DEGREE PROJECT, IN COMPUTER SCIENCE , SECOND LEVEL
STOCKHOLM, SWEDEN 2015

A Domain-Specific Language for Normalization of Financial Derivatives Data

LUDVIG JONSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION (CSC)

KTH Computer Science and Communication

A Domain-Specific Language for Normalization of Financial Derivatives Data

Ett domänspecifikt språk för normalisering av finansiella derivat-data

LUDVIG JONSSON
ludjon@kth.se

Master's Thesis in Computer Science
School of Computer Science and Communication (CSC)
Royal Institute of Technology, Stockholm
Supervisor: Olov Engwall
Examiner: Olle Bälter
Project commissioned by: Tobias Widén at TriOptima AB

Abstract

A Domain-Specific Language (DSL) is a language tailored for a specific problem domain with the purpose of improving developer productivity and communication with domain experts. In this thesis we investigate how a DSL for normalization of financial derivatives data can be designed and implemented.

The thesis includes research on the general subject of DSL engineering and previously established approaches and guidelines for DSL development.

We describe a development process that consists of three phases: domain analysis, language design and implementation. The proposed solution was evaluated according to a set of predefined quality criteria.

The report concludes with a discussion about data normalization as a DSL domain as well as what impact decisions made during the development process had on the proposed solution.

The thesis fulfills its purpose of being an exploratory study of DSL development and the conclusions listed in the final chapter should apply to all data normalization DSLs.

Referat

Ett domänspecifikt språk (eng. Domain-Specific Language, DSL) är ett språk som är skräddarsytt för ett specifikt problemområde med syftet att förbättra utvecklares produktivitet och kommunikation med domänexperter. I den här uppsatsen undersöker vi hur ett domänspecifikt språk för normalisering av data som beskriver finansiella derivataffärer kan utformas och implementeras.

Uppsatsen omfattar utforskning av det generella ämnet domänspecifika språk och tidigare etablerade tillvägagångssätt och riktlinjer för utveckling av sådana språk.

Vi beskriver en utvecklingsproess som består av tre faser: domänanalys, språkutformning och implementation. Den föreslagna lösningen utvärderades enligt en mängd fördefinierade kvalitetskriterier.

Rapporten avslutas med en diskussion om datanormalisering som domän för ett domänspecifikt språk och en analys av vilken inverkan beslut som togs under utvecklingsprocessen hade på det slutgiltiga resultatet.

Arbetet uppfyller sitt syfte att vara en utforskande studie i DSL-utveckling och slutsatserna som listas i det avslutande kapitlet bör gälla alla domänspecifika språk för normalisering av data.

Preface

This degree project marks the end of my studies at the Master's programme in Computer Science at the Royal Institute of Technology (KTH). I would like to thank every one who has helped and supported me throughout this project. In particular, I would like to thank my supervisors, Tobias Widén at TriOptima and Olov Engwall at the School of Computer Science and Communication (CSC), for their invaluable support and guidance.

Stockholm
2015-02-18
Ludvig Jonsson

Contents

1 Introduction **1**
1.1 Background . . . 1
1.1.1 TriOptima and triCalculate . . . 1
1.1.2 Financial Derivatives . . . 1
1.2 Problem Statement . . . 2
1.2.1 Purpose . . . 2
1.2.2 Intended Audience . . . 3
1.2.3 Delimitation . . . 4

2 Theory **5**
2.1 Domain-Specific Languages . . . 5
2.1.1 Definition . . . 5
2.1.2 Why Use a DSL? . . . 5
2.1.3 The Phases of the DSL Development Process . . . 6
2.2 Internal and External DSLs . . . 6
2.2.1 An example . . . 6
2.2.2 Advantages and Disadvantages . . . 7
2.3 Domain Analysis . . . 8
2.4 Designing a DSL . . . 9
2.4.1 Approaches . . . 9
2.4.2 Guidelines . . . 9
2.5 Implementing a DSL . . . 10
2.5.1 Semantic Modelling . . . 10
2.5.2 Parsing . . . 11
2.5.3 Interpretation . . . 12
2.5.4 Code Generation . . . 13
2.6 Evaluating a DSL . . . 14
2.7 Related Work . . . 14
2.8 Synthesis . . . 15

3 Method **17**
3.1 Domain Analysis . . . 17
3.2 Language Design . . . 17

3.3 Implementation 17
3.4 Evaluation 18

4 Domain Analysis **19**
4.1 Sources 19
4.2 Domain Description 19
4.2.1 Client Submitted Data in triCalculate 19
4.2.2 Data Normalization 20
4.2.3 Data Verification 21
4.3 Requirements 21

5 Language Design **24**
5.1 Design Process 24
5.2 Language Elements 25

6 Implementation **28**
6.1 Programming Environment 28
6.1.1 Python 28
6.1.2 ANTLR 28
6.2 Implementation Process 29
6.2.1 Formalizing the Grammar 31
6.2.2 Configuring the Generated Parser 32
6.2.3 Implementing the Semantic Model 32
6.2.4 Error Handling 33

7 Results **35**
7.1 Example Use of the Proposed Solution 35
7.2 Findings of Focus Group Interview 1 35
7.2.1 Comprehensibility 35
7.2.2 Expected Elements 36
7.2.3 Comparison with Previous Solutions 36
7.2.4 Proposed Improvements 36
7.3 Findings of Focus Group Interview 2 37
7.3.1 Functional Suitability 37
7.3.2 Defining the Problem Domain 37
7.3.3 Proposed Improvements 38
7.4 Summary of the Analysis of the Remaining Quality Characteristics . 38
7.4.1 Expressivity 38
7.4.2 Extensibility 39

8 Discussion **40**
8.1 The Proposed Solution 40
8.2 The Development Process 41
8.3 Data Normalization as a DSL Domain 42

9 Conclusions **43**
9.1 Summary 43
9.2 Future Work 44

Bibliography **45**

Appendices **46**

A Sample Data **47**
A.1 Input data sample 47
A.2 Normalized data sample 48

B Focus Group Interview Questions **49**
B.1 Questions from the first focus group interview 49
B.2 Questions from the second focus group interview 49

C Sample DSL Specification **50**

Chapter 1

Introduction

The aim of this degree project was to explore how a domain-specific language (DSL) for normalization of financial derivatives data could be designed and implemented. A larger goal of the project was to contribute to a more scientifically established methodology for DSL development by providing an experience report from the development of a DSL for an industrial problem domain.

1.1 Background

1.1.1 TriOptima and triCalculate

TriOptima is a provider of post trade infrastructure and risk management services for the OTC (Over-the-counter) derivatives market.

During 2015 TriOptima aims to release a new service called triCalculate. The purpose of this service is to provide portfolio risk analysis powered by numerical solution techniques.

triCalculate is a Software as a Service (SaaS) where clients (which are mainly banks) upload their portfolios and market data. The data is then being normalized to the format expected by the rest of the system. Based on the normalized data TriOptima performs a risk analysis of the trade portfolio and the results from the analysis are then fed back to the clients via a web interface.

1.1.2 Financial Derivatives

A financial derivative is a financial instrument. A financial instrument is a contract that gives financial rights and/or responsibilities to the parties involved. Durbin describes financial derivatives as price guarantees [1], since a derivative agreement specifies a price at which some asset can or must be bought at (or before) a specified date. It is called a derivative because its value is derived from the expected future price movements of the underlying asset to be bought or sold.

An example of this could be an agreement where the buyer is granted the right (but not the obligation) to make an exchange between two given currencies at a

specified exchange rate at a specified future date. This is called a foreign-exchange option and is just one of many instrument types that can be categorized as financial derivatives.

The foreign-exchange option is simple in the sense that it consists of only one payment. Most financial derivatives trades consists of legs of cash flows (which are streams of payments) going back and forth between the counter-parties.

Financial derivatives can be used for either risk management (e.g to protect yourself from future movements of the exchange rate between two currencies) or speculation (to make an agreement with the expectation of a favorable price movement of the asset to be bought or sold).

The data handled in this thesis describes derivative trades and consists of a number of different properties depending on the instrument type.

1.2 Problem Statement

The format of the client submitted data in triCalculate may differ slightly depending on the client's IT infrastructure and needs. This means that depending on the client the data has to be normalized in a number of different ways and thus the component handling the normalization needs to be customized for each client. In this thesis, data normalization refers to the process of having data in a number of different input formats and transforming the data into the same output format.

One way to facilitate the development and maintenance of the client adaptations would be to design and implement a domain-specific language (DSL) for the problem. A DSL is a high-level programming language that is specifically designed to allow the developer to describe solutions in the terminology and at the level of abstraction of a particular problem domain.

The purpose of this DSL would be to allow users to describe different input formats in specifications that are used to configure a file parser that normalizes the input data to the format that triCalculate expects.

1.2.1 Purpose

The purpose of this thesis was to explore how a DSL for normalization of financial derivatives data could be designed and implemented. The proposed solution was analyzed according to a set of quality criteria and it was discussed what impact decisions made during the development process had on the final result.

We defined the following criteria which were reflected upon in the evaluation of the proposed solution:

- Comprehensibility: Is the DSL understandable for domain experts?
- Functional suitability: Is the DSL an appropriate abstraction of the problem domain? Do the language abstractions allow the users to express solutions for a sufficient share of the possible instances of the problem?

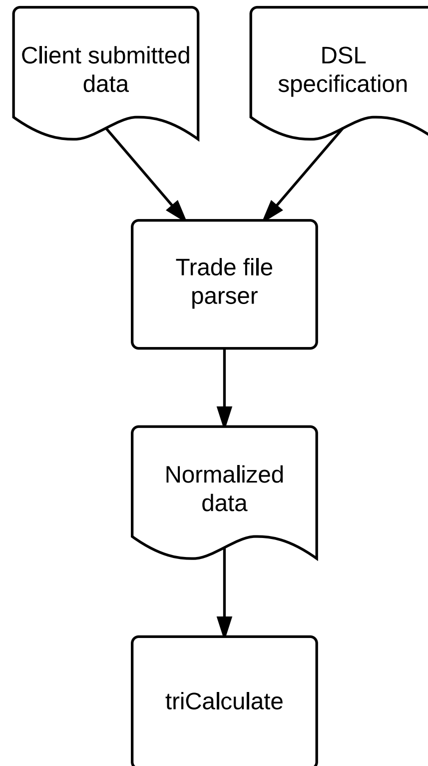

Figure 1.1: Visualization of the purpose of the DSL

- Extensibility: Is it complicated to extend the DSL with further abstractions? Does the implementation allow for users to add new features to the DSL?
- Expressivity: Do the language abstractions enable the user to express concise solutions for the problem?

1.2.2 Intended Audience

The central topic of this thesis is DSL engineering and the results from this work should be interesting to anyone interested in that subject and especially the ones looking into designing and implementing a DSL for the purpose of data normalization.

The readers do not need to have any previous experience or knowledge of DSL engineering, since chapter 2 provides a thorough description and explanation of the basic concepts of the subject.

The readers are expected to have a basic understanding of programming languages and system development in general.

1.2.3 Delimitation

One part of the delimitation was that the prototype of the DSL would only cover a subset of the functionality of the existing solution in triCalculate. This subset was selected in consultation with TriOptima. The point of the prototype language was that it would be sufficient to fulfill the purpose of this thesis. It was not meant to be a complete solution that could replace the current solution in triCalculate.

Another part of the delimitation was that the evaluation would only consider quality characteristics that are possible to draw fair conclusions about without performing a full-scale user study or deploying and using the proposed solution over a longer period of time.

Chapter 2

Theory

The purpose of this chapter is to give the reader an introduction to the general subject of DSL engineering, the different phases of the DSL development process and a summary of related work.

2.1 Domain-Specific Languages

2.1.1 Definition

A domain-specific language (DSL) is a programming language tailored to a particular problem domain [2, 3, 4]. Compared to general-purpose programming languages, DSLs offer substantial gains in expressiveness by allowing users to describe solutions in the terminology and at the level of abstraction of the respective problem domain (as exemplified in subsection 2.2.1) [2, 4].

Examples of commonly used DSLs are HTML (a markup language used to create web pages [5]) and Make (a language for building executable programs [6]).

2.1.2 Why Use a DSL?

DSLs are popular for two main reasons: they improve developer productivity and they improve communication with domain experts [3].

The improvement in productivity is due to the DSL providing an expressive and understandable way of manipulating the underlying model that is the abstraction of the problem domain. The expressiveness of the DSL is limited to its problem domain and thus the possibilities of manipulating the model in erroneous ways and causing unexpected behaviour is also limited [3].

The improvement in the communication with domain experts is due to the solutions being expressed in a terminology and at a level of abstraction that the domain experts are familiar with.

2.1.3 The Phases of the DSL Development Process

Mernik et al. [4] divides the DSL development process into five phases: decision, analysis, design, implementation and deployment.

In the decision phase it is decided whether or not the eventual benefits of the DSL will be worth the time that needs to be invested in the development of it.

A decision to start the development of a new DSL is followed by an analysis phase. In this phase, the domain of the DSL is defined and some kind of domain description is produced. The analysis phase is described more thoroughly in section 2.3.

The goal of the design phase is to capture the important concepts of the domain in a set of language constructs. These language constructs are gathered in a design description which describes the syntax and semantics of the DSL. Common DSL design approaches and guidelines are described in section 2.4.

In the implementation phase the language constructs are implemented. The main approaches for implementing DSLs are described in section 2.5.

In the deployment phase, the DSL is deployed and used. Visser [7] adds maintenance as the last phase, where the DSL is altered and evolved.

Mernik et al. [4] state that even though this appears to be a sequential process, that is often not the case in practice [4]. It is explained that the reason for this is that the phases often influence each other. For example, the design of the language is often influenced by implementation considerations.

2.2 Internal and External DSLs

2.2.1 An example

DSLs are generally divided into two categories: internal DSLs and external DSLs [3]. The difference between the two is best explained with an example. The example will show three ways in which a simple graph model can be populated: using a general-purpose language, using an external DSL and using an internal DSL.

The following code shows how the graph model can be populated using the general-purpose language Java:

```
1 Graph graph = new Graph();
2 graph.addEdge("A", "B", 10);
3 graph.addEdge("B", "C", 20);
4 graph.addEdge("D", "E", 30);
```

An external DSL has a custom syntax and the language itself is separated from the language of the application it is intended to work with [3]. Programs written in the external DSL are parsed into data that the host language can understand. The following code shows how the graph model can be populated using an external DSL:

```
1 Graph {
2   A -> B (10)
3   B -> C (20)
4   D -> E (30)
5 }
```

An internal DSL is just a particular way of using a general-purpose language [3]. It has the feel of being a custom language, but its syntax is valid syntax in the host language and can thereby be executed as any other code written in that language. Compared to traditional APIs (sometimes referred to as command-query APIs) consisting of a set of methods that can be understood individually, an external DSL is a more fluent interface where the individual method names only make sense when the method calls are chained together into expressions reminiscent of natural language sentences [8].

The following code shows how the graph model can be populated using an internal DSL implemented in Java using method chaining:

```
1 graph()
2   .edge()
3     .from("A").to("B").weight(10)
4   .edge()
5     .from("B").to("C").weight(20)
6   .edge()
7     .from("D").to("E").weight(30);
```

The main difference between internal and external DSLs lies in the parsing of the languages [3]. External DSLs are parsed by external parsers and internal DSLs are parsed when executing the DSLs within their respective host language environment.

2.2.2 Advantages and Disadvantages

Internal and external DSLs both have advantages and disadvantages that should be taken into consideration when choosing one of the two techniques. The degree to which these advantages and disadvantages are present depends on the problem domain and the target audience [3].

Internal DSLs

Learning Curve For a developer that does not have any prior experience of DSL development or language development in general, the development of an internal DSL may be more approachable since it does not involve having to learn about things like parsers and grammars [3]. There is no need to implement a parser since the parsing is done by the execution environment of the host language.

There is however a need to implement an expression builder [3] that allows the user to build the fluent expressions of the internal DSL. This often involves the use of rather obscure language techniques like method chaining and other unconventional uses of the host language [3]. This means that the development of an internal DSL can be tricky even for someone who has a lot of experience of the host language.

Tools Since the internal DSL code is valid code in the host language, tools like integrated development environments (IDEs) (with features like code completion and syntax highlighting) available for the host language can be used for the internal DSL as well. It is also possible to mix the DSL code with regular code written in the host language. This can be a big advantage in some situations, but it can also be confusing to users that are not familiar with the host language [3].

External DSLs

Syntactic Flexibility The main strength of external DSLs lies in the syntactic flexibility [3]. When developing external DSLs, the developer has full control over the language syntax. Internal DSLs will always have at least some amount of syntactic noise (like semicolons) and constraints regarding how things can be expressed. The degree of the syntactic noise and constraints will of course differ between languages; some languages are better suited for being internal DSL host languages. This will not matter if the people in the target audience are familiar with the host language, but if they are not it can be a factor that hinders the process of learning how to use the DSL [3].

The Need for a Parser Developing an external DSL either means having to implement a parser for the language or generating one using a parser generator [3]. If a parser generator is used, the grammar of the language must be defined using a formal notation technique for grammars such as the commonly used Backus-Naur Form (BNF).

There is a lot to learn just to get started with external DSL development, in contrast to internal DSL development where the developer can start off simple and learn new techniques during the development of the DSL [3].

2.3 Domain Analysis

When the decision has been made to develop a new DSL, the first step is to gather domain knowledge and identify the problem domain [4]. During the domain analysis, the basic properties and requirements of the domain are analyzed [7].

The domain analysis is often done in an informal manner by consulting domain experts and analyzing sources such as technical documentation and eventual existing general-purpose language code [4]. The output from an informal domain analysis consists of a compilation of domain terminology and semantics.

The domain analysis can also be done using a formal domain analysis method such as Feature-Oriented Domain Analysis (FODA).

The difference between the formal and informal approaches is that the output from the formal analysis consists of a more substantial and specific domain model [4].

2.4 Designing a DSL

2.4.1 Approaches

Mernik et al. [4] state that DSL design approaches are categorized by their relation to existing languages and the formal nature of the design description.

Relation to Existing Languages

A DSL can be designed either by language exploitation (basing the DSL on an existing language) or by language invention (developing a new DSL from scratch) [4]. Basing the DSL on an existing language may lead to an easier implementation and result in a DSL familiar to the users [4, 9] if the users have previous experience of the existing language [4]. For obvious reasons, developing a new DSL from scratch offers a greater flexibility when designing the language syntax.

Formal Nature of the Design Description

An informal design description is expressed in natural language, often accompanied by a set of example programs written in the DSL [4]. A formal design description consists of a specification of the syntax and semantics written in formal notations [4] like BNF grammars for the syntax and Abstract State Machines for the semantics. As with the domain analysis, the informal approach is likely to be sufficient in most cases [4] but the formal approach can help bring forward problems with the design before the actual implementation of the DSL [4].

2.4.2 Guidelines

According to Fowler [3] the overall goal for a DSL is clarity for the reader. The readers of the DSL (programmers and domain experts) should be able to understand code written in the DSL as quickly as possible. Designing a new domain-specific language is a complex, error-prone and often time consuming task [3, 4, 9]. Previous studies have established a set of design guidelines to aid this task.

Iterative Design Process

Fowler [3] and Visser [7] recommend an iterative design process and to try out ideas on the target audience along the way. Finding the right path to a good language design involves providing multiple alternatives for each design decision and choose

what seems to be the most appropriate alternative in consultation with the target audience [9, 3].

Sticking to Conventions

Even if the DSL is not directly based on an existing language, it is still a good idea to stick to common language conventions and reuse previous language definitions when designing the new DSL [9, 3]. This will allow some of the users to identify familiar notations in the DSL [9] and quickly get an understanding of the language.

Simplicity

Simplicity is one of the main targets when designing languages in general, since it enhances the comprehensibility of the language [9]. If a language is simple, it is easier to understand. When a language is easy to understand, it lowers the barrier of introducing it in a working environment [9]. Unnecessary complexity also counteracts the purposed benefits of a DSL [9, 3].

A key part of achieving this simplicity is to avoid unnecessary generality [9, 3, 4]. For example, generalizing the design for future extension is generally a bad idea since it will make the current design more complex than it needs to be [9].

The number of language elements should be kept to a minimum, since a language with a large number of elements will be more difficult to understand [9].

Consistency

As stated in the definition of a DSL, it is a language that offers expressive power in a specific problem domain. Each feature of the DSL should either contribute to this purpose or be omitted [9].

When designing the language elements, the same style should be used everywhere to allow the users to obtain an intuitive understanding of the DSL [9].

2.5 Implementing a DSL

2.5.1 Semantic Modelling

Programming languages in general are often discussed in terms of syntax and semantics [3]. The syntax is the valid expressions of a language and the semantics are what the expressions mean and what happens when they are executed. When designing a DSL, the semantics are often implemented in an underlying semantic model (which can be an object model or simply a pure data structure with the behavior in separate functions) and the DSL itself is just a readable way of populating that model [3]. Programs written in the DSL will then just be descriptions of different configurations of the semantic model.

Although a separate semantic model is not necessary (the semantics could for example be defined directly in a parser instead), Fowler states that a separate

semantic model is a vital part of any well-designed DSL [3]. The reason for this is that a semantic model provides a clear separation of concerns between the parsing of a language and the implementation of its semantics. This separation enhances the testability of the DSL, since it allows the semantic model and the DSL to be tested independently [3].

It also allows the semantic model and the DSL to be evolved independently, adding features to the semantic model before figuring out how to represent them in the DSL [3].

A semantic model is not bound to a specific DSL, it is possible to build multiple DSLs on top of the same semantic model. For example, it is possible to have both an internal and an external DSL for populating the same semantic model.

2.5.2 Parsing

As mentioned earlier, the parsing of internal DSLs is done by the host language during the execution. Since an external DSL uses a custom syntax and the DSL itself is separated from the language of the application it is intended to work with, a parser is a natural part of any external DSL.

The parsing of a DSL can be divided into two steps [3, 8]: lexical analysis and generation of a parse tree. The DSL code is first fed into a lexical analyzer that tokenizes the input stream and categorizes each token as one of a set of predefined language elements. Based on a set of grammar rules (which defines the valid combinations of the language elements), the parser processes the tokens and generates a parse tree [8].

The following is an example of a grammar expressed in Extended Backus-Naur Form (EBNF) that could be applied to the previous external DSL example:

```
1 GraphBlock ::= 'Graph {' Edge+ '}'
2 Edge ::= Vertice '->' Vertice '(' Weight ')'
3 Vertice ::= [a-zA-Z]+
4 Weight ::= [0-9]+
```

The parser tree is often reduced to an Abstract Syntax Tree (AST) [3]. The AST is a simplification of the parse tree where all unnecessary nodes are omitted, see figure 2.2 and 2.3. It is abstract in the sense that not all elements of the language syntax are represented in it. For example, grouping parentheses and brackets are implicit in the tree structure so their presence in it serves no purpose.

A parse tree is a convenient way of grouping language elements related to each other. This grouping makes it easier to collect the information needed to populate the semantic model.

When developing a parser manually, the grammar rules are implicitly defined by the code base. This may not be a problem for trivial parsers, but for nontrivial parsers this means that any changes to the grammar require significant changes in

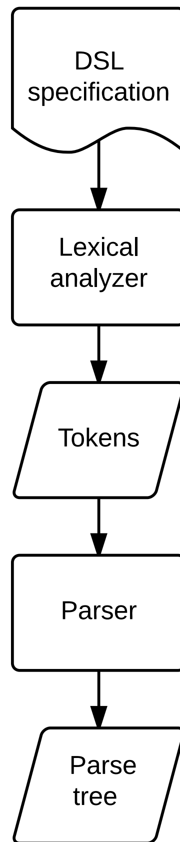

Figure 2.1: The DSL parsing process

the parser code [8]. Developing a parser manually also involves writing the code that walks the syntax tree and processes the information in it.

As mentioned earlier, an alternative to writing a parser manually is to use a parser generator that generates a parser based on a set of given grammar rules and actions to execute on recognition of these grammar rules. An example of such a parser generator is ANTLR [10] which is a Java-based parser generator that is able to generate parsers in a number of different target languages.

2.5.3 Interpretation

Just like any other programming language, a DSL can either be interpreted or compiled [3]. When a DSL is interpreted, the DSL script is processed to populate a semantic model and then the semantic model itself is executed to provide the purposed behavior. The parsing and the execution is done in a single process, like with any other interpreted programming language. The advantages of the interpretation approach compared to the compilation approach are greater simplicity, greater

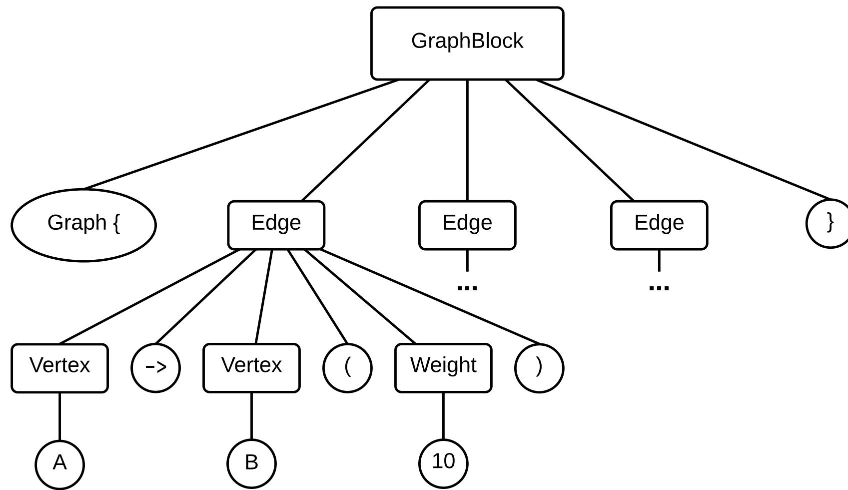

Figure 2.2: Parse tree for the external DSL example in subsection 2.2.1

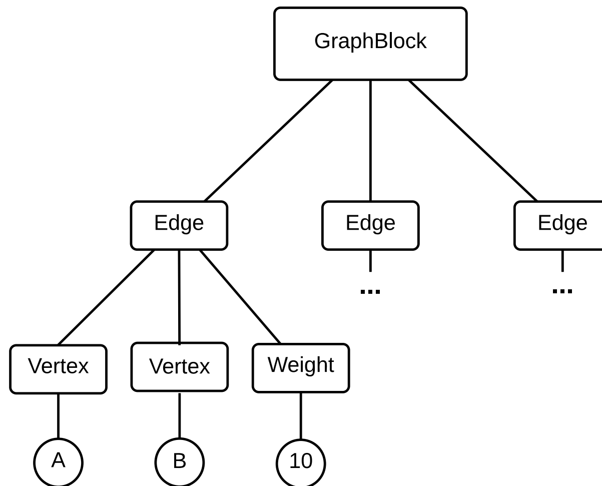

Figure 2.3: Abstract Syntax Tree (AST) for the external DSL example in subsection 2.2.1

control over the execution environment and easier extension [4].

2.5.4 Code Generation

The compilation approach is often referred to as code generation [3]. Code generation means that the populated semantic model is translated to base language constructs and library calls [4]. This code is then separately compiled and run. This approach can be useful when the target platform on which the semantic model

is supposed to run has a limited set of language choices [3].

2.6 Evaluating a DSL

A number of studies have been done on the subject of DSL evaluation. Kahraman et al. [11] gathers the quality characteristics established in these studies and propose a framework called FQAD (A Framework for Qualitative Assessment of DSLs). FQAD consists of 10 quality characteristics and 26 sub-characteristics. An example of a quality characteristic is functional suitability, which has the sub-characteristics completeness and appropriateness.

A DSL can be qualitatively assessed by comparing the characteristics of the DSL to the established quality characteristics. The first step in FQAD is to select what quality characteristics are relevant for the goal of the DSL. FQAD then provides a questionnaire where the evaluator can give feedback on the support levels of the sub-characteristics. The meanings of the different quality characteristics are defined in the same questionnaire.

The assessment result is based on the support levels of the sub-characteristics of the characteristics that were selected as important.

Comprehensibility, functional suitability, extensibility and expressivity were the characteristics that were considered to be within the delimitation described in subsection 1.2.3. FQAD describes these characteristics as follows [12]:

- Comprehensibility: *The concepts and symbols of the language are learnable and rememberable (e.g., ease of learning DSL language elements, ease of learning to develop a program, effective DSL documentation)*

- Functional suitability: *Functional suitability of a DSL refers to the degree to which a DSL supports developing solutions to meet stated needs of the application domain.*

- Extensibility: *DSL has general mechanisms for users to add new features (adding new features without changing the original language)*

- Expressivity: *The degree to which a problem solving strategy can be mapped into a program naturally.*

As mentioned, FQAD provides a set of sub-characteristics to facilitate the evaluation of these characteristics. This does not apply to comprehensibility, since it itself is a sub-characteristic of usability.

2.7 Related Work

When exploring previous work in the area of DSL engineering, it was obvious that the most central work in the area deals with general guidelines and approaches

for the design and implementation of DSLs. Voelter [13] states that since DSLs by definition are domain-specific, published case studies on the subject of DSL engineering might not be very helpful when deciding whether or not (and how) to develop a new DSL for a given problem domain. This may explain the low number of such case studies and other studies describing the development of individual DSLs.

During the exploration of previous work, the work that was found to be the most closely related to this thesis was a case study by Hautus [14] that describes the development of a DSL for normalization of financial data sets. Hautus states that using a custom syntax rather than basing the DSL on an existing language (in this case XML) is worth the effort and that a DSL needs to be periodically refined as more domain knowledge is gathered.

Most examples of previously developed DSLs for the purpose of transforming data are visual DSLs allowing the user to interactively specify different kinds of data transformations in a graphical user interface (GUI). An example of this is Wrangler [15], a DSL for interactive visual specification of data transformation scripts introduced by Kandle et al.

Visser [7] describes the development of WebDSL, a DSL for building web applications. This study is not related to data transformation, but it is a case study on the subject of DSL engineering with the purpose of investigating when and how to develop a DSL. Visser concludes that you must obtain a good understanding of the application domain and that there should exist a considerable code base for systems in the domain before starting the development of a DSL. These conclusions concern when to develop a DSL, but Visser also provides a number of guidelines on how to develop a DSL. Visser also provides a list of DSL evaluation criteria which largely corresponds the list provided by Kahraman et al.

2.8 Synthesis

This section contains a compilation of the main lessons learned from the theory review and a description of their impact on the continued work with this thesis.

Based on the comparison of internal and external DSLs, it was decided that the DSL designed and implemented in this thesis would be an external one. The main reason for this was the greater syntactic flexibility offered by external DSLs, something that was considered to be a great advantage when designing a DSL to be as concise and expressive as possible.

Another learning from the theory review was that the DSL development process should always start with a thorough domain analysis, since it is hard to design a language for a domain you do not understand. It was decided that an informal analysis method would suffice for this project, mostly because the domain in question is small. If a formal domain analysis method would be used, more time would probably be spent on formalities than on the actual analysis.

The design guidelines listed in subsection 2.4.2 does contain some concrete advice on how to achieve a concise and expressive language design. It was decided to let

this list guide the language design process. Even though one of the guidelines was to involve the intended users of the DSL in an iterative design process, it was decided that there was not enough time and resources allocated for this degree project to be able to do that.

To avoid spending a too big part of the time allocated for this project on implementing a parser from scratch, it was decided to use a parser generator. It was also decided that the implementation would follow the advice of separating the parsing of the DSL from its semantic model, since this kind of modular system design gives a separation of concerns that enhances the testability and allows for the system components to be evolved independently.

Since it was decided to implement the external DSL using a parser generator, it was known that the grammar of the language would eventually have to be expressed using a formal notation such as BNF.

Regarding the evaluation of the DSL, it was concluded that doing a full assessment using FQAD (described in section 2.6) was outside the scope of this thesis. It was also concluded that that FQAD could still be a useful as a basis for the evaluation, since it contains a a comprehensive description of DSL quality characteristics.

Chapter 3

Method

This chapter describes the phases of the method used for this thesis. These are presented in a chronological order. These phases were largely inspired by the description of the DSL development process provided by Mernik et al. [4] (described in subsection 2.1.3). The first three phases (domain analysis, language design and implementation) are described more thoroughly in chapter 4, chapter 5 and chapter 6.

3.1 Domain Analysis

During the domain analysis phase the problem domain was defined. This was done by consulting domain experts and analyzing the current solution at TriOptima and resulted in a compilation of domain terminology and domain concepts. This compilation was used to form a specification of the functionality that the language would have to cover.

3.2 Language Design

The goal of the design phase was to design a language to be as concise as possible without losing the technical flexibility required by the problem domain. This resulted in a description of the syntax (the valid expressions of the language) and the semantics (what the expressions mean and what happens when they are executed). The design description consisted of a set of example specifications expressed in the DSL.

3.3 Implementation

The implementation phase involved formalizing the design description produced in the language design process, generating a parser for it using an appropriate parser generator, configuring that parser to fit our purposes and implementing the semantic model to be configured by the DSL.

This was an iterative process starting off with the most basic functionality and then continually adding and improving functionality until the previously mentioned specification was met.

When the implementation of the prototype language was completed a number of example solutions covering common use cases were implemented using the DSL.

3.4 Evaluation

The evaluation phase was based on the DSL quality characteristics listed in subsection 1.2.1. These quality characteristics were based on the list of DSL quality characteristics provided by Kahraman et al. [11] and the list of DSL evaluation criteria provided by Visser [7].

The evaluation process was divided into two parts, where the first part consisted of two focus group interviews and the second part was an objective analysis of the proposed solution. The reason for conducting the focus groups interviews was that it would not be possible to objectively evaluate the comprehensibility and the functional suitability of the proposed solution without feedback from non-biased external parties.

In consultation with the supervisor at TriOptima, two groups of interviewees were selected. The first group was selected from the business side of the company, and consisted of a group of five people who all had great experience of different client data normalization solutions from existing projects within the company. These people were considered to be a good representation of the hypothetical users of the proposed DSL. The purpose of this focus group interview was to evaluate the comprehensibility of the proposed solution.

The second group was selected from the technology side of the company and also consisted of five people. This selected group of people were all software engineers with a good understanding of the technical side of the problem domain. The purpose of this focus group interview was to evaluate the functional suitability of the proposed solution.

Both interview sessions began with a presentation of the DSL. The presentation from the second interview session was a bit more substantial with an added technical description of the proposed solution. These presentations were followed by an open discussions guided by two sets of questions, one for each interview session. The questions from the focus group interviews can be found in Appendix B. The interview sessions lasted for one hour each and were video recorded to allow for later review.

The quality characteristics not covered by the interview sessions, expressivity and extensibility, were evaluated separately by comparing the proposed solution to the relevant quality characteristics and sub-characteristics described in FQAD.

Chapter 4

Domain Analysis

In this chapter we describe the domain analysis phase of this project. It starts with a description of the sources used during the analysis, continues with a description of the domain and concludes with a summary of the domain concepts that needed to be captured by the DSL.

4.1 Sources

During the domain analysis, the following sources were used:

- The existing general-purpose language solution (which is a rule-based file parser written in Python)
- Documentation specifying the expected format of the client submitted data
- Sample portfolio data and the respective output when processed by the existing file parser

4.2 Domain Description

4.2.1 Client Submitted Data in triCalculate

For the portfolio risk analysis in triCalculate to be possible to perform, the clients must submit their trade data to the service. This data is submitted as CSV (comma-separated values) files which follow a given specification of the expected data fields and their respective header names.

Even though the submitted data follow the same specification, the data format may still differ slightly depending on the client (as mentioned in section 1.2). Examples of things that may differ do not only include the format of the CSV file (header names, column order, delimiter character and such) but also the format of the actual values. An example of this is the format of date values, which can follow a number of different standards.

Following the submission of the portfolio data, the data must be normalized and verified in a number of different ways described in the following two sections.

4.2.2 Data Normalization

The normalization process can be divided into two parts: transformation of the data format from CSV to JSON (JavaScript Object Notation) and transformation of the individual field values.

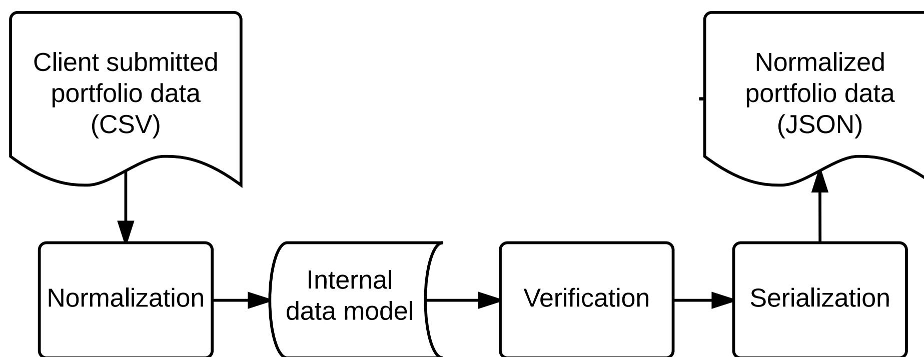

Figure 4.1: The normalization and verification process

The first part applies to all cases, since the portfolio data is submitted as CSV files and the rest of the triCalculate system expects the normalized data to be in a hierarchical JSON format. In the existing solution, this transformation is done by populating an in-memory object model with the trade data from the CSV file. The internal object model is then being serialized into a JSON file. For this to be possible, the fields in the CSV file must somehow be mapped to the corresponding fields in the internal object model.

Since CSV is a flat format, it must somehow be specified how to group the CSV data in the hierarchical structure of the JSON format. As described in subsection 1.1.2, a financial derivative trade often consists of a set of legs that consists of a set of cash flows. This hierarchical structure is represented in the internal trade model. Each row in the CSV file describes a cash flow, but does also contain information about the leg and the trade the cash flow belongs to. This information can be used to create the hierarchical structure of the JSON format. A very simplified example of the transformation from the flat CSV format to the hierarchical JSON format is shown in listings 1 and 2.

The second part of the normalization process applies to all cases where the formats of the field values differ from the given specification. These transformations are done per field, since it can be assumed that all values of a certain field are in the same format.

```
trade_party, trade_counterparty, trade_id, leg_currency, cashflow_
   payment_date, cashflow_nominal
A,B,1,EUR,2015-02-11,100
A,B,1,EUR,2015-02-12,200
A,B,1,USD,2015-02-13,300
A,B,1,USD,2015-02-14,400
```

Listing 1: The flat CSV format

4.2.3 Data Verification

Other than being normalized, the data must also be verified before being allowed into the rest of the triCalculate system. Just like the normalization process, the verification process can be divided into two parts.

The first part is to verify that all required fields are present in the submitted data. What fields are required depends on the instrument type of the trade to be extracted.

The second part is to verify that the data in those fields is valid. For example, for a field that specifies the trade currency it must be verified that the field value is one of a set of predefined currency codes.

4.3 Requirements

As a summary of the domain analysis, this section will describe the domain concepts that need to be captured by the DSL designed and implemented for this thesis.

Input Configuration

The DSL must offer a way of specifying the input configuration for the normalization and verification process. Examples of configuration that is needed for the input is the path to the client submitted trade file, the delimiter of the CSV format and the instrument type of the trades that should be extracted. The input configuration must also allow the user to specify how to group the data from the flat input format in the hierarchical structure of the internal trade model.

Field Mapping

To make the transformation from the flat CSV format to the hierarchical JSON format possible, the DSL must offer a way of mapping the fields of the CSV file to the corresponding fields of an internal object model. Since the CSV files may or may not have a header row containing the column names, it has to be possible to map a field either by its column name or by its column index.

Field Transformations

The DSL must offer a set of field-specific transformations and a way of specifying what transformations should be applied to which fields.

```json
[
 {
   "trade_id": 1,
   "trade_party": "A",
   "trade_counterparty": "B",
   "legs": [
     {
       "currency": "EUR",
       "cashflows": [
         {
           "payment_date": "2015-02-11",
           "nominal": 100
         },
         {
           "payment_date": "2015-02-12",
           "nominal": 200
         }
       ]
     },
     {
       "currency": "USD",
       "cashflows": [
         {
           "payment_date": "2015-02-13",
           "nominal": 300
         },
         {
           "payment_date": "2015-02-14",
           "nominal": 400
         }
       ]
     }
   ]
 }
]
```

Listing 2: The hierarchical JSON format

Chapter 5

Language Design

In this chapter we describe the language design phase, where the domain concepts gathered from the domain analysis (described in chapter 4) were captured in a set of language constructs. It begins with a brief description of the design process and ends with a description of the resulting language design.

5.1 Design Process

As mentioned in chapter 2, DSL design approaches are categorized by their relation to existing languages and the formal nature of the design description.

When designing the DSL for this thesis, the strategy was to stay as close to the Python syntax as possible. The reason for this was mainly because Python is the main language used at TriOptima and partly because Python has a simple syntax that emphasizes readability [16].

The development of the design description was guided by the design guidelines described in chapter 2.

This design description (which consisted of a set of example specifications written in the DSL) was developed in iterations beginning with an abstract sketch of the language elements and then continually making the description more concrete. This was done in consultation with the supervisor at TriOptima.

When designing the individual language elements, the goal was to keep the syntax as simple as possible without losing the flexibility needed to meet the requirements described in section 4.3. The main design guideline learned from studying the relevant theory was to design only what is necessary and to be careful not to over-generalize the language. This was something that guided the whole design process with the hope that it would result in a language as concise as possible.

The last guideline described in the theory chapter was that the language should be consistent in the sense that all language elements should either contribute to the purpose of increasing the expressive power of the language or be omitted. The only language element that may not contribute to this purpose is the comment element, but it was decided to keep this feature anyway because it simplifies communication

and collaboration between developers [9].

5.2 Language Elements

This section describes the syntax and semantics of the set of language elements that were produced during the design process. The set of language elements correspond to the set of requirements gathered from the domain analysis, with the exception of the comment element.

Input Configuration

The input configuration is a set of configuration values describing the input file. The input configuration consists of a description of the CSV format of the input file and also information about the type of trade data it contains, how to filter out the rows that contain the data to be extracted and how the trade data should be grouped in the internal trade model.

```
1  input:
2      delimiter = ","
3      has_header = True
4      instrument_type = IRSWAP
5      row_filter_column = "trade_instrument_type"
6      row_filter_value = "IRSWAP"
7      trade_group_column = "trade_id"
8      leg_group_column = "leg_float_rate_index"
```

Listing 3: The syntax of the input configuration

Field Mapping

The main element of the language is the mapping of the fields of the input file to the corresponding fields of the internal trade model. As described in the previous chapter, this must be possible to do in two different ways: either by column name or by column index (since it can not be assumed that all input files contain a header row).

A third option is to map a field in the trade model to a default value that will be used instead of extracting a value from a column in the input file. An example of a situation where this could be useful is when some column is missing in the input file and it is preferred to set the corresponding field in the trade model to a static value instead of leaving it empty.

```
trade_counterparty -> column_name("counterparty")
trade_instrument_type -> column_index(5)
trade_party -> default_value("Party1")
```

Listing 4: Field mapping by column name, by column index and to a default value

Field Transformations

The last requirement gathered from the domain analysis was that the DSL must offer a set of transformations and a way of specifying which transformations should be applied to which fields. When a value is extracted from the trade file, it is first transformed using the specified transformations (if any) before it is stored in the internal trade model.

Below is an example of a transformation section written in the DSL. It contains three different transformations. The first transformation adds a prefix to the value. The second splits the value by a given delimiter and returns the element at a given index of the resulting list. The third and last one sets the value to "FLOAT" if the extracted value is not empty and sets the value to "FIXED" if it is. This is an example of a rather specific transformation that in some cases needs to be done. It is also an example that shows the importance of easily being able to extend the language with new transformations.

```
transformation:
    trade_id -> add_prefix("IRSWAP-")
    leg_float_rate_period -> split("_", -1)
    cashflow_type -> if_present("FLOAT", "FIXED")
```

Listing 5: Example transformations

Comments

The last element of the language is the comment element. This element was not part of the requirements gathered from the domain analysis, but it was decided later on that it should be added since it improves the documentation of the specifications and possibly also the communication between users of the language. Comments in the DSL work exactly like single-line comments in Python, starting with a number sign and ending with a new line.

```
1 # This is a comment
```

Listing 6: Example comment

Chapter 6

Implementation

In this chapter we describe the implementation process, where the objective was to implement a file parser that could be configured using the DSL described in chapter 5. This chapter consists of descriptions of the programming environment, the implementation process and the resulting implementation.

6.1 Programming Environment

This section describes the main parts of the programming environment used in the implementation process.

6.1.1 Python

The only programming language used for this project was Python, which is an interpreted, object-oriented, high-level programming language that emphasizes readability and provides increased productivity [16].

The main reason for choosing Python was that it (at the time of writing this thesis) was the main language used at TriOptima. This meant that previous file parsing solutions at the company could be used as references when implementing this one. The promised increase in productivity also contributed to this decision, since it seemed possible that it would help to keep up with the schedule of this degree project.

The only drawback of this decision seemed to be that programs written in Python are known to run slower than equivalent programs written in other languages like Java or C++ [17]. This was not considered to be an issue, since the running time of the prototype would not affect the outcome of this thesis.

6.1.2 ANTLR

ANTLR (abbreviation for "Another Tool For Language Recognition") is a parser generator that takes a grammar expressed in EBNF and generates the source code

for a parser for the language specified by that grammar. ANTLR itself is implemented in Java, but has the ability to generate parsers in a number of target languages.

The reasons for choosing ANTLR as the parser generator for the implementation of the DSL were that it is a widely used and recognized tool, that it was easy to set up and get started with and that its functionality was undoubtedly sufficient for this project.

In its initial state, the parser generated by ANTLR does nothing but to read an input file and return an error message if the file does not conform to the syntax specified by the given grammar.

The parser can be made useful by specifying actions to execute on recognition of the different grammar rules. These actions are written in the same language as the parser is generated in and gathered in something that ANTLR calls a listener. The listener is then hooked up to something called a walker (which is also generated by ANTLR). The walker traverses the parse tree and notifies the listener as it passes the different elements of the language.

ANTLR 4, the latest version at the time of writing this thesis, supports generating parser code in Java, C# and Python. The Python support was added in the summer of 2014, just a couple of months before starting this thesis. This meant that there was no official documentation available at the time, so it was impossible to know what the generated Python code would look like before generating it.

6.2 Implementation Process

The objective of this process was to implement a file parser that converts a CSV file to a JSON file given a specification written in the DSL designed for this thesis. Just like the process of designing the DSL, the implementation process began with a rough sketch of the system components which was then improved as the implementation process progressed.

Figure 6.1 gives an overview of the final set of system components and the relationships between them.

The following subsections describe the different parts of the implementation process.

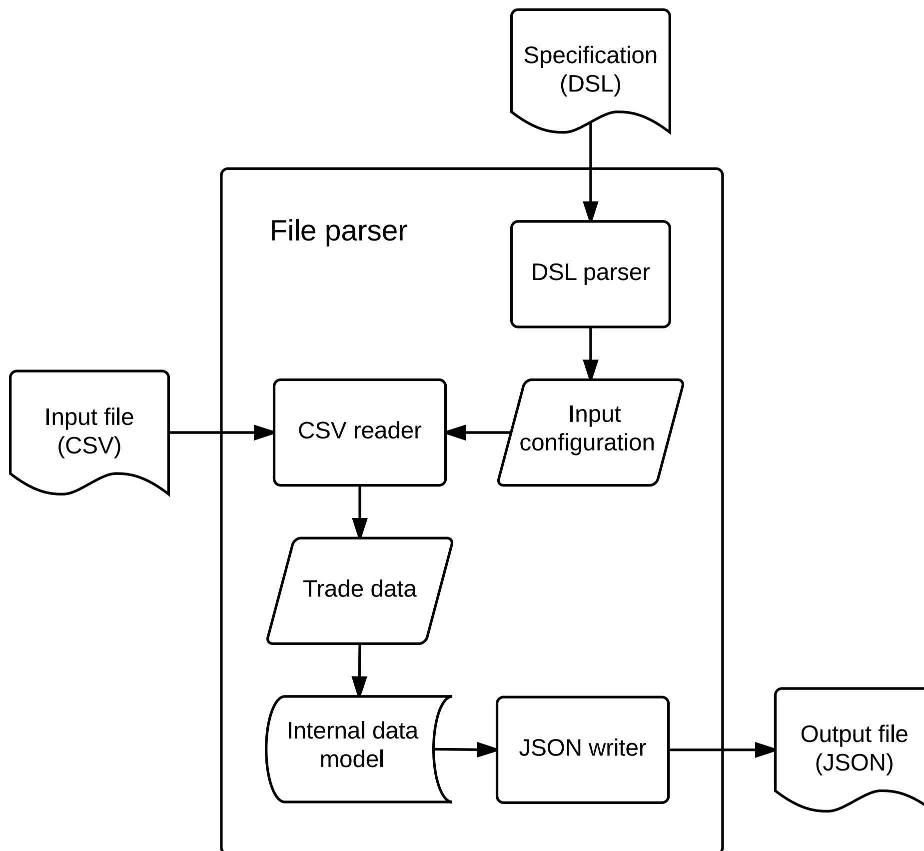

Figure 6.1: System overview

6.2.1 Formalizing the Grammar

ANTLR needs a grammar expressed in EBNF to be able to generate a parser. For the implementation process, this meant that the informal syntax description described in the previous chapter had to be translated into EBNF.

The first version of the EBNF grammar was very specific and did not only cover the structure of the language but also validation of things like field and transformation references. This meant that instead of just specifying the format of the references, the grammar specified a list of valid field and transformation names. As the implementation progressed, these validation parts were moved to the listener actions. The reasoning behind this was that it made more sense to let the grammar specify the general syntax of the language and only return syntax errors when the the input does not conform to this syntax. This resulted in a cleaner and more readable EBNF grammar. Handling the checking of field references and such later in the parsing process also allowed for more specific error messages.

```
grammar DSL;

specification: inputConfig fieldMap transformation EOF;

inputConfig: 'input:' configStatement+;

fieldMap: 'field_map:' mapStatement+;
mapStatement: ID '->' mapType '(' (STRING | NUM) ')';
mapType: ID;

transformation: 'transformation:' transformationStatement+;
transformationStatement: ID '->' transformationType '('
        argument? (',' argument)* ')';
transformationType: ID;

configStatement: ID '=' configValue;
configValue: ID | BOOLEAN | STRING | NUM;
argument: BOOLEAN | STRING | NUM;

NUM: ('-')?[0-9]+;
BOOLEAN: 'True' | 'False';
ID : [a-zA-Z] [a-zA-Z0-9_]*;
STRING:  '"' ~('\n'|'\r'|'"')* '"';
COMMENT: ('#' ~[\r\n]* '\r'? '\n') -> skip;
WS: [ \t\r\n]+ -> skip;
```

Listing 7: The formalized grammar

6.2.2 Configuring the Generated Parser

For the generated parser to be useful, actions to execute on recognition of the different grammar rules must be specified in a listener (as mentioned in the ANTLR section). This part is best explained with an example.
The grammar rule for map statements looks like this:

```
1 mapStatement: ID '->' mapType '(' (STRING | NUM) ')';
```

An example of a line that conforms to this syntax is the following:

```
1 trade_counterparty -> column_name("counterparty")
```

The parser will recognize this line as a map statement and the corresponding node in the parse tree will be tagged as a map statement and contain the field identifier, the map type and the map argument.
When the walker passes this node in the parse tree it will notify the listener which executes the action specified for map statements. Since it was decided to separate the parsing of the DSL from the implementation of its semantics, no actual semantics were implemented in the DSL parser itself. Instead, all semantics were implemented in a set of separate components (which are the CSV reader, the internal data model and the JSON writer in figure 6.1). The DSL is only used to configure these components, which can be seen as the semantic model of this implementation.

6.2.3 Implementing the Semantic Model

As mentioned in the previous section, the semantic model of this implementation consists of three components: a CSV reader, an internal data model and a JSON writer.
A traditional CSV reader would allow the user to extract the values from the CSV file either by column index or by column header name. The CSV reader component in this implementation is an extension of this. When the CSV reader has been configured with field mappings and transformations, values from the input file can be extracted by the name of the corresponding field in the internal data model.
If a transformation has been specified for the field, the value will also be transformed before being stored in the data model. How this was implemented is explained in the following example.
This line expressed in the DSL adds a prefix transformation to the field named "trade_id":

```
1 trade_id -> add_prefix("IRSWAP-")
```

All transformations available in the DSL are implemented as configurable transformation classes. The transformation classes all have the same structure, with a

single method named transform that simply takes a value, transforms it according to the configured transformation and returns it. The CSV reader component keeps a map where the fields are mapped to their respective transformations.

```
class PrefixTransformation(Transformation):

    def __init__(self, prefix):
        self.prefix = prefix

    def transform(self, value):
        if not value:
            return None
        return self.prefix + value
```

Listing 8: A transformation class

This means that when a value for a trade ID is extracted from the trade file, it is run through this transform method which adds the prefix "IRSWAP-" to the value before storing it in the internal trade model.
What the internal data model looks like depends on the derivative type of the trades being extracted, since different types of derivative trades have different structures. However, the way of populating the data model does not differ between the different types.
As mentioned in subsection 1.1.2, derivative trades typically consists of legs of cash flows (which are streams of payments). In the input CSV data, each row describes one of these cash flows. The rows also contain information about the leg that the cash flow belongs to and the trade that the leg belongs to. By specifying in the input configuration which columns distinguish different trades and different legs it is possible to group cash flows by leg and legs by trade in the hierarchical data model. In this case, the model will contain a list of trade objects which contain a set of leg objects. The leg objects will in turn contain a set of cash flow objects.
When the data model is populated it is validated that it contains all required fields (the set of required fields also differs between derivative instrument types) and that the data in those fields is valid. For example, it is validated that a field specifying the currency of a leg contains one of a predefined set of currency codes.
If no errors are found during the validation of the data model, the model is serialized to JSON and written to an output file.

6.2.4 Error Handling

Other than the validation of the internal data model before serializing it to JSON, the error checking of the system consists of a thorough checking of the input format specification.

If the specification does not conform to the grammar of the DSL, the system will exit with a syntax error message specifying the row at which the error occurred.
If the specification contains unknown configuration keys, non-existing field names or non-existing transformation types, the system will exit with an error message describing the element it did not recognize.
The purpose of this error checking is to make sure that the system does not produce invalid or incomplete output data. The system will not produce any output data at all if any errors are found in the input format specification.

Chapter 7

Results

In this chapter the results from the evaluation of the proposed solution are presented. To give the reader a better understanding of the solution, the chapter begins with a description of an example use of it.

7.1 Example Use of the Proposed Solution

A sample input format specification written in the DSL can be found in Appendix C. Using this specification and the data found in section A.1 as input to the file parser would produce the normalized data found in section A.2.

7.2 Findings of Focus Group Interview 1

This section gives a summary of the findings of the first interview session, where the comprehensibility of the proposed solution was the main subject of discussion. The group of interviewees consisted of five people from the business side of TriOptima, who all had great experience of different client data normalization solutions from existing projects within the company.

7.2.1 Comprehensibility

The interviewees agreed that the general concept of the proposed solution is easy to understand, but that some of the language elements were harder to understand than others.
The language elements that were considered to be the easiest to understand were the field mappings and the input configuration statements strictly related to the input format (delimiter character, "has header" and the row filter configuration).
It was explained by the interviewees that the concept of the field transformations was considered to be understandable, but that it was hard to understand what some of the transformations did just by looking at the use of them in the DSL. It was stated that the transformation specifications were very similar to general purpose

language code, and that the use of the transformations would require documentation of the available transformations and the expected parameters.
It was stated that it was clear that the data in the input file needs to be grouped somehow to be able to output the JSON format (as explained in section 4.3), but that it was not obvious how a user could decide which columns to specify as grouping columns in the input configuration.

7.2.2 Expected Elements

When asked about which language elements were expected and which were not, the interviewees answered that all language elements were expected and corresponded to elements of previous solutions at the company. It was stated that the proposed solution is simpler and more understandable than previous solutions, but that it was also considered to be less flexible.
During the interview session, it was mentioned that comments are an important element of the DSL since it allows for users to explain choices made when writing specifications and thus help other users to understand them.

7.2.3 Comparison with Previous Solutions

As mentioned in subsection 7.2.1, the proposed solution was considered to be less flexible than previous solutions for the problem. The DSL was in particular compared to a previous solution where adaptions for different client data formats were specified in plugins written in Python. The interviewees agreed that offering such a high level of flexibility to non-programmers has proven to lead to a system that is difficult to maintain. It was explained that the reason for this is that the flexibility of the general-purpose language Python often was often used in unconventional ways and that the development of a new adaption for a client data format often began with copying and modifying an existing solution that the user does not completely understand.
It was concluded that not letting the users of the DSL write their own general purpose language code is a good thing, and that letting regular system developers extend the set of available transformations of the DSL on request by its users is a good way of solving this.

7.2.4 Proposed Improvements

It was discussed that many of the client-specific specifications written in the DSL would overlap to some degree. It was proposed that the possibility to inherit properties from some kind of default specification may be a way to facilitate the maintenance of these specifications.
It was also stated that the development of new specifications would begin with copying and modifying an existing specification and that it would be less time consuming to allow the users to specify the mappings and transformations in a GUI that generates the DSL specifications.

7.3 Findings of Focus Group Interview 2

This section gives a summary of the second interview session, where the functional suitability of the proposed solution was the main subject of discussion.
The group of interviewees consisted of five people from the technology side of the TriOptima, who were all software engineers with a good understanding of the technical side of the problem domain.

7.3.1 Functional Suitability

The general opinion regarding the proposed solution was that it covers the problem domain as defined by the delimitation of this thesis, but that the problem domain is a greatly simplified version of the actual problem domain. In particular, it was stated that the one-to-one mapping of columns in the input file to fields in the internal data model would not suffice for many real cases since the set of columns in the client submitted data will differ between clients. It was also stated that the proposed solution covers the main concepts of the normalization process, but that the assumptions made about the input formats exclude a large proportion of all plausible formats. The interviewees agreed that there were no obvious obstacles that would make it impossible to extend the proposed solution to reach a full coverage of all plausible input formats.
It was discussed whether or not the output format (which is the structure of the internal data model) should be a part of the DSL specifications. It was stated that including the output format as a part of the specifications (instead of implicitly specifying the output format by specifying the grouping configuration and the type of trades to be extracted) would make the solution more general, but it was also argued that it would create unnecessary complexity for the users.

7.3.2 Defining the Problem Domain

A recurring topic throughout the interview session was the difficulties in defining the problem domain. It was discussed that one of the tricky parts of developing a DSL is that a domain has to be chosen and defined, and that it in this case is difficult to determine which parts are included in the domain. A number of different types of files are being parsed at TriOptima, and it must be decided if the DSL should be a part of a general file parsing solution or if there should be one focused DSL for each of the types of files being parsed. When this has been decided, the problem domain is still difficult to define since it depends on what assumptions can be made about the formats of the data that will be normalized. Since the requirements regarding the formats of the client submitted data in triCalculate are loose, it is difficult to determine what functionality is needed in order to normalize all the client submitted data.

7.3.3 Proposed Improvements

As mentioned in subsection 7.3.1, the main weakness of the proposed solution was considered to be the one-to-one mapping of columns in the input file to fields in the internal data model. It was discussed that a complete solution for the problem would have to allow the user to map and transform values from several columns in the input file into the same field in the internal trade model. A proposed solution for this was to let the mappings and the transformations be done in several steps where the values are stored in intermediate fields before ending up in the final data model. For example, it would be possible to transform values from two different columns in the input file into a single value that is stored in an intermediate field. A second transformation could then be applied to the value in that intermediate field before finally storing the value in the trade model.
The topic of using a GUI to generate the DSL specifications was also brought up during the interview session. Just like in the first interview sessions it was stated that a GUI where the users could specify mappings, and transformations would help the development of new specifications. It was added that the possibility to preview the results of the mappings and the transformations would help this process even further.

7.4 Summary of the Analysis of the Remaining Quality Characteristics

This section gives a summary of the analysis of the quality characteristics not covered by the focus group interviews. Since the analysis was based on the sub-characteristics presented in FQAD, the results are presented as comments regarding the descriptions of these sub-characteristics.

7.4.1 Expressivity

- *A problem solving strategy can be mapped into a program easily.*

  It was deemed that evaluation of this sub-characteristic would require a user study and that it thereby is not within the delimitation of this thesis.

- *The DSL provides one and only one good way to express every concept of interest (unique).*

  Any input specification written in the DSL is defined by its set of input configuration statements, field mappings and field transformations. There is only one way of expressing a certain input configuration and if any part is added or removed it will no longer represent the same input configuration.

- *Each DSL construct is used to represent exactly one distinct concept in the domain (orthogonal).*

Three of the four elements of the DSL (input configuration, field mapping and field transformations) correspond to the three main concepts of the domain (described in section 4.3). The fourth element, the comment element, does not correspond to a domain concept but was decided to be included anyway (for reasons described in section 5.2).

- *The language constructs correspond to important domain concepts. The DSL does not include domain concepts that are not important.*

  As mentioned regarding the previous sub-characteristic, all elements of the DSL except the comments correspond to the most important parts of the domain.

- *The DSL does not contain conflicting elements.*

  The main elements of the language (input configuration, field mappings and field transformations) all have different purposes. The only part that can be done in more than one way is the field mappings which can be done either by column name or by column index.

- *The DSL is at the right abstraction level such that it is not more complex or detailed than necessary.*

  This sub-characteristic was discussed during the second focus group interview (described in section 7.3).

7.4.2 Extensibility

- *The DSL has general mechanisms for users to add new features (adding new features without changing the original language).*

  The proposed solution does not offer a way for users to add new features (such as additional transformations and fields in the trade model) without implementing them in the semantic model. Additional transformations and fields in the internal trade model can however be added to the DSL without modifying the grammar or the DSL parser.

Chapter 8

Discussion

In this chapter we discuss the results from the development and evaluation of the proposed solution. The chapter is divided into three parts. In the first part, the findings from the evaluation of the DSL are discussed. In the second part, we discuss the impact of choices made during the development process. In the last part, we discuss general lessons learned about data normalization as a DSL domain.

8.1 The Proposed Solution

The proposed solution meets the requirements listed in section 4.3, but it was clear during the second focus group interview that the problem domain defined by the delimitation of this thesis was considered to be a simplified version of the actual problem domain. Since the intention of the thesis was to investigate how a DSL for this purpose could be designed and implemented rather than providing a complete solution for the data normalization problem, this was not considered to be a problem that would affect the outcome of this thesis in a negative way.

The findings from the first focus group interview show that the proposed solution in general is comprehensible for domain experts and that the language elements correspond to elements of previous solutions at the company. The elements that were considered to be the most understandable were the field mappings and the configuration statements that were strictly related to the input format. The reason for this was probably that the interviewees from the first interview all had extensive experience of working with client submitted CSV files and that the field mappings are just a way of pointing out where in those files the expected data columns (according to a trade data specification) can be found.

The remaining parts of the DSL, the transformations and the grouping configuration, was considered to be less understandable but for different reasons. The concept of the transformations was considered to be easy to understand, but the use of the transformations was deemed to require documentation of the available transformations. This probably applies to the solution as a whole, proper documentation would be needed for it to be useful.

Regarding the grouping configuration, the interviewees from the first interview stated that it was not obvious how a user was supposed to decide which columns to specify as grouping columns. This is an interesting point that may not be obvious for someone who knows the structure of the underlying trade model. The need for this grouping configuration is due to design choices made during the implementation of the semantic model and can thus be seen as a leak in the abstraction rather than a natural part of the DSL. This is related to the discussion from the second interview session about whether or not the output format should be a part of the DSL specifications. Since the normalization process takes as input a flat format and outputs a hierarchical format, the hierarchy must somehow be defined. Having the whole output format as a part of the DSL specifications instead of just having the grouping configuration would probably be a more suitable and understandable abstraction of the problem domain.
Another interesting finding from the first interview session was that the restricted flexibility of the proposed solution was considered to be a quality that would not only enhance the readability but also the maintainability of the specifications. Allowing users with little or no programming experience to extend the DSL by themselves or even allowing them to mix the DSL specifications with GPL code was considered to lead to a system that is hard to maintain due to code duplication and unconventional use of the flexibility offered by a GPL. This indicates that the effects of having general mechanisms for users to add new features, which is described as a quality characteristic by FQAD, depend on the intended use of the DSL and the level of programming experience of the users.
The proposed solution is expressive according to the sub-characteristics provided by FQAD, but it could be argued that these sub-characteristics can be a bit misleading when evaluating smaller DSLs. A DSL that consists of only a few elements will be less likely to have conflicting elements and more likely to offer just one way of expressing a given solution. This means that it is difficult to determine if the DSL is expressive because the language elements are well-designed and have a clear separation of concerns or simply because it is a small language.

8.2 The Development Process

The choice of developing an external DSL allowed for the design of a syntax that is arguably concise and contains a minimal amount of syntactic noise. It is possible that this concise syntax could be achieved with an internal DSL as well, but the design of an internal DSL will always be limited by the syntax of the host language. When developing an external DSL the syntax of the language can be designed before thinking about how it should be implemented. This gives a clear separation between the design and the implementation of the DSL, which will probably result in a language design that is less influenced by implementation details.
The development of an external DSL also allowed for a complete control of the execution environment and restriction of the ways in which the language could be

used. The elements of the language and any interactions between those elements must be designed and implemented by the developer of the language (if there are no mechanisms for users to extend the set of language elements by themselves). This enables the developer to clearly restrict the flexibility of the language. As described in section 8.1, the interviewees believed that this restricted flexibility enhances the maintainability of the system (especially if the intended users have little or no programming experience).
Using a parser generator meant that an understanding of formal grammar notations had to be acquired before starting the implementation process, but the amount of time spent on this is not comparable to the time that would have been spent on developing a parser from scratch. Using a parser generator also allowed for the syntax of the DSL to be modified just by modifying the grammar specification, which probably also reduced the amount of time spent on the implementation. In general, the use of a parser generator made the expected steep learning curve of developing an external DSL less evident.

8.3 Data Normalization as a DSL Domain

As stated in subsection 7.3.2, there are aspects of the data normalization problem in triCalculate that make the problem domain difficult to define. The aspect of deciding whether or not the DSL should be a general solution for all data normalization needs at the company is rather specific for the problem domain of this thesis, but the aspect of what assumptions can be made about the input formats should apply to all data normalization problems.
The overall goal of DSL design is to produce a language that is as simple as possible without losing the flexibility needed to cover the problem domain. One of the main guidelines (mentioned in subsection 2.4.2) for the DSL design process is to keep the language simple by avoiding all unnecessary generality. When the problem domain is data normalization (of some kind), this means that the level of flexibility of the DSL should be decided by the set of possible input data formats. This set will depend on what assumptions can be made about the input data formats. These assumptions could be based either on actual requirements on the input data formats or on an analysis of a representative data set. If there are no requirements on the input formats and there is no representative data set available for analysis, not much can be assumed about the formats of the input data and it will be very difficult to clearly define the problem domain and decide on an appropriate level of flexibility for the DSL.

Chapter 9

Conclusions

In this concluding chapter we present a summary of this thesis and the conclusions that could be drawn from it. We also suggest future work in the area.

9.1 Summary

The purpose of this thesis was to explore how a DSL for normalization of financial derivatives data could be designed and implemented. The proposed solution is an external DSL implemented in Python using the parser generator ANTLR.
The following list is a summary of the conclusions that could be drawn from the evaluation of the proposed solution and from this degree project as a whole:

- The development of a DSL for normalization of data should only be initiated when there are either clear requirements regarding the formats of the input data or if there is a representative data set available for analysis.

- Even though the output format of the normalization process will be the same regardless of input, including the output format as a part of the DSL specifications will probably result in a more understandable abstraction of the problem domain.

- Offering a high degree of flexibility and extensibility to users with little or no previous programming experience may come at the high price of a system that is difficult to maintain.

- The development of an external DSL allows for a complete control of both the language syntax and the flexibility in the use of the language.

- Using a parser generator when implementing an external DSL makes the steep learning curve (which is often seen as the main disadvantage of external DSLs compared to internal ones) less evident.

Since the problem statement of the thesis was very open-ended, there was no concrete definition of done. This means that not much can be said about whether or not the problem statement has been sufficiently answered. Given more time, the delimitation of the problem domain could have been wider and the proposed solution more comparable to the existing solution in triCalculate. This would probably have resulted in a more general solution that would have allowed for a deeper understanding of the suitability of data normalization as a DSL domain. Nonetheless, the project fulfills its purpose of being an exploratory study of DSL development. The conclusions listed above should apply to all data normalization DSLs and the last three should apply to all cases of DSL development.

9.2 Future Work

During the interview sessions a number of improvements to the proposed solution were suggested. One suggestion mentioned during both of the sessions were to create a GUI that would be used to specify field mappings, apply transformations, preview the results and generate the DSL specifications. It would be interesting to investigate how such a GUI could be designed based on previous related work (e.g. the work on Wrangler [15] by Kandle et al.).

In general, it would be interesting to see more studies where established approaches and guidelines for DSL development are applied on industrial problem domains.

Bibliography

[1] Michael Durbin. *All about derivatives.* McGraw Hill Professional, 2006.

[2] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[3] Martin Fowler. *Domain-Specific Languages.* Pearson Education, 2010.

[4] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[5] W3C website. `http://www.w3.org/html/`. Accessed October 20, 2014.

[6] Make website. `http://www.gnu.org/software/make/`. Accessed October 20, 2014.

[7] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II*, pages 291–373. Springer, 2008.

[8] Debasish Ghosh. *DSLs in action.* Manning Publications Co., 2010.

[9] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design guidelines for domain specific languages. In *The 9th OOPSLA workshop on domain-specific modeling*, pages 7–13. Citeseer, 2009.

[10] ANTLR website. `http://www.antlr.org`. Accessed October 28, 2014.

[11] Gökhan Kahraman and Semih Bilgen. A framework for qualitative assessment of domain-specific languages. *Software & Systems Modeling*, pages 1–22, 2013.

[12] FQAD assessment forms. `http://www.metu.edu.tr/~e160061/`. Accessed January 15, 2014.

[13] Markus Voelter. Best practices for dsls and model-driven development. *Journal of Object Technology*, 8(6):79–102, 2009.

[14] E. Hautus. Case study: A DSL for normalization of financial data sets. Presented at the Software Development Automation Conference (SDA) in Amsterdam, 2014.

[15] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3363–3372. ACM, 2011.

[16] "What is Python? Executive Summary". `https://www.python.org/doc/essays/blurb/`. Accessed January 15, 2014.

[17] "Comparing Python to Other Languages". `https://www.python.org/doc/essays/comparisons/`. Accessed December 9, 2014.

Appendix A

Sample Data

A.1 Input data sample

This is a sample describing the format of the input data. The sample data specifies a foreign exchange swap trade.

```
trade_counterparty,trade_netting_set,trade_id,trade_instrument_type,
    trade_date,trade_quantity,leg_currency,leg_business_center,leg_
    business_day_conv,leg_notional,cashflow_payment_date,cashflow_
    notional,trade_valuation_date,trade_valuation_currency,trade_pv
P1,P1,1,FXSWAP,2013-05-17,1,EUR,GBLO,FOLLOWING
    ,-69120000.0,2019-10-18,-69120000.0,2014-12-03,EUR,-922179.596276
P1,P1,1,FXSWAP,2013-05-17,1,USD,GBLO,FOLLOWING
    ,89833678.0,2019-10-18,89833678.0,2014-12-03,EUR,-922179.596276
```

A.2 Normalized data sample

This is a sample that describes the format of the normalized data. This data corresponds to the sample input data in section A.1.


```
[
  {
    "pv": -922179.596276,
    "trade_date": "2013-05-17",
    "instrument": {
      "instrument_type": "FXSWAP",
      "legs": [
        {
          "currency": "EUR",
          "nominal": -69120000.0,
          "cash_flows": [
            {
              "payment_date": "2019-10-18",
              "nominal": -69120000.0,
              "type": "FIXED_AMOUNT"
            }
          ]
        },
        {
          "currency": "USD",
          "nominal": 89833678.0,
          "cash_flows": [
            {
              "payment_date": "2019-10-18",
              "nominal": 89833678.0,
              "type": "FIXED_AMOUNT"
            }
          ]
        }
      ]
    },
    "trade_id": "FXSWAP-1",
    "party": "ABC",
    "counterparty": "P1",
    "pv_currency": "EUR",
    "quantity": 1
  }
]
```

Appendix B

Focus Group Interview Questions

B.1 Questions from the first focus group interview

- Which of the language elements do you find easy to understand?
- Which of the language elements do you find hard to understand?
- Which of the language elements were expected?
- Which of the language elements were not expected?

B.2 Questions from the second focus group interview

- Is the design of the language an appropriate abstraction of the problem domain?
- Which plausible input data formats can not be handled by the proposed solution?
- Do you think that it would be possible to extend the proposed solution to obtain a complete coverage?

Appendix C

Sample DSL Specification

```
1  # Default FXSWAP specification
2
3  input:
4      delimiter = ","
5      has_header = True
6      instrument_type = FXSWAP
7      trade_group_column = "trade_id"
8      leg_group_column = "leg_currency"
9      row_filter_column = "trade_instrument_type"
10     row_filter_value = "FXSWAP"
11
12 field_map:
13     trade_party -> default_value("ABC")
14     trade_counterparty -> column_name("trade_counterparty")
15     trade_id -> column_name("trade_id")
16     trade_date -> column_name("trade_date")
17     trade_quantity -> column_name("trade_quantity")
18     trade_pv -> column_name("trade_pv")
19     trade_pv_currency -> column_name("trade_valuation_currency")
20     trade_instrument_type -> column_name("trade_instrument_type")
21     leg_currency -> column_name("leg_currency")
22     leg_nominal -> column_name("leg_notional")
23     cashflow_type -> default_value("FIXED_AMOUNT")
24     cashflow_payment_date -> column_name("cashflow_payment_date")
25     cashflow_nominal -> column_name("cashflow_notional")
26
27 transformations:
28     trade_id -> add_prefix("FXSWAP-")
```

www.kth.se