Antti Jokipii

GRAMMAR-BASED DATA EXTRACTION LANGUAGE (GDEL)

Master of Science Thesis

in Information Technology (Software Engineering)

10th October 2003

University of Jyväskylä

Department of Mathematical Information Technology

Author: Antti Jokipii

Contact: anoljo@jyu.fi or antti.jokipii@republica.fi

Title: Grammar-based Data Extraction Language (GDEL)

Title in Finnish: Kielioppiperustainen tiedon irroittamis kieli (GDEL)

Number of Pages: 101 + 14

ACM Taxonomy: Data mapping (D.2.12.a), Syntax (D.3.1.b), Parsing (D.3.4.h) (F.4.2.d), Data communication aspects (E.0.a), Data description languages (H.2.3.a), Data translation (H.2.5.a), Data sharing (H.3.5.b), Format and notation (I.7.2.b), Languages and systems (I.7.2.e), Markup languages (I.7.2.f), Electronic data interchange (K.4.4.c).

Keywords: XML, Enterprise Application Integration, Enterprise Information Integration, Shared Information System, Wrapper, Conversion Problem, Legacy Data Problem, Semi-Structured Data, Language Engineering, Syntax Definition, Generalized LR parsing, Scanner less Parsing, Data Transformation

Abstract: This thesis examines the wrapper – a data transformation component – approach to the data integration and XML as a general interchange format. This thesis will show that quite well known context-free parsing methods can be used as the general wrapper. The constructive part of the thesis introduces “Grammar-based Data Extraction Language” shortly GDEL, a state of art data conversion language. GDEL describes conversions from any format, defined as a context-free language, into XML.

Abstract in Finnish: Pro gradu-työ tutkii wrapper-tietomuunnoskomponenttipohjaista lähestymistapaa tiedon integrointiin ja XML-formaattia yleisenä siirtoformaattina. Työ osoittaa kuinka kohtuullisen hyvin tunnettuja kontekstivapaita parsintametodeja voidaan käyttää yleisinä tietomuunnoskomponentteina. Työssä on kehitetty kielioppeihin tukeutuva kieli GDEL, jonka avulla kaikki kontekstivapailla kieliopeilla kuvattavat tietoformaatit voidaan muuntaa XML-muotoon.

Preface

This thesis is one of the results of the project ***Semantic Web***, supported by University of Jyväskylä with financial support from Republica Corporation and TEKES (The Technology Agency of Finland). This thesis was years in preparation. Partly the reason was that I got tired of writing it, but also because GDEL implementation and my job was taking too much time.

Acknowledgements

Many people contributed to development of this thesis. Here I would like to thank them: Jouko Salonen for offering me the most interesting work at Republica; Eetu Ojanen for thesis supervising at the Republica; Timo Airaksinen for project management; Kalle Björklid, Marko Yli-Rämi, Peter Mikula, Diego Ballvé, Jussi Lemmetty, Tommi Eskelinen, Marko Jalonen, Eero Lempinen for feedback; Antti Vuorenmaa, Pete Räsänen for advices for Performer using; all the rest of the Republica gang. Jonne Itkonen my thesis supervisors at the University of Jyväskylä; Professor Pekka Neittaanmäki for pressure that getting gears rolling when they got stuck; my parents for their love and support, and my girlfriend Ulla Venäläinen for proof reading. Many people have written great books and other publications, their names can be found in References. I must still point out that all errors are mine.

Antti Jokipii

Jyväskylä
October 2003

Terms and Abbreviations

ASCII	American Standard Code for Information Interchange defines how computers write and read characters.
ASF	Algebraic Specification Formalism. A language for equation specification of abstract data types.
AST	Abstract Sytax Tree. A tree representation of language syntax.
B2B	Business To Business. A business operation in e-commerce through which businesses can offer and market their products and services.
B2Bi	Business-To-Business Integration. B2Bi makes possible the connection between trading partners over the Internet.
Bi-directional	The ability to extract, cleanse, and transfer data in two directions.
BNF	Backus-Naur Form. Originally Backus Normal Form. A formal Meta syntax used to express context-free grammars. Backus Normal Form was renamed Backus-Naur Form at the suggestion of Donald Knuth.
CFG	Context-Free Grammars. See section 4.3.
CSG	Context-Sensitive Grammars. See section 4.3.
Data Integration	Data integration supports the seamless exchange of various data formats between applications and between trading partners over the Internet.

Data Content Quality	The accuracy and validity of the actual values of the data, in contrast to issues of schema.
Data Mining	Name of technology to find important nuggets of information in voluminous texts.
DTD	Document Type Definition. Grammar of XML or SGML data. See: XML and SGML.
EAI	Enterprise Application Integration. EAI is a category of infrastructure software aimed at facilitating the transfer of information between various systems in order to update the information and its consistency. This makes it easier to establish communications between systems initially unable to communicate with each others.
HTML	Hypertext Mark-up Language, The mark-up language used to create hypertext documents for use on the WWW, based on SGML.
Information Extraction	Systems that accurately extract, correlate, and standardize important information from text. For example, converting business articles to a structured business database.
Legacy	Pre-existing applications or data that is not possible to automatically integrated to new applications.
Lexical Analyzer	(Or "scanner") The initial input stage of a language processor (e.g. a compiler), the part that performs lexical analysis.
Mediation Service	Covers value-added processing on resulting content, as converting and filtering.

Mediator	Component to provide mediation services.
Middleware	Software designed to establish a permanent relationship (including filtering and transformation) between source systems and logical models.
Scanner	See Lexical Analyser.
Schema	The organization or structure for a data.
SDF	Syntax Definition Formalism. A language for lexical and syntactic specification for context free grammars developed by the Programming Research Groups of the University of Amsterdam and the Centre of Mathematics and Computer Science at Amsterdam.
SGML	Standard Generalized Mark-up Language defines device-independent layout for textual data.
Tag	In the world of XML/SGML, a tag is a marker embedded in a document.
URS	Universal Rewriting System. See section 4.3.
Wrapper	Component that provides data transformation service.
XML	Extensible Mark-up Language, a Meta language for defining specialized mark-up languages that are used to transmit formatted data. XML is conceptually related to HTML and based on SGML.
XML Schema	W3C Recommendation for Schema and Data Content Quality for XML.

Contents

Table of Figures

1 Introduction

The thesis examines wrapper – a data transformation component – approach to data integration and XML as a general interchange format. The thesis will show that the usage of quite well known context-free parsing methods can be used as general wrapper. The thesis will include an analysis of general grammar based parser techniques and grammar representation formats. It also contains description of grammar systems and used XML technologies.

The constructive part of the thesis introduces "Grammar-based Data Extraction Language" shortly GDEL, a state of art data conversion language for describing conversions from any format defined by a context-free language into XML. This language will be designed and its usage will be evaluated against some common usage scenarios in the thesis. The thesis shows mechanisms that semi-automatically generate GDEL rules from BNF or other grammar representation languages. At last but not least the thesis will contain GDEL processor design and implementation considerations.

The need for enterprise application integration and B2B integration is generating direct need to solve data integration problems. Problems of the data integration are a direct result of using different data representations. Recently markup languages, particularly XML, have been used to address these problems as a general interchange format [Hasselbring00]. The research of XML translation studies mostly XML-to-XML transformations and XML down transformation back to the legacy format. Area of the up translation is studied mostly in specific contexts such as HTML translation to XML [Jokipii99].

In 1998 we in Republica tryed XML/Xlink portal solution [Salonen98]. These issues raises the question about the generic syntactic translation on the data from the native data format of the application to the XML interchange format. In 1999 we in Republica believed that solution could be XML based generic data extraction language (DEL), which is capable to transform any unstructured data to XML [Salonen99]. This development resulted to X-Fetch Wrapper component and DEL language that is contributed W3C as a note [Lempinen01]. The three years of intensive usage of the DEL language from our own and our customer's experience has shown that the approach is the best-known practice. DEL is already in a second version, which changes it to more XSLT-like and therefore simpler to use and learn.

The DEL approach is pattern based and we have found, that extracting data from formats that are designed with grammars, need complex control structures in all pattern-based languages. This experience shows that the approach needs the expansion that is capable to handle the grammars directly. Idea for GDEL is born.

Short look to different parser generators and grammar formalisms in autumn 2001 showed that there is not a single general formalism for grammars. In that time we could not find powerful enough parsing algorithms to parse full range of context-free grammars. In the spring 2002 we found works of Eelco Visser [Visser97a-d]. They show that there are possibilities to parse full range of context-free grammars and also complete formalism to write grammars. Then Republica offers possibility to make thesis from that idea. The thesis is result of that process.

The structure of this thesis is presented in the figure 1. First chapter introduces the thesis, its organization, motivation and some background. Second chapter describes techniques of data integration used in the thesis. Particularly it introduces XML as a data exchange format and an ideology of wrappers. Third chapter introduces family of XML techniques used later in the thesis.

Chapters 4 through 6 are devoted to context-free grammars. Chapter 4 introduces the basic concept of context-free grammars. Disambiguation, abbreviations and restrictions are considered in Chapter 5. Some of the known syntactic meta languages, like BNF, are evaluated in Chapter 6. The common context-free parsing techniques are described in Chapter 7, especially GLR- and Earley-parsing.

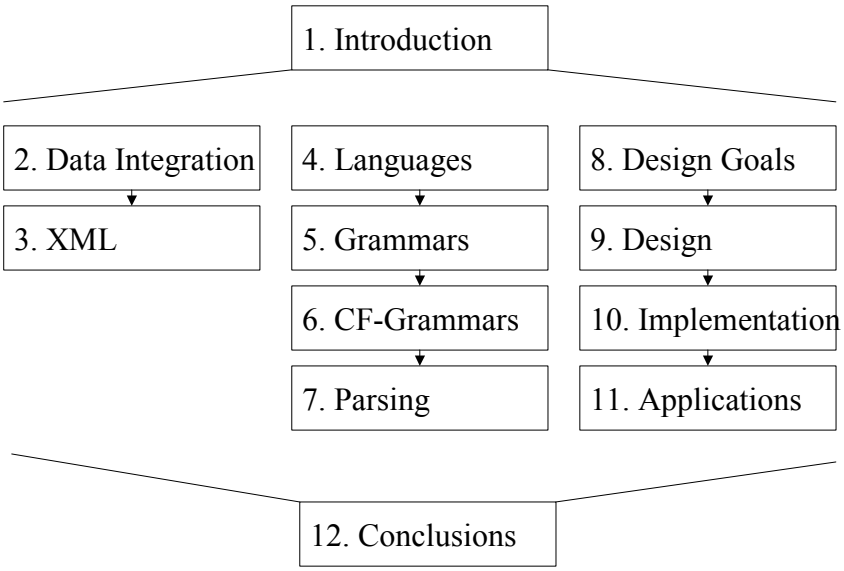

Figure 1. Overview of chapters content.

The next two chapters present the GDEL desing. Chapter 8 sets design goals. Design decisions made are covered in Chapter 9.

Chapter 10 evaluates GDEL processor as a software component. It also contains the processor implementation considerations. Chapter 11 introduces some applications, especially possible only with GDEL. The last chapter presents conclusions and avenues for future work.

2 Data Integration

Most organizations have many autonomous information systems or pre-existing applications (legacy systems). Typically, these applications are developed and deployed independently from needs of individual business units. Enterprise application integration (EAI) tries to solve a problem how to interoperate between applications within individual businesses [Pinkston01], [Erlikh01]. The advent of the Internet and new co-operative business models has exacerbated another problem – how to automate business-to-business (B2B) interactions efficiently, by using business-to-business integration (B2Bi) [Lublinsky02], [Pinkston01]. Most organizations would get benefits from integration; it could be even a critical success factory for the order to stay competitive.

Even the middleware products significantly simplify the integration task, they don't offer a seamless solution. From the technical point of view, integrating heterogeneous hardware platforms, operating systems, database management systems, and programming languages can be challenging. Without conventions that protect integrator from the details of particular environments, it would be almost impossible to integrate such a variety of information systems [McLean98]. One way to make these conventions is standard bodies like Extensible Markup Language (XML), which helps to create data exchange formats and could be used to solve one of the key aspect of all integration – the data exchange. State of the art integration technology also uses loosely coupled integration components, which are expected to lower costs and increase the system stability through reuse of components [Hasselbring00], [Wijegunaratne00]. For this end we need transformation components called wrappers, which are able to convert system-specific data representations to the chosen XML format.

This chapter studies in more detail some data integration techniques. First section studies XML as a data exchange format. Different kind of wrappers and their implementations are studied in the next sections. The last section makes an overview of GDEL approach.

2.1 XML as Data Exchange Format

The *Extensible Markup Language* (XML) is a standard meta language for describing data. XML promises to be a key enabling technology for seamless data exchange between applications running on multiple platforms. The World Wide Web Consortium (W3C) developed it in 1998 [XML], [XMLse].

The basic idea underlying XML is very simple – the separation of the content and structure of data [Bosak97]. This makes it possible to freely describe structured and unstructured data, and allows defining their relationships without any application or vendor constraints. XML standard allows to define structure of document – grammar – as Document Type Definition (DTD) that describes relationships between elements via nesting and references [Deutsch00]. XML tags have no fixed semantic meaning [Bosak98], which allows so called late-binding where the meaning of tag is defined in usage context just before execution. One consequence of missing semantics is that XML completely separates the data from presentation [Usdin98] and therefore one XML document could have multiple presentations. For data exchange XML has several benefits and features, which allow it to act as a data exchange format:

- XML is an open standard which makes it cheap and available to all sizes of companies, anywhere in the world. Standardization prevents single vendor to control markets and the locking of customers in proprietary technology [Widergren99], [Bosak98], [Bos99], [Bapst99], [Lear99].

- XML has a strong vendor support [Kotok00], [Worder00].

- XML syntax is very simple - hence its documents are easy to create and use [Deutsch00].

- Users could define and extend XML data structures by modeling it at any level of complexity [Bosak98], [Usdin98].

- XML allows internationalization and media independence [Bosak98].

- XML data is automatically reusable which preserves and promotes strategic investment in company's data. It can provide greater levels of service and tighter integration by making more detailed data available when needed, for example portal syndication where data and components can be re-used between several companies [Bapst99], [Lear99].

- It is self-describing and therefore more understandable and processable [Widergren99]. Furthermore meta-data could be used to make applications and data more accessible to software agents [Lie99], [Glushko99].

Some examples of the XML usage as interchange format can be found in references [Ebert99] [Herman00], [Holt00], and [Kienle01]. Further elaboration on how these benefits can facilitate electronic commerce, collaboration [Koch99], [Sanborn00], portal services [Lowry00], [Lowry01] and information brokers [Lu00] can be found in these references.

XML will play a fundamental role in the integration of heterogeneous data [Gao99], [Lee00], [Petrou99]. As the integration layer data model XML can be used as the least common denominator for representing information [Emmerich99], [Gross01]. However, XML itself does not solve the whole problem – we need to translate data between disparate end point entities that are wanted to interoperate. XML documents are easy to transform to proprietary format by using technologies such as XSL [XSL]. Wrapper components could be used to automate the translation of information from proprietary format to XML. This is a way to use XML, with appropriate enabling software components, to integrate otherwise incompatible systems.

All these characteristics make XML excellent for exchanging information between organizations or within an organization. Closer view of different standards in the XML standard family is taken in next chapter.

2.2 Wrappers

Generally wrappers are software components or middleware software that change data to desired representation from its original source format. If a wrapper is considered in a broad sense it is any part of the code that changes data representation. From this point of view most wrappers are hand coded and parts of more sophisticated applications. In the thesis are considered only wrappers that are software components and therefore well separated from applications.

Even in the software component approach wrappers could be hand-coded. The oldest approach to help wrapper generation is to make languages with pattern matching support. One example of language designed for wrapper generation is Perl [Cross01]. Data Extraction Language (DEL) uses similar approach, but have been specially developed for any-to-XML transformations [Lempinen01]. It's current second version is state of the art language for any-to-XML conversions.

There are lots of examples of pattern approach: [Adelberg98], [Ashish97a-b], [Ek01], [Hammer97], [Hsu98]. Some of these also contain more sophisticated user interface that make generation of wrappers semi-automatic [Ojanen01], [Muslea99], [Seymor99], [Thomas99]. For example W4F (Wysiwyg World Wide Web Factory) [Sahuget98] is a toolkit for generating wrappers for Web sources. It assists users to create wrappers fast and easily and provides WYSIWYG (What-You-See-Is-What-You-Get) function via some wizards. Another example is XWRAP (eXtensible Wrapper Generation System) [Liu00]. It is a wrapper generation system, which enables wrappers for Web information sources to be constructed semi-automatically. This system can transform non-XML documents into XML documents and extract information that user wants.

There is also another general way to do a wrapper: the grammar-based method [Fankhauser93], [Klein97], [Nakhimovsky01]. Pattern matching only describes interesting parts and is very flexible, whereas grammars give an exact description of the document, but adapting irregularities is more difficult. Actually, there is no sharp borderline between the two approaches. The next two subsections introduce in more detail some of the grammar-based wrappers.

2.2.1 JEDI

Peter Frankhauser and Gerald Huck from German National Research Center wrote JEDI for Information Technology (GMD - Forschungszentrum Informationstechnik GmbH).

JEDI is an extensible, fault tolerant parser component for the extraction of semi-structured data from textual sources, especially from the World Wide Web. It uses simple, grammar-based syntactical source descriptions and an associated lightweight data model to provide for sophisticated and flexible rewriting and restructuring facilities which grant seamless, integrated access to various heterogeneous sources [Huck98].

The parser can cope with incomplete and ambiguous source specifications by a novel parsing technique that chooses always the most specific rule among several applicable rules. When it cannot find an applicable rule, it skips as little as possible of document – i.e. fallback rules – to continue with an applicable rule. JEDI uses a lightweight generic object model as a mediator. Furthermore, the generic output routines are supported to display views in XML or any other structured format. JEDI is available at: http://www.darmstadt.gmd.de/oasys/projects/jedi/index.html.en.

2.2.2 XTAL

XTAL is general Java Package written by Oliver Zeigerman. XTALs architecture consists of the front end for reading data in and the back end to produce outputs. Today there exist only one XM front end and two back ends to produce XML or TeX. XTAL is based on ANTLR description language and Java. XTAL is available at: http://www.zeigerman.de/xtal.html

2.3 GDEL Approach

The black box view of GDEL processor shows overall process – see next figure. We can separate there four main parts: specification, input material, processor and output XML.

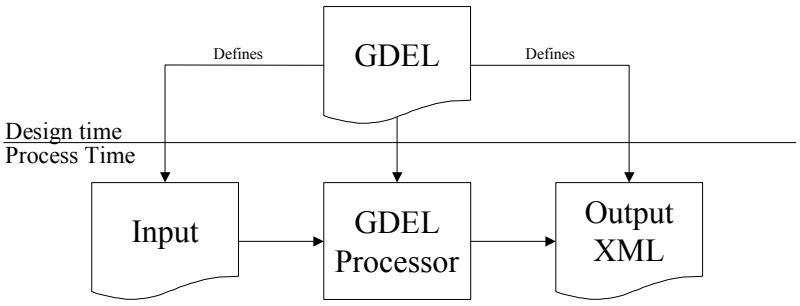

Figure 2. The black box view of GDEL processor.

In design time we need to create the specification that defines the grammar for input, and tag names for output. In run time we give input to processor, which then parses input described by grammar of specification. Parsing creates a parse tree, or forest if grammar is ambiguous, and returns it in XML format.

The difference between GDEL approach and previously presented grammar-based wrappers is mainly that GDEL has wider usage area. The GDEL must allow usage of any context-free grammars to describe input while JEDI is limited to only some parts of context-free grammars, because it way to handle ambigious grammars, and XTAL only for LL(k) grammars and coded input and output materials.

3 Family of XML Technologies

The XML is not just a meta language for defining markup languages. In fact, XML derives its strength from a variety of supporting technologies, namely presentation, structure, and transformation. In this chapter we examine the XML core and its surrounding technologies.

The XML core includes the XML itself, based on the XML 1.0 specification [XML], namespaces [Namespaces], and XML Information Set (InfoSet) [InfoSet]. Namespaces are used to solve the problem of clashing names when documents from different sources to be combined. InfoSet provides a consistent set of definitions used in other specifications that need to refer to the information in XML document. For data typing we look at XMLSchema [XMLSchema]. For transformation, we examine XSL Transformations (XSLT) and XPath [XSLT], [XPath].

3.1 XML Standard

The Extensible Markup Language (XML) became World Wide Web Consortium (W3C) recommended standard in February 1998 [XML], [XMLse]. XML has a long development history. It was developed to overcome the limitations of the Hypertext Markup Language (HTML), which was created 1989 in CERN European Nuclear Research Facility [HTML]. HTML is application based on Standard Generalized Markup Language (SGML), which became an international standard ISO 8879:1986 [ISO8879]. SGML roots can be found back to1969, when an IBM team leaded by Charles Goldfarb developed a document description language (the Generalized Markup Language, GML) to solve the problem of different document formats of various systems [Goldfarb90]. XML is simplification of SGML that retains most of the features of the standard, but makes it easier to implement and use in the World Wide Web (WWW) environment.

Next subsection introduces the structure and syntax of XML documents. The purpose is not to give a full description about every detail of the language, but rather to explain the principal components that form an XML document.

3.1.1 Structure

Each XML document has a physical structure, composed of units called entities. An entity may refer to other entities to cause their inclusion in the document. Each XML document has at least one entity called the document entity. The logical structure of document contains declarations, elements with attributes, comments, character references, and processing instructions, all of which are indicated in the document by explicit markup.

A document entity should begin with an XML declaration which specifies the version of XML being used and could contain explicit information of used encoding and information, if document contain external entities. Other external entities should begin with text declaration, which is similar to XML declaration but could not contain information of external entities. Example:

```
<?xml version=”1.0” encoding=”UTF-8” standalone=”yes”?>
```

3.1.2 Elements

Elements are the core of the markup. Each XML document contains one so-called root element. Elements are used to identify the content in the document. Syntactically tags with element name, which usually indicates their intended meanings, that surround the element content, form elements. For example:

```
<element>content</element>
```

Elements can have an empty content and use the abbreviation:

```
<element/>
```

Elements can contain attributes, whose order is irrelevant. Attributes are listed as name – value pairs that occur within the start-tag after the element name. The same name cannot appear twice in an attribute list. Attributes refine the nature of the element by defining its characteristics. Example:

```
<price currency=”euro”>123</price>
```

3.1.3 Character Data

Elements can contain character data, which is either parsed character data or CDATA sections. In parsed character data entity references are replaced by content of entity. All other characters are handled as plain character data.

CDATA sections could be used anywhere character data may occur. CDATA sections begin with the string '<![CDATA[', end with ']]>' and can contain any characters except string ']]>'. All characters inside CDATA sections are recognized as plain character data and therefore sections could be used to escape blocks of text that would otherwise be recognized as markup.

Entity References

Entities are predefined content that can be added to documents with entity references. In XML, some characters have been reserved to identify the start of the markup. For example, the left angle bracket '<' normally identifies the beginning of an element start-tag or end-tag. Entity references are a way to insert these characters into the document content. They can also be used to represent often repeated or varying text and to include the content of external files. Entity references are marked with sequence of an ampersand '&', entity name, and a semicolon ';'. Table 1 shows the five predefined entities in the XML specification [XML].

Entity Reference	Content	Character
<	<	Opening angle bracket
>	>	Closing angle bracket
&	&	Ampersand
'	'	Apostrophe
"	"	Double quotation mark

Table 1. Predefined character entities in XML.

3.1.4 Comments

Comments are the way to add human readable explanations in XML documents. They begin with '`<!--`' and end with '`-->`' Comments are not a part of the textual content of an XML document. The XML specification does not require XML processors to pass them to applications.

3.1.5 Processing Instructions

Processing Instructions (PIs) can be used to provide information to applications. They are not a part of the document character data but XML processors are required to pass them to applications. PIs have the form '`<?pitarget pidata?>`' where pitarget is used to identify the application to which the instruction is directed. The pidata part is optional, it is meant for the application that recognizes the target. The target names '`XML`', '`xml`', and all variations of these are reserved for XML standardization [XML].

3.1.6 Document Type Definition (DTD)

After XML declaration, documents may contain a document type declaration, which specifies the document type in terms of the grammar [XML]. This grammar is known as a document type definition (DTD). DTD model restrictions are defined in reference [Brueggemann-Klein98]. The document type declaration can point to an external subset, containing declarations, or it can contain the declarations directly in an internal subset, or it can do both. If both subsets have same declarations internal subset overwrites declarations of external subset.

DTD provides a way of defining the logical structure of the XML document by describing all the elements that can be used, their attributes, and how they can be related to each other. Therefore we are able to enforce certain restrictions on how the XML document can be composed and this makes it easy to create applications that process these XML documents – especially when XML data binding is used to automatic generation of application template classes. A DTD has its own non-XML syntax containing any number of declarations of the following types:

The element type declaration restricts the kind of children an element can have, and in which order they appear. It can say that an element is empty and does not have child elements, content is without any restrictions, specify content model, or content may consist of character data and child elements whose names are listed. A content model specifies sequence matched by regular expressions. Regular expressions consist of child element names, list of alternatives, and list of sequences. Each of these can have an optional occurrence operator. The occurrence operator could be one or more '+', zero or more '*', or zero or one '?'. The absence of operator means that content must appear exactly once. In this context, white space is called ignorable white space, and may be freely interspersed with the elements.

The attribute list declaration contains attributes associated with a particular element. Each attribute has a name, a type and possible default value. The type determines the range of values that the attribute may hold. The allowed types are: 'CDATA', 'NMTOKEN', 'NMTOKENS', 'ENTITY', 'ENTITIES', 'ID', 'IDREF', 'NOTATION' or name group. The default value allows specifying if the attribute is required, implied, default or fixed.

The entity declaration defines entities that are used to avoid repetition in XML documents. They are declared once and can be referred many times. Both internal and external entities are allowed in DTDs. Internal entities are defined within the current DTD, while external entities reside in separate locations.

The notation declaration is used to refer data that is not in XML format. Notations can also be linked with entities by using the 'NDATA' keyword.

It is not important to present the syntax in more detail here, but see appendix 1 for an example of a DTD.

3.1.7 Well-formedness

According to XML specification, a textual object is a well-formed XML document if it obeys the syntax rules of XML. Walsh [Walsh98] lists the rules in his technical introduction to XML. By definition, if a document is not well-formed, it is not XML. This implies that there is no such thing as an XML document that is not well-formed. XML processors are not allowed to parse such documents [Walsh98]. In well-formed documents, it is allowed to add any element with any attribute in any hierarchy, on the condition that the rules for well formedness are obeyed.

3.1.8 Validity

The use of a DTD allows us to check for validity of the XML document. Everything that is in the XML document must conform the rules and the model defined in an associated DTD specification. The validation is a process for ensuring that documents conform to structures defined in the DTD. In valid XML, the model of the document is explicit in the set of declarations (DTD), while in well-formed XML it is implicit in the hierarchy of the data [Holman99].

3.2 Uniform Resource Identifier (URI)

To access a unique item over the Internet, it is needed to know how to identify that one object among everything else out there. URIs provide a way of uniquely identifying each of those items. Described in detail by Request for Comments 1630 [RFC1630], this specification spells out the rules used in many different protocols within the URI framework. A URI has the form

```
<scheme>:<scheme-specific-part>
```

When the scheme-specific-part contains slashes '/', those slashes indicate some hierarchical structure within the path. The best-known type of URI is the Uniform Resource Locator (URL).

3.3 XML Namespaces

XML namespaces define a set of unique names within a given context [Namespaces]. Namespaces uses unique URI for each prefix of the element. For example, it allows us to create address element in the two different namespaces ‘office.org/office/namespace’ and ‘computer.org/memory/namespace’:

```
<office:address
xmlns:office=”office.org/office/namespace”/>
<memory:address
xmlns:memory=”computer.org/memory/namespace”/>
```

Putting these similar structures into unique namespaces helps prevent the concepts from clashing each others and allows the computer to unequivocally determine which structure is being referenced.

3.4 XML Schemas

An XML Schema provides a superset of the capabilities found in DTD [XMLSchema]. They both provide a method for specifying the structure of an XML document. Whereas both allow for element definitions, only schemas allow specifying the type information. The XML Schema itself is an XML document, and therefore easy to process.

The XML Schema forms a data type hierarchy where all data types derive, directly or indirectly, from the root ‘anyType’. The ‘anyType’ can be used to indicate any value. Below ‘anyType’, the hierarchy branches into two groups consisting of simple types and complex types. Derivation uses a facet to define an aspect of a value space.

3.4.1 Facets

To aid with the definition and validation of data, an XML Schema uses facets to define characteristics of a specific data type. A value space is the set of all valid values for a given data type. The XML schema document specifies two types of facets: fundamental and non-fundamental facets. A fundamental facet is an abstract property that characterizes the values of a value space. These include the following facets:

Equal: Defines the notion of two values of the same data type being equal. The following rules apply to this concept:

1. For any two values (a, b), a is equal to b (denoted a = b) or a is not equal to b ($a \neq b$).
2. No pair of values (a, b) exists such that a = b and $a \neq b$.
3. For every valid value a, a = a.
4. For any two values (a, b) in the value space, a = b if and only if b = a.
5. For any three valid values (a, b, c), if a = b and b = c then a = c.

Order: This specifies a mathematical relation to set the total order of members in the value space. For every pair of values (a, b), their relationship is either $a < b$, $b < a$, or a = b. For every triple (a, b, c), if $a < b$ and $b < c$ then $a < c$.

Bounds: This simply states that a given value space may be bounded above or bounded below. If a value U exists for all values v in the value space the statement $v \leq U$ is true, U represents the upper bound of the value space (bounded above). If a value L exists for all values v in the value space the statement $v \geq L$ is true, L represents the lower bound of the value space (bounded below). If the data type has both an upper and lower bound, then that data type is bounded.

Cardinality: Some value spaces have a finite set of values. Others have an unlimited set of values. A data type has the cardinality of the value space, which is either finite or countable infinite.

Numeric: If the values of the data type are quantities in any mathematical number system, then the data type is numeric. Everything else is nonnumeric.

Non-fundamental or constraining facets are optional properties that can be applied to a data type to constrain its value space. The following facets are possible: length, minLength, maxLength, pattern (a regular expression), enumeration, maxInclusive, maxExclusive, minInclusive, minExclusive, precision, scale, encoding (hex, base64), duration and period. Using all of these facets you can constrain existing data types. This helps to perform tasks such as data validation and verifying the overall correctness of an XML document. Combined with facets, the XML Schema data types can help to give meaning to the items contained by schema.

3.5 XPath

XPath gets its name from its use of a path notation to the addressing parts of XML documents [XPath]. XPath models XML documents as abstraction of a hierarchical tree of nodes instead of its surface syntax. XPath's primary function is the selection of a node or a node set from the document. For that purpose it allows distinguishing between different types of nodes, including element nodes, attribute nodes, and text nodes. It also contains predicates that could be used to filtering wanted set of nodes. XPath uses a compact, non-XML syntax. It is an important XML technology due to its role in providing a common syntax and semantics for functionality in XSLT and other specifications.

3.6 XSLT

XSLT (XSL Transformations) is an XML-based language used to transform XML documents [XSLT]. Although XSLT is designed primarily for XML-to-XML transformations, it is possible to output any other textual formats as well.

To perform an XSL transformation, a program referred as an XSLT processor reads both an XML document and an XSLT document that defines how to transform the XML. The XSLT processor has the capability to read the XML source document, and rearrange and reassemble it with advanced pattern matching mechanisms, inserting and copying data and document fragments, using conditionals and accessing data via XPath.

3.7 DOM

The DOM standard allows a program to interact with the XML document by presenting XML tree nodes as objects [DOM]. The DOM API defines these objects. The advantage of the DOM standard is not in the resolution of possible ambiguity; it is simply allowing programmable interaction with abstract syntax. A similar approach provides support for query and transformation languages for XML.

3.8 XML as Tree

The popularity of XML based formats for representing structured information appears from a certain point of view as recognition of the importance of abstract syntax trees. This is exemplified in particular by approaches that provide XML data bindings for programming languages [Reinold99]. Unfortunately the “concrete syntax of the abstract syntax” provided by XML and its document type definitions are hardly readable for readers outside from the area of classical document processing.

4 Language Specifications

A language provides a way to share information in means of communication. Languages are under heavy study from many aspects e.g. philosophical, linguistic, etc. A study of language syntax and semantics is one of most important parts of modern computer science. In this chapter we mainly focus a general concepts and aspects of data formats.

4.1 Syntax, Semantics and Pragmatics

Natural languages and computer languages are alike in several respects. The terminology of linguistics could be also used to define three main aspects of languages:

1. **Syntax** defines forms of well-formed sentences in the language. Each of the sentences is composed of words or tokens or sets of strings of symbols. Thereby syntax provides structural description of the various expressions of the language, without any consideration of their meaning.

2. **Semantics** define the *meaning* of the syntactically correct expression in a language. For a programming language, semantics describe the relationship between the syntax and the model of computation.

3. **Pragmatics** define those aspects of language that involve the users of the language, namely psychological and sociological phenomena such as utility, scope of application, and effects on the users. For programming languages, pragmatics includes issues such as ease of implementation, efficiency in application, and programming methodology. Similarly as syntax must be defined before semantics, semantics needs to be formulated before considering the issues of pragmatics, since interaction with human users can be considered only for expressions whose meanings are understood.

In the rest of the thesis main concern is the syntactic aspects of language. The concepts and terminology for describing the syntax of languages derive from Noam Chomsky's work for the description of linguistic structure [Chomsky56], [Chomsky59]. His classification of grammars and the related theory was the basis of much further work on formal language theory and theory of computation. Many books contain results on the expressiveness and limitations of the classes of grammars and on derivations, derivation trees, and syntactic ambiguity, for example see [Hopcroft79], [Martin91], [McCarthy65], and [Meyer90].

In fact, much of the early work in mathematical linguistics concerned with efficient methods of parsing that eventually found a better home in compiler design, see [Aho86], [Appel98], [Parson92] and [Slonger95]. These books contain extensive discussions and examples of syntax specification, derivation trees, and lexical and syntactic analysis. Compiler writers typically disagree with distinction between syntax and semantics, putting context constraints with semantics under the name static semantics [Meek90].

4.2 Symbols, Alphabets and Strings

In trying to specify data format translator, one must be aware of some features of formal language theory. We need few definitions.

Definition: Symbol is an undividable entity.

Definition: An alphabet A is a finite nonempty set of symbols.

For example, an alphabet can consist of the 128 symbols of ASCII alphabet. The new standard is an alphabet called Unicode, which contains over 38,000 glyphs that could form symbols including all symbols from nearly all of the world's languages [Unicode]. All the important aspects of formal languages can be modeled using the simple two-letter alphabet $A = \{0,1\}$.

Definition: A string x over alphabet A is a finite, possible empty, sequence of symbols from A.

Definition: The number of symbols i.e. the length of string x is denoted by |x|.

Definition: A zero length string is denoted by symbol ε.

Definition: The set of all strings over alphabet A is called a universal set and marked as A*.

Example: the universal set of alphabet A = {a, b} is A* = {ε, a, b, aa, ab, ba, …}.

Definition: A sentence is valid string of language.

4.3 Languages and Grammars: Chomsky's hierarchy

At the simplest level, languages are sets of sentences, each consisting of a finite sequence of symbols from some finite alphabet. Any language of real interest has an infinite number of sentences. This does not mean that it has an infinitely long sentence but that there is no maximum length for all the finite length sentences.

A grammar defines a language and it is possible to define same language with a number of different grammars. Chomsky's work formulated three theoretical models for grammars, one based on Finite-State Automata (FSA), one based on Context-Free Grammars (CFGs), and one on context-sensitive grammars (CSGs) and/or the even more powerful Unrestricted Rewriting Systems (URSs) [Chomsky56], [Chomsky59]. A formal grammar consists of a finite set of nonterminals (also known as "production symbols" or "grammatical types"), a finite set of terminal symbols (the letters of the sentences in the formal language), a start symbol and an unordered set of production rules with a left hand side and a right hand side consisting of a sequences of these nonterminals and terminal symbols.

Definition: A grammar G is a quadruple $G = (A, N, s, P)$, where A is a finite nonempty set of the terminal symbols, N is a finite nonempty set of nonterminal symbols. Further $N \cap A = \emptyset$. s is a distinguished nonterminal called the start symbol and $s \in N$. P is set of production rules of the form $L \rightarrow R$, where $L \in (N \cup A)^*$ is called left hand side, $R \in (N \cup A)^*$ is called right hand side of production.

Grammar defines the formal language of all words consisting solely of terminal symbols that can be reached by a sequence of rewriting steps (also known as derivation) from the start symbol. In each rewriting step a rule may be applied to a production symbol by replacing the left hand side by the right hand side.

Definition: Rewriting step: $u \rightarrow v$ is defined if and only if $u = xyz$, $v = xy'z$ and $y \rightarrow y' \in P$ for some $x, y, y', z \in (N \cup A)^*$.

Definition: Derivation: '$\Rightarrow$' is the transitive, reflexive closure of rewriting steps '$\rightarrow$', i.e. $u \Rightarrow v$ iff $\exists(w_0, w_1, \ldots, w_j)$, where $u = w_0$, and $w_j = v$, and $j \geq 0$; $w_0 \rightarrow w_1$, $w_1 \rightarrow w_2$, …, $w_{j-1} \rightarrow w_j$.

Definition: The language defined by grammar G is $L(G) = \{w \in A^* \mid s \Rightarrow w)$, where w is the set of all terminal strings derivable from start symbol s.

Various restrictions on the productions define different types of grammars and corresponding languages in the Chomsky hierarchy:

- Type-0 grammars (unrestricted grammars) include all formal grammars and do not have any restrictions. They generate exactly all languages that can be recognized by a Turing machine. The language that is recognized by a Turing machine is defined as all the strings on which it halts. These languages are also known as the recursively enumerable languages.

- Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages: $|L| \leq |R|$, exception: $S \rightarrow \varepsilon$ is allowed if S never occurs on any right hand side. In normal form these grammars rules have the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ with A a nonterminal and α, β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. It can also include the rule $S \rightarrow \varepsilon$. If it does, then it must not have an S on the right side of any rule. These languages are exactly all languages that can be recognized by linear-bounded automata – Turing machine whose tape is bounded by a constant times the length of the input.

- Type-2 grammars (context-free grammars) generate the context-free languages. $L \in N$. These are defined by rules of the form $A \rightarrow \gamma$ with A a nonterminal and γ a string of terminals and nonterminals. These languages are exactly all languages that can be recognized by a pushdown automaton. Context free languages are the theoretical basis for the syntax of most programming languages.

- Type-3 grammars (regular grammars) generate the regular languages. $L \in N$, $R = a$ or $R = aX$, where $a \in A$ and $X \in N$. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed by a single nonterminal. The rule $S \rightarrow \varepsilon$ is also allowed if S does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally, this family of formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages.

These grammar types are arranged according to the complexity required to determine does a given string belong to a given language. Every regular language is context-free, every context-free language is context-sensitive and every context-sensitive language is recursively enumerable. These are all proper inclusions, meaning that there exist recursively enumerable languages, which are not context-sensitive, context-sensitive languages, which are not context-free and context-free languages, which are not regular.

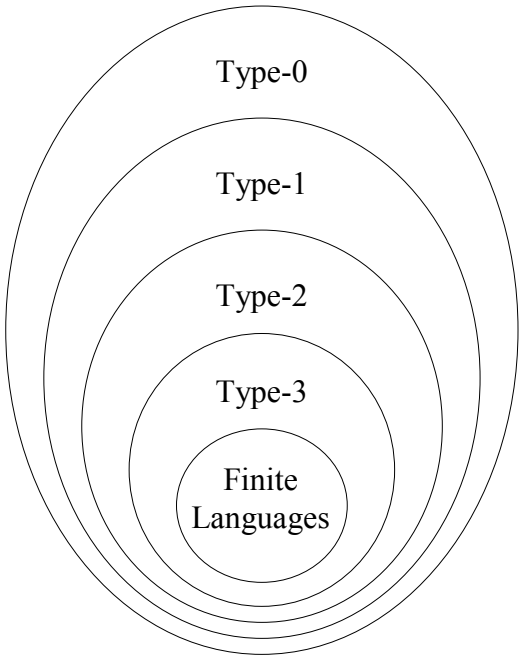

Figure 3. Chomsky hierarchy.

4.3.1 Closure Properties

Suppose we have grammars for the languages $L_1(G_1)$ and $L_2(G_2)$, for grammars $G_1 = (A, N_1, s_1, P_1)$ and $G_2 = (A, N_2, s_2, P_2)$ where the sets N_1 and N_2 are disjoint. Then we could give following definitions for the set-theoretic operations:

Definition: Union $L_1 \cup L_2$ is generated by the grammar (A, N, s, P) where s is a fresh nonterminal, $N = N_1 \cup N_2 \cup s$ and $P = P_1 \cup P_2 \cup \{s \to s_1\} \cup \{s \to s_2\}$.

Definition: Concatenation L_1L_2 is generated by the grammar (A, N, s, P) where s is a fresh nonterminal, $N = N_1 \cup N_2 \cup s$ and $P = P_1 \cup P_2 \cup \{s \rightarrow s_1\ s_2\}$.

Definition: Kleene closure L_1^* is generated by the grammar (A, N, s, P) where s is a fresh nonterminal, $N = N_1 \cup s$ and $P = P_1 \cup \{s \rightarrow s_1\ s\} \cup \{s \rightarrow \varepsilon\}$.

Note that the above definitions do not contain the complement language, the difference between two languages and the intersection of two languages, because there are no equivalent operators for composing grammars that correspond to such intersection and difference operators even they are possible operators for languages.

For a given type of language, we are often interested in the so-called closure properties of the type.

Definition: A type of language is said to be closed under a particular operation – union, intersection, complementation, concatenation, and Kleene closure – if every application of the operation on language of the class yields a language of the same type.

Closure properties are often useful in constructing new languages from existing languages, and for proving many theoretical properties of languages and grammars. The closure properties of the four types of languages in the Chomsky hierarchy are summarized below. Proofs can be found in [Hopcroft79].

1. The class of unrestricted languages is closed under union, intersection, concatenation, and Kleene closure, but not under complementation.

2. The classes of context-sensitive and regular languages are closed under all of the five operations.

3. The class of context-free languages is closed under union, concatenation and Kleene closure, but not under intersection or complementation.

4.4 Regular Expressions

The language of regular expressions is a formal language, which is composed from symbols from alphabet A in conjunction with a few other meta symbols, which represent operations that allow:

- Concatenation: Symbols or strings may be concatenated by writing them next to one another, or by using the meta symbol '·' dot if further clarity is required. For example, regular expression 'a·b' the concatenation of 'a' and 'b', denotes the language that contains the string 'ab'. In general, the concatenation of two regular languages consists of strings that extend each string of the first language with all the strings of the second language.

- Alternation: A choice between two symbols a and b is indicated by separating them by the union operator, which is marked with the meta symbol '|' bar. For example, regular expression 'a|b' denotes the language that contains the strings 'a' and 'b'.

- Repetition: A expression followed by the meta symbol '*' star indicates that a sequence of zero or more occurrences of expression is allowable. This is also known as Kleene closure.

- Grouping: A group of symbols may be grouped by the meta symbols '(', ')' parentheses. For example, regular expressions '(a|b)' and '(a)|(b)' describe the same language as expression 'a|b'. Grouping is sometimes necessary to ensure that the expression has the intended meaning. Note that regular expression 'ab(a|b)' represents the language of 'aba' and 'abb', whereas the language described by 'aba|b' contains the strings 'aba' and 'b'.

Although the presentation of regular expressions is self-explanatory, it is helpful for many readers to give a little thought to the differences between the formal language theory and the practical use of regular expressions. The theory is concerned with recognizing which strings belong to a defined regular language. In practice regular expressions are used to search matches. If some substring matches to expression, its start and end position is returned to the user. For this kind of usage results could be ambiguous. For example if we search expression '(ab)*' from string 'aababababb', result is that first match starts from second character and ends to third, fifth, or seventh character. The behavior that returns the last result is known as greedy and behavior that returns the first possible result is known as reluctant.

5 Context-Free Grammars

This chapter describes the features that mainly concern context-free languages and grammars. The main focus is to find features that could help the building of the GDEL implementation. The first section defines a parse tree and how it is related to grammars. Equivalence and transformations of grammars are defined in second section. Third section discusses methods to clean up grammars. Ambiguities of grammars are described in the fourth section and last section contains information about abstract syntax trees.

5.1 Parse Tree

Parsing analyses the structure of string according to given grammar. The string with derivation structure is called a parse tree.

Definition: A labeled and ordered tree T is a parse tree for a context-free grammar $G = (A, N, s, P)$ iff the label of the root node of the tree is s. Each non-leaf node of the tree lebelled l_0 with child nodes labeled $l_1, \ldots, l_n$ defines the production $(l_0 \rightarrow l_n) \in P$. Each leaf node of the tree labeled l_0 defines the production $(l_0 \rightarrow \varepsilon) \in P$ or $l_0 \in A$.

Reverse operation is called yield. Yield computes a string from a tree by concatenating the labels of leaves of the tree from left to right. More precisely, it is defined as follows:

Definition: A yield of parse tree T is yield of the root node of the tree. A yield of node of the tree labeled l is l iff $l \in A$; otherwise it is concatenation of yields of the child nodes.

5.2 Equivalence of Grammars

In general we may be able to find several equivalent grammars for any language. Two grammars are said to be equivalent if they describe the same language, that is, can generate exactly the same set of sentences.

Definition: Two grammars G_1 and G_2 are weakly equivalent iff $L(G_1) = L(G_2)$.

This does not necessarily mean that grammars use the same nonterminals and productions.

Definition: Two grammars G_1 and G_2 are strongly equivalent if they are weakly equivalent and have the same structure in derivation trees.

If two grammars are strongly equivalent they are just notational variants, i.e. they are same grammar with different nonterminal names.

5.2.1 Grammar Transformations

Equivalence of grammars is needed to understand transformations of the grammars [Pepper99], [Behrens00]. When we say that a grammar G_1 can be transformed in another grammar G_2, we mean that there exists some procedure to obtain G_2 from G_1, and that G_1 and G_2 are at least weakly equivalent.

5.3 Purification

Context-free grammars can suffer only a small number of ailments. The only requirement is that there is exactly one nonterminal in the left-hand side of it's each rule.

There could still exist some classes that are almost certainly an error and should therefore be handled as errors in case of a user-specified grammar. These classes are undefined nonterminals, non-reachable nonterminals, and non-productive nonterminals. To clean up a grammar, it is necessary to first remove the non-productive nonterminals, then the undefined ones and then the non-reachable ones [Sippu88].

Undefined nonterminals: The right-hand sides of some rules may contain undefined nonterminals. This does not seriously affect the sentence generation process, if production of grammar containing an undefined nonterminal, there will be no match, and undefined nonterminal will be discarded. The rule with the right-hand side containing the undefined nonterminal will never be an issue and can be removed from the grammar. If we do this, we may of course remove the last definition of another nonterminal, which will then become undefined.

From a theoretical point of view there is nothing wrong with an undefined nonterminal, but if a user-specified grammar contains one, there is almost certainly an error, and any grammar-processing program should warn user if such an error occurs.

Non-reachable nonterminals: A nonterminal is called reachable or accessible if there exists at least one sentential form, derivable from the start symbol, in which it occurs. Else the nonterminal is non-reachable, actually they do not occur in any right-hand side of a reachable nonterminal. Again this is no problem, but almost certainly implies an error somewhere.

To clean up a grammar we need to find the reachable nonterminals and remove the non-reachable non-terminals rules. For this, we can use the following scheme: At first, the start symbol is marked: it is reachable. Then, any time as a yet unmarked nonterminal is marked, all nonterminals occurring in any of its right-hand sides are marked. In the end, the unmarked nonterminals are not reachable.

It is interesting to note that removing non-reachable nonterminals does not introduce non-productive nonterminals. However, first removing non-reachable nonterminals and then removing non-productive nonterminals may produce a grammar that contains again non-reachable nonterminals.

Non-productive nonterminals: Any Nonterminals that do not produce a sublanguage – no terminal derivation – is non-productive. In fact a nonterminal has a terminal derivation if and only if it has a right-hand side consisting of symbols that all have a terminal derivation.

To find out non-productive nonterminals we use a following scheme: We mark the nonterminals that have a right-hand side containing only terminals. Next, we repeat the process that marks all nonterminals that have a right-hand side consisting only of terminals and already marked nonterminals, while this process can mark even one new nonterminal. Now, the non-productive nonterminals are the ones that have not been marked.

Now it is possible to remove all rules that contain a non-marked nonterminal in either the left-hand side or the right-hand side. This removes non-productive nonterminals and non-productive rules of productive nonterminals. Note that this removal process does not remove all rules of a marked nonterminal, as there must be at least one rule for it with a right-hand side consisting only of terminals and marked nonterminals. On the other hand all rules for the start-symbol could be removed. In that case the grammar describes an empty language and the removal of undefined and non-reachable nonterminals leads to an empty grammar.

5.4 Ambiguous Grammars

Definition: A context-free grammar G is ambiguous if there is a string $x \in L(G)$ that has two or more distinct derivation trees. Otherwise G is unambiguous.

It is in general unresolvable whether or not a given context-free grammar is ambiguous. This implies that it is impossible to write a program that determines the (non) ambiguity of a context-free grammar. Normally ambiguity is a very undesirable property because of the lack of a unique interpretation of each sentence in the language. But in some cases it can be very useful, for example in software re-engineering to parse legacy programs [Brand98].

5.4.1 Disambiguation Rules

Recall that ambiguity means we have two or more derivations for the same input, or equivalently, that we can build more than one parse tree for the same input. A simple arithmetic expression grammar is a common example:

```
E → E + E | E * E | (E) | id
```

Parsing input: ‘id+id*id’ can produce two different parse trees because of ambiguity. The problem could be overcome by re-writing the grammar to introduce new intermediate nonterminals that enforce the desired precedence. A better solution is extending the parser with priority and associativity operators to disambiguate sentences containing occurrences of these operators. The same example with these operators is presented in section 6.5 where it is expressed in the SDF2 formalism.

5.4.2 Inherently Ambiguous Grammars

It is worth to notice that there are context-free languages that cannot be generated by any unambiguous context-free grammar. Such languages are said to be inherently ambiguous [Hopcroft79]. For example $L = \{a^m b^m c^n d^n \mid m; n > 0\} \cup \{a^m b^n c^n d^m \mid m; n > 0\}$. The reason is that every context-free grammar G must yield two parse trees for some strings of the form $x = a^n b^n c^n d^n$, where one tree intuitively expresses that x is a member of the first set of the union, and the other tree expresses that x is in the second set.

5.4.3 Loops

The above definition makes useful all rules that can be involved in the production of a sentence, but there still is a class of rules that are not really useful: rules of the form $A \rightarrow A$. Such rules are called loops. Loops can also be indirect: $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$. A loop can legitimately occur in the production of a sentence, but if it does there must also be a production of that sentence without the loop. Loops don’t contribute to the language. Any production that involves a loop is infinitely ambiguous, that produces infinitely many production trees for it. Algorithms for loop detection are given in Section 7.2.

Different parsers react differently to grammars with loops. Some – most of the general parsers – faithfully attempt to construct an infinite number of parse trees, some collapse the loop as described above and some – most deterministic parsers – reject the grammar. The problem is aggravated by the fact that loops can be concealed by ε-rules: a loop may only become visible when certain nonterminals produce ε [Grune90].

5.5 Abstract Syntax

The goal of this section is to introduce abstract syntax, and to show how to obtain an abstract syntax tree (AST) from a concrete syntax [Wile97]. Concrete syntax views data as written text. Parser generates a parse tree that reflects concrete syntax. Abstract syntax is a more abstract view of data – structured entities – which are the basis for describing the semantics of data. Abstract syntax trees reflect precisely the significant components, they are as simple as can be.

Let's consider how an AST differs from a parse tree. An AST can be thought of as a condensed form of the parse tree:

- Operators appear at internal nodes instead of at leaves.
- Chains of single productions are collapsed.
- Lists are flattened.
- Syntactic details (e.g., parentheses, commas, semi-colons) are omitted.

A common informal approach in the literature is to specify abstract syntax using standard context-free grammars, same as those used for concrete syntax. Thus, the right sides of rules are strings; however, the reader should interpret them as trees.

The informal approach is quite appropriate for the presentation and discussion of small language fragments. Because it is easier to write and understand a tree than a grammar, the tree can inform the reader of its concrete syntax. Names for the syntactic categories, and the used structures, are not often needed in a discussion. The most important point is that the reader knows what are the components of the structure. However, the approach is inadequate for the specification of the abstract syntax of full languages, especially if the syntax is to be implemented in a compiler, or used as a standard for exchanging programs between systems. Then, a precise specification must be provided.

Thinking of fragments as trees rather than strings is an important conceptual step. Once preformed, it is possible to go further. One may add operations to the structures used in the syntax trees that allow a user to traverse them and manipulate their components [Comon99]. The data structures thus become objects [Crew97].

In designing a CFG for the concrete syntax of a language, several issues need to be considered: ambiguity, operation associativity and precedence. But, in abstract syntax, these are non-issues. The important thing about trees is that, unlike strings, their compositional structure is inherently unambiguous. And they can represent expression structure and evaluation inherently in its structure without a grammar.

An XML document is defined as a tree and could been seen as a “concrete syntax for abstract syntax”. Note that the tag notation of XML ensures there is no syntactic ambiguity. Therefore XML could be the standard choice for representing syntax trees. XML also allows standard tools for syntax tree manipulation.

6 Syntactic Meta languages

The syntactic meta languages are used to describe possible infinite languages with notation that is finite. This chapter describes some common syntactic meta languages. Idea is to find good formalism on which to base building of GDEL grammar meta language.

6.1 BNF

Backus-Naur Form (BNF) is a syntactic meta language equivalent to context-free grammar. John Backus and Peter Naur defined BNF in conjunction with the group that developed Algol60 [Naur63]. The BNF uses abstractions to represent classes of syntactic structures – they act like syntactic variables, also known as nonterminal symbols. For example the rule that describes structure of while statement:

```
<whilestatement> ::= while <logicexpression> do <statement>
```

In rules variables are enclosed between '<' and '>'. The symbol '`::=`' is used to separate the left-hand side (LHS) which has a single nonterminal symbol and the right-hand side (RHS) which contains one or more terminal or nonterminal symbols. The grammar is a finite nonempty set of rules. In RHS of the rule the symbol bar '`|`' can be used to separate alternatives. Terminals are represented by themselves or are written in a typeface different from the symbols of the BNF.

Here is an example of expression grammar defined as BNF:

```
<goal> ::= <expression>
<expression> ::= <term> | <expression> - <term>
<term> ::= <factor> | <term> * <factor>
<factor> ::= a | b | c
```

6.2 ABNF

A modified version of BNF, called Augmented BNF (ABNF) is specified in Internet Engineering Task Force (IETF) Request for Comments (RFC) 2234 [RFC2234]. ABNF doesn't extend the expressive power of BNF, but tries to make it easier and more compact. It is intended mainly to help writing of other RFC's and Internet specifications. The differences between BNF and ABNF involve:

- Rule Naming: The name of a rule is simply the name itself. Enclosing angle brackets '<', '>' are not required. Rule names are also case-insensitive.
- Terminals: The terminal characters must be enclosed in quotes '"'
- Rule: The separator of RHS and LHS is '=' instead of '::='.
- Repetition: The repetition operator that can indicate least and most occurrences of element. A default repetition operator allows any number, including zero.
- Alternatives: Alternatives are separated by forward slash '/'. An alternative can be also so called incremental alternative which is represented as rule where separator of RSH and LSH is '=/'.
- Order-independence: The binding order of operators is, from (binding tightest) strings, names formation, comment, value range, repetition, grouping, optional, concatenation to alternative (binding loosest).
- Value ranges: A range of alternative numeric values can be specified compactly, using dash '-' to indicate the range of alternative values.

Here is an example of expression grammar defined as ABNF:

```
goal = expression
expression = term / expression "-" term
term = factor / term "*" factor
factor = %x61-%x63
```

6.3 EBNF

Many slightly different notations of BNF are in use. Different notations don't extend the expressive power of BNF formalism, but improve the readability. Extended BNF (ISO/IEC 14977:1996) is general-purpose, and its adoption will save time by avoiding the need to choose one of several suggested notations, which must be then amended to overcome their limitations [ISO14799].

Standardized version is based on one of many variations of BNF. The version from refence [Wirth77] has now become rather widely used. In this notation for EBNF: Nonterminals are written as single words, rather than inside enclosing angle brackets '<', '>' as in BNF notation. Terminals are written in quotes. Equal sign '=' is used in place of the sequence '`::=`'. Bar '`|`' is used, as before, to denote alternatives. Spaces are essentially insignificant and dot '`.`' is used to denote the end of each production.

Further defined metasymbols are: parentheses '`(`', '`)`' to denote nesting, brackets '`[`', '`]`' to denote the optional appearance of a symbol or a group of symbols, braces '`{`', '`}`' to denote optional repetition of a symbol or group of symbols, and sequences '`(*`', '`*)`' are used in some extensions to allow comments.

6.4 SDF

Heering et al. [Heering89] developed the Syntax Definition Formalism (SDF). A quick introduction to SDF can be found in reference [Visser00]. SDF defines the semantics by mappings to other formalisms. The Lexical syntax is mapped to a regular grammar and the Context-free syntax is mapped to a context-free grammar. SDF still integrates these two formalism in the level of it's own syntax. A parse tree for a string according to grammar is viewed as an abstract syntax tree over the signature by means of semantic definition. The correspondence of context-free grammars and algebraic signatures to define the semantics of programming languages were showed by Goguen et al. [Goguen77].

Heering claims that: "*SDF emphasizes compactness of syntax definitions by offering (a) a standard interface between lexical and context-free syntax; (b) a standard corresponding between context-free and abstract syntax; (c) powerful disambiguation constructs; (d) list constructs; and (e) efficient incremental implementation which accepts arbitrary context-free syntax definitions.*"

6.5 SDF2

SDF2 was developed by Visser [Visser97a, 115-213] as a generalization of SDF. It is a complete family of syntax definition formalisms. This is possible by using simple kernel and extensions. Extensions are defined in terms of the primitives of the kernel by means of normalization, which is provided through rewriting.

The kernel of SDF2 is a context-free grammar with additionally defined character classes, priorities, reject productions and follow restrictions. The grammar is defined in terms of a list of used nonterminals (called sorts) and list of productions. Priority declarations are defined by using two productions connected with priority relation which is one of; 'left', 'right', 'assoc', 'non-assoc', or '>'. Reject productions are marked. Follow restrictions are defined as character class that cannot follow a symbol (nonterminal or terminal).

The kernel of this family can be extended with many orthogonal extensions containing modules, literals, lexical and context-free syntax, aliases, regular expressions, variables, renaming. These features are eliminated in normalization process. The normalized expression has the same meaning as the original. Thus, normalization is a mapping from the language onto the same language. Normalization of SDF2 is defined as pipeline of normalizations and is therefore easily extensible and modifiable.

Following example of expression language should clarify SDF2 functionality:

```
sorts Id Exp
lexical syntax
  [a-z]+   -> Id
  [\ \t\n] -> LAYOUT
context-free syntax
  Id          -> Exp
  Exp "*" Exp -> Exp {left}
```

```
    Exp "+" Exp -> Exp {left}
    "(" Exp ")" -> Exp {bracket}
  context-free priorities
    Exp "*" Exp -> Exp >
    Exp "+" Exp -> Exp
```

The first line defines the nonterminals used in the grammar. Next line declares the start of the block that contains lexical syntax of language. The third line says that identifiers are list of one or more lowercase letters – note that '+' is actually kernel extension. The fourth line contains symbol '`LAYOUT`' with special meaning as definition of layout that can appear between context-free tokens. Layout is defined to contain spaces, tabs and new lines. The fifth line starts a block of context-free syntax. The sixth line contains a production rule that denotes that expression can be derived from the identifier – note that the production rules left- and right-hand sides are flipped other way around to make similarity to the function declaration more apparent. The next two lines contain two more production rules to expression, with an attribute that declares productions to be left associative. The next line starts a block of priority rules and last two lines contain a priority rule that declares that multiplication has a higher priority. Note that the grammar without association or priority rules is ambiguous.

6.6 Related Specifications

Various extensions of context-free grammars have been developed. Some of these extensions attach semantics to grammars like attribute grammars [Knuth68], affix grammars [Koster71], extended affix grammars [Watt77], definite clause grammars [Peraire80], assertion grammars [Ragget99], blindfold grammars [Hawke01], and SDF+ASF [Heering89]. Parser generator tools also uses extensions like YACC [Jonhson75], all of these extensions are specific to the tool used.

6.7 Conclusion

Simplest formalizations collect only those productions with equal left-hand side into one rule by allowing alternatives like BNF. An advantage of this approach has a strong relation to the underlying mathematical model. Drawbacks are the need of additional recursive rules to describe iterative structures (lists, etc…) and only way to resolve ambiguities of the grammar is grammar modification, which makes definitions lengthy and often difficult to follow.

Next levels extend metalanguages by more convenient description of repetitive and optional constituents, like EBNF. Remarkably, these formalisms have also a mathematical background as they describe solutions if the rules of a grammar are interpreted as a system of equations. The main disadvantage is the complex binding between nonterminals and nodes of syntax tree.

Due to the use of parser generators, metalanguages are enhanced by unambiguity declarations. Additionally, some other metalanguages are tailored to special needs and contain therefore some special features. SDF2 is pure syntactic metalanguage, with unified, modular and extensible feature set. Seems that using it, as a base of GDEL grammar metalanguage, would be sensible.

7 Context-free Parsing Techniques

From a practical point of view, grammars may be used to solve *membership problem* – given a string over A, does it belong to language L(G), where grammar G = (A, N, s, P) representing some language. Another problem is the so-called *parsing problem* – Find a sequence of rewriting steps from the grammar's start symbol to the given sentence. Parsing can be seen as structuring the input according the given grammar. The algorithm that makes structuring is called a parser.

This chapter describes approaches to solve *parsing problem*, with parsing algorithms that are able to handle any context-free grammars, even ambiguous ones. The idea of chapter is to describe parsing algorithms that are suitable for GDEL processor.

7.1 Common parsing algorithms

The most commonly known context-free parsing algorithms are top-down and bottom-up parsing. In top-down parsing parser begins with the start symbol of the grammar and attempts to generate the same sentence that it is attempting to parse. The most commonly known top-down parsing algorithms are LL, which is described in detail in most compiler texts, including the reference [Aho86].

In bottom-up parsing parser matches the input of the right-hand side of the productions and builds a derivation tree in reverse. The bottom-up parsing uses traditionally one symbol lookahead to guide the choice of action. The most commonly known LR, SLR and LALR algorithms are described in detail in most compiler texts, including references [Aho72], [Aho86], [Appel98] and [Grune90].

Commonly these parsing algorithms are limited to working on subclasses of context-free grammars [Grune90]. Hierarchies of subclasses are shown in the figure 4. Descriptive powers of these subclasses are limited, which means in most cases that syntax specification needs modifications before it is are in a usable form. Using a modified grammar has the disadvantage that the resulting parse trees will differ to a certain extent from the ones implied by the original grammar.

Available parser generator tools commonly support only some subclasses. Yacc [Johnson75], SableCC [Gagnon98], CUP [Hudson97] and most of the other supports LALR(1). ANTLR [Parr95], PCCTS [Parr97] and some others support LL(k) parsing.

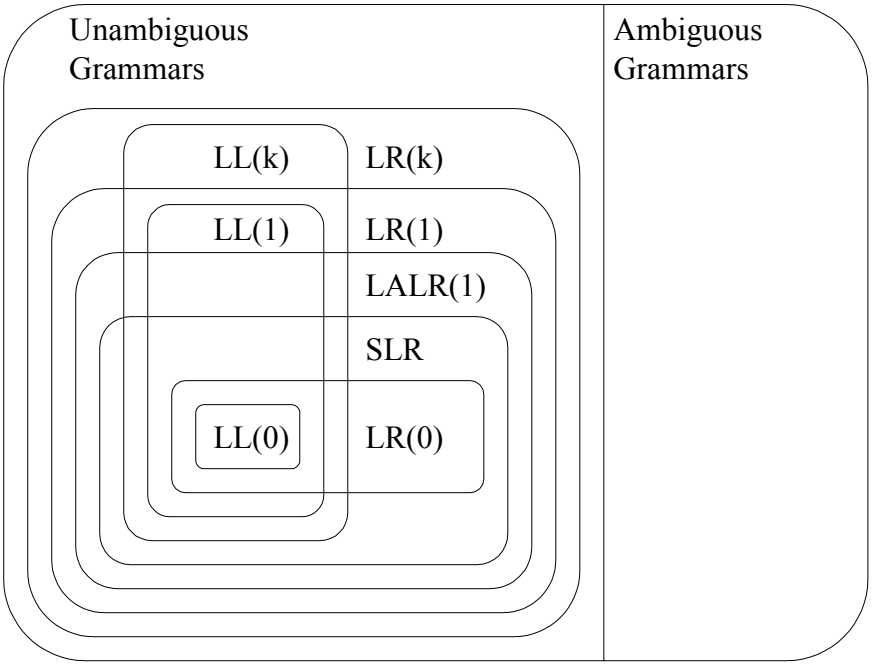

Figure 4. Hierarchy of Context-free grammar classes [Appel98].

7.2 Unger parsing

An Unger parser [Unger68] is the simplest known method to parse any context-free grammar. It's exponential time requirement limits it's applicability to occasional use. The Unger parser presented as Pascal program can be found at reference [Grune90, 253-263]. Algorithm goes as follow:

For each right-hand side production of grammar we must first generate all possible partitions of the input sentence. Generating partitions is not difficult: if we have m productions in right-hand side, numbered from 1 to m, and n is length of input, numbered from 1 to n, we have to find all possible partitions such that the numbers of the characters for each production are consecutive, and any production does not contain lower-numbered characters than any character in a lower-numbered production.

Partition fails if a terminal symbol in a right-hand side does not match the corresponding part of the partition. The non-failed partition results will all lead to similar split-ups as sub-problems. These sub-problems must all be answered in the affirmative, or the partition is not the right one.

For a grammar that contains loops there are infinitely many derivations to be found. So, the process needs to avoid the problem by cutting off the search in these cases. Maintaining a list of partitions that we are currently investigating can do this. If a new partitioning already appears in the list, we do not investigate that and proceed as if the partition was answered negatively. Fortunately, if the grammar does not contain such a loop, a cut-off will not do any harm either, because the search is doomed to fail anyway.

The original paper extends the described method with a series of tests to avoid partitioning that could never success. For instance, a section of the input is never matched against a nonterminal if it begins with a token that no production of the nonterminal could begin with. Several such tests are described and ways are given to statically derive the necessary information (FIRST sets, LAST sets, EXCLUDE sets) from the grammar. Although none of this changes the exponential character of the algorithm, the tests do result in a considerable speed-up in practice.

7.3 Generalized LR parsing (GLR)

The basic idea of GLR parsing is exploring all of the possible actions of LR automaton on a given input string, and outputting a parse forest of all of the possible derivations of string. Original idea was proposed by Lang [Lang74]. First implementation, based on this approach, has been given by Tomita [Tomita86]. The Tomita algorithm was later found to fail to terminate on grammars with hidden left recursion, and the correction is provided by Nozohoor-Farshi [Nozohoor-Farshi91]. It also could not handle looping grammars. Rekers has generalized Tomita's algorithm to looping grammars [Rekers92]. He also describes method for the incremental parser construction. Pseudocode of algorithm goes as follow [Reakers92, 25-27]:

```
PARSE (Grammar, a1 … an) :
    an+1 := EOF
    global accepting-parser := ∅
    create a stack node p with state START-STATE (Grammar)
    global active-parsers := {p}
    for i := 1 to n + 1 do
        global current-token := ai
        PARSEWORD
    if accepting-parser ≠ ∅ then
        return the tree node of the only link of accepting-parser
    else
        return ∅

PARSEWORD :
    global for-actor := active-parsers
    global for-shifter := ∅
    while for-actor ≠ ∅ do
        remove a parser p from for-actor
        ACTOR (p)
    SHIFTER

ACTOR (p) :
    forall action ∈ ACTION (state (p), current-token) do
        if action = (shift state´) then
            add <p, state´> to for-shifter
        else if action = (reduce A ::= α) then
            DO-REDUCTIONS (p, A ::= α)
        else if action = accept then
```

```
        accetting-parser := p

DO-REDUCTIONS (p, A ::= α) :
    forall p´ for which a path of length(α) from p to p´ exists do
        kids := the tree nodes of the links which form the path from p to p´
        REDUCER (p´, GOTO (state (p´), A), A ::= α, kids)

REDUCER (p¯, state, A ::= α, kids) :
    if ∃p ∈ active-parsers whith state(p) = state then
        if there already exist a direct link link from p to p¯ then
            ADD-RULENODE (treenode(link), rulenode)
        else
            n := GET-SYMBOLNODE (A, rulenode)
            add a link link from p to p¯ with tree node n
            forall p´ in (active-parsers – for-actor) do
                forall (reduce rule) ∈ ACTION (state(p´), current-token) do
                    DO-LIMITED-REDUCTIONS (p´, rule, link)
    else
        create a stack node p with state state
        n := GET-SYMBOLNODE (A, rulenode)
        add a link link from p to p¯ with tree node n
        add p to active-parsers
        add p to for-actor

DO-LIMITED-REDUCTIONS (p, rule, link) :
    forall p´ for which a path of length(α) from p to p´ through link exists do
        kids := the tree nodes of the links which form the path from p to p´
        REDUCER (p´, GOTO (state (p´), A), A ::= α, kids)

SHIFTER :
    active-parsers := ∅
    create a term node n with token current-token
    forall <p¯, state´> ∈ for-shifter do
        if ∃p ∈ active-parsers whith state(p) = state´ then
            add a link from p to p¯ with tree node n
        else
            create a stack node p with state state´
            add a link from p to p¯ with tree node n
            add p to active-parsers

GET-RULENODE (r, kids) :
    return a rule node with rule r and elements kids

ADD-RULENODE (symbolnode, rulenode) :
```

add *rulenode* to the possibilities of *symbolnode*

GET-SYMBOLNODE (s, rulenode) :
return a symbol node with symbol *s* and possibilities { *rulenode* }

GLR parsing can be summarized as doing breadth-first search over those parsing decisions that are not solved by the LR automaton – shift-reduce and reduce-reduce conflicts in parse table. Search paths are investigated simultaneously and, whenever an inadequate state is encountered on the top of the stack, the following steps are taken:

1. For each possible reduce in the state, a clone of the stack is made and the reduce is applied to it. This removes part of the right end of the stack and replaces it with a nonterminal; using this nonterminal as a move in the automaton, we find a new state to put on the top of the stack. If this state allows again reductions, this step is repeated until all reductions have been treated, resulting in equally many stacks.

2. Stacks that have a right-most state that does not allow a shift on the next input token are discarded.

Note that if all stacks are discarded in step 2 the input was in error, at that specific point. For looping grammars process needs additional termination rules. There are two solutions: upon creating a stack, check if it is already there and then ignore it or check the grammar in advance for loops and then reject it.

Actual implementations of the GLR parser use a graph-structure stack, which is a generalization of the parse stack, to represent multiple parse stacks. Graph-structure stacks use pointers to connect stack elements, which allow shared parts of the parse stack represented only once and therefore make fast stack cloning possible.

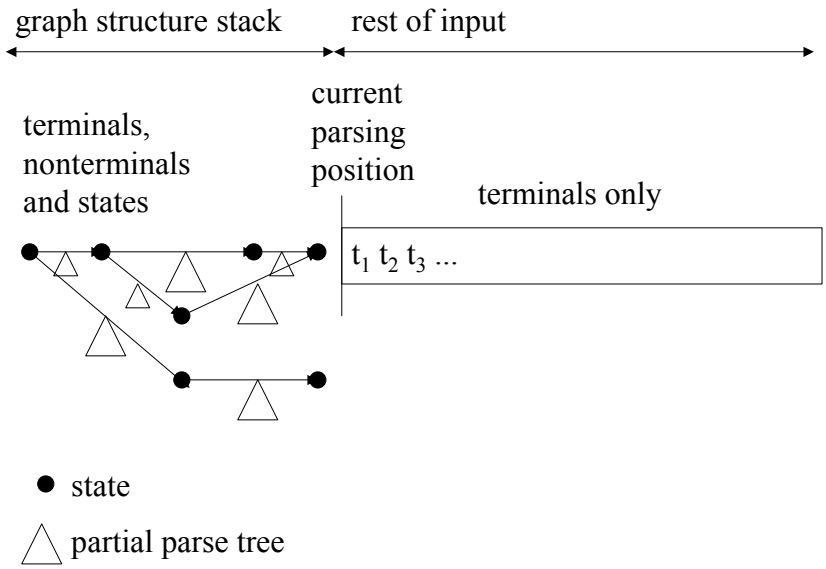

Figure 5. The structure of GLR parse.

Experimental data on what type of table is optimal to restrict the breadth-first search can be found at reference [Reakers92, 15]. If the automaton uses look-ahead, this is of course taken into account in deciding which reduces are possible in steps 1. An LR(0) table is relatively easy to construct and should give reasonable results even if it causes more stack copies and subsequently discard operations than any other automaton. SLR(1) table construction needs only an additional follow set calculation and might be preamble. In a view of the additional construction effort, an LALR(1) table may not have any advantage over the SLR(1) table in this case. An LR(1) table probably requires too much space.

The worst-case time complexity of GRL algorithm is $O(n^{p+1})$, where n is the length of the input and p is the length of the longest rule of the right side [Nederhof92]. The bounds as mentioned above are, however, for the absolute worst cases. For the most practical grammars, like those of programming languages, the GLR parsing algorithm gives a linear performance. The theoretical solution is given in [Kipps91] as variant, that has fixed the worst-case time complexity $O(n^3)$. In the practical cases this solution worsens time complexity instead of improving it. Current research makes faster versions of GLR algorithm [Aycoc99], [Aycoc01a].

7.4 Earley Parsing

Earley's parser can be described as a breadth-first top-down parser with a bottom-up recognition [Earley70]. It can also be seen as breadth-first bottom-up parsers that will restrict the fan-out to reasonable proportions – reductions are restricted to only those reductions that are derived from the start symbol.

Just as in the case of non-restricted algorithm, at all times there is a set of partial solutions that is modified by each symbol that is read. The sets shall be written between the input symbols; earlier sets have to be kept, since the algorithm will still use them.

Unlike the non-restricted algorithm, in which the sets contained stacks, the sets consist of what are technically known as Earley items. An item is a grammar rule with a gap in its right-hand side; the part of the right-hand side to the left of the gap – which may be empty – has already been recognized, the part to the right of the gap is predicted. An Earley item is an item with an indication of the position of the symbol at which the recognition of the recognized part started.

The sets of items contain exactly those items a) of which the part before the dot in the Earley item has been recognized so far and b) of which we are certain that we shall be able to use the result when they will happen to be recognized in full – but we cannot, of course, be certain that recognition will happen.

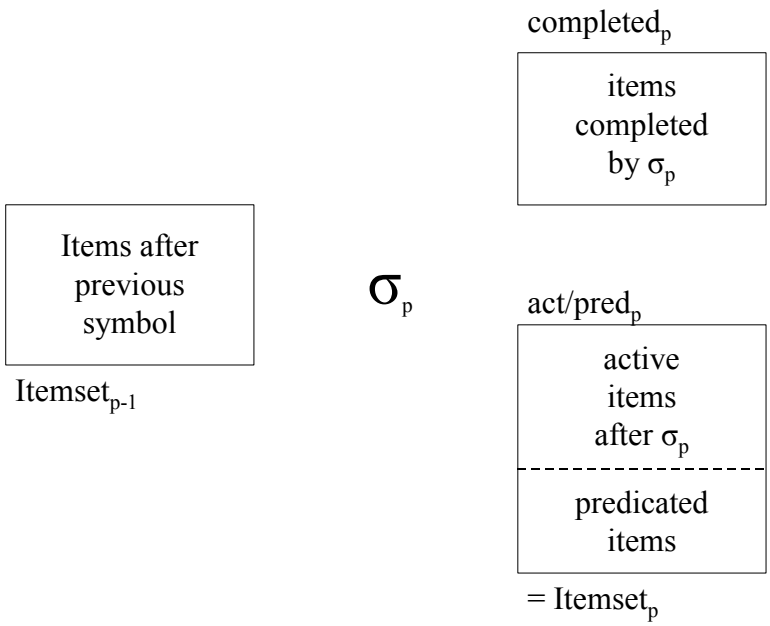

Figure 6. The Earley item set for one input symbol [Grune98].

The construction of an Earley item set from the previous Earley item set proceeds in three phases, 'Scanner', 'Completer' and 'Predictor'. Scanner corresponds to 'shift' and Completer correspond to 'reduce' in the non-restricted algorithm. The Predictor is new and is related to the top-down component. Pseudocode of algorithm goes as follow:

```
PARSE (Grammar, a1 … an) :
    an+1 := EOF
    global chart[n + 1]
    chart[0] := { [0, 0, START-STATE (Grammar)] }
    PREDICT (0)
    COMPLETE (0)
    for i := 1 to n + 1 do
        SCAN (i, ai)
        COMPLETE (i)
        PREDICT (i)
    if chart[n + 1] ≠ ∅ then
        return MAKE-TREE (chart)
    return ∅

PREDICT (position) :
    forall [i, position, A ::= B * C D] ∈ chart[position] do
```

forall ($C ::= E$) $\in$ *Grammar* **do**
ADD-EDGE ([*position*, *position*, $C ::= * E$], *position*)

SCAN (*position*, α) :

forall [*i*, *position* - 1, $A ::= B * \alpha\ C$] $\in$ *chart*[*position* - 1] **do**
ADD-EDGE ([*i*, *position*, $A ::= B\ \alpha * C$], *position*)

COMPLETE (*position*) :

forall [*j*, *position* - 1, $A ::= B\ C\ D\ *$] $\in$ *chart*[*position*] **do**

forall [*j*, *i*, $E ::= F * A\ G$] $\in$ *chart*[*j*] **do**
ADD-EDGE ([*i*, *position*, $E ::= F\ A * G$], *position*)

ADD-EDGE (*edge*, *position*) :

if not *edge* $\in$ *chart*[*position*] **then**
add *edge* on *chart*[*position*]

The Scanner, Completer and Predictor deal with four sets of items for each token in the input; 1) an item set contains the items available just before token is completed, 2) a completed set, which is the set of items that have become completed due the token, 3) an activated set contains the non-completed items that passed token, 4) a predicted set of newly predicted items.

Initially, set 1 is filled, as a result of processing earlier token, or initialized by algorithm if tokens are not yet read, and the other sets are empty.

The Scanner goes through the whole item set and makes copies of all items that contain token as the next symbol. All other items are ignored. Consequently, the Scanner changes place of the item. If the item is now at the end, it stores the item in the set completed; otherwise it stores it in the set active.

Next the Completer inspects the completed set, which contains the items that have just been recognized completely and can now be reduced as follows. For each item the Completer calls the Scanner with an item set that Earley item shows, which is now directed to work on the token recognized by the Completer and given the item set. It will make copies of all items in given item set featuring changes of item place and store them in either set completed or in the set active, as appropriate. This can add indirectly recognized items to the set completed, which means more work for the Completer. After a while, all completed items have been reduced, and the Predictor's turn has come.

The Predictor goes through the active set – which was filled by the Scanner – and the predicted set – which is empty initially – and considers all nonterminals that have an Earley item in front of them; these we expect to see in the input. For each predicted nonterminal and for each rule for that nonterminal, the Predictor adds an Earley item that points to next token of input in front of productions and put that construction to the set predicted. This may introduce new predicted nonterminals in set predicted which cause more predicted items. After a while, this will stop too.

The sets active and predicted together form the new item set for the next input token. If the completed set for the last symbol in the input contains an Earley item that spanning the entire input and reducing to the start symbol, we have found at least one parsing.

All this does not directly give us a parse tree, just a recognizer. However, the intermediate sets contain enough information about fragments and their relations to construct a parse tree easily. A simple top-down Unger-type parser can serve for this purpose, since the Unger parser needs only the lengths of the various components of the parse tree and that is exactly information that the Earley parser provides. Earley's original article gives a method of constructing the parse trees while parsing, by keeping with each item a pointer back to the item that caused it to be present, but it has shown to produce incorrect parse trees on certain ambiguous grammars [Tomita86, 74-77].

The correspondence between the Earley and the CYK algorithms has been analyzed by Graham and Harrison [Graham76]. A simple, robust and efficient version of the Earley parser has been resulted in a combined algorithm described by Graham, Harrison and Ruzzo [Graham80].

The worst-case time complexity of Earley parsing requires a time proportional to $O(n^3)$ for ambiguous grammars, at most $O(n^2)$ for unambiguous grammars and at most O(n) for grammars for which a linear-time method would work. These time complexity calculations does not contain a time needed to ouput parse tree(s). A fast implementation of algorithm can be found from reference [Aycoc01b].

7.5 Other Algorithms

A generalized version of recursive descent parser (GRD) is proposed by Johnstone and Scott [Johnstone97a], [Johnstone97b]. It can parse any non-left recursive grammar. In principle this technique could be used to any context free grammar by removing left recursion, by using a well-known algorithm from reference [Aho86, 176-178].

Generalized Left-Corner Parsing (GLC) has been examined by Nederhof [Nederhof92] and Leemakers even without any mention of connections with left corner parsing [Leemakers91]. It is very interesting because it should beat both Earley and GLR in time complexity.

7.6 Conclusion

By allowing any context-free grammar, maximal freedom is given to the writers of specifications. Finding so many parsing algorithms, which can be a reasonable time to parse any context-free grammar and are therefore possible to use in the implementation of GDEL processor, is amazing.

The existence of powerful enough algorithms should mean existence of parser generators that uses them, which is not the case. We really need feedback from theory into engineering practice, which currently uses unpleasant tricks to solve problems of limitedly powered parser generators.

There is also a good opportunity for the future study of relative speeds of different algorithms. There could be also a possibility to predict which algorithm gives best speed with given grammar and build a state of the art parser generator as a result.

8 GDEL Language Design

This chapter presents the design of Grammar-based Data Extraction Language (GDEL). The first section contains the goals of overall design. The second section describes selection of grammar syntax, while actual design decisions about the XML-syntax of the grammar found in next chapter. The third section considers how language semantics could be expressed. The fourth section contains considerations about the syntax tree binding to XML-format.

8.1 Design Goals

The main design goal of Grammar-based Data Extraction Language is a high-level specification language that can specify transformation of any context-free language to XML.

Keywords MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when they appear in requirements, are used in the sense of how they would be used within other documents with the meanings as specified in Internet Engineering Task Force (IETF) Request for Comments (RFC) 2119 [RFC2119].

All requirements are given with reference numbers, where sub requirements are separated with dots. All requirements are given with RATIONALE, which gives a more verbose explanation why the requirement is given.

General GDEL language requirements:

1. GDEL language MUST be descriptive, expressive, formal and concise. RATIONALE: The language should be as easy to write and understand as possible.

2. GDEL language MUST be executable in the sense of an algorithm to convert from the instance of the specification to a sequence of operations that carry out the XML transformation task. RATIONALE: The language processor must be implementable and runnable without any further information from user.

2.1. GDEL language MUST define the context-free grammar of the input. RATIONALE: The context-free grammar class is the largest class of grammars where input can be efficiently parsed with general parser. Secondly without defined grammar process has no sense.

3. GDEL language MUST be defined in XML format. RATIONALE: The language processor implementation should be as easy as possible and XML format as language definition allows implementers to use general XML parsers to implement language processor. XML format allows writing GDEL scripts which automatically or semiautomatically generates GDEL scripts from other syntax specifications like BNF or SDF. XML format also allows writing XSL scripts, which transfers the specification to documentation or other specifications.

4. Complete list of errors MUST be defined. RATIONALE: The user of the language should know what kinds of errors are possible and why these errors would occur.

4.1. Each occurrence of errors MUST specify the severity level as warning, error or fatal error. RATIONALE: Different severities are needed to specify different behavior of processor of the language.

4.1.1. The occurrence of the warning SHOULD NOT affect to processing. RATIONALE: Example non-reachable nonterminals in grammar do not affect to processing, and an error can therefore be a warning severity. It is also possible to define warning in cases where error changes result, without consequences.

4.1.2. The occurrence of errors SHOULD NOT stop processing. RATIONALE: The error shows to user that the result of processing is not correct. Errors should not lead to complete failure of the process.

4.1.3. The occurrence of the fatal error MUST stop processing. RATIONALE: The fatal errors lead to the complete failure of process.

5. GDEL language SHOULD be modular. RATIONALE: A good modular structure is important, because it allows more elegant specifications in the means of reuse and lifetime support.

6. GDEL language SHOULD be visualizable. RATIONALE: This is intended to help users to create and understand the transformation specifications.

7. GDEL language MUST be extensible. RATIONALE: This allows GDEL grammars to be used on other tasks that need grammars. It also allows extensions that help the writing of more complex specifications.

8. GDEL language MUST allow the use of any context-free grammar to specify input. RATIONALE: This is intended to allow reuse of grammars from other specifications without a complex grammar re-engineering, but not without a syntax transformation to GDEL syntax.

9. GDEL language SHOULD define grammars for the outputs (schema for XML output). RATIONALE: The output grammar depends on the input grammar and has some additional information like XML tag names. This information given as XML Schema helps users.

9.1. GDEL processor MUST guarantee that the output belongs to the defined class. RATIONALE: The processor implementation is certainly erroneous if this requirement is not true.

10. Creation of GDEL grammar MUST be possible automatically or semiautomatically from other grammar formalisms. RATIONALE: This requirement is intended to ensure that the transformation of other grammar syntaxes to GDEL syntax is possible.

8.2 Syntax design

The GDEL specification must define the context-free grammar of the input as requirement 2.1 states. This grammar should be presented as XML as the requirement 3 states. There are many syntactic meta-languages as we have seen in chapter 6 but none of these are defined in XML format. In this section XML syntax presentations are consider for these meta-languages and finally define XML-based grammar format.

Another points of design are modularity and extensibility stated in requirement 5 and 6. Modularity is gained by using simple kernel and extensions. Extensions are defined in terms of the primitives of the kernel by means of normalization, which is provided through rewriting functions. This approach is adopted from Eelco Visser PhD Thesis where approach is used to define Syntax Definition Formalism (SDF2) [Visser97a]. For more detail description of SDF2 see section 6.5.

The first design point of kernel is to limit kernel features. SDF2 kernel contains context-free grammar, character-classes, and priorities, reject productions and follow restrictions. Eelco Visser's further work shows that context-free grammar and priorities could also be expressed as character-class grammar [Visser97d]. The selection of kernel features is equal to SDF2 kernel; because usage of character class-grammars binds the implementation too tightly to scanner less generalized LR-parsing. The selection of SDF2 also allows direct translation of SDF2 grammars to GDEL grammar format, and to the other direction as well. Actual design decisions about the XML-syntax of grammar found in next chapter.

8.3 Semantics of GDEL

Formal specifications play an important role in the software design, especially in the language design. The most important point is that the formal specifications could be used as an unambiguous interface between designers, programmers and users. An algebraic specification could be used for purpose.

The algebraic specification consists of signature as a definition of the syntax and a set of axioms to specify the semantics. The meaning of the algebraic specification is specified as the isomorphic class of all initial models. These models are any algebraic structures satisfying given axioms. A unique symbolic model constructed from Herbrand's universe of all well-formed terms by the smallest congruence relation generated by axioms of a specification always exists. Therefore it could be used as starting point if it could be found. The result is so-called decision procedure for the equality problem in the defined class. A decision procedure is usually modeled by a term rewriting system.

How to construct an appropriate term rewriting system for a given algebraic specification? A simple method is to transform the term rewriting system to Prolog or another similar high-level logical programming language [Bergstra89]. The other possibility is to use dedicated languages and design environments like SDF+ASF Meta Environment [Brand01].

In case of GDEL design variation of the first approach could be chosen, even the later approach is already used to define SDF2 formalism, which GDEL grammar is based. XSL transformations [XSL] could be used for term rewriting purpose, because GDEL is an XML-based language. This approach would fit in the most natural way. The input specification could be expressed as an XML-document. The signature could be expressed as an XML Schema [XMLSchema]. The semantic part of a specification would be converted into XSL-code and used as a rewriting system. The prototyped expression can be formed as an XML-term, which could be rewritten with the help of XSL into a canonical form – the meaning of this term.

8.4 Syntax tree binding to XML format

Syntax tree binding to the XML format could be done several ways. One possibility is to set an id attribute for each production of the grammar and then point each production from a separate part where XML syntax is declared. This approach allows a flexible way to define how the syntax tree is outputted as XML, but at the same time it makes GDEL language harder to write and more complex to implement. Another way is to set the id attribute for each production of the grammar and use the id as an XML element name in output. This approach fits better to our needs so we select it. It is also allows an easy way to define XSLT stylesheets when the more complex output is needed.

9 GDEL Grammar Syntax

This chapter contains actual design decisions concerning the GDEL grammar syntax. The first section considers a basic syntax of the grammar. The second section defines XML syntax for terminals. The third section considers regular expression syntax. The fourth section introduces syntax for disambiguation productions. The fifth section defines a layout and sixth modularization of the grammar.

The syntax for the GDEL grammar is defined with prototyping method. Initial basic syntax definition is taken from BNF and then features are added incrementally towards SDF2 definition. Each step contains also explanation why certain changes are made. The final definition for GDEL grammar can be found at Appendix 4.

9.1 Basic Syntax

Here is an example of an expression grammar defined as BNF:

```
<goal> ::= <expression>
<expression> ::= <term> | <expression> - <term>
<term> ::= <factor> | <term> * <factor>
<factor> ::= a | b | c
```

The expression grammar example above could be described as follow:

```
<?xml version="1.0"?>
<Grammar>
  <Rule>
    <Rhs>goal</Rhs>
    <Lhs>
      <Nonterminal>expression</Nonterminal>
    </Lhs>
  </Rule>
  <Rule>
    <Rhs>expression</Rhs>
    <Lhs>
      <Alternative>
        <Nonterminal>term</Nonterminal>
      </Alternative>
      <Alternative>
        <Nonterminal>expression</Nonterminal>
        <Terminal>-</Terminal>
```

```
        <Nonterminal>term</Nonterminal>
      </Alternative>
    </Lhs>
  </Rule>
  <Rule>
    <Rhs>term</Rhs>
    <Lhs>
      <Alternative>
        <Nonterminal>factor</Nonterminal>
      </Alternative>
      <Alternative>
        <Nonterminal>term</Nonterminal>
        <Terminal>*</Terminal>
        <Nonterminal>factor</Nonterminal>
      </Alternative>
    </Lhs>
  </Rule>
  <Rule>
    <Rhs>factor</Rhs>
    <Lhs>
      <Alternative>
        <Terminal>a</Terminal>
      </Alternative>
      <Alternative>
        <Terminal>b</Terminal>
      </Alternative>
      <Alternative>
        <Terminal>c</Terminal>
      </Alternative>
    </Lhs>
  </Rule>
</Grammar>
```

XML syntax could be made more concise as the GDEL language requirement 1 says:

```
<?xml version="1.0"?>
<Grammar>
  <Rule nonterminal="goal">
    <Nonterminal name="expression"/>
  </Rule>
  <Rule nonterminal="expression">
    <Nonterminal name="term"/>
  </Rule>
  <Alternative nonterminal="expression">
    <Nonterminal name="expression"/>
    <Terminal>-</Terminal>
    <Nonterminal name="term"/>
```

```
    </Alternative>
    <Rule nonterminal=”term”>
      <Nonterminal name=”factor”/>
    </Rule>
    <Alternative nonterminal=”term”>
      <Nonterminal name=”term”/>
      <Terminal>*</Terminal>
      <Nonterminal name=”factor”/>
    </Alternative>
    <Rule nonterminal=”factor”>
      <Terminal>a</Terminal>
    </Rule>
    <Alternative nonterminal=”factor”>
      <Terminal>b</Terminal>
    <Alternative>
    <Alternative nonterminal=”factor”>
      <Terminal>c</Terminal>
    </Alternative>
  </Grammar>
```

The DTD of complete XML BNF format can be found at Appendix 1.

The next SDF2 example in section 6.5 should be considered. It’s XML form could be follow:

```
  <?xml version=”1.0”?>
  <Grammar>
    <Sorts>
      <Sort type=”Id”/>
      <Sort type=”Exp”/>
    </Sorts>
    <LexicalSyntax>
      <Rule nonterminal=”Id”>
        <Terminal>[a-z]+</Terminal>
      </Rule>
      <Rule nonterminal=”LAYOUT”>
        <Terminal>[\ \t\n]</Terminal>
      </Rule>
    </LexicalSyntax>
    <ContextFreeSyntax>
      <Rule nonterminal=”Exp”>
        <Nonterminal name=”Id”/>
      </Rule>
      <Rule nonterminal=”Exp” assoc=”left” id=”Mul”>
        <Nonterminal name=”Exp”/>
        <Terminal>*</Terminal>
```

```
      <Nonterminal name="Exp"/>
    </Rule>
    <Rule nonterminal="Exp" assoc="left" id="Add">
      <Nonterminal name="Exp"/>
      <Terminal>*</Terminal>
      <Nonterminal name="Exp"/>
    </Rule>
  </ContextFreeSyntax>
  <ContextFreePriorities>
    <Priority type=">">
      <Ref rule="Mul">
      <Ref rule="Add">
    </Priority>
  <ContextFreePriorities>
</Grammar>
```

Note, that the given SDF2 and BNF are quite similar in the selected XML form, exceptions are following: BNF contains elements named as Alternative. In BNF elements named as Rule are children of the root element, while SDF2 elements are children of LexicalSyntax and ContextFreeSyntax elements. SDF2 contains elements named as Sorts, Sort, LexicalSyntax, ContextFreeSyntax, ContextFreePriorities, and Priority. In some cases the SDF2 element named Rule contains the attributes named associativity and id. Also contents of Terminal elements are different.

If Alternative element is considered, it is possible to write BNF without the element in the means of converting all Alternative elements to Rule elements, which copy the attribute nonterminal from the previous-sibling element. Additional elements and attributes that SDF2 has, are all additional features of SDF2. A similarly changed place of Rule elements is a result of the separation of the lexical and context-free syntax.

9.2 Terminals

The contents of Terminal elements, as in the previous example, are the only problems. Terminals in BNF are always single characters or literal strings (see subsection 9.2.1). In SDF2 terminals can be characters or character classes. There is also a difficulty caused by the XML syntax. The XML specification [XML] defines follow:

```
Char ::= #x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD]
| [#x10000-#x10FFFF]
```

The meaning is that XML uses all other Unicode characters except surrogates – Unicode reserves these to represent the start of characters '`[#x10000-#x10FFFF]`' – and control characters '`[#x00-#08]`', '`#x0B`', '`#x0C`' and '`[#x0E-#1F]`'. The control characters can not be presented as XML characters. This causes problems, especially with binary formats. A solution for these problems is to redefine the Terminal elements. An example:

```
<Terminal>*</Terminal>
```

should be changed to

```
<Char value="*"/>
```

Hexadecimal character '`#x1D`' would be presented as:

```
<Char value="0x1D"/>
```

Character classes

```
<Terminal>[a-z]+</Terminal>
<Terminal>[\ \t\n]</Terminal>
```

should be changed to

```
<CharacterClass>
  <OneOrMore>
    <Range start="a" end="z"/>
  </OneOrMore>
</CharacterClass>
<CharacterClass>
  <Char value="0x20"/>
  <Char value="0x09"/>
  <Char value="0x0A"/>
</CharacterClass>
```

In addition to the normal union, SDF2 character class can also be a difference or an intersection of two character classes, or a complement respect to the complete character class. The following example shows three different ways to construct equal character classes:

```
<CharacterClass>
  <Difference>
    <Range start="a" end="m"/>
    <Range start="j" end="m"/>
  </Difference>
</CharacterClass>
<CharacterClass>
```

```
    <Intersection>
      <Range start="a" end="m"/>
      <Range start="a" end="i"/>
    </Intersection>
  </CharacterClass>
  <CharacterClass>
    <Range start="a" end="d"/>
    <Range start="e" end="i"/>
  </CharacterClass>
```

9.2.1 Literal Strings

Literal strings would be defined as:

```
<String value="xyz"/>
```

where '`xyz`' is literal string. In SDF2 the normalization of strings creates the new production that contains the string as separated terminal characters – see section 10.2 for information about GDEL normalization process.

9.3 Regular Expressions

Commonly context-free grammars contain patterns, that occur again and again. Some examples of such patterns are optional constructs and iterations. Therefore many formalisms are extended with regular operators, that provide shortcuts for such patterns. SDF2 regular operators are defined via extra productions generated by a normalization function – see section 10.2 for information about GDEL normalization process.

In SDF2 the regular expressions provide abbreviations for grouping '`(`', '`)`', zero or more iteration '`*`', one or more iteration '`+`', optional constructs '`?`', and alternatives '`|`'. An example of regular expression features:

```
Identifier ("=" (Number|Identifier))?
```

There are several possibilities for the XML syntax. The DSD uses following elements: ‘<Sequence>’ for groups, ‘<Optional>’ for ‘?’, ‘<ZeroOrMore>’ for ‘*’, ‘<OneOrMore>’ for ‘+’, and ‘<Union>’ for alternatives [DSD], [Karlund00]. The XML Schema uses a different scheme [XMLSchema]: ‘<sequence>’ for groups, ‘<choice>’ for alternatives, and attributes minOccurs and maxOccurs to limit the occurrence of other constructions. Both syntaxes use similar elements for grouping. For alternatives could also be considered an element ‘<OneOf>’. The final version of expression of the previous example as the XML is following:

```
<Nonterminal name=”Identifier”/>
<Optional>
  <Char value=”=”/>
  <Alternatives>
    <Nonterminal name=”Number”/>
    <Nonterminal name=”Identifier”/>
  </Alternatives>
</Optional>
```

SDF2 provides an abbreviation for following operators: concatenation, grouping, zero or more, one or more, optional constructs, alternatives, iteration, list, set expressions, and permutation. The complete XML syntax for the GDEL grammar can be found at Appendix 4.

9.4 Productions for Grammar Disambiguation

SDF2 uses priority declarations, reject productions, and follow restrictions to disambiguate the grammar. Priority declarations can be a priority-chain or an associativity relation: ‘left’, ‘right’, ‘assoc’, or ‘non-assoc’. The highest priority in the priority-chain comes first. The priority declaration can be written inside a separated tag ‘<Priorities>’. The associativity reletations can be also written directly to the production rule attribute, named ‘assoc’. Objects of these declarations are single productions or groups of productions.

The reject productions are marked directly to the production rule attribute, named ‘reject’ with the value ‘true’. The follow restriction contains the class of characters that cannot follow the production. Follow restrictions are used to simulate greedy matching (see chapter 4.4). The follow restriction syntax is:

```
<FollowRestriction rule=”rulename”>
  <CharacterClass>
    <Range start=”a” end=”z”/>
  </CharacterClass>
</FollowRestriction>
```

The complete XML syntax for the GDEL grammar can be found at Appendix 4.

9.5 Layout

SDF2 defines both the syntax of sentences – the context-free syntax – and the syntax of tokens – the lexical syntax – within single definition. The distinction between tokens and sentences is that the tokens making up sentence can be separated by layout – whitespaces etc. – while the characters making up a token cannot. The XML syntax for lexical and context-free parts are allredy defined in the first section of this chapter.

The layout that can occur between tokens must be also specified. In SDF2 the symbol ‘LAYOUT’ is reserved for this purpose and it must be defined inside the lexical syntax section. In the XML syntax layout can be written inside a ‘<Layout>’ element. In example, layout is defined as

```
<Layout>
  <Nonterminal name=”Comment”/>
</Layout>
```

meaning that comments are layout.

Notice that the abstract syntax tree do not contain layout tokens. Therefore result XML do not contain layout tokens. SDF2 generic layout definition mechanism works well for programming languages. It also works for simple data formats, but it fails to provide the correct abstract syntax and output for complex data formats such as EDIFACT [EDIFACT]. The reason is that these formats uses a kind of layout to pointing different parts. Therefore GDEL allows the '`<Layout>`' element usage also inside the context-free syntax section, there the '`<Layout>`' element define parts that are not included to the abstract syntax.

9.6 Modularization

SDF2 uses alias and module mechanism for reusing parts of syntax definitions. The XML syntax could use '`<Alias>`' element, with the attribute '`name`' when it is defined, and with the attribute '`ref`' when it is referenced.

Last syntax definition need to be defined is syntax for modules. Consider following module:

```
module Substitutions
import Terms
       Tables[Key => Var]
```

As following XML representation:

```
<?xml version="1.0"?>
<Module name="Substitutions">
  <Import module="Terms"/>
  <Import module="Tables">
    <Rename nonterminal="Key" to="Var"/>
  </Import>
</Module>
```

XML Schema for grammar found at appendix 4.

10 GDEL Processor

This chapter describes two different implementation approaches – interpreter and compiler – for the GDEL processor. These approaches create the framework where GDEL specifications are easy to write and test from developer point of view and give enough performance in production environments. From developer point of view interpreter approach is a better solution, because time consuming parse table generation phase can be made incremental for rules that are currently needed for parsing [Reakers92]. For production environments, processor with compiled specification gives a better performance and should therefore be a better solution. We have also developed a prototype implementation of GDEL processor, which is briefly described in the last section.

10.1 Overview

The black box view of processor shows the overall process – see figure 2 in page 9. Four main parts can be identified: GDEL specification, input material, processor, and output XML.

In design time it is necessary to create the specification that defines the grammar of input material, and XML tag names for output. In the interpreter approach the specification is directly set to processor before actual processing starts. In the compiler approach the specification is compiled to actual processor.

At run time input material is given to the processor, which then parses input as described by the grammar of the specification. Parsing creates a parse tree – or a forest if the grammar is ambiguous – and returns it in XML format.

If we look in more detail inside the processor, we can find four subproceses: the normalization of specification, the generation of the parse table or the parser, the actual parsing process, and the process that generates the XML output.

The normalization process takes the specification as input and rewrites it to the core language. This phase simplifies the parser generation dramatically, because only basic functionality needs to be supported. Normalization also allows using a very simple extension mechanism to extend GDEL specifications, because we can just add a new normalization step for each extension of specification language.

The parser generation creates the actual parser from the normalized specification. This process separation allows the selection of the parse algorithm used. It also gives the possibility to use nearly same processor for interpreters and compilers. The result of the process is a parse table, if we use interpreter approach, or compiled parser component, if we use compiler approach.

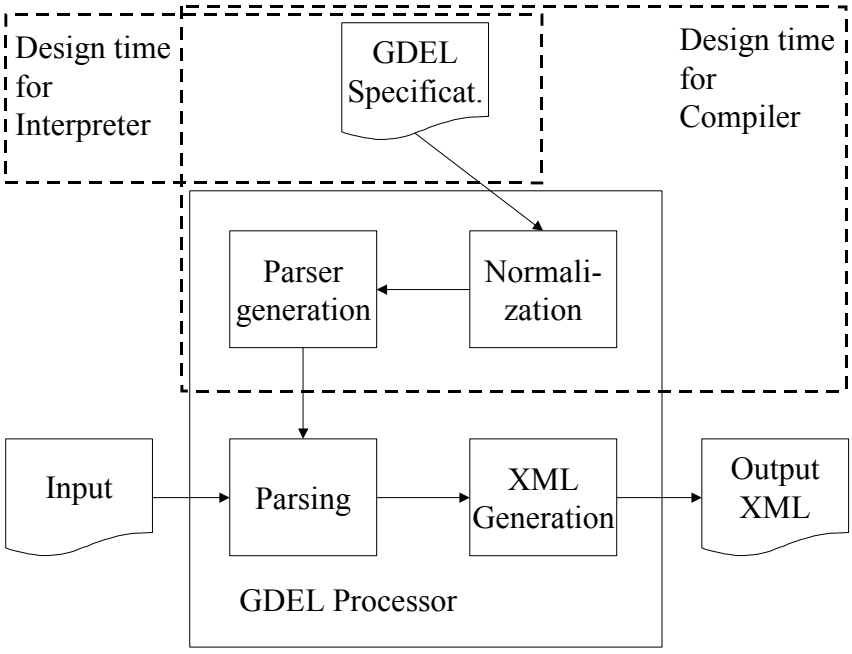

Figure 7. A more detailed picture of the processor.

The parsing process takes the input material and parses it. Parsed material is then changed to XML in the XML generation process. The XML generation process is separated for two reasons. First, its removal allows the GDEL specifications to be used as a classic parser generator, and secondly, it allows an easy way to support different XML APIs – like SAX, DOM – for output.

10.2 Normalization

The normalization takes the specification as input and rewrites it to the core language. The core language contains only basic syntax, terminals without literal strings, and productions for grammar disambiguation. This phase simplifies the parser generation dramatically, because only the basic functionality needs to be supported. Other fuctionality like literal strings and regular expression operators are normalized back to core language. The normalization also allows using very simple mechanism to extend GDEL specifications, because there could just be added a new normalization step for each extension of specification language.

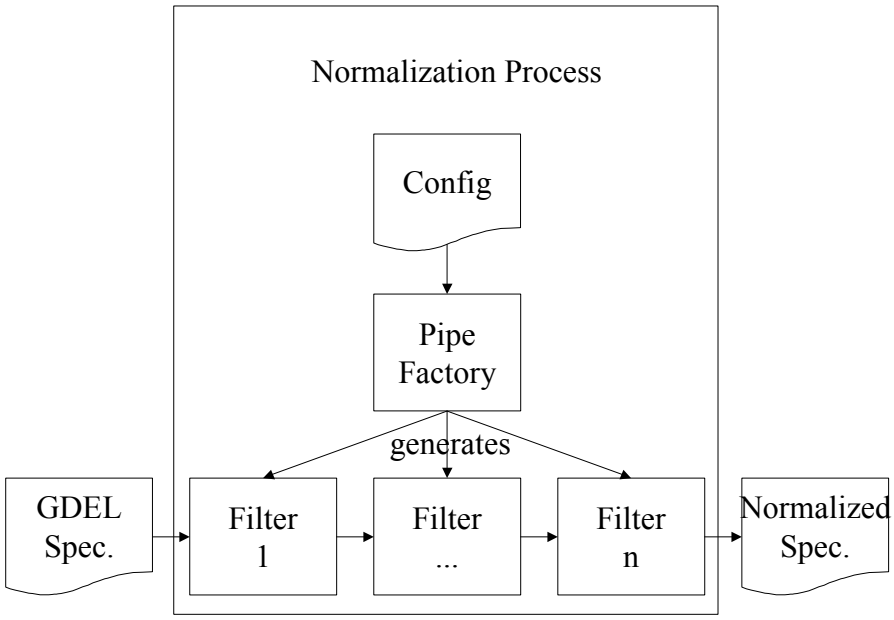

Figure 8. Normalization process

Actual normalization process is a pipeline of normalization steps. Because the normalization input and output are in XML form we use XSLT for each step except the character class normalization. Using the commonly supported XSLT makes implementation of normalization very easy and it also serves people who want to make extensions to the language.

The best design of normalization process uses Pipes & Filters design pattern [Buschmann96] with Factory pattern [Gamma95] that configures the process. More details of these design patterns can be found from [Gamma95], [Buschmann96]. Another description of the used design pattern can be found in reference [Spinellis01]. The configuration information should be defined in a way that allows extensions of the language without any code changes for GDEL processor implementation.

The normalization must also expose syntactic errors from the specification and report them as a fatal error. If each normalization step is reliable, the syntactic errors should be tested only in the first step. Other sources of fatal errors are missing specification or errors in configuration.

10.3 Parser generation

The parser generation creates an actual parser from normalized specification. This process separation allows selection of used parse algorithm. It also gives possibility to use nearly same processor for interpreters and compilers. The result of process is a parse table if we use interpreter approach or compiled parser component, if we use compiler approach.

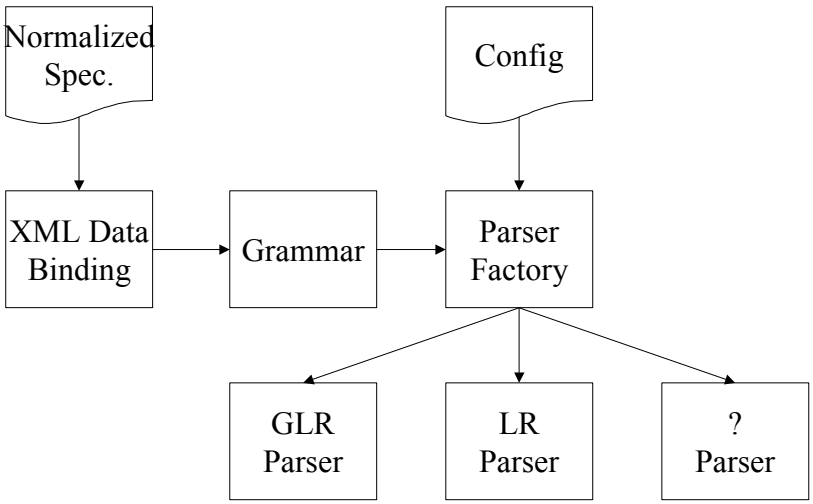

Figure 9. Parser generation process.

According to our experiences, the best way to make a grammar object is to use the technique known as XML data binding, which marshals the data from XML instance into the properties of classes [JAXB]. The main benefit of data binding is much sorter and cleaner code for XML handling. From implementer point of view it could also reduce errors and development time. The grammar object is used for internal representation of given specification. The grammar object allows querying needed parts of grammar.

A parser factory can use the configuration data to select the wanted parser type and finally create a parser. It should be noticed that in case of LR and GLR parsers their parse table generation is similar and LR parsing should always be used when the result parse table does not contain ambiguities. It is also notable that Earley parser uses nearly similar items that are used in the LR parse table generation. These facts allow lots of reuse in the finer levels of design. We believe that actually there is a possibility to create a very good parser framework with little code if design is correct.

For interpreter it is also possible to use an incremental parsing method. That means the parse table is constructed at run time, and only the parts that are currently needed are calculated. This allows faster parsing especially in cases where the grammar changes constantly. See [Reakers92] and [Visser97a].

The parser generation process should generate an error if the given grammar is ambiguous and the currently configured parsing method does not support ambiguous grammars. For better usability it could support automatic change of parsing method and continuation of process. If supported, this feature must be configurable. Else it should report a fatal error and stop. Another usability feature that could be supported is purification. For more details on purification see section 5.3. Every problem that is purified should generate warning and processing should continue normally.

10.4 Parsing

The parsing process reads the given input material, parses it and creates a parse tree as a result. The most practical approach for parsing is scanner less parsing, because it simplifies design and implementation. For more detailed information of the advantages see [Visser97a, 30-36].

The instance of the reader is used for the input material reading. Reader's actual task is to support the wide variety of the input encoding – like: ASCII, UTF-8, EBCDIC – and change these to the Unicode supported by parser. A scanner could replace the reader if it is needed. The Factory pattern is used to create the reader of the configured type. It allows the selection of optimized readers for different encoding. The actual parser and the parse tree generation depend on the type of the parser.

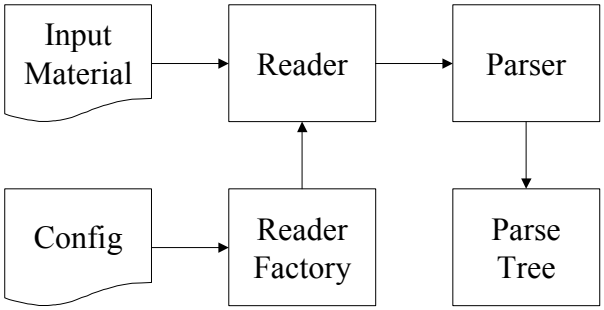

Figure 10. Parsing process.

If reading of the input fails – because of the I/O errors etc. – the processor must report a fatal error and stop. If input material contains syntactic errors, parsing fails and the processor must generate an error report. If the processor has a recovery mechanism and it succeeds, it could continue, else the error is fatal. For better usability the parsing error reports should identify location from input material, where the error is detected – in most cases the actual cause of an error is before that position – and current state of parsing.

10.5 XML generation

The parsing process generates an internal representation of a parse tree or a forest. The XML generation process creates the actual XML representation from the internal representation. The XML generation process should be separated for three reasons. First it makes the implementation cleaner, secondly it gives an easy way to support different XML APIs – like SAX and DOM – for output. And third its removal makes it possible to use the GDEL processor like classic parser generators.

The best way to implement the output method selection is the Factory pattern that creates the instance of output component. It takes the information of which output component should be created from the configuration.

An actual parse tree or forest can be gone through by the generic Visitor pattern that uses given output component. The Visitor design patterns can be found from [Gramma95], [Cooper98]. Note that output components could have a very simple interface, because the generated XML contains only elements and parsed character data. If more complex XML structure is needed the Visitor implementation should be changed to use SAX API directly. The Visitor also needs the instance of grammar, which it uses for looking up the names of output XML elements.

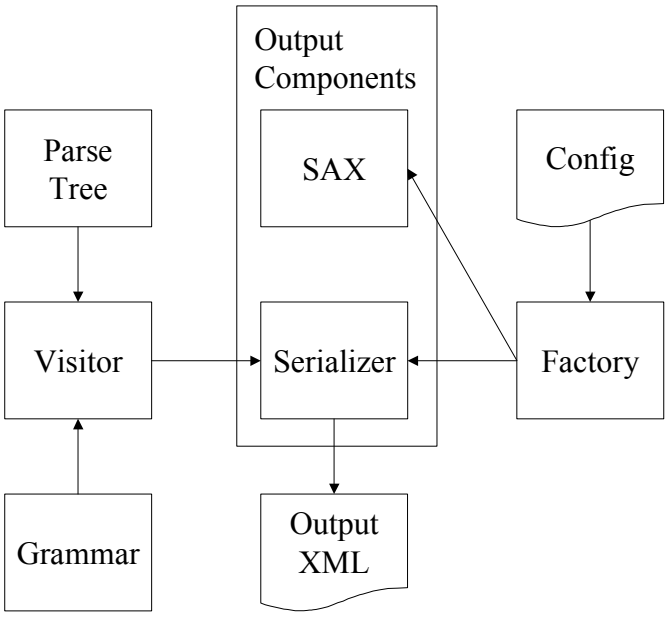

Figure 11. XML generation process.

10.6 Prototype Implementation

A prototype implementation of the GDEL processor has been developed in the Java programming language [Java]. Java was chosen because it is platform independent. The implementation conforms to the JAXP 1.2 Transformation API for XML, also known as TrAX [JAXP].

A prototype implementation needs an environment in which has been installed: Java virtual machine version 1.3 or higher [JavaVM], XSLT processor that supports TrAX – prototype has been tested with Xalan-Java version 2.4.1 and version 2.5.1 [Xalan] and Saxon version 6.5.2 [Saxon] –, Log4j version 1.2.8 or higher [Log4j] – for logging component messages in a common way – and X-Fetch Performer version 2.2 or higher [Performer].

10.6.1 TrAX

There is a need for Java applications that are able to transform XML and related data structures. The TrAX attempts to define a model that is clean and generic, yet fills general application requirements across a wide variety of uses. It defines objects and methods for processing input and producing output in a variety of formats.

The abstract model of TrAX usage is following: A TrAX TransformerFactory object processes transformation instructions and produces Templates objects. A Templates object provides Transformer objects, which can transform one or more Source objects into one or more Result objects.

There might be one minor missing feature from TrAX. A TransformerFactory object is normally a globally declared property and there's no need to worry anything about an implementation. In this case it's needed to use the XSLT processor during usage of the GDEL processor. Both have their own TransformerFactory objects. One more line in the GDEL configuration can tell where XSLT TransformerFactory is and setting global property point to GDEL TransformerFactory can solve the problem. As a result it is impossible to run the XSLT from other applications, because the GDEL processor doesn't understand XSLT stylesheets.

How to solve the problem? Because both languages are in the XML format it is possible to automatically detect the transformation language. If configuration tells how all possible root tags and namespaces are related to the TransformerFactory object it is possible to redirect used instructions to the correct object. The implementation of the GDEL processor has now this feature. Hopefully someday every TrAX user can enjoy it.

10.6.2 X-Fetch Performer

X-Fetch Performer is a second-generation XML processor. It combines features of filtering, SAX and DOM interfaces, and also provides help for XML Data Binding. Actually X-Fetch Performer is only dynamic XML Data Binding solution. It also solves some of the speed and memory problems of XML handling. For more details, see the X-Fetch web site (http://www.x-fetch.com/).

X-Fetch Performer makes XML very easy to handle. The actual implementation contains only two small classes to handle XML, one to solve TrAX transformation language redirection – mentioned in previous section – and another to bind the normalized GDEL specification to Java objects. The same task by using SAX or DOM interface would need ten times more implementation work, which also raises the possibility of errors, makes future changes harder, and increases the code maintenance work.

10.6.3 Normalization

The implementation of normalization process uses the standard XSLT processor. Running many XSLT stylesheets in the pipeline is not as efficient as it should be. One reason could be poorly optimized XSLT processor implementations – We have been testing Java versions of Xerces-Java versions 2.4.1 and 2.5.1 [Xerces] and Saxon 6.5.2 [Saxon]. One possible solution is to use Xerces XSLT compiler and compile each stylesheet. As the better approach we suggest a new model where XSLT transformation pipeline is used as the design time representation and compiling phase changes these to single stylesheet – in theory sequential transformations can be replaced by a single transformation.

The normalized grammar is bonded to the Java grammar object through the X-Fetch Performer. In the grammar all strings – names of elements, nonterminal names, etc. – are stored to the container class that bases on the ternary tree structure. The container allows minimal memory usage, it has O(1) time complexity of direct get operations, and O(log(n)) time complexity for put, seek and get operations. The priority and follow restriction information are both stored to own classes that are optimized. All other grammatical information is stored to an object network, which makes some operations easy to implement but is not very efficient.

10.6.4 Parsers

The prototype parser generation and parsing supports LR and GLR parsing algorithms with LR0, SLR, LR1 parse tables. Adding the LALR parse table generation support needs changes only in one class. Also adding a new parsing algorithm affects only one class.

Because of implementation conforms to TrAX requirements, it must also support compiled transformers. This is currently done in a minimal possible level by using serializable Templates objects. We can see only little performance benefits if these are changed to real byte code level compiling.

10.6.5 XML generation

The XML generation supports currently natively only serialized output, otherwise it is implemented in the way described earlier in this chapter. SAX and DOM support work through XSLT transformation, which does not change the result.

10.6.6 Notes

Most of the testing of prototype is done by functional tests. These tests should form later the base for the complete regression test set. Ideally the same test set could be used to other possible implementations and also for benchmarking purposes.

10.7 Conclusion

This chapter has shown one of the possible designs of the GDEL processor. It is based on good design principles like the separation of concerns, modularity, and extensibility. The design also shows that design patterns could improve designing. The prototype implementation shows that the processor is quite easy to create. Especially it shows that the usage of external parts could reduce the implementation time.

In this chapter one missing minor feature of TrAX has been pointed out, and a way it could be corrected has been presented. There is also a need for a new kind of software that could optimize transformation pipeline to a single transformation that concerns especially sequential XSLT processing.

11 Applications of GDEL

This chapter describes some applications of the GDEL that are essential for any grammar system. These applications do not contain normal usage, which is clear enough – some common usage scenarios can be found at reference [Lempinen03]. These applications are very easy to create in GDEL and normally quite hard for other grammar systems.

The first application shows how GDEL can be used to create new GDEL specifications from other grammar specifications formalism by extracting the necessary information. The second application automatically generates XSLT stylesheet that convert extracted data back to its original form. The third section describes an application that generates XML schema for output XML. The fourth section considers some other possible applications.

11.1 Converting other specifications to GDEL

As shown in Chapter 6 there are lots of different grammar specification formalisms. One of the desired properties of GDEL is that there is the possibility of writing specifications to extract new specifications from other grammar formalisms. This is possible because nearly all grammar metalanguages have their own descriptions as grammars. Using the grammar of a grammar in GDEL form allows us to obtain data from grammar in XML form. XSLT could be then used to transform the obtained XML form of grammar to the new GDEL specification. Generally this seems to be the most practical way to transform existing grammars to GDEL specification.

11.2 Bi-directional transformations

In some use cases there is need for a bi-directional transformation. Bi-directionality means that data can be transformed in both directions. This is also a desired property of GDEL. The bi-directional transformations are easy to support, because there could be made XSLT stylesheet that transforms the given GDEL specification to XSLT stylesheet that outputs original data from extracted data (see figure 12). The actual XSLT stylesheet that does the necessary transformation can be found at Appendix 2. In some ways similar bi-directional transformations system for SDF2 could be found in [Brand96].

11.3 Generation of XML Schema for output

In some cases it is desired that the XML Schema of the processor output is known. It is possible to generate the needed schema instance directly from GDEL specification by using XSLT transformation. The XSLT stylesheet for this operation can be found at Appendix 3.

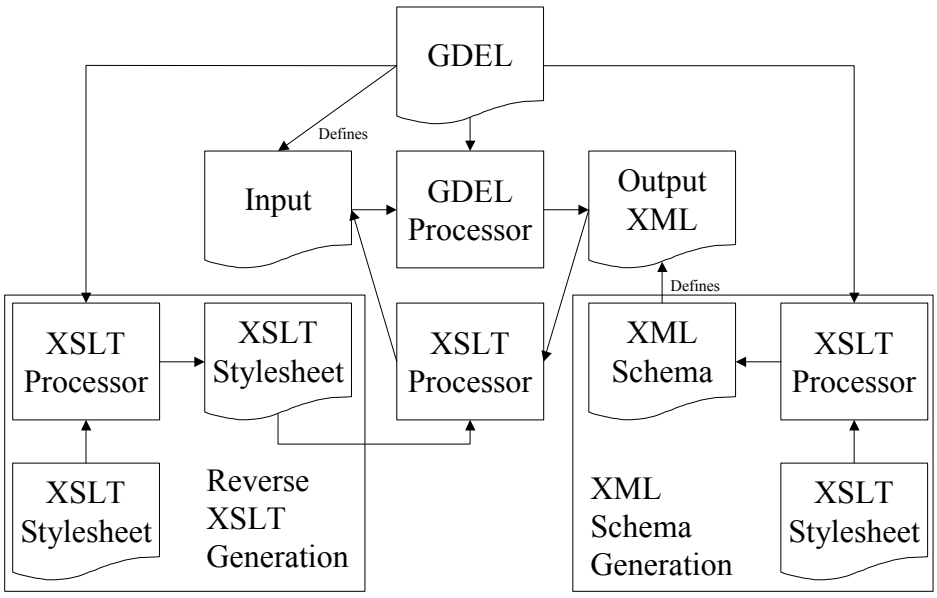

Figure 12. GDEL transformation to reverse XSLT and XML schema.

11.4 Other applications

Grammar visualization could be also done as a simple XSLT transformation from GDEL specification to the SVG picture. It is also possible to do other kinds of reports from the grammar in similar manner.

Several markup techniques for the source code have been proposed: [Badros00], [Boshernitsan00], [Holt00], [Mamas00] and [Maletic02]. They do not give the solution for straightforward source code transformation to marked up code. We think that GDEL specification and processing could help to create solution for that kind of problems. It could be also interesting to investigate GDEL usage to the area of program transformations [Feather86], [Jonge00a-b], [Karsai00] and [Murata97].

11.5 Conclusion

As seen, the GDEL allows an easy creation of many applications that are hard to implement in other kinds of grammar systems. It is basically the result of the choice that GDEL specifications are in the XML form.

12 Epilogue

Famous Richard Hamming used to ask three questions for new hires at Bell Labs [Hamming86]. The first question is “What are you working on?” the second “What's the most important open problem in your area?” and the final question, “Why aren't they the same?” How do I answer these questions? The First is simply, “Data transformations systems.” second “Find to final solution to so called legacy data problem. The final solution must prevent legacy data to even exist.” and answer for final question, “They are actually same. First we need the bi-directional transformations and then the automatic generation of these. Then legacy is no more legacy.”

12.1 Conclusions

The minimal standard of the well-defined grammar systems contains two fundamental properties. First it must allow different applications in a given implementation, and secondly different implementations in a given application. In this sense GDEL is a well-defined grammar system.

The grammar part of the GDEL syntax was a derived form of SDF2 [Visser97a]. If differences between these formalisms are thought, there can be found: GDEL semantics is not currently as well defined as SDF2 semantics. XML makes grammars bigger and quite hard to write with plain text editors, but there is also a possibility to write grammar in another formalism and transform it to GDEL. Benefits of XML usage are more important. It shows an easier way to extend language through normalization by writing the XSLT stylesheet. The implementation of the processor is easier because of XML data binding and very easy implementation of the normalization process. XML also allows easier usage of grammar information in the other applications like reverse transformation and grammar visualization.

GDEL uses XML as an output format of parsed data. If we consider data integration usage, this approach is clearly good. For other possible usages – like parser generators – it could create too much overhead in final application. This is taken account in the modular processor design.

12.2 Future Directions

In the near future Republica will include GDEL processor to the family of X-Fetch components. It will be freely available for academic use. There is also a small possibility that processor implementation would be published as an open source. If that happens there could also be a library of grammars for GDEL – similar to the grammar base for SDF grammars [GBASE]. The future development of GDEL specification would be donated to some standardization organization or community, because otherwise it's significance will be small.

There are lots of possibilities for further improvements. From the parser implementation view, there is a good opportunity for the future study of relative speeds of different algorithms. There could also be a possibility to predict which algorithm gives the best speed with the given grammar and build state of the art parser generator as a result. The grammar visualization and it's visual editing could also be interesting.

The automatic optimisation of the transformation pipeline could be an interesting research issue. Another open research issue is the possibility to integrate DEL specification to GDEL as a so-called island grammar [Moonen01], [Moonen02]. We also see that term rewriting with XSLT as a general extension mechanism for XML-based languages could be studied more closely.

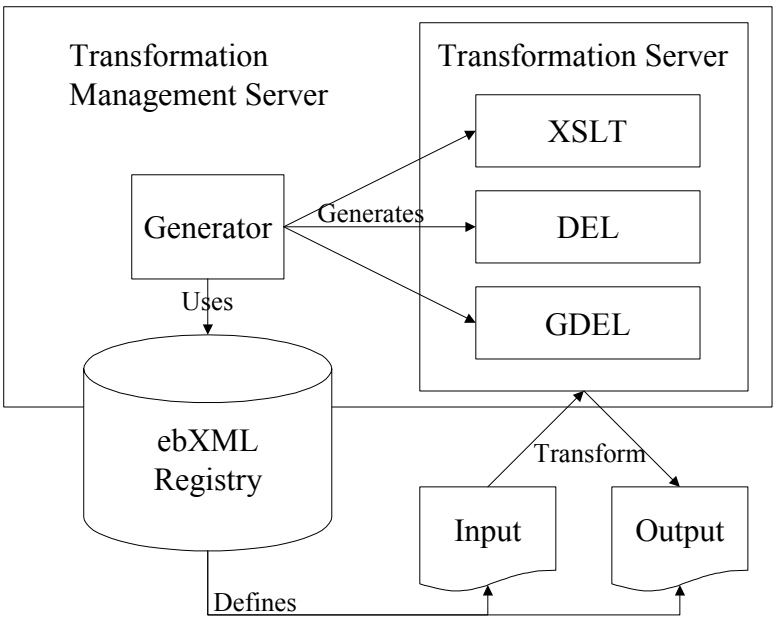

Figure 13. Transformation Management Server.

From the data integration point of view bi-directionality is important, but not enough to solve all data integration problems. The complete solution must also manage these transformations in terms of semantics. My future research would be concentrated to Transformation Management Server architecture shown in figure 13. There ebXML registry is used to define semantics for transformed data and additional layout properties of possible formats of data [ebRS][ebXML]. Therefore it could allow an automatic generation of the transformation specifications – at last in a level where semantically similar parts are transformed to each other [Behrens00]. In this architecture GDEL could be a valuable part.

References

The reference entries are indexed by the (first) author of the document if the source is a book or an article, and by the title of the document if the source is a specification. All online resources are accessed on 25th of June 2003.

[Adelberg98] B. Adelberg, "NoDoSe: A Tool for Semi-Automatically Extracting Semi-Structured Data from Text Documents", SIGMOD Conference 1998: 283-294.

[Aho72] A. V. Aho, J. D. Ullman, "The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing", Prentice-Hall, 1972.

[Aho86] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers. Principles, techniques, and Tools", Addison-Wesley, 1986.

[Alur01] D. Alur, J. Crupi, D. Malks, "Sun Java Center J2EE Patterns", First Public Release: Version 1.0 Beta, the Sun Java Center, March 2001, Available at: http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/

[Appel98] A. W. Appel, "Modern Compiler Implementation in Java", Cambridge University Press, 1998.

[Ashish97a] N. Ashish, C. A. Knoblock, "Wrapper Generation for Semi-structured Internet Sources", SIGMOD Record 26(4), pp. 8-15, 1997, Available at: http://www.isi.edu/sims/naveen/sig.ps

[Ashish97b] N. Ashish, C.A. Knoblock "Semi-automated Wrapper Generation for Internet Information Sources", Proceeding of 2nd IFCIS Int. Conference on Cooperative Information Systems (Coopis '97), Kiawah Island, June 1997, Available at: http://www.isi.edu/sims/naveen/autowrap.ps

[Aycock01a] J. Aycock, R. Horspool, J. Janousek, B. Melichar, "Even Faster Generalized LR Parsing", Acta Informatica, vol. 37, 9 (2001), pp. 633-651. Available at: http://www.csr.uvic.ca/~nigelh/Publications/czech.pdf

[Aycock01b] J. Aycock, N. Horspool, "Directly-executable Earley parsing", In Proceedings of the 10th International Conference on Compiler Construction (LNCS #2027). Springer-Verlag, 2001, pp. 229-243.

[Aycock99] J. Aycock, R. Horspool, "Faster Generalized LR Parsing", Proceedings of CC'99, 8th Intl. Conference on Compiler Construction, Amsterdam, March 1999. LNCS 1575, Springer-Verlag, pp. 32-46. Available at: http://www.csr.uvic.ca/~nigelh/Publications/cc99-paper.pdf

[Badros00] G. Badros, “JavaML: a markup language for java source code”, In WWW9, Ninth International World Wide Web Conference, Amsterdam, May 2000, Available at: http://www.cs.washington.edu/homes/gjb/papers/badros-javaml-www9.pdf

[Bapst99] F. Bapst and C. Vanoirbeek, “XML documents production for an electronic platform of requests for proposals”, presented at Reliable Distributed Systems, 1999. Proceedings of the 18th IEEE Symposium on, 1999.

[Bergstra89] J. A. Bergstra, J. Heering, P. Klint, “Algebraic Specification”, ACM Press, ISBN 0-201-41635-2, Addison-Wesley, 1989.

[Behrens00] R. A Behrens, “Grammar Based Model for XML Schema Integration”, Proceedings of British National Conference on Databases (BCNOD), 17., 2000.

[Bos99] B. Bos, “XML in 10 points”, vol. 2000: W3C, 1999, Available at: http://www.w3.org/XML/1999/XML-in-10-points.html

[Bosak 97] J. Bosak, “XML, Java, and the future of the web”, October 1997, Available at: http://sunsite.unc.edu/ pub/sun-info/standards/xml/why/xmlapps.html

[Bosak 98] J. Bosak, “Media-independent publishing: four myths about XML”, Computer, vol. 31, pp. 120-122, 1998, Available at: http://www.ibiblio.org/pub/sun-info/standards/xml/why/4myths.htm

[Boshernitsan00] M. Boshernitsan, S. L. Graham, “Designing an XML-Based Exchange Format for Harmonia”, Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00), 2000, Available at: http://www.cs.berkeley.edu/Research/Projects/harmonia/papers/harmonia-xml.pdf

[Brand01] M. van den Brand, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, J. Visser, “The ASF+SDF Meta-Environment: a component-based language development environment”, In Compiler Construction 2001 (CC 2001), LNCS. Springer, 2001, Available at: http://www.cwi.nl/~jvisser/papers/meta01.pdf

[Brand96] M. van den Brand, E. Visser, “Generation of Formatters for Context-Free languages”, ACM Transactions on Software Engineering and Methodology, Volume 5, Issue 1, January 1996.

[Brand98] M. van den Brand, A. Sellink, C. Verhoef, “Current Parsing Techniques in Software Renovation Considered Harmful”, In the proceedings of the International Workshop on Program Comprehension, Italy, 1998, Available at: http://carol.wins.uva.nl/~x/ref/

[Brueggemann-Klein98] A. Brueggemann-Klein, D. Wood, “One-Unambiguous Regular Languages”, Information and Computation, 140, pp. 229-253, 1998, A preliminary version of this paper available at: http://www.cs.ust.hk/~dwood/preprints/infocomp.ps.gz

[Buschmann96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, “Pattern-Oriented Software Architecture: A System Of Patterns”, John Wiley & Sons Ltd., 1996.

[Chomsky56] N. Chomsky, “Three Models for the Description of Language”, IRE Transactions on Information Theory, 2 (1956), pp. 113-124, 1956.

[Chomsky59] N. Chomsky, “On Certain Formal Properties of Grammars”, Information and Control, 1 (1956), pp. 137-167, 1959.

[Crew97] R. F. Crew, “ASTLOG: A language for examining abstract syntax trees”, In Proceedings of the USENIX Conference on Domain-Specific Languages, Santa Barbara, CA, October, 1997, Available at: http://www.usenix.org/publications/library/proceedings/dsl97/full_papers/crew/crew.pdf

[Cross01] D. Cross, “Data Munging with Perl”, Manning Publications, 2001.

[Comon99] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, “Tree Automata Techniques and Applications”, 1999, Available at: http://www.grappa.univ-lille3.fr/tata/

[Cooper98] J. W. Cooper, “The Design Patterns Java Companion”, Addison-Wesley, 1998, Available at: http://www.patterndepot.com/put/8/JavaPatterns.htm

[Deutsch00] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, “XML-QL: A Query Language for XML”, vol. 2000: W3C, 1998, Available at: http://www.w3.org/TR/NOTE-xml-ql/

[DOM] W3C Recommendation, “Document Object Model Level 1”, V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, L. Wood, 1 October 1998, Availale at: http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/

[DSD] N. Klarlund, A. M¢ller, and M. I. Schwartzbach, “Document Structure Description 1.0”, AT&T & BRICS, October 1999, BRICS NS-00-7, Available at: http://www.brics.dk/DSD/specification.html

[Earley70] J. Earley, “An efficient context-free parsing algorithm”, Communications of the ACM 13, 2 (Feb. 1970), 94-102.

[Ebert99] J. Ebert, B. Kullbach, A. Winter, "GraX - An Interchange Format for Reengineering Tools", Proceedings of the Sixth Working Conference on Reverse Engineering, 1999, Available at: http://www.gupro.de/winter/Papers/ebert+1999.pdf

[ebRS] OASIS/ebXML Registry Technical Committee, “OASIS/ebXML Registry Services Specification V2.0”, 6 Dec 2001, Available at: http://www.ebxml.org/specs/ebrs2.pdf

[ebXML] ebXML Technical Architechture Project Team, ”ebXML Technical Architechture Specification v1.0.4”, 16 Feb. 2001, Available at: http://www.ebxml.org/specs/ebTA.pdf

[EDIFACT] UN/EDIFACT Standard Directory, Available at: http://www.unece.org/trade/untdid/welcome.htm

[Ek01] M. Ek, H. Hakkarainen, P. Kilpeläinen, E. Kuikka, T. Penttinen, ”Describing XML Wrappers for Information Integration”, University of Kuopio, Department of Computer Science and Applied Mathematics, Technical Reports, Series A, Report A/2001/2, 2001.

[Emmerrich99] W. Emmerich, W. Schwarz, and A. Finkelstein, “Markup Meets Middleware”, In Proc. of the 7 th IEEE Workshop on Future Trends in Distributed Systems, Capetown, South Africa. IEEE Computer Society Press, 1999, Available at: http://www.cs.ucl.ac.uk/staff/W.Emmerich/publications/FTDCS99/m3.pdf

[Erlikh01] L. Erlikh, L. Goldbaum, “EAIs Missing Link: Lecacy Integration”, EAI Journal, Apprill 2001, pp. 12-16, Available at: http://www.eaijournal.com/PDF/EAI%20Missing%20Link.pdf

[Fankhauser93] P. Fankhauser, Y. Xu, ”MarkItUp! An incremental approach to document structure recognition”, Electronic Publishing, vol. 6(4), pp. 447-456, Dec. 1993, Available at: ftp://ftp.darmstadt.gmd.de/pub/oasys/reports/P-94-07.ps.Z

[Feather86] M S. Feather, “A Survey and Classification of some Program Transformation Approaches and Techniques”, Information Sciences Institute, University of Southern California, April 1986, presented at IFIP WG2.1 Working Conference on Program Specification and Transformation, Bad Toelz, Germany, April 1986.

[Gamma95] E. Gamma, et al., “Design Patterns. Elements of Reusable Software”, Addison-Wesley, 1995

[Gagnon98] E, Gagnon, “SableCC, an Object-Oriented Compiler Framework”, PhD thesis, School of Computer Science, McGill University, Montreal, March 1998, Available at: http://www.sablecc.org/thesis.pdf

[Gao99] X. Gao, L. Sterling, “AutoWrapper: automatic wrapper generation for multiple online services”, 1999, Available on the web at: http://www.cs.mu.oz.au/~xga/apweb99/index.htm

[GBASE] M. de Jonge, E. Visser, and J. Visser, “The Grammar Base”, Available at: http://www.program-transformation.org/gb/

[Glushko99] R. J. Glushko, J. M. Tenenbaum, and B. Meltzer, “An XML framework for agent-based E-commerce”, Communications of the ACM, vol. 42, pp. 106-114, 1999, Available at: http://www.commerce.net/research/technology-applications/1999/99_23_r.pdf

[Goguen77] J. A. Goguen, J.W. Thatcher, E. G. Wagner, J.B. Wright, “Initial algebra semantics and continuous algebras”, Journal of ACM, 24 (1), pp. 68-95, 1977.

[Goldfarb90] C. F. Goldfarb “The SGML Handbook”, Oxford University Press Inc., 1990.

[Graham76] S. L. Graham, M. A. Harrison, “Parsing of general context-free languages”. In Advances in Computing, Vol. 14, Academic Press, New York, p. 77-185, 1976.

[Graham80] S.L. Graham, M.A. Harrison, W.L. Ruzzo, “An improved context-free recognizer”, ACM Trans. Prog. Lang. Syst., vol. 2, no. 3, p. 415-462, July 1980.

[Grosh01] G. Grosh, “Data Imperatives: Patterns in EAI behaviour”, EAI Journal, Septemper 2001, Available at: http://www.eaijournal.com/PDF/DataImperatives.pdf

[Grune90] D. Grune, C. J. H. Jacobs, "Parsing Techniques: A Practical Guide", Ellis Horwood, 1990.

[Hammer97] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, A. Crespo, “Extracting Semistructured Information from the Web”, Proceedings of the Workshop on Management of SemiStructured Data in conjunction with the 1997 ACM, Available at: ftp://db.stanford.edu/pub/papers/extract.ps

[Hamming86] R. Hamming, "You and Your Research", Talk given at Bell Labs, March, 1986, Available at: http://www.cs.wisc.edu/~cs736-1/cs736.html/Papers/hamming.ps.gz

[Hasselbring00] W. Hasselbring, “Information System Integration”, Communications of the ACM 43 (6), pp. 33-38, June 2000, Available at: http://se.informatik.uni-oldenburg.de/publications/PDF/CACM-ISI2000.pdf

[Hawke01] S. Hawke, “Blindfold Grammars”, Available at: http://www.w3.org/2001/06/blindfold/grammar

[Heering89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. “The syntax definition formalism SDF — Reference manual”. SIGPLAN Notices, 24(11), pp. 43–75, 1989.

[Herman00] I. Herman and M. S. Marshall, “GraphXML – An XML based graph interchange format” Technical Report INS-R0009, CWI, 2000, Available at: http://www.cwi.nl/ftp/CWIreports/INS/INS-R0009.pdf

[Holman99] K. G. Holman, “The XML family of standards”, XML Finland'99: SGML Users Group Finland - Conference Paper, September 23, 1999.

[Holt00] R. Holt, A. Winter, A. Schürr, “GXL: Toward a standard exchange format”, Fachbericht Informatik 1/2000, Institutfür Informatik, Koblenz, Universitü at Koblenz-Landau, Mai 2000. Available at: http://www.gupro.de/techreports/RR-1-2000.

[Hopcroft79] J. Hopcroft, J. Ullman, “Introduction to Automata Theory, Languages, and Computation”, Addison-Wesley, Reading, MA, 1979.

[Hsu98] C. N. Hsu, “Initial Results on Wrapping Semistructured Web Pages with Finite-State Transducers and Contextual Rules”, AAAI'98 Workshop 'AI and Information Integration, Madison, July 1998, Available at: http://www.iis.sinica.edu.tw/~chunnan/DOWNLOADS/softmealy.ps.gz

[HTML] “HTML 4.0 Specification”, W3C Recommendation, D. Raggett, A. Le Hors, I. Jacobs. Available at: http://www.w3.org/TR/html4/

[Huck98] G. Huck, P. Frankhauser, K. Aberer, E. Neuhold, “Jedi: Extracting and Synthesizing Information from web”, IEEE Computer Society Press, New York, August, 1998, Available at: http://www.darmstadt.gmd.de/oasys/projects/jedi/jp.pdf

[Hudson97] S. E. Hudson, “CUP parser generator for Java”, 1997. Available at: http://www.cs.princeton.edu/appel/modern/java/CUP/

[InfoSet] “XML Information Set”, W3C Recommendation, J. Cowan, R. Tobin, 24 October 2001, Available at: http://www.w3.org/TR/xml-infoset/

[ISO8879] International Organization for Standardization, “Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)”, Ref. No. ISO/IEC 8879:1986(E), 1986.

[ISO14799] International Organization for Standardization, “Extended BNF”, Ref. No. ISO/IEC 14799:1996(E), 1996.

[Java] J. Gosling, B. Joy, G. Steele, G. Bracha, “The Java Language Specification (2nd edition)”, Addison-Wesley, 2000.

[JavaVM] Available at: http://java.sun.com/downloads/

[JAXB] Java Architecture for XML Binding (JAXB), Available at: http://java.sun.com/xml/jaxb/

[JAXP] Java API for XML Processing (JAXP), Available at: http://java.sun.com/xml/jaxp/

[Jokipii99] A. Jokipii, “Data Wrapping Technologies for HTML”, B.Sc Thesis, University of Jyväskylä, 1999.

[Jonge00a] M. de Jonge and J. Visser. “Grammars as contracts. In Generative and Component-based Software Engineering (GCSE)”, Erfurt, Germany, Oct. 2000, Available at: http://www.cwi.nl/~mdejonge/papers/GrammarsAsContracts.pdf

[Jonge00b] M. de Jonge, R. Monajemi, “Grammar re-engineering for language centered software engineering”, draft, oct 2000, Available at: http://www.cwi.nl/~mdejonge/papers/GrammarReengineeringforLCSE.pdf

[Johnson75] S. C. Johnson, “YACC - Yet Another Compiler-Compiler”, Technical Report Computer Science 32, Bell Laboratories, Murray Hill, New Jersey, 1975, Available at: http://epaperpress.com/lexandyacc/download/yacc.pdf

[Johnstone97a] A. Johnstone, E. Scott, “Generalized recursive descent part1: Language desing and parsing”, Technical Report TR-97-18, Royal Holloway, University of London, Computer Science Department, October 1997.

[Johnstone97b] A. Johnstone, E. Scott, “Generalized recursive descent part2: Some underlying theory”, Technical Report TR-97-19, Royal Holloway, University of London, Computer Science Department, December 1997.

[Karsai00] G. Karsai, "Design Tool Integration: An Exercise in Semantic Interoperability", Proceedings of the 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2000, Available at: http://www.isis.vanderbilt.edu/publications/archive/Karsai_G_3_0_2000_Design_Too.pdf

[Kienle01] H. Kienle, ”Exchange format bibliography”, ACM Software Engineering Notes, Volume 16, Number 1, pages 56-60, January 2001, Available at: http://cedar.csc.uvic.ca/twiki/kienle/pub/Main/MyPublications/Kienle:SEN:01.ps

[Kipps91] J. R. Kipps, ”GLR parsing in time $O(n^3)$”, In [Tomita91], chapter 4, pp. 43-59.

[Klarlund00] N. Klarlund, A. M¢ller, and M. I. Schwartzbach, “DSD: A schema language for XML”, In ACM SIGSOFT Workshop on Formal Methods in Software Practice (FMSP’00), 2000, Available at: http://www.brics.dk/DSD/

[Klein97] B. Klein, P. Fankhauser, “Error tolerant document structure analysis”, International Journal of Digital Libraries 1997, pp. 344-357, Available at: ftp://ftp.darmstadt.gmd.de/pub/oasys/reports/P-97-18.ps.Z

[Knuth68] D. E. Knuth, “Semantics of context-free languages”, Mathematical Systems Theory, 2 (2), pp. 127-145, Springer-Verlag, 1968.

[Koch99] T. Koch, “XML in practice: the groupware case”, presented at Enabling Technologies: Infrastructure for Collaborative Enterprises, 1999. (WET ICE '99)., 1999, Available at: http://bscw.gmd.de/Papers/wetice99/wetice99.pdf

[Koster71] C. H. A. Koster, ”Affix Grammars”, In J. E. L. Peck, Editor, “Algol-68 Implementation”, North-Holland, Amsterdam, 1971.

[Kotok00] A. Kotok, “Extensible and more: An Updated Survey of XML Business Vocabularies”, August 2000, Available at: http://www.xml.com/pub/2000/08/02/ebiz/extensible.html

[Lang74] B. Lang, "Deterministic techniques for efficient non-deterministic parsers", In Automata, Languages, and Programming (LNCS #14). Springer-Verlag, 1974, pp. 255-269.

[Lear99] A. C. Lear, “XML seen as integral to application integration”, IT Professional, vol. 1, pp. 12-16, 1999, Available at: http://www.soc.staffs.ac.uk/~cmryl/refpapers/00793665.pdf

[Lee00] S. Lee, “Template-based XML Data Integration System”, International Conference on Electronic Commerce, Seoul, Korea, 2000

[Leemakers91] R. Leemakers, “Non-deterministic recursive ascent parsing”, In Fifth Conference of the European Chapter of Assosiations for Computational Linguistics, Proceeding of the Conference, pp. 63-68, Berlin, Germany, April 1991.

[Lempinen01] E. Lempinen, H.Saarikoski, Editors, A. Jokipii, E. Ojanen, Submitters, ”Data Extraction Language DEL, submission”, W3C, 2001, Available at: http://www.w3.org/Submission/2001/10/

[Lempinen03] E. Lempinen, "Tekstitiedon muuntaminen XML-dokumenteiksi", M.Sc thesis, 15th October 2003, draft

[Lie99] H. W. Lie and J. Saarela , “Multipurpose Web publishing using HTML, XML, and CSS”, Communications of the ACM, vol. 42, pp. 95-101, 1999, Available at: http://www.w3.org/People/Janne/porject/paper.html

[Liu00] L. Liu, C. Pu, and W. Han, “XWRAP: An XML-enabled Wrapper Construction System for Web Information Sources”, presented at 16th International Conference on Data Engineering, 2000.

[Log4j] Available at: http://jakarta.apache.org/log4j/docs/download.html

[Lowry00] P. B. Lowry, “XML, an enabler to extend intra-company collaboration through e-commerce: benefits, issues, and implementation strategies”, University of Arizona, Tucson Working Paper, May 31 2000.

[Lowry01] P. B. Lowry, “XML data mediation and collaboration: a proposed comprehensive architecture and query requirements for using XML to mediate heterogeneous data sources and targets,” 34th Annual Hawaii International Conference On System Sciences (HICSS), Maui, Hawaii, 2001, Available at: www.cmi.arizona.edu/personal/plowry/ papers\2001 HICSS xml.pdf

[Lu00] J. Lu, J. Mylopoulos, J. Ho. ”Towards extensible information brokers based on xml”, In To appear in 12th Conference on Advanced Information Systems Engineering, 2000, Available at: http://www.cs.toronto.edu/~jglu/pub/caiseLNCS.pdf

[Lublinsky02] B. Lublinsky, “Approaches to B2B integration”, EAI Journal, February 2002, pp. 38-47, Available at: http://www.eaijournal.com/PDF/B2BApproachesLublinksy.pdf

[Lämmel01a] R. Lämmel, ”Grammar Adaption”, Proc. Formal Methods Europe (FME) 2001, LNCS(2021): 550-570, Springer-Verlag, 2001, Available at: http://www.cwi.nl/~ralf/fme01/

[Lämmel01b] R. Lämmel, C. Verhoef, “Semi-automatic Grammar Recovery”, Software: Practice and Experience, vol. 31, no. 15, December 2001, pp. 1395-1438, Available at: http://www.cs.vu.nl/~x/ge/ge.pdf

[Maletic02] J. Maletic, M. Collard, A. Marcus, ”Source code files as structured documents”, In Tenth International Workshop on Program Comprehension, Paris, France, June 2002. To appear. Available at: http://trident.mcs.kent.edu/~jmaletic/papers/iwpc02.pdf

[Mamas00] E. Mamas, C. Kontogiannis, “Towards Portable Source Code Representations using XML”, in Proceedings of 7th Working Conference on Reverse Engineering (WCRE '00), Brisbane, Queensland, Australia, pp. 172-182, November 2000, Available at: http://www.swen.uwaterloo.ca/~evan/Papers/wcre2000.pdf

[Martin91] J. C. Martin, “Introduction to Languages and the Theory of Computation”, McGraw-Hill, New York, 1991.

[McCarthy65] J. McCarthy, “A Basis for a Mathematical Theory of Computation”, In Computer Programming and Formal Systems, edited by P. Braffort and D. Hirschberg, North-Holland, Amsterdam, 1965, pp. 33-70.

[Mclean98] Thom McLean, Leo Mark, David Rosenbaum, Jack Sheehan, Mike Hopkins: “Self-Defining Data Interchange Formats: A Mechanism for Preserving Interface Investment”, presented at the Fall Simulation Interoperability Workshop, Orlando, FL, 1998.

[Meek90] B. Meek, "The Static Semantic File", ACM SIGPLAN Notices, 25(4), 1990, pp. 33-42, Available at: http://www.kcl.ac.uk/kis/support/cit/staff/brian/statsem.html

[Meyer90] B. Meyer, "Introduction to the Theory of Programming Languages", Prentice Hall, Hemel Hempstead, UK, 1990.

[Moonen01] L. Moonen, ”Generating robust parsers using island grammars”, In Proceedings of the 8th Working Conference on Reverse Engineering, pages 13-22. IEEE Computer Society Press, October 2001, Available at: http://www.cwi.nl/~leon/papers/wcrc2001/wcrx2001.pdf

[Moonen02] L. Moonen, ”Lightweight impact analysis using island grammars”, In Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002). IEEE Computer Society Press, June 2002, Available at: http://www.cwi.nl/~leon/papers/iwpc2002/iwpc2002.pdf

[Murata97] M. Murata, ”Transformation of documents and schemas by patterns and contextual conditions”, In Principles of Document Processing '96, volume 1293 of Lecture Notes in Computer Science, pages 153-169. Springer-Verlag, 1997.

[Muslea99] I. Muslea, S. Minton, C. A. Knoblock, “Hierarchical Approach to Wrapper Induction”, 3rd conference on Autonomous Agents, 1999, Available at: http://www.isi.edu/~muslea/PS/hwi_aa99.ps

[Nakhimovsky01] A. Nakhimovsky, “Parser generators for legacy data integration”, Paper at the annual XML-Europe Conference, Berlin, May 21-25, 2001, Available at: http://www.gca.org/papers/xmleurope2001/papers/html/s07-3.html

[Namespaces] W3C Recommendation, "Namespaces in XML", Editors: T. Bray, D. Hollander, A. Layman, 14 January 1999. Available at: http://www.w3.org/TR/REC-xml-names

[Naur63] P. Naur, editor, “Revised Report on the Algorithmic Language Algol 60”, Communications of the ACM, 6.1, January 1963, pp. 1-20, Available at: http://burks.brighton.ac.uk/burks/language/other/a60rr/report.htm

[Nederhof92] M. J. Nederhof, “Generalized Left-Corner Parsing”, Technical report no.92-21, Unversity of Nijimegen, Department of Computer Science, August 1992.

[Nozohoor-Farsi91] R. Nozohoor-Farsi, “GLR parsing for ε-grammars”, In [Tomita, 1991], chapter 5, pp. 61-75.

[Ojanen01] E. Ojanen, "Method and Apparatus for Regrouping Data", US Patent Application #20020129005, 2001.

[Parr95] T. J. Parr and R. W. Quong, “ANTLR: A predicated-LL(k) parser generator”, Software –Practice and Experience, 25(7):789–810, July 1995, Available at: http://www.antlr.org/papers/antlr.ps

[Parr97] T. J. Parr, “Language Translation Using PCCTS & C++”, Automata Publishing Company, 1997. ISBN: 0962748854.

[Parsons92] T. W. Parsons, "Introduction to Compiler Construction", Computer Science Press, New York, 1992.

[Pepper99] P. Pepper, “LR Parsing = Grammar Transformation + LL Parsing”, Technical Report CS-99-05, TU Berlin, Apr. 1999, Available at: http://www.cs.tu-berlin.de/cs/ifb/TeBericht/99/tr99-5.ps

[Pereira80] F. C. N. Pereira, D. H. D. Warren, ”Definite Clause Grammars for language analysis - a survey of the formalism and comparison with augmented transition networks”, Artifical Inteligence, 13, pp. 231-278, 1980.

[Performer] Available at: http://www.x-fetch.com/performer.html

[Petrou99] C. Petrou, S. Hadjiefthymiades, D. Martakos, “An XML-based, 3-tier scheme for integrating heterogeneous information sources to the WWW”, presented at Tenth International Workshop on Database and Expert Systems Applications, 1999.

[Pinkston01] J. Pinkston, “The Ins and Outs of Integration – How EAI differs from B2B Integration”, EAI Journal, August 2001, pp. 48-52, Available at: http://www.eaijournal.com/PDF/Ins&OutsPinkston.pdf

[Ragget99] D. Raggett, “Assertion grammars”, May 1999, Available at: http://www.w3.org/People/Raggett/dtdgen/Docs/

[Reinold99] M. Reinold, “An XML Data-Binding Facility for the Java Platform”, Palo Alto, 1999. Available at: http://java.sun.com/xml/white-papers.html.

[Rekers92] J. Rekers, “Parser Generation for Interactive Environments”, Ph.D. thesis, University of Amsterdam, 1992, Available at: ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z

[RFC1630] T. Berners-Lee, “Universal Resource Identifiers in WWW”, RFC 1630, June 1994. Available at: http://www.ietf.org/rfc/rfc1630.txt

[RFC2119] S. Bradner, Editor, “Key words for use in RFCs to Indicate Requirement Levels”, RFC 2119, March 1997. Available at: http://www.ietf.org/rfc/rfc2119.txt

[RFC2234] D. Crocker, Ed., “Augmented BNF for syntax specifications: ABNF”, RFC 2234, November 1997. Available at: http://www.ietf.org/rfc/rfc2234.txt

[Salonen98] J. Salonen, E. Ojanen, M. Paananen, “XML & Structurally and Semantically Advanced Knowledge Retrieval and Distribution”, SGML/XML Finland '98, Conference Proceedings, 1998

[Salonen99] J. Salonen, E. Ojanen, M. Paananen, T. Eskelinen, J. Lemmetty and A. Jokipii, “XML and Information Brokering; How E-services will Change Business Communication”, XML Finland '99 Conference Proceedings, 1999

[Sanborn00] S. Sanborn, “Portal proliferation benefits e-business”, InfoWorld, vol. 22, pp. 30, 2000, Available at: http://www.infoworld.com/articles/hn/xml/00/04/24/000424hnenabler.xml

[Sahuguet98] A. Sahuguet, F. Azavant, “W4F: the WysiWyg Web Wrapper Factory”, Technical report, University of Pennsylvania, Department of Computer and Information Science, 1998, Available at: http://db.cis.upenn.edu/~sahuguet/WAPI/wapi/

[Saxon] Available at: http://saxon.sourceforge.net/

[Seymore99] K. Seymore, A. McCallum, R. Rosenfeld, “Learningh hidden Markov model structure for information extraction”, AAAI'99 Workshop on Machine Learning for Information Extraction, 1999, Available at: http://www.cs.cmu.edu/~mccallum/papers/iestruct-aaaiws99.ps.gz

[Sellink00] M. Sellink, C. Verhoef, ”Development, assessment, and reengineering of language descriptions”, In J. Ebert, C. Verhoef, Editors, Proceeding of the Fourth European Conference of Software Maintenance and Reengineering, pages 151-160, IEEE Computer Society, March 2000, Available at: http://adam.wins.uva.nl/~x/ase98/ase98.html

[Sippu88] S. Sippu, E. Soisalon-Soininen, ”Parsing Theory: Languages and Parsing. Volume 1”, EATCS Monographs on Theoretical Computer Science 15, Springer-Verlag, 1988

[Slonger95] K. Slonneger, B. L. Kurtz, “Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach”, Addison-Wesley, 1995, Available at: http://www.cs.uiowa.edu/~slonnegr/plf/Book/

[Spinellis01] D. Spinellis, “Notable design patterns for domain specific languages”, Journal of Systems and Software, 56(1):91-99, February 2001, Available at: http://softlab.icsd.aegean.gr/~dspin/pubs/jrnl/2000-JSS-DSLPatterns/html/dslpat.html

[Thomas99] B. Thomas, “Learning T-Wrappers”, Information Extraction Workshop on machine learning on human language technology, Advanced Course on Artificial Intelligence 1999 (ACAI'99), pp. 5-16, July 1999, Available at: http://www.informatik.uni-koblenz.de/~bthomas/PAPERS/ACAI-99/full-paper.ps.gz

[Tomita86] M. Tomita, “Efficient Parsing for Natural Languages”, Kluwer Academic Publisher, 1986.

[Tomita91] M. Tomita, editor. “Generalized LR parsing”, Kluwer Academic Publisher, 1991.

[Unger68] S.H. Unger, "A global parser for context-free phrase structure grammars", Commun. ACM, vol. 11, no. 4, p. 240-247, April 1968.

[Unicode]. The Unicode Consortium, “The Unicode Standard, Version 4.0”, Reading, MA, Addison-Wesley, 2003.

[Usdin98] T. Usdin and T. Graham, “XML: not a silver bullet, but a great pipe wrench”, Standard View, vol. 6, pp. 125-132, 1998, Available at: http://www.cs.uiowa.edu/~rlawrenc/teaching/296/Notes/XML_not_silver_bullet.pdf

[Visser97a] E. Visser, “Syntax Definition for Language Prototyping”, PhD thesis, University of Amsterdam, 1997, Available at: http://www.cs.uu.nl/people/visser/ftp/Vis97.ps.zip

[Visser97b] E. Visser, “Character classes”, Technical Report P9708, Programming Research Group, University of Amsterdam, August 1997, Available at: http://www.cs.uu.nl/people/visser/ftp/P9708.ps.zip

[Visser97c] E. Visser, “A family of syntax definition formalisms”, Technical Report P9706, Programming Research Group, University of Amsterdam, July 1997, Available at: http://www.cs.uu.nl/people/visser/ftp/P9706.ps.zip

[Visser97d] E. Visser. “From context-free grammars with priorities to character class grammars”, In Liber Amicorum Paul Klint. CWI, Amsterdam, November 1997, Available at: http://www.cs.uu.nl/people/visser/ftp/P9717.ps.zip

[Visser00] J. Visser, J. Scheerder, “A quick introduction to SDF”, Draft, 2000, Available at: http://www.cwi.nl/~jvisser/papers/sdfintro.pdf

[Watt77] D. A. Watt, “The Parsing Problems of Affix Grammars”, Acta Informatica, 8 (1), 1-20, 1977.

[Xalan] Available at: http://xml.apache.org/xalan-j/downloads.html

[XML] W3C Recommendation , “Extensible Markup Language (XML) 1.0 Specification”, T. Bray, J. Paoli, C. M. Sperberg-McQueen, 10 February 1998. Available at: http://www.w3.org/TR/REC-xml

[XMLSchema] W3C Recommendation , “XML Schema Part 0: Primer”, D. C. Fallside, editor, 2 May 2001, Available at: http://www.w3.org/TR/xmlschema-0/

[XMLse] W3C Recommendation , “Extensible Markup Language (XML) 1.0 (Second Edition)”, T. Bray, J. Paoli, C. M. Sperberg-McQueen, October 2000, Available at: http://www.w3.org/TR/REC-xml

[XPath] W3C Recommendation , “XML Path Language”, J. Clark, S. DeRose, editors, 16 November 1999, Available at: http://www.w3.org/TR/xpath

[XSL] W3C Recommendation , “Web Style Sheets”, May 2000, Available at: http://www.w3.org/Style/XSL/

[XSLT] W3C Recommendation, “XSL Transformations (XSLT)”, J. Clark, editor, 16 November 1999, Available at: http://www.w3.org/TR/xslt

[Walsh97] N. Walsh, “A Guide to XML”, World Wide Web Journal, vol. 2, issue 4, fall 1997, O'Reilly & Associates, 1997, revised and XML 1.0 up-to-date edition is available at: http://www.xml.com/pub/98/10/guide0.html.

[Watt91] D. Watt, “Programming Language Syntax and Semantics”, Prentice Hall International, Hemel Hempstead, UK, 1991.

[Widergren99] S. Widergren, A. deVos, J. Zhu, “XML for data exchange”, presented at Power Engineering Society Summer Meeting, 1999, Available at: http://ce.sejong.ac.kr/~dshin/Papers/XML/00787426.pdf

[Wile97] D. S. Wile. “Abstract syntax from concrete syntax”. In Proceedings of the 19th International Conference on Software Engineering (ICSE ’97), pages 472–480, Berlin - Heidelberg – New York, May 1997, Available at: http://mr.teknowledge.com/wile/Popart/ConcreteToAbstract.pdf

[Wirth77] N. Wirth, “What can we do about the unnecessary diversity of notation for syntactic definitions?”, Communications of the ACM, 20(11), 822-823, 1977.

[Worder00] R. Worden, “XML e-business standards: Promises and pitfalls”, vol. 2000: XML.com, 2000, Available at: http://www.xml.com/pub/a/2000/01/ebusiness/

[Wijegunaratne00] I. Wijegunaratne, G. Fernandez, J. Valtoudis, “A federated architecture for enterprise data integration”, Software Engineering Conference, 2000. Proceedings. 2000 Australian , pp. 159 –167, 2000, Available at:

Appendices

Appendix 1. DTD for BNF

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is the DTD for the BNF grammars. -->

<!-- ================================================= -->
<!-- ENTITIES -->
<!-- ================================================= -->
<!-- Member elements for Rules and Alternatives -->
<!ENTITY % members "(Nonterminal|Terminal)+">

<!-- ================================================= -->
<!-- GRAMMAR ELEMENTS -->
<!-- ================================================= -->

<!-- this is the top-level element -->
<!ELEMENT Grammar (Rule, Alternative*)+>

<!-- this is definition for nonterminal rules -->
<!ELEMENT Rule %members;>
<!ATTLIST Rule
          nonterminal CDATA #REQUIRED
>

<!—
This is definition for alternative rules to preceeding sibling rule.
 -->
<!ELEMENT Alternative %members;>

<!ELEMENT Nonterminal EMPTY>
<!ATTLIST Nonterminal
          name CDATA #REQUIRED
>

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

Appendix 2. XSLT stylesheet to generate reverse XSLT

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

<!--
  This XSLT stylesheet generates reverse XSLT stylesheet for the given GDEL
specification.
  Author: Antti Jokipii, Republica Corp. (2002)
 -->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

   <xsl:output method="xml" indent="yes"/>
   <xsl:strip-space elements="*"/>

   <xsl:template match="Grammar">
      <xsl:element name="xsl:stylesheet">
         <xsl:attribute name="version">1.0</xsl:attribute>
         <xsl:element name="xsl:output">
            <xsl:attribute name="method">text</xsl:attribute>
         </xsl:element>
         <xsl:element name="xsl:strip-space">
            <xsl:attribute name="elements">*</xsl:attribute>
         </xsl:element>
         <xsl:apply-templates/>
      </xsl:element>
   </xsl:template>
```

```
   <xsl:template match="Rule">
      <xsl:element name="xsl:template">
         <xsl:attribute name="match">
            <xsl:value-of select="@id"/>
         </xsl:attribute>
         <xsl:if test=".//*[self::Char or self::String]">
           <xsl:apply-templates/>
         </xsl:if>
         <xsl:if test="not(.//*[self::Char or self::String])">
            <xsl:element name="xsl:apply-templates"/>
         </xsl:if>
      </xsl:element>
   </xsl:template>


   <xsl:template match="Nonterminal">
      <xsl:element name="xsl:apply-templates">
         <xsl:attribute name="select">
            <xsl:text>*[</xsl:text>
            <xsl:for-each select="/.//Rule[@nonterminal = current()/@name]">
               <xsl:text>self::</xsl:text>
               <xsl:value-of select="@id"/>
               <xsl:if test="position() != last()">
                  <xsl:text> or </xsl:text>
               </xsl:if>
            </xsl:for-each>
            <xsl:text>]</xsl:text>
            <xsl:if test="count(../Nonterminal[@name = current()/@name]) > 1">
               <xsl:text>[</xsl:text>
               <xsl:value-of select="count(preceding-sibling::Nonterminal[@name =
current()/@name])+1"/>
               <xsl:text>]</xsl:text>
            </xsl:if>
         </xsl:attribute>
         <xsl:apply-templates/>
      </xsl:element>
   </xsl:template>


   <xsl:template match="OneOrMore/Nonterminal | ZeroOrMore/Nonterminal">
      <xsl:element name="xsl:apply-templates">
         <xsl:attribute name="select">
            <xsl:for-each select="/.//Rule[@nonterminal = current()/@name]">
               <xsl:text>self::</xsl:text>
               <xsl:value-of select="@id"/>
               <xsl:if test="position() != last()">
                  <xsl:text> | </xsl:text>
               </xsl:if>
            </xsl:for-each>
         </xsl:attribute>
         <xsl:apply-templates/>
      </xsl:element>
   </xsl:template>


   <xsl:template match="Char">
      <xsl:element name="xsl:text">
         <xsl:if test="string-length(@value) = 1">
            <xsl:value-of select="@value"/>
         </xsl:if>
         <xsl:if test="string-length(@value) > 1">
            <xsl:text disable-output-escaping="yes">&amp;#x</xsl:text>
            <xsl:value-of select="@value"/>
            <xsl:text>;</xsl:text>
         </xsl:if>
      </xsl:element>
   </xsl:template>
```

```
   <xsl:template match="String">
      <xsl:element name="xsl:text">
         <xsl:value-of select="@value"/>
      </xsl:element>
   </xsl:template>


   <xsl:template match="CharClass">
      <xsl:element name="xsl:value-of">
         <xsl:attribute name="select">
            <xsl:text>substring(text(),</xsl:text>
            <xsl:if test="not(parent::OneOrMore | parent::ZeroOrMore)">
               <xsl:value-of select="count(preceding-sibling::*[self::CharClass])+1"/>
               <xsl:text>,</xsl:text>
               <xsl:text>1</xsl:text>
            </xsl:if>
            <xsl:if test="parent::OneOrMore | parent::ZeroOrMore">
               <xsl:value-of select="count(../preceding-
sibling::*[self::CharClass])+1"/>
               <xsl:text>,string-length(text()) - </xsl:text>
               <xsl:value-of select="count(../preceding-sibling::*[self::CharClass])"/>
            </xsl:if>
            <xsl:text>)</xsl:text>
         </xsl:attribute>
      </xsl:element>
   </xsl:template>


   <xsl:template match="OneOrMore[Nonterminal] | ZeroOrMore[Nonterminal] |
Optional[Nonterminal]">
      <xsl:element name="xsl:for-each">
         <xsl:attribute name="select">
            <xsl:text>*[</xsl:text>
            <xsl:for-each select="/.//Rule[@nonterminal = current()/Nonterminal/@name]">
               <xsl:text>self::</xsl:text>
               <xsl:value-of select="@id"/>
               <xsl:if test="position() != last()">
                  <xsl:text> or </xsl:text>
               </xsl:if>
            </xsl:for-each>
            <xsl:text>]</xsl:text>
         </xsl:attribute>
         <xsl:apply-templates/>
      </xsl:element>
   </xsl:template>


   <xsl:template match="OneOrMore | ZeroOrMore | Optional">
      <xsl:apply-templates/>
   </xsl:template>


</xsl:stylesheet>
```

Appendix 3. XSLT stylesheet to generate XML Schema

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

<!--
  This XSLT stylesheet generates XML Schema for the given GDEL specification.
  Author: Antti Jokipii, Republica Corp. (2003)
 -->
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <xsl:output method="xml" indent="yes"/>
   <xsl:strip-space elements="*"/>
```

```
   <xsl:template match="Grammar">
      <xsl:element name="xsd:schema">
         <xsl:for-each select="//Rule[not(@nonterminal=preceding-
sibling::Rule/@nonterminal)]">
            <xsl:element name="xsd:group">
               <xsl:attribute name="name">
                  <xsl:value-of select="@nonterminal"/>
               </xsl:attribute>
               <xsl:element name="xsd:choice">
                  <xsl:for-each select="//Rule[@nonterminal=current()/@nonterminal]">
                     <xsl:element name="xsd:element">
                        <xsl:attribute name="name">
                           <xsl:value-of select="@id"/>
                        </xsl:attribute>
                     </xsl:element>
                  </xsl:for-each>
               </xsl:element>
            </xsl:element>
         </xsl:for-each>
         <xsl:apply-templates/>
      </xsl:element>
   </xsl:template>


   <xsl:template match="Rule">
      <xsl:element name="xsd:element">
         <xsl:attribute name="name">
            <xsl:value-of select="@id"/>
         </xsl:attribute>
         <xsl:if test=".//Nonterminal">
            <xsl:element name="xsd:complexType">
               <xsl:element name="xsd:sequence">
                  <xsl:apply-templates/>
               </xsl:element>
            </xsl:element>
         </xsl:if>
         <xsl:if test="not(.//Nonterminal)">
            <xsl:element name="xsd:simpleType">
               <xsl:element name="xsd:restriction">
                  <xsl:attribute name="base">xsd:string</xsl:attribute>
               </xsl:element>
            </xsl:element>
         </xsl:if>
      </xsl:element>
   </xsl:template>


   <xsl:template match="Nonterminal">
      <xsl:element name="xsd:group">
         <xsl:attribute name="ref">
            <xsl:value-of select="@name"/>
         </xsl:attribute>
      </xsl:element>
   </xsl:template>


   <xsl:template match="OneOrMore">
      <xsl:element name="xsd:sequence">
         <xsl:attribute name="minOccurs">1</xsl:attribute>
         <xsl:attribute name="maxOccurs">unbounded</xsl:attribute>
         <xsl:apply-templates/>
      </xsl:element>
   </xsl:template>


   <xsl:template match="ZeroOrMore">
      <xsl:element name="xsd:sequence">
         <xsl:attribute name="minOccurs">0</xsl:attribute>
         <xsl:attribute name="maxOccurs">unbounded</xsl:attribute>
         <xsl:apply-templates/>
```

```
      </xsl:element>
   </xsl:template>


   <xsl:template match="Optional">
      <xsl:element name="xsd:sequence">
         <xsl:attribute name="minOccurs">0</xsl:attribute>
         <xsl:attribute name="maxOccurs">1</xsl:attribute>
         <xsl:apply-templates/>
      </xsl:element>
   </xsl:template>


   <xsl:template match="String">
      <xsl:element name="xsd:element">
         <xsl:attribute name="name">
            <xsl:value-of select="@value"/>
         </xsl:attribute>
         <xsl:attribute name="type">xsd:string</xsl:attribute>
      </xsl:element>
   </xsl:template>


</xsl:stylesheet>
```

Appendix 4. XML Schema for GDEL

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- Antti Jokipii (Republica Corp) 2003 -->
<xs:schema targetNamespace="http://www.x-fetch.com/GDEL"
xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:gdl="http://www.x-fetch.com/GDEL"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="Gdel">
    <xs:annotation>
      <xs:documentation>The root element of GDEL specification.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="gdl:Annotation"/>
        <xs:element ref="gdl:Property"/>
        <xs:element ref="gdl:Grammar"/>
      </xs:choice>
      <xs:attribute name="version" use="optional" fixed="1.0">
        <xs:annotation>
          <xs:documentation>GDEL language version. The GDEL processor verifies the
compatibility from this attribute. (In version 1.0 of GDEL Language Specification, for
example, this attribute must have value 1.0.)</xs:documentation>
        </xs:annotation>
        <xs:simpleType>
          <xs:restriction base="xs:string"/>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="Property">
    <xs:annotation>
      <xs:documentation>The property element defines a value of the named
property.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="name" type="gdl:PropertyNames" use="required">
        <xs:annotation>
          <xs:documentation>Name of property.</xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="value" type="xs:string" use="required">
        <xs:annotation>
          <xs:documentation>Value of property</xs:documentation>
        </xs:annotation>
```

```
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="FollowRestriction">
    <xs:annotation>
      <xs:documentation>Defines the follow restriction for the rule. Follow restrictions
are used as "a prefer longest match" to lexical tokens.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="gdl:CharClass"/>
      </xs:sequence>
      <xs:attribute name="rule" type="xs:IDREF" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="Grammar">
    <xs:annotation>
      <xs:documentation>Defines the grammar.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="Params">
          <xs:annotation>
            <xs:documentation>Define needed parameters.</xs:documentation>
          </xs:annotation>
          <xs:complexType>
            <xs:sequence maxOccurs="unbounded">
              <xs:element ref="gdl:Nonterminal"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Import">
          <xs:annotation>
            <xs:documentation>Import selected module to this modele.</xs:documentation>
          </xs:annotation>
          <xs:complexType>
            <xs:sequence>
              <xs:element name="WithParams" minOccurs="0">
                <xs:annotation>
                  <xs:documentation>Import module with these
parameters.</xs:documentation>
                </xs:annotation>
                <xs:complexType>
                  <xs:sequence maxOccurs="unbounded">
                    <xs:element ref="gdl:Nonterminal"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:sequence minOccurs="0" maxOccurs="unbounded">
                <xs:element name="Rename">
                  <xs:annotation>
                    <xs:documentation>To avoid name clashes terms of imported mmodule
could been renamed. This element defines renaming for imported
module.</xs:documentation>
                  </xs:annotation>
                  <xs:complexType>
                    <xs:attribute name="sort" type="xs:string" use="required">
                      <xs:annotation>
                        <xs:documentation>Name of imported nonterminal to be
renamed.</xs:documentation>
                      </xs:annotation>
                    </xs:attribute>
                    <xs:attribute name="to" type="xs:string" use="required">
                      <xs:annotation>
                        <xs:documentation>New name.</xs:documentation>
                      </xs:annotation>
                    </xs:attribute>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
```

```
            </xs:sequence>
            <xs:attribute name="Module" type="xs:anyURI" use="required">
              <xs:annotation>
                <xs:documentation>An import of module M into this module means that the
syntax of M is included in the syntax of this module.</xs:documentation>
              </xs:annotation>
            </xs:attribute>
          </xs:complexType>
        </xs:element>
        <xs:element name="Alias">
          <xs:annotation>
            <xs:documentation>Defines alias.</xs:documentation>
          </xs:annotation>
          <xs:complexType>
            <xs:complexContent>
              <xs:extension base="gdl:RuleType">
                <xs:attribute name="name" type="xs:ID" use="required">
                  <xs:annotation>
                    <xs:documentation>Name of alias.</xs:documentation>
                  </xs:annotation>
                </xs:attribute>
              </xs:extension>
            </xs:complexContent>
          </xs:complexType>
        </xs:element>
        <xs:element ref="gdl:LexicalSyntax"/>
        <xs:element ref="gdl:FollowRestriction"/>
        <xs:element ref="gdl:ContextFreeSyntax"/>
        <xs:element ref="gdl:Priorities"/>
        <xs:element ref="gdl:Rule"/>
        <xs:element ref="gdl:Annotation"/>
        <xs:element ref="gdl:Layout"/>
      </xs:choice>
      <xs:attribute name="module" type="xs:string" use="optional">
        <xs:annotation>
          <xs:documentation>Name of module.</xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="LexicalSyntax">
    <xs:annotation>
      <xs:documentation>Defines lexical tokens.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element ref="gdl:Rule"/>
        <xs:element ref="gdl:FollowRestriction"/>
        <xs:element ref="gdl:Annotation"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name="ContextFreeSyntax">
    <xs:annotation>
      <xs:documentation>Defines the context-free productions.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element ref="gdl:Rule"/>
        <xs:element ref="gdl:FollowRestriction"/>
        <xs:element ref="gdl:Annotation"/>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name="Range">
    <xs:annotation>
      <xs:documentation>The range of characters in the character
class.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
```

```
      <xs:attribute name="start" type="gdl:CharValue" use="required">
        <xs:annotation>
          <xs:documentation>Start character of range.</xs:documentation>
        </xs:annotation>
      </xs:attribute>
      <xs:attribute name="end" type="gdl:CharValue" use="required">
        <xs:annotation>
          <xs:documentation>End character of range.</xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="Rule">
    <xs:annotation>
      <xs:documentation>Defines the rule for the nonterminal
production.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="gdl:RuleType">
          <xs:attribute name="nonterminal" type="xs:string" use="required">
            <xs:annotation>
              <xs:documentation>Name of nonterminal of rule.</xs:documentation>
            </xs:annotation>
          </xs:attribute>
          <xs:attribute name="id" type="xs:ID" use="required">
            <xs:annotation>
              <xs:documentation>Identifier of rule. Identifier is used as element name
in output XML.</xs:documentation>
            </xs:annotation>
          </xs:attribute>
          <xs:attribute name="assoc" type="gdl:AssocValue" use="optional">
            <xs:annotation>
              <xs:documentation>Type of associativity relation of
rule.</xs:documentation>
            </xs:annotation>
          </xs:attribute>
          <xs:attribute name="reject" type="xs:boolean" use="optional" default="false">
            <xs:annotation>
              <xs:documentation>Defines reject production.</xs:documentation>
            </xs:annotation>
          </xs:attribute>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="Layout" type="gdl:RuleType">
    <xs:annotation>
      <xs:documentation>The content of this element is the layout and does not appear in
output XML.</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="Nonterminal">
    <xs:annotation>
      <xs:documentation>Refers to the named nonterminal.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="name" type="xs:string" use="required">
        <xs:annotation>
          <xs:documentation>Name of nonterminal referred</xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="Char">
    <xs:annotation>
      <xs:documentation>Terminal character.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="value" type="gdl:CharValue" use="required">
```

```
        <xs:annotation>
          <xs:documentation>Character or hexadecimal value of
character.</xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="CharClass">
    <xs:annotation>
      <xs:documentation>A set of terminal characters. The content of this element is
handled as union.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element ref="gdl:Range"/>
        <xs:element ref="gdl:Char"/>
        <xs:element name="Difference">
          <xs:annotation>
            <xs:documentation>Difference of two character clases.</xs:documentation>
          </xs:annotation>
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="gdl:CharClass"/>
              <xs:element ref="gdl:CharClass"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Intersection">
          <xs:annotation>
            <xs:documentation>Intersection of two character classes.</xs:documentation>
          </xs:annotation>
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="gdl:CharClass"/>
              <xs:element ref="gdl:CharClass"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element ref="gdl:CharClass"/>
        <xs:element ref="gdl:Alias"/>
        <xs:element ref="gdl:Annotation"/>
      </xs:choice>
      <xs:attribute name="negated" type="xs:boolean" use="optional" default="false">
        <xs:annotation>
          <xs:documentation>Is this character class negated.</xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="Optional" type="gdl:RuleType">
    <xs:annotation>
      <xs:documentation>The element content is optional.</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="OneOrMore" type="gdl:RuleType">
    <xs:annotation>
      <xs:documentation>The element content occurs once or several
times.</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="ZeroOrMore" type="gdl:RuleType">
    <xs:annotation>
      <xs:documentation>The element content occurs zero or more
times.</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="String">
    <xs:annotation>
      <xs:documentation>A string i.e. sequence of characters.</xs:documentation>
    </xs:annotation>
```

```
    <xs:complexType>
      <xs:attribute name="value" type="xs:string" use="required">
        <xs:annotation>
          <xs:documentation>Actual string value.</xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="Alternatives" type="gdl:RuleType">
    <xs:annotation>
      <xs:documentation>One of content elements.</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="Sequence" type="gdl:RuleType">
    <xs:annotation>
      <xs:documentation>The explicit sequence of content elements.</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="Alias">
    <xs:annotation>
      <xs:documentation>An alias reference.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:attribute name="ref" type="xs:IDREF" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="List">
    <xs:annotation>
      <xs:documentation>The list of items separated with the
separator.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Separator" type="gdl:RuleType">
          <xs:annotation>
            <xs:documentation>Definition of list separator.</xs:documentation>
          </xs:annotation>
        </xs:element>
        <xs:element name="Item" type="gdl:RuleType">
          <xs:annotation>
            <xs:documentation>Definition of Items of list.</xs:documentation>
          </xs:annotation>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Permutation" type="gdl:RuleType">
    <xs:annotation>
      <xs:documentation>Content elements occur exactly once in any
order.</xs:documentation>
    </xs:annotation>
  </xs:element>
  <xs:element name="Priorities">
    <xs:annotation>
      <xs:documentation>Defines the priority chain of rules.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element ref="gdl:Annotation"/>
        <xs:element ref="gdl:Group"/>
        <xs:element name="Chain">
          <xs:annotation>
            <xs:documentation>Defines the chain of priority rules.</xs:documentation>
          </xs:annotation>
          <xs:complexType>
            <xs:choice maxOccurs="unbounded">
              <xs:element name="Rule">
                <xs:annotation>
                  <xs:documentation>Reference to rule.</xs:documentation>
                </xs:annotation>
```

```
                <xs:complexType>
                  <xs:attribute name="ref" type="xs:IDREF" use="required">
                    <xs:annotation>
                      <xs:documentation>Reference to specific rule.</xs:documentation>
                    </xs:annotation>
                  </xs:attribute>
                </xs:complexType>
              </xs:element>
              <xs:element ref="gdl:Group"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
        <xs:element name="Priority">
          <xs:annotation>
            <xs:documentation>Declares single priority relation.</xs:documentation>
          </xs:annotation>
          <xs:complexType>
            <xs:attribute name="rule" type="xs:IDREF" use="required">
              <xs:annotation>
                <xs:documentation>Rule referen.</xs:documentation>
              </xs:annotation>
            </xs:attribute>
            <xs:attribute name="hasRelation" type="gdl:PriorityRelations"
use="required">
              <xs:annotation>
                <xs:documentation>Type of priority.</xs:documentation>
              </xs:annotation>
            </xs:attribute>
            <xs:attribute name="toRule" type="xs:IDREF" use="required">
              <xs:annotation>
                <xs:documentation>Rule reference.</xs:documentation>
              </xs:annotation>
            </xs:attribute>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name="Iteration">
    <xs:annotation>
      <xs:documentation>The element content occurs as many times as the attribute value
defines.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="gdl:RuleType">
          <xs:attribute name="times" type="xs:int" use="required">
            <xs:annotation>
              <xs:documentation>How many iterations.</xs:documentation>
            </xs:annotation>
          </xs:attribute>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="Group">
    <xs:annotation>
      <xs:documentation>A Group of mutually associative rules.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Rule">
          <xs:annotation>
            <xs:documentation>Reference to rule.</xs:documentation>
          </xs:annotation>
          <xs:complexType>
            <xs:attribute name="ref" type="xs:IDREF" use="required">
              <xs:annotation>
                <xs:documentation>Reference to specific rule.</xs:documentation>
              </xs:annotation>
```

```
            </xs:attribute>
          </xs:complexType>
        </xs:element>
      </xs:choice>
      <xs:attribute name="assoc" type="gdl:AssocValue" use="required">
        <xs:annotation>
          <xs:documentation>Type of associativity of rules in group.</xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="RuleType">
    <xs:annotation>
      <xs:documentation>Type for rule content.</xs:documentation>
    </xs:annotation>
    <xs:choice minOccurs="0" maxOccurs="unbounded">
      <xs:element ref="gdl:Nonterminal"/>
      <xs:element ref="gdl:Char"/>
      <xs:element ref="gdl:CharClass"/>
      <xs:element ref="gdl:String"/>
      <xs:element ref="gdl:Optional"/>
      <xs:element ref="gdl:ZeroOrMore"/>
      <xs:element ref="gdl:OneOrMore"/>
      <xs:element ref="gdl:Iteration"/>
      <xs:element ref="gdl:Alternatives"/>
      <xs:element ref="gdl:Sequence"/>
      <xs:element ref="gdl:Permutation"/>
      <xs:element ref="gdl:List"/>
      <xs:element ref="gdl:Annotation"/>
      <xs:element ref="gdl:Layout"/>
      <xs:element ref="gdl:Alias"/>
    </xs:choice>
  </xs:complexType>
  <xs:simpleType name="PropertyNames">
    <xs:annotation>
      <xs:documentation>An enumeration of property names.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
      <xs:enumeration value="GrammarStartSymbol"/>
      <xs:enumeration value="InputEncoding"/>
      <xs:enumeration value="CharacterSet"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="CharValue">
    <xs:annotation>
      <xs:documentation>The character value or it's hexadecimal
representation.</xs:documentation>
    </xs:annotation>
    <xs:union>
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:minLength value="1"/>
          <xs:maxLength value="1"/>
        </xs:restriction>
      </xs:simpleType>
      <xs:simpleType>
        <xs:restriction base="xs:hexBinary">
          <xs:minLength value="1"/>
          <xs:maxLength value="3"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>
  <xs:simpleType name="AssocValue">
    <xs:annotation>
      <xs:documentation>An enumeration of assoc values.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:NMTOKEN">
      <xs:enumeration value="left"/>
      <xs:enumeration value="right"/>
```

```
      <xs:enumeration value="assoc"/>
      <xs:enumeration value="non-assoc"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="PriorityRelations">
    <xs:annotation>
      <xs:documentation>A union of priority relations and assoc
values.</xs:documentation>
    </xs:annotation>
    <xs:union>
      <xs:simpleType>
        <xs:restriction base="gdl:AssocValue"/>
      </xs:simpleType>
      <xs:simpleType>
        <xs:restriction base="xs:NMTOKEN">
          <xs:enumeration value="greater"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>
  <xs:complexType name="openAttrs">
    <xs:annotation>
      <xs:documentation>The type is extended by almost all schema types to allow
attributes from other namespaces to be added to user schemas.</xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:restriction base="xs:anyType">
        <xs:anyAttribute namespace="##other" processContents="lax"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="annotated">
    <xs:annotation>
      <xs:documentation>The type is extended by all types which allow
annotations.</xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="gdl:openAttrs">
        <xs:sequence>
          <xs:element ref="gdl:Annotation" minOccurs="0"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="Appinfo">
    <xs:annotation>
      <xs:documentation>The element content is intended for for automatic
processing.</xs:documentation>
    </xs:annotation>
    <xs:complexType mixed="true">
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:any processContents="lax"/>
      </xs:sequence>
      <xs:attribute name="source" type="xs:anyURI">
        <xs:annotation>
          <xs:documentation>An optional URI reference to supplement the local
information.</xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="Documentation">
    <xs:annotation>
      <xs:documentation>The element content is intended for human
consumption.</xs:documentation>
    </xs:annotation>
    <xs:complexType mixed="true">
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:any processContents="lax"/>
      </xs:sequence>
```

```
      <xs:attribute name="source" type="xs:anyURI">
        <xs:annotation>
          <xs:documentation>An optional URI reference to supplement the local
information.</xs:documentation>
        </xs:annotation>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="Annotation">
    <xs:annotation>
      <xs:documentation>An annotation is information for human and/or mechanical
consumers. The interpretation of such information is not defined in this
specification.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="gdl:openAttrs">
          <xs:choice>
            <xs:element ref="gdl:Appinfo"/>
            <xs:element ref="gdl:Documentation"/>
          </xs:choice>
          <xs:attribute name="id" type="xs:ID"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
</xs:schema>
```