VAWi.de
Virtueller Weiterbildungstudiengang
Wirtschaftsinformatik Master of Science

Concept and implementation of a rule framework to dynamically transform data and queries for heterogeneous collections

Masterarbeit

Im Virtuellen Weiterbildungsstudiengang Wirtschaftsinformatik

Verfasser: Tobias Gradl
Matrikelnummer: 1589906
10. Fachsemester
Sperberweg 8
92720 Schwarzenbach

Betreuer: Prof. Dr. Andreas Henrich
Otto-Friedrich-Universität Bamberg
Lehrstuhl für Medieninformatik
An der Weberei 5
96047 Bamberg

Abgabe: 02.07.2014
SS 2014

Contents

List of Figures

List of Tables

List of Listings

Acronyms

AHDS	Arts and Humanities Data Service
ANTLR	ANother Tool for Language Recognition
AST	Abstract Syntax Tree
ATL	ATLAS Transformation Language
ATN	Augmented transition network
BSD	Berkeley Software Distribution
CDWA	Categories for the Description of Works of Art
DC	Dublin Core
DFA	Deterministic finite automaton
DSL	Domain Specific Language
DTA	Deutsches Textarchiv
EBNF	Extended Backus–Naur Form
ETL	Extract, transform, load
IR	Intermediate Representation
JSON	JavaScript Object Notation
LIFO	Last In First Out
MARC	Machine Readable Cataloguing

MDA	Model-driven architecture
MODS	Metadata Object Description Schema
OAI-PMH	Open Archives Initiative - Protocol for Metadata Harvesting
OCLC	Online Computer Library Center
PHP	PHP: Hypertext Preprocessor
POS	Part-Of-Speech
QVT	Query/View/Transformation
TEI	Text Encoding Initiative
VRA	Visual Resources Association
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XPath	XML Path Language
XSLT	Extensible Stylesheet Language Transformations

1 Chapter
Introduction

Research data of the arts and humanities is usually stored in collections such as museums, archives and libraries. Similar to the traditional counterparts, the landscape of digital collections can be characterized as distributed system. In combination with the autonomy of their owning institutions and the focused research communities and disciplines, the geographical and logical[1] distribution resulted in the development of numerous collection-specific and standardized export schemata (compare e.g. Polfreman 2005; Vierkant 2013).

Despite the typically negative connotation of the term *heterogeneity* in database and data integration research—in the particular context of the arts and humanities, heterogeneity on the structural and semantic levels correspond to the diversity of the disciplines, its research questions and communities. For this reason, *heterogeneity* must not exclusively be considered as integration problem, but requires a more sophisticated perspective: As discussed in Gradl (2014), the preliminary work to this thesis, heterogeneity can be classified in *technological* and *domain* aspects—facilitating:

- the dedicated allocation of technological and domain experts to respective tasks, and
- the determination between technical integration, which can often be solved by means of traditional data integration approaches—and context-related integration.

[1] diversification of the context and content (e.g. object types, sources, coverage)

Ideally, technical heterogeneity problems such as conflicting access protocols or data encodings are solved in a generic fashion, which allows the abstraction from technical aspects. Initial steps towards this goal have been taken by the work in Gradl (2014) by providing the theoretical foundation required to separate the problems of model (i.e. schema) and metamodel (i.e. schema languages) integration—allowing semantic integration to be performed on the basis of schemata in the sense of regular tree grammars and thus abstracting from the primarily technical task of converting between formats such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON).

Objectives and goals of this thesis

The contextual objective of the presented thesis builds on this preliminary, theoretical work and focuses on the semantic extension of schemata, which Gradl (2014, 34-37) introduced as *labeling functions*, and on the execution of *concept mappings* (Gradl 2014, 41-42)—transforming source data into their conceptually equivalent target forms. Whereas Gradl (2014) provided a means for the description of data and semantic correlations on the schema-level, this thesis focuses on the description of instances and their transformation.

The approach presented in this thesis is based on the hypothesis that the data integration problem in the particular context of the arts and humanities can be interpreted and solved on the theoretical foundation of formal languages. The recognition of a provided input, its transformation into an internally rewritten representation and the generation of output are typical tasks of language applications, which form the conceptual base of this thesis.

Despite the contextual focus, the concept and implementation of the transformation framework is intended to be developed as an autonomous and widely context-neutral component, which can be reused and extended.

Structure of the thesis

After this introduction, chapter 2 provides an overview of the application domain of the arts and humanities and the characteristics that are relevant for this thesis: digital collections and the contained research data. A short glance over abstract

use-cases within the domain shows the dynamic character that is required of any integrative solution in the arts and humanities.

Chapter 3 then introduces the theoretical foundation of the rule framework, which consist in the basics of formal language theory, the structure and behavior of language applications mainly with respect to lexical, syntactical and semantic analysis of input data, as well as the preliminary work in (Gradl 2014), which introduces a formal metamodel for the specification of schemata and mappings. The discussion of foundations for the conceptual work in this thesis is continued with chapter 4, which presents an overview of ANTLR, a supporting framework for the creation of language applications that is used for the task of language recognition in this thesis.

After a discussion of the primary problems that the concept of this thesis is intended to solve, the conceptual work in chapter 5 is initiated by providing an overview of the logical architecture of the rule framework as a form of language application. The phases of data description and data transformation are separated primarily to provide a high level of expressiveness, while keeping the complexity of language specifications at a reasonable level. Important aspects of the implementation of the rule framework and the prototypical web application are presented in chapter 6, after which chapter 7 concludes the conceptual work by presenting an initial proof-of-concept by implementing exemplary language specifications and transformation rules and evaluating the runtime performance of the system.

Chapter 8 concludes this thesis by providing a summary of the central building blocks of the developed concept and presenting future tasks and extension points.

2 Chapter
Context and motivation

As a branch of academic disciplines, the arts and humanities include numerous specific fields—each focusing on particular aspects related to human constructs and culture. With regard to the integrative goals of this thesis and the motivating research context, specific conclusions can be inferred from the characteristics of the academic landscape of the arts and humanities.

In section 2.1 those characteristics of the arts and humanities are introduced, which are of particular relevance for data integration—the primary motivation for the concept of the data transformation framework in this thesis. Section 2.2 then presents a classification of abstract use-cases—providing a contextual base for the conceptual work.

2.1 Application domain

From a holistic perspective, the application domain of the arts and humanities as well as the landscape of digital collections are characterized by their high degrees of distribution, heterogeneity and autonomy. Often referenced as orthogonal dimensions of information integration (see Leser & Naumann 2007, 50), these characteristics typically originate from a logical and geographical *distribution* of institutions and data sources. Without the existence of superior authorities focusing on the consolidation and coordination of data structures and data processing practices, the additional aspect of *autonomy* further promotes the development of

heterogeneity on various abstraction levels (Sheth & Larson 1990, 185-189). Despite the negative connotation of the term *heterogeneity* when reducing data integration to its subset of technical problems, the semantic, context-specific aspects of *heterogeneity* often reflect the diversity and complexity of the respective application domain. However, from a traditional perspective (compare e.g. Lenzerini 2002, Sheth & Larson 1990), the task of data integration often consists in the unification of heterogeneous data in terms of a globally integrative schema.

As indicated by the following sections, traditional integration approaches are, however, not applicable to an integrative solution for the holistic context of the arts and humanities—especially if detailed perspectives on an integrated set of research data are required.

2.1.1 Digital collections

Comparable to traditional collections like museums, archives or libraries, digital collections contain objects such as drawings, music or texts, which are often[2] provided in terms of exhibitions, education- or research-oriented activities. In digital collections, objects are stored in digital forms, which are often encapsulated or referenced by metadata records—allowing descriptions of the objects in terms of e.g. aggregated information or annotations. Aside from new opportunities to enrich objects without a need to modify the original, physical resource, digital collections especially provide the benefits of location- and time-independence if web-based access mechanisms are provided.

As premise for integrative services, digital collections are required to provide resources though machine-accessible interfaces, of which a prominent and standardized example is available in terms of the Open Archives Initiative - Protocol for Metadata Harvesting (OAI-PMH) (Lagoze et al. 2002). OAIster[3] is an example of an aggregation service for OAI-PMH-accessible, digital collections, which was instantiated as research project at the University of Michigan in 2002 and is currently maintained by the Online Computer Library Center (OCLC) (Hagedorn 2003). Although OAIster requires the registration of repositories by the contribut-

[2] *often* because there are private collections without public accessibility
[3] http://oaister.worldcat.org/

ing organizations, its search engine currently provides access to "over 30 million records contributed by over 1,500 organizations"[4]—indicating the amount of digital collections that are of potential relevance for research questions in the arts and humanities.

2.1.2 Data schemata

Irrespective of whether they have been created as additional form of access to traditional collections or they exist only in virtual form, digital collections are typically owned by autonomous institutions—focusing on particular academic disciplines or research communities. As a consequence, heterogeneity between collections exists in terms of the used schemata and the context- and discipline-specific knowledge that is required to understand the usage of the schemata and the contained data.

Scholars of the individual disciplines usually focus on a specific set of concepts and the subset of properties that is relevant for their particular research question. Situated within such academic contexts, the digitization and description of research objects in digital collections is influenced by the context-specific requirements. As an example, consider photographs and descriptions of churches being produced by scholars with backgrounds in art history, theology and architecture. It can be assumed that the concepts addressed by the photographs and descriptions will differ due to (1) the relevance of individual properties and (2) the level of applicable knowledge (e.g. in relation to the theological symbolism or period-related architectural facets). As a consequence, different custom or standardized data and metadata schemata might qualify for these context-specific descriptions.

Only few studies of the use of data and metadata standards in the arts and humanities exist. An example can be found in the work of Polfreman (2005), who presents findings based on the document collection of the Arts and Humanities Data Service (AHDS). As such, the analyzed data originated from a broad context of visual and performing arts, archaeology, history, literature, language and linguistics. The analysis concluded in a list of metadata *standards*[5], which includes

[4] according to http://www.oclc.org/oaister/about.en.html
[5] accessible at http://www.ahds.ac.uk/metadata/arts-humanities-metadata-formats.htm

generic schemata like Dublin Core (DC), Machine Readable Cataloguing (MARC) or Metadata Object Description Schema (MODS) as well as discipline- and/or resource-type specific standards such as Categories for the Description of Works of Art (CDWA), Text Encoding Initiative (TEI) header and Visual Resources Association (VRA) core. In contrast, the 2012 Census of Open Access Repositories in Germany (Vierkant 2013) analyzed 141 repositories of scientific publications, which total in overall 704,121 records. The analysis was conducted with respect to the structure and content of the repositories and—among other findings—concluded that DC was the only metadata format that has been widely adopted—thus deserving to be called a *standard*. (Vierkant 2013, 21)

A general conclusion on the use and applicability of such standards or custom schemata is not possible from a holistic perspective of the arts and humanities because each specific research or collection context might have different requirements. In addition, these requirements are often only identifiable by experts of the respective field or context.

Two exemplary records are presented to illustrate the wide spectrum of commonly utilized schemata: Listing 2.1 shows a complete record[6], which follows the constraints of simple DC and belongs to the data set of Pangaea, a data publisher for earth and environmental science[7]. The record illustrates a context-specific adaption of a standardized schema (DCMI 2012) e.g. in that it encapsulates non-atomic content in the *creator*, *coverage* and *subject* elements, which each follow a context-specific substructure.

```
<oai_dc:dc xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:xsi="http://
    www.w3.org/2001/XMLSchema-instance" xmlns:oai_dc="http://www.
    openarchives.org/OAI/2.0/oai_dc/"    xsi:schemaLocation="http://www.
    openarchives.org/OAI/2.0/oai_dc/ http://www.openarchives.org/OAI/2.0/
    oai_dc.xsd">
  <dc:title>Ice rafted debris (&gt; 2 mm gravel) distribution in sediment
      core PS2646-5</dc:title>
  <dc:creator>Grobe, Hannes</dc:creator>
  <dc:source>Alfred Wegener Institute, Helmholtz Center for Polar and
      Marine Research, Bremerhaven</dc:source>
  <dc:publisher>PANGAEA</dc:publisher>
  <dc:date>1996-02-29</dc:date>
  <dc:type>Dataset</dc:type>
```

[6] http://ws.pangaea.de/oai/?verb=GetRecord&metadataPrefix=oai_dc&identifier=oai:pangaea.de:doi:10.1594/PANGAEA.50542

[7] http://www.pangaea.de/

```
  <dc:format>text/tab-separated-values, 1148 data points</dc:format>
  <dc:identifier>http://doi.pangaea.de/10.1594/PANGAEA.50542
    </dc:identifier>
  <dc:identifier>doi:10.1594/PANGAEA.50542</dc:identifier>
  <dc:language>en</dc:language>
  <dc:rights>CC-BY: Creative Commons Attribution 3.0 Unported</dc:rights>
  <dc:rights>Access constraints: unrestricted</dc:rights>
  <dc:coverage>LATITUDE: 68.556667 * LONGITUDE: -21.210000 * DATE/TIME
      START: 1994-09-19T14:56:00 * DATE/TIME END: 1994-09-19T14:56:00 *
      MINIMUM DEPTH, sediment/rock: 0.0 m * MAXIMUM DEPTH, sediment/rock:
      11.5 m</dc:coverage>
  <dc:subject>ARK-X/2; AWI_Paleo; Denmark Strait; Gravity corer (Kiel type)
      ; Ice rafted debris; IRD-Counting (Grobe, 1987); Paleoenvironmental
      Reconstructions from Marine Sediments @ AWI; Polarstern; PS2646-5;
      PS31; PS31/162</dc:subject>
</oai_dc:dc>
```

Listing 2.1: PangaeaDC example

On the other hand, listing 2.2 shows a shortened[8] TEI document[9] that has been taken from the Deutsches Textarchiv (DTA)[10]. Following the TEI guidelines (Burnard & Bauman 2014), the document contains a *teiHeader* with metadata elements such as *title*, *author* and *editor* within the *title statement* and references to bibliographic resources in the *source description*. In addition, the document contains a digitized version of the original textual resource (*text* element) along with structural information and annotations.

```
<TEI xmlns="http://www.tei-c.org/ns/1.0">
  <teiHeader>
    <fileDesc>
      <titleStmt>
        <title type="main">Abriß der neuesten Staatswissenschaft der
          vornehmsten Europäischen Reiche und Republicken</title>
        <title type="sub">zum Gebrauch in seinen Academischen Vorlesungen
        </title>
      ...
    </fileDesc>
  </teiHeader>
  <text>
    <front>...</front>
    <body>
      <div n="1">
        <head><hi rendition="#b"><hi rendition="#g">Vorbereitung.</hi></hi>
            </head>
```

[8] the original document contains around 759k characters
[9] http://www.deutschestextarchiv.de/book/download_xml/achenwall_staatswissenschaft_1749
[10] en: German Textarchive (DTA), see http://www.deutschestextarchiv.de/

```
        ...
        <div n="2">
          <head>. 1.</head>
          <lb></lb>
          <p>
            <hi rendition="#in">D</hi>er Begriff der &#x017F;ogenannten<hi
                rendition="#fr">Statistic ,</hi>das<lb></lb>i&#x017F;t, der<
                hi rendition="#fr">Staatswi&#x017F;&#x017F;en&#x017F;chaft
                einzelner Rei—<lb></lb>che</hi>wird &#x017F;ehr ver&#x017F;
                chiedentlich angegeben, und man<lb></lb>trifft unter der
                gro&#x017F;&#x017F;en Menge Schriften davon<lb></lb>nicht
                leicht eine einzige an, welche in der Zahl<lb></lb>und
                Ordnung ihrer Theile mit der andern ...
          </p>
        </div>
        ...
    </body>
  </text>
</TEI>
```

Listing 2.2: DTA TEI example

The complexity of other schemata that are common in the arts and humanities (see e.g. the list of (Polfreman 2005)) can be expected to range between the simple metadata standard of DC and the encapsulation of a digitized object within a record in TEI. CDWA and VRA could for example be considered as metadata standards as the described resource is not encoded within a particular record. However, both standards allow a faceted description of objects and as such often result in the creation of detailed information.

2.2 Use-cases

The process of designing integrative data integration systems according to Lenzerini involves the execution of an extensive requirements analysis, which is intended to provide details about the concepts contained in the integrated local schemata and leads to the derivation of a global schema. The appropriateness of such a global schema can be determined according to four requirements expressed in Batini et al. (1986, 337):

- *Completeness*: The schema includes all concepts that are contained in the local schemata and that are relevant to the considered application domain.

- *Correctness*: Any local concept that is represented by the global schema must be reflected with equivalent semantics.
- *Minimality*: Concepts with identical semantics are represented only once.
- *Understandability*: Labels of elements are chosen in a fashion that prevents ambiguities and enhances overall understandability.

Resulting from the consideration of the arts and humanities as application domain of data integration, the methodological execution of an extensive requirements analysis and the subsequent derivation of appropriate unifying schemata is prevented by the complexity and size of the domain: If the design of a global schema aims a the requirement of completeness, (1) a large amount of concepts have to be included within the schema that are irrelevant in specific use cases and (2) the understandability and correctness are potentially impacted by the complexity of the schema.

In addition, the preliminary work in Gradl (2014, 22-23) identified generic use-cases for the integration of heterogeneous data in the context of the arts and humanities—reflecting the need for alternatives to the traditional integration approach as of Lenzerini (2002)—from an abstract level:

- *Broad Search*: A large set of heterogeneous collections is selected as potentially relevant and needs to provide means for answering queries. The contextual intersection of all selected collections is limited due to the disciplinary breadth.
- *Deep search*: Collections that are relevant for a particular community or research question are identified. Queries over the integrative view of the semantically related collections are expected to provide means for utilizing the semantic cohesion e.g. in the form of search facets or appropriate visualization techniques.
- *Data integration*: Materialized data integration is often required for migrating data between different systems or the application of long-running analyses and visualization techniques. The relevant set of collections is required to be determined by domain experts with respect to particular research questions and ranges between a large set of unrelated and smaller sets of tightly associated collections.

Aside from typical characteristics of distributed systems, the application domain of the arts and humanities requires an integrative system that especially respects the *dynamics* of individual use-cases. In particular, the set of collections and hence the utilized schemata, the purpose and the execution of any data transformation and integration have to be specifically adaptable to the context needs and hence determinable by domain experts.

3 Chapter
Theoretical foundation

The concept of a data transformation framework in this thesis is largely based on the theoretical foundation of formal languages. The work in Gradl (2014), which provides the contextual base of this thesis introduces the definition of schemata as an extended form of regular tree grammars. As the exemplary records presented in section 2.1.2 showed, the terminal nodes of such schemata (the element values) can contain non-atomic content, which could be further decomposed and processed if the implicit syntax and semantics are explicated.

Since the conceptual work in this thesis proposes means for the formal description of such non-atomic context based on language specifications, section 3.1 provides an overview of the basic terminology and concepts of formal language theory. In section 3.2 the structure and behavior of language applications are introduced. Section 3.3 concludes this chapter with a discussion of the theoretical foundation of schema and mappings as of Gradl (2014).

3.1 Formal language theory

Formal language theory focuses on structural patterns and as such the lexical and syntactical features of languages and allows the definition of formal language specifications: the *grammars* (Parr 2013, 57-82). A central characteristic of a formal language thereby consists in its exact specification, which facilitates the implementation of algorithms for the validation and processing of input with

respect to the defined language.

3.1.1 Basic Terminology

For every formal language, an alphabet forms a finite set of atomic symbols—elements, which cannot be decomposed in a meaningful way. Valid exemplary alphabets could be formed as the set of lowercase latin characters $\{a, b, c, d, \ldots, y, z\}$ or a set of digits $\{0, 1, 2, \ldots, 9\}$. Based on a defined alphabet A, a token x over A can be formed by concatenating any finite set of elements of that alphabet:

- $x \in A^*$ allows any token over A, including the empty token.
- $x \in A^+$ denotes any non-empty token over A.

Considering an alphabet $A = \{a, b\}$, then the language over the alphabet A is defined as infinite set of tokens $A^* = \{\epsilon, a, b, aa, bb, ab, ba, aaa, bbb, aba \ldots\}$—if no further syntactical constraints are specified. Essentially, a language L over A can be defined as any set of tokens, which can be formed from the elements of A^*, or: $L \subseteq A^*$.

In literature the concept of a token is often referred to as *word* or *string* (see e.g. Chomsky 1956, 114-115; Crespi-Reghizzi 2009, 8). Since the concept of a *word*, however, generally addresses concatenations over an alphabet that expose a meaning in the sense of natural languages and *string* often relates to any character stream e.g. in common programming languages, the term *token* is preferred in this thesis in order to prevent ambiguities in the conceptual and implementation-related sections. Based on the fundamental understanding of a formal language as $L \subseteq A^*$, further constraints can be introduced to specify token combinations, which are considered valid over L. Such valid combinations of tokens are—again referring to natural languages—called *sentences*.

In order to detail the definition of a formal language L, the set of valid tokens could be further restricted by mathematically narrowing the set definition e.g. in the form of $L_1 = \{\epsilon, a, b, aa, bb, ab, ba\}$ for a finite language or $L_2 = \{a, ab, abb, abbb, abbbb, \ldots\}$ for an infinite language. For both finite and infinite languages, token production functions can be utilized to define more complex restrictions on the set of valid tokens for a language. Although equivalent, a more precise specification of the above L_2 can be formalized as $L_2 = \{ab^i : i \geq 0\}$.

In software development practice, simple languages are often constructed implicitly by implementing parsers that process provided input strings. Examples can be found in configuration file readers, where configuration keys are followed by an '=' symbol, the assigned value of the key and a terminating newline character. A suitable parser could sequentially process all lines of a configuration file, split the content of every line removing the '=' symbol and assigning the first substring to a *key* field, the second substring to a *value* field of a *property* object.

As a third alternative, a formal language can be defined as a finite set of rules, called *grammar* (Cooper 2012, 11). The rules of a grammar define production functions that generates exactly the set of valid sentences of a language. Chomsky (1956, 114) defines a grammar as "a device of some sort that produces all of the strings that are sentences of L and only these".

3.1.2 Language derivation

Chomsky (1956, 117) presents his original definition of a parse-structure grammar as triple $\langle V, \Sigma, F \rangle$, with V as finite vocabulary, a finite set of initial strings $\Sigma \subseteq V$ and a finite set of rewrite rules $X \rightarrow Y$, where $X, Y \subseteq V$. Whereas this original definition provides an adequate base for the description of unrestricted type 0 grammars, Chomsky also introduces the distinction between terminal and nonterminal symbols of grammars to reflect intermediary steps in the production of sentences. As a result, a generative grammar can be adapted from Chomsky (1956, 117) and Parkes (2008, 30-31) in terms of a quadruple $\langle N, T, P, S \rangle$, where:

- N is a finite set of nonterminal symbols, which reflect intermediary states of language production and do not occur in the ultimately produced sentences.
- T constitutes the finite set of terminal symbols—valid tokens, which are used to form sentences
- P is the finite set of production rules of the form $x \rightarrow y$, with $x \in (N \cup T)^+$, $y \in (N \cup T)^*$, if the grammar is not further restricted (type 0 grammar) and
- $S \in N$ as start symbol is the designated root of the grammar, with which the generative language production process begins.

The classification and distinction between grammars of the Chromsky hierarchy is based on the distinction of certain patterns in P, which not only impact the

expressiveness and power of the produced languages, but also the complexity of its algorithmic processing.

3.1.3 Classification of formal languages

Grammars and the languages they produce have been originally classified in Chomsky (1956). Since then, the classification has remained unchanged and can be found in recent textbooks and publications[11] due to the precision and simplicity of distinction between the grammars and the ability to assign the types of recognizers that are required to parse a generated language.

Grammar	*Types of rules*	*Type of recognizer*
Type 0 (unrestricted)	$x \rightarrow y$, with $x \in (N \cup T)^+, y \in (N \cup T)^*$	Turing machine
Type 1 (context-sensitive)	$x \rightarrow y$, with $x \in (N \cup T)^+, y \in (N \cup T)^+, \lvert x\rvert \leq \lvert y\rvert$	Restricted form of turing machine (e.g. linear bounded automation)
Type 2 (context-free)	$x \rightarrow y$, with $x \in N, y \in (N \cup T)^*$	Push-down automation
Type 3 (regular)	$w \rightarrow x$, or $w \rightarrow yz$, with $w \in N, x \in (T \cup \epsilon), y \in T, z \in N$	Finite state automation

Table 3.1: Chromsky classification of grammars (based on Chomsky 1956, 113-124; Crespi-Reghizzi 2009, 87-91; Parkes 2008, 36)

Type 3 grammars Regular grammars constitute the most restrictive type of grammars in the Chromsky hierarchy and allow a single nonterminal on the left-hand side of production rules and either ϵ, a single terminal or a single terminal followed by a single nonterminal on the right.

Regular grammars and expressions have been widely adopted and expose sufficient expressiveness e.g. in the context of semi-structured data as shown in Murata et al. (2005) and Gradl (2014). Due to the simplicity of the production rules in type 3 grammars, recognizers for any regular grammar can be based on finite automata (Crespi-Reghizzi 2009, 94-95). However, regular tree grammars show limitations e.g. for constructs such as precedence in an arithmetic operation $2 + 3 * 4$ or the

[11] compare e.g. Crespi-Reghizzi (2009), Parkes (2008) or Murata et al. (2005)

validation of token order: the semantics of a nested statement like $(a, b, (c, d))$ cannot be exactly reflected by means of a regular grammar (Parr 2013, 24).

Type 2 grammars Context free grammars extend the type of allowed production rules compared to type 3 grammars: while the left-hand side is still required to be a single nonterminal element $x \in N$, the right-hand side allows any combination of terminal and nonterminal symbols—including ϵ.

Context-free grammars have evolved to become the common theoretic core of language applications with a need for a “more powerful notation than regular expressions that still leads to efficient recognizers” (Cooper 2012, 86).

Type 1 and 0 grammars As the name indicates, unrestricted grammars allow any element with the exception of the empty element ϵ on the left-hand side of a production rule $x \in (N \cup T)^+$ and anything (including ϵ) on the right $y \in (N \cup T)^*$.

Context sensitive grammars can be distinguished from type 0 grammars only by the characteristic that ϵ is not allowed to occur on either side. In addition, the length of the left-hand side of the rule needs to be less or equal to the length of the right-hand side. Type 0 grammars depend on turing machines in order to detect whether a sentence belongs to an unrestricted language, type 1 grammars are exposing less complexity and can be solved on the basis of linear bounded automation Parkes (2008, 36).

As suggested by Crespi-Reghizzi, unrestricted and context-sensitive grammars—while being mathematically and logically interesting—can be considered as "almost irrelevant for language engineering and computing" (Crespi-Reghizzi 2009, 91). This view is confirmed by Ghosh and Parr, who rather introduce practical variations to context-free grammars in order to cope with the context-sensitive features of actual significance in a Domain Specific Language (DSL): context-sensitive validation in Ghosh (2011, 271-272) and semantic predicates in Parr (2013) both show attempts to include information about the context of parsed phrases in order to resolve ambiguous sentences in context-free grammars.

3.2 Language applications

Traditional forms of language applications are formed by compilers—i.e. computer programs, which parse source code specified in terms of a computer language and translate the instructions within this code into machine executable code. Cooper (2012, 1) understands compilers more generally as "computer programs that translate a program written in one language into a program written in another language", indicating that compilers can be considered as a kind of unidirectional *translator* between a human-readable source language and a target language. Compilation is performed against human readable source code and results in immediately executable binary code or an intermediate representation. In the latter case, a runtime environment[12] is needed to interpret such intermediate representations at the execution of the program. Instead of translating into an executable target language, interpreters execute the instructions encoded within the source code.

Despite the focus on the complex tasks of source code compilation or interpretation, the meaning of *language applications* can be interpreted from a wider perspective according to (Parr 2010, 13) as "any program that processes, analyzes, or translates an input file." Although applications or components e.g. for processing configuration files or importing data from external files are typically not defined as language applications, they are based on the (often implicit) specification of a language providing rules, which input should be considered as valid and how it should be further processed.

3.2.1 Compilers and data structures

Compilers have developed into large and complex computer programs, which can be structured into the coherent structural elements of the *front end*, *optimizer* and *back end*—each encapsulating an particular type of logics (compare Parr 2010, 20-21; Cooper 2012, 6-21):

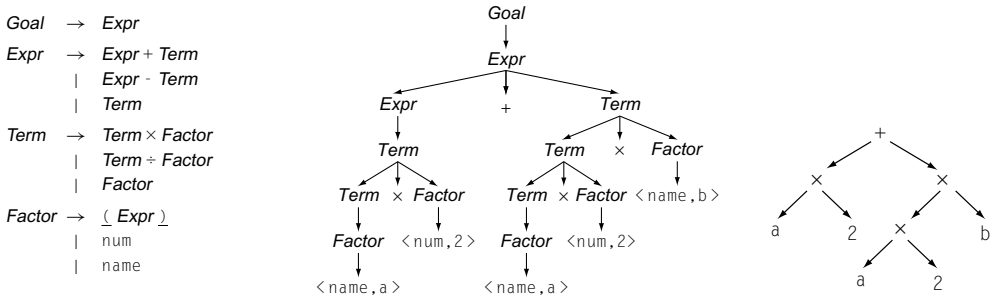
- The *front end* aggregates the functionality required to recognize languages: Any input that is provided as potential sentence of the implemented language

[12] common examples are the Java Virtual Machine (JVM) and Microsoft's .NET Framework

is processed in a chain of lexical analysis, syntax analysis and the generation of an Intermediate Representation (IR) of the provided input.

- The task of an *optimizer* can be summarized as semantic analyzer, which collects information about a provided IR, annotates and/or rewrites it according to the needs of a surrounding application and functional goal and—in the case of an interpreter—executes the instructions encoded in the IR. The optimization thereby consists in the often iterative annotation and transformation of the IR, which finally results in the generation of an either executable or exportable output.
- The *back end* of traditional compilers generate output for the hardware abstraction layer—including the selection of the executed instructions, the allocation of registers and the scheduling of the instructions (Cooper 2012, 16-21).

The exchanged data structure is called an IR, which is usually modified multiple times during a compilation process—until a final representation is provided as output. The choice of IRs depends on multiple factors such as the goal of the current stage of compilation or—abstracting from compilers—the type of language applications in general.



(a) Parse tree reflecting language derivation (b) AST

Figure 3.1: Common graphical IRs (Cooper 2012, 226-227)

Cooper (2012, 223-243) distinguishes three classes of IRs:

- *Graphical IRs* include all graph- or tree-like data structures that encapsulate the knowledge of a language application at a certain phase. Common variants include *parse trees* and *abstract syntax trees*.

- *Linear IRs* represent linear sequences of instructions, which are sequentially executed e.g. by assemblers. Entries in the sequence can contain multiple instructions, which are executed in parallel.
- *Hybrid IRs* combine properties of graphical and linear IRs.

Figure 3.1 introduces two prominent forms of graphical IRs, which are often found in compiler engineering and language applications in general: *Parse trees* typically represent the derivation of production rules when processing input (Cooper 2012, 89-94). Every nonterminal node of the parse tree represents a grammatical rule, the terminals relate to the originally parsed tokens. An initial optimization step often consists in the reduction of the size of IRs that need to be held in memory. An AST reflects such a reduced form, which should be (Parr 2010, 77):

- *dense* by including only relevant nodes,
- *convenient* by being optimized for tree traversal and
- *meaningful* by focusing on instructions and actions that result from the original parse tree, not the syntactical structure of the parse.

3.2.2 Language application abstraction

Irrespective of being explicitly or implicitly defined, language applications implement the syntactical and semantic rules of a language by performing tasks according to sentences of the specified language. To provide a broad overview of language applications and their functionalities, figure 3.2 combines the perspective on the structure of compilers as presented in Cooper (2012, 6-21) and the discussion of the types of language applications in Parr (2010, 20-21).

Abstracting from the concept of compilers, language applications realize the functional building blocks of front ends, optimizers and back ends as required by the application domain. Parr classifies four categories of language applications (Parr 2010, 21-22):

- *Reader*: A reader implements the front end of a language application by scanning and parsing provided input, thus determining whether the input is valid with respect to the implemented language. Readers are often components of other computer applications and can take the form of configuration file readers, source code analyzers etc.

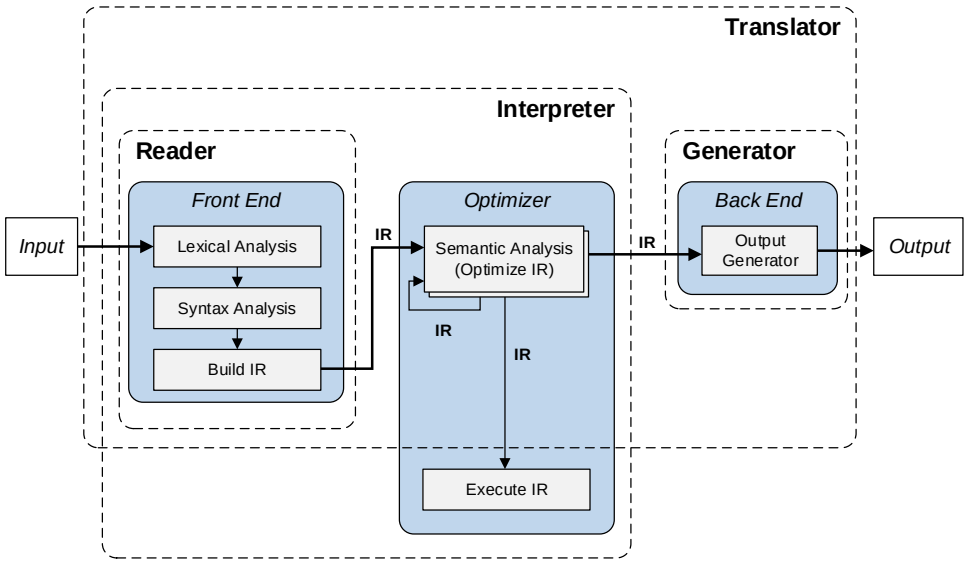


Figure 3.2: Language application abstractions (based on Parr 2010, 20-21; Cooper 2012, 6-21)

- *Interpreter*: In addition to the functionalities provided by a reader, interpreters execute the instructions encoded in a provided input. The IR created in the front end is processed in terms of a possibly iterative semantic analysis in order to determine the meaning of syntactical components, after which the instructions are executed.
- *Generator*: A generator traverses an IR in order to generate output in a different language. Such generators can be found in compilers, where code in a target language such as machine code is generated.
- *Translator*: A translator implements the complete language processing pipeline including language recognition in the front end, interpreting and rewriting IRs in optimizing phases and generating an output in terms of a target language.

The categories allow the definition and classification of language applications aside from source code compilers and interpreters and illustrate that language implementation does not necessarily equal the construction of a compiler. As a result of the abstraction from the clearly defined subtasks within the compilation process, custom business logic is implemented in terms of the optimizer, after the

traditional tasks of the front have been completed by generating an appropriate type of IR. Whereas the subsequent tasks of semantic analysis, the reaction to recognized sentences or the translation into a target language are part of the domain-specific logic, the phase of language recognition can be considered as a common characteristic of any language application: Languages are formal concepts and describe valid constructs over an alphabet specified in terms of grammars.

With respect to the recognition of languages, the determination of the validity of the lexical and syntactical structure of a provided input can be identified as essential phases.

3.2.3 Lexical analysis

Lexical analysis constitutes the first task executed by a reader in order to understand a provided input and focuses on the character level. A *lexer* (or *scanner*) reads input in terms of a stream of characters and produces a stream of tokens (Cooper 2012, 26). Every recognized token is assigned to a syntactic category in order to reflect the syntax at the character level, the *lexical structure* (Parr 2010, 43).

A simple processing form for western languages could be based on a four-step algorithm:

1. Look for the next whitespace in the character stream
2. Group all alphanumeric characters to the left of the whitespace from left to right into a token
3. Find a lexical category that matches the identified token, e.g. ID if the pattern [a-zA-Z] is matched or NUMBER if [0-9] is matched.
4. Put the recognized token on the token stream and proceed with 1 until the end of the character stream

Properties of the analysis are often defined in terms of language specifications, such as the collection of lexical categories, the processing orientation (e.g. right-to-left for arabic languages) or the types of whitespaces to ignore (the newline character can be ignored e.g. when processing Java source code or JSON, but is required e.g. for parsing Python since it represents the command terminator).

Lexers are generally considered as a simple task of language recognition as they are typically built upon few grammatical rules that are required to reflect lexical

properties. In addition, lexers e.g. for different western languages or programming languages show similarities with respect to the lexical rules and can thus often be reused.

3.2.4 Syntax analysis

In contrast to lexers, parsers focus on an analysis of the syntax of a provided input and as such need to determine, if the stream of tokens presented by the lexer forms a valid sentence with respect to the language specification. For this purpose, parsing requires a machine-readable specification that allows to determine the validity of provided input. Formal language specifications are grammars, which can—aside from the theoretical interpretation—be understood as "executable programs written in a domain-specific language (DSL) specifically designed for expressing language structures" Parr (2010, 38).

From a formal perspective, the goal of syntax analysis consists in the application of production rules on the provided input on the basis of an underlying grammar in order to derive an IR. As the Chromsky classification of grammars in table 3.1 indicated, the complexity of recognition depends on the type of grammar that is required to generate the desired language. Especially due to the syntactical limitations of regular grammars and the high complexity to recognize context-sensitive and unrestricted grammars, context-free grammars have evolved to form the theoretical base of many modern language processing applications (compare Cooper 2012, 85-89, Crespi-Reghizzi 2009, 30-33).

Two primary classes of parsers for context-free languages can be distinguished (Parr 2010, 38-48, Cooper 2012, 96-140):

- The functionality of *bottom up* or *LR* parsing is mainly characterized by building parse-trees from the leaves to the root. As initial step in bottom up parsing, each token of the parsed stream is represented as a leaf. Iterations of the parser match production rules based on the topmost nodes of each stage and associate the parent nonterminals.
- *Top down* or *LL* parsers build parse trees from the root by applying the production rules left to right. At each iteration of the parse, subtrees that rewrite the current nonterminals are appended.

Although according to e.g. Cooper (2012, 95) bottom-up parsing is applicable to more types of languages, top-down parsing has gained more practical relevance due to more efficient implementations. Top-down parsing is one of the functional building blocks of this thesis and will be further discussed in subsequent sections.

3.3 Preliminary work

The primary focus in Gradl (2014) consists in the creation of a theoretical model for the representation and integration of semi-structured schemata that are used by the digital collections of the arts and humanities. The traditional approach to the schema-level integration of semi-structured data is characterized in Lenzerini (2002, 233-234) from a theoretical perspective, based on the formalization of a data integration system $I = \langle G, S, M \rangle$, where

- G represents the global schema utilized to present a unifying view over heterogeneous data,
- S constitutes the source schema of a data source, whose data should be presented in terms of the global schema and
- M typically describes a set of value correspondences, of which each represents an associated pair of elements of the source and global schema.

This traditional approach to data integration reduces the complexity of the data integration problem by defining a system-wide integration view to which all local sources are mapped. Due to the limitations of such an approach with respect to the support of discipline- or research-specific views on data, the work in Gradl (2014) provides a theoretical approach for the representation of heterogeneous schemata and their interrelations: Based on the abstraction from technical aspects of heterogeneity, the technology-independent definition of *schema* is reducing the complexity by separating the integration problems into technological and context-related aspects.

3.3.1 Conceptual architecture

With a focus on XML documents and schema languages, Murata et al. (2005, 663) and Zhang et al. (2008, 424-425) showed that a formal description and anal-

ysis of XML schemata can be based on the foundation of regular tree grammars. Abstracting from the syntactical specifics of XML, Gradl (2014) proposed the modeling architecture as shown in figure 3.3, which is based on the four layer modeling architecture of the Model-driven architecture (MDA). Despite the focus of the MDA on software development and particularly the generation of platform-independent models and their semi-automatic and incremental transformation to implementation-oriented models (see e.g. Bézivin 2005; Atkinson & Kühne 2003), the adaption of the architecture facilitates the formalization of semi-structured data.

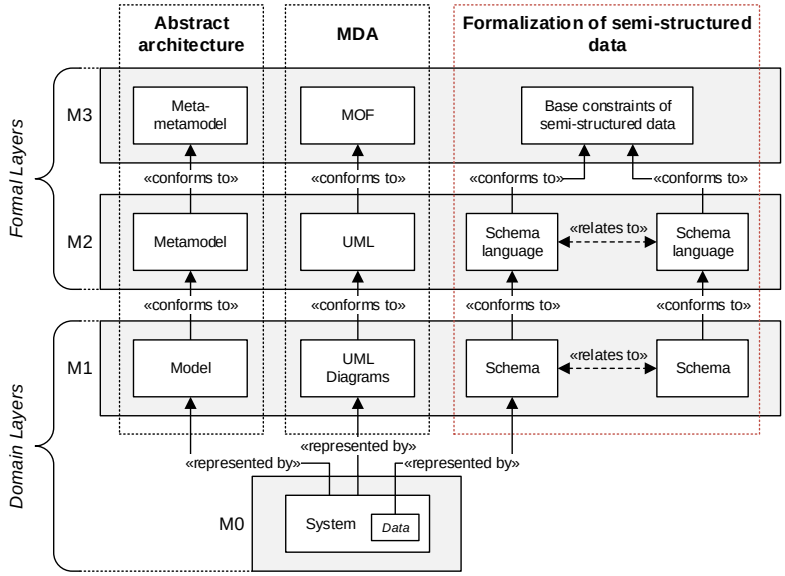


Figure 3.3: Four layer modeling architecture (based on Bézivin 2005, 178; Atkinson & Kühne 2003, 38)

Buneman generically introduces semi-structured data as graph-or-tree-like structures, which conform to an inherent layout. In contrast to structured data, semi-structured data is considered self-describing, which means that the definition of an external schema is not required: The structural and schematic information necessary for processing is embedded within the data (Buneman 1997, 117). However, for processing semi-structured data, external schemata provide benefits e.g. with respect to data validation. Formal schema languages (M2) can be defined based on the basic constraints of semi-structured data (M3). Prominent examples, such as

the W3C XML Schema[13] or JSON schema[14] allow the specification of domain-level schemata (M1)—constraints to which data instances need to conform and can be validated against.

3.3.2 Schema metamodel

Based on the matching-oriented foundation in Zhang et al. (2008, 695-697) and the formal perspective in Murata et al. (2005, 663-665), schemata can be interpreted in terms of finite structures $\langle N, T, R, P \rangle$—a regular-tree grammar with the finite sets of nonterminals (N) and terminals (T), the root symbol ($R \in N$) and the set of production rules (P). Due to the generalization from strings to trees, regular tree grammars allow production rules of the form $n \rightarrow te_c$, where

- $n \in N$,
- $t \in T$ and
- $e_c \subset N$ reflects the content model that is defined over the set of non-terminals.

Based on actual schemata and documents of the arts and humanities such as the TEI and DC examples presented in section 2.1.2, the above definition for schemata is considered to represent the *parsing-oriented view*, which allows an initial validation and processing of external data, but does not necessarily reflect the full extent of the semantic structure and content that is encoded within a document. In the case of the DC example, at least the elements of *creator*, *coverage* and *subject* contained non-atomic content, which could be further decomposed according to additional semantic rules:

- *creator* contains the full name of creators, which could (in the particular example) be resolved to subordinate last and first name elements.
- The content of *coverage* can be split at the '*' character to produce key/value pairs of individual properties
- *subject* is formed as list of subjects, which could be separated at the ';' symbol in order to receive multiple atomic subject elements.

[13] see http://www.w3.org/XML/Schema
[14] see http://json-schema.org/

To facilitate the representation of substructures or alternative elements within the formal definition of a schema, Gradl (2014, 34-37) proposes a *semantic extension*—resulting in the definition of a schema as 6-tuple $S = \langle N, T, R, P, E, F \rangle$, where N, T, R and P form grammatical components and as such the parsing-oriented view as introduced above. The components of L and F provide the semantic extension of the original structure of the schema, where:

- L forms a set of labels and
- F is a set of labeling functions $x \rightarrow le_l$, where:
  - $x \in (N \cup L)$,
  - $l \subseteq L$ and
  - $e_l := \{I, op\}$ defining a function over a set of input values $I \subseteq N$ and an operation of the arity $|I|$.

In order to prevent the introduction of logical cycles, a particular label can be produced by exactly one labeling function. As a result of the domain and co-domain of the labeling functions, sub-trees ar generated for which a root node $x_r \in N$ is selected from the set of non-terminal symbols, any subsequent node $x_s \neq x_r, x_s \in L$ is a label and the edges are constituted by a function $f \in F$.

Gradl (2014) provides further details on the static structure and the extensions of schemata and further introduces a derivation strategy (Gradl 2014, 44-49) for composite schemata and collection-specific adaptions of generic base schemata. In contrast to the concept of labeling functions, which are of particular interest for the concept in this thesis, the aspects of derivation extend the expressiveness of the metamodel and are not within the focus of this thesis.

3.3.3 Mappings

For the creation of unifying views over heterogeneous semi-structured data, mappings between the relevant source schemata and the selected target schema need to be evaluated and applied. Data that is specified in terms of local schemata are transformed into the corresponding representation of the integrative view. Figure 3.4 shows the common understanding of mappings as set of value correlations—i.e. pairs of source and target elements, which have been semi-automatically or manually selected as equivalents.

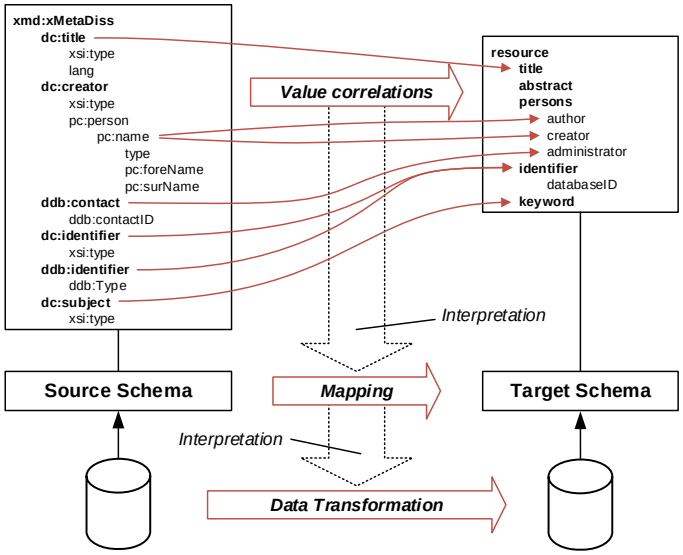


Figure 3.4: Value correlations, mapping and data transformation (based on Leser & Naumann 2007, 125)

Sheth & Larson (1990, 192) defines mappings as functions for the correlation of source and target objects—thus indicating that the associations of individual elements might not suffice to represent related objects. To address the semantic gap between value correlations and semantically associated objects, Gradl (2014, 41-42) introduces the notion of *concept mappings* $cm = \langle E_{S_S}, E_{S_T}, f \rangle$ as correlation between a set of source elements E_{S_S} and a set of target elements E_{S_T}, where:

- $E_{S_S} = \{e_{S_S,i} \dots e_{S_S,j}\} \mid e_{S_S} \in (N_{S_S} \cup L_{S_S})$
- $E_{S_T} = \{e_{S_T,k} \dots e_{S_T,l}\} \mid e_{S_T} \in (N_{S_T} \cup L_{S_T})$
- $f \colon E_{S_S} \to E_{S_T}$

A schema mapping $M_{S_S \rightsquigarrow S_T}$ is then defined as a set of concept mappings $cm_1 \dots cm_n$ defined over a source and target schema S_S and S_T. Aside from the previously introduces labeling functions, the mapping functions f of concept mappings form another use-case for the concept of the data transformation framework in this thesis.

Chapter 4
ANTLR

ANother Tool for Language Recognition (ANTLR)[15] is a parser generator, which creates Java, C# or C++ source code for scanning, tokenizing and parsing input with respect to the constraints of a formal language specification. The ANTLR project has been initiated by Terence Parr[16], a professor of computer science, analytics, and health informatics at the University of San Francisco. The latest stable release[17] of ANTLR has been released on April 6, 2014 under the permissive terms of the revised Berkeley Software Distribution (BSD) license.[18]

As this thesis is primarily oriented on particular use-cases of data transformation, this section provides a broad picture on ANTLR and its role in language applications. Due to the complexity of ANTLR with respect to its implementation and theoretical background, the discussion thereby focuses on aspects that are of particular relevance for the concept of this thesis. More detailed information on ANTLR from a practical perspective are presented in Parr (2013). The theoretical background is thoroughly introduced in Parr & Fisher (2011) and Parr (1993).

[15] http://www.antlr.org
[16] http://parrt.cs.usfca.edu/
[17] v4.2.2: https://github.com/antlr/antlr4/releases
[18] http://opensource.org/licenses/BSD-3-Clause

4.1 Overview

ANTLR is as a generic framework that facilitates the implementation of language applications, i.e. software that reads, parses and transforms input. For data to be processable by means of an ANTLR generated parser, common language patterns of that data need to be identified and specified in terms of a context-free grammar, to which the data is required to conform. Typical use-cases for language applications range from the interpretation of configuration files, the transformation of data between JSON and XML formats to more complex tasks such as the implementation of code compilers.

```
grammar Properties;

file  : prop+ ;
prop  : key '=' value NEWLINE;
key   : ID;
value : STRING
      | NUMBER;

ID      : [a-zA-Z][a-zA-Z0-9]+;
STRING  :  '"' (~'"')* '"' ;
NUMBER  : [0-9]+;
NEWLINE : '\r'? '\n'

WS    : [ \t]+  -> skip;
```

Listing 4.1: Simple ANTLR grammar example

As an initial example, listing 4.1 presents a grammar specified in terms of the Extended Backus–Naur Form (EBNF), a standard notation for the explication of context-free grammars and the required base format of ANTLR: Rules noted in lowercase characters (*file*, *prop*, *key* and *value*) specify syntactical instructions that are utilized by a parser in order to (1) validate a provided input and (2) generate the parse tree. The uppercase rules instruct the lexer on how to tokenize an input character stream and assign one of the lexical categories *ID*, *STRING*, *NUMBER* or *NEWLINE*. The *WS ... -> skip* rule instructs the lexer not to recognize the specified whitespaces as tokens. In this particular case, only concatenations of spaces and tabulators are skipped, as the line-feeds are required as terminal tokens of each property assignment. The presented grammar thus contains all information required for the ANTLR tool to generate the basic language recognition components

for an application that needs to process data against the constraints of a grammar.

Figure 4.1 provides a first overview of the basic principles of the ANTLR framework and the relation between the building blocks of the ANTLR tool and runtime. In addition, this overview also indicates where the boundaries of a generic language support are found and custom domain logic has to be applied to complete a language application as introduced in section 3.2.2. Figure 4.1 shows the two primary packages of ANTLR and provides an overview of their usage within language applications based on the grammar in listing 4.1:

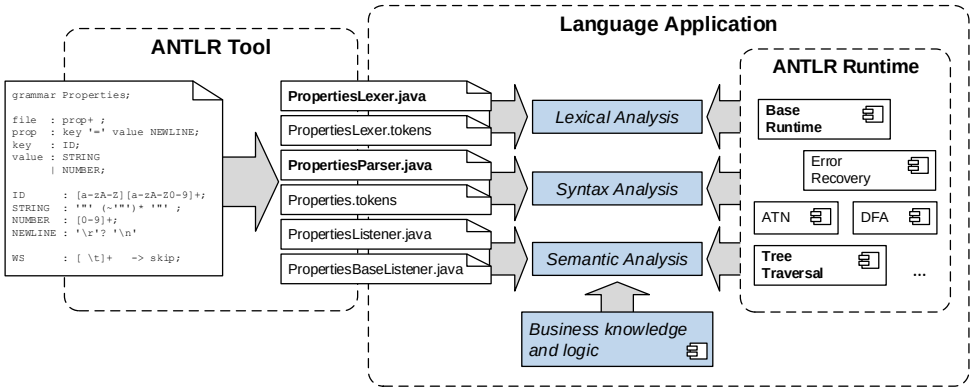


Figure 4.1: Overview of ANTLR components and their usage

- The *ANTLR Tool*[19] primarily contains the functionality to generate the source code of the lexer and parser of a provided grammar. As such, the tool is typically used as standalone application, whereas the generated code for language recognition is incorporated as part of the designed language application.
- The *ANTLR Runtime*[20] contains the runtime support of the framework and comes with a variety of components and buildings blocks, of which some of the most important can be concluded as:
  - *Base runtime* (*org.antlr.v4.runtime*): in comparison to generated lexers and parsers, the abstract base implementations *Lexer* and *Parser* in the runtime package contain the generic functionality necessary for lexical and semantic analysis. In total the base runtime contains 38

[19] see http://mvnrepository.com/artifact/org.antlr/antlr4
[20] see http://mvnrepository.com/artifact/org.antlr/antlr4-runtime

classes including functionality such as *parsing error recovery* as well as sophisticated character and token stream handing.

- *Augmented transition network (ATN)* (*org.antlr.v4.runtime.atn*) and *Deterministic finite automaton (DFA)* (*org.antlr.v4.runtime.dfa*) contain the implementations of the formal aspects of state machines (DFA) and an appropriate, graph-theoretically based operational structure (ATN) to represent a grammatical derivation (Parr & Fisher 2011; Parr 2013, 9-16).
- *Tree traversal* (*org.antlr.v4.runtime.tree*) with interfaces and stub implementations of the visitor and listener patterns assist with the implementation of a semantic analysis. In contrast to the extensive support of the lexical and syntax analysis, however, the semantic analysis needs to be implemented mainly in context of the specific business knowledge and logic.

Among other auxiliary packages, ANTLR primarily consists in a tool, which—based on the formal language specification of a context-free grammar—generates the source code required to lexically and syntactically analyze a provided character input against the underlying grammar. The generated source code is intended to be imported into a domain-specific Java project and to be compiled along with the incorporating application, which is then enabled to validate and process input according to the original grammar. Whereas the tasks of lexical and syntactical analysis as well as the generation of an IR are completely solved by means of the generated parser and lexer classes in combination with the ANTLR runtime, further semantic processing or translation is considered domain-specific and needs to be solved by the application—in the case of this thesis: the concept and implementation of rule framework in chapters 5 and 6.

4.2 Language recognition

The overall process of language recognition in terms of ANTLR is shown in figure 4.2. Based on the exemplary grammar introduced in the previous section (see listing 4.1), the language recognition pipeline is initiated by a lexer, which receives an input character stream. The lexer finishes by producing a token stream, which

is then semantically analyzed by the parser. Following a narrow understanding as e.g. expressed in Crespi-Reghizzi (2009, 5) and Parr (2013, 10-11), the recognition against a formal language specification concludes with the generation of an IR—i.e. in the case of ANTLR a *parse tree*: The separation of language recognition as lexical and syntactical analysis on one hand, and the semantic aspects of language processing—i.e. determining an encoded meaning and reacting appropriately—also facilitates the distinction between generic functionality implemented by the ANTLR framework and domain-specific components of a language processing application.

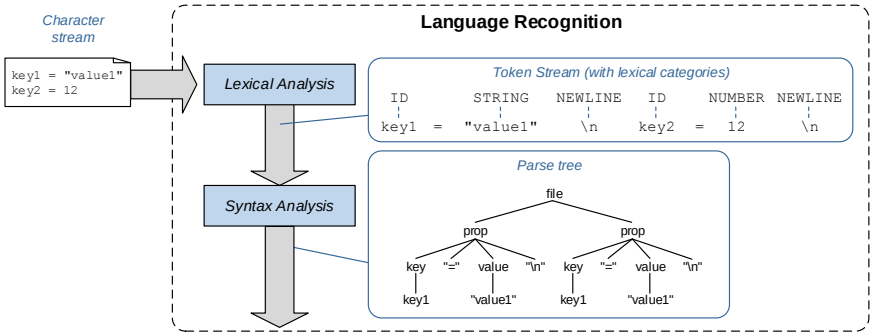


Figure 4.2: Lexical and syntax analysis in language recognition

In the remainder of this chapter, the primary building blocks of language recognition based on ANTLR are introduced as *grammars*, *lexical analysis* and *syntax analysis*.

4.2.1 Grammars

Grammars for ANTLR are based on the EBNF (ISO/IEC 1996), a notation for the formal specification of context-free grammars. In ANTLR, lexer and parser rules can be specified in combined or separated grammar files following the convention that lexer rules start with an uppercase letter and parser rules start with a lowercase letter. The introductory grammar in listing 4.1 contained both types and resulted in the generation of both a lexer and a parser.

ANTLR grammars are based on four fundamental language patterns required to build context-free-languages (ISO/IEC 1996, 3-5; Parr 2013, 62-68): *sequence*,

Syntax	*Usage*
r : …	define rule *r*
r : …| …| …	define rule *r* with alternatives
x	match token or subrule *x*
x y z	match sequence of tokens or subrules
(…| …| …)	subrule with alternatives
x∗	repetition symbol (zero or more occurrences of *x*)
x+	repetition symbol (one or more occurrences of *x*)
x?	optional symbol (zero or one occurrence of *x*)
∽	except symbol (e.g.: ∽*[0-9]* match anything but digits)

Table 4.1: Common notation elements in ANTLR grammars (based on Parr 2013, 67; ISO/IEC 1996, 1-2)

choice, *token dependency* and *nested phrases*. Table 4.1 shows the core notation for the specification of ANTLR grammars.

Sequence The language pattern of sequences addresses syntactical elements which have to occur in a particular order. The *Properties* grammar in listing 4.1 included various sequences—both in lexer and parser rules.

- The grammar rule prop (*prop : key '=' value NEWLINE;*) for example specifies a sequence of *key*, followed by a '=', *value* and finally *NEWLINE*.
- The lexer rule ID (*ID : [a-zA-Z][a-zA-Z0-9]+;*) dictates that a token of the lexical category ID must start with alphabetic character, followed by one or more alphanumeric characters.

Choice Alternatives in languages are specified with the help of the | operator. The introductory *Properties* grammar contained exactly one alternative, indicating that the nonterminal *value* (*value : STRING | NUMBER;*) could be represented by a terminal of the *STRING* or *NUMBER* lexical category.

Token dependency If a particular token must be followed by a counterpart token in the stream, there exists a dependency between these token. An example can be found in array declarations of the form [1, 20, 16, 2], where an opening bracket needs to be answered by its closing counterpart. A matching parser rule

could be specified as *array : '[' NUMBER (',' NUMBER)+ ']'*, also ensuring the proper sequence of commas and NUMBER tokens.

Nested phrases PARR (2013, 65) introduces a nested phrase as a "self-similar language structure"—i.e. a phrase, whose sub-phrases conform to the structure of the parent phrase. Nested phrases require a parser to provide a capacity for recursive rules—an exemplary nested array could be specified as *array : '[' (NUMBER | array) (',' (NUMBER | array))+ ']'*.

Listing 4.2 shows of a complete ANTLR grammar that is based on the four basic language patterns and specifies the lexical and syntactical constraints of JSON.

```
grammar JSON;

json   : object
  | array
  ;

object  : '{' pair (',' pair)* '}'
  | '{' '}' // empty object
  ;

pair   : STRING ':' value ;

array : '[' value (',' value)* ']'
  | '[' ']' // empty array
  ;

value : STRING
  | NUMBER
  | object  // recursion
  | array   // recursion
  | 'true'  // keywords
  | 'false'
  | 'null'
  ;

STRING   : '"' (ESC | ~["\\])* '"' ;

fragment ESC     : '\\' (["\\/bfnrt] | UNICODE) ;
fragment UNICODE  : 'u' HEX HEX HEX HEX ;
fragment HEX     : [0-9a-fA-F] ;

NUMBER  : '-'? INT '.' INT EXP? // 1.35, 1.35E-9, 0.3, -4.5
  | '-'? INT EXP    // 1e10 -3e4
  | '-'? INT    // -3, 45
  ;
```

```
fragment INT : '0' | [1-9] [0-9]* ;// no leading zeros
fragment EXP : [Ee] [+\-]? INT ; // \- since - means "range" inside [...]

WS  : [ \t\n\r]+ -> skip ;
```

Listing 4.2: ANTLR grammar for parsing JSON input

4.2.2 Lexical analysis

In the theoretical context of compiler engineering, the tasks of parsers and lexers are clearly distinguished as the task of a lexer being to "transform a stream of characters into a stream of words in the input language" (Cooper 2012, 25), while the task of a parser consists in determining whether a "stream of classified words produced by the scanner is a valid sentence in the programming language" (Cooper 2012, 83). In practice, the distinction between lexical and syntactical analysis is not as exact because—depending on the particular language—e.g. strings that are considered legal sentences of a language might be irrelevant for parsing or individual characters could be relevant for parser rules an thus considered as tokens.

For this reason, the line between lexers and parsers in ANTLR is more of a logical character than an enforced distinction and lexer rules in ANTLR can instruct more complex tasks than accumulating characters to words. In general, an ANTLR-generated *lexer* transforms a character stream into a stream of tokens as specified in terms of lexer rules of the underlying grammar. Whereas a valid and simple implementation of such lexer rules could be based on whitespace- and punctuation-separated characters to words, the lexers in ANTLR can be perform sophisticated analysis—thus possibly reducing the required complexity of parsers.

4.2.3 Syntactical analysis

ANTLR generated parsers follow the LL(*) parsing paradigm introduced in Parr & Fisher (2011). In general, LL(*) parsers follow the top-down principles of LL(k) parsing by (1) processing provided input from left-to-right—constructing the left-most derivation. In contrast to the most common form of LL parsers with one lookahead token (Cooper 2012, 95), ANTLR parsers allow a lookahead until the

end of the token stream and as such the predictive parsing of input—an extension to DFA-based parsing: Whereas the latter transitions from state to state based of the next token, predictive parsing allows decisions based on the next k tokens and allows the specification of *recursive rules* (like *expr : expr '+' expr;* in a calculator application). Internally, ANTLR works both with DFA and ATN. Parsing is initiated by building paths in a DFA attempting to proceed with 1 lookahead token. Upon detecting ambiguities in the grammar when applying DFA, ANTLR switches to the LL(*) parsing strategy, looking forward in the token stream until the ambiguity can be resolved.

As the complex features of ANTLR parsers are best described in the context of a practical parsing problem, many of the features are discussed in the concept and evaluation chapters of this thesis, which will however still not utilize the full capabilities of ANTLR e.g. with respect to the error detection and recovery mechanisms, semantic predicates and embedded actions. For an in-depth discussion of the theoretical background of the LL(*) parsing strategy please see Parr & Fisher (2011), for an introduction to additional functionalities provided by ANTLR see Parr (2013).

4.3 Tree processing

Whereas the primary concern of language recognition consists in the identification of valid sentences within a provided character stream, language applications need to process the parser-generated, intermediate representations to execute necessary actions and generate domain-oriented functionality. In compilers, such processing is typically associated with the functionalities of optimizers and compilation back ends. Abstracting from the specifics of source code compilation, intermediate representations of processed input have been completely validated against the specifications of a language and are presented in a decomposed and structured form for easier subsequent processing.

The representation produced by an ANTLR parser obtains the form of a parse tree—a finite, directed tree, where each parsed grammatical symbol is represented by a node and applied production rules are captured by the edges of the tree. As such, a parse trees contain the complete information of the grammatical derivation

for a provided input (Cooper 2012, 226)—at the expense of the resources of the execution system, which needs to allocate memory for each of the nodes and edges.

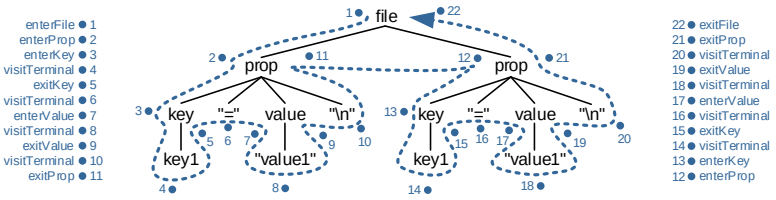


Figure 4.3: Tree traversal on the base of the listener pattern

Based on parse-trees generated by ANTLR parsers, further processing of provided data is facilitated as the unstructured input has been lexically and syntactically analyzed, validated against a defined language specification and transformed into a structured, intermediate representation. Parse-tree processing can be specifically implemented with respect to the required functionality of a domain. However, ANTLR provides *listeners* and *visitors* as tree-walking mechanisms (Parr 2013, 17-19):

- With the help of the *listener* pattern (see figure 4.3), parse trees are completely walked—notifying any attached listener about entering and exiting nonterminal nodes and visiting terminal nodes. The ANTLR generated base listener implementation defines method stubs for every possible *enter...*, *exit...* and *visit...* event of the grammar, which can be implemented as needed by the language application.
- The *listener* pattern defines visit...methods for every nonterminal and terminal node of the grammar. Whereas in the listener pattern, parse-trees are automatically traversed, visitor implementations need to explicitly call subordinate *visit...* methods to perform the traversal—thus allowing potentially more efficient implementations of the tree traversal.

For the conceptual work and especially the implementation in this thesis, the visitor pattern is applied because of its reduced algorithmic overhead. For further development of the rule framework in terms of a productive application, the runtime benefits of the visitor pattern should to be evaluated on the basis of real-world use cases.

5 Chapter
Concept

Based on the understanding of language applications as programs that react on the basis of input sentences, languages can be found in various formats, protocols and files—i.e. any input that conforms to formal language specifications in terms of grammars and thus can be processed by a recognition component of a language application. With the following concept, this thesis focuses on the application of the language theoretical foundation on the task of integrating research data stored in distributed and heterogeneous digital collections.

Various transformation languages such as the Query/View/Transformation (QVT) and ATLAS Transformation Language (ATL) have been designed to allow the generic specification of model transformation functions. Other transformation languages can be identified in the Extensible Stylesheet Language Transformations (XSLT), the standard language for XML data transformation, AWK, a generic text processing language or Perl, an interpreted, general-purpose programming language. The main characteristic of these transformation languages consists in their expressiveness and complexity, which are required e.g. for model transformations in the context of software engineering or the Extract, transform, load (ETL) process in data warehouses. In the particular context of the arts and humanities, data definition and the design of transformation rules are both tasks, which cannot be performed—as in typical integrative scenarios—for the whole application domain in an extensive analytical phase, but requires a stepwise, continuous explication of knowledge about collections and their data. For reasons of the ease-of-use and

the acceptance of a developed system, domain experts of the arts and humanities should not be required to learn complex general-purpose languages to order to be able to describe the structure of data or a concept mapping function.

For this reason, the presented concept focuses on the creation of an rule framework, which (1) allows a generic transformation of data provided by digital collections and (2) reduces the complexity of the specification of functions for data transformation. After a discussion of the domain-specific problems in section 5.1, an overview of data processing in terms of the rule framework is presented in 5.2. The so called data processing pipeline shows the fundamental idea of a classification of the rule framework into two phases: the *data description*, which is detailed in section 5.3 and the *data transformation* phase, which is the focus of section 5.4. The conceptual work is concluded by the summary of the designed components and their interrelation in section 5.5.

5.1 Problem definition

The primary objective of this thesis consists in the design of a framework, which facilitates the specification and application of data transformation rules for research data of the arts and humanities. The discussion of the context in chapter 2 and the conceptual frame of the modeling architecture in section 3.3 allow the derivation of integration problems, which are intended to be solved in terms of the rule framework.

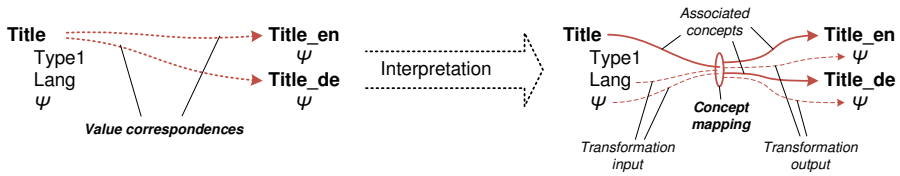
The following discussion not only provides an overview of these problems, but especially shows the similarities between the use-cases—thereby facilitating the conception of the rule framework as a reusable solution.

5.1.1 Labeling vs. mapping functions

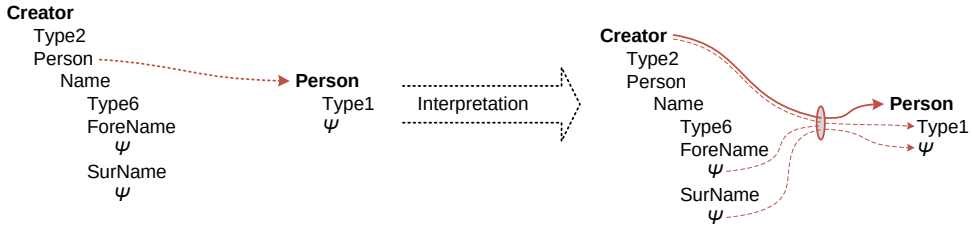
With respect to the metamodel developed in Gradl (2014), which forms the formal framework for the representation of schemata and mappings[21] in this thesis, two primary application classes of the rule framework can be distinguished:

[21] see also section 3.3

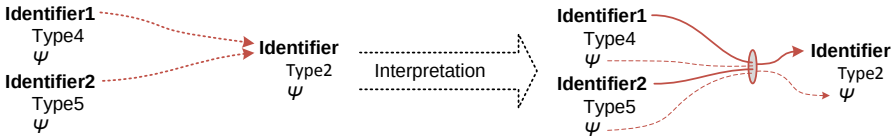
- A *mapping function* $E_{S_S} \rightarrow E_{S_T}$ transforms a set of elements of the source schema S_S into semantically equivalent elements of the target schema S_T (Gradl 2014, 42).
- A *labeling function* $x \rightarrow le_l$ produces a label l from a nonterminal node or label $x \in (N \cup L)$ (Gradl 2014, 35).



(a) 1:N Concept Mapping



(b) 1:1 Concept Mapping



(c) N:1 Concept Mapping

Figure 5.1: Concept mapping examples (Gradl 2014, 44)

A mapping function is assigned to a *concept mapping*, which forms a logical construct to consolidate value correspondences between the sets of source and target elements that describe a particular semantic concept. Figure 5.1 illustrate the idea of concept mappings based on three examples.

a) The title of a document is represented by an element *Title* in the source schema—the language of the title is determined by a subordinate element. In the target schema the distinction between languages is reflected in the name of the elements, thus resulting in the distinctive concepts of *Title_en* and *Title_de*.

b) Despite the intuitive association of the *Person* element in both schemata, the generation of the target element depends on the parent *Creator* source element for specifying the *Type1* sub-element. For this reason, *Creator* and *Person* are the semantically equivalent concepts.

c) Two different identifier elements of the source schema are mapped to the one existing target element. The mapping function would need to either select the appropriate identifier based on the instance (e.g. use *Identifier1* if not empty, otherwise use *Identifier1*) or generate two target *Identifier* elements.

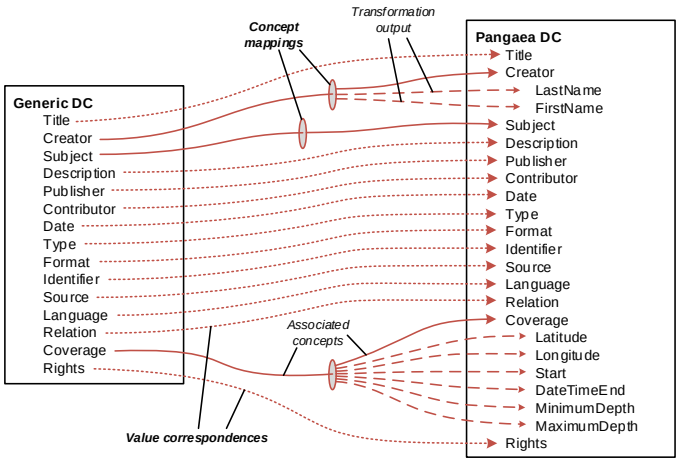


Figure 5.2: Labeling vs. mapping functions example

Whereas mapping functions contain instructions to transform data from a source to a specified target representation, labeling functions operate within one particular schema with the purpose to semantically enrich original data based on the specifications of domain experts. According to the above definition adapted from Gradl (2014, 35), labeling functions can produce subtrees, with only the roots $x_r \in N$ selected from nonterminal elements of the original regular-tree grammar. Labeling functions can be interpreted as transformation functions between the static, parsing-oriented structure of a schema (the regular-tree-grammar component, see section 3.3.2) and its enriched version incorporating the semantic extension. Figure 5.2 illustrates this interpretation based on an exemplary extension of the DC schema in its context-specific usage by Pangaea.

In conclusion, the use-cases and requirements on the data transformation framework can be summarized as *generic support of data transformation functions* of the form $E_S \rightarrow E_T$, where E_S forms the set of source elements and E_T the set of target elements. In order to support both function types, the sets can be part of the same or different schemata. If $E_T, E_S \subseteq S$ are part of the same schema, $E_S \cap E_T = \emptyset$ in order to prevent circular dependencies.

5.1.2 Data vs. query integration

The developed concept needs to address the integration of heterogeneous data by the generation of unifying views, which are utilized to harmonize data. The preparatory work in Gradl (2014) distinguished between *materialized* and *virtual* forms of data integration (Poulovassilis 2009, 586) and identified the latter as preferable for integrative use-cases within the domain of the arts and humanities. The primary arguments for the preference of the virtual form, which can also be referred to as *data federation*, can be summarized as Gradl (2014, 2, 40):

- *Data preservation*: In order to be considered citable in academic contexts, data needs to remain in its original and genuine form.
- *Semantic flexibility*: The suitability of unifying perspectives on data depend on the academic context of the federated collections and the scholar. To prevent an information loss induced by traditional global schema approaches (Lenzerini 2002), data needs to be dynamically federated with respect to the particular context.

The support for virtual data federation resulted in the creation of the integration metamodel, which has been introduced in 3.3 and—by separating data integration into a technical and context-dependent problems—allows the abstraction from the technical aspects (e.g. protocol and encoding heterogeneity) and a focus on those aspects, which require the interaction with domain experts. In addition, the developed metamodel does not prevent materialized integration as it might be beneficial in particular use-cases—e.g. when long-running analyses are executed on data.

Data integration addresses such use-cases, which require the combination of data from different sources for a unified processing or for the migration of data

into new or external systems. The implementation of use-cases that require a search within heterogeneous data depends on a unified view only for the time that is required to execute a query. As identified in section 2.2, the formulation and execution of queries depends on the set of relevant data sources, the set of schemata used by these data sources as well as the coherence of these schemata in terms of semantic associations—the concept mappings. Queries thus cannot be executed on a harmonized index, but require a virtual integration of data, which—in order to provide a convenient user experience—has to be performed in terms of milliseconds.

For the concept of the rule framework, a query can be understood as a semi-structured document for which similarity measures can be applied in order to find documents. This interpretation correlates with the basic ideas of some classic information retrieval models, such as the vector space model: A promising approach based on structure-aware indexing and the consideration of both structure and query terms as dimensions of the vector space can be found in Liu et al. (2004). However, no particular implications on the applicability of specific retrieval models need to be drawn. Essentially, a query on semi-structured data can be considered as a set of partial queries, which each target the content of one or more nodes of the document tree and need to be combined in order to produce the overall query.

5.2 Language processing overview

In software development, *general purpose* programming languages allow developers to implement applications that are specifically targeted at *particular use-cases and users*. Although requirements and circumstances of software development projects vary and are typically gathered within dedicated analysis and design phases, the expressiveness of programming languages such as Java or PHP facilitates the realization of supporting systems. Adapting this principle to the task of data transformation in the context of this thesis, the rule framework can be based on the assumption that rules for data integration can be formulated in terms of a general-purpose transformation language if that language satisfies requirements for expressiveness.

However, as discussed in the previous section, with an increasing complexity of

a data transformation language, the technical knowledge required to formulate appropriate transformation rules increases as well—thus possibly reducing the acceptance of the developed concept and system. In order to reduce the complexity of data transformation function specifications, the presented concept is based on the hypothesis that an appropriate description and decomposition of transformation input reduces the expressiveness required of the transformation language. For this reason, the phase of data description is identified as an additional step, which allows the description of data in terms of a DSL. This data description is considered optional: atomic content cannot be decomposed and is therefore an immediate input to transformation functions. However, the following concept often assumes the existence of data descriptions in order to conceptualize the rule framework with particular respect to this more complex cases.

Instead of requiring the description of specific details in terms of a general-purpose language, the domain experts are enabled to describe data in terms of the definition of DSLs. Such meta-programming facilities (Ghosh 2011, 118) allow users to be more productive within their specific domain because the complexity of the environment can be encapsulated: An expert of a particular discipline or collection is enabled to express the information, which is required to describe data, the syntax and semantics, in terms of a technology-agnostic notation: context-free-grammars in EBNF.

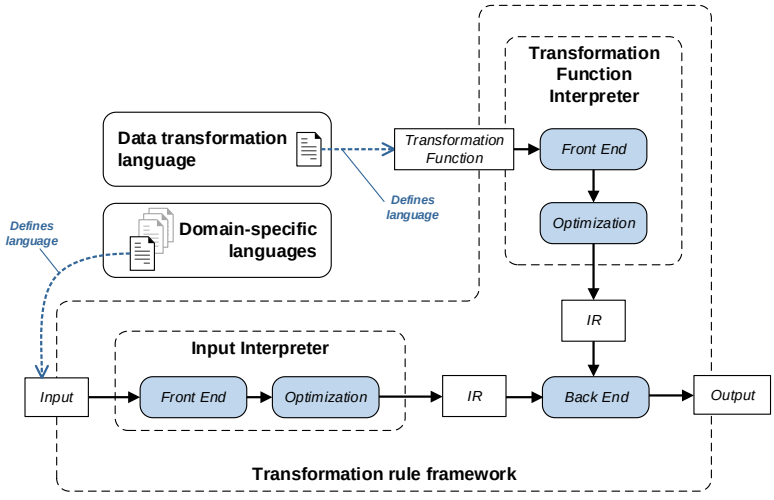


Figure 5.3: Overview of the transformation rule framework

Figure 5.3 shows the overall concept of the transformation rule framework, which forms a combination of the application of DSLs for the description of data and a transformation language for the formulation of transformation rules. The building blocks within the framework are summarized in two interpreters, one for the validation and processing of input against a DSL and one for the validation and processing of a transformation function against the transformation language. Both are conceptualized as autonomous language application components accomplishing the following tasks (Parr 2010, 22-26):

- *Process input* utilizing grammatical rules—identifying tokens and token types in terms of a *lexical analysis*.
- *Construct an IR* as a result of the *syntax analysis*.
- *Traverse the IR* to extract information or perform transformations to rewrite the IR according to the needs of the concluding back end (*semantic analysis* or *optimization*).

In contrast to the typical structure of compilers, the back end of the rule framework needs to combine two parse trees, one with encoded transformation instructions and the other with a decomposed and structured version of the input. The back end finally concludes individual transformation tasks by:

- *interpreting input sentences* executing the encoded transformation instructions and
- *generating the output* as output tree that forms a representation of the input in a target structure (mapping function) or that is included under the input element in the same structure (labeling function).

In summary, two separate language applications are combined in the rule framework in order to reduce the overall complexity for the application domain: The specification of the data description language as well as the formulation of operations to transform data require an in-depth knowledge of the collection and hence the input of a domain expert, but abstract from technical aspects of the execution. The specification of the data transformation language, however, can be considered primarily as technical task and requires a software development background in order to implement the operations required to transform the data.

5.3 Data description

The specification of data with respect to syntactical and semantic constraints is accomplished by means of element-specific DSLs. The identification of language patterns is considered as domain specific task, which requires knowledge about the particular context of the data: the collection, discipline and usage. As a consequence, a goal of the rule framework consists in the specifiability of languages by domain experts.

After an overview of the fundamental idea for the specification and application of DSLs in terms of the rule framework in section 5.3.1, the specification of the required runtime behavior of the data description component in 5.3.2 provides conceptual details for the implementation of the framework.

5.3.1 Conceptual basics

The discussion of the context in chapter 2 introduced the heterogeneity of the research data and digital collections as well as the dynamic integration use-cases as primary characteristics of the application domain, which prevent the utilization of traditional data integration approaches based on system-wide global schemata. As a consequence, the semantic associations between schemata are specified in terms of direct mappings between source- and target schemata, which allow an immediate relation of concept representations in the schemata.

Assuming normalized schemata, the content of any terminal node of the parsing-oriented view on schemata (N, T, R, P) contains atomic content and data integration could rely on semantic associations and operations on the M1 modeling layer (see figure 3.3 on page 24): Considering e.g. atomic values in a field *creator_last_name*, which are associated to the atomic *author_LName* field in a target schema, the task of integration consists in placing the atomic content specified in terms of the source schema into an instance that conforms to constraints of the target schema. Such schema-integration scenarios are common in structured data integration (see e.g. Sheth & Larson 1990, 218-220).

With respect to semi-structured data—and in particular the research data of the arts and humanities—the assumption of atomic terminal nodes does not hold: As the Pangaea DC and the DTA TEI examples in section 2.1.2 showed, substructures or

unstructured content is often encoded within the elements of semi-structured data. Whereas the specification of DC contains no restriction of the content of its 15 core elements (compare DCMI 2012), the TEI guidelines specifically define the *text* element as container for digitized or digital texts: "A full TEI document combines metadata describing it, represented by a <teiHeader> element, with the document itself, represented by a <text> element" (Burnard & Bauman 2014, 150). In order to utilize explicable semantic on non-atomic content, Gradl (2014) introduced the extension of the static, parsing-oriented perspective on schemata by a semantic extension, which produces subtrees under defined nonterminal elements—based on labeling functions that are executed on instances.

5.3.1.1 Instance-level perspective

In contrast to the schema-level, element values of semi-structured data do not necessarily follow a defined model and can occur in terms of atomic values, semi-structured or unstructured text. In addition to a missing formal definition on the schema-level, the collection- and domain-specific notation and interpretation of data prevents the definition of a common instance-based language for the description of data. The reasoning is thereby similar to the argument against the definition of static, global schemata (see section 2.1.2): to fulfill the requirements for expressiveness and flexibility as needed for research-specific use-cases, the grammar of such a unifying language would need to be either

- *too complex* in order to include all possible syntactic patterns and results in a large language specification, in which problems such as the ambiguity of grammatical rules can hardly be identified, or
- *abstracting* from specific syntactical aspects to control the complexity of the language and potentially result in insufficient level of expressiveness for specific use-cases of the language.

Aside from the expressiveness and customization requirements from research use-cases, the specification of the syntax and semantics of data—beyond the schema-level constraints—significantly depends on the particular objectives of a language designer:

- *Collection orientation* as intuitive motive of data description focuses on an

explication of knowledge about collection-specifics of data in general. The resulting description is generic in a sense that it is not primarily influenced by particular data integration use-case, but allows the reuse of the description for multiple tasks and mappings.

- *Task orientation*: When processing complex data such as natural language, concrete tasks facilitate the identification of relevant syntactical and semantic patterns and produce less ambiguity for further processing of that data. A particular task on natural texts could be found in the detection of named entities, which benefits from a task-oriented description of data.
- *Mapping orientation* is expected to be the main focus if data needs to be integrated under a target schema. If the target schema for a particular use-case is known, mapping-oriented descriptions facilitate a virtual or materialized integration under that schema.

5.3.1.2 Derivation of an example

In favor of a domain-motivated example for the specification of a language in terms of ANTLR grammars, this thesis to this point did not provide an extensive illustration of a generic grammar. In this section, the derivation of lexical and syntactic patterns of a particular DSL is presented. To improve understandability and to indicate that specific grammars could be designed by domain experts, the following steps document the definition of the *PangaeaCoverage* grammar, which is part of the use-case further detailed in section 7.1. An exemplary content of the *coverage* element of a Pangaea DC dataset is presented in listing 5.1.

```
<dc:coverage>
LATITUDE: −70.339167 * LONGITUDE: −11.656833 * DATE/TIME START: 1988−02−25
    T17:49:00 * DATE/TIME END: 1988−02−25T17:49:00 * MINIMUM DEPTH,
    sediment/rock: 0.0 m * MAXIMUM DEPTH, sediment/rock: 10.2 m
</dc:coverage>
```

Listing 5.1: Pangaea coverage example input

As an inital step, the basic language structure could be implemented in a grammar as shown in 5.2:

- The substructure (*substruct*) of the *coverage* is composed of at least one subelement (*subelem*)

```
grammar PangaeaCoverage;

substruct :   subelem+;

subelem   : key ': ' value '*';
key       : ID;
value     : ID;

WS  : [ \t\r\n]+ -> skip;
ID  : ~(':'|'*')+;
```

Listing 5.2: Pangaea coverage grammar – step 1

- The *subelem* rule identifies a key/value-pattern.
- Both key and value of a subelement are of the lexical category ID
- IDs are defined to be any character everything but ':' and '*'—the separating characters.
- Whitespaces are not removed from IDs because they are part of keys and values (e.g. *10.2 m*), they are however removed between tokens as specified by the *WS* lexer rule.

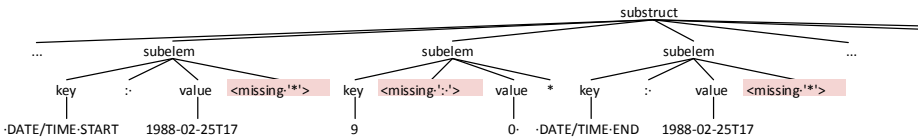


Figure 5.4: Parse tree of the Pangaea Coverage grammar (step 1)

Figure 5.4 shows the resulting parse tree when recognizing the exemplary coverage in listing 5.1 against the defined grammar—showing that the grammar resulted in parser errors starting short after the *DATE/TIME START* terminal node.

The grammar in listing 5.3 corrects the erroneous first version by composing the parser rule *data* as three ID tokens separated by colons—the symbol, which is also used as key/value separator. Assigning the *date* alternative before *ID* in the *value* rule, *date* receives a higher precedence and is recognized before an *ID*. The parse tree in figure 5.5 shows that one parse error remains because the last element within the substructure of the *coverage* element does not end with an asterisk. The grammar in listing 5.4 removes the error by allowing an alternative

```
grammar PangaeaCoverage;

substruct :   subelem+;

subelem   : key ': ' value '*';
key       : ID;
value     : date
          | ID;

date      : ID ':' ID ':' ID;


WS  : [ \t\r\n]+ -> skip;
ID  : ~(':'|'*')+;
```

Listing 5.3: Pangaea coverage grammar – step 2

within the *subelem* rule without the terminating '*' symbol.

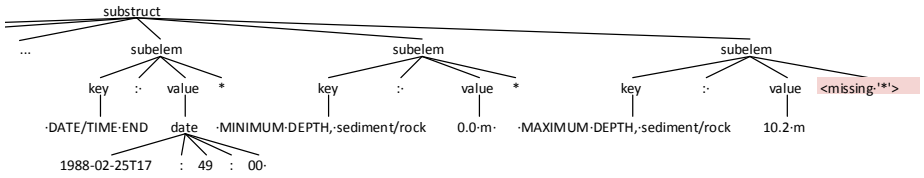


Figure 5.5: Parse tree of the Pangaea Coverage grammar (step 2)

Although the presented grammar validly parses the provided input, further modifications can be performed to further improve the language specification and the generated IR:

- *Named parser rules*: The sub-elements are explicitly labeled according to their content, which facilitates the identification of input parameters by consuming data transformation rules.
- *Irrelevant alternative*: The additional otherElem rule catches unspecified and key/value-pairs, which is required for a language specification, if other keys are presented in different instances.
- *Date lexer rule*: The date is now correctly processed at the lexer level.
- *Whitespaces in IDs*: the lexer rule for IDs removes whitespace from the start and end of the token.

```
grammar PangaeaCoverage;

substruct :   subelem+;

subelem   : key ': ' value '*'
      | key ': ' value;

key     : ID;
value   : date
      | ID;

date    : ID ':' ID ':' ID;


WS  : [ \t\r\n]+ -> skip;
ID  : ~(':'|'*')+;
```

Listing 5.4: Pangaea coverage grammar – step 3

The grammar that results from the above modifications is shown in listing 5.5, the parse tree in figure 5.6.

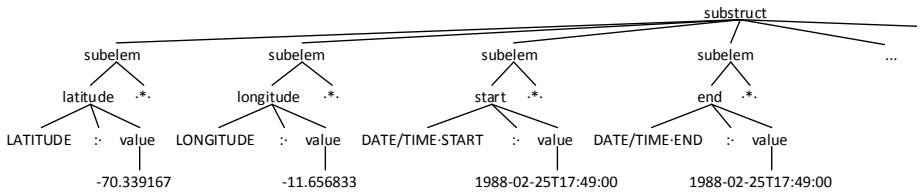


Figure 5.6: Parse tree of the Pangaea Coverage grammar (step 4)

5.3.2 Runtime behavior

On the formal foundation of language specifications in terms of context-free grammars, the rule framework processes input data according to the conditions and requirements expressed by domain experts. As a result of the discussed specificity and diversity of the research data in digital collections, the data description component of the designed rule framework is required to process multiple languages in a generic and extensible fashion.

In order to be able to parse, interpret and process data according to specified rules, intermediate tasks need to be accomplished as shown in figure 5.7.

```
grammar PangaeaCoverage;

substruct :   subelem+;

subelem   : (longitude | latitude | start | end | minDepth | maxDepth |
    otherElem) SEPARATOR?;


longitude : 'LONGITUDE' ': ' value;
latitude  : 'LATITUDE' ': ' value;
start     : 'DATE/TIME START' ': ' value;
end       : 'DATE/TIME END' ': ' value;
minDepth  : 'MINIMUM DEPTH, sediment/rock' ': ' value;
maxDepth  : 'MAXIMUM DEPTH, sediment/rock' ': ' value;

otherElem : key ': ' value;

key       :   ID;
value     : DATE
      | ID;

ID  : ~(' '|':'|'*') ~(':'|'*')+ ~(' '|':'|'*');
DATE  : YEAR '-' MONTH '-' DAY 'T' HOUR ':' MIN ':' SEC;
SEPARATOR : ' '? '*' ' '?;

fragment  YEAR  : [1-2][0-9][0-9][0-9];
fragment  MONTH : [0-1][0-9];
fragment  DAY   : [0-3][0-9];
fragment  HOUR  : [0-2][0-9];
fragment  MIN   : [0-6][0-9];
fragment  SEC   : [0-6][0-9];

WS  : [ \t\r\n]+ -> skip ;
```

Listing 5.5: Pangaea coverage grammar – step 4

5.3.2.1 Parser & Lexer generation

The implementation of language application font-ends consists in the creation of components for the *lexical and syntactical analysis* of sentences with respect to the implemented language. Based on such lexers and parsers, (1) the syntax and semantics of a provided input can be validated and (2) an intermediate representation is created to facilitate further processing of the input.

The introductory discussion of the features and principles of the ANTLR framework in chapter 4 distinguished compile-time (parser generation) and run-time

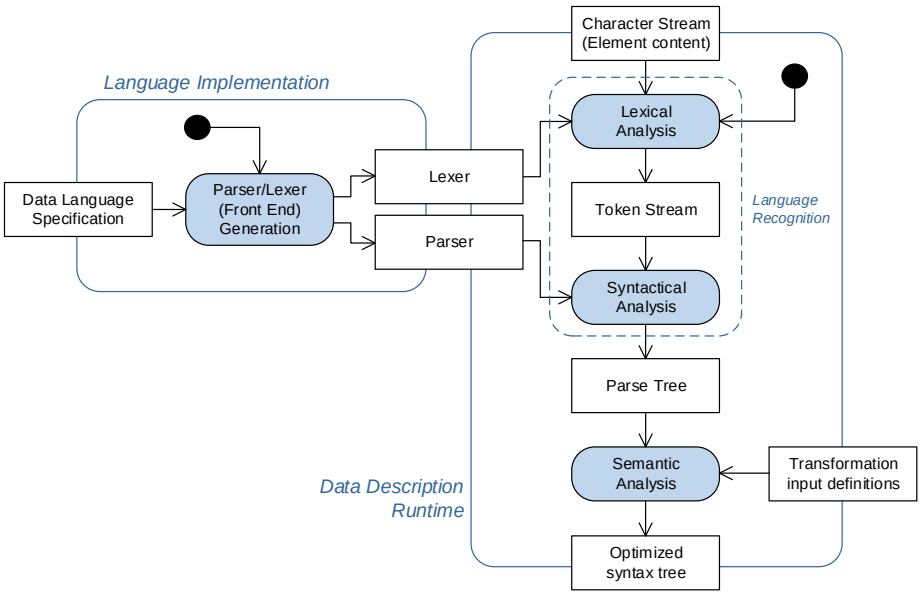


Figure 5.7: Data description functionality overview

(lexing and parsing) components of ANTLR, which are—in contrast to the typical usage—both employed during the execution of the rule framework. After a grammar has been designed and published to the rule framework, a two-phase compilation process is initiated:

1. The ANTLR tool is used to create the Java source code of the lexer and parser for the designed language. If the process fails, the user is informed and provided with hints to complete the grammar.
2. With the help of the Java compiler[22], the generated source code is compiled to executable Java byte-code.

This compilation process is executed from within the rule framework. Immediately after the process, the language recognition logic becomes available for data translation: compiled class files are therefore not saved to the default classpath of the rule framework, but to an external directory. A file-based classloader is utilized to load the classes as needed, leading to two main benefits:

- Data description logic is loaded (and potentially unloaded) on demand. This reduces both the memory footprint of the rule framework and increases

[22] see http://docs.oracle.com/javase/7/docs/api/javax/tools/JavaCompiler.html

startup time.
- Without the need to restart the rule framework, parser and lexer classes can be loaded immediately after their generation.

Compared to typical language processing applications as discussed in Parr (2013, 83-108) and Cooper (2012), the data description component of the designed rule framework is required to expose a *dynamic functionality* with respect to the handling of language specifications:

Dynamic code integration In general, compiler or parser generators create language recognition code from a language specification. In the particular example of ANTLR, lexical and syntactical constraints of a language are collected in terms of text files[23]. Upon execution of the ANTLR tool, provided grammar files are parsed and processed to generate the source code of the lexer, parser and auxiliary classes—or resulting in the output of errors that occurred upon grammar processing.

The standard method of incorporating the ANTLR-produced source code consists in the import of the source files into a software project and compiling the language recognition functionality along with the application. Despite the intuitive character of the standard approach, the rule framework is required to be able to dynamically integrate new language interpretation functionality without requiring a restart or recompilation of the software.

Language modifiability One particular reason for the dynamic code integration consists in the necessity to handle a larger set of languages. Whereas the traditional method is satisfactory for implementations of a small set of languages that are not subject to frequent changes, the developed rule framework intends to facilitate the creation and adaption of domain-specific languages. For this reason an implementation is required, which—at the runtime of the rule framework (1) generates Java source code from language specifications, (2) compiles the source code files and (3) loads the compiled classes.

As a negative side-effect of the language modifiability requirement and the resulting dynamic integration of language recognition code, the implementation of the generated base ANTLR visitor and listener interfaces are prevented: In typical

[23] ANTLR grammar file extension: *.g4*

language processing scenarios based on ANTLR, the implementation of a language is concluded by implementing the generated base visitor or listener interfaces. Implementing these interfaces, the traversal of the tree by means of the provided ANTLR functionality results in calls to exit and enter methods for the specific nodes. By separating the logic of data description and data transformation, this side-effect is, however, rendered irrelevant as section 5.4 will show.

On-the-fly applicability Language specifications are expected to be developed by experts of a particular academic domain or digital collection. In order to provide a convenient user experience, graphical interfaces need to be designed and implemented with particular respect to usability considerations.

Particularly if these interfaces are provided in terms of a web-based application, users should receive immediate feedback on a provided grammar—without the need to restart the application hosting the rule framework. Instead, the grammar should be executable e.g. in terms of provided sample data—indicating the applicability and eventual shortcomings of the specification.

5.3.2.2 Language application execution

Although the overall goal of the rule framework is completed by combining the data description and data transformation component, the phase of data description alone forms an autonomous language processing application with defined input and output parameters:

- Any stream of characters can be provided as *input* to the data description component, which (1) validates if the input can be processed in terms of a specified language and (2) renders the input in terms of a parse tree.
- The output of the language application consists in a modified tree, which has been optimized with respect to the specified semantics of a consuming functionality—in the case of this thesis: the data transformation component.

Following the terminology of Parr (2013) and Cooper (2012), the functionality of the data description runtime can be classified in the phase of *language recognition*, containing the lexical and syntactical analysis of an input and the subsequent *semantic analysis*, generating an intermediate representation with respect to external

specifications.

With the help of the ANTLR framework, the lexical and syntactical analysis of unstructured content can be facilitated as shown in the example in section 5.3.1.2. As the rules for the analysis are based on an identification and expression of language patterns by domain experts, produced parse trees encapsulate additional information, which can be exploited by data transformation.

Primarily to facilitate the further processing in terms of data transformation, but also in order to reduce the memory footprint and processing load of the rule framework, an additional task of *semantic analysis* is introduced to the process of data description—creating a variation of the parse tree—according to the needs of the subsequent transformation functions. As further detailed in the section on data transformation in section 5.4, transformation functions refer to input parameters with the '@'-symbol followed by a element-rooted selector. To retrieve the longitude value in the Pangaea coverage parse tree in figure 5.6, *@substruct.subelem.longitude.value* is specified in the data transformation function. Semantic analysis in this context can be summarized as receiving the input parameter specifications of all data transformation functions, which are executed after the data description process executes and to use this information in order to generate a AST that is—in order to simplify the selection in data transformation—optimized with respect to the label-based selection of the nodes.

ANTLR parsers are intended to be traversed by listeners or visitors, which implement the specifically generated interfaces in order to make sense of grammar rules. Implementing the base class of the *org.antlr.v4.runtime.tree.ParseTreeListener*, a generic listener implementation can react only to four generic events by implementing the following four base methods:

- *void visitTerminal(TerminalNode node)*: Called by the parser when a leaf of the parse tree is walked.
- *void visitErrorNode(ErrorNode node)*: If the parser was able to recover from errors in the input, a parse tree that—among regular terminal and nonterminal nodes—contains error nodes is generated. The parser calls this method when walking such error nodes.
- *void enterEveryRule(ParserRuleContext ctx)*: Called when the parser arrived at any nonterminal, non-error node and before any of its child nodes are

walked.

- *void exitEveryRule(ParserRuleContext ctx)*: Like the enterEveryRule with the exception that the method is called after child nodes have been walked.

Figure 5.8 shows the runtime processing within the data description phase, which combines the information of the parse tree with the hierarchical input parameter definitions of depending data transformation functions in order to produce an AST that can be considered as *dense*, *convenient* and *meaningful* with respect to further processing (Parr 2010, 77).

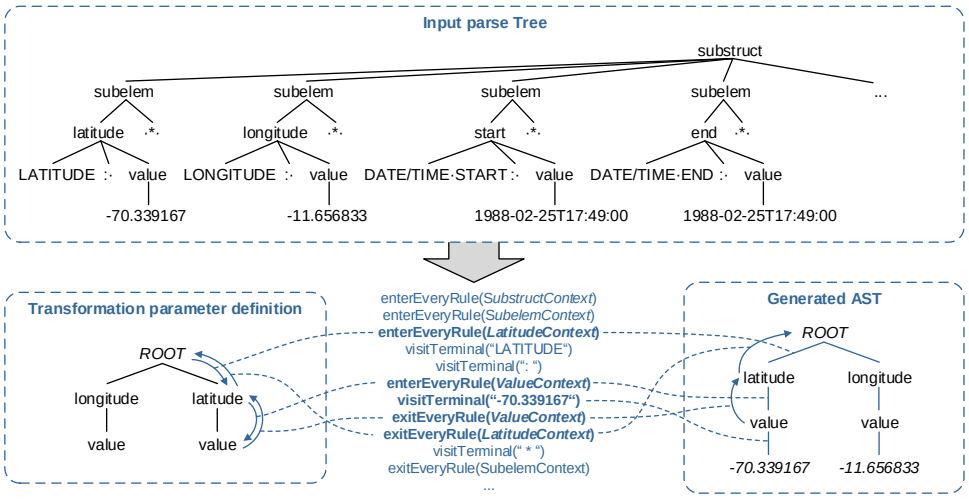


Figure 5.8: Data description runtime processing

The semantic transformation of the input parse tree to the optimized AST involves three steps upon the traversal of the parse tree:

1. enterEveryRule(-): compare the provided nonterminal node (called context) with the child nodes of the current root of the transformation parameter definition tree. If the context matches the name of a child input parameter, create a nonterminal node and push it to the current root of the AST.
2. exitEveryRule(-): if the provided context matches the currently regarded parent, move up both in the AST and the transformation parameter definition hierarchy.
3. visitTerminal(-): create a terminal node and assign it to the topmost nonterminal.

5.4 Data transformation

As discussed in the previous section, the definition of syntactical and semantic features of data is facilitated through DSLs that can be specified at the nonterminal nodes and labels of the schema metamodel. With the help of a DSL that is specifically designed for a particular context and purpose, the application of transformation in this section is based on source data, which has been decomposed and preprocessed according to the specifications of domain experts. Based on a generated AST, data needs to be further processed in order to complete the language processing pipeline of the rule framework by producing output versions of original data—again based on the specific definition of a domain experts. Like data descriptions, transformation rules can be specified in collection-, task- or mapping-oriented fashion[24] to produce specific results.

The focus of section 5.4.1 consists primarily in the derivation of a powerful and extensible data transformation language, which satisfies the best practices of DSL design. Based on the defined data transformation language, section 5.4.2 then discusses the required capabilities of an implementation within the rule framework.

5.4.1 Language design

The primary purpose of the data transformation language—in the context of this thesis—is to provide the capabilities for explicating functions that transform a provided input in terms of an AST as specified in section 5.3.2.2. Although existing model transformation languages such as the ATL or QVT provide specifications that could serve as blueprint for the implementation of a data transformation language within the rule framework, the data transformation language in this thesis is constructed as a specifically designed DSL—mainly to address the abstraction requirements discussed in Ghosh (2011, 23):

- *Minimalism*: Only include elements in the language, which are implemented in the framework. Generic languages generate a substantial implementation overhead, which is prevented by limiting the language specification to the required and intended implementation.

[24] see page 47

- *Distillation*: The DSL should contain only essential and required constructs to improve understandability and—like in the requirement above—reduce implementation overhead.
- *Extensibility*: As the language is implemented with respect to the minimalism requirement above, some required functionality might not be included in initial versions of the language, which should hence be easily extensible.
- *Composability*: Language elements should be designed to be composable with other elements to create higher-order abstractions.

Based on these best-practices for the creation of a DSL, the design of the transformation language in this thesis has a focus on *simplicity* while still providing the *expressiveness* required to allow the flexible definition of transformation functions—transforming an input AST to possibly hierarchical output elements.

5.4.1.1 Fundamental language patterns

In language application scenarios such as compiler engineering, grammatical rules can often be derived from existing sentences of the language or existing specifications (Parr 2013, 58-61). This principle has been shown to be applicable for the specification of data description languages in section 5.3.1.2), where grammars can be based and tested on the data they are intended to define.

For the construction of the data transformation language, central requirements need to be derived from the main purpose of the language: the specification of functions for the transformation of elements of semi-structured data.

Element assignment

In its basic form, an assignment $e_S \rightarrow e_T$ specifies the relation of a source and target element. Although this simple assignment is not required to be specified in terms of a transformation function as such value correspondences could be specified on the schema-level, it nevertheless forms the base of the data transformation language.

```
output1 = @input1;
output2 = @input2.subvalue;
output3 = "String constant";
output4 = 12;
```

Listing 5.6: Element assignment syntax

Listing 5.6 shows the syntax of a simple element assignment. For improved readability, the right-to-left assignment operator (*output* = *input*) is preferred over the function notation (*input* → *output*):

- Although domain experts are not assumed to be experienced programmers, common programming languages use the right-to-left assignment operator.
- Operations are performed on the input before a result is assigned to an output. The output-first notation allows an easier navigation through more complex expressions.

Mainly for the reason of improved readability input elements are annotated with a preceding @ symbol (as metaphor for 'the data (at) the element'), which facilitates the disambiguation of input elements (*output1* and *output2*) and constants (*output3* and *output4*). Following the example of common programming languages, statements are closed with a terminating semicolon with the additional parsing benefit of whitespaces as line-breaks and -feeds between tokens becoming irrelevant.

Object assignment

The previous listing indicated that assignments can refer to subelements of a provided input element, which provide an initial step to to support the idea of concept mappings and hence to allow semantic associations. Since specified output elements could consist of subelements, the semantics of an object assignment are introduced to the data transformation language.

Listing 5.7 reflects an example assignment, where the two individual input elements (*@input1* and *@input2*) are assigned to the according subelements of an *object* object.

```
output = {
  subelement = @input1;
  subelement = @input2;
};
```

Listing 5.7: Object assignment syntax

Please notice that the example in listing 5.7 cannot be equivalently replaced by two individual assignment statements: Whereas the object assignment results in the creation of one *object* with the properties *subelement1* and *subelement2*, two individual statements of the form *output.subelement1 = @input1; output.subelement2 = @input2;* raise an ambiguity because either one *output* with both properties or two *output* objects could be intended. In order to prevent such output assignment ambiguities, the notation *output.subelement = @input* is not allowed in the data transformation language.

Transformation commands

With the help of element and object assignments, grammatically decomposed and preprocessed input data can be assigned to corresponding output elements. Data integration often requires the application of operations that change input data by performing transformations such as aggregation or data cleansing.

Listing 5.8 introduces the command syntax to the data transformation language. Calls to transformation commands show similarity to common programming languages and follow the form *COMMAND(arg$_1$, arg$_2$, ..., arg$_n$);* with a syntactically unlimited list of arguments. Commands can be used as assignment source or recursive argument of other commands.

```
output1 = CONCAT(@input1, '\t', @input2);
output2 = {
  minimum = MIN(@input3, 0);
  maximum = MAX(@input3, 100);
};
output3 = IF( EQ(@input4, 'matchString'),
              @input5.subelement1,
              @input5.subelement2
          );
```

Listing 5.8: Transformation command syntax

Extensibility is a requirement of particular importance to the concept of the data transformation language. Whereas standard commands as in the example above are supported by a core implementation of the rule framework, the set of available commands must be extensible with respect to new and potentially complex functionality. Without necessary changes to the syntax and semantics to the data transformation language, existing text analysis and processing functionality[25] can be injected[26] to the framework e.g. by creating wrappers that accept calls in terms of the presented command syntax—improving the overall functionality of the data transformation engine.

Multiplicity

Both previous examples implicitly assumed single occurrences of assigned input elements, which results in the creation of one equivalent output element in the target schema. The existence of multiple instances of an input element depicts a commonly found pattern of semi-structured documents.

The XML Schema language allows the restriction of the multiplicity of elements with the minBounds and maxBounds keywords. Additionally XML prevents the existence of multiple attributes with the same name under the same element and hence contains. Whereas in XML, multiplicity is reflected by the creation of multiple instances of the same element, the language specification of JSON identifies arrays as means to reflect multiplicity.

For the concept of the data transformation language, multiplicity is resolved at the place of an input element selection and must be implemented as an array assignment for the following reasons:

- Language applications can later decide what to do with array vs. element list
- Commands are n-ary operators
- For cases where a set of input elements needs to be transformed to a smaller set or one individual element, commands can be implemented e.g. to create output1, output2 etc.

[25] e.g. Heideltime or OpenNLP
[26] dependency injection receives further attention in chapter 6

Object scopes and filters

Multiplicity introduces a new challenge that needs to be solved in terms of the data transformation language. Consider the JSON example in listing 5.9, which represents an enriched version of DC: By means of a data description language, the original *creator* element has been decomposed into its components of *LastName* and *FirstName*.

```
{
  "DublinCoreEnriched" : {
    "Title" : "Sedimentology and susceptibility of core MD88-769",
    "Creator" : {
      "LastName" : "Bareille",
      "FirstName" : "Gilles"
    },
    "Creator" : {
      "LastName" : "Grousset",
      "FirstName" : "Francis"
    },
    "Creator" : {
      "LastName" : "Labracherie",
      "FirstName" : "Monique"
    }
  }
}
```

Listing 5.9: Multiplicity and scoping example

Based on the semantics of object assignments, a potential concept mapping to an exemplary *Author* element with the sub-elements *LName* and *FName* would be defined as shown in listing 5.10. Based on the previously introduced multiplicity resolution concept, the assignment result in the generation of one *Autor* element with one *FName* sub-element, containing an array of first names *["Gilles", "Francis", "Monique"]* and one *LName* sub-element with the set of last names *["Gilles", "Francis", "Monique"]*. The collection of all selected input elements is expected and can e.g. be compared to the XML Path Language (XPath) expression */Creator/FirstName* on an equivalent XML document.

```
Author = {
  FName = @Creator.FirstName;
  LName = @Creator.LastName;
};
```

Listing 5.10: Incorrect object assignment example

In order to facilitate the intended behavior, a scope parameter is introduced to the language as shown in listing 5.11. The example has been extended by an additional *Addr* sub-object, which is produced from the equivalent sub-object of the input element—thus requiring the consideration of an *@Address* scope subordinate to the main *@Creator* scope.

```
Author = @Creator {
  FName = @FirstName;
  LName = @LastName;
  Addr = @Address[@Type=="business"] {
    Street = CONCAT(@Street, ' ', @Number);
    City = CONCAT(@ZipCode, ' ', @City);
  };
};
```

Listing 5.11: Object assignment example

If the scope parameters is accompanied by an optional scope filter as in listing 5.11, a filter is applied when selecting input subtrees from a provided AST.

5.4.1.2 Language definition

Based on the language patterns identified in the previous section, a grammar can be derived in order to implement the data transformation language. 'Implementation' at this point refers to the general concept of language applications and is completed with the compilation of a lexer and parser—allowing the lexical and semantical analysis of input sentences.

Listing 5.12 shows the data transformation language defined in terms of an ANTLR grammar.

```
grammar DataTransformation;

/** Parser rules ------------------------------------------------- **/
func        : stmt+;

stmt        : output  '=' (object | assign) ';';
output      : selector;

/** Object assignments (optional scope) **/
object      : scope? '{' stmt+ '}';
scope       : '@' selector ('[' scopeFilter ']')?;
scopeFilter : '@' selector '==' filterExpr;
```

```
/** Element assignments **/
assign      : input
            | command
            | value;
input       : '@' selector;
command     : function '(' assign (',' assign)* ')';

/** Reused rules **/
selector    : ID ('.' ID)*;
function    : ID;
value       : NUMBER
            | STRING;
filterExpr  : NUMBER
            | STRING;

/** Lexer rules ------------------------------------------------- **/
ID          : LETTER (LETTER|DIGIT)*;
NUMBER      : '-'? (DIGIT+ ('.' DIGIT*)?);
STRING      : '"' ('\\"'|.)*? '"';

fragment LETTER : [a-zA-Z];
fragment DIGIT  : [0-9];

WS          : [ \t\r\n]+ -> skip;
```

Listing 5.12: Grammar of the data transformation language

Please note that this language specification can be extended by further component and represents a premature version—e.g. the *scopeFilter* rule should probably allow more complex boolean expressions. However, the language specification shows the general applicability of an ANTLR based processing for data transformation.

Commands have been chosen to be not literally specified in terms of the data transformation grammar for reasons of extensibility: If new functions are implemented and made available to the rule framework, the transformation language would have to be recompiled on every change. Instead, the *enterCommand* event is required to trigger a check, whether a specified command is supported.

The modification and recompilation of the transformation language results in different and possibly conflicting dialects of the data transformation language. By associating specified transformation functions with an exact version of the data transformation language, future languages could be implemented without altering the behavior of previously specified functions.

5.4.2 Runtime behavior

As indicated in the overview to this section (see figure 5.3 on page 44), the primary runtime behavior at the transformation stage of the processing pipeline is characterized by its input: the *input parse tree* and the *transformation function*—as well as its purpose: the creation of an output parse tree. The required runtime functionality is reflected by the two main activities *transformation function compilation* and *parse tree combination* as shown in figure 5.9.

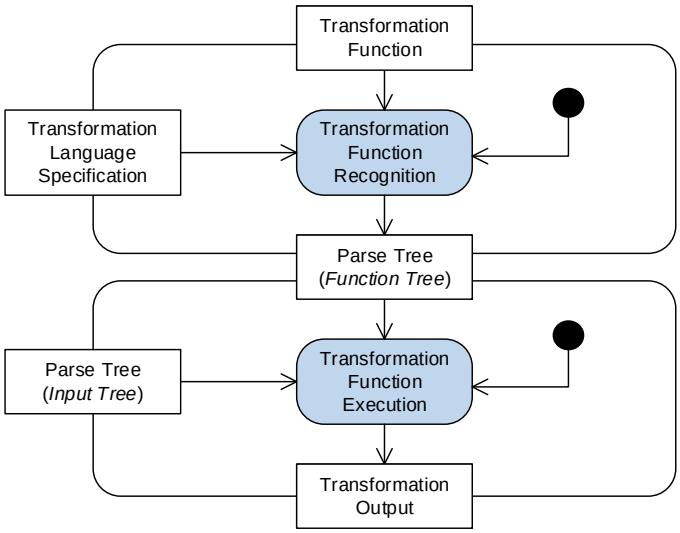


Figure 5.9: Runtime behavior of data transformation

5.4.2.1 Transformation function compilation

Data transformation functions are sentences of the language defined in the previous section. As such, the processing of data transformation functions can be compared to that of data description:. Whereas the language definition in the data description phase depends on the schema definition of the processed data, one global grammar for data transformation functions exists (as specified in section 5.4.1.2).

The parse tree in figure 5.10 shows the result of the semantical analysis of the exemplary data transformation function in listing 5.11 on the basis of the grammar specified in the previous section. Similar to the handling of data description functions, the task of language recognition ends with the generation of a parse

tree, which is then walked and processed in subsequent steps.

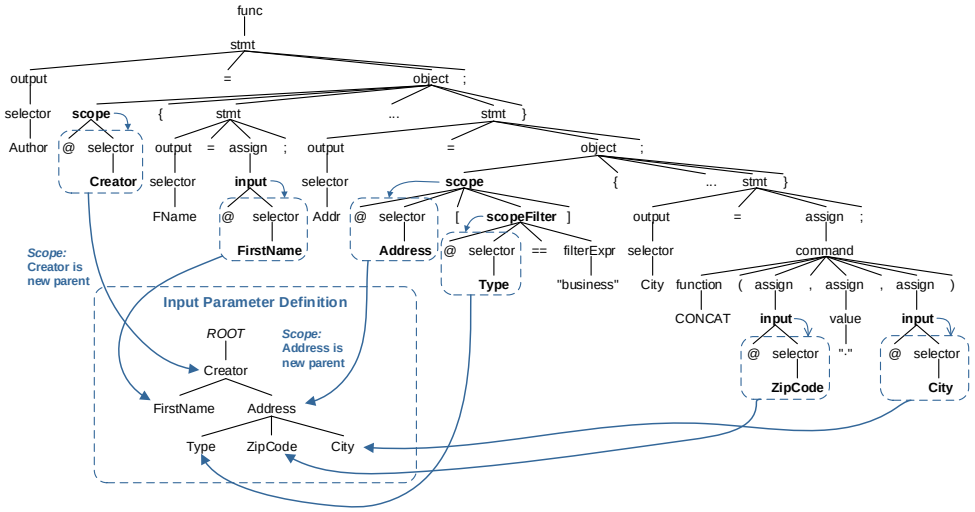


Figure 5.10: Data transformation parameter collection

Figure 5.10 also shows the functionality of the data transformation parameter collection, which is required in the data description phase in order to generate ASTs.

5.4.2.2 Function execution

The execution of data translation functions depends on two input parse trees:

- The *function tree* with the semantics of data transformation—i.e. rules, to which the rule framework needs to respond.
- The *input tree*, which contains the input parameters in hierarchical form. Terminal symbols of the input tree correspond to values, which are intended to be inserted in place of the input parameters of the function tree. Non-terminal symbols reflect the structure of parsed input, which is utilized to select the values.

The order and logic of a data transformation function is thus represented by a function tree. Traversing the function tree, the two primary tasks consist in (1) identifying input placeholders in the function and replacing them with input values found in the input tree and (2) executing rules such as assignments and

commands to render output values. The data transformation grammar as specified in section 5.4.1.2 produces functions, which are intended to be processed in terms of a depth-first, pre-order logic, which can be verified on the basis of the function tree presented above.

The concept for an interpreter of data transformation functions is based LIFO stacks, which are further referenced as stacks—with respect to the basic computing terminology and the corresponding Java type—and both relate to the common language application pattern of symbol tables (see e.g. Parr 2013, 138-145):

Output label stack Labels of output parameters are put on this stack as soon as the left hand side of a statement is processed. Keeping output parameter labels on a separated structure in memory is required because a statement can create multiple output parameters as indicated in section 5.4.1.1 (Multiplicity)

Output parameter stack The *output parameter stack* serves as container for output parameters within the current scopes and is an extended form of a *symbol table for nested scopes* as discussed in Parr (2010, 161-169). The pattern is typically used for interpreting or compiling programming languages, which allow scoped fields: e.g. class variables are hidden by method variables. An extension of the typical form of the symbol table is required in order to reflect the combination of input and function trees: Whereas in the context of programming languages, "each function has its own scope that is nested within a global or class scope" (Parr 2010, 161), in the context of the data transformation functions of this thesis, each selected subtree of the input tree to which an assignment is applied has its own scope and the output parameter scope receives the following characteristics:

- Each entry in the stack consists of a list of output parameters, reflecting the current object scope applied to the input tree.
- Each entry in the list of output parameters of a particular scope represents a subtree of the original input tree, which has been selected by the scope selector—based on the parent scope subtree or subtrees.

Command stack Output parameters are (1) labeled according to the topmost item of the output label stack and (2) assigned the value of the right hand of the

'=' operator, which is a choice of the alternatives *command*, *value*, *input* and *object*. Whereas constant and input values can be assigned to output parameters, object assignments result in the creation of output parameter hierarchies—i.e. the parent parameter as nonterminal node is assigned child nodes, which are then recursively processed. Command assignments differ from other alternatives since it (1) like values and inputs result in the assignment of values to output parameters and (2) requires a possibly nested calculation based on values, inputs or subsequent commands. Command stacks at the instances of generated output parameters can be utilized to support nested commands:

- Commands are executed per identified subtree of the current scope.
- Commands change the application target of subsequent assignments—i.e. assignments then target arguments passed to the current command and not the output parameters of the current scope.

Table 5.1 summarizes the actions, which are required of the rule framework upon entering and/or exiting specific nodes of the function parse tree.

Rule	*Enter/Exit*	*Functionality*
stmt	enter	Put the label of the subordinate output rule on the output label stack
	exit	Remove the topmost entry from the output label stack
object	enter	For every item in the topmost entry in the output parameter stack: add a new child output parameter with the name of the current topmost label on the output label stack; every parameter contains a reference to its corresponding item in the input AST
	exit	Remove the topmost entry from the output parameter stack
command	enter	Create a new command parameter and put it in the command stack of every output parameter in the topmost entry in the output parameter stack
	exit	Remove the topmost entry from every command stack, execute the specified command for every item in the topmost entry of the output parameter stack and assign the result (1) if available to the next entry in every respective command stack or (2) create a terminal output parameter with the value set to the respective result and assign it to the respective parent
input	exit	For every referenced subtree of every item in the topmost entry of the output parameter stack, lookup the reference path and set the determined value like in exitCommand (1) to the next entry in every command stack or (2) create a subordinate terminal output parameter for every output parameter in the topmost entry of the output parameter stack
value	exit	Assign the specified constant like exitCommand and exitInput to (1) to the topmost entry in every command stack, if available, or (2) create a subordinate terminal output to the current parameter items in the topmost entry of the output parameter stack

Table 5.1: Function tree dependent actions for data transformation

5.5 Conclusion

Based on the phases of data description and data transformation, the concept presented a generic rule framework, which facilitates the domain-specific description of the language of data and the subsequent formulation of transformation functions. Based on this task separation, the complexity of the resulting languages could be reduced by employing

- a standardized form for the specification of context-free grammars in terms of the EBNF for the definition of descriptive DSLs, and
- an expressive transformation language, which provides extension points for both an easy syntactical and a functional extension.

Figure 5.11 concludes the conceptual work of this thesis by showing the application of the rule framework in the context of schema metamodels.

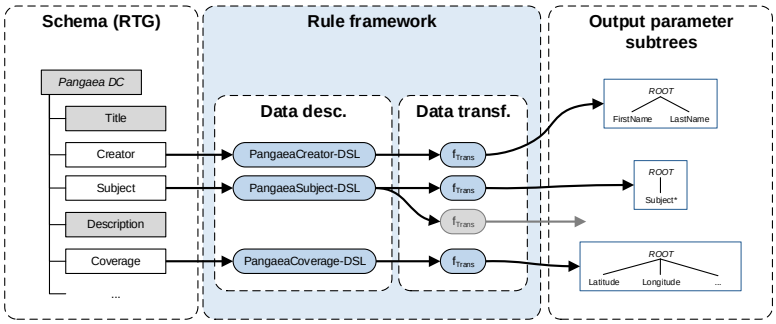


Figure 5.11: Rule framework application overview

Applications embedding the functionality of the rule framework need to decide about the further processing of the generated output parameter subtrees. For the particular example in figure 5.11, the original input elements of the schema (as regular tree grammar) could be used as *ROOT* of the subtrees—implementing the labeling functions. The specified transformation could also be specified in terms of a concept mapping, which results in the subtrees presenting the actual structure to be set in terms of the target schema. As this transformation can be performed on data and structured queries, the rule framework is applicable for the integration problems defined in section 5.1.

Three further extension points should be explicitly detailed at this point:

- *Iteration*: The rule framework can be applied in an iterative fashion as is. Assuming the application of the example in 5.11 in terms of labeling functions, the produced labels (e.g. Latitude, Longitude) can form the base nodes (compare the definition of the metamodel in section 3.3.2) for a further application of the rule framework.
- *Reuse of descriptions*: Figure 5.11 shows an additional, unused transformation function after the application of the PangaeaSubject DSL to show the re-usability of data descriptions for collection-, task- or mapping-oriented transformation functions.
- *Multiple descriptions*: Although not further detailed, the mapping of elements in the schema with the desired data descriptions and hence entry points to the rule framework is not limited to exactly one DSL.

6 Chapter Implementation

The developed prototype is primarily contained in two coherent Java projects: the rule framework (*transformation*), which contains the implementation of the designed concept and a Java-based web application (*transformation-testapp*), which is intended to show the application of the framework within user-oriented views.

The rule framework is implemented as generic component, which combines the following functionality:

- *Grammar generation*: During the runtime of the rule framework—e.g. as component of a Java web application—a user can provide a context-free grammar, which is (1) converted to Java source code, (2) compiled to Java bytecode and (3) accessed in terms of a dynamic class loader.
- *Transformation function generation*: Based on grammatical rules of the data description phase, users can define transformation functions in terms of the language specified in section 5.4.1.2
- *Data transformation*: By applying the declaratively defined rules, data is transformed.

6.1 Logical architecture

Although the functionality of the rule framework is implemented in a generically reusable fashion and could be presented as such, its integration within an existing data processing pipeline is assumed to result in a better understandability of the

implementation.

Structure

Figure 6.1 provides an overview of the components that are required for the realization of a particular use-case, the semantic enrichment of semi-structured data.

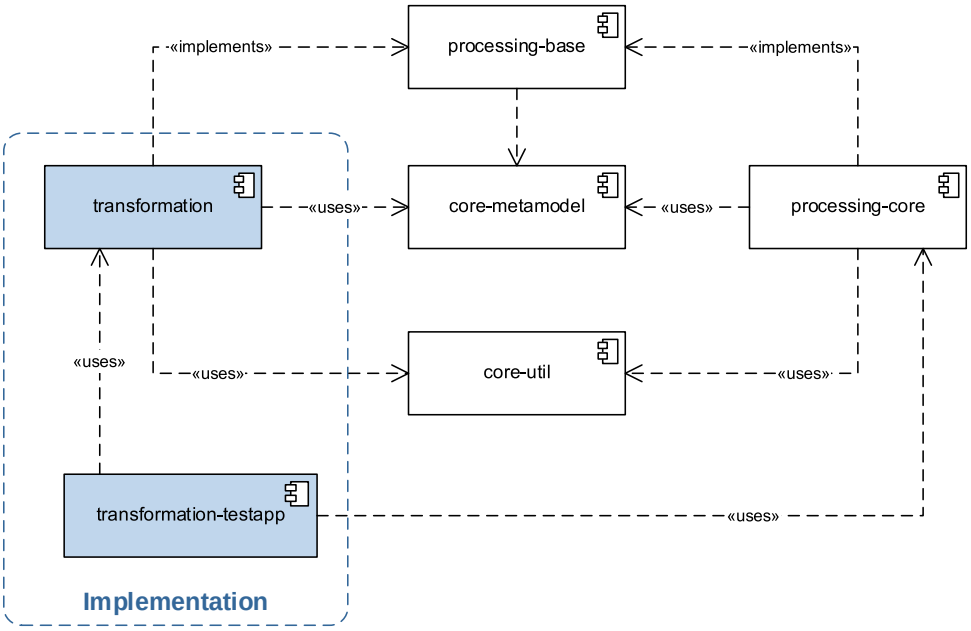


Figure 6.1: Implementation context of the transformation rule framework

The components shown in figure 6.1 are each individual Java projects—the *uses* and *implements* stereotypes indicate dependencies between projects.

- *core-metamodel* contains the implementation of the schema and mapping meta-model as developed in Gradl (2014) and presented in section 3.3 of this thesis.
- *core-util* is project of auxiliary classes without an immediate relation to the primary functionality of the rule framework, but which e.g. facilitate the initialization of Java applications or tests based on Spring[27].

[27] see https://spring.io/

- *processing-base* contains interfaces and base implementations for processing semi-structured *documents* and *data*—i.e. that conforms to regular tree grammars as specified in Gradl (2014).
- *processing-core* implements fundamental aspects of document and element level processing for generic XML and OAI-PMH sources. By separating base interfaces in *processing-base* from generic implementations in *processing-core*, the extension of the data processing framework by more specific aspects of data processing is facilitated.
- *transformation* contains the functionality of the rule framework and forms a extension of the fundamental processing by allowing the analysis and transformation of the data contained within terminal nodes of the regular tree grammar represented by the schemata.
- *transformation-testapp* is a simple web application that is intended to present an overview of early ideas to integrate the rule framework within a web-based user interface.

Behavior

The two important components for the application of the rule framework to the data enrichment use-case are formed by the projects *processing-core* and *transformation*. The behavior of both components (along with any other implementations of *processing-base*) are configured in terms of a schema configuration. Listing 6.1 shows a relevant section of the serialized configuration that is used as for the presentation of an example throughout this chapter. Please note that although the text-based representation is required for an autonomous execution of the rule framework (e.g. by means of unit tests), the schema configuration is intended to be viewed and edited in terms of a user interface (see figure 6.6 on page 82).

The presented configuration is an excerpt of the context-specific adaption of the generic DC schema that specifies (1) the decomposition of the nonterminal *Creator* element by recognizing actual content by means of the *PangaeaCreator* grammar (see listing 6.2) with the entry rule *fullName*. The results of the data description phase are then used as input to the subordinate data transformation function, which finally creates the output required to fill the *FirstName* and *LastName* elements.

```
{
  "class" : "eu.dariah.de.minfba.core.metamodel.xml.XmlSchema",
  "uuid" : "pangaea_dc",
  "root" : {
    "id" : 1,
    "name" : "OAI DC",
    "functions" : null,
    "childNonterminals" : [{
        "name" : "Creator",
        "functions" : [{
            "baseMethod": "fullName",
            "grammarName": "PangaeaCreator",
            "dataTransformationFunctions": [{
                "externalInputElements": null,
                "outputElements": [{
                    "name": "FirstName",
                    "functions": null,
                    "transient": false
                  },{
                    "name": "LastName",
                    "functions": null,
                    "transient": false
                  }],
                "function": "FirstName = @firstName; LastName = @lastName;"
              }]
          }],
      ...
      }],
    "transient" : false
  }
  ...
}
```

Listing 6.1: Excerpt from an exemplary element configuration

```
grammar PangaeaCreator;

fullName : lastName ', ' firstName;

lastName :  ID;
firstName : ID;

WS  : [ \t\r\n]+ -> skip ;
ID  : ~(',')+;
```

Listing 6.2: Pangaea creator grammar

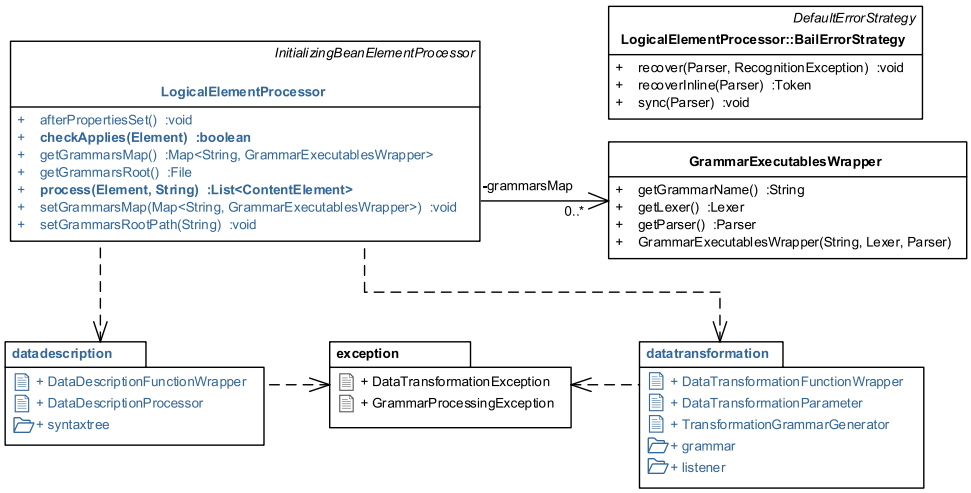


Figure 6.2: Class diagram of the rule framework

6.2 Rule framework

The *transformation* project contains the implemented functionality required to apply domain-specific language recognition to input data and to further process the data in terms of data transformation functions. The class diagram in figure 6.2 shows the overall structure of the rule framework—highlighting the three most important components:

- The *LogicalElementProcessor* class—as implementation of the *InitializingBeanElementProcessor* interface of the *processing-base* project—provides the main entry point to the rule framework for the focused use-case of data enrichment.
- The *datadescription* package (see section 6.2.1) contains the functionality (1) to process and compile domain-specific grammars first to Java source code then bytecode and (2) to process data against such compiled grammars.
- The *datatransformation* package (see section 6.2.2) implements the process of (1) parsing and processing formulated data transformation functions that are specified against the predefined transformation language and (2) to actually transform data with respect to these functions.

At every configured nonterminal element of the schema[28], the processor interacts with the *LogicalElementProcessor* to determine, whether the content of the processed element qualifies to be processed by the rule framework by calling *checkApplies(-)*.

Upon a call to *checkApplies(-)*, the *LogicalElementProcessor*:

1. Verifies the passed element configuration (e.g. the entire object with *"name" : "Creator"* in listing 6.2) and returns `false` if the element is not configured to produce sub-elements. Otherwise:
2. Checks if the *grammarsMap*[29] contains required parser and lexer classes for the grammar (encapsulated within *GrammarExecutablesWrapper* objects):
   - Returns *true*, if the grammar is contained in the map.
   - Tries to load the appropriate classes from a configured base location and register them in the *grammarsMap*. Returns *true*, if classes were found and loaded, otherwise *false*.

In case of a positive answer to *checkApplies(-)*, the schema processor again provides the element configuration in addition to the element value to the *process(-,-)* method of the *LogicalElementProcessor*. For every data description grammar that is configured to be applied to the element value, the *LogicalElementProcessor* first identifies the applicable domain-specific grammars for *data description* and utilizes the *data description* (see 6.2.1) component to generate parse trees, respectively. For every configured data description grammar, the *LogicalElementProcessor* iterates over the configured data transformation functions and concludes the rule framework processing by interaction with the *data transformation* (see 6.2.2) component.

6.2.1 Data description

The implementation of the data description functionality in terms of domain-specific languages is summarized in the class diagram in figure 6.3.

[28] *schema* in the sense of regular-tree-grammar

[29] The *grammarsMap* serves as grammars cache for a quickly accessing the language recognition classes, as they are required for every configured element of any provided document

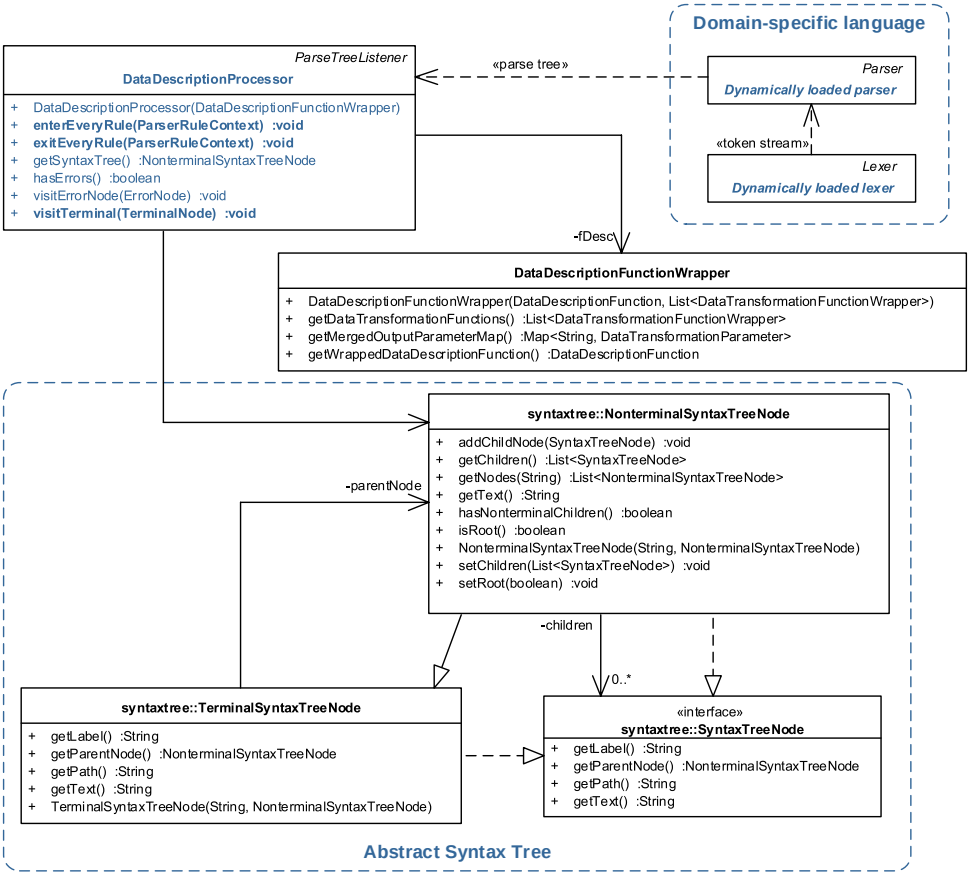


Figure 6.3: Class diagram of the data description package

As implementation of the generic listener pattern as specified in the *ParseTreeListener* interface of the ANTLR framework, the *DataDescriptionProcessor* class contains the functionality for traversing provided parse trees. As indicated in the class diagram, the *DataDescriptionProcessor* is required to react to language specifications, which are not part of the compiled *transformation* Java project, but are defined and possibly modified by domain experts at the runtime of the rule framework.

Based on an instance of the *DataDescriptionFunctionWrapper*, the *DataDescriptionProcessor* has the information required to determine, which rules of a provided parse tree are relevant for any of the subordinate data transformation functions and thus to determine the nodes of the input parse tree, which need to be reflected

in terms of the generated AST. As specified in the concept, the *DataDescriptionProcessor* operates on two stacks:

- The *inputDefinitionStack* for representing the current hierarchy level of the parameters that are requested by the data transformation functions and
- The *syntaxTreeStack*, which represents the current hierarchy level of the generated syntax tree.

6.2.2 Data transformation

After a provided input value has been recognized in terms of a domain-specific grammar, the phase of data description creates an AST from the original parse tree—based on required input parameters of the data transformation phase. In order to determine the set and hierarchy of these parameters, data transformation functions have to be parsed and analyzed for *input* and *scope* language elements. Aside from the functionality for an actual *execution* of data transformation functions, a *preparatory* phase with the particular goal of identifying and collecting the parameters is introduced to implementation.

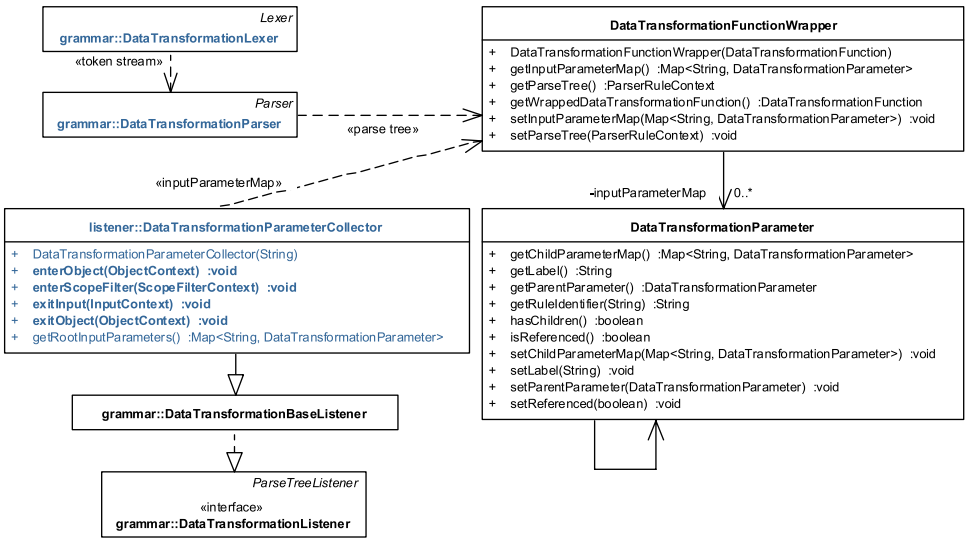


Figure 6.4: Class diagram of the data transformation preparation

Preparation

In order to prepare the execution of data transformation functions, an instance of the *DataTransformationParameterCollector* reacts to the input-parameter related events of the tree traversal—creating a hierarchy of *DataTransformationParameter* objects. The anonymous root object is required for the *DataDescriptionProcessor* to determine the required structure of the generated AST.

If multiple data transformation functions are to be executed on the basis of a produced AST, a *DataTransformationParameterCollector* listener is created and utilized to walk each function respectively. The *LogicalElementProcessor* merges the *DataTransformationParameter* hierarchies of each function—ensuring that the input needs to be walked only once and hence only on AST is required to be produced.

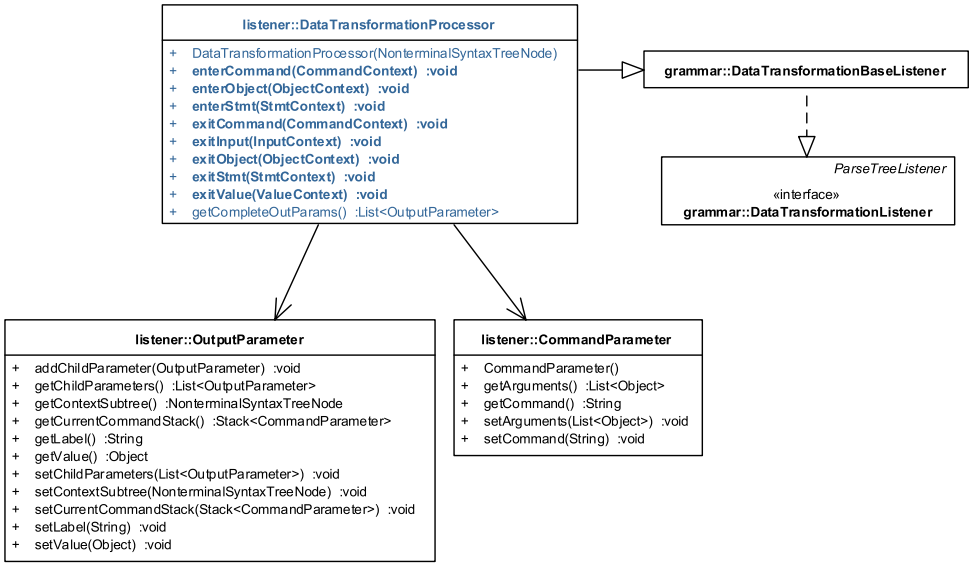


Figure 6.5: Class diagram of the data transformation execution

Execution

The class diagram finally presents the main *DataTransformationProcessor*, which combines the information of the transformation function parse tree and the input AST in order to generate the desired output—according to the provided instructions.

6.3 Web interface

In addition to the generic implementation of the rule framework and its integration into the preexisting XML and OAI-PMH processing pipeline, an initial prototype is implemented in order to present the applicability of the rule framework in terms of a web application.

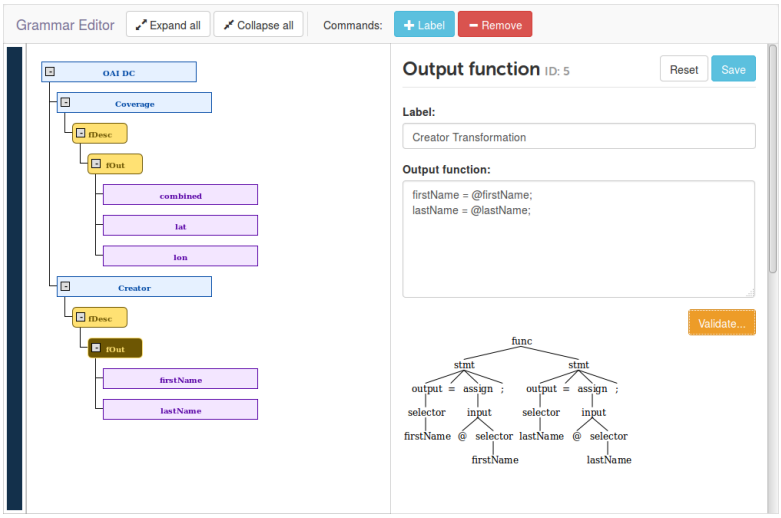


Figure 6.6: Screen of the grammar editor prototype

The screen in figure 6.6 shows an editing interface, which allows the extension of the parsing-oriented view of schemata by the semantic extension as presented in section 3.3.2. The definition of nonterminals of the regular tree grammar are presented in blue, whereas functions for the data description and data transformation are displayed in yellow and produced labels in purple. Upon selecting the node of a transformation function, the user is presented with editing elements on the right hand side of the screen, which allow the specification and validation of provided transformation functions.

Figure 6.7 presents the application of the rule framework on an exemplary set of Pangaea documents. Please note that the resulting output contains only those elements of the regular tree grammar that have been specified in 6.6. The transformation viewer is expected to be especially beneficial for users testing description grammars and transformation functions on actual data before committing them to

a processing system.

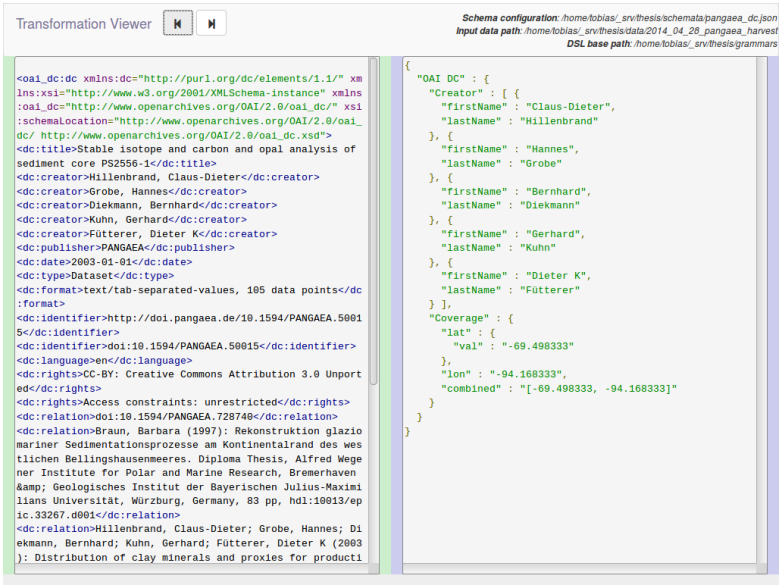


Figure 6.7: Screen of the transformation viewer prototype

7 Chapter
Evaluation

Although the concept and core implementation of the rule framework in this thesis form a generically reusable software component, it has been developed with regard to the particular application domain of the arts and humanities. As such, the core implementation is required to be embedded within software that transforms data according to the research-oriented specifications in order to generate an actual proof of the presented concept.

The results of this evaluation should be interpreted as early perspectives on the use-cases that could be supported by the data transformation framework—the actual applicability, however, needs to be evaluated within the domain.

7.1 Pangaea

The dataset of the earth and environmental science service, Pangaea, has served as continuous example throughout this thesis—particularly because of earlier experiments in terms of the work in Gradl (2014), but also because the service provides a large (>700,000 records) dataset that is OAI-PMH accessible and exhibits some relevance to research within environment-related fields of the arts and humanities, such as archeology or art history.

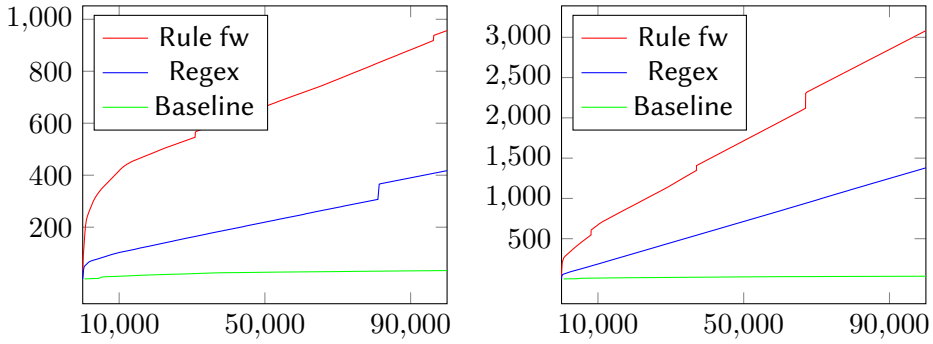
Based on the harvested Pangaea dataset, the generation of parse-trees from the contents of the *coverage* element has been measured and compared to that of data processing in terms of a regular expression. For the particular case of the Pangaea

coverage element, the matched expression

```
(?<Key>\\w[\s\w-,]*):(?<Value>[^\*]*)*/
```

results in the decomposition of an accordingly specified input string.

The results in figure 7.1 show the processed record count on the x-axis and the cumulative processing time in milliseconds on the y-axis. The red plot corresponds to the generation of the parse tree by the rule framework, the blue plot to the regular expression processing and the green line reflects the overhead of the underlying Java implementation.



a) 4 key/value pairs a) 20 key/value pairs

Figure 7.1: Comparison with regular expression processing

Both for the creation of four and twenty key/value-pairs, the parse-tree generation in terms of ANTLR as employed by the rule framework performs slower than regular expressions. Surprisingly, the gap between the regular expression and the rule framework implementation is not impacted by the size of the produced parse tree and hence the complexity of the parse: the execution of the rule framework completed with a factor of 2.29 compared to regular expression processing (956ms vs. 417ms) with four pairs and with a factor of 2.23 with twenty pairs (3082ms vs. 1379ms).

To which degree the performance of the rule framework could be further improved, as well as whether the performance of the current or future versions is sufficient for the application domain requires further analyses and evaluation.

7.2 Wikidata

The wikidata evaluation presents a qualitative test that has been executed in terms of the rule framework. For the test, datasets of a recent dump of the wikidata database[30] have been processed according to a custom grammar (see listing 7.1).

```
grammar WikidataTextContent;

object      : '{' statement (',' statement)* '}';

statement   : label
            |   description
            |   genericProp;

label       : '"label"' ':' (langObj | langObjArray);
description : '"description"' ':' (langObj | langObjArray);

genericProp : STRING ':' (array | obj | value );
obj         : '{' pair (',' pair)* '}'
            | '{' '}';

array       : '[' value (',' value)* ']'
            | '[' ']';

claimItem   : '[' STRING ',' claimPropertyId ',' claimPropertyType ','
    claimRefOrValue ']';

claimPropertyId   : NUMBER;
claimPropertyType : STRING;
claimRefOrValue   : obj
                  | STRING
                  | NUMBER;

langObj       : '{' pair (',' pair)+ '}';

langObjArray  : '[' langObj (',' langObj)* ']'
              | '[' ']';

pair    : key ':' value;
key     : STRING;
value   : claimItem
        | STRING
        | NUMBER
        | obj
        | array
        | 'true'
        | 'false'
        | 'null';
```

[30] http://dumps.wikimedia.org/backup-index.html

```
STRING  : '"' (ESC | ~["\\])* '"';
WS      : [ \t\r\n]+ -> skip ;

NUMBER  : '-'? INT '.' [0-9]* EXP?
        | '-'? INT EXP
        | '-'? INT;

fragment INT : '0' | [1-9] [0-9]*;
fragment EXP : [Ee] [+\-]? INT;

fragment ESC : '\\' (["\\/bfnrt] | UNICODE);
fragment UNICODE : 'u' HEX HEX HEX HEX;
fragment HEX : [0-9a-fA-F];
```

Listing 7.1: Wikidata grammar

In general, the data description grammar decomposes JSON content that is embedded within an XML element of the embodying wikidata record. As such, the presented grammar combines syntactical aspects required for processing JSON data with the context-specific requirements of the subsequent data transformation functions.

Figure 7.2 shows an excerpt of an exemplary parse tree. The data transformation functions in the listings 7.2 and 7.3 indicate how the extracted data can be further processed to generate common properties for every valid wikidata record as well as specific properties for datasets about human entries.

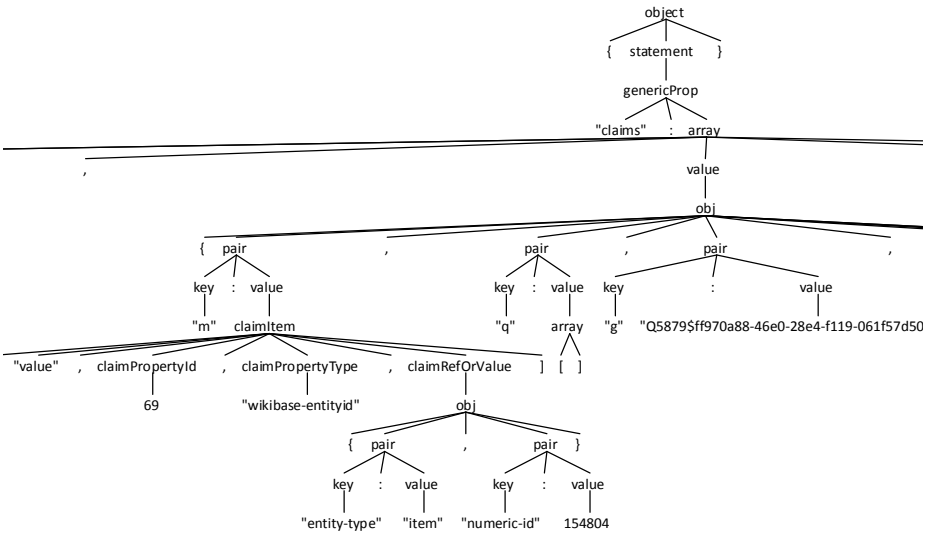


Figure 7.2: Parse tree excerpt of an example wikidata input

```
GndID = @claimItem[@claimPropertyId == 227]{
  Value = @claimRefOrValue;
};
TitleDe = @label.langObj.pair[@key == \"de\"]{
  Title = @value;
};
TitleEn = @label.langObj.pair[@key == \"en\"]{
  Title = @value;
};
InstanceOf = @claimItem[@claimPropertyId == 31] {
  Object = @claimRefOrValue.obj.pair[@key == \"numeric-id\"] {
    NumericId = @value;
    RefId = CONCAT(\"Q\", @value);
    WikiLink = CONCAT(\"http://www.wikidata.org/wiki/Q\", @value);
  };
};
SubclassOf = @claimItem[@claimPropertyId == 279] {
  Object = @claimRefOrValue.obj.pair[@key == \"numeric-id\"] {
    NumericId = @value;
    RefId = CONCAT(\"Q\", @value);
    WikiLink = CONCAT(\"http://www.wikidata.org/wiki/Q\", @value);
  };
};
```

Listing 7.2: Transformation of generic wikidata properties

```
Human = {
  Occupation = @claimItem[@claimPropertyId == 106] {
    Object = @claimRefOrValue.obj.pair[@key == \"numeric-id\"] {
      NumericId = @value; RefId = CONCAT(\"Q\", @value);
      WikiLink = CONCAT(\"http://www.wikidata.org/wiki/Q\", @value);
    };
  };
  CountryOrCitizenship = @claimItem[@claimPropertyId == 27] {
    Object = @claimRefOrValue.obj.pair[@key == \"numeric-id\"] {
      NumericId = @value;
      RefId = CONCAT(\"Q\", @value);
      WikiLink = CONCAT(\"http://www.wikidata.org/wiki/Q\", @value);
    };
  };
  AlmaMater = @claimItem[@claimPropertyId == 69] {
    Object = @claimRefOrValue.obj.pair[@key == \"numeric-id\"] {
      NumericId = @value;
      RefId = CONCAT(\"Q\", @value);
      WikiLink = CONCAT(\"http://www.wikidata.org/wiki/Q\", @value);
    };
  };
  FieldOfWork = @claimItem[@claimPropertyId == 101] {
    Object = @claimRefOrValue.obj.pair[@key == \"numeric-id\"] {
      NumericId = @value;
      RefId = CONCAT(\"Q\", @value);
```

```
      WikiLink = CONCAT(\"http://www.wikidata.org/wiki/Q\", @value);
    };
  };
  ...
};
```

Listing 7.3: Transformation of human-related wikidata properties

7.3 POS tagged texts

The exemplary use-case of this section is intended to show (1) the potential of the rule framework with respect to the specification of DSLs and (2) the extensibility of the framework in general. Please consider the following scenario: A terminal node of a schema contains a large, unstructured text (such as in the TEI example presented in section 2.1.2), which should be analyzed. Assuming an accordingly implemented function, an initial data transformation rule is formulated over this element as

posTagged = TEXTANALYSIS.POSTAG(@unstructuredInput);

Based on the resulting *posTagged* label, which is now expected to contain data as shown in figure 7.3, a data description DSL is formulated as shown in listing 7.4—here specifically targeting a (rather random) natural language construct, which is intended to be processed by a data transformation function.

The/DT new/JJ cathedral/NN was/VBD consecrated/NN **6/CD May/MD ,/, 1012/CD** ,/, and/CC in/IN 1017/CD Henry/NNP II/NNP founded/VBD on/IN Mount/NNP St./NNP Michael/NNP ,/, near/IN Bamberg/JJ ,/, a/DT Benedictine/JJ abbey/NN for/IN the/DT training/NN of/IN the/DT clergy/NN ./. The/DT emperor/NN and/CC his/PRP wife/NN gave/VBD large/JJ temporal/JJ possessions/NNS to/TO the/DT new/JJ diocese/NN ,/, and/CC it/PRP received/VBD many/JJ privileges/NNS out/IN of/IN which/WDT grew/VBD the/DT secular/JJ power/NN of/IN the/DT bishop/NN (/(cf/IN ./.

Figure 7.3: Example full-text input with Part-Of-Speech (POS) tags

```
grammar Pos;

@header {
  import java.util.*;
  import java.lang.*;
}

@parser::members {
```

```
  boolean qualifiesAsYear() {
    try {
      int i = Integer.parseInt(getCurrentToken().getText());
      if (i<1800 && i>800) {
        return true;
      } else {
        return false;
      }
    } catch (Exception e) {
      return false;
    }
  }
}

/** The start rule; begin parsing here. */
tags  :   tag+;

tag : construct
    | np
    | year
    | cd
    | in
    | comma
    | any;

construct : cd np comma? year;

np    : ID '/' ('NN' | 'NNP');

year  : {qualifiesAsYear()}? ID '/' 'CD';
cd    :  {!qualifiesAsYear()}? ID '/' 'CD';

in    : ID '/' 'IN';
comma : ',' '/' ',';
any   :   ID '/' ID;


WS    : [ \t\r\n]+ -> skip ;  // Define whitespace rule, toss it out
ID    : ~(' '|'/')+;
```

Listing 7.4: Grammar for POS-tagged texts

Aside from the lexer and parser rules, which have already been introduced in this thesis, the grammar in listing 7.4 implements the idea of semantic predicates, which were shortly introduced in section 3.1.3 and are discussed in greater detail in Parr (2013, 286-291). The predicate employed in the presented grammar specifically addresses context-sensitive information required to decide about the semantics of a particular grammar rule. Whereas this distinction can also be performed in

terms of data transformation functions, this example shows how grammars can be extended by embedded Java actions, which will be included in generated parsers to also run natively during the parse.

A relevant part of the resulting parse tree is shown in figure 7.4—indicating, how relevant parts can be extracted from large and noisy context.

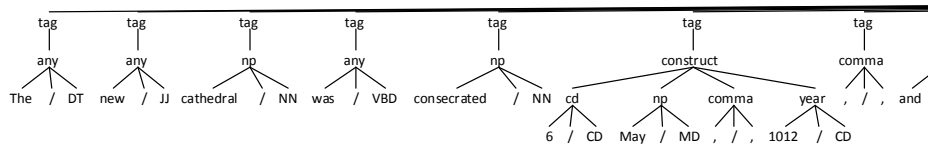


Figure 7.4: Parse tree excerpt of an example POS-tagged input

Numerous other interesting tasks can be facilitated by the rule framework. One additional example consists in the specification of content assertions—e.g. the domain expert first specifies the decomposition of content and additionally defines a subsequent recombination of this decomposed content. By comparing original input and the recomposed data in terms of a transformation function, the expert could detect problems in the specification of the transformation or the data themselves—e.g. with the help of the transformation viewer shown in figure 6.7 on page 83.

8 Chapter
Conclusion and future work

The presented thesis provides a new approach for the integration of heterogeneous data in contexts, which:

- require data to remain in their original structural and semantic form in order to allow a dynamic ad-hoc integration based on individual use-cases and
- need a particular focus on lexical and syntactical patterns, which are only bound to the instance-level of data and thus not covered by schema level integration.

Based on the preliminary work in Gradl (2014), the concept of a data transformation framework is developed on the contextual base of the formal interpretation of semi-structured schemata as regular tree grammars. As the content of the terminal nodes has been identified to be often not in an atomic form, the rule framework is applied on contained data in order to describe implicitly existing patterns and translate the data into semantically enriched forms.

After the context and motivation of the thesis are introduced in chapter 2, an overview of the foundational building blocks of formal language theory, language applications and the theoretical findings of Gradl (2014) are presented in chapter 3. As practical foundation for the implementation of the framework, the parser generator ANTLR is introduced in chapter 4.

The concept presented in chapter 5 of this thesis first introduces the fundamental problems that were intended to be solved by a developed rule framework. Based on the definition of the overall language processing pipeline required to parse

and transform provided input strings, the conceptual work distinguishes two primary phases of the framework: *data description* is primarily characterized by the concept of individually specific DSLs—i.e. each terminal element of the schema-level perspective can be defined in terms of a specifically designed DSL. The facilitation of DSLs provides specificity benefits and results in condensed and easily understandable grammars—especially when compared to a *global data language* approach. The second phase, *data transformation*, utilizes the semantically enriched form of the input, which is presented in terms of ASTs to finally transform data to defined output values. Data transformation in this thesis is based on a specifically designed data transformation language, which is based on the best-practices of DSL abstractions presented in Ghosh (2011, 23). After a presentation of the required runtime behavior of the rule framework, the concept concludes and leads to a first prototypical implementation of the rule framework and a supporting web application in chapter 6.

Chapter 7 provides ideas of use-cases and illustrates the qualitative applicability of the concept. Especially when integrating advanced aspects of ANTLR such as semantic predicates or when composing multiple sequential language application pipelines, the capabilities of the rule framework are significantly extended by declaratively specifying correlations and languages.

Future work

As this thesis could only cover certain aspects and use-cases with respect to data integration, many specific questions remained unasked and provide potential for future research. Some of the most enticing ideas could be summarized as:

- The thesis focused on context-related aspects of data integration—abstracting from technical heterogeneity. Due to the generic implementation, the rule framework might qualify to implement transformations on the M2 layer of the modeling architecture presented in figure 3.3 on page 24 and as such the rule based integration of protocol-dependent schema languages with the internal representation.
- Semantic enrichment as labeling functions and mappings were the focused use-cases in this thesis because their implementation is required for the

integration. The concept of the rule framework could, however, be used for many other tasks such as highlighting output according to syntactic rules, duplicate detection etc.

- User interfaces for the support of specifying grammars or transformation functions can be interpreted as just another language, which could be transformed to the target language—i.e. the grammar or function respectively. A analogy is found in C++ compilation, which is based on the initial translation of C++ to C (see figure 8.1). The utilization of simpler (visual) languages could follow the same principle.

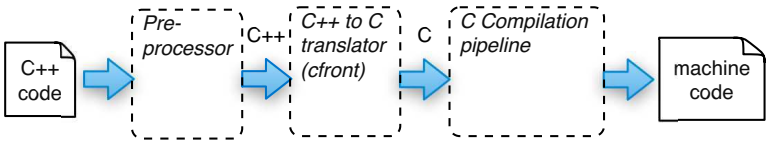


Figure 8.1: C++ compilation pipeline (Parr 2010, 33-34)

Bibliography

Atkinson, C; Kühne, T (2003): *Model-driven development.* In: IEEE Software, 20 (5):36–41.

Batini, C; Lenzerini, M; Navathe, SB (1986): *A comparative analysis of methodologies for database schema integration.* In: ACM Computing Surveys, 18(4):323–364.

Buneman, P (1997): *Semistructured Data.* In: Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '97,), p. 117–121, New York, NY, USA. ACM. doi:10.1145/263661.263675. http://doi.acm.org/10.1145/263661.263675.

Burnard, L; Bauman, S (2014): *TEI P5: Guidelines for Electronic Text Encoding and Interchange*, 2014. http://www.tei-c.org/release/doc/tei-p5-doc/en/Guidelines.pdf, (Last checked: 2014-06-28).

Bézivin, J (2005): *On the unification power of models.* In: Software & Systems Modeling, 4(2):171–188.

Chomsky, N (1956): *Three models for the description of language.* In: IRE Transactions on Information Theory, 2:113–124. http://www.chomsky.info/articles/195609–.pdf – last visited th$ January 2009.

Cooper, KD (2012): *Engineering a compiler.* 2nd ed, Elsevier/Morgan Kaufmann, Amsterdam ; Boston. ISBN:9780120884780.

Crespi-Reghizzi, S (2009): *Formal languages and compilation* (Texts in computer science.). Springer, London. ISBN:9781848820494.

DCMI (June 2012): *Dublin Core Metadata Element Set, Version 1.1*, June 2012. http://dublincore.org/documents/dces/, (Last checked: 2014-06-28).

Ghosh, D (2011): *DSLs in action.* Manning, Greenwich, Conn. ISBN:9781935182450.

Gradl, T (January 2014): *Towards the federation of digital collections within the arts and humanities: Concept of a generic integration model for semi-structured research data.* Project thesis, University of Bamberg, Bamberg.

Hagedorn, K (2003): *OAIster: a "no dead ends" OAI service provider.* In: Library Hi Tech, 21(2):170–181.

ISO/IEC (December 1996): *ISO/IEC 14977:1996. Information technology – Syntactic metalanguage – Extended BNF*, December 1996. http://www.iso.org/iso/catalogue_detail.htm?csnumber=26153, (Last checked: 2014-05-08).

Lagoze, C; Van de Sompel, H; Nelson, M; Warner, S (June 2002): *The Open Archives Initiative Protocol for Metadata Harvesting, Protocol Version 2.0*, June 2002. http://www.openarchives.org/OAI/openarchivesprotocol.html, (Last checked: 2014-06-28).

Lenzerini, M (2002): *Data integration: a theoretical perspective.* p. 233. ACM Press, 2002. doi:10.1145/543613.543644. http://portal.acm.org/citation.cfm?doid=543613.543644.

Leser, U; Naumann, F (2007): *Informationsintegration: Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen.* Dpunkt-Verl., Heidelberg. ISBN:3898644006 9783898644006.

Liu, S; Zou, Q; Chu, WW (2004): *Configurable Indexing and Ranking for XML Information Retrieval.* In: Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '04,), p. 88–95, New York, NY, USA. ACM. doi:10.1145/1008992.1009010. http://doi.acm.org/10.1145/1008992.1009010.

Murata, M; Lee, D; Mani, M; Kawaguchi, K (November 2005): *Taxonomy of XML Schema Languages Using Formal Language Theory.* In: ACM Trans. Internet Technol., 5(4):660–704. doi:10.1145/1111627.1111631. http://doi.acm.org/10.1145/1111627.1111631.

Parkes, A (2008): *A concise introduction to languages and machines* (Undergraduate topics in computer science.). Springer, New York ; London. ISBN:9781848001206.

Parr, T (August 1993): *Obtaining Practical Variants of LL(k) and LR(k) for k>1 by Splitting the Atomic k-Tuple.* Phd thesis, Purdue University, Lafayette, Indiana. http://www.antlr.org/papers/parr.phd.thesis.pdf.

Parr, T (2010): *Language implementation patterns: create your own domain-specific and general programming languages* (The pragmatic programmers.). Pragmatic Bookshelf, Raleigh, N.C. ISBN:9781934356456.

Parr, T (2013): *The definitive ANTLR 4 reference*. The Pragmatic Bookshelf, Dallas, Texas. ISBN:9781934356999 1934356999.

Parr, T; Fisher, K (2011): *LL(*): The Foundation of the ANTLR Parser Generator*. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11,), p. 425–436, New York, NY, USA. ACM. doi:10.1145/1993498.1993548. http://doi.acm.org/10.1145/1993498.1993548.

Polfreman, M (2005): *Commonly-used metadata formats in the Arts and Humanities*. http://www.ahds.ac.uk/metadata/arts-humanities-metadata-formats.htm.

Poulovassilis, A (2009): *Data Integration Architectures and Methodology for the Life Sciences*. In: Liu, L; Özsu, MT (Eds.): Encyclopedia of database systems (Springer reference,). Springer, New York, 2009. ISBN:9780387355443.

Sheth, AP; Larson, JA (September 1990): *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases*. In: ACM Comput. Surv., 22 (3):183–236. doi:10.1145/96602.96604. http://doi.acm.org/10.1145/96602.96604.

Vierkant, P (2013): *Leuchttürme der deutschen Repositorienlandschaft*. http://de.slideshare.net/paulvierkant/leuchttrme-der-deutschen-repositorienlandschaft.

Zhang, Z; Shi, P; Che, H; Gu, J (August 2008): *An Algebraic Framework for Schema Matching*. In: Informatica, 19(3):421–446. ISSN:0868-4952. http://dl.acm.org/citation.cfm?id=1454341.1454348.

Eidesstattliche Erklärung

Ich erkläre hiermit gemäß §16 der Prüfungsordnung für den Masterstudiengang VAWi, dass ich die vorliegende Masterarbeit mit dem Titel „Concept and implementation of a rule framework to dynamically transform data and queries for heterogeneous collections“ selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt, Zitate kenntlich gemacht und die Arbeit noch keiner anderen Stelle zu Prüfungszwecken vorgelegt habe.

Ort, Datum

Tobias Gradl