERM_FORMULA_QT_AND top_level_query_term+ condition_clause_qt?)
      | ( 'or' ( '(' top_level_query_term ','? (top_level_query_term ','?)+ ')' )
808 | (top_level_query_term ','? top_level_query_term)
      ) condition_clause_qt? )
810 → ^(TERM_FORMULA_QT_OR top_level_query_term+ condition_clause_qt?)
812 ;

```

```

814 document_specification_qt :
      'in' xml_document_specification_qt
816 → ^(DOCUMENT_SPECIFICATION_QT xml_document_specification_qt)
      ;
818
xml_document_specification_qt :
820   'xml-document' ( (('('location_qt xml_decl_qt? doctype_decl_qt? ')'))
      | (location_qt xml_decl_qt? doctype_decl_qt?)
822     ) xml_document_children_qt
→ ^(XML_DOCUMENT_SPECIFICATION_QT location_qt xml_document_children_qt
824   xml_decl_qt? doctype_decl_qt?)
      ;
826
location_qt :
828   'location' '=' ( iri
→ ^(LOCATION_QT_IRI iri)
830     | literal_variable_qt)
→ ^(LOCATION_QT_LITERAL_VARIABLE literal_variable_qt)
832   ;
834
xml_decl_qt_standalone :
      'standalone' '=' (
836   'true'
→ ^(XML_DECL_QT_STANDALONE_TRUE)
838   | 'false'
→ ^(XML_DECL_QT_STANDALONE_FALSE)
840   | literal_variable_qt
→ ^(XML_DECL_QT_STANDALONE_LITERAL_VARIABLE literal_variable_qt)
842   )
      ;
844
xml_decl_qt_version :
846   'xml-version' '=' (
      '1.0'
848 → ^(XML_DECL_QT_VERSION_1_0)
      | '1.1'
850 → ^(XML_DECL_QT_VERSION_1_1)
      | literal_variable_qt
852 → ^(XML_DECL_QT_VERSION_LITERAL_VARIABLE literal_variable_qt)
      )
854   ;
856
xml_decl_qt :
      xml_decl_qt_standalone? xml_decl_qt_version?
858 → ^(XML_DECL_QT xml_decl_qt_standalone? xml_decl_qt_version?)
860   ;
862
doctype_decl_qt : ('system-id' '=' sysid=identifier_qt)?
      ('public-id' '=' pubid=identifier_qt)?
864   ('root-name' '=' rootid=identifier_qt)?
→ ^(DOCTYPE_DECL_QT $sysid? $pubid? $rootid?)
866   ;
868
xml_document_children_qt
      : ( '[' xml_document_content_qt ']' )
870 → ^(XML_DOCUMENT_CHILDREN_QT_DOUBLEBRACKETS xml_document_content_qt)
      | ( '{' xml_document_content_qt '}' )
872 → ^(XML_DOCUMENT_CHILDREN_QT_DOUBLEBRACES xml_document_content_qt)
      | ( '[' xml_document_content_qt ']' )
874 → ^(XML_DOCUMENT_CHILDREN_QT_SINGLEBRACES xml_document_content_qt)
      | ( '[' xml_document_content_qt ']' )
876 → ^(XML_DOCUMENT_CHILDREN_QT_SINGLEBRACKETS xml_document_content_qt)
      ;
878
xml_document_content_qt :
880   (annotation_content_qt ';?)* top_level_query_term
      (annotation_content_qt ';?)*
882 → ^(XML_DOCUMENT_CONTENT_QT top_level_query_term annotation_content_qt*)
      ;

```

```

884 annotation_content_qt :
886     comment_qt
→ ^(ANNOTATION_CONTENT_QT_COMMENT comment_qt)
888     | processing_instruction_qt
→ ^(ANNOTATION_CONTENT_QT_PROCESSING_INSTRUCTION processing_instruction_qt)
890     ;

892 /* CONSTRUCT TERMS */
894 construct_term :
     term_level_declare_ct
896 → ^(CONSTRUCT_TERM_TERM_LEVEL_DECLARE term_level_declare_ct)
     | reference_ct
898 → ^(CONSTRUCT_TERM_REFERENCE reference_ct)
     | modified_ct
900 → ^(CONSTRUCT_TERM_MODIFIED modified_ct)
     | structured_ct
902 → ^(CONSTRUCT_TERM_STRUCTURED structured_ct)
     | content_ct
904 → ^(CONSTRUCT_TERM_CONTENT content_ct)
     | term_variable_ct
906 → ^(CONSTRUCT_TERM_TERM_VARIABLE term_variable_ct)
     ;

908 reference_ct :
910     '^' identifier_ct
→ ^(REFERENCE_CT identifier_ct)
912     ;

914 term_level_declare_ct :
     'declare' ns_declaration_ct ( ((' (construct_term ',?)* ')')
916     | (construct_term?)
     )
918 → ^(TERM_LEVEL_DECLARE_CT ns_declaration_ct construct_term*)
     ;

920 identifier_ct :
922     ncname
→ ^(IDENTIFIER_CT_NCNAME ncname)
924     | iri
→ ^(IDENTIFIER_CT_IRI iri)
926     | string
→ ^(IDENTIFIER_CT_STRING string)
928     | literal_variable_ct
→ ^(IDENTIFIER_CT_LITERAL_VARIABLE literal_variable_ct)
930     ;

932 ns_declaration_ct :
     (ns_prefix_declaration_ct ',?)*
934     (ns_default_declaration_ct ',?)?
     (ns_prefix_declaration_ct ',?)*
936 → ^(NS_DECLARATION_CT ns_default_declaration_ct? ns_prefix_declaration_ct*)
     ;

938 ns_prefix_declaration_ct :
940     'ns-prefix' identifier_ct '=' (
     iri
942 → ^(NS_PREFIX_DECLARATION_CT identifier_ct iri)
     | literal_variable_ct)
944 → ^(NS_PREFIX_DECLARATION_CT identifier_ct literal_variable_ct)
     ;

946 ns_default_declaration_ct :
948     literal_content_ct
→ ^(NS_DEFAULT_DECLARATION_CT_LITERAL_CONTENT literal_content_ct)
     | comment_ct
950 → ^(NS_DEFAULT_DECLARATION_CT_COMMENT comment_ct)
     | processing_instruction_ct
952 → ^(NS_DEFAULT_DECLARATION_CT_PROCESSING_INSTRUCTION processing_instruction_ct)

```

```

954         ;
956 content_ct :
957     literal_content_ct
958 → ^(CONTENT_CT_LITERAL_CONTENT literal_content_ct
959     | comment_ct
960 → ^(CONTENT_CT_COMMENT comment_ct)
961     | processing_instruction_ct
962 → ^(CONTENT_CT_PROCESSING_INSTRUCTION processing_instruction_ct)
963     ;
964 literal_content_ct :
965     string
966 → ^(LITERAL_CONTENT_CT_STRING string)
967     | literal_variable_ct
968 → ^(LITERAL_CONTENT_CT_LITERAL_VARIABLE literal_variable_ct)
969     ;
970
972 comment_ct :
973     'xcerpt' ':' 'comment' ( '(' ')' )
974     ?
975     // | (
976         ) '[' literal_content_ct ']'
977 → ^(COMMENT_CT literal_content_ct)
978     ;
979
982 processing_instruction_ct :
983     ('xcerpt' ':' 'processing-instruction') | ('xcerpt' ':' 'pi')
984     ( '(' 'target-name' '=' identifier_ct ')'
985     | ('target-name' '=' identifier_ct
986     ) '[' literal_content_ct ']'
987 → ^(PROCESSING_INSTRUCTION_CT identifier_ct literal_content_ct)
988     ;
989
992 structured_ct :
993     local_spec_ct children_list_ct
994 → ^(STRUCTURED_CT local_spec_ct children_list_ct)
995     ;
996
998 children_list_ct :
999     ('[' (construct_term '?'* ']')
1000 → ^(CHILDREN_LIST_CT_BRACKETS construct_term*)
1001     | ('{' (construct_term '?'* '}')
1002 → ^(CHILDREN_LIST_CT_BRACES construct_term*)
1003     ;
1004
1006 local_spec_ct :
1007     term_identifier_ct? ns_label_ct attr_term_list_ct
1008 → ^(LOCAL_SPEC_CT ns_label_ct attr_term_list_ct term_identifier_ct?)
1009     ;
1010
1012 term_identifier_ct :
1013     identifier_ct '@'
1014 → ^(TERM_IDENTIFIER_CT identifier_ct)
1015     ;
1016
1018 ns_label_ct :
1019     (ns=identifier_ct ':')? val=identifier_ct
1020 → ^(NS_LABEL_CT $val $ns?)
1021     ;
1022
1024 attr_term_list_ct :
1025     ('(' (attr_term_ct '?'* ')')
1026     | (attr_term_ct? )
1027 → ^(ATTR_TERM_LIST_CT attr_term_ct*)
1028     ;
1029
1031 attr_term_ct :
1032     base_attr_term_ct
1033 → ^(ATTR_TERM_CT_BASE_ATTR_TERM base_attr_term_ct)

```

```

1024         | term_variable_ct
→ ^(ATTR_TERM_CT_TERM_VARIABLE term_variable_ct)
1026         | modified_attr_term_ct
→ ^(ATTR_TERM_CT_MODIFIED_ATTR_TERM modified_attr_term_ct)
1028         ;

1030 base_attr_term_ct :
        ns_label_ct '=' literal_content_ct
1032 → ^(BASE_ATTR_TERM_CT ns_label_ct literal_content_ct)
        ;

1034 top_level_construct_term :
        top_term_level_declare_ct
1036 → ^(TOP_LEVEL_CONSTRUCT_TERM_TOP_TERM_LEVEL_DECLARE top_term_level_declare_ct)
        | structured_ct
1038 → ^(TOP_LEVEL_CONSTRUCT_TERM_STRUCTURED structured_ct)
1040         ;

1042 top_term_level_declare_ct :
        'declare' ns_declaration_ct ( ((' top_level_construct_term ')
1044         | (top_level_construct_term)
        )
1046 → ^(TOP_TERM_LEVEL_DECLARE_CT ns_declaration_ct top_level_construct_term)
        ;

1048 variable_ct :
        term_variable_ct
1050 → ^(VARIABLE_CT_TERM_VARIABLE term_variable_ct)
        | literal_variable_ct
1052 → ^(VARIABLE_CT_LITERAL_VARIABLE literal_variable_ct)
1054         ;

1056 term_variable_ct :
        var_specification_ct
1058 → ^(TERM_VARIABLE_CT var_specification_ct)
        ;

1060 literal_variable_ct :
        var_specification_ct
1062 → ^(LITERAL_VARIABLE_CT var_specification_ct)
1064         ;

1066 bughelper4 : 'variable' | 'var' ;
var_specification_ct :
1068         bughelper4? ncname
→ ^(VAR_SPECIFICATION_CT ncname)
1070         ;

1072 modified_ct :
        grouping_ct
1074 → ^(MODIFIED_CT_GROUPING grouping_ct)
        | optional_ct
1076 → ^(MODIFIED_CT_OPTIONAL optional_ct)
1078         ;

modified_attr_term_ct :
1080         grouping_attr_term_ct
→ ^(MODIFIED_ATTR_TERM_CT_GROUPING_ATTR_TERM grouping_attr_term_ct)
1082         | optional_attr_term_ct
→ ^(MODIFIED_ATTR_TERM_CT_OPTIONAL_ATTR_TERM optional_attr_term_ct)
1084         ;

1086 grouping_ct :
        grouping_modifier ( ((' (construct_term '?')* ')
1088         | ( construct_term? )
        ) groupby? orderby?
1090 → ^(GROUPING_CT grouping_modifier construct_term* groupby? orderby?)
1092         ;

```

```

1094 grouping_attr_term_ct :
           grouping_modifier ( ( '(' attr_term_ct? ')' )
1096         | ( attr_term_ct? )
           ) groupby? orderby?
1098 → ^(GROUPING_ATTR_TERM_CT grouping_modifier attr_term_ct? groupby? orderby?)
           ;
1100
1102 grouping_modifier :
           'all'
1104 → ^(GROUPING_MODIFIER_ALL
           | ('some' number_ct)
1106 → ^(GROUPING_MODIFIER_SOME number_ct)
           | ('first' interval_ct)
1108 → ^(GROUPING_MODIFIER_FIRST interval_ct)
           ;
1110
           orderby_list :
1112         optional_variable
           → ^(ORDERBY_LIST_OPTIONAL_VARIABLE optional_variable)
1114         | variable_ct
           → ^(ORDERBY_LIST_VARIABLE variable_ct)
1116         ;
1118
           orderby :
           'order-by' ( '(' ( orderby_list '?')* ')' )
1120         | ( orderby_list? )
           ) order_relation?
1122 → ^(ORDERBY orderby_list* order_relation?)
           ;
1124
           order_relation :
           'ascending'
1128 → ^(ORDER_RELATION_ASCENDING)
           | 'descending'
1130 → ^(ORDER_RELATION_DESCENDING)
           | nname
1132 → ^(ORDER_RELATION_NCNAME nname)
           ;
1134
           groupby_list :
1136         optional_variable
           → ^(GROUPBY_LIST_OPTIONAL_VARIABLE optional_variable)
1138         | variable_ct
           → ^(GROUPBY_LIST_VARIABLE variable_ct)
1140         ;
1142
           groupby :
           'group-by' ( '(' ( groupby_list '?')* ')' )
1144         | ( groupby_list? )
           ) equivalence_relation?
1146 → ^(GROUPBY groupby_list* equivalence_relation?)
           ;
1148
           optional_variable :
           optional_modifier variable_ct
1152 → ^(OPTIONAL_VARIABLE optional_modifier variable_ct)
           ;
1154
           equivalence_relation :
           nname
1156 → ^(EQUIVALENCE_RELATION nname)
           ;
1158
           interval_ct :
           (number_ct '-' number_ct)
1160 → ^(INTERVAL_CT_FROM_TO number_ct+)
           | (number_ct '-')

```

```

1164 → ^(INTERVAL_CT_FROM number_ct)
      | '+'
1166 → ^(INTERVAL_CT_PLUS)
      ;
1168 number_ct :
1170         xcerpt_int
      → ^(NUMBER_CT_XCERPT_INT xcerpt_int)
      | literal_variable_ct
1172 → ^(NUMBER_CT_LITERAL_VARIABLE literal_variable_ct)
1174 ;

1176 optional_ct_default :
      ('with-default' ( '(' (construct_term ',?)* ')' )
      | (construct_term?)
      )
1178 → ^(OPTIONAL_CT_DEFAULT construct_term*)
      ;
1182 optional_ct :
1184         optional_modifier ( '(' (construct_term ',?)* ')' )
      | (construct_term?)
      ) optional_ct_default?
1186 → ^(OPTIONAL_CT optional_modifier construct_term* optional_ct_default?)
1188 ;

1190 optional_attr_term_ct_default :
1192         ('with-default' ( '(' (attr_term_ct ',?)* ')' )
      | (attr_term_ct?)
      )
1194 → ^(OPTIONAL_ATTR_TERM_CT_DEFAULT attr_term_ct*)
1196 ;

1198 optional_attr_term_ct :
      optional_modifier ( '(' (attr_term_ct ',?)* ')' )
1200 | (attr_term_ct?)
      ) optional_attr_term_ct_default?
1202 → ^(OPTIONAL_ATTR_TERM_CT optional_modifier attr_term_ct* optional_attr_term_ct_default?)
      ;
1204 optional_modifier :
1206         'optional'
      → ^(OPTIONAL_MODIFIER)
1208         | 'opt'
      → ^(OPTIONAL_MODIFIER)
1210 ;

1212 /* DATA TERMS */
1214 data_term :
      term_level_declare_dt
1216 → ^(DATA_TERM_TERM_LEVEL_DECLARE term_level_declare_dt)
      | reference_dt
1218 → ^(DATA_TERM_REFERENCE reference_dt)
      | structured_dt
1220 → ^(DATA_TERM_STRUCTURED structured_dt)
      | content_dt
1222 → ^(DATA_TERM_CONTENT content_dt)
      ;
1224 reference_dt :
1226         '^' identifier_dt
      → ^(REFERENCE_DT identifier_dt)
1228 ;

1230 term_level_declare_dt :
      'declare' ns_declaration_dt ( '(' (data_term ',?)* ')' )
1232 | (data_term?)
      )

```



```

1234 → ^(TERM_LEVEL_DECLARE_DT ns_declaration_dt data_term*)
      ;
1236 identifier_dt :
1238         ncname
      → ^(IDENTIFIER_DT_NCNAME ncname)
1240         | iri
      → ^(IDENTIFIER_DT_IRI iri)
1242         | string
      → ^(IDENTIFIER_DT_STRING string)
1244         ;

1246 ns_declaration_dt :
1248         (ns_prefix_declaration_dt ',?)*
         (ns_default_declaration_dt ',?)?
         (ns_prefix_declaration_dt ',?)*
1250 → ^(NS_DECLARATION_DT ns_default_declaration_dt? ns_prefix_declaration_dt*)
      ;

1252 ns_prefix_declaration_dt :
1254         'ns-prefix' identifier_dt '=' iri
      → ^(NS_PREFIX_DECLARATION_DT identifier_dt iri)
1256         ;

1258 ns_default_declaration_dt :
1260         'ns-default' iri
      → ^(NS_DEFAULT_DECLARATION_DT iri)
1262         ;

1264 content_dt :
1266         literal_content_dt
      → ^(CONTENT_DT_LITERAL_CONTENT literal_content_dt)
1268         | comment_dt
      → ^(CONTENT_DT_COMMENT comment_dt)
1270         | processing_instruction_dt
      → ^(CONTENT_DT_PROCESSING_INSTRUCTION processing_instruction_dt)
      ;

1272 literal_content_dt :
1274         string
      → ^(LITERAL_CONTENT_DT string)
      ;

1276 comment_dt :
1278         'xcerpt' ':' 'comment' ( '(' ' ' )
      ?
1280         ) '[' literal_content_dt ']'
      → ^(COMMENT_DT literal_content_dt)
1282         ;

1284 processing_instruction_dt :
1286         ('xcerpt' ':' 'processing-instruction') | ('xcerpt' ':' 'pi')
         ( '(' 'target-name' '=' identifier_dt ')' )
         | ('target-name' '=' identifier_dt
1288         ) '[' literal_content_dt ']'
      → ^(PROCESSING_INSTRUCTION_DT identifier_dt literal_content_dt)
1290         ;

1292 structured_dt :
1294         local_spec_dt
      → ^(STRUCTURED_DT_LOCAL_SPEC local_spec_dt)
1296         | children_list_dt
      → ^(STRUCTURED_DT_CHILDREN_LIST children_list_dt)
      ;

1298 children_list_dt :
1300         ( '[' (data_term ',?)* ']' )
      → ^(CHILDREN_LIST_DT_BRACKETS data_term*)
1302         | ( '{' (data_term ',?)* '}' )
      → ^(CHILDREN_LIST_DT_BRACES data_term*)

```

```

1304         ;
1306 local_spec_dt :
1307     term_identifier_dt? ns_label_dt attr_term_list_dt
1308 → ^(LOCAL_SPEC_DT ns_label_dt attr_term_list_dt term_identifier_dt?)
1309     ;
1310 term_identifier_dt :
1311     identifier_dt '@'
1312 → ^(TERM_IDENTIFIER_DT identifier_dt)
1313     ;
1316 ns_label_dt :
1317     (left=identifier_dt ':'? right=identifier_dt
1318 → ^(NS_LABEL_DT $right $left)
1319     ;
1320 attr_term_list_dt :
1321     ((' (attr_term_dt ':'?)* ')
1322     | (attr_term_dt)? )
1323 → ^(ATTR_TERM_LIST_DT attr_term_dt*)
1324     ;
1326 attr_term_dt :
1327     base_attr_term_dt
1328 → ^(ATTR_TERM_DT base_attr_term_dt)
1329     ;
1332 base_attr_term_dt :
1333     ns_label_dt '=' literal_content_dt
1334 → ^(BASE_ATTR_TERM_DT ns_label_dt literal_content_dt)
1335     ;
1336 top_level_data_term :
1337     top_term_level_declare_dt
1338 → ^(TOP_LEVEL_DATA_TERM_TOP_TERM_LEVEL_DECLARE top_term_level_declare_dt)
1339     | structured_dt
1340 → ^(TOP_LEVEL_DATA_TERM_STRUCTURED structured_dt)
1341     ;
1344 top_term_level_declare_dt :
1345     'declare' ns_declaration_dt ( ((' top_level_data_term ')
1346     | (top_level_data_term)
1347     )
1348 → ^(TOP_TERM_LEVEL_DECLARE_DT ns_declaration_dt top_level_data_term)
1349     ;
1350
1352 /* LITERAL STRUCTURES */
1354 // NOTE: These definitions are NOT complete and thus NOT conforming to any specification !!!
1355 NCNAME : (LETTER | '_' ) NAMECHAR* ;
1356 ncname : NCNAME
1357 → ^(LITERAL_NCNAME NCNAME)
1358     ;
1360 iri : ncname
1361 → ^(IRI ncname)
1362     ;
1363 string : "" (options {greedy=false;} : NCNAME) ""
1364 → ^(STRING NCNAME)
1365     ;
1366 stringcharacter : ncname// | line_feed | carriage_return ;
1367 → ^(STRINGCHARACTER ncname)
1368     ;
1369 line_feed : '\n' ;
1370 carriage_return : '\r' ;
1371 number : xcerpt_int
1372 → ^(NUMBER xcerpt_int)
1373     ;

```

```

1374 xcerpt_int : DIGIT*
      ;
1376 regexp : NCNAME;/'/' NCNAME '/';
      //TODO
1378 ere_expression : ;
      whitespace : ;
1380 comment_char : ;
      end_of_line : ;
1382 end_of_line_comment : ;
      block_comment : ;
1384
      //keyword : 'declare' | 'xcerpt' | 'optional' | 'opt' /* ... */;
1386 WHITESPACE : ( '\t' | ' ' | '\r' | '\n' | '\u000C' )+ { $channel = 99; } ;
      LETTER : 'a'..'z' | 'A' .. 'Z' ;
1388 DIGIT : '0' .. '9' ;
      NAMECHAR : LETTER | DIGIT | '-' | '_' | '.' ;
1390 CONSTRUCT : '_';
      PROGRAM : '_';
1392 DECLARE : '_';
      GOAL : '_';

```


Bibliography

- [1] S. Abiteboul and P. C. Kanellakis. Object Identity as a Query Language Primitive. *Journal of the ACM*, 45(5):798–842, 1998.
- [2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wienerm. The Lorel Query Language for Semistructured Data. *Intl. Journal on Digital Libraries*, 1(1):68–88, 1997.
- [3] K. Apt, H. Blair, and A. Walker. Towards a Theory of Deductive Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 2, pages 89–148. Morgan Kaufmann, 1988.
- [4] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2):97–137, 1976.
- [5] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S. Zdonik. The Object-oriented Database System Manifesto. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-oriented Database System: The Story of O2*, Morgan Kaufmann Series In Data Management Systems, chapter 1, pages 1–20. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
- [6] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised Report on the Algorithm Language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.
- [7] J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and Semantic Web Query Languages: A Survey. In J. Maluszinsky and N. Eisinger, editors, *Reasoning Web Summer School 2005*, number 3564 in LNCS. Springer-Verlag, 2005.
- [8] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes Second Edition. Recommendation, W3C, 2004.
- [9] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Working draft, W3C, 2005.
- [10] T. Bray, D. Hollander, and A. Layman. Namespaces in XML. Recommendation, W3C, 1999.
- [11] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). Recommendation, W3C, 2004.

- [12] L. Braz. Visual Syntax Diagrams for Programming Language Statements. In *Proc. Intl. Conf. on Systems Documentation*, pages 23–27. ACM Press, 1990.
- [13] F. Bry, T. Furche, L. Badea, C. Koch, S. Schaffert, and S. Berger. Querying the Web Reconsidered: Design Principles for Versatile Web Query Languages. *Journal of Semantic Web and Information Systems*, 1(2), 2005.
- [14] F. Bry, T. Furche, S. Schaffert, and A. Schröder. Simulation Unification. Deliverable I4-D5, REWERSE, 2005.
- [15] J. Clark. Associating Style Sheets with XML Documents, Version 1.0. Recommendation, W3C, 1999.
- [16] J. Clark. XSL Transformations, Version 1.0. Recommendation, W3C, 1999.
- [17] J. Clark. RELAX NG Compact Syntax. Committee specification, OASIS, 2002.
- [18] J. Clark and M. Murata. RELAX NG Specification. Committee specification, OASIS, 2001.
- [19] E. F. Codd. Extending the Database Relational Model to Capture more Meaning. *ACM Transactions on Database Systems*, 4(4):397–434, 1979.
- [20] B. Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.
- [21] J. Cowan and R. Tobin. XML Information Set (2nd Ed.). Recommendation, W3C, 2004.
- [22] D. Crocker and P. Overell. Augmented BNF for Syntax Specifications: ABNF. Request for Comment (RFC) 2234, IETF, 1997.
- [23] U. Dayal, N. Goodman, and R. H. Katz. An Extended Relational Algebra with Control over Duplicate Elimination. In *Proc. ACM Symposium on Principles of Database Systems*, pages 117–123, New York, NY, USA, 1982. ACM Press.
- [24] S. DeRose, E. Maier, and D. Orchard. XML Linking Language (XLink) Version 1.0. Recommendation, W3C, 2001.
- [25] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *Proc. Intl. World Wide Web Conf.*, 1999.
- [26] A. Dovier, C. Piazza, and A. Policriti. An efficient Algorithm for Computing Bisimulation Equivalence. *Theoretical Computer Science*, 311(1-3):221–256, 2004.
- [27] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs). RFC (Request for Comments) 3987, IEEE, 2005.
- [28] D. C. Fallside and P. Walmsley. XML Schema Part 0: Primer Second Edition. Recommendation, W3C, 2004.
- [29] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model. Working draft, W3C, 2005.
- [30] T. Furche, F. Bry, and S. Schaffert. Initial Draft of a Language Syntax. Deliverable I4-D6, REWERSE, 2006.

- [31] R. Gentilini, C. Piazza, and A. Policriti. From Bisimulation to Simulation: Coarsest Partition Problems. *Journal of Automated Reasoning*, 31(1):73–103, 2003.
- [32] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Third Edition*. Addison-Wesley Professional, 3rd edition, 2005.
- [33] T. A. Group. Ieee standard 1003.1, 2004 edition (aka posix.1). IEEE Standard 1003.1, IEEE, The Open Group, 2001-2004.
- [34] S. Grumbach and T. Milo. Towards Tractable Algebras for Bags. In *Proc. ACM Symposium on Principles of Database Systems*, pages 49–58. ACM Press, 1993.
- [35] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing Simulations on Finite and Infinite Graphs. In *Proc. Symp. on Foundations of Computer Science (FOCS)*, page 453, Washington, DC, USA, 1995. IEEE Computer Society.
- [36] ISO/IEC. *ISO/IEC 14977:1996, Syntactic Metalanguage – Extended BNF*, 1996.
- [37] M. Kay, N. Walsh, H. Zongaro, S. Boag, and J. Tong. XSLT 2.0 and XQuery 1.0 Serialization. Working draft, W3C, 2005.
- [38] S. N. Khoshafian and G. P. Copeland. Object Identity. In *Proc. Intl. Conf. on Object-oriented Programming Systems, Languages and Applications*, pages 406–416, New York, NY, USA, 1986. ACM Press.
- [39] A. Klausner and N. Goodman. Multirelations — Semantics and Languages. In *Proc. Intl. Conf. on Very Large Data Bases*, volume 11, pages 251–258. Morgan Kaufmann, 1985.
- [40] G. M. Kuper and M. Y. Vardi. The Logical Data Model. *ACM Transactions on Database Systems*, 18(3):379–413, 1993.
- [41] L. Libkin and L. Wong. Query Languages for Bags and Aggregate Functions. *Journal of Computer and System Sciences*, 55(2):241–272, 1997.
- [42] A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. Working draft, W3C, 2005.
- [43] J. Marsh. XML Base. Recommendation, W3C, 2001.
- [44] J. Marsh, D. Veillard, and N. Walsh. xml:id Version 1.0. Recommendation, W3C, 2005.
- [45] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: a database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.
- [46] R. Milner. An Algebraic Definition of Simulation Between Programs. In *Proc. Intl. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.
- [47] Object Management Group. UML 2.0 Superstructure Specification. Specification, Object Management Group, 2005.
- [48] T. Przymusiński. On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 5, pages 193–216. Morgan Kaufmann, 1988.

- [49] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, University of Munich, 2004.
- [50] S. Schaffert, F. Bry, and T. Furche. Initial Draft of a Possible Declarative Semantics for the Language. Deliverable I4-D4, REVERSE, 2005.
- [51] D. W. Shipman. The Functional Data Model and the Data Languages DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.
- [52] M. Stonebraker, G. Held, E. Wong, and P. Kreps. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, 1976.
- [53] D. H. D. Warren, L. M. Pereira, and F. Pereira. Prolog - the Language and its Implementation compared with Lisp. In *Proc. Symposium on Artificial Intelligence and Programming Languages*, pages 109–115, 1977.
- [54] M. M. Zloof. Query By Example. In *AFIPS National Computer Conference*, 1975.