Bridging the Worlds of Textual and Projectional Language Workbenches

Bartels, J.

Award date:
2020

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

Bridging the worlds of textual and projectional language workbenches

Master Thesis

Jur Bartels

Supervisors:

Prof. Dr. M.G.J. van den Brand
M. Verano Merino, MSc

1.0

Eindhoven, January 2020

Abstract

Language composition is a very common approach used by language engineer used to combine features of two separate languages into one. In order to apply this approach however, the two languages need to exist within the same "world". We can not arbitrarily compose languages where one exists as a grammar in the textual language workbench (LWB), and one exists as a model in the projectional LWB. In this project we bridge the textual and projectional LWBs by creating a process for transforming textual grammars into projectional models. Language composition will then be possible by "importing" a textual language into the projectional LWB, and applying the composition with other projectional languages there. There are three defined sub-goals. First of all, we need to be able to map the structure of source textual grammar rules to equivalent projectional models. This step also includes the problem of how information is to be exchanged between the LWBs. Secondly, we want to improve the usability of the imported language within the projectional LWB. There are differences between the textual and projectional LWBs which make languages designed for one less usable in the other. To remedy this we apply several heuristics to the imported language within the projectional LWB, which improve the "default" usability. Finally, after the language has been imported into the projectional LWB, we also want to be able to import programs written for the textual source grammar as projectional models. This removes the need to recreate already written programs within the projectional LWB manually. In the end, we aim to create a tool which allows for the bridging of both LWBs with the least amount of manual effort required.

Preface

I would like to thank Mauricio Verano Merino for his tutorship, advice, time and effort. His knowledge of Rascal was an invaluable resource. I would like to thank Prof. Van den Brand for supervising this thesis. I would also like to thank my parents for their continued support throughout my studies.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Problem statement

When attempting to solve any problem we face, we often have to choose which of our available tools to apply. Some tools are generic and can be applied in different contexts, but may have suboptimal results. Other tools may be limited to a specific use case, but are optimised. The Jack-of-all-trades versus just the right thing. The same very much hold for the field of programming languages. Generic all-purpose programming languages such as Java, C++ and Python can be used create programs over many different domains, from games to accounting software and mobile apps. However, they are large languages with many rules and layers that require time and effort to learn, while also being limited by the need to remain generic. Domain-specific languages (DSLs) are programming languages developed for a particular problem domain [24]. By using domain knowledge a DSL can be optimised to the specific problem at hand, and offer a smaller, more focused language. This in turn can make developing solutions for problems within the specific domain more streamlined and effective [24].

There is however a price to pay for the specific approach of DSLs. Due to their specific nature, they can often not be reused between different problems, or may lack functionality when domains overlap for a certain problem. Developing a DSL from scratch takes time and domain knowledge. To make the process of developing DSLs more efficient for the language engineer, tools were developed for efficiently developing, prototyping and using DSLs. These tools are called language workbenches (LWB) [14]. LWBs typically support the definition, reuse and composition of DSLs, and an integrated development environment (IDE) in which the DSL can be used.

Historically, DSLs are defined textually [22, 20]. The language definitions are written in plain text using the specified formalism of the LWB such as ANTLR [21], Rascal[18] or the Xtext grammar language [10]. The LWB can take these textual language definitions and generate tools for the language, such as parsers and IDEs. An alternative, more recent approach [24] is that of projectional language workbenches, where language definitions are model-based rather then text-based, which can have several advantages such as being able to apply model-based engineering and projectional editing. Due to their fundamental difference in how language definitions are represented, textual and projectional LWBs are not directly interoperable. It is not possible to define a language in a textual LWB, and later reuse or extend this language in a projectional LWB without fully recreating the language definitions within the new environment. This presents a major problem for the adoption of projectional LWBs, as previous work cannot be easily reused without spending time to migrate languages by hand. Time and effort which could be spend actively solving new problems. To this end we arrive at the goal of this thesis: to create and implement an approach for bridging textual and projectional language workbenches. Specifically, the approach should be able to:

- Generate a projectional language definition given a textual language definition.
- Generate a projectional editor definition derived from a textual language definition.
- Generate a projectional program model given some source code, provided the language of said source code is available in the projectional language workbench.

The overall process of moving a language or artifact from the textual to the projectional LWB should be as user-friendly and efficient as possible. This means that it should be straight-forward to use and require the least amount of user interaction possible without compromising on the correctness of the language or artifact that is moving between the LWBs.

Not that in this work, we will explore the bridge in one direction. That is, we create an approach for transferring languages and artifacts from the textual LWB to the projectional LWB. Moving in the other direction, which has its own reasons for being desirable, is discussed in Chapter 7.

1.2 Use cases

In this section we will list 2 of the main use cases for the bridge between the textual and projectional LWBs :

1.2.1 Reuse of existing textual languages

Historically, the textual approach has been the the most popular approach to language engineering by far. Therefor it also has the most existing artifacts such as grammars and programs. Any language engineer wishing to take advantage of features of the projectional LWB will have to recreate this previous work in the new LWB manually, which takes time and effort to do correctly. By automating this step of moving languages and artifacts between the LWBs, the language engineer can spend more time on developing new ideas instead of re-implementing old work in a different LWB.

1.2.2 Composition of textual and projectional languages

Language composition is a common approach used by language engineers in order to combine features of multiple languages without re-implementing them. Take for example language A which defines the language of a* (0 to n occurrences of the character “a”) and Language B which defines the language b*. Now a certain problem calls for a language containing both a* and b*. Once solution would be to create new language C which defines both a* and b*. However since we already have languages which implement both, it would be simpler and more efficient to reuse the already existing definitions to create language AB. This is know as language composition: combining one or more element of one language with one or more elements of another to create a new language which combines both. This is not possible for two languages developed using different types of LWBs, as they do not have a common interface. By allowing a textual program to be imported within the projectional LWB, it can be then be composed with other projectional languages.

1.3 Structure of the thesis

This section shows the structure of this thesis. First, we introduce some background information on SLE, more specifically on textual LWBs and projectional LWBs, their features and differences. This is the basis for understanding our solution. Chapter 6 presents some related work to show the current landscape of bridging the gap between the LWBs. Chapter 3 presents a high-level overview of the proposed approach on how solve the previously set-out sub-goals. This leads into the more practical real-world implementation of the approach for two specific LWBs, one textual

and one projectional. We show the results of the process and evaluate them. Finally we finish up with discussing the future work and conclusion of this thesis.

Chapter 2

Preliminaries

2.1 Programming Languages

Whether we are working within a textual or projectional language workbenche (LWB), we are creating and manipulating one thing: programming languages. Maurizio Gabbrielli et al. [15] identify four major components for programming languages: syntax, semantics, pragmatics and implementation. Generally, a syntax defines what strings are valid for a given language. Semantics defines what a valid string means in terms of execution. Pragmatics are concerned with how to use valid strings (programs) whereas implementation, which is a component unique to programming languages and differentiates them from linguistic languages, determines how a syntactically correct string is executed in accordance to the semantics.

All these components exist in both LWBs, although their representation and implementation differ. In this project we focus on the syntax and pragmatics of languages.

2.2 Language Workbenches

A language workbench is a tool for simplifying the creation and use of programming languages [13]. Martin Fowler, who coined the term language workbench [14], identified the following components of a language workbench:

- Specification of the language concepts or metamodel
- Specification of the editing environments for the domain-specific language
- Specification of the execution semantics, e.g. through interpretation and code generation

Rascal and Jetbrains MPS are both examples of such language workbenches. The both define their own specifications and representations of languages. However, as we will see, the difference in representation makes interoperability between both language workbenches difficult.

2.3 Textual languages

In textual languages, as the name implies, the definition of the language components is done in plain text format. We will quickly go over them.

Syntax

The syntax of a textual language usually takes the form of a Context-Free Grammar (CFG [16]). A CFG describes a set of rewriting rules from which valid string for a given language can be derived. A CFG has four components:

- A set of non-terminal symbols
- A set of terminal symbols
- A set of production rules with a non-terminal on the left hand side and one or more non-terminal or terminal symbols on the right hand side.
- A start symbol

For example, if we want to describe the CFG of a language for strings of any length containing only the letter "a", so "a", "aa", "aaa" and so forth, the grammar would look something like this:

$$Start \rightarrow A$$
$$A \rightarrow AA$$
$$A \rightarrow a$$

Where the capital A denotes a non-terminal symbol, the lower case a is a terminal symbol and "Start" denotes the start symbol.

By applying these rewrite rules we can form strings:
$Start \rightarrow AA \rightarrow aA \rightarrow aa$

Once a string contains no more non-terminal symbols we can not rewrite it any further.
For programming languages, if there exists such a derivation from the start symbol to the string representing the program, the program is syntactically valid.

Semantics

There are various types of semantics that can be applied to textual languages, and these depend mostly on the implementation. Examples are type systems and evaluators. Just as the other components, these are usually described in some formal way within the meta-programming environment.

Pragmatics

Pragmatics are a little less clearly defined. Once concept that we can point to is layout. For textual languages, since they are represented as a string, there is often a lot of freedom in how to form the layout. Some meta-programming languages also can impose restrictions on layout and whitespace.

2.3.1 Rascal

Rascal [19] is a (textual) meta-programming language and a language workbench designed and developed at Centrum Wiskunde & Informatica (CWI). Rascal has its own grammar formalism for the syntax of languages and supports an expressive type system. It has native implementations for parsing programs using Rascal grammars and code generation from the resulting parse trees.

An example of a production rule in Rascal is shown in Listing 2.1. There are both terminal (e.g.,"for", "do") and non-terminal (`Statement`, `Expression`) symbols.

```
1       forStat: "for" Statement s1 ";" Expression e1 ";" Statement s2 "do" {Statement ";"}* s3 "od"
```

Listing 2.1: Example of a Rascal production rule

In this project we will be using Rascal as our textual language workbench (LWB).

2.4 Projectional languages

2.4.1 Meta-models in projectional editors

A projectional language is represented by a meta-model of the Abstract Syntax Tree (AST) of the language, with which the language engineer can interact directly. Language elements are represented as nodes, which can have attributes and relations to other nodes, such as references or child nodes. Language users can create programs by constructing a valid instance of the meta-model.

For example, let's take the same language as before, the language of all strings containing one or more occurrences of the character "a". The sketch of an AST meta-model of such a language can be seen in Figure 2.1. This model have a root node `Start` which has a single child node, of type `A\_Interface`. The model contains two nodes implementing this interface, and either can be substituted for *A_Interface* during the construction of a program. A valid program AST contains no interface nodes (all interfaces must be replaced by a node implementing the interface). The node *A_2* contains a single leaf node with a string literal "a". It is a leaf node since it has no children or references, and therefor no more elements will spawn from this node. A valid projectional program must have only leaf nodes at the end of every branch.

Now that we have a meta-model, we can create models of it to represent programs in the language. Figure 2.2 shows the AST representing the string "a", while Figure 2.3 shows the AST for the string "aaa".

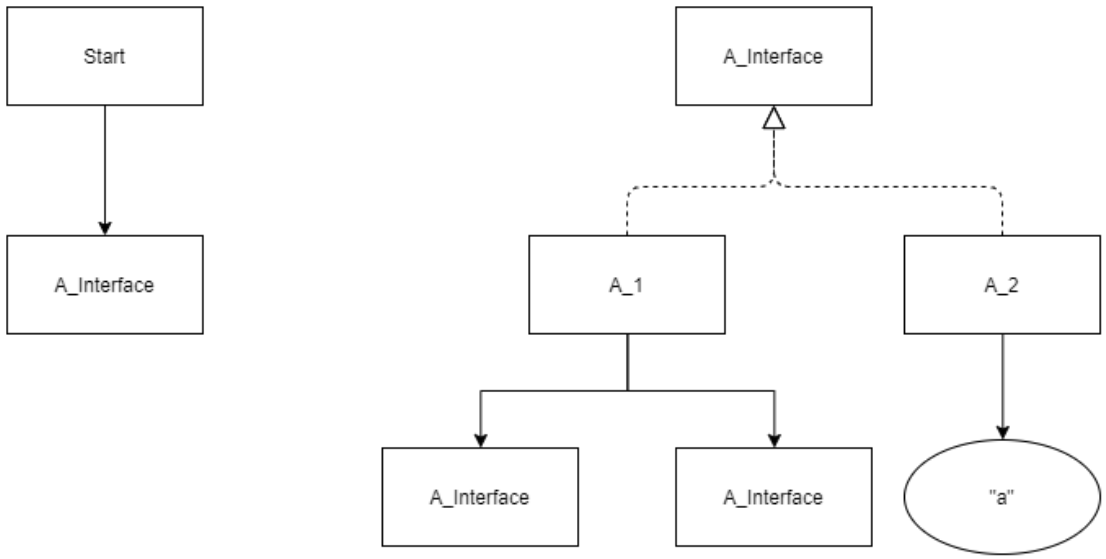

Figure 2.1: AST meta-model sketch

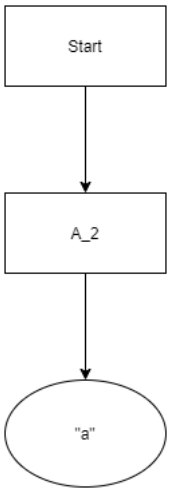

Figure 2.2: AST representing the string "a"

2.4.2 Jetbrains MPS

MPS is a projectional language workbench. It offers powerful facilities for creating model-based projectional languages, and has an integrated integrated development environment (IDE) to interact and use said languages.

Languages in MPS are defined using so-called language Aspects. There are several pre-defined aspects available, such as *Structure*, *Editor*, *Behaviour* and *Constraints*. Language engineers can also implement their own concepts. The core component of a MPS language is the *Structure* Aspect, which is the only aspect which must be defined for a language to be valid. This thesis will focus on the Structure and Editor Aspects. We will take a closer look at both aspects to make it easier to understand the implemented solution later on.

Structure

The *Structure* aspect is the core of MPS languages. It defines the AST meta-model on which the language is build. It establishes the language concepts and their relations to each other.

An example of the definition of a language element, or a concept in MPS is shown in Figure 2.4. Concepts have names and may have attributes, and most importantly, define the child nodes of the for loop. In this case, the for loop consists of a statement (loop variable declaration), an expression (loop condition) and another statement (loop increment). Then there is a list of 0 to N statements forming the body of the loop. Note that there are no literals such as "for", "do" in the structure aspect. These belong to the Editor aspect.

Editor

The *Editor* aspect of a MPS language defines the visualization of the language nodes. A separate visualization can be defined for each node in the language that can appear in the editor. If the language does not specify a visualization for a node, MPS will generate a default reflective editor based on the information available in the node[5].

Figure 2.5 shows the editor aspect for the *forStat* concept introduced in Figure 2.4. The variables are references to the child nodes that are available for the concept, i.e the *s1* is a single

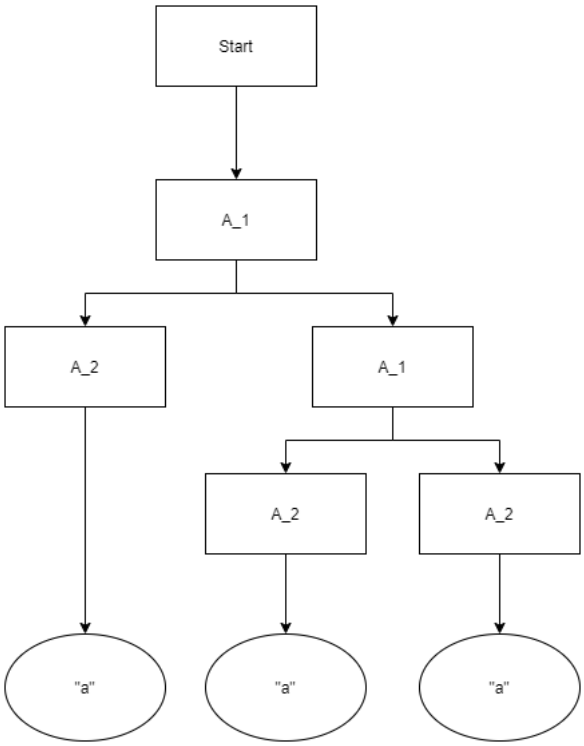

Figure 2.3: AST representing the string "aaa"

Statement.

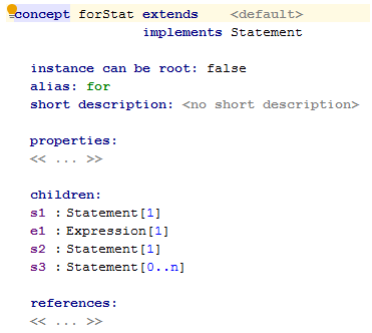

Figure 2.4: MPS strcuture aspect definition of a for loop

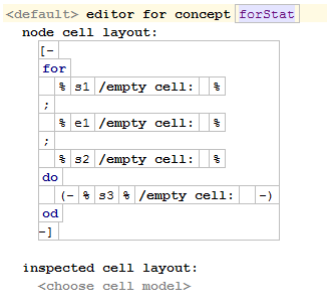

Figure 2.5: MPS editor aspect definition of a for loop

2.5 SModel language

MPS also has its own meta-model in the SModel language. We will use the SModel language to construct projectional language definitions such as the concepts of the structure aspect and the editor cell models.

2.6 Comparison of textual and projectional languages

As seen in the previous sections, projectional LWBs and textual LWBs differ on how languages and programs are represented and interacted with. There are also some other important aspects in which they differ, as shown in Table 2.1. We will go into further detail for the last three aspects from said table as they have not been covered by the previous sections

2.6.1 Language interaction

Through their different forms of representation, the way language engineers and users interact with the language can differ greatly in textual and projectional languages. Usability is a big factor here. The creation and editing process of both languages and their programs should be efficient for the language engineer and user. For an example we can look at the difference in interaction between Rascal and MPS. Since Rascal is text-based, the language is written in plain text format, using free typing in any desired text editor. Typing can be fast, and offers a lot of personal freedom in layout. However, it requires the memorization of the language's syntax. Spelling mistakes and wrong commands will lead to invalid programs, which will only be found out when parsing. This can lead to a write, parse, and rewrite cycle until the program is syntactically correct.

This stands in contrast to MPS as projectional editor. In MPS, the editor is based entirely

	Language representation	Program representation	AST generation	Interaction with the language	Tool integration	Maturity
Projectional	AST meta-model	AST model	Directly from program	Model construction and transformation	Narrow, must interface with specific models	Immature
Textual	Plain text	Plain text	Generated by parser	Free typing	Broad, can interface through plain text	Mature

Table 2.1: Comparison of textual and projectional language workbenches

on the auto-completion menu. At any time, the language engineer or user can bring up a contextual menu[6] that displays the elements that can be inserted at this node. in fact, these are the only nodes allowed. It is not possible to create a syntactically incorrect program of an MPS language. This is possible due to the AST meta-model, which guides MPS exactly which nodes are candidates at any given time. However This auto-completion method has its disadvantages. Searching the menu can become quite cumbersome when there are many possibilities. The auto-completion menu also adds user actions in the form of button presses every time a new element is inserted. There are ways to improve the default behaviour and optimize it for a given language, but this takes time and effort on the languages engineer's side.

2.6.2 Tool integration

A big advantage of the textual world is the ubiquitous nature of the representation format. There exist many tools that operate on this format, and creating new one is trivial using most modern programming languages. There is nothing proprietary about the Rascal file format, any tool that can open plain text can open and interact with the language representation.

The projectional representation is model-based. How these models are stored is very specific to the given environment. MPS for example, has a default storage format based on XML. It also allows for the definition of alternate custom-made storage formats for storing and loading MPS models [4]. Interacting with the model representation from outside of MPS can be cumbersome, as we will see in the our implementation later on. It requires knowledge of MPS internal API and the internal model representation. To load MPS models in an environment other than MPS itself is far from trivial, so creating tools for MPS mostly has to happen inside MPS. This limits the scope and capabilities of tools to those allowed by MPS itself.

2.6.3 Maturity

Textual languages have been around for a long time and are widely spread. There has been a lot of research within this field. Projectional languages are quite young in comparison, and their use, whilst growing recently, is still far below that of textual languages [13]

Chapter 3

High-level approach

3.1 General approach

In this chapter we will introduce the proposed solution to achieving the sub-goals of importing textual languages and programs into the projectional language workbench.
We will first do so in an abstract manner, looking into the process rather than the implementation on specific LWBs. The next chapter will show an implementation of the approach on two such tools, however it is important to show that the solution is not limited to these tools, and can be applied to LWBs in general.

3.2 Grammar structure

The first goal is to be able to generate a projectional language definition from a textual language definition. This goal gives rise to a few different questions:

1 What elements (symbols, production rules, lexicals etc.) of the textual language definition do we need to construct the projectional language definition?

2 How do we get these required elements from the textual LWB.

3 How do we map the elements between the textual LWB and the intermediary format?

4 How do we map the elements between an intermediary format and the projectional LWB

5 How do we construct an projectional language definition from these elements once they have been transferred to the projectional LWB.

Answering these five questions will lead us to the desired process able to communicate a program from the textual LWB to the projectional LWB.

The next few sections will each answer some of the questions above.

3.2.1 The AST

First, following question 1, we need to identify what information is required to construct a language in the projectional LWB. As described in Chapter 2, a language in the projectional LWB consists of a model of the abstract syntax tree, which is directly manipulated to change the language itself. The AST model itself is quite simple in structure. An AST is a tree structure with nodes denoting non-terminals and leaves denoting terminals. For example, lets take a look at the AST of an "add" operation, which can be applied to all numerals. Numerals are defined as either integers

or doubles. The AST of such an operation would look something like Figure 3.1. ADD references two nodes of type NUMERAL, which in turn reference two options for the leaf nodes, either an integer or a double.

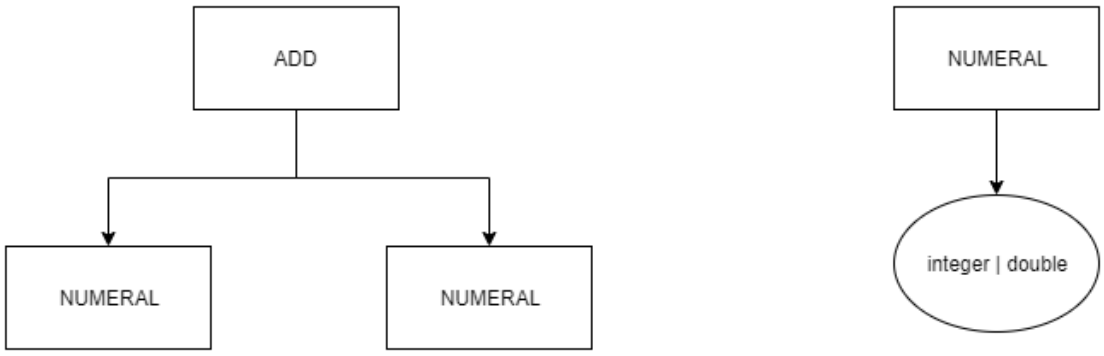

Figure 3.1: AST of an add operation

So to go back to our original question 1, we need to be able to construct such an AST model for the entire language. that means we need the following elements:

- Non-terminal nodes
- Terminal leaf nodes
- The relationships between the nodes

The next question is then how we can obtain these from the grammar in the textual LWB.

3.2.2 Textual grammars to ASTs

The notion of having an AST representation of is not exclusive to the projectionl LWB. In the textual LWB, parsers use a program in combination with a grammar definition to construct an AST of said program, which can in turn be used for further processing, such as code generation. However, for in our case we do not have a program, just the grammar. If we can somehow create an AST from the grammar alone, without the program, we would immediately have the elements required for construcing an equivalent AST in the projectional LWB.

So lets take a look at context-free grammars and whether we can reconstruct an AST from the rules alone.

A context free grammar consists of the following elements:

- A set of non-terminal symbols
- A set of terminal symbols
- A set of production rules
- A start symbol.

Note that immediately, the terminal and non-terminal symbols can be used as leaves and nodes respectively. We just need to figure out the relations between the nodes to construct the AST. We can use the production rules and the start symbol to figure out the structure of the AST.

take the following example where ADD is the start symbol:

$ADD \rightarrow NUMERAL + NUMERAL$
$NUMERAL \rightarrow integer$
$NUMERAL \rightarrow double$

Using the start symbol as our AST root, we look at the production rules that follow from it. In this case, it is only one: $ADD \rightarrow NUMERAL + NUMERAL$. Here we see that this production rule contains three elements: Two NUMERAL non-terminals and one literal, "+". For now we will discard the literals as they will not lead to any subtrees. We add two nodes to our AST root of type NUMERAL. Since NUMERAL is a non-terminal, it should also have at least one production rule. In the example, it has two. We take the first rule and find that NUMERAL can be replaced by a terminal symbol integer. Similarly, the second rule gives us that a NUMERAL can also be replaced by a double. However, in both cases NUMERAL only references one element. Thus we add one child node to the NUMERAL node, which can be either of type "integer" or type "double". These steps are also shown in Figure 3.2.

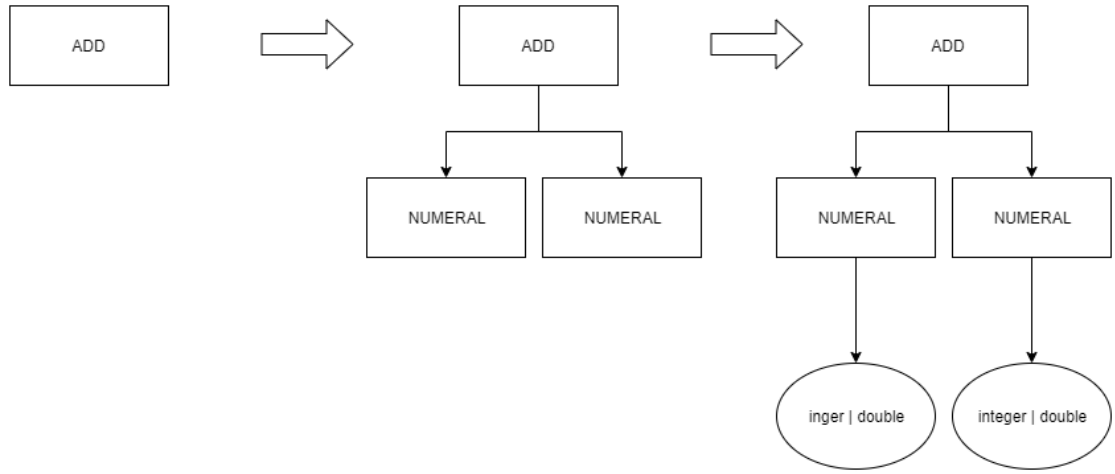

Figure 3.2: AST construction from a grammar

In this form however, the AST contains duplicate information for the NUMERAL node, since we define it twice. Instead, we can make the AST a forest, in which child nodes are just references to the definition, as shown in Figure 3.3.

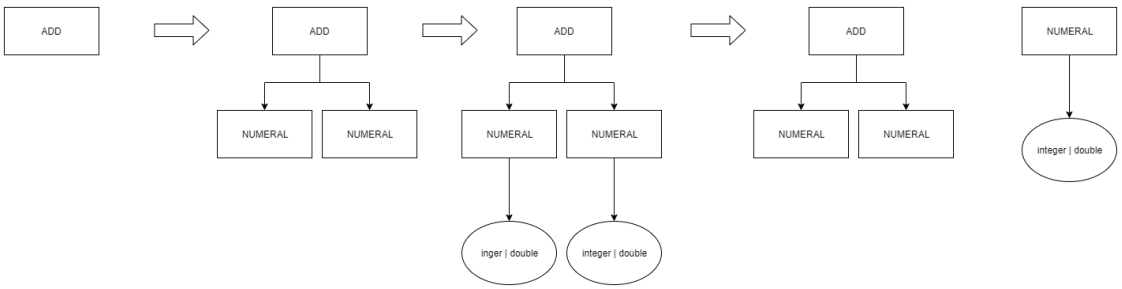

Figure 3.3: AST forest construction from a grammar

This structure now contains all the elements required for the AST construction in the projectional LWB. However, to make use of it we first need to be able to communicate it across the LWBs in the intermediary format.

3.2.3 Intermediary format

After we obtain the language definition of the textual language, we must transfer them to the projectional LWB in order to construct a projectional language definition. There are several ways this could be achieved:

- A direct bridge between the textual and projectional LWBs where one can directly communicate with the other.
- An indirect bridge through an intermediary storage format. Here one LWB produces the intermediary which the other can consume. The LWBs do not communicate directly.

Both approaches have merits. The direct approach allows for a more simple solution, where the projectional LWB can ask the textual LWB for the desired information, and the textual LWB can directly respond with said information. However, this does require specific knowledge of the other LWB. For example, communicating with the Rascal LWB might be very different from communicating with ANTLR, as both use different formalisms for language definitions. There is no standardised interface between the textual LWbs that can be used to abstract over these language definitions. At some level, the mapping from the language formalism of the textual LWB and the projectional LWB must be defined. As a result, every time we introduce a new language workbench, we must define this mapping again.

Alternatively, we can communicate indirectly, through the use of an intermediary format. In this approach, the textual and projectional LWBs have no direct knowledge of each other. They only know the intermediary format, which acts as a contract. The textual language workbench produces the intermediary format. The projectional language workbench then consumes this intermediary format, using it as input to generate the projectional language definition. The mapping is now split into two smaller mappings instead of one: First there is a mapping from the textual language definition the intermediary format, and secondly there is a mapping from the intermediary format to the projectional language definition. Splitting the mapping into two allows us to reuse either part. Now if we want to support a new textual language workbench, we only need to define the mapping of the language formalism to the generic intermediary format.

In our approach we use an intermediary format. The level of reusability gained by this approach is worth the slightly more complex two-step process of production - consumption versus the simple one step approach direct communication can give.

3.3 Projectional editor - Visual aspect

3.3.1 Usability in the projectional editor

Before we can introduce our solution to the second goal of this thesis, we should make clear exactly what problem it solves in the context of the overall goal, to bridge textual and projectional LWBs. As stated in Chapter 1, One of the goals of this project was to improve the usability of the language after it had bridged the LWBs. But what do we mean with "usability" in the abstract context of languages in general, without referencing specific implementations such as Rascal or MPS?

To understand the problem here, we can look back at the previous section of this chapter and identify what exactly it is that we have available to us right now: The structure of the language, and the structure alone, in the form of an AST. However, there are many language features that are not directly present in the AST, but are still important for the use of a language.

One such feature is that of the *editor*. LWBs offer editors as a way for users to interact with languages. Often these can be customised by language definitions, such as declaring keywords for highlighting, layout specifications and code-completion facilities. As the user interface to the language, editors are an important aspect of a language definition in a language workbench. Thus when generating a projectional language definition from a textual language definition, we must ensure that the generated editor is as close as possible to the editor of the original language workbench.

To allow for the recreation of these features within the projectional LWB, we will extend the AST with additional information within the textual LWB. Node specific data is added to the node it references, while global data is added to the root node.

We will now go over several such additional features and how they are encoded within the extended AST.

3.3.2 Literals

We see an example of a production rule with literals in Listing 3.1. Here, the strings “for”,“do” and “od”, as well as the semicolons are literals. A textual parser can use literals to distinguish between production rules when constructing the AST. However, this information is then discarded. As soon as the parser knows a ”for” node can be inserted, the precise syntax of the string is irrelevant, only the terminal and non-terminal symbols it contains. Since literals are very important to the look and feel of a language, we will add literal information to our intermediary format and will use this information to generate a projectional editor with the literals included.

```
1       forStat: "for" Statement s1 ";" Expression e1 ";" Statement s2 "do" {Statement ";"}* s3 "od"
```

Listing 3.1: For loop production rule

3.3.3 Layout

Adding the literals alone is not enough however to recreate the visualization of production rules. We also need to know the order in which literals, terminal and non-terminal symbols are placed. To this end, we add a layout attribute to each node. This attribute is a list of elements, which can either be a symbol (either terminal or non-terminal), or a literal. The order in which the elements appear in the list is the same orders as in which they appear in the production rule. An example of this is shown in Figure 3.4

3.3.4 AST pruning

There is one additional step that we take when moving the extended AST from the textual to the projectional language workbench. In practise, large production rules are often split up in several smaller production rules by introducing additional non-terminals for readability. For example, the rule

$$A \rightarrow A|a|b|c$$

might be split into

$$A \rightarrow A|B$$
$$B \rightarrow a|b|c$$

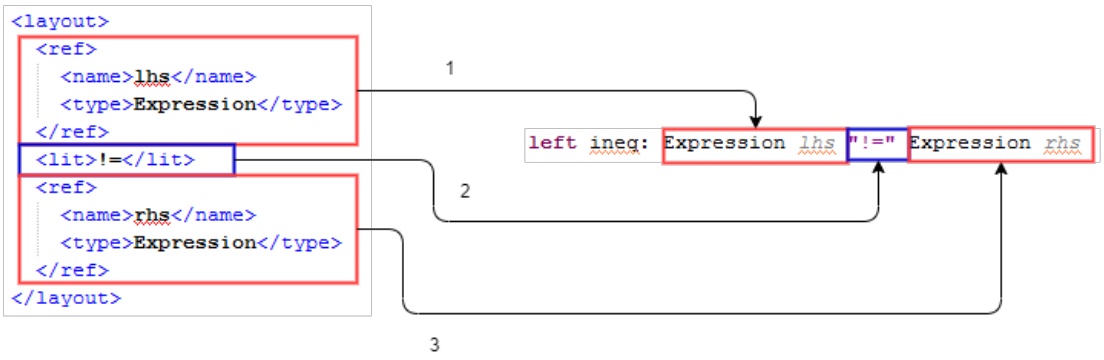

Figure 3.4: Order equivalence between the production rule and XML representation

When modelling the equivalent AST, we get the AST as shown Figure 3.5. The introduction of the new non-terminal B necessitates the introduction of a new AST node, which is only referenced once, by the production rule for which the simplification was introduced. Removing these nodes will simplify the AST and therefor the code-completion menu of the projectional editor. Thus, after before constructing the projectional model, we traverse the AST and merge any node that is only referenced once back into the referencing node. This eliminates the additional non-terminal nodes. The AST after pruning is shown in Figure 3.6.

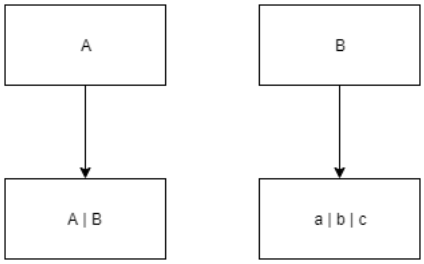

Figure 3.5: AST of split production rules

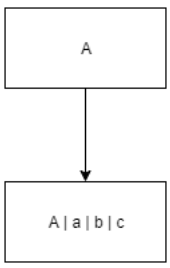

Figure 3.6: AST of merged production rules

3.4 Textual program to projectional model

The third and final goal of this thesis was to be able construct projectional models of programs from textual source code, provided the language was made available to the projectional workbench

first. The research questions attached to this goal are in many ways similar to that of importing a grammar structure. First we will parse and generate an AST representation of the program. This representation is written to an intermediary format. This intermediary format, combined with the language meta-model is then used as input to construct the projectional model of the program.

Chapter 4

Rascal2MPS implementation

4.1 Implementation architecture and overview

In this section we will discuss our implementation of the approach presented in Chapter 3. Our implementation will use Rascal as the source textual language workbench and Jetbrains MPS as the target projectional language workbench. A detailed architectural overview of the implementation is shown in Figure 4.1. Figure 4.2 and Figure 4.3 show the general process of transferring a language definition or program from the textual language workbench to the projectional language workbench.

The implementation [7] is split into two sub-components. The Rascal2XML Rascal project contains all code for the generating XML representations of Rascal grammars and programs. These XML representations are then used as input for the XML2MPS MPS plugin, which uses it to generate a MPS language definition or program model. We will discuss each of these components in more detail in their own section in this chapter.

Furthermore, we will discuss several features that were added to the implementation which are not part of the general approach, but are more tied to MPS specifically as a language workbench. While these features are not directly translatable to other projectional language workbenches, they do enhance the quality and usability of the generated MPS Language significantly.

Finally we discuss limitations that were placed on the source grammar. For each of these limitations we will give our reason as to why it was necessary for this limitation to exist, what the impact might be on existing grammars and what possible future work could do to remove this limitation.

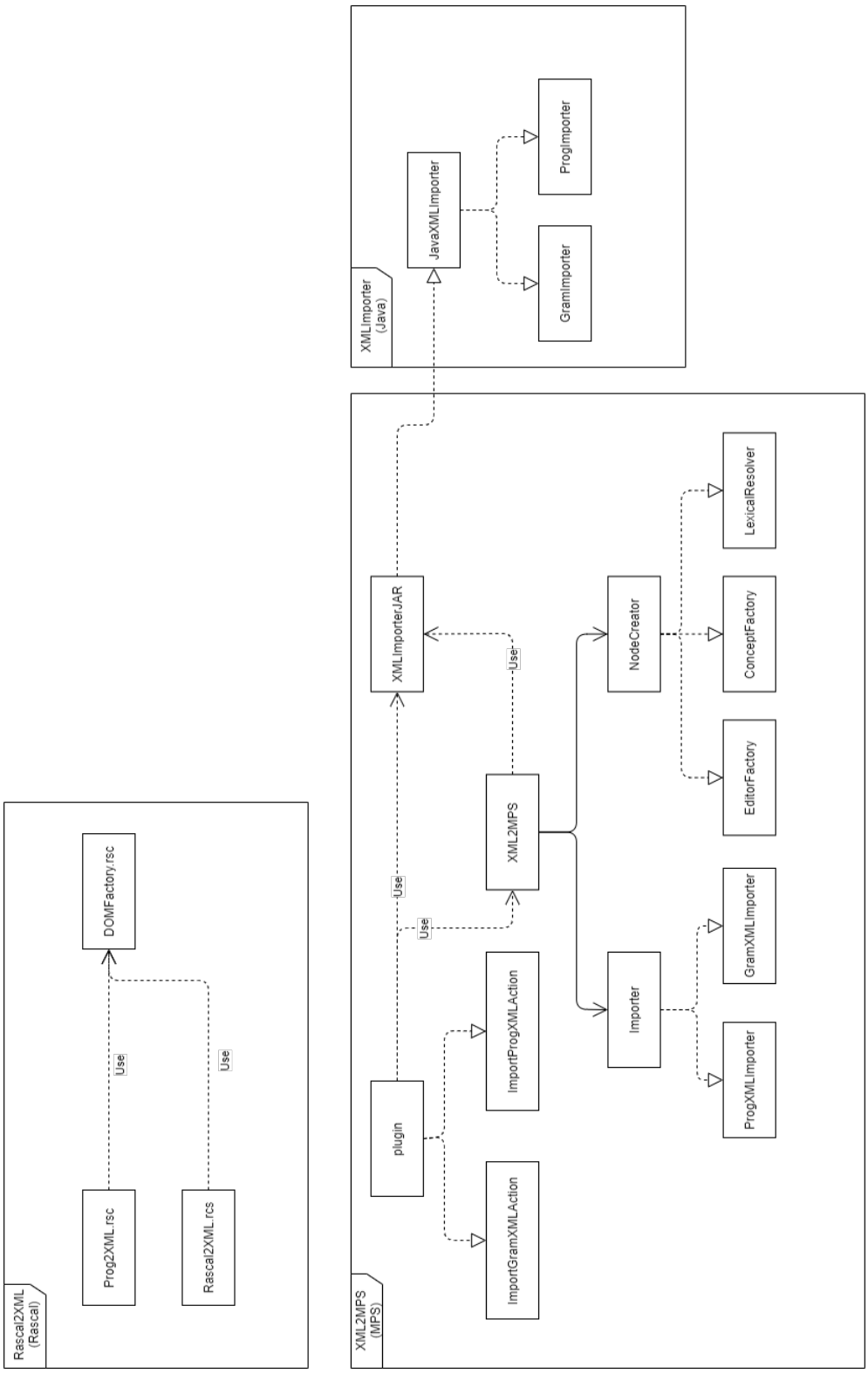

Figure 4.1: Architectural overview

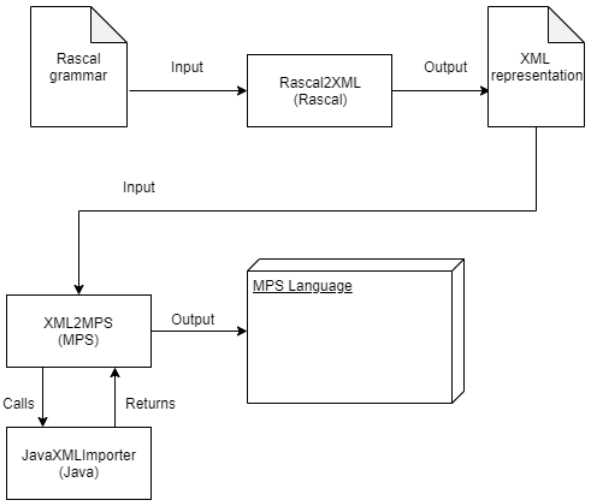

Figure 4.2: Process overview for transferring language definitions

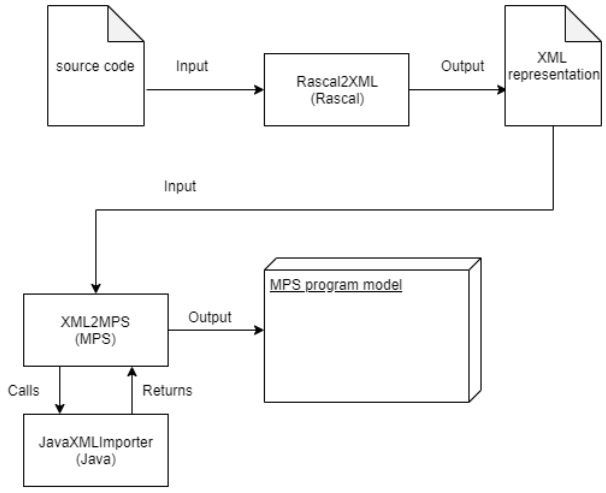

Figure 4.3: Process overview for transferring programs

4.2 XML as intermediary

We will start of by explaining our choice of XML as the file-based intermediary format. When we set out to determine what format to use, we had four main requirements:

1 Strong structural connections, preferably tree-like.

2 Efficient to generate and parse.

3 (Native) support for parsing and generating in both Rascal and MPS.

4 Human readable.

(1) contains the requirement that it should be easy to encode the tree-like structure of the grammar into the intermediary format. Requirement (2) identifies the need for a reasonable performance, i.e. we do not want the reading and writing to be a bottleneck on the import or export process. (3) is mostly a quality of life requirement for developing this project, as we would rather not spend time implementing a custom file reader and writer for parsing and generating the files. Finally, (4) was required to simplify debugging and development.

We considered three candidates for our format: XML [8], JSON [3] and YAML [9]. All three satisfy points 1, 2 and 4. The main difference is in the support on the language workbench level. MPS contains a native implementation of Java called BaseLanguage which contains many of the standard Java libraries. It is also capable of calling Java jars directly as libraries if they are not a part of BaseLanguage. Therefor MPS supports any intermediary format which has a Java implementation, which all three have. However this does not hold for Rascal. While Rascal is also implemented in Java, there is no native support for calling Java libraries. Rascal does natively contain support for XML generation through DOM (Document Oject Model) structures. There is no support for either JSON or YAML. Thus we chose XML, since it fulfilled all the requirements. Of course, using JSON or YAML would still have been possible by writing our own parser and writer tools, but we determined there to be too little benefit to invest time into doing so.

4.3 Lexicals

Before we discuss the implementation of Rascal2MPS, we must discuss one aspect that is circumvented rather than solved by the Rascal2MPS project: Lexicals. Lexicals are a way to limit user input using regular expressions. Listing 4.1 shows an example of some lexicals of the Pico language. When parsing a program, the parser will check whether the input satisfies the regular expression of the lexical and throws a parse error if not. However, the mapping of regular expressions between different language workbenches, which may use different regular expression engines, is not always clear, and outside of the scope of this thesis. Thus instead we apply a workaround which still allows the creation of lexicals in the resulting MPS Language models, at the cost of more user interaction, by making the mapping of the Rascal lexical to MPS Constraint Datatype up to the language engineer. This has effect on both the Rascal and MPS side of the Rascal2MPS project:

Rascal On the Rascal side, we limit lexicals to either be manually mapped, or use a preset name which will map to a default unconstrained MPS string. Any lexical which is not manually mapped on the MPS side and does not have a preset name will not be recognised by Rascal2MPS.

MPS On the MPS side, Lexicals are implemented as Constraint datatypes, which are a MPS construct that can take a string value as long as it satisfies the given regular expression. Which regular expression is applied depends on mapping which must be defined by the language engineer in the Rascal2MPS project.

4.3.1 Example

Lets take the example lexicals in Figure 4.1 and map them as follows: Id will become an unconstrained MPS string while we will manually map Natural and String. The edited lexical defintion is shown in Figure 4.2. We then put the regular expressions for Natural and String in the (hard coded) mapping of Rascal2MPS, as shown in Figure 4.4. Now when we import the full language, Rascal2MPS will use this mapping to generate the correct datatype for the lexicals. Using this workaround makes it possible to have accurate lexicals in the MPS Language model, but it lacks the desired automation. This is part of the future work for this project.

```
1       lexical Id = ([a-z][a-z0-9]* !>> [a-z0-9]) \ PicoKeywords;
2       lexical Natural = [0-9]+ ;
3       lexical String = "\"" ![\"]* "\"";
```

Listing 4.1: Pico lexical definitions

```
1       lexical LexId = lex_id: PrimitiveString;
2       lexical I_Natural = [0-9]+ ;
3       lexical I_String = l_string: "\"" ![\"]* "\"";
```

Listing 4.2: Altered Pico lexical definitions

```
private static HashMap<string, string> regexConstraints = new HashMap<string, string>() {
  {
    put("Natural", "[0-9]+");
    put("String", "\"[^\"]*\"");
  }
};
```

Figure 4.4: Mapping of regular expressions to names

4.4 Rascal2XML

The Rascal2XML component is fully implemented as a Rascal project. It contains the following modules:

- Rascal2XML: Generates the XML representation of Rascal Grammars.
- Prog2XML: Generates the XML representation of programs that can be parsed using a Rascal grammar.
- DOMFactory: Auxiliary module for constructing DOMs.

4.4.1 Rascal2XML

Rascal2XML contains all the functionality for parsing a Rascal grammar and generating a XML representation of said grammar. Essentially, we perform a tree-walk over the concrete syntax tree (CST) which Rascal can generate for any Rascal grammar. The psuedocode for this tree-walk is shown in Algorithm 1.

At the top level we make a distinction between two types of symbols we can find in the CST. First there are the non-terminal symbols: Symbols which contain one or more production rules which they can be rewritten as (see Chapter 2). Secondly, there are Lexical, which in the case of Rascal function as the terminal symbols. These symbols capture user input, and can be restrained

Algorithm 1 Tree-walk over Rascal Grammar

```
1: procedure treeToXML(CST)
2:     DOM ⟵ root
3:     for each symbol in CST do
4:         if symbol == Nonterminal then
5:             Add new NonTerminal Node nonterminal to the DOM root node
6:             for each production p of the Nonterminal do
7:                 Add a production Node production as a child node to nonterminal
8:                 for each nonterminal symbol in p do
9:                     Add a new arg Node as child node to p
10:                end for
11:            end for
12:        end if
13:        if symbol == Lexical then
14:            Add new Lexical Node to DOM root node
15:        end if
16:    end for
17:    mergeDucplicates()
18:    xml ⟵ domToXML(DOM)
19:    write(xml)
20: end procedure
```

using regular expressions. These regular expression constraints pose some complexities which will be discussed in their own section.

For the non-terminal symbol case, we will explore the process using an example from the Pico Rascal grammar. We will use a simple non-terminal "Program" containing only a single production labeled "prog" as shown in Listing 4.3. The resulting XML is shown in Listing 4.4

```
1 start syntax Program
2   = prog: "begin" Declarations decls {Statement ";"}* body "end" ;
```

Listing 4.3: Pico Program production rule

Listing 4.4 shows what information is recorded in the XML file.

4.4.2 Prog2XML

The process of generating an XML representation of source code such that it can be used to create a projectional model is very similar to that of grammars. The main difference lies in the existence of leaf nodes for values. First we apply the built-in Rascal parser with the chosen grammar to the source code to generate a Rascal parse tree. We then recursively traverse this tree depth-first. During this traversal we create nested DOM nodes for each language element we discover. Figure 4.5 shows the traversal order of a partial parse tree of a Pico program containing a single variable declaration. The number in the left corner of each node shows the order in which the element is discovered, while the number on the right indicates the order in which a DOM node for the element is created. Listing 4.5 shows the XML representation of this partial parse tree. This representation preserves both the structure of the original parse tree due to its nesting, and the

```
<nonterminal>
    <name>Program</name>
    <production>
      <name>prog</name>
      <arg>
        <name>decls</name>
        <type>Declarations</type>
        <card>1</card>
      </arg>
      <arg>
        <name>body</name>
        <type>Statement</type>
        <card>*</card>
      </arg>
      <layout>
        <lit>begin</lit>
        <ref>
          <name>decls</name>
          <type>Declarations</type>
        </ref>
        <ref>
          <name>body</name>
          <type>Statement</type>
        </ref>
        <lit>end</lit>
      </layout>
    </production>
  </nonterminal>
```

Listing 4.4: XML representation of non-terminal symbol "program" of the Pico grammar

string values in the leaf nodes. The name attributes refer to the production labels of the grammar, which are required to map back to model elements of the projectional model which was generated from said grammar. Once the whole tree has been traversed, the DOM structure is written to an XML file which can be used as input for the MPS plugin.

```
<node>
      <name>prog</name>
      <node>
        <name>decls</name>
        <node>
          <name>decl</name>
          <leaf>
            <name>lex_id</name>
            <value>ab</value>
          </leaf>
          <node>
            <name>naturalType</name>
          </node>
        </node>
```

Listing 4.5: XML representation of a partial parse tree

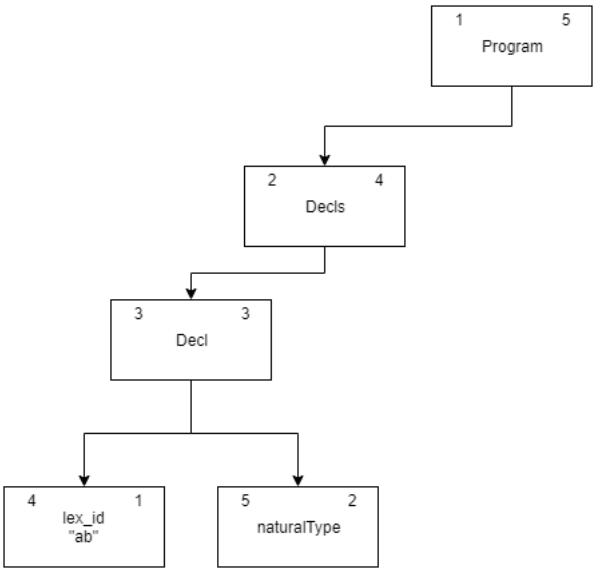

Figure 4.5: Traversal of partial parse tree

4.5 XML2MPS

The XML2MPS MPS project is responsible for generating MPS Language models from XML representations of grammars, and MPS Models of XML representations of programs. These responsibilities are divided between two MPS plugins. There are several helper classes implemented in MPS Baselanguage. There is also a pure Java component for parsing XML and creating an internal representation. The architecture of this project can be seen in Figure 4.1. Again, we will discuss the MPS Language model generation and program model generation separately.

4.5.1 XML to MPS Language model

As explained in Chapter 2, a language model in MPS consists of several *aspects* which represent language features such the structure, type system and visual representation of the model. In this project, we are concerned with two *aspects*: the Structure aspect and the Editor aspect. The Structure aspect defines the meta-model of the language and is the basis of any MPS Language. The Editor defines the visual representation used in the MPS editor, and plays a major part in the usability of the language. First we will discuss how the Structure of the language is generated from the XML representation of the language. Afterwards we will discuss the Editor aspect and how we improve on the reflective editor.

Structure Aspect

At the beginning of the import process we start of where we ended with the export process: with an XML representation of the Rascal grammar, such as can be seen in Listing 4.4. From this representation we need to generate a valid MPS Language model.

First we must determine the mapping of grammar components to MPS Language elements. As described in Chapter 2, an EBNF grammar consists of the following elements:

- A set of non-terminal symbols

- A set of terminal symbols
- A set of production rules.
- A start symbol.

The base element of a MPS language is a Concept. A Concept represents a node in the AST model of the MPS language, and can be given attributes as well as references to further child nodes. MPS also allows for hierarchical abstraction in the form of Interfaces and Abstract concepts, which can be implemented or extended by Concepts. We can now define a mapping based on these elements:

- Any non-terminal symbol of the grammar is mapped to a MPS Interface node
- Any terminal symbol is mapped to a Concept with no child nodes
- Any production rule of a non-terminal symbol is mapped to a MPS Concept implementing the interface of the non-terminal, and symbols which occur in the production rule become child nodes of the Concept.
- The start symbol becomes the root of the AST model.

This mapping ensures that: (i) the relation between the non-terminal symbol left of a production rule and the production rule itself is maintained through the implementation of an Interface, (ii) all components of the grammar are mapped to MPS Language model elements and (iii) the relations between production rules and the symbols that occur with the rule are maintained as child node references.

By maintaining all components as well as all the relations, we can generate a MPS Language with behaviour equivalent to that of the original textual grammar.

Algorithm 2 shows the MPS Language generation procedure. Step (2) simply parses the XML and creates and internal tree representation of Java objects which we can traverse. Some special attention should be paid to steps 9-14 and 18. Arguments here refers to the symbols which occur in the production rule, i.e the "arg" nodes in Listing 4.4. Each argument has a type attribute, which can be either a non-terminal or terminal symbol. However, since we are traversing a tree in the order of elements they are found in, which is determined by the Rascal parser, there is no guarantee that the Concepts for this symbol that is referenced has been created and added to the language yet. Due to how child nodes work in MPS, we can not add a child node for which the type has not been declared as a Concept. There are two options to fixing this issue: Either we try to analyze the tree and attempt to find a schedule such that each symbol has been added to the language as Concept or Interface before it is referenced as an argument, or we delay the linking of child nodes until all Interfaces and concepts have been created, at which point we have a guarantee that all types exist. We chose for the latter, since it is difficult to prove such schedule would even exist that would satisfy the conditions of the former (For example, cyclic references can exist in grammars and would not be able to be scheduled). So, if the type of an argument has not yet been created as Concept or Interface at the moment it is first encountered, the argument and a reference to the Concept it is an argument for is added to a queue. After all non-terminal symbols have been visited, at which point all Concepts and Interfaces have been created, the queue will be executed, meaning that for each pair of argument and Concept, the argument is added to the Concept as child node.

After running the XMLToMPS procedure, we end up with the structural model of our language in the form of an MPS Language model structure aspect. We can then add this populated aspect to the MPS Language itself to obtain a functional MPS Language.

Algorithm 2 MPS Language generation algorithm

```
1: procedure XMLToMPSModel(XML)
2:     tree ⟵ parseXML(XML)
3:     Language ⟵ ∅
4:     queue ⟵ ∅
5:     for each Nonterminal n in tree do
6:         language.addInterFace(n)
7:         for each production p of n do
8:             langauge.addConcept(p, n)
9:             for each argument a in p do
10:                if a.type has been created then
11:                    langauge.getConcept(p).addChildReference(a)
12:                else
13:                    queue.add(a, p)
14:                end if
15:            end for
16:        end for
17:    end for
18:    queue.execute()
19: end procedure
```

Editor aspect

While the previous section dealt with the Structural aspect of a MPS Language, this is not enough for creating a usable language. By default, MPS will generate a reflective editor for any Concept that has no explicitly declared Editor in the Editor aspect of the MPS Language. The reflective editor however only features elements derived from the Concept it is based on, such as the name of the concept, child nodes and attributes. Figure 4.6 shows an example of a program using the reflective editor. There are no literals, all white space is uniform and too much information is presented. The specific concept name for each node is not very useful for the user to be shown at all times, since they can already use the build-in MPS inspector to find the concept of any node at any given time. Showing the names at all times then only adds to the cognitive load. Instead, we would much rather have the visual representation be much closer to the production rule in the grammar. To this end XML2MPS will generate a custom Editor for each Concept in the Structure aspect as created by the previous algorithm.

The first step of generating the Editor for a Concept is gathering the required information. The XML representation of the grammar has a special "layout" node for each production rule which lists all the elements of the grammar, including literals, which are not present in the structure aspect. The elements are in the order they appear in the production rule. An example can be seen in Listing 4.4. We will then recreate this layout in MPS by iterating over the elements and adding literals as constant strings and references as either a child node reference, or a collection of child node references, depending on the cardinality. The resulting Editor is shown in Figure 4.7.

Until this point, we have all the elements of the layout, but they are flattened on a single line. To better match the look and feel of the textual language workbench, we must introduce whitespace to our editors. However, whitespace is not registered directly in the AST. Thus we must turn to heuristics to discover where to add whitespace.

Indentation is added based on the heuristics for pretty printing as described by Van den brand et al [23]. The authors identify the a classification of production rules based on the structure of the rule. A production rule is classified as:

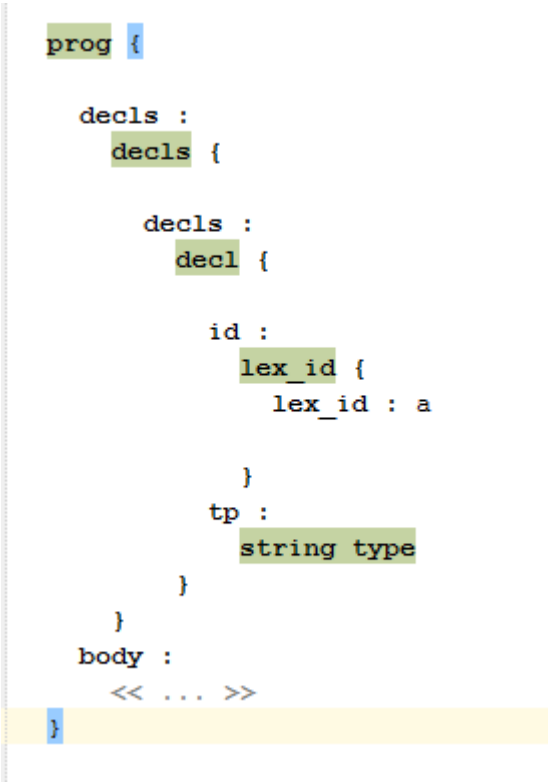

Figure 4.6: Example of a Pico program using the relective editor in MPS

- Indented if the rule follows the pattern t N (t N)* t or t N (t N)+
- Nonindented otherwise

Where t denotes a terminal symbol of the grammar and N denotes a non-terminal symbol.

To illustrate the heuristics, we will apply it to the example of an "if then else" statement, which is a common construct in programming languages. Listing 4.6 shows the Pico definition of the "if then else" statement. We then attempt to match this production rule to our classifiers: "if" (t) Expression (N) "then" (t) Statement (N) "else" (t) Statement (N) "fi". The pattern is thus t N t N t N t, which matches the pattern t N (t N)* t. Therefor this is an example of an indented rule, and indentation and whitespace is added to the editor concept, as shown in Figure 4.8. Alternatively, we can apply the heuristics to the "add" production rule shown in Listing 4.7. The pattern of this rule is Expression (N) "+" (t) Expression (N). This pattern does not match either of the indentation patterns, and thus this production rule is not indented. The resulting editor is shown in Figure 4.9.

```
1  ifElseStat: "if" Expression cond "then" {Statement ";"}* thenPart "else" {Statement ";"}*
       elsePart "fi"
```

Listing 4.6: Pico "if" statement production rule

```
1  Expression lhs "+" Expression rhs
```

Listing 4.7: Pico "add" statement production rule

```
<default> editor for concept prog
  node cell layout:
    [-
    begin
      % decls /empty cell: % (- % body % /empty cell: -)
    end
    -]

  inspected cell layout:
    <choose cell model>
```

Figure 4.7: Example of a custom Editor in MPS

```
<default> editor for concept ifElseStat
  node cell layout:
    [-
    if
      % cond /empty cell: %
    then
      (- % thenPart % /empty cell: -)
    else
      (- % elsePart % /empty cell: -)
    fi
    -]
```

Figure 4.8: If then else MPS editor

Code-completion menu

MPS heavily leverages the contextual code-completion menu. The code-completion menu shows the user which element can be inserted at the current location. Thus it is very important to make sure that the menu is informative, such that the user is able to make the right choice between the available language elements to insert. MPS offers several ways to improve the user experience of the code-completion menu, such as custom filtering of to-be-shown elements based on the current context. However this requires either domain knowledge of the language, which we can not assume, or good heuristics which can be applied in the general case, which is beyond the scope of this thesis. We do improve the code-completion menu user experience in three more straightforward ways:

1 Prune the AST of any nodes with a single parent node and single child node. This pruning process as described in Chapter 3 also simplifies the code-completion menu.

2 Set aliases for all concepts to the first literal in the production rule. While simplistic, this heuristic seems to often work well in our experience, especially for binary operators where the first literal is usually the operator itself. Setting the alias in this way also allows us to type the operator directly instead of using the full concept name.

3 Set the short description of all concepts describe the production rule. This is especially helpful to differentiate between rules with similar names but different implementations.

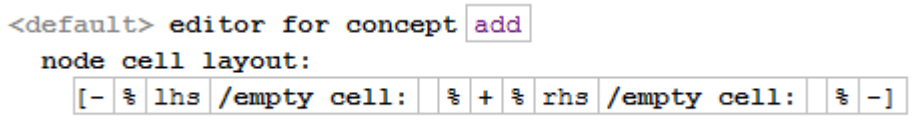

Figure 4.9: Add MPS editor

4.5.2 Program XML to MPS model

Given that the required MPS Language model is available, we can now generate a MPS model from the XML representation of source code of said language. We previously described how the XML representation was generated and what it looks like. The XML representation of the parse tree is a tree structure itself, where nodes refer to language components and leaves contain string values for lexicals. To generate a valid MPS model, we will once again traverse the XML representation recursively depth-first, creating MPS model elements along the way. Here we can use the relation between the grammar production labels in the parse tree, which are use for the name attribute of the XML nodes, and the names of the MPS concept, which follow directly follow from the production rule in the grammar. Thus, the name attribute of a node in the XML representation directly maps to a MPS concept. The MPS API is used to search for these concepts and create instances of them. The recursive depth-first traversal ensures that for any parent node, all child nodes have been created before the parent itself, so the child nodes can be correctly linked.

4.6 Limitations on the source grammar

There are several constraints that are placed on source Rascal grammars. Exporting a grammar that does not satisfy these constraints, or importing an XML file that was generated from a grammar that does not satisfy the constraints may lead to errors or invalid MPS models. We will first list the constraints, and then go into detail for each specific one.

1 Non-terminal names must be unique within the grammar.

2 Production rules labels must be unique within the grammar.

3 Symbol labels names must be unique within the production rule.

4 Lexicals must either be one of the predefined MPS primitives or must be declared in the Lexical mapping.

5 Each production rule must be labeled.

6 Each symbol within a production rule must be labeled.

Constraints (1) and (2) have the same cause: The name is used as the name for either an interface (non-terminal) or concept (production rule label) within the generated MPS Language. MPS simply requires that these names are unique. Therefor the names in the grammar must be unique. Alternatively, this could be solved using some renaming scheme to catch and resolve naming conflicts before the model is generated. However, here we must keep in mind the effect this can have on the usability of the generated MPS Language. Since the editor also uses the name for the auto complete feature, names must be descriptive to some extend. Straightforward naming schemes such as appending an incremental number can lead to confusing menus as shown in Figure 4.10. Ultimately we decided against implementing such solutions for this thesis. Constraint (3) is also related, since the symbol labels within the production rule is used as variable name for the child reference of Concept in MPS. These must also be unique within the concept, but not for the whole model.In MPS Concept A and Concept B can both have a child reference named

Figure 4.10: Example of the autocomplete menu using a renaming scheme

"c", but A can not have multiple child references named "c". Thus symbol label names must be unique within the production rules, but can be reused between different production rules.

Constraint (4) follows from how lexicals are handled in translation between Rascal and MPS, see the section on lexicals. Since the regular expression must be mapped, the lexical itself must be in the mapping.

Constraints (5) and (6) are due to these labels being used by Rascal2XML as the names for either Concepts or child reference names. Since we must supply some name, the alternative would have been to generate some dummy name if none was given. However this quickly created more problems than it solved. Besides the issue of descriptive names as previously stated for constraints (1) and (2), it runs into problems for matching attributes when importing programs.

4.7 Version control

Rascal2MPS does not currently implement any kind of version control over the versions of grammars and/or programs that are being imported from the textual LWB. This can create issues if different versions of programs and grammars are mixed. For example, take textual language A_t which has been imported into a projectional LWB to create language A_p. Now A_t is changed in some way and becomes language $A2_t$. A program $A2_{prog}$ is written in $A2_t$. Now the language engineer, forgetting that the language imported in the projectional editor was an older version of said language, tries to import the source code as a program of language A_p. What happens during the import depends on the changes made to the grammar:

- Removed language elements can by definition not occur in the AST of programs, so nothing will happen and the program should import correctly.
- If language elements were added to the grammar and used in the program, XML2MPS will not be able to match these elements to concepts, and notify the user.
- If language elements were altered, such as adding or removing parameters, the import process may crash due to out-of-bound errors on certain internal datastructures.

This can also happen the other way around: if a grammar is imported into a projectional editor and the resulting projectional language is somehow changed, importing source code written using the original textual grammar might lead to incorrect program models.
Overall care should be taken to only import programs written in the same version of a grammar as was imported into the projectional language workbench. A more strict automatic version system is future work.

Chapter 5

Evaluation

In this Chapter we will evaluate the presented approach and the implementation of Rascal2MPS. We will revisit the problem statement and sub-problems as defined in Chapter 1 and identify whether the project goals were met in the implementation and whether there are any shortcomings to the current implementation of Rascal2MPS that limit either the functionality or the overall usability of the tool in practice.

Originally, we identified three goals that should be met by the tool: (i) The tool should be able to map a textually defined grammar to a projectional model. (ii) The tools should improve the usability of said model in the context of the projectional world. (iii) The tool should be able to map textually defined programs to projectional models, given that the language in which the program was defined is available. Each of these goals is met by the Rascal2MPS implementation.

5.1 Mapping a grammar structure to a projectional model

Rascal2MPS is able to generate a projectional model in the form of an MPS Language from any given valid Rascal grammar, as long as the grammar satisfies the constraints given in Chapter 4. Some of the constraints are the results of different design choices between the respective workbenches, such as the need for unique identifiers in MPS for Concepts and Interfaces. Others, such as the somewhat cumbersome manual lexical mapping are related to the scope of this project, where there are possible solutions that go beyond what we are attempting in this thesis. None of the constraints limit the expressiveness of the grammar. In all cases except that of lexicals it only requires changing identifiers or adding some information in the form of labels. One can certainly imagine a pre-proccessor being used on the grammar to automate some of these steps further.

One stated metric for success of the tool was the ease of use and efficiency for the language engineer. to this end, the two-step approach is initially somewhat counter-intuitive as a one-step process such as implemented in the Ingrid [25] project, where the grammar parser is linked directly to the MPS plugin, requires less user interaction and us defined in one environment only. However, we felt that the advantages created by explicitly decoupling the source and target workbench are worth the additional step in the process.

5.2 Improving the usability of the projectional model

Improving the usability of the MPS language model generated from the source Rascal grammar comes in two forms: Changing elements of the structure aspect to better interact with the model-based environment, or generation of MPS Editors for the language concepts through heuristics. The challenge in editor generation is based on having to do so with no prior domain knowledge of the language. The end result of the implemented editor generator is certainly not meant to work

for any given language out of the box; but more so to make a first attempt and allow the language engineer to adjust as necessary. Overall, the improvements to the editor as shown in Chapter 4 combined with the AST pruning, Aliases and short descriptions make for a much improved editor experience when compared to the usability of the default model generated from the grammar.

5.3 Mapping textually defined programs to projectional models

Rascal2MPS is able to generate MPS models from any valid textual source code given that the language it was written for is made available, either by creating the MPS Language manually, or importing a grammar. The only limitation to this process is that the grammar that was used to parse the program on the Rascal side must be unambiguous in order to guarantee a correct model on the MPS side, as discussed in Chapter 4. Programs parsed using an ambiguous grammar that happen to contain some ambiguity may lead to incorrect models, since Rascal2MPS will simply pick the first option presented by the parser, which may be incorrect. However, in practice we find that even if there are some incorrect elements in the model due to ambiguity, replacing these manually is often still much faster then manually recreating the whole program.

5.4 Javascript language

As stated before, the challenge of generating arbitrary language models lies in the lack of domain-specific knowledge, which is often critical when creating usable language models. In order to both show the application of Rascal2MPS to a real-world example of a widely used general purpose language and the usability of the generated language, we imported a Rascal implementation of JavaScript using Rascal2MPS. The grammar was changed to satisfy the constraints on the source grammar, which in the case of the Rascal implementation most meant adding labels for production and variable and changing the lexicals to the default MPS string. The grammar was then successfully exported to an XML representation, which in turn was used to generate an MPS Language model. To evaluate the usability of this generated MPS Language model, we will compare it to a hand-made MPS implementation for JavaScript in ecmascript4mp [2]. This implementation was made with full domain knowledge of the JavaScript Language. For the comparisons we will use several examples of MPS Language elements and models created using both MPS Languages. To avoid confusion, we will refer to the MPS Language model generated by Rascal2MPS as RascalJS and the manually implemented MPS Language model as NativeJS.

We will compare RascalJS and NativeJS on two points: The visual representation in the Editor aspects and the overall usability of the Language model.

5.4.1 Editor aspect

To compare the Editors of both versions of Javascript implemented in MPS, we will compare how the same program is visualised in each respective editor. The RascalJS version was generated using the Rascal2MPS program import functionality, where the source JS program was exported to XML and imported as MPS model. The MPS Language model used for this model was RascalJS, i.e the previously imported Rascal implementation of the Javascript grammar. The Language model was unchanged, so no manual editing of the model or the Editor definitions was done. The resulting program can be seen in Figure 5.1 and is continued in Figure 5.2 due to the size. The second model, which was created manually using the NativeJS Language model is shown in Figure 5.3.

The first notable difference is the overall length of the resulting programs: The program generated using RascalJS is much longer than its counterpart in NativeJS. This is mostly due to

the application of whitespace, which breaks up multiple parts due to the applied heuristics. For example, the loop initializer, condition and update are split among several lines, as opposed to the NativeJS version where they are on the same line, although variables have to be declared outside of the loop due to structural issues with variable identifiers, more on that in the next section. There is also a difference in dot(.) expressions: RascalJS will place spaces before and after the dot operator, since it identifies it as any other binary operator such as + or -, for which a space before and after the operator is desirable. This is a clear case where the default heuristics are too general, and an exception for dot expressions could be added such that this does not happen on future imports, or the Editor definition can be manually changed to remove the spaces.

Overall, the manually created Editor definitions of NativeJS lead to visually more pleasing programs then those generated by the Heuristics of Rascasl2MPS. This is to be expected, as domain knowledge is able to applied in the first case, while the heuristics are limited to only the knowledge and context we gain from the concrete syntax tree. However, to create the elaborate Editors of NativeJS from scratch is a time consuming process. By being able to generate a default Editor definition using Rascal2MPS, and manually refining the Editors afterwards, the language engineer can save valuable time.

5.4.2 Usability

In this section we discuss the usability of the MPS Language models of RascalJS and NativeJS within the MPS editor. The usability under discussion here refers to the ease of creating and editing program models, i.e. models which use a Language model as meta-model. The first important usability feature to discuss is the code completion menu , which is a central point of editing any MPS program. We can compare the code-completion menus for creating a for loop in both RascalJS in Figure 5.4 and NativeJS in Figure 5.5. Both menus show similar information: the name of the concept and a short description. The RascalJS version has slightly more information by also including information about child nodes in the short description. Overall the quality of the code completion menu of RascalJS is as good as that of NativeJS.

```
function
  substrings
(
  str1
)
{ var
    array1 = []
  ;
  for
  ( var
    x = 0
    y = 1
  ;
    x < str1 . length
  ;
    x ++
    y ++
  )
    { array1 [ x ]
      = str1 . substring ( x
        y
      )
      ;
    }
  var
    combi = []
  ;
  var
    temp = ""
  ;
  var
    slent = Math . pow ( 2
      array1 . length
    )
  ;
```

Figure 5.1: MPS Substring program part a

```
for
( var
  i = 0
;
  i < slent
;
  i ++
)
  { temp = "" ;
    for
    ( var
      j = 0
    ;
      j < array1 . length
    ;
      j ++
    )
      { if
        ( ( i & Math . pow(2,j)
          )
        )
          { temp += array1 [ j ]
            ;
          }
      }
    if
    ( temp !== ""
    )
      { combi . push ( temp
        )
        ;
      }
  }
console . log ( combi . join("\n")
)
;
}
```

Figure 5.2: MPS Substring program part b

```
program substring
--------------------------------------------------------------------------
function substring(str1) {
  var array1 = [],
      x = 0,
      y = 1;
  for (; x < str1.length; x++) {
    array1[x] = str1.substring(x, y);
  }
  var combi = [];
  var temp = '';
  var slent = 'Math.pow(2, array1.length)';
  var i = 0;
  for (; i < slent; i++) {
    var temp = '',
        j = 0;
    for (; j < array1.length; j++) {
      if (i & 'Math.pow(2,j)')
        {
          temp += array1[j];
        }
    }
    if (temp !== '')
      {
        combi.push(temp);
      }
  }
  'console.log(combi.join("\n"))';

}
```

Figure 5.3: Substring program model with ecma4MPS

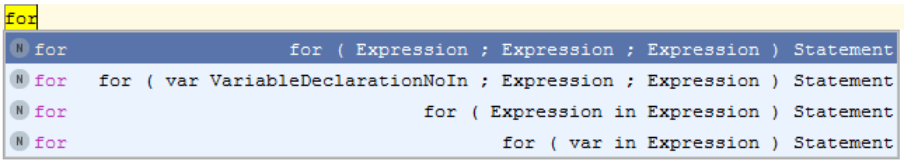

Figure 5.4: Code completion menu of RascalJS

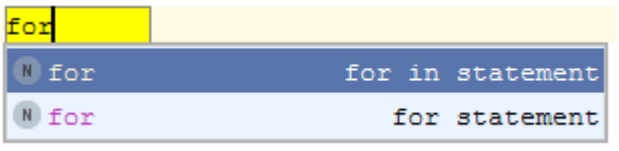

Figure 5.5: Code completion menu of NativeJS

Chapter 6

Related work

6.1 Grammar to model

A Major component of this thesis is the generation of models from textual grammars. In order to place ourselves within the current state of the art we identify the following related work.

The goal of the Ingrid [25] project quite similar ours: bridging the gap between the textual and projectional language workbenches. In their case, they chose the same projectional language workbench, Jetbrains MPS, but a different textual meta-language in ANTLRv4[1]. Ingrid has an implementation of their solution as a hybrid Java/MPS project.

Ingrid approaches the problem in three steps: First, the grammar must be parsed and relevant information about the structure and other required language elements is stored as linked Java objects. Secondly, the stored structure is traversed and equivalent MPS model nodes and interfaces are constructed. Finally, an editor is generated for each MPS Language Concept Node.

There are many high-level similarities between Ingrid and Rascal2MPS. The steps taken for parsing, gathering information about the language, generating an intermediary structure to represent the language, and finally generating a model from said intermediary structure are performed by both projects. The main differences are in the details, and the choices made in the implementations of both projects, which have various consequences for the use of the respective tools.

The main architectural difference is in the choice of intermediary structure. Whereas we chose for an external file-based format (see chapter 3), Ingrid uses an internal representation of linked Java objects. This is enabled by the use of the an ANTLRv4 parser implemented in Java and the ability of MPS to directly call into Java executables. Thus, the Ingrid MPS plugin is able to call the parser and start the process of data extraction internally. This is in contrast to Rascal2MPS, where the projectional language workbench and textual language workbench are kept completely separate, only communicating through an external intermediary format. There are several advantages to the Ingrid's integrated approach:

- The solution becomes a one-step process, making it more efficient for the language engineer.
- All implementation is done on one side of the gap (The projectional language workbench). This simplifies development.
- The language engineer does not need to maintain both the textual and projectional language workbench.

However, there is also a major downside to this approach which is why we ultimately decided against this it. Since the projectional language workbench has to call the grammar parser directly, there is a strong binding between the projectional language workbench and the specific grammar parser. In the case of Ingrid, the MPS plugin calls into the Java ANTLR parser. However, ANTLR is far from the only textual language workbench. If we now wished to extend Ingrid for a new textual language workbench, such as Rascal, we would need to replace the calls to the

ANTLR parser with calls to a Rascal parser. This can lead to several problems:(i) The architecture has to allow this replacement. This can be partially solved with smart use of interfaces and abstraction over the parser, but the problem of potentially different APIs remains. In the worst case, a complete mapping from ANTLR parser function calls to Rascal parser function calls would have to be made. (ii) The parser needs to be implemented in Java. ANTLRv4 already has a Java-based parser available, and thus is a prime candidate for integration with the also Java-based MPS. However, this is not necessarily true for any given textual LWB. If one is not available, the language engineer would either have to create one in Java, or find some way to expose the parser of specific LWB to a Java environment.

Part of the original problem statement for Rascal2MPS was to create a more general approach to the problem of bridging the gap between the textual and projectional worlds. This also extends to the implemented tool. Neither side of the solution is aware of the other; only of the intermediary file-based format. This format server as a contract between the different parts of the solution. Thus whether the intermediary file was generated by from an ANTLR-, Rascal- or Xtext-based grammar is irrelevant to the implementation on the side of the projectional language workbench.

Another difference between Ingrid and Rascal2MPS lies in the Editor generation within MPS. While Ingrid does identify the problem of usability of the reflective editor and discusses several solutions, such as heuristics or prompts during the import process, they are not implemented. Ingrid only generates an editor containing the structural elements of the node, i.e. the literals and references to other nodes. It is then left up to the language engineer to manually apply whitespace to the editor. Rascal2MPS goes further and applies heuristics to automatically apply whitespace during the import process. While this does not fully eliminate the need to manually edit the editor definitions (See Chapter 5), it can save time given the right set of heuristics.

Finally, Ingrid does not address the problem of language artifacts, i.e. programs created within the textual world. Thus even after a language has been imported, programs written using said language in the textual language workbench need to be manually recreated as MPS models of the imported language. Rascal2MPS does implement the ability to construct MPS program models using textual source code.

Wimmer et al. [26] describe a generic semi-automatic approach for bridging the technical space between Extended Backus–Naur form (EBNF), a popular grammar formalism, and Meta-Object Facility (MOF), a standard for model driven-engineering. In this approach an attributed grammar, which describes EBNF and the mapping between EBNF and MOF, is used to generate a Grammar Parser(GP). This GP can then be used to generate MOF meta-models from grammars. This is not far from the approach implemented by this thesis. However, this approach fixates on MOF as the target meta-model directly. In the case of going between language workbenches in separate worlds, we do not want to be specific in the target. Instead, out approach focuses on the contract in the form of the intermediary format, and makes the source and target formalism up to implementation. Another downside of the given approach is that it requires grammar annotations and additional manual improvements of the generated model in order to refine the generated model. We seek to limit the actions of the language engineer, especially concerning the source grammar.

The Gra2Mol [17] is another project which seeks to bridge the gap between the textual grammar and model worlds. The authors define a domain-specific model transformation language which can be applied to some source program which conforms to a grammar, and generates a model which conforms to a target meta-model. This language can be used to write a transformation definition consisting of transformation rules. In this way, the presented approach abstracts over the generated meta-model, which would be quite useful in our use-case, as we would be able to simply give the meta-model of the target language workbench as input with the transformation definition. In practice however, this runs into problems when the desired target model is specific

rather than generic. For example, the standard storage format for Jetbrains MPS is a custom XML format. The models contain much information tied specifically to MPS, such as the node ID's, layout structures etc. Generating these from outside of MPS would be quite tedious, and also would introduce a dependency on the MPS model format, which may change in the future. It is thus best to interact with MPS model from within MPS itself, where MPS can do the heavy lifting of generating the models.

6.2 Editor generation

Another aspect of the project is the generation of the Editor Aspect, or the visual representation of language model elements. This is closely related to the pretty printing problem.

Van de Vanter et al [12] identifies part of the core problem between the textual and model-based approach. From a system's perspective, a model-based editor allows for easier tool integration and additional functionality. However, language users are often more familiar and comfortable with text-based editing. In this paper the authors propose a compromise based on lexical tokens and fuzzy parsing. This is not unlike what is offered by MPS. MPS Editors are highly customizable and can be made to closely resemble the text-based editing experience.

The BOX language for formatting text as introduced by Van den Brand et al [23] is closely related, as the heuristics for generation white space between language elements is reused in this project. The BOX language is further used in other work on pretty printing of generic programming languages, such as GPP (Generic Pretty Printer) [11], which constructs tree structures of the layout of a language element, which can then be consumed by an arbitrary consumer.

Syntax-directed pretty printing [22] also identifies several structures for creating language-independent pretty printers. In this approach a grammar extended with special pretty printer commands is used as input to generate a pretty printer of for that specific language. The generated pretty printer can then be reused for any program written in the language the pretty printer was generated for. The annotated grammar approach does limit the form the final pretty printer can have due to the lack of options. Also, annotating an entire grammar can be tedious work. In our approach, we attempt to limit the required user interaction with the source grammar, although we did not eliminate it entirely.

Chapter 7

Future work

While the presented approach and the implementation of Rascal2MPS provide a solid tool for generating usable projectional MPS Language models from textual Rascal grammars and MPS program models from textual source code, there are many improvements that could still be made. The most work still to be done is in the usability aspect of the Rascal2MPS implementation. The comparison in Chapter 5 between a MPS Language model generated from a Rascal grammar and an MPS Language model created by hand has shown that while an improvement over the reflective editor generated by MPS for most c-like languages, the Editor aspect generated by Rascal2MPS still is not as good as one created with domain knowledge. There are several avenues open to improve the generated MPS Editors:

The current set of heuristics are meant to catch certain very broadly defined cases. However, more heuristics could be identified and applied in order to generate better Editors. For example, from the evaluation with the Javascript program, we identified brackets as a case where the heuristics could be improved.

Applying machine learning to "learn" code styles from a given set of source code files of the grammar. This approach was implemented by an extension of Ingrid, although this was not part of an academic work and the results are unclear. the approach however seems sound: Use machine learning to identify patterns in the use of language elements in the source code, and recreate these patterns as MPS Editors for said language elements. Of course, as with most machine learning applications, care must be taken take the input bias into account. For optimal results high quality source code must be used, but what exactly constitutes high quality is often up for debate. A potential approach could be to let heuristics define a baseline Editor and use machine learning to improve upon these where possible.

Another potential improvement to the implementation would be use use of more persistent styling options for the Editor. Currently, there is no way to save manual changes to the MPS Language model outside of MPS. This is an issue if the grammar your MPS Language model was generated from is not yet stable and could change over time. Of course, a change in the grammar might require the grammar to be imported again to match the MPS Language model. However, any changes to the MPS language model that were done after the language was imported would need to be repeated on the new model. One way of preventing this would be to save changes in some form, such as a style sheet, and have an option to reapply them once a grammar is imported.

A more general future work in the process of creating a bridge between the textual and projectional LWBs is that of implementing the other side of the bridge we implemented in this work, namely the mapping of the projectional to textual LWBs. This would mean generating textual grammars of some textual LWB formalism given a projectional language definition. This would enable grammars to be imported in a projectional workbench as a projectional model, have some

operations applied to the model, and then regenerating the grammar from said model for further use in the textual LWB.

Chapter 8

Conclusions

In this thesis we have identified the problem and use cases of bridging textual and projectional LWBs. Textual language workbenches are mature and familiar to language engineers, as they were one of the first approaches to software language engineering. Projectional language engineering offers new possibilities such structural editing and integration of model-based techniques. However textual languages are not compatible with the projectional approach out of the box, and recreating them manually in the projectional environment is a cumbersome process. Our solution presented in this thesis offers a general approach for bridging the gap from textual to projectional language workbenches, such that languages defined in a textual language workbench can be semi-automatically mapped to projectional language definitions usable by a projectional language workbench. We implemented this solution on one source language workbench in Rascal, and one target projectional language workbench in Jetbrains MPS. The implementation was evaluated on the different sub-goals we set out for such a tool: It should be able to generate projectional models from a textually defined grammar, it should improve the usability of the generated language model and it should be able to generate projectional models from textual source code if the language of the given source code is available. All three of these goals were met by the Rascal2MPS implementation. Based on the results of the implementation, we also identified future work to improve upon the current solution and implementation of Rascal2MPS, such as removing limitations on the source grammar, further improvements to the usability of the MPS editors and the ability to generate textual grammars from projectional models.

Bibliography

[1] ANTLR. https://www.antlr.org/. 38

[2] ecmascript4mps. https://github.com/mar9000/ecmascript4mps. 33

[3] JSON. https://www.json.org/. 21

[4] Mps custom persistance. https://www.jetbrains.com/help/mps/custom-persistence-cookbook.html. 10

[5] Mps editor documentation. https://www.jetbrains.com/help/mps/editor.html. 7

[6] Mps editor menu documentation. https://www.jetbrains.com/help/mps/commanding-the-editor.html. 10

[7] Rascal2mps code repository. https://github.com/JurBartels/Rascal2MPS. 18

[8] XML. https://www.w3.org/XML/. 21

[9] Yaml. https://yaml.org/. 21

[10] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016. 1

[11] Merijn De Jonge. Pretty-printing for software reengineering. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 550–559. IEEE, 2002. 40

[12] Michael L. Van de Vanter, Marat Boshernitsan, and San Antonio Avenue. Displaying and editing source code in software engineering environments. 2000. 40

[13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24 – 47, 2015. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014). 4, 10

[14] Martin Fowler. Language workbenches: The killer-app for domain specific languages. 2005. 1, 4

[15] Maurizio Gabbrielli and Simone Martini. *How to Describe a Programming Language*, pages 27–55. Springer London, London, 2010. 4

[16] Seymour Ginsburg. *The Mathematical Theory of Context Free Languages.[Mit Fig.]*. McGraw-Hill Book Company, 1966. 5

[17] Javier Luis Cánovas Izquierdo, Jesús Sánchez Cuadrado, and Jesús Garcıa Molina. Gra2mol: A domain specific transformation language for bridging grammarware to modelware in software modernization. In *Workshop on Model-Driven Software Evolution*, pages 1–8, 2008. 39

[18] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. pages 168–177, 01 2009. 1

[19] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177. IEEE, 2009. 5

[20] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. Frame-based editing: Easing the transition from blocks to text-based programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education*, WiPSCE '15, pages 29–38, New York, NY, USA, 2015. ACM. 1

[21] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013. 1

[22] Lisa F. Rubin. Syntax-directed pretty printing—a first step towards a syntax-directed editor. *IEEE Transactions on Software Engineering*, (2):119–127, 1983. 1, 40

[23] Mark Van Den Brand and Eelco Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):1–41, 1996. 27, 40

[24] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000. 1

[25] Premysl Vysokỳ, Pavel Parízek, and Václav Pech. Ingrid: Creating languages in mps from antlr grammars. 2018. 32, 38

[26] Manuel Wimmer and Gerhard Kramler. Bridging grammarware and modelware. In *International Conference on Model Driven Engineering Languages and Systems*, pages 159–168. Springer, 2005. 39