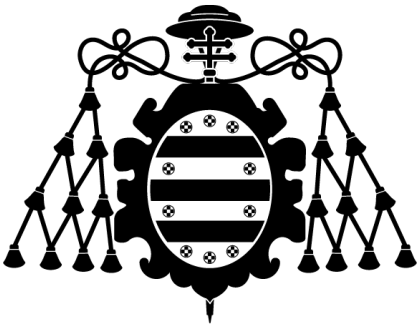
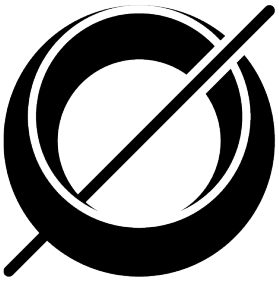
UNIVERSITY OF OVIEDO





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 16, 2020

School of Computer Science
University of Oviedo

ShEx-Lite

Final Degree Project presented on July 2020 at the School of Software Engineering, Oviedo University. All the source code related to the implementation explored during this book is available at `github.com/weso/shex-lite`.

Directors
Dr. Jose Emilio Labra Gayo
Daniel Fernández Álvarez

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

// TODO

Spanish
// TODO

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.

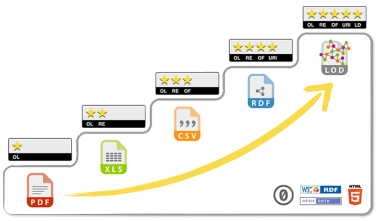


Figure 1.1: The 5 star steps of Linked Data.

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.

Besides to this, a common problem is that some companies 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.

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.

1.2 Main usage scenarios

Section 1.1 introduced some profiles that benefit from using ShEx-Lite.

Non-technical users

We can find an example of this profiles in the Wikidata Community. Wikidata is a free and open knowledge base that can be read and edited by both humans and machines and it acts as central storage for the structured data of its Wikimedia sister projects including Wikipedia, Wikivoyage, Wiktionary, Wikisource, and others. This is very important as even Google Search uses this knowledge base.

But the most important point about Wikidata is its community, who is formed by multidisciplinar profiles whose aim is to introduce data in to an open knowledge base. This data is in RDF format in order to solve the 3Vs problem seen in Section 1.1 but not all of the profiles who introduce data have technical skills, they are what is called domain experts.

For example an archaeologist might be the greatest expert on its domain without having any RDF technical skill and its knowlegde is very valuable for the Wikidata knowlegde base. So Wikidata performs the validation of the RDF introduced by the user to ensure a minimum data quality and if the validation is not sucessful it notifies the user about possible solutions. Currently this is done with Shape Expressions but in order to improve the error messages and a simplify the validation language we propose ShEx-Lite.

Quizás muy directo?

Companies or organizations

Companies and organizations also benefit from ShEx-Lite, they have requested features that allow to sincronise their RDF contraint schemas and the domain object models that they use in their applications. The perfect example of this is the Hércules[7] European project whose aim is to create a management system based on linked data for the research at Spanish Universities. In this project an ontology is created by means of Shape Expressions that model the constraints that the RDF instances of the ontology must meet. After, they need to translate this biug schema in to an object domain model that can be invoked from the java applications they are developing.

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

The feature requested is called "automatic generation of domain object models from shape expressions" and it is fully covered by ShEx-Lite and currently the Hércules project is using it.

ShEx developers

Another very interested profile in ShEx-Lite are the ShEx developers who have the need to implement new features in the ShEx environment but they find that this environment tends to be very complicated and the learning curve is difficult sometimes. For example there is a parallel project of creating online interactive documentation from the comments in the source code of the shape expressions. This is a very interesting project for the ShEx community and thanks to ShEx-Lite the project is now being implemented as a proof of concept.

The benefits of ShEx-Lite are not only limitted to the developers of the language itselft but also to the developers of the ShEx ecosystem tools. For example ShEx IDEs benefit from the API architecture of the ShEx-Lite compiler as they now can use incremental compilation or use only the sintax or semantic valdiation without the waste of time of generating code.

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

This compiler works over a defined subset of the Shape Expressions Compact Syntax, defined at [8] that allows expressing basic constraints. It is implemented with the paradigm "compiler as a library" because the influence of modern compilers like Swift, Rust or Roslyn [9–11] and it is able to parse a schema, analyze it and generate the syntactic and semantic errors that the schema contains.

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

[9]: Inc (), *Swift.org*
[10]: (), *What is rustc? - The rustc book*
[11]: (), *dotnet/roslyn*

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

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 [12] and RDF systems. An example of this is the European Project ASIO Hércules [7] , where the automatic transformation of schemas in to POJOs [13] is the tool that joins the Semantic Architecture and the Ontology Infrastructure.

[12]: (2020), *List of object-oriented programming languages*

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

[13]: (2019), *Plain Old Java Object*

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. This sections is devoted to carefully explain those concepts to the needed deepth to fully understand this dissertation, but for those readers that want a deeper explanation a more detailed view of the concepts presented here is offered in [5, 14, 15] .

[5]: Gayo et al. (2017), 'Validating RDF data'
[14]: Prud'hommeaux et al. (2014), 'Shape expressions: an RDF validation and transformation language'
[15]: (2020), *Programming language*

2.1 RDF

Resource Description Framework (RDF) is a standard model for data interchange on the web, started in 1998 and the first version of the specification was published in 2004 by the W3C according to [16] . 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.

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

```
<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.

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. Figure 2.4 shows an example of how different triples can be use to compose a graph, this graph represents the same as the Figure 2.2

```
<http://example/bob> <http://example/knows> <http://example
    /alice> .
<http://example/alice> <http://example/knows> <http://
    example/peter> .
```

Figure 2.2: RDF N-Triples Graph Example. This exmaple shows the n-triples that generate the graph from Figure 2.4.

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 [17] where he explain the issues of the liked data and why is RDF so important.

[17]: Berners-Lee et al. (1998), *Semantic web road map*

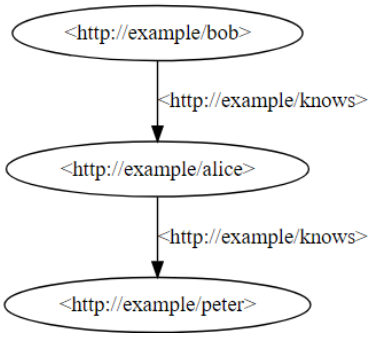


Figure 2.3: RDF Example graph.

2.2 Validating RDF

RDF therefore allows to represent and store data, and with this ability emerges the need to validate that the schema of the graph is correct. In order to perform the validation of RDF data there have been previous attempts, described in Section 2.2, this dissertation will focus on Shape Expressions. But in order to validate RDF data every technology will need to face the following RDF concepts:

- the form of a node (the mechanisms for doing this will be called "node constraints");
- the number of possible arcs incoming/outgoing from a node; and
- the possible values associated with those arcs.

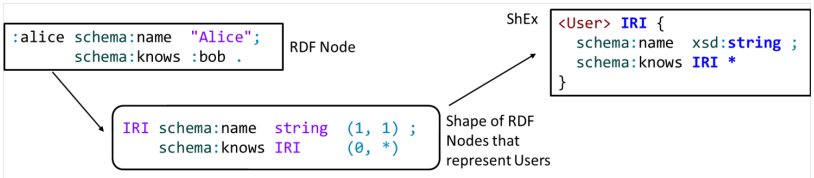


Figure 2.4: RDF node and its shape.

Figure 2.4 illustrates those RDF concepts by means of the Shape Expression that validates users. There we can see that the shape of the RDF node that represents Users represents the form of a node, the number of possible arcs and the possible value associated with those arcs.

Shape Expressions

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

As defined in [5] 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 [18] . The language was described in a paper [14] and codified in a June

[18]: Van der Vlist (2003), *Relax ng: A simpler schema language for xml*

[14]: Prud'hommeaux et al. (2014), 'Shape expressions: an RDF validation and transformation language'

2014 W3C member submission which included a primer and a semantics specification. This was later deemed "ShEx 1.0".

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.

```
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 2.5: 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.

ShEx Compact Syntax: ShExC

The ShEx compact syntax (ShExC) was designed to be read and edited by humans. It follows some conventions which are similar to Turtle or SPARQL.

- `PREFIX` and `BASE` declarations follow the same convention as in Turtle. In the rest of this chapter we will omit prefix declarations for brevity.
- Comments start with a # and continue until the end of line.
- The keyword a identifies the `rdf:type` property.
- Relative and absolute IRIs are enclosed by `<` `>` and prefixed names (a shorter way to write out IRIs) are written with prefix followed by a colon.
- Blank nodes are identified using `_:label` notation.
- Literals can be enclosed by the same quotation conventions (`'`, `"`, `"'`, `"""`) as in Turtle.
- Keywords (apart from a) are not case sensitive. Which means that `MinInclusive` is the same as `MININCLUSIVE`.

A ShExC document declares a ShEx schema. A ShEx schema is a set of labeled shape expressions which are composed of node constraints and shapes. These constrain the permissible values or graph structure around a node in an RDF graph. When we are considering a specific node, we call that node the focus node.

Figure 2.6 shows the first level of a shape expression, we have a label and the shape itself that is what we asing to the `:User`

label. Then, the shape is composed by triple expressions. The triple expression structure is explained in Figure 2.7, and as its name indicates it is composed of three elements, the property, the node constraint and the cardinality.

Shape Expressions Compact Syntax is much bigger and containts other multiple features that give ShEx its power, and all of them can be explored in [5] but they are not needed to understand this dissertation.

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

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.

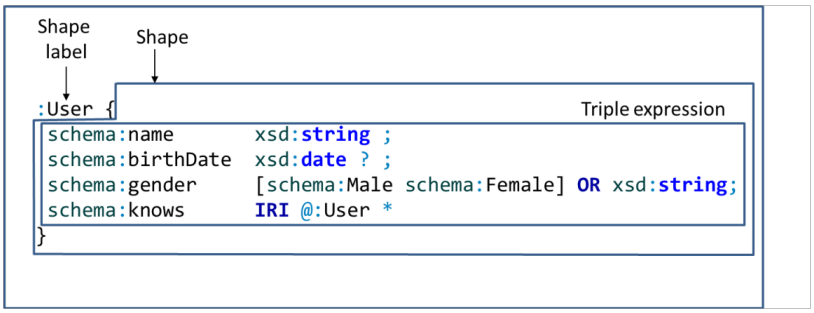


Figure 2.6: Shapes, shape expression labels and triple expressions.

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/;
- 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 [5] , 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 [6] ." The main difference that made us choose ShEx over SHACL are that ShEx emphasized human readability,

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

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

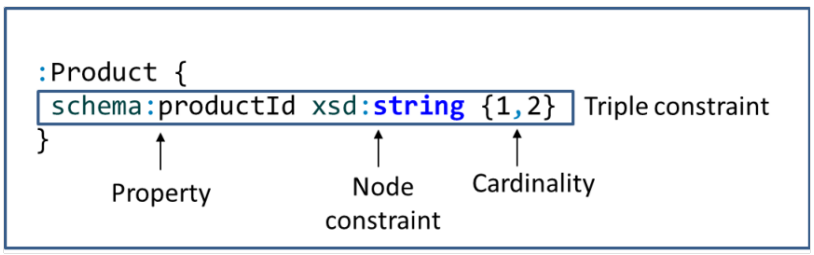


Figure 2.7: Parts of a triple expression.

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

[15]: (2020), *Programming language*

[19]: (2020), *Domain-specific language*

[20]: (2020), *Turing completeness*

According to [15] "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 [19] , 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 [20] .

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 2.8, 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:

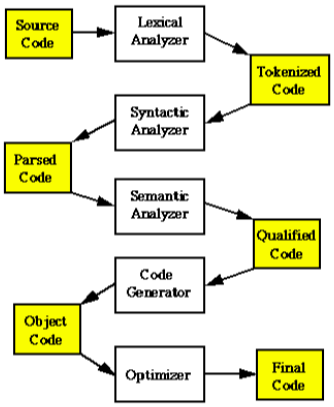


Figure 2.8: Compiler stages.

Lexycal Analyzer

The lexical analyzer task is to get the input and split it in to tokens [21] , 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.

[21]: (2020), *Lexical analysis*

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 [22] you can see an example of some optimizations that can be done at compile time to make your code faster.

[22]: (2020), *Optimizing compiler*

Conventional Compilers

Conventional compiler are a big monolith where each stage Figure 2.8 is executed automatically after the previous stage, if the compiler has eight steps you need to execute them all at once. This approach have been the "old-fashion" but it presents some drawbacks:

[23]: (2020), *Entorno de desarrollo integrado*

- A poor IDE [23] 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 [11] . 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.

[11]: (), *dotnet/roslyn*

Also the second problem is solved as testing individual parts of the compiler is much more easy than the hole compiler at once.

Related Work 3

Some work has already been done in the field of Shape Expressions and RDF validation technologies. In this chapter we will go overthe main studies related to our project, exploring what they have achieved and someof their limitations.

3.1 Simplifications of ShEx

The S language

In 2019 at [24] was defined a language called **S** as a simple abstract language that captures both the essence of ShEx and SHACL. This is very relevant as this language is intended to be the input of a theoretical abstract machine that will be used for graph validation for both ShEx and SHACL. Also in the same paper the authors carefully describe the algorithm for the translation from ShEx to S and from SHACL to S.

[24]: Labra-Gayo et al. (2019), 'Challenges in RDF validation'

Although the theoretical abstract machine has not been implemented yet the intention of the WESO Research Group, where this S language was defined, is to devote more efforts in to this project during the 2021.

Other definition of an abstract language based on uniform schemas can be found at [25] . This language is focused on schemas inference rather on validation, but needs to be taken into account as they also perform an abstraction of both ShEx and SHACL.

[25]: Boneva et al. (2019), 'Semi Automatic Construction of ShEx and SHACL Schemas'

And to the best of our knowledge and after the research process carried out for this project no other language based on a subset of Shape Expressions has been designed nor implemented yet.

ShEx Micro Spec

3.2 ShEx Ecosystem Tools

We already know that ShEx and SHACL have been the two main technologies for RDF validation and some tools emerged around them, we thinks that some of them might benefit from ShEx-Lite. Here we introduce briefly those that had the biggest impact in the community.

Validators

Since the beginning of ShEx and SHACL as languages the RDF community started to build tools that take as input the schemas defined and validate graphs.

This kind of tools can benefit from SHEx-Lite from the point of view that new functionalities can be easily implemented and tested in the lite version of the language before even touching the stable releases of both tools. In the case of ShEx this is more obvious as ShEx-Lite and ShEx are both implemented in Scala and if good design principles are used functionalities can be just migrated and expanded for the rest of the language.

The most important validators are:

Shaclex

1: https://github.com/weso/shaclex

According to the Shaclex[1] official website it is an Open Source Scala pure functional implementation of an RDF Validator that supports both Shape Expressions and SHACL. It was initially developed by Dr. Jose Emilio Labra Gayo and is being maintained by an active community on GitHub. It is used by different projects around the globe and its goal is to validate RDF graphs against schemas defined in Shape Expression or in SHACL. This implementation of a ShEx validator is very important for us as ShEx-Lite is completly inspired by it and aims to transfer the sintactic an semantic validation enhancements to it.

ShEx.js

Another example of os a ShEx validator implementation is `ShEx.js` which is JavaScript based and also open source on GitHub. This implementation is very important for the ShEx community as they defined the serialization of the AST in this implementation as the abstract syntax of ShEx.

IDEs

In order to facilitate the task of writing schemas some engineers decide to implement specific IDEs for the Shape Expressions Language.

This tools will completely benefit from ShEx-Lite and there are currently collaborations in process. At the time they work with Shaclex, which is structured as a conventional compiler, but with the API architecture of ShEx-Lite IDEs can access directly to the

syntactic and semantic modules so features like advances coloring syntax or incremental compilation are available.

YASHE

YASHE[2] (Yet Another ShEx Editor), is a Shape Expressions IDE which started as a fork ofYASQE(which is based on SPARQL). This tool performs lexical and syntactic analysis of the content of the editor, thus offering the user a realtime syntactic error detector. It has features like: syntax highlighting, visual aid elements (tooltips) and autocomplete mechanisms. In addition, it offers a simple way of integrating into other projects.

2: https://github.com/weso/YASHE

Protégé

Protégé is a piece of software developed by the University of Stanford focused on ontology edition. During the last year they added support for Shape Expressions dition on their own software so they became another ShEx IDE.

VSCode

VSCode is a source code light-weight editor developed by Micorsoft and supported by Linux, macOS and Windows. By default this editor does not support any programming language, the way it works is with packages that the community develops and extends the functionality. One of those packages adds support for Shape Expressions Compact syntax and transforms VSCode into a ShEx IDE. This plugin does not add semantic validation and it is a clear target to benefit from ShEx-Lite features.

Others

Other researches focused their efforts in to inferring schemas to existing data sets and creating tools to that evolved from ShEx in order to transform existing data.

Shexer

Shexer[3] is a python library aimed to perform automatic automatic extraction of schemas in both ShEx and SHACL from an RDF input graph. That is if all the other tools take the schemas as the input and validate a graph with it, this tool takes a graph and from it it infers the schemas that it might contain. Its work is fully described in [25, 26] .

3: https://github.com/DaniFdezAlvarez/shexer

[25]: Boneva et al. (2019), 'Semi Automatic Construction of ShEx and SHACL Schemas'
[26]: Fernández-Alvarez et al. (2016), *Inference and serialization of latent graph schemata using shex*

ShExML

4: https://github.com/herminiogg/ShExML

[27]: Garcia-Gonzalez et al. (), 'ShExML: An heterogeneous data mapping language based on ShEx'

ShExML[4] is a language based on ShEx (not a simplification nor an abstraction of ShEx) that can map and merge heterogeneous data formats into a single RDF representation. The main idea behind this tool is written at [27] .

An example of how this different tools can work together thanks to ShEx-Lite would be the following, illustrated at Figure 3.1. Wikidata currently holds millions of registers that do not have any schema that validates them. And they need to make consumer that represents the data in to an object domain model. Without any tool this is just almost impossible, but this shexer you can infer the schemas to ShEx-Lite syntax and with the ShEx-Lite compiler you can automatically create the object domain model in your favorite OOL.

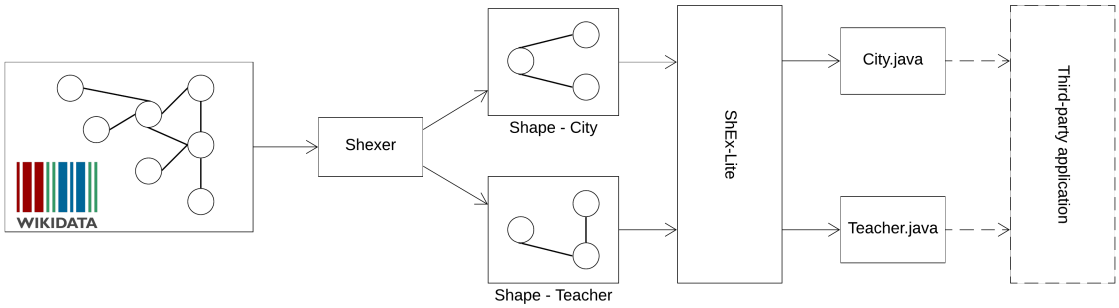


Figure 3.1: ShEx-Lite integration with Shexer for automatically generating java domain object models for the Wikidata schemaless existing data. This shoes the schemaless data from wikidata from which shape expressions are infeered by shexer and later transformed to java plain objects by means of ShEx-Lite so third party apllications can implement the domain model.

System Analysis 4

In this chapter we present the analysis that was carried out in order to design the compiler. First we sumarize the objectives and the target users of the compiler, this will be the starting point of the analysis. After, we will define an scope from which we will stract the use cases and requirements that, finally, will help to identify possible subsystems for the design phase, explained in Chapter 5.

4.1 Objectives

The objectives are those results that we want to achieve or features that we want the compiler to have once it is implemented. The main objectives that we have identified for the compiler are the following ones:

- Provide the following tools from the compiler:
  - Get the ST (syntax tree) from a source file.
  - Get the AST (abstract syntax tree) from a ST.
  - Get the SIL (ShEx-Lite Intermediate Language) from an AST.
  - Generate sources in a given OOL from a SIL.
  - For all the previos functionalities detect and inform or any event considered by the compiler as an error or warning.
- Natively support, at least, Java and Python as target languages for the code generation.
- Allow the previous tools to be called from CLI.
- Allow the previous tools to be encapsulated in other programs written in any Java based language.
- Include examples of how the compiler works with a representative set of shape expressions.

The ShEx-Lite compiler is mean to be a compiler for a syntax based on a subset of the Shape Expressions Compact Syntax. The compiler is aimed at people with a few technical skills that need to write schemas to validate RDF but don't want to learn the full ShEx Language, but also to more experienced developers that would like to automatically generate domain object models and test new functionalities without the drawback of implementing them on a production system.

4.2 Target Users

Every system has some target users, in the case of the SeX-Lite compiler we will focus it on three different types of users:

- People with low technical skills that want to make schemas to validate RDF by means of shape expressions. This people will be able to compile their schemes and check for syntax / semantic errors or warnings through the CLI with a very simple instruction set.
- People with medium technical skills that want to transform a set of shape expressions in to a domain object model automatically. For this people the compiler will provide a manual that will explain each step of the transformation and an usage example.
- ShEx Community developers that want to test a new feature by implementing and testing its behaviors in a controlled small environment. For this people the compiler provides not only the public API but the source code and the corresponding technical documentation.

4.3 Use Cases

The first stage to analyze our system is to see the use cases view, from there we will try to capture posible requirements and with all that information design our system.

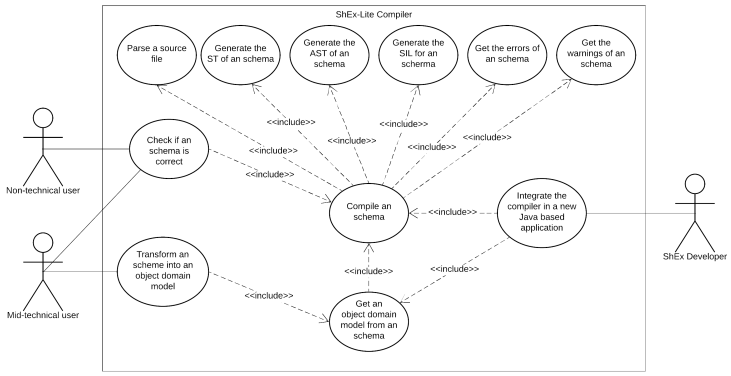


Figure 4.1: Use cases view of the ShEx-Lite compiler for the three different actors of the system.

In Figure 4.1 we can see the high level view of the different use cases that the three different target users of the systems might have, now we will analyze each one individually, some the internal use cases that have no direct interaction with the target user will be explored after.

Use Case Number	1
Description	Check if an schema is correct from the CLI tool.
Actor	Non technical user.
Flow	The actor wants to check if schema is correct or not. If it is not correct wants to see all the warnings / errors. For this purpose the actor introduces the schema in the CLI tool and starts the flow. Once the flow is complete the actor wants to see information that helps to decide if the schema is correct or not.

Table 4.1: Definition of the use case number 1 for the non technical user.

Non technical user

The non technical target user is intended to use only the compiler to validate if the schemas they have defined are syntactically and semantically correct, or if they have any error or warning to be aware of them. Figure 4.2 shows the interaction diagram for this actor, and Table 4.1 describes the represented use case.

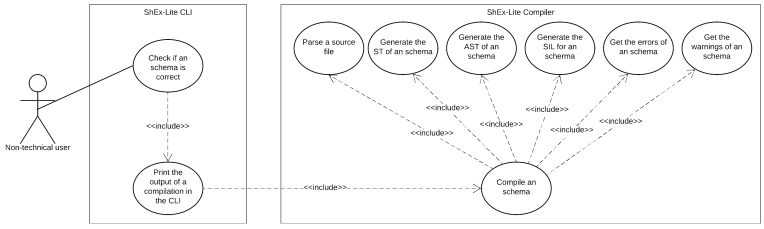


Figure 4.2: Extension of the use case view for a non technical user.

Mid technical user

The use cases expected for a mid technical user are two, in one side the compilation in order to validate if their schemas are syntactically and semantically correct, or if they have any error or warning. And on the other hand, automatically generate the domain object models for the defined schema. For this user we can find the representation of the interaction diagram at Figure 4.3 and both associated use cases descriptions on Table 4.2 and Table 4.3.

ShEx Developer

A ShEx developer is expected to use the compiler in two ways, first to integrate it in to other applications and secondly to generate object domain models for a given schema but not from the CLI, from an API instead. Figure 4.4 shows the interaction diagram of

Table 4.2: Definition of the use case number 2 for the mid technical user.

Use Case Number	2
Description	Generate an object domain model for a given schema from the CLI.
Actor	Mid technical user.
Flow Conditions	The actor wants to generate an object domain model for a given schema from the CLI tool. For this purpose the actor introduces the schema in the CLI tool and starts the flow. Once the flow in complete the actor wants the generated object model to be persist as source files on his computer. For this flow to end correctly it is mandatory that the schema that the user uses to generate the domain object model is correct.

this actor with the compiler and Table 4.4 descrives this use case scenario.

The previous use cases might seem like the system is really simple, but far from truth the fact that the interface with the target users is simple does not define the complexity of the system, as this, lives inside of it. In order to capture this complexity we will proceed now to analyze the requirements that the implemented system must meet.

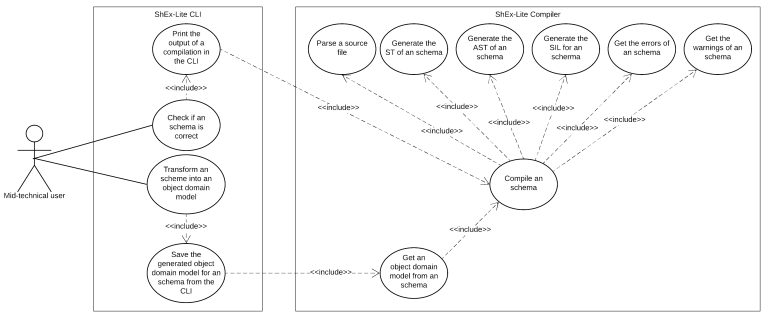


Figure 4.3: Extension of the use case view for a mid technical user.

Use Case Number	3
Description	Check if an schema is correct from the CLI tool.
Actor	Mid technical user.
Flow	The actor wants to check if schema is correct or not. If it is not correct wants to see all the warnings / errors. For this purpose the actor introduces the schema in the CLI tool and starts the flow. Once the flow is complete the actor wants to see information that helps to decide if the schema is correct or not.

Table 4.3: Definition of the use case number 3 for the mid technical user.

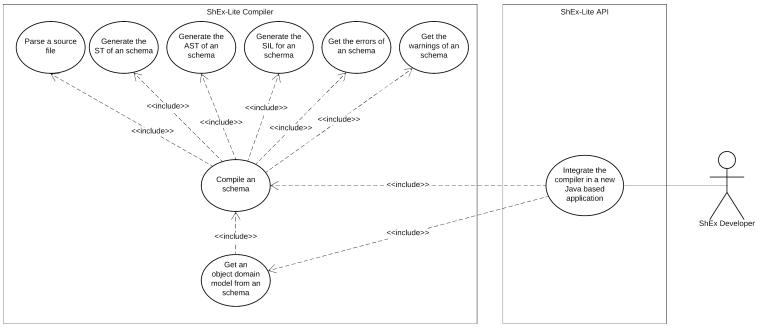


Figure 4.4: Extension of the use case view for a mid technical user.

Use Case Number	4
Description	Integrate the compiler in a new Java based application.
Actor	ShEx developer.
Flow	The actor wants to integrate the compiler in to a new java base application. For that purpose the actor will use a public API that allow the actor to integrate the actions described in the use cases 2 and 3.

Table 4.4: Definition of the use case number 4 for the ShEx developer user.

4.4 Requirements

In this section we are going to enumerate the requirements that we have obtained for the compiler. Since the line that separates functional and non-functional requirements can be sometimes hard to define, we decided to specify the requirements following the taxonomy introduced in the IEEE 830 standard for the specification of software requirements. The following types of requirements will be evaluated:

- External interfaces: Requirements that affect user, hardware, software or communication interfaces.
- Functional requirements: Requirements related to the functions of the system.
- Performance requirements: These requirements are related to the load that the system should tolerate.
- Logical database requirements: These requirements are related to access or constraints with the system's database.
- Design constraints: Constraints imposed by standards, hardware limitations. . .
- Other constraints.
- System attributes: Quality attributes of the system, such as usability, accessibility. . .
- Other requirements that do not belong to any of the categories above.

External Interfaces

Functional Requirements

Performance Requirements

Design Constraints

Other Constraints

System Attributes

Other Requirements

System Design 5

In this section we will explore que design of ShEx-Lite language and compiler, from the most general diagram to a deeper level that gives a clear idea of the architecture of ShEx-Lite compiler.

5.1 ShEx-Lite Language Design

5.2 Compiler Design

There were several elements that conditioned the architecture of this system. First of all, there is no database or data access layer that needs to be taken into account. There isn't also any presentation or visual interface of data other from the CLI. The system consists mainly of functionality that is offered to other users to use from the CLI or create their own systems.

All of these elements simplified noticeably the architecture of the system. Figure 5.1 sumarizes the high level architecture of the ShEx-Lite compiler. Notice that the elements that appear here are the same that where identified in Chapter **??**.

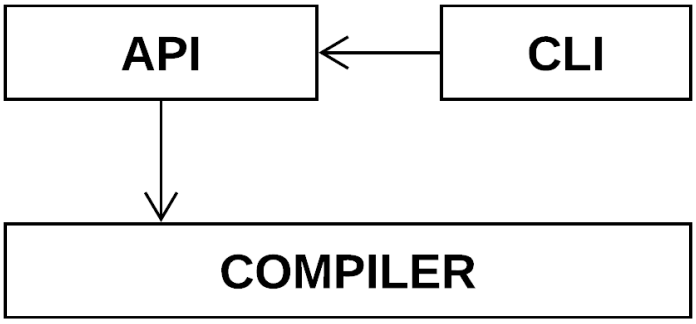


Figure 5.1: High level view of the ShEx-Lite system. This figure shows how the CLI is build on top of the API which depends on the compiler implementation.

Figure 5.2 shows the three main subsystems that the ShEx-Lite compiler contains, the compiler itself, the API composed by the interfaces that the compiler exposes and the CLI subsystem that acts as a client implementation of the API subsystem. At this point it is important to take a moment to explain the architectural pattern that ShExLite employs. From the beginning we introduce the pattern "compiler as an API", this pattern is perfectly explained in the Theoretical Background and it affects the design of the system

in the following way. Each component of the compiler subsystem can be executed independently of the others, exposing an interface and producing and output. This produces a subsystem that does not contain any implementation but requires the highest effort for a good design, the API subsystem. Now we will explore the architecture of the three subsystems independently.

Compiler

The compiler contains the implementation for the API interfaces. Its task is to give the functionality to the designed interfaces, to do so it implements different programming patterns.

The Figure 5.3 shows from a very high point of view a complete sequence of compilation for the ShEx-Lite compiler. From there we can already capture that the compiler will be composed of the following elements:

- **Parse:** Takes as input the source files and produces the Syntax Tree.
- **ASTBuild:** Takes as an input the Syntax Tree and generates an Abstract Syntax Tree.
- **Sema:** From the Abstract Syntax Tree generated in the previous step it produces the ShEx-Lite Intermediate Language.
- **SIL Analysis:** Analyses the ShEx-Lite intermediate code and decides whether the corresponding Intermediate Representation can be dispatched or not.
- **IRGen:** Generates the corresponding Intermediate Representation.

Notice that those elements are independent one from another. Also from the Figure 5.3 we can already extract that the compiler will have at least the following structures:

- **S:** Holds the Sources that the compiler will process. Those sources are `.shexl` files that contain the schemas that the user wants to compile.

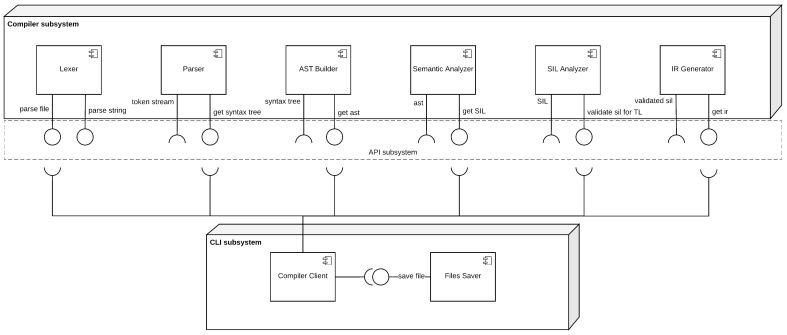


Figure 5.2: Components diagram for ShEx-Lite Compiler.

- **ST:** Holds the output of the parse process. The parse process produces the Syntax Tree but also might generate warnings and errors.
- **AST:** Contains the output of building the AST. This AST has not been type-checked nor annotated.
- **SIL:** Contains the ShEx-Lite Intermediate Language. The SIL is the name for the typechecked annotated AST that has been transformed in to a graph structure. But it might also contains semantic error / warnings.
- **IR:** Contains the generated Intermediate Representation. The IR is the sources that represent the generated domain object model, it contains those sources and all the compilation information.

From the Figure 5.4 we can see that in the compiler each process (Parse, sema, . . .) contains multiple components. For example the parse stage from Figure 5.3 is composed by a lexer and a parser, that will correspond with the first two components of the compiler subsystem from Figure 5.2. This components will be deeply explored at the class diagram section.

Figure 5.3: High level view of the compiler flow.

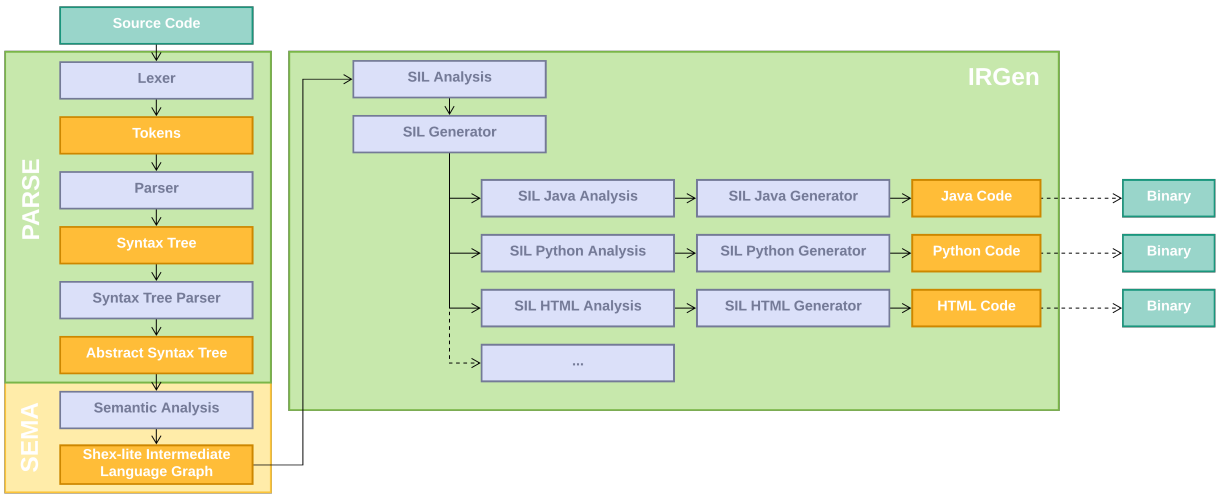


Figure 5.4: Low level view of the compiler flow, grouped by compiler-action.

System Implementation 6

6.1 ShEx-Lite Language Implementation

6.2 Compiler Implementation

Plannings and Budget 7

Conclusions 8

Annexes

A

ShEx-Lite Lexical Specification

B

ShEx-Lite Syntax Specification

C

Publications

References

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, 8, 10, 11).

[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 pages 2, 11).

[7] *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 pages 4, 5).

[8] *Shape Expressions Language 2.1*. URL: https://shex.io/shex-semantics/#shexc (visited on 05/23/2020) (cited on page 5).

[9] Apple Inc. *Swift.org*. en. Library Catalog: swift.org. URL: https://swift.org (visited on 05/24/2020) (cited on page 5).

[10] *What is rustc? - The rustc book*. URL: https://doc.rust-lang.org/rustc/index.html (visited on 05/24/2020) (cited on page 5).

[11] *dotnet/roslyn*. en. Library Catalog: github.com. URL: https://github.com/dotnet/roslyn (visited on 05/24/2020) (cited on pages 5, 14).

[12] *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 5).

[13] *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 5).

[14] 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 pages 7, 8).

[15] *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 pages 7, 12).

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

[17] Tim Berners-Lee et al. *Semantic web road map*. 1998 (cited on page 8).

[18] Eric Van der Vlist. *Relax ng: A simpler schema language for xml*. " O'Reilly Media, Inc.", 2003 (cited on page 8).

[19] *Domain-specific language*. en. Page Version ID: 945660040. Mar. 2020. URL: https://en.wikipedia.org/w/index.php?title=Domain-specific_language&oldid=945660040 (visited on 05/26/2020) (cited on page 12).

[20] *Turing completeness*. en. Page Version ID: 954916159. May 2020. URL: https://en.wikipedia.org/w/index.php?title=Turing_completeness&oldid=954916159 (visited on 05/26/2020) (cited on page 12).

[21] *Lexical analysis*. en. Page Version ID: 955735363. May 2020. URL: https://en.wikipedia.org/w/index.php?title=Lexical_analysis&oldid=955735363 (visited on 05/26/2020) (cited on page 13).

[22] *Optimizing compiler*. en. Page Version ID: 949660274. Apr. 2020. URL: https://en.wikipedia.org/w/index.php?title=Optimizing_compiler&oldid=949660274 (visited on 05/26/2020) (cited on page 14).

[23] *Entorno de desarrollo integrado*. es. Page Version ID: 126109294. May 2020. URL: https://es.wikipedia.org/w/index.php?title=Entorno_de_desarrollo_integrado&oldid=126109294 (visited on 05/26/2020) (cited on page 14).

[24] Jose Emilio Labra-Gayo et al. 'Challenges in RDF validation'. In: *Current Trends in Semantic Web Technologies: Theory and Practice*. Springer, 2019, pp. 121–151 (cited on page 15).

[25] Iovka Boneva et al. 'Semi Automatic Construction of ShEx and SHACL Schemas'. In: (2019) (cited on pages 15, 17).

[26] Daniel Fernández-Alvarez, Jose Emilio Labra-Gayo, and Herminio Garcıa-González. *Inference and serialization of latent graph schemata using shex*. 2016 (cited on page 17).

[27] Herminio Garcia-Gonzalez, Daniel Fernandez-Alvarez, and Jose Emilio. 'ShExML: An heterogeneous data mapping language based on ShEx'. In: () (cited on page 18).

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.