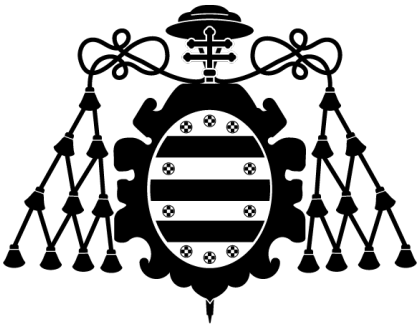
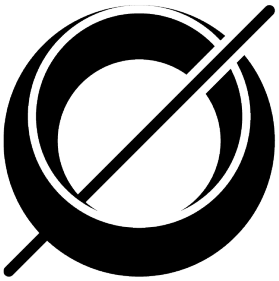
UNIVERSITY OF OVIEDO





SCHOOL OF SOFTWARE ENGINEERING

FINAL DEGREE PROJECT

"shex-lite: Automatic generation of domain object models through a subset of a shape expressions language"

Vº Bº Project Director

DIRECTORS: Dr. Jose E. Labra Gayo,
Daniel Fernández Álvarez.

AUTOR: Guillermo Facundo Colunga

Final Degree Project

ShEx-Lite

Automatic generation of domain object models through a subset of a Shape Expressions Compact Syntax.

Guillermo Facundo Colunga

June 4, 2020

School of Computer Science
University of Oviedo

The harmony of the world is made manifest in Form and Number, and the heart and soul and all the poetry of Natural Philosophy are embodied in the concept of mathematical beauty.

– D'Arcy Wentworth Thompson

Aknowledgments

There are many people thanks to whom I write this work today. First of all I would like to thank my teachers Dr. Jose Emilio Labra Gayo and Daniel Fernández Álvarez, as well as the semantic web research group of the University of Oviedo for the trust placed in me to carry out this project. However, this project is done as a summary of what has been my time at the university and a personal stage. That is why I would like to make a small dedication to those people who in one way or another have helped me to write this project today. To my mother, Esther, for waiting until I was ready to leave us. To my uncle Andrés for being my guardian angel. To my grandparents, for giving me everything they had. To my coach Vic, who unwittingly has become a father to me. To Laura, the impossible girl. Ricardo, although I have never counted on you, you have always been. To Monica, for being you. To Pablo, without you I would not be who I am and I would never have finished this degree. To Álvaro, for letting me be your “manín". To Sari, for laughing at me when I needed it. To Cotito, for teaching me to accept myself. To Alejandro, for getting me out of my comfort zone. To Pablín, for being my prettiest doctor. To all my friends and family, those who I did not mention and those that I left along the way I dedicate this work to you for the moments you give me. And finally to my CAU family, thank you, really. I hope that this work and I will not be a disappointment to any of you, really, each and every one of you are exceptional.

Spanish

Son muchas las personas gracias a las que hoy escribo este trabajo. En primer lugar me gustaría agradecer a mis tutores Aquilino Juan Fuerte y Jose Emilio Labra Gayo, así como al grupo de investigación de web semántica de la Universidad de Oviedo y a Izertis S.A. la confianza depositada en mi para la realización de este proyecto. Sin embargo este proyecto se realiza como resumen de lo que ha sido mi paso por la universidad y por una etapa personal. Es por eso que me gustaría realizar una pequeña dedicatoria a aquellas personas que de una forma u otra han ayudado a que este escribiendo este proyecto hoy. A mi madre, Esther, por esperar a que estuviera preparado para marcharse. A mi tío Andrés por haber sido mi angel de la guarda. A mis abuelos, por dejarse la vida en mí. A mi entrenador Vic, quien sin quererlo se ha convertido en un padre. A Laura, la chica imposible. A Ricardo, aunque jamás he contado contigo siempre has estado. A Mónica, por ser tú. A Pablo, sin ti no sería quién soy y jamás habría terminado la carrera. A Álvaro, por dejarme ser tu manín. A Sari, por reñirme cuando lo necesitaba. A Cotito, por ser enseñarme a aceptarme. A Alejandro, por sacarme de mi zona de confort. A Pablín, por ser mi médico más cuqui. A todos mis amigos y amigas que dejé por el

camino os dedico este trabajo por los momentos que me distéis. Y finalmente a mi familia del CAU, gracias, de verdad. Espero que este trabajo y yo no seamos una decepción para ninguno de vosotros, de verdad, todos y cada uno sois excepcionales.

Abstract

This end of degree project is about creating a compiler for a subset of the Shape Expressions Compact Syntax, focused on syntactic and semantic validation and the generation of domain models in object oriented languages.

Completar...

Contents

List of Figures

List of Tables

Introduction 1

This project was born in the bar of a pub where the parents of RDF graph validation were talking about creating a tool that would allow people who were not computer-scientist to get started and later work with RDF graph validation schemes.

1.1 Motivation

Each day more and more devices generate data both automatically and manually, and also each day the development of application in different domains that are backed by databases and expose these data to the web becomes easier. The amount and diversity of data produced clearly exceeds our capacity to consume it.

[1]: Chen et al. (2014), 'Big data: A survey'

To describe the data that is so large and complex that traditional data processing applications can't handle the term big data [1] has emerged. Big data has been described by at least three words starting by V: volume, velocity, variety. Although volume and velocity are the most visible features, variety is a key concept which prevents data integration and generates lots of interoperability problems.

In order to solve this key concept RDF (Resource Description Framework) was proposed as a graph-based data model [2] which became part of the Semantic Web [3] vision. Its reliance on the global nature of URIs[1] offered a solution to the data integration problem as RDF datasets produced by different means can seamlessly be integrated with other data.

[2]: Levene et al. (1995), 'A graph-based data model and its ramifications'

[3]: Berners-Lee et al. (2001), 'The semantic web'

1: A Uniform Resource Identifier (URI) is a string of characters that unambiguously identifies a particular resource. To guarantee uniformity, all URIs follow a predefined set of syntax rules, but also maintain extensibility through a separately defined hierarchical naming scheme. Ref.https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

Also, and related to his is the concept of Linked Data [4] that was proposed as a set of best practices to publish data on the Web. It was introduced by Tim Berners-Lee and was based on four main principles:

[4]: Heath et al. (2011), 'Linked data: Evolving the web into a global data space'

- Use URIs as names for things.
- Use HTTP URIs so that people can look up those names.
- When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
- Include links to other URIs. so that they can discover more things.

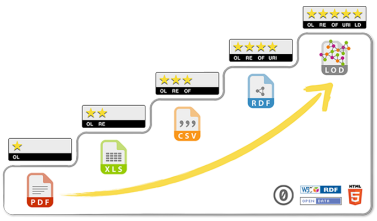


Figure 1.1: The 5 star steps of Linked Data.

This four principles are called the 5 stars Linked Open Data Model, illustrated in Figure 1.1. RDF is mentioned in the third principle as one of the standards that provides useful information. The goal of this principles is that data is not only ready for humans to navigate through but also for other agents, like computers, that may automatically process that data.

[5]: Gayo et al. (2017), 'Validating RDF data'

All the above motivations helped to make RDF the language for the Web of Data, as described in [5] . And the main features that it presents are: Disambiguation, Integration, Extensibility, Flexibility and Open by Default. All this concepts will be deeply explored in Section 2.1, but with the features also some drawbacks are associated, the most important one and the one we will focus is the RDF production/consumption dilema.

RDF production/consumption dilema states that it is necessary to find ways that data producers can generate their data so it can be handled by potential consumers. For example, they may want to declare that some nodes have some properties with some specific values. Data consumers need to know that structure to develop applications to consume the data.

Although RDF is a very flexible schema-less language, enterprise and industrial applications may require an extra level of validation before processing for several reasons like security, performance, etc.

[6]: Ryman et al. (2013), 'OSLC Resource Shape: A language for defining constraints on Linked Data.'

To solve that dilema and as an alternative to expecting the data to have some structure without validation, Shape Expressions (ShEx) where proposed as a human-readable and high-level open source language for RDF validation. Initially ShEx was proposed as a human-readable syntax for OSLC Resource Shapes [6] but ShEx grew very fast to embrace more complex user requirements coming from clinical and library use cases.

Another technology, SPIN, was used for RDF validation, principally in TopQuadrant's TopBraid Composer. This technology, influenced from OSLC Resource Shapes as well, evolved into both a private implementation and open source definition of the Shapes Constraint Language (SHACL), which was adopted by the W3C Data Shapes Working Group.

From a user point of view the possibilities of ShEx are very large, from the smallest case to just validate a node with one property to a scientific domain case where we need to validate the human genome (a real use case of ShEx). As seen, ShEx is a new powerful language, but it can became complicated on the corner cases, but most of day-to-day uses can be solved with a subset of the language. This is the point where this project borns. We will call this subset ShEx-Lite. The simplicity of ShEx-Lite is not only focus on computer

scientists who have experience the pain of new languages but also for other non-technical profiles that need to validate RDF data.

1.2 Main usage scenarios

Section 1.1 introduced some profiles that might benefit from using ShEx-Lite. We can find an example of this profiles in the Wikidata Community. Wikidata is formed by a multidisciplinar community whose aim is to introduce RDF data in to an open knowledge base used by other companies like Google Search. The only problem is that the introduced RDF data needs validation to ensure a minimum data quality, but the profiles that introduce the data, usually, are domain experts whose knowledge about computer science is limited.

Extender ejemplo wikidata.

Besides to this, a common problem is that some companies like Wikidata or even Universities use ShEx to define the constraints of the RDF data that they own. But then, when developing applications with object oriented languages they need to translate those schemas in to a domain model to support their data. Furthermore if the Shape Expressions used to validate their data changes for some reason they need to rewrite that domain model in the OOL again.

Adornar un poco.

Finally, from a ShEx developer point of view sometimes appears the need to try new features in a small playground that allow easy an fast testing, for example a feature that appeared after this project was implemented is to automatically generate documentation webpages for the schemas defined in ShEx, but the first target of this feature won't be ShEx, will be ShEx-Lite as it is perfect for he proof of concept.

Enumerar tipos de er-ramientas que se benefi-ciarían.

1.3 Content of the proposal

After Section 1.1 and Section 1.2 this section descibes the developed system to solve the deficiencies and different profile-users requests.

First A compiler for a language defined as a subset of the shape expressions language focused on helping the non-expert user on solving problems with their schemas.

Secondly A functionality in this compiler, that allows to automatically create domain object models in object-oriented programming languages, from the defined schemas.

ShEx-Lite Compiler

[7]: (), *Shape Expressions Language 2.1*

This compiler works over a defined subset of the Shape Expressions Compact Syntax, defined at [7] that allows expressing basic constraints. It is implemented with the paradigm "compiler as a library" and it is able to parse a schema, analyze it and generate the syntactic and semantic errors that the schema contains.

The ShEx-Lite Compiler is composed of the following components:

Syntax analysis

The syntax analysis phase covers the transformation of the input in to an Abstract Syntax Tree. That is, lex and parse the file, generate the parse tree, raise any errors or warnings and finally build the AST.

Semantic analysis

The semantic analysis covers the validation and transformation of the AST in to the SIL (ShEx-Lite Intermediate Language). Is during this stage where the AST gets validated, type-checked and transformed from a tree to a graph, is this graph the one that gets the name of SIL.

Automatic generation of domain object models

[8]: (2020), *List of object-oriented programming languages*

[9]: (), *Hércules - Universidad de Murcia*

[10]: (2019), *Plain Old Java Object*

But by far, the biggest difference with existing tools, is the automatic generation of domain object models from the schemas defined.

The idea behind this is to enhance interoperability between object oriented languages [8] and RDF systems. An example of this is the European Project ASIO Hércules [9] , where the automatic transformation of schemas in to POJOs [10] is the tool that joins the Semantic Architecture and the Ontology Infrastructure.

Also it is important to remark here that we are perfectly conscious about the fact that not every object oriented language allows to model exactly the same restrictions as types differ, therefore each OOL needs to validate or map the schema to a representation on the language whose meaning is the same, that is create the image of the schema in the corresponding language.

1.4 Structure

The dissertation layout is as follows:

Chapter 2 Indicates the state of the art of the existing RDF validation technologies, tools for processing Shape Expressions and other related projects.

Chapter 3 Describes the goals that the project aim to achieve after its execution and possible real-world applications.

Chapter 4 Contains a detailed initial planning and budget for the project, this is the designed planning followed during the execution of the project and the initial estimated budget.

Chapter 5 Gives a basic theoretical background that it is needed to fully understand the concepts explained in the following chapters.

Chapter 6 Provides a technical description of the design and implementation of the compiler itself. This includes, analysis, design, the technological stack choices, diagrams, implementation decisions and tests.

Chapter 7 Compares the initial planning developed in chapter 4 with the final one. This includes the genuine execution planning of the project and the reasons and events that modified the one from chapter 4.

Chapter 8 Summarizes the analysis and results given over the project, gives an outlook for future work continuing the development of the implemented solution. And includes the diffusion of results done during the project.

Chapter 9 Includes all the set of references used during this document. It is fully recommended to read them carefully and use them as source of truth for any doubt.

Chapter 10 Attaches every document related to the project and referenced from other chapters that has been developed during the project. Here we include detailed budget, system manuals, and other documents.

Theoretical Background 2

For a proper understanding of this documentation and the ideas explained on it it is needed to know some theoretical concepts that are the fundaments of Linked Data, RDF, RDF Validation, programing languages and compilers. A more detailed view of the concepts presented here is offered in [5, 11, 12] .

[5]: Gayo et al. (2017), 'Validating RDF data'
[11]: Prud'hommeaux et al. (2014), 'Shape expressions: an RDF validation and transformation language'
[12]: (2020), *Programming language*

2.1 RDF

RDF started in 1998 and the first version of the specification was published in 2004 by the W3C and according to [13] RDF is a standard model for data interchange on the Web. RDF has features that facilitate data merging even if the underlying schemas differ, and it specifically supports the evolution of schemas over time without requiring all the data consumers to be changed. Another important feature is that RDF supports XML, N-Triples and Turtle syntax, the Figure 2.1 shows an example of how a triplet can be written in RDF N-Triples Syntax. RDF extends the linking structure of the Web to use URIs to name the relationship between things as well as the two ends of the link (this is usually referred to as a "triple" or "triplet"). Using this simple model, it allows structured and semi-structured data to be mixed, exposed, and shared across different applications. This linking structure forms a directed, labeled graph, where the edges represent the named link between two resources, represented by the graph nodes. This graph view is the easiest possible mental model for RDF and is often used in easy-to-understand visual explanations. Also, related to this we strongly recommend the Tim Berners-Lee's writings on Web Design Issues where he explain the issues of the liked data and why is RDF so important.

[13]: Manola et al. (2004), 'RDF primer'

check reference

```
<http://example/subject1> <http://example/predicate1> <http
    ://example/object1>
```

Figure 2.1: RDF N-Triples Example. From this example we can see that each triplet is composed of three elements, the subject the predicate and the object.

2.2 Validating RDF

In the previous point we just see that easiest possible mental model for RDF is a graph, and that's correct. At the end RDF represents a graph. And with the ability of representing and storing data emerges the need to validate that the schema of the graph is correct. At the time RDF was introduced, as it was based on XML the usage of XML-Schema. But this was a very esoteric way of doing it. In order to solve this problem in 2017 the book Validating RDF Data [**gayo2017validating**] was published. In the book different alternatives for validating RDF are explained under Chapters 3, 4 and 5. But also describes the possible applications of RDF validation in Chapter 6 and finally in Chapter 7 they make an small comparison about RDF validation technologies. The most wide used validation technology is Shape Expressions even though the W3C standard points to SHACL which is was based in the Shape Expressions: An RDF validation and transformation language [**prud2014shape**] paper. That is the main reason why this technology will be the one that we will explore deeply.

[**gayo2017validating**]: **gayo2017validating** (**gayo2017validating**), **gayo2017validating**

[**prud2014shape**]: **prud2014shape** (**prud2014shape**), **prud2014shape**

Shape Expressions

Link references.

As defined in [2] Shape Expressions (ShEx) is a schema language for describing RDF graphs structures. ShEx was originally developed in late 2013 to provide a human-readable syntax for OSLC Resource Shapes. It added disjunctions, so it was more expressive than Resource Shapes. Tokens in the language were adopted from Turtle and SPARQL with tokens for grouping, repetition and wildcards from regular expression and RelaxNG Compact Syntax [16]. The language was described in a paper [1] and codified in a June 2014 W3C member submission [17] which included a primer and a semantics specification. This was later deemed "ShEx 1.0". The W3C Data Shapes Working group started in September 2014 and quickly coalesced into two groups: the ShEx camp and the SHACL camp. In 2016, the ShEx camp split from the Data Shapes Working Group to form a ShEx Community Group (CG). In April of 2017, the ShEx CG released ShEx 2 with a primer, a semantic specification and a test-suite with implementation reports. As of publication, the ShEx Community Group was starting work on ShEx 2.1 to add features like value comparison and unique keys. See the ShEx Homepage `http://shex.io/` for the state of the art in ShEx. A collection of ShEx schemas has also been started at `https://github.com/shexSpec/schemas`.

Use of ShEx

Strictly speaking, a ShEx schema defines a set of graphs. This can be used for many purposes, including communicating data structures associated with some process or interface, generating or validating data, or driving user interface generation and navigation. At the core of all of these use cases is the notion of conformance with schema. Even one is using ShEx to create forms, the goal is to accept and present data which is valid with respect to a schema. ShEx has several serialization formats:

- a concise, human-readable compact syntax (ShExC);
- a JSON-LD syntax (ShExJ) which serves as an abstract syntax; and
- an RDF representation (ShExR) derived from the JSON-LD syntax.

These are all isomorphic and most implementations can map from one to another. Tools that derive schemas by inspection or translate them from other schema languages typically generate ShExJ. Interactions with users, e.g., in specifications are almost always in the compact syntax ShExC. As a practical example, in HL7 FHIR, ShExJ schemas are automatically generated from other formats, and presented to the end user using compact syntax. See Section 6.2.3 for more details. ShExR allows to use RDF tools to manage schemas, e.g., doing a SPARQL query to find out whether an organization is using dc:creator with a string, a foaf:Person, or even whether an organization is consistent about it.

ShEx Implementations

Check links.

At the time of this writing, we are aware of the following implementations of ShEx.

- shex.js for Javascript/N3.js (Eric Prud'hommeaux) https://github.com/shexSpec/shex.js/;

```
PREFIX :       <http://example.org/>
PREFIX schema: <http://schema.org/>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>

:User {
  schema:name          xsd:string  ;
  schema:birthDate     xsd:date?  ;
  schema:gender        [ schema:Male schema:Female ] OR xsd
    :string ;
  schema:knows         IRI @:User*
}
```

Figure 2.2: Shape Expression Example. This example describes a shape expression that describes a user as a node that has one name of type string, an optional bithd date of type date, one gender of type Male, Female or free string and a set between 0 and infinite of other users represented by the knows property.

- Shaclex for Scala/Jena (Jose Emilio Labra Gayo) https://github.com/labra/shaclex/;
- shex.rb for Ruby/RDF.rb (Gregg Kellogg) https://github.com/ruby-rdf/shex;
- Java ShEx for Java/Jena (Iovka Boneva/University of Lille) https://gforge.inria.fr/projects/shex-impl/; and
- ShExkell for Haskell (Sergio Iván Franco and Weso Research Group) https://github.com/weso/shexkell.

There are also several online demos and tools that can be used to experiment with ShEx.

- shex.js (http://rawgit.com/shexSpec/shex.js/master/doc/shex-simple.html);
- Shaclex (http://shaclex.herokuapp.com); and
- ShExValidata (for ShEx 1.0) (https://www.w3.org/2015/03/ShExValidata/).

Other Technologies

As other validation technologies we will just explore the existence of them as it is very interesting to know how other tools approach the same issue.

SHACL

Also in [2], Chapter 5, it is fully explained that Shapes Constraint Language (SHACL) has been developed by the W3C RDF Data Shapes Working Group, which was chartered in 2014 with the goal to "produce a language for defining structural constraints on RDF graphs [18]." The main difference that made us choose ShEx over SHACL are that ShEx emphasized human readability, with a compact grammar that follows traditional language design principles and a compact syntax evolved from Turtle.

JSON Schema

JSON Schema born as a way to validate JSON-LD, and as turtle and RDF can be serialized as JSON-LD it is usual to think that JSON Schema can validate RDF data, but this is not fully correct. And the reason is that the serialization of RDF data in to JSON-LD is not deterministic, that means that a single schema might have multiple serializations, which interferes with the validation as you cannot define a relative schema.

2.3 Programming Languages

According to [20] "a programming language is a formal language comprising a set of instructions that produce various kinds of output." When we talk about programming languages we need to know that they are split into two, General Purpose Languages (GPL) and Domain Specific Languages (DSL). The main difference overtime is that, as said in [21], a domain-specific language (DSL) is a computer language specialized to a particular application domain in contrast to a general-purpose language (GPL), which is broadly applicable across domains. In the specific case of ShEx-Lite we will be talking about a Domain Specific Language, and more deep we would classified it as a Declarative one, that means that it is not Touring Complete [22].

2.4 Compilers

A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language). Is during this translation process where the compiler validates the syntax and the semantics of the program, if any error is detected in the process the compiler raises an exception (understand as a compiler event that avoids the compiler to continue its execution).

Internal Structure

In order to decompose the internal structure of a compiler they have been split in to the most common task they do (Figure 4), of course this doesn't mean that there are compilers with more or less stages, but at the end everything can be group into any of the groups that we will explain:

Lexycal Analyzer

The lexical analyzer task is to get the input and split it in to tokens [23], which are build from lexemes. If the compiler cannot find a valid token for some lexemes in the source code will generate an error, as the input cannot be recognized.

Syntactic Analyzer

The syntactic analyzer takes the tokens generated during the lexical analysis and parses them in such a way that try's to group tokens so the conform to the language grammar rules. During this stage if there is any error while trying to group the tokens then the compiler will rise an error as the input cannot be parsed.

Semantic Analyzer

The semantic analyzer has two main tasks, usually. First it validates that the source code semantics are correct, for example 4 + "aaa" would not make sense. And the second task is to transform the Abstract Syntax Tree in to a type-checked and annotated AST. Usually that means relate the invocations and variables to its definition, very useful for type-checking.

Code Generator

The task of the code generator as its name indicates is to generate the target code, it can be byte code, machine code or even another high-language code.

Code Optimizer

The code optimizer is the last step before the final target code is generated, it rewrites the code that the code generator produced without changing the semantics of the program, its aim is just to make code faster. At [24] you can see an example of some optimizations that can be done at compile time to make your code faster.

Conventional Compilers

Conventional compiler are a big monolith where each stage (Figure 4) is executed automatically after the previous stage, we can think about them as a car wash tunnel where you put the input and you cannot touch anything until your car is out on the other side of the wash tunnel. From the name we can deduce that that was the "old-fashion" way of implementing compilers, but they present some drawbacks:

- A poor IDE [25] integration. IDE's need to perform incremental compilations in matter of nanoseconds so the user doesn't feel lag when typing the program. With conventional compilers as you need to go through all the compilation process at once they where very slow and companies like Microsoft need to develop different compilers, one for the IDE and another for the final compilation of the program itself. This lead to several problems like that if a feature gets implemented in the final compilation compiler but not in the IDE one the IDE would not support the feature meanwhile the language would.
- Difficult to debug. As the conventional compilers where a blackbox the only way to test intermediate stages was by throwing an input and waiting the the feature you wanted to test was thrown for that input.

Modern Compilers

After the problems Microsoft had with the C# compiler they decide to rewrite the whole compiler and introduce a concept called "compiler as an API" with Roslyn [15]. This concept has been perfectly accepted and solved many problems. In this concept each stage has an input and an output that can be accessed from outside the compiler and stages can be executed independently on demand. This means that for example if an IDE just want to execute the Lexer the Parser and the Semantic analysis it can. That translates in to speed for the user. Also the second problem is solved as testing individual parts of the compiler is much more easy than the hole compiler at once.

Annexes

A

Heading on Level 0 (chapter)

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

A.1 Heading on Level 1 (section)

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Heading on Level 2 (subsection)

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Heading on Level 3 (subsubsection)

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Heading on Level 4 (paragraph) Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

A.2 Lists

Example for list (itemize)

- First item in a list
- Second item in a list
- Third item in a list
- Fourth item in a list
- Fifth item in a list

Example for list (4*itemize)

- First item in a list
  - First item in a list
    - First item in a list
      - First item in a list
      - Second item in a list
    - Second item in a list
  - Second item in a list
- Second item in a list

Example for list (enumerate)

1. First item in a list
2. Second item in a list
3. Third item in a list
4. Fourth item in a list
5. Fifth item in a list

Example for list (4*enumerate)

1. First item in a list
   a) First item in a list
      i. First item in a list
         A. First item in a list
         B. Second item in a list
      ii. Second item in a list
   b) Second item in a list
2. Second item in a list

Example for list (description)

First item in a list
Second item in a list
Third item in a list
Fourth item in a list
Fifth item in a list

Example for list (4*description)

First item in a list
First item in a list
First item in a list
First item in a list
Second item in a list
Second item in a list
Second item in a list
Second item in a list

Bibliography

Here are the references in citation order.

[1] Min Chen, Shiwen Mao, and Yunhao Liu. 'Big data: A survey'. In: *Mobile networks and applications* 19.2 (2014), pp. 171–209 (cited on page 1).

[2] Mark Levene and George Loizou. 'A graph-based data model and its ramifications'. In: *IEEE Transactions on Knowledge and Data Engineering* 7.5 (1995), pp. 809–823 (cited on page 1).

[3] Tim Berners-Lee, James Hendler, and Ora Lassila. 'The semantic web'. In: *Scientific american* 284.5 (2001), pp. 34–43 (cited on page 1).

[4] Tom Heath and Christian Bizer. 'Linked data: Evolving the web into a global data space'. In: *Synthesis lectures on the semantic web: theory and technology* 1.1 (2011), pp. 1–136 (cited on page 1).

[5] Jose Emilio Labra Gayo et al. 'Validating RDF data'. In: *Synthesis Lectures on Semantic Web: Theory and Technology* 7.1 (2017), pp. 1–328 (cited on pages 2, 7).

[6] Arthur G Ryman, Arnaud Le Hors, and Steve Speicher. 'OSLC Resource Shape: A language for defining constraints on Linked Data.' In: *LDOW* 996 (2013) (cited on page 2).

[7] *Shape Expressions Language 2.1*. URL: https://shex.io/shex-semantics/#shexc (visited on 05/23/2020) (cited on page 4).

[8] *List of object-oriented programming languages*. en. Page Version ID: 950660258. Apr. 2020. URL: https://en.wikipedia.org/w/index.php?title=List_of_object-oriented_programming_languages&oldid=950660258 (visited on 05/25/2020) (cited on page 4).

[9] *Hércules - Universidad de Murcia*. es-ES. Library Catalog: www.um.es. URL: https://www.um.es/web/hercules/ (visited on 05/25/2020) (cited on page 4).

[10] *Plain Old Java Object*. es. Page Version ID: 120228634. Oct. 2019. URL: https://es.wikipedia.org/w/index.php?title=Plain_Old_Java_Object&oldid=120228634 (visited on 05/25/2020) (cited on page 4).

[11] Eric Prud'hommeaux, Jose Emilio Labra Gayo, and Harold Solbrig. 'Shape expressions: an RDF validation and transformation language'. In: *Proceedings of the 10th International Conference on Semantic Systems*. 2014, pp. 32–40 (cited on page 7).

[12] *Programming language*. en. Page Version ID: 958722580. May 2020. URL: https://en.wikipedia.org/w/index.php?title=Programming_language&oldid=958722580 (visited on 05/26/2020) (cited on page 7).

[13] Frank Manola, Eric Miller, Brian McBride, et al. 'RDF primer'. In: *W3C recommendation* 10.1-107 (2004), p. 6 (cited on page 7).

Notation

The next list describes several symbols that will be later used within the body of the document.

c Speed of light in a vacuum inertial frame

h Planck constant

Greek Letters with Pronounciation

Character	Name	Character	Name
α	alpha *AL-fuh*	ν	nu *NEW*
β	beta *BAY-tuh*	ξ, Ξ	xi *KSIGH*
γ, Γ	gamma *GAM-muh*	o	omicron *OM-uh-CRON*
δ, Δ	delta *DEL-tuh*	π, Π	pi *PIE*
ϵ	epsilon *EP-suh-lon*	ρ	rho *ROW*
ζ	zeta *ZAY-tuh*	σ, Σ	sigma *SIG-muh*
η	eta *AY-tuh*	τ	tau *TOW (as in cow)*
θ, Θ	theta *THAY-tuh*	υ, Υ	upsilon *OOP-suh-LON*
ι	iota *eye-OH-tuh*	ϕ, Φ	phi *FEE, or FI (as in hi)*
κ	kappa *KAP-uh*	χ	chi *KI (as in hi)*
λ, Λ	lambda *LAM-duh*	ψ, Ψ	psi *SIGH, or PSIGH*
μ	mu *MEW*	ω, Ω	omega *oh-MAY-guh*

Capitals shown are the ones that differ from Roman capitals.