FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Přemysl Vysoký

Grammar to JetBrains MPS Convertor

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	RNDr. Pavel Parízek, Ph.D.
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2016

Title: Grammar to JetBrains MPS Convertor

Author: Bc. Přemysl Vysoký

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: JetBrains MPS is a language workbench focusing on domain-specific languages. Unlike many other language workbenches and IDEs, it uses a projectional editor for code. The developer directly manipulates the program in its tree form (AST) and not by editing a text source code. This brings many advantages, but on the other hand requires time-consuming and complicated MPS language definition. The thesis elaborates on the possibility of automating the process of creating MPS language definition from its grammar description. It introduces the MPS editor, evaluates approaches of related projects and describes author's efforts to implement an MPS plugin that allows this import. The chosen approach and the selection of tools used for implementation are justified in the thesis. We point out important problems that any similar project might deal with and we introduce some possible solutions. Furthermore, the thesis contains examples of imported languages, showing the potency of the chosen approach. The thesis also aims to lay groundwork for future extensions and suggest possible improvements.

Keywords: JetBrains MPS, grammar, convertor, import, programming language, projectional editor

I would like to thank my supervisor Pavel Parízek for his advices, observations and scrutiny with the text of the thesis. Further I would like to thank Václav Pech and Alex Shatalin, developers of MPS, for their overall support during the work on the thesis and the plugin development.

Contents

1. Introduction

As software development becomes more and more important discipline, it evolves every year dramatically. The languages and the tools that we use for writing code also become more and more advanced and full of features. They help us deliver better software and write nicer code much faster than before.

One of these tools is the JetBrains MPS [1], which approaches languages from a different point of view. MPS doesn't use the textual representation of code as usual, but rather works with the actual abstract syntax tree that holds the code's structure. This has some positive implications as it gives us a very powerful tool rich on features but also introduces some new problems we haven't encountered before. One of them being the need for a special definition of languages created inside MPS so that the IDE can understand them and work with them.

The MPS editor offers a lot of flexibility and possibilities when it comes to designing custom languages. It also becomes a very powerful tool once the language has already been created and you start using it inside MPS. There is a huge variety of new languages constructed for MPS that usually solve domain specific problems. However, building these languages is a complicated and time-consuming process and a lot of effort must be put into them before they are ready for use. This thesis will look into the possibility of automatic import of already existing general purpose languages into MPS using a grammar description of their syntax.

Usually, languages created in MPS are tied to a very narrow domain and solve very specific problems. They are mostly small extensions of existing languages or some stub languages used for educational purposes. It would, however, be very nice if we could leverage all the features that MPS has to offer together with general purpose languages we know from the outside world. We are talking about languages such as C++, JavaScript or Python. If there would a possibility to code in these languages using the projectional editor, programming could, for example, open its doors to many people, who want to take it up, but are rather put off by its complexity.

General purpose languages are usually more complex when it comes to their structure and the overall syntax variety than the usual DSL extensions that are created using MPS. Currently, there exists an almost full port of the Java language called BaseLanguage [2] extended with some MPS specific features. It was imported manually by JetBrains and it is still undergoing changes as Java itself is evolving. There are other attempts where for example the C language is also manually tailored for MPS within the mbeddr project [3]. These examples show that recreating a full language inside MPS is not an easy task and a lot of time must be spent on implementing all aspects of the language. A big part of this effort is, however, quite straightforward and could be possibly automated, which would speed up the process of adoption of new languages.

1.1 Main Goals

The main goal of this thesis is to explore the possibility of automatic import of already existing languages into MPS from the grammar description of their syntax. One of the results of this effort will be an MPS plugin allowing users to carry out this import.

It is certainly not expected that these imported languages will be ready-to-use full-fledged MPS languages as that is a much bigger, if not impossible, challenge. The thesis would rather explore the problem, suggest a possible solution and make first steps in this unexplored area, possibly preparing ground for further follow-up work. Nonetheless, the plugin should do the heavy lifting so that the imported language contains full structure found in the grammar and some more aspect definitions that will help users of this language in creating code. It is expected that the imported language will be adjusted by the end user and some of its complication will be resolved manually by a human.

There will be complications along the way that will arise once we will dive deeper into the problem and we will pay more attention to them. Not all of these problems have an optimal solution, and therefore it is up to the author to choose and defend a path that will support our cause in the best way possible.

Once we will introduce the MPS editor more closely, we will revisit the goals of the thesis in more detail.

1.2 Thesis Overview

In the beginning of the thesis, we will describe the MPS editor. We will briefly introduce fundamental basics of the target environment, as it is needed for an understanding of what we are trying to achieve.

We will continue with research of different existing grammar notations and choose the most suitable one for our cause. We will analyze existing related projects that are trying to accomplish similar goals, and the author will weigh advantages and disadvantages of these approaches. The author will then consider, if this thesis will follow up on these efforts or whether an entirely different path will be taken.

After the initial analysis, the author will describe an MPS plugin that will allow the user to import a language into MPS. The author will talk about all phases of the import process and walk the reader through them, describing encountered obstacles using an example language.

Next, we will talk about the implementation of the plugin itself. We will describe its architecture and talk about some problems connected to the process of implementation. We will also show how to install and use the plugin and also show some example languages imported with it.

In the last part of the thesis, we will look at some problems that grammar import might pose. We will talk about these problems with respect to the obstacles we tried to overcome and which we described in the middle part of the thesis that is devoted to the import process. We will also look into possible follow-up work that might build on top of this thesis, and discuss some potential problems it might hold.

2. JetBrains MPS

Before we can dive into the problem itself, we need to give readers a good understanding of how the target environment works. We will walk through some basic features of the JetBrains MPS [1] so that we know what the goal of this thesis is.

JetBrains is a company with a long history of IDE (Integrated Development Environment) development. Some of their tools, such as IntelliJ IDEA[1] or Resharper[2], are widely used by professionals around the world and are considered to be top quality products. Their project called MPS (Meta Programming System) is a development environment that allows building custom domain specific languages or extending existing ones. Developers can use these newly created languages inside MPS and, out of their programs, they can generate actual code in a given target language such as Java. The project itself started in 2003 as a research project, but JetBrains have been using it in the development of their own products since 2006. It is being developed as an open-source product under the Apache 2.0 license.

In this chapter, we will explain fundamental basics of MPS as it is crucial in the understanding of what we are trying to achieve in this thesis.

2.1 Abstract Syntax Tree

At the first sight, MPS might look like a typical IDE, but it differs from the rest in one important aspect. MPS does not work with the textual representation of the code as usual, but rather with its AST (abstract syntax tree) that is the model of the code. The code itself, which is later compiled and run, is built out of this tree. So when users are creating programs using MPS, they are basically assembling the tree out of defined building blocks of the language. The definition of the language used dictates, which statements or elements can be nested inside each other and what structure the AST can have.

Keeping the model of the code in the AST form has several advantages. One of them is that MPS then only allows composing the program strictly following the syntax of the language, which results in an inability to actually write syntactically invalid code. It is generated out of the always-valid AST beyond user's reach. This is extremely handy when used, for example, for educational purposes, as you can guide students through code creation easily using auto-completion or other tools available. It also means that MPS understands the code much better and is able to perform some interesting actions or refactoring.

After the program's tree is assembled like this, there can be rules defined on how to generate code out of that tree. These defined generators can target any language and, possibly, we can transform this model into more different languages' models if such generators exist. This generated code can then be

[1]https://www.jetbrains.com/idea/
[2]https://www.jetbrains.com/resharper/

for example platform dependent, but the model itself is not tied to any specific platform or environment. This mechanism gives us, for instance, the possibility to extend existing languages easily and to create various syntactic sugar, targeting the same language again, but hiding complicated mechanisms from the user.

2.2 Projectional Editor

As stated before, MPS is not an ordinary tool, but its most distinctive feature might be the projectional editor. The projectional editor is the place where "interaction" between the code and the programmer takes place. The idea behind projectional editing is following. When the user is not working with the text source code directly, as usual, every node of the AST can be represented in any form and take any shape. It is just up to the designer of the language what shape will each node take inside the projectional editor. For a better understanding, we will give a small example.

Let's assume we have a language with typical if/else structure similar to the one in Java:

```
if (condition)
    then statement block
else
    else statement block
```

Its abstract syntax tree representation then might look something like shown on Figure 2.1.

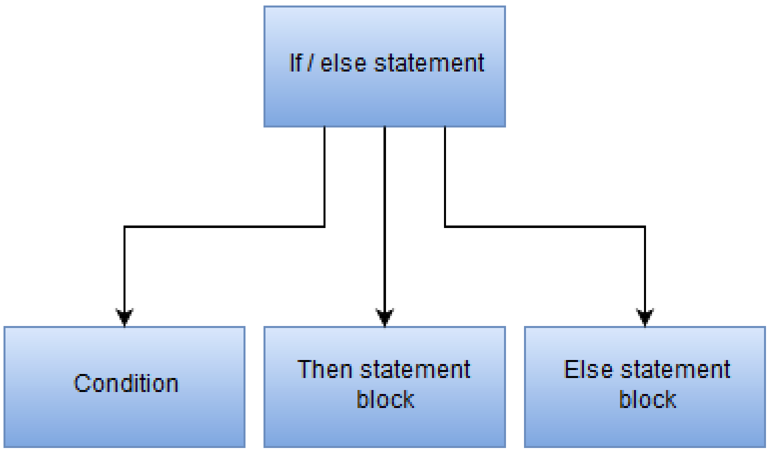


Figure 2.1: "If statement" abstract syntax tree

The underlying child nodes have their own children and so on. Their type is narrowed down by the language structure definition (meaning that we restrict the condition to be an expression, statement block to be a list of statements, etc.).

Now, the representation of code inside the projectional editor can have any visual form and is not bound to the textual representation at all. It can be a text-like representation similar to what we are used to, or for example blocks of both branches can be aligned next to each other like shown in Figure 2.2.

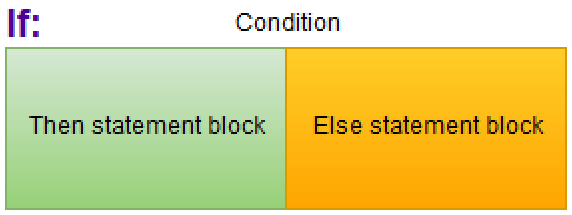


Figure 2.2: "If statement" projectional editor example

We have an absolute free will in how do we design each node of the AST. We can style the node graphically and define its layout in any way. This, of course, comes with some drawbacks such as designing a representation like this is quite complicated, even for a human.

For a definition of concept's appearance, JetBrains have developed a cellular system that allows placing concept's properties and children into different cells. These cells then can be styled to user's liking. There are many different types of cells, each behaving a little bit different towards its contents. We are talking about various horizontal or vertical lists etc. More extra cells can be added on top of that holding no content but specifying additional layout adjustments such as indentation. Each cell can have special context menus bound to it, its formatting can be customized and so on. In Figure 2.4, you can see what a real editor definition might look like for the if statement of the BaseLanguage language.

2.3 Languages and Solutions

The editor itself breaks up into two main parts — language and solution editors. The former is used for language definition, where we model languages' structure. The latter one then imports these MPS languages and enables the user to write code using them. Multiple languages can be used together inside one solution, effectively combining their features into one project.

2.3.1 Plugin Solutions

Users can also create a "plugin solution" which is a special kind of solution that can also import languages, but more importantly can tamper with the MPS editor itself. It is able to introduce elements inside the MPS such as menu items, window panels and more. The big advantage of MPS is that everything is happening live and just by mere regeneration (rebuilding) of a plugin solution, changes are immediately incorporated without the need of restarting the IDE. Plugin solutions are important in this thesis and were used to accomplish an important part of the work as later described.

2.4 Creating a Language in MPS

This section will illustrate how a language is created inside MPS, what elements make it up and what needs to be done in order to make the language usable.

Main building blocks of any MPS language are concepts. You can imagine they are the nodes of the AST and we build the code out of them. Every concept can have a definition of one or more different editor aspects. Every aspect defines the concept from a different point of view. One aspect might say where the concept in AST can be included, another one what code is generated out of it and so on. Concepts took over some design patterns known from object oriented programming. The concept can be inherited from a parent (abstract) concept and it can implement a special interface concept. Most importantly, concepts have child concepts and that is how the AST itself is created.

We will describe some of these aspects more closely, as it will give the reader better understanding of the problem that is being solved. Some are less important for our cause, some are vital.

2.4.1 Structure Aspect

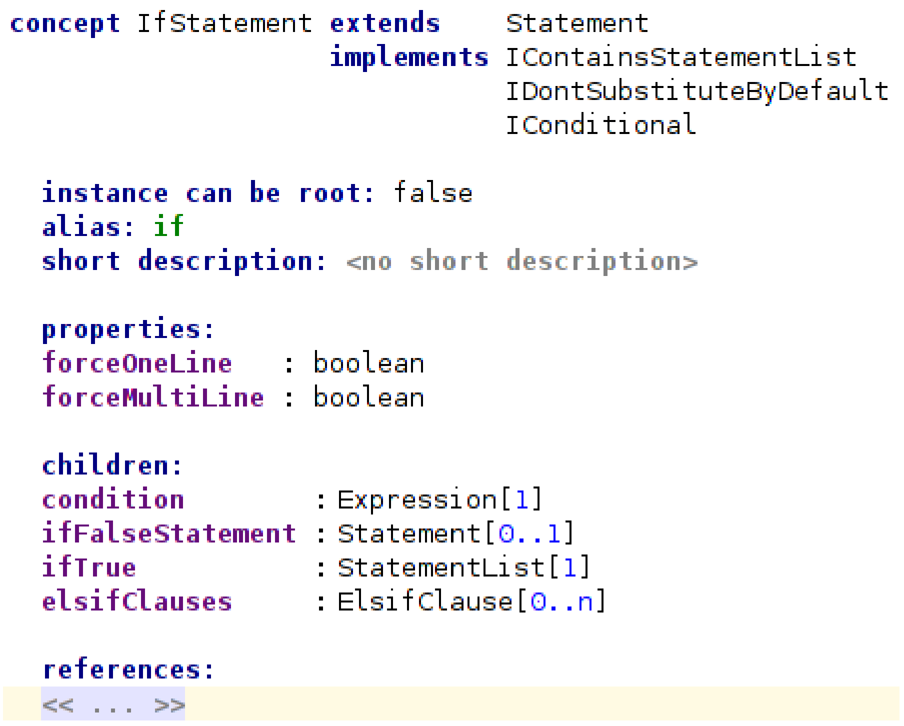
The structure aspect is the most important aspect, as all concepts must be first defined here. This aspect treats the concept somewhat like a Java class. User can define the name, inheritance, interfaces and concept fields:

- **Children** – a list of child concepts and their cardinality
- **References** – a list of references to already existing nodes (e.g. think reference to a node of a method in a method call statement)
- **Properties** – arbitrary fields of any type that can hold value, just like class fields in Java

The concept can be further marked as a *rootable* one, which means it can be a top level element for some code in a given language. In Figure 2.3, you can see the structure aspect definition for the if statement.

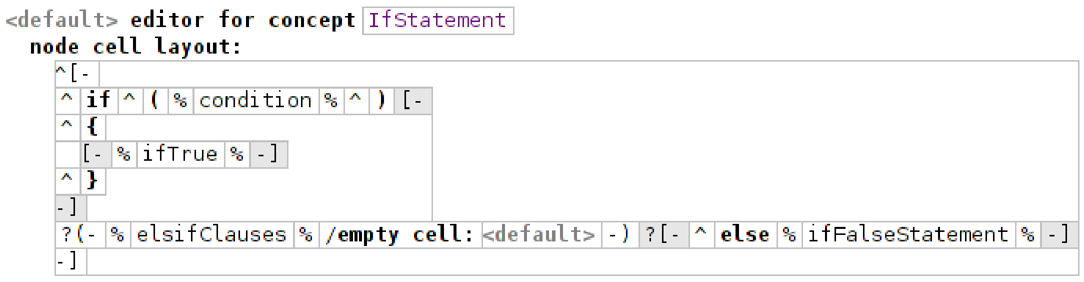
2.4.2 Editor Aspect

Editor aspect is where the user defines what the projectional editor representation of a code fragment (an AST node) looks like. The user usually incorporates all children, references, and properties inside the representation so that future user of the language can insert their value into these placeholders. Any other static components can be included too. All elements can be styled using a language similar to CSS. If editor concept is not defined, MPS will provide a default one that stylizes the concept into a JSON-like form, displaying all its fields nested in a JSON-like object. This provides basic ability to use the concept when coding but it is not very user-friendly.



```
concept IfStatement extends    Statement
                    implements IContainsStatementList
                               IDontSubstituteByDefault
                               IConditional

  instance can be root: false
  alias: if
  short description: <no short description>

  properties:
  forceOneLine   : boolean
  forceMultiLine : boolean

  children:
  condition         : Expression[1]
  ifFalseStatement  : Statement[0..1]
  ifTrue            : StatementList[1]
  elsifClauses      : ElsifClause[0..n]

  references:
  << ... >>
```

Figure 2.3: "If statement" structure aspect definition



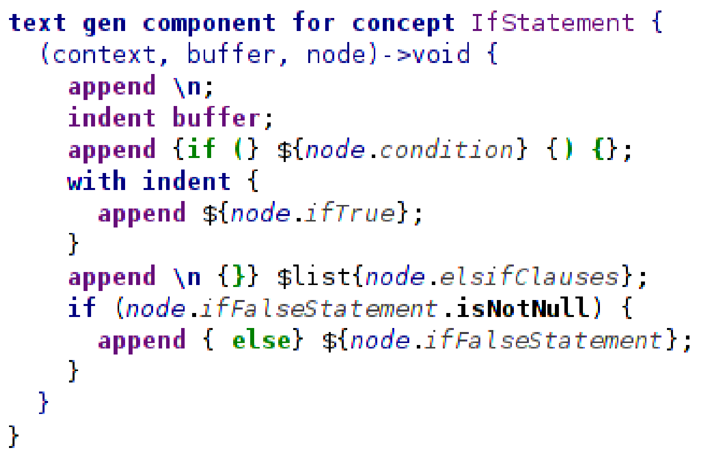
```
<default> editor for concept IfStatement
  node cell layout:
    ^[-
    ^ if ^ ( % condition % ^ ) [-
    ^ {
      [- % ifTrue % -]
    ^ }
    -]
    ?(- % elsifClauses % /empty cell: <default> -) ?[- ^ else % ifFalseStatement % -]
    -]
```

Figure 2.4: "If statement" editor aspect definition

2.4.3 TextGen Aspect

The TextGen aspect is a one we also care about. It defines how a certain node will be translated into a text source code. This will allow us to generate source code in text form out of an AST model and ultimately to generate a text program out of the code user writes in the imported language.

The TextGen aspect is basically just one method definition for each concept of the language. This method has several parameters such as the currently processed node and some contextual information. It is, however, not returning a string as perhaps expected, but rather manipulates an output buffer/stream using some built-in functions. MPS then calls this method for the root concept of the program and it is up to this concept's TextGen method to append its children into the stream by calling their TextGen methods. Again, we included an example of the if statement (Figure 2.5).



```
text gen component for concept IfStatement {
  (context, buffer, node)->void {
    append \n;
    indent buffer;
    append {if (} ${node.condition} {) {};
    with indent {
      append ${node.ifTrue};
    }
    append \n {}} $list{node.elsifClauses};
    if (node.ifFalseStatement.isNotNull) {
      append { else} ${node.ifFalseStatement};
    }
  }
}
```

Figure 2.5: "If statement" TextGen aspect definition

2.4.4 Other Aspects

Other aspects that can be found inside MPS, are not interesting for us from this thesis' point of view. They are mostly high level and very complicated aspects such as type checking or data-flow analysis (unreachable code detection etc.). Creating these automatically and programmatically wouldn't be possible without human intervention as the information needed just isn't contained inside the grammar file. These aspects will be used for adjusting the imported language and improving the usability of the language.

3. Goals Revisited

Now that we have introduced the background of the problem we will try to explain what this thesis is trying to achieve in a more technical detail.

3.1 The Structure Aspect

The first and the most important aspect of any language is the structure aspect. It is somewhat similar to a grammar. We will have to translate and transfer the structure of the language that is captured by the grammar, into the MPS structure aspect. This is the part where we will be able to spare the user from a lot of hours of tedious and sometimes quite challenging work.

Transferring grammar rules into the language's structure might present some interesting challenges. The grammar serves a little bit different purpose and the way we write down grammar rules causes some problems that must be overcome when we are recreating the language inside MPS. These obstacles were not known, when the work on this thesis had started, but were later discovered in the midst of the research. It is also one of the efforts of this thesis — to describe these problems more closely as they are common for everyone tackling a similar problem.

3.2 The Editor Aspect

For a language to be comfortably usable inside MPS, some other aspects of the language must be defined too. Sole import of language's structure will not be sufficient as we won't be able to use the language. The default projectional editor that is automatically generated for each concept, when no editor aspect is defined, is a very chatty, not user-friendly and practically unusable. It almost equals assembling the AST manually. The editor aspect is the second one that we will have to generate, in order to make the import practically useful.

One of the biggest problems that arise here is that the grammar description of a language's structure does not hold any information about the code layout whatsoever. Structural rules only tell us, what the syntax tree looks like and how the code is broken into nodes and child nodes. It, however, says nothing about indentation, line breaks, and another formatting. This thesis will discuss possible solutions to this problem as there must be some intermediary step whose purpose will be to generate this information. It will happen either automatically using some heuristics or with user's help in an interactive manner. It is expected that this step might be a topic for future efforts, following up on this thesis that might improve it.

3.3 The TextGen Aspect

Finally, we must make sure that the user is able to generate a real text source code out of the AST they had built inside MPS. For this, we will have to create the

TextGen aspect for each concept. Again, we will face some interesting challenges connected to the code layout, whitespaces and more.

4. Related Projects

From the beginning, it was clear that there are more ways how to approach the problem of a grammar import. The first step in the exploration of this topic was research of projects that try to tackle the same or similar problem, as we do. Part of the goal of this thesis was to evaluate pros and cons of other approaches because we had to decide whether we will build on top of these projects, use their parts or just be inspired by them. Some of those decisions were made together with employees of the JetBrains company as they also have some vested interest in this thesis' outcome and wish for some specific functionality.

Studying similar projects turned out to be a really good choice for the first step, as it gave the author of this work much better insight in what problems might arise along the way. More importantly, it put goals of this thesis in perspective and helped us realize how we want to try to solve this problem.

4.1 PE4MPS

PE4MPS [4] is an open source project that is trying to solve the same problem as we do. It is dealing with the fact that grammars lack the layout aspect in them by creating a new grammar notation called PE Grammars [5] (the PE abbreviation comes from projectional editing).

4.1.1 Project Description

The author decided to mimic an existing grammar notation called ANTLRv4 [6] and enrich its syntax with own constructs. These extensions hint the parser, what the AST node layout should look like. The parser then uses this information when generating the projectional editor for this node. The author described the PE syntax using the ANTLRv4 notation [7] and then auto-generated an ANTLRv4 parser for PE grammar files. This PE parser then reads any PE file and stores the language structure found inside to a custom representation of Java objects.

On top of the PE parser project, the PE4MPS project is built. The PE4MPS project contains an MPS plugin, similar to what we are trying to build in this thesis. This plugin uses the parser to build the PE file representation (the aforementioned tree-like structure of Java objects) and then creates concepts and their aspects inside MPS. The extended syntax that PE brings, describes the layout of each element, e.g. it tells the plugin that this set of child nodes should be displayed horizontally, another set should be vertical — each child on a separate line with some indentation and so on.

4.1.2 Approach Evaluation

Even though this project has some interesting ideas, we decided not to follow its approach with our own implementation. The author of this theses by no means wishes for defaming the PE project or declare it as a bad one. We only came to a conclusion that it deals with a similar problem slightly differently in a way that does not suite our cause. We will try to explain to the reader what were major factors that lead to this decision.

The main goal of this thesis is to automatize as much as possible. We would like to import full language structure and then do a lot of mundane and time-consuming work connected to creating projectional editors. The problem we identify within PE is that it does not really spare us this work, but only shifts this work from projectional editor to writing PE grammars. So what the user could have been doing inside MPS UI using the projectional editor, which specially made for this, now has to be done using a text editor when editing grammars. The text editor is not as user-friendly and, more importantly, a very error prone environment. Even more so as so far there are only a few extensions implemented such as horizontal and vertical lists and some indentation rules and even these few features make the already not-so-simple syntax much more complex. We conclude that adding more features would continue on making the grammar language even more complicated.

Another reason for deciding to abandon this approach is that MPS always strives for simplicity when it comes to the end user of MPS languages. The idea is helping the user as much as possible to code inside MPS, using aids such as auto-completion. This enables the user to leverage the advantage of having a better understanding of the code thanks to the AST representation than regular IDEs usually have when only parsing text. On the contrary, users of PE would have to study ANTLR and PE's new syntax. Users would also need to study particular ANTLR grammar of the language they're importing so that they know which parts of the language should be adjusted. Some grammars have thousands of these rules, which make orientation in the structure quite difficult for a human user and would make this process even more complicated. We believe this is an inferior approach to using MPS projectional editor UI that was designed exactly for this purpose. It guides the user through the process, using tools like auto-completion, and of course already, has a lot of available learning materials and documentation available. The user is then shielded from this complexity.

There is another issue connected with this that we have discovered later in the process of our own implementation. Grammars are written by humans and therefore have some structures inside of them that help a human brain understand it better. However, these structures add unnecessary levels in its syntax hierarchy which later lead to problems when using this language inside MPS. The user of PE would have no idea how to know this and which parts of the grammar are troublesome like this. They might adjust parts of grammar that will be left out later in the process when making the grammar simpler. We will discuss this problem later in this thesis when describing obstacles we have met on our own (Chapter 6).

Another problem we see in PE's approach is that it, from what we understood, does not implement full ANTLR syntax. This means that every grammar might need a non-trivial adjustment before its usage. One of our goals is for us to be able to import as many languages out-of-the-box as possible and adopting the full specification.

Even though we have found some issues with this approach, the PE4MPS project served us a huge aid when dealing with MPS itself. MPS has good documentation and learning materials describing working with it (how to create languages and use them). A big obstacle for us was, however, a lacking MPS documentation when it came to its advanced internal API. This API is used for example for programmatic language generation inside MPS which is rather a deep level use case. There is no reference and coding inside MPS using the BaseLanguage language is not easy. For example, discovering the API is (without proper intelligence) not a pleasant experience at all because the programmer needs to advance in a trial-and-error fashion. The author of the PE4MPS project has dealt with this too (we discovered that through some correspondence with them) and had to tunnel through some non-trivial obstacles. This helped us a lot because we had something to hold on to when trying to use the API ourselves.

Overall, the PE4MPS project showed us what the added complexity of projectional editor generation is. It gave us an understanding of how big of a chunk of information concerning the code layout is in the grammar in fact missing. It also helped us to realize how we do not want to tackle this problem and how the automation should work.

4.2 ANTLR_MPS

Another project that is dealing with a similar problem is ANTLR_MPS [8]. The author of this project, Fabien Campagne, is the author of official MPS user handbooks ([9][10]), which also helped us when working on this thesis. The ANTLR_MPS project also uses ANTLRv4 grammar notation [6] and tries to import grammars inside MPS. It does not, however, deal with the problem of generating projectional editor at all, probably because it is in its early stage of development. The import process itself is quite different from what we've seen in the PE4MPS project, showing us the variety of approaches that are possible to go with.

4.2.1 Project Description

Using this project to import a grammar is a very complicated process. We will try to briefly describe some basic steps showing how the plugin is used. We have skipped some more steps because we weren't trying to describe the whole procedure, merely giving the reader a brief overview on how complicated and a bit confusing the process is. Full tutorial can be found inside the project's official repository [1].

The author created a whole ANTLRv4 MPS language which is an MPS port of the grammar notations' syntax. To import a language, the user utilizes this

[1]https://github.com/CampagneLaboratory/ANTLR_MPS/blob/ebc35f346ad1bfc67022056199ea6096221e601e/Tutorial.pdf

MPS ANTLR language. They have to import the textual grammar into MPS taking the form of the MPS's ported grammar language (so that the textual grammar is converted into MPS nodes). This can be done automatically and it means the author of the project needed to parse the original grammar and then recreate it again using the MPS's ANTLRv4 language. This seems like a little bit unnecessary step. The plugin then creates an ANTLR visitor, which is a special Java class used for parsing. The user must manually compile this visitor into a .jar package and reference this file inside the MPS project. After that, concepts are created for each grammar rule. They need to be mapped manually together with the grammar's concepts which needs to be done by the user of the plugin. For some languages, there are hundreds of rules defined and this becomes a very tiresome and prolonged effort. There are no children or properties created for these concepts that is also up to the user of the plugin. There are no editor nor TextGen concepts created neither.

4.2.2 Approach Evaluation

From the process description, it is quite clear that this might not be the best approach for us to pursue. We are trying to automate as much as possible. Unfortunately, there is not much we could leverage from in this project. There is no attention paid to the problem of projectional editor generation and text generation is neglected too. We blame it on the project being in its early stage of development. That is maybe why we couldn't see the author dealing with some advanced problems concerning the grammar structure that we have observed ourselves. The code is not documented much and is quite dispersed across different aspects, so it is quite complicated to locate some functionality to discover author's ideas. From reasons stated above we decided not to follow up on this project.

5. Source Grammar Notation

One of the early decisions to be made was the source of imported languages. It was clear that we needed to start with some grammar description of the language as the main part we cared about was the structure of the language. There exists a large amount of different notations that dictate, how the grammar is written down[1]. When analyzing these notations, following aspects were taken into account:

- **For how many languages there exists a representation in this notation?**

  Once we choose a notation, we would like to be able to import as many languages as possible without the need for writing the grammar ourselves.

- **Is there a possibility of generating parsers from these grammars?**

  And if there is, is Java supported (we are targeting Java, because MPS is written in Java)? How can we use these parsers? Later we will need to be able to automatically parse code and recreate its AST representation inside MPS.

- **What is the overall complexity of the notation?**

  How difficult will it be to transfer the grammar into MPS? Does the notation contain any extra information about the code layout?

- **What other tools are there working with this notation?**

  Can we use them? We are talking for example about some graphical AST viewers that might help us in development.

- **What did authors of similar projects use?**

  What were their reasons to go with this decision? Was it a good decision or more of an obstacle?

Based on these preconditions an ANTLRv4 grammar notation [7] was chosen. It is a widely used grammar notation that:

- has a large amount of languages that have their syntax captured using this notation [11].
- has a whole framework [6] built around it allowing generating parsers. It is supporting Java (it is written in Java). It has a tree walker design that allows listening to events when walking the code and reacting to them.
- has a thorough reference documentation of its syntax [7] which we will appreciate when parsing grammars themselves.
- is in an EBNF (Extended Backus-Naur Form) [2] form that will too come in handy when parsing grammars as it meets the object oriented design that is used when describing MPS concepts.

[1]https://en.wikipedia.org/wiki/Comparison_of_parser_generators
[2]https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form

- there exist a variety of tools[3] for, such as grammar editors, syntax highlighters, tree visualizers and more.
- was also chosen by authors of similar projects such as ANTLR_MPS [8] or PE4MPS [4], which are more closely discussed in Chapter 4. They achieved their goals using this notation quite effectively thanks to parser generation and other tools (e.g. the ANTLR_MPS projects creates a diagram of the notation).
- has its own syntax captured in its own syntax[4] (there is an ANTLRv4 grammar for the ANTLRv4 notation). This is a very important aspect as it allows us to create a parser for ANTLRv4 grammars so that we can parse other languages grammars.

[3]http://www.antlr.org/tools.html
[4]https://github.com/antlr/grammars-v4/tree/master/antlr4

6. Building The Plugin

This part of the thesis will describe several phases of a gradual implementation of the MPS plugin. We will show all steps needed and we will keep this chapter in the form of an implementation diary, talking about the way the author proceeded. It will allow us to slowly walk through all the obstacles the author has encountered. We believe that this will give readers a better insight into the problematics than just describing the final solution. It might also help others who might deal with similar problems and help them to understand the problems more deeply, maybe choose a different path or perhaps just to avoid some pitfalls we have discovered on our own.

The chapter first defines our custom language that we will be showing examples on. Then we will talk about parsing grammar files. Lastly, we will talk about several approaches on how to tackle generation of language's aspects (structure, editor, and TextGen).

6.1 The SimpleXML Language

Throughout the whole chapter, we will be using an example language so that we are not switching from one context into another frequently. We will be showing what this language needs for its good usability and what problems it contains as we will add more and more features into the import process.

The language is a simplified version of XML[1] and we will be calling it **SimpleXML**. We created the grammar of the language by taking the official XML ANTLRv4 grammar[2] and stripped it down from some not so interesting features such as XML entities. The author has decided to remove these features as they don't add any extra value to our cause and it will be easier to reason about the language as the grammar notation becomes shorter. We have also made some further adjustments that make the grammar better suitable for our cause. We will talk about these later in Chapter 9 and explain why they were necessary.

[1]https://en.wikipedia.org/wiki/XML
[2]https://github.com/antlr/grammars-v4/tree/master/xml

6.1.1 SimpleXML Grammar

Below, you can find the grammar describing the SimpleXML language. We will be working with this grammar when implementing the MPS import plugin. As stated before, we are using the ANTLRv4 notation [7].

```
grammar SimpleXML;

document      :   prolog? comment? element ;

prolog        :   '<?xml ' attribute* '?>' ;

comment       :   '<!--' TEXT '-->' ;

element       :   '<' Name attribute* '>' content* '</' Name '>'
              |   '<' Name attribute* '/>'
              ;

attribute     :   Name '="' TEXT '"' ;

content       :   TEXT
              |   element
              |   comment
              |   CDATA
              ;

Name          :   NameStartChar NameChar* ;

fragment
DIGIT         :   [0-9] ;

fragment
NameChar      :   NameStartChar
              |   '-' | '_' | '.'
              |   DIGIT
              ;

fragment
NameStartChar:    [:a-zA-Z] ;

TEXT          :   ~[<"]* ;

CDATA         :   '<![CDATA[' .*? ']]>' ;
```

We have colored the grammar out so it is easier to find our way around it when referencing it. The grammar contains following basic elements:

- **ANTLRv4 keywords** – In black, they don't add any extra value in this particular scenario. Nonetheless, they are required by the specification.
- **Parser rules** – Stated in blue, they are the rules describing language's structure. The ANTLRv4 notation dictates that all parser rules must start with a lowercase letter and there is a semicolon behind the last alternative. Every rule consists of a set of alternatives that the element might break into. These alternatives are separated by the pipe (|) character. For example, the basic XML element (rule called element), can appear in two different forms — either containing both the start and the end tag or its simplified self-closing form when the tag is empty. The element rule, therefore, contains two alternatives describing both forms.
- **Lexer rules** – In yellow, always starting with an uppercase letter, they are rules describing terminal symbols. The parser is matching these against the input. Lexer rules can also break into more sub rules in the same manner parser rules break into alternatives. This breaking, however, eventually stops at string values and regular expressions at the bottom level, so, in the end, these rules are describing some string. Parser rules also eventually break into lexer rules too. The **fragment** keyword states that the lexer sub-rule is just a helper rule that brings more clarity into the notation but is not visible in the parser output.
- **String values (literals)** – In red, always inside of a pair of apostrophes, they are similar to lexer rules. They describe a terminal string, but not using regular expressions but exact string match.
- **Regular expressions** – Colored green, they describe a string token to be matched using a regular expression with the special ANTLRv4 regex notation[3].
- **Rule operators (?+*)** - Left in black, following any element, they quantify the number of occurrences the prepending element can appear in. They are appended to one or more elements (enclosed in braces) of a rule's alternative. These operators can also be prepended by an asterisk (*), meaning they are non-greedy. This allows us for example to simplify rules such as the CDATA one. The parser will use this rule as expected while there is no need to exclude the '`]]>`' sequence inside the regular expression.

[3]https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md

6.2 Programming Inside MPS and the BaseLanguage

In order to create some of the aspects of the imported language, such as the TextGen aspect, we will need to add complex functionality inside the aspect definition. This can be done using programming in the BaseLanguage language (also an MPS language, just like the one we are creating). BaseLanguage is the official JetBrains port of Java for MPS and is used for all additional programming inside MPS. There are some additional extensions for this language that enable us using the MPS API, such as the structure or editor language. These extensions contain concepts allowing us to work with language definitions or the editor environment itself (menu items etc.). When building our plugin, we needed to:

1. Use the BaseLanguage API to generate language elements (concepts, aspect definitions).
2. Generate BaseLanguage code inside aspect definition of the imported language that will make the language more usable.

6.2.1 Generating BaseLanguage Code

Because we are inside MPS and BaseLanguage is an MPS language, BaseLanguage code is not a text source code, but again, an AST built out of concept nodes belonging to the BaseLanguage language. This means that generating code inside MPS consists of generating some AST out of BaseLanguage nodes. It is a very interesting problem since it is a little bit more challenging than just generating plain text code. For example, take a look at this simple BaseLanguage statement, where we instantiate an object into an object's property:

```
foo.bar = new node<Element_1>();
```

As simple as it looks, in order to programmatically generate this statement, we need to do many things:

1. We need to find the declaration of the foo variable.
2. Look up foo's property bar.
3. Tell the MPS, we would like to build an assignment statement.
4. Set the left side to some sort of a property-access-expression, using the foo declaration.
5. Set the right side to contain a new statement with some specific type of the template.

The full AST tree that needs to be built, in order to create a statement like this, is shown in Figure 6.1. We need to be building this from bottom up, node by node.

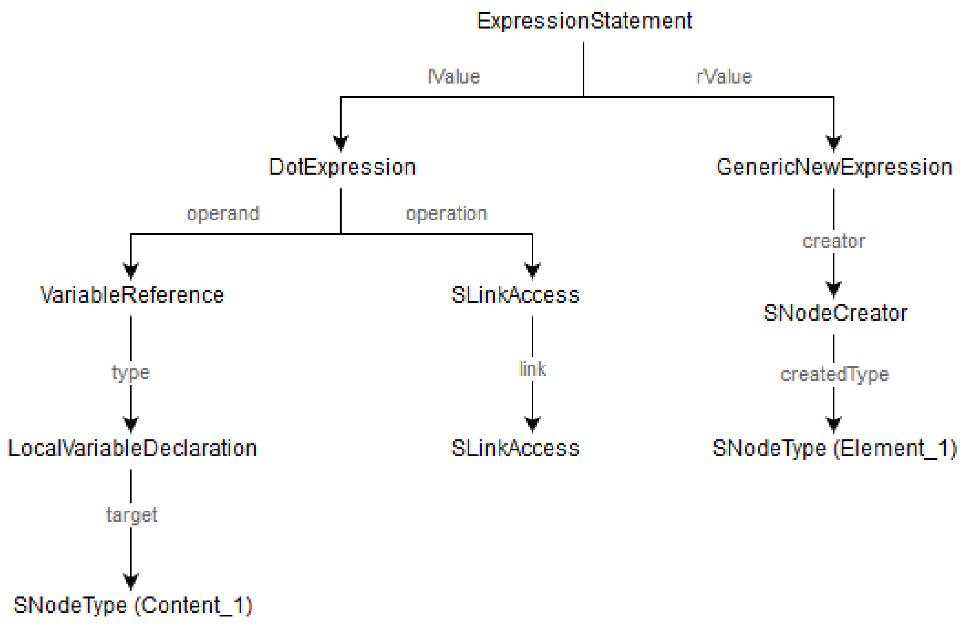


Figure 6.1: BaseLanguage "New statement" abstract syntax tree

6.2.2 Quotation

The example above showed, how many nodes are needed in order to represent quite basic simple statement. Luckily for us, the developers of MPS have thought about this and implemented a special notation that helps to build expressions. This notation is called *quotation* and it enables us to insert statements into a special quoted block, similarly like an eval function can parse a string into a real code.

Furthermore, inside this quotation block, we can insert an anti-quotation block, from which we can reference variables in our scope outside of the original quotation. This is what the above statements looks like when quotation (light blue) and anti-quotation blocks (yellow) are used:

```
<%( fooRef )%.%( barProp )% = new node <^( type )^>();>>
```

The quotation block will yield the AST tree from Figure 6.1 without the need for building it node by node.

6.2.3 Generating Dynamic Code

The quotation comes in handy when we are generating static code — meaning, we know, in the time of implementation, what code we want to generate. But what if our code depends on input data? There are several parts of the import process, where the generated BaseLanguage code depends solely on our grammar's structure. In those cases, we cannot simply use quotation. We are left with generating statement's tree node by node as in the first example. The quotation can make some of these parts a little bit shorter, but doesn't solve the problem entirely.

6.3 Parsing the Grammar

The first stage of the import process is parsing the grammar file so that we can get an AST tree representing the grammar. The parser must be able to read any grammar file and extract the structure information. As stated in the chapter defending the grammar notation selection (Chapter 5), there exists an ANTLRv4 grammar for the ANTLRv4 notation [7]. This means that we can use the ANTLR library [6] to generate a Java ANTLR parser of the ANTLRv4 notation.

6.3.1 Grammar Representation

The abstract syntax tree that comes out of the automatically generated parser, is a little bit complicated and full of information not relevant to us. It also contains all structures including ANTLR's special characters (pipes, semicolons, comments...), because ANTLR doesn't understand the grammar and just parses it. That is why we decided to translate the complex ANTLR AST into our own simplified tree made out of our own objects. We discarded as much irrelevant information as possible and only kept vital data. We can understand this simplified tree way better and it brings more clarity into our solution.

Furthermore, we process the grammar in two iterations. In the first, we build a tree holding names of other referenced rules in the form of strings. In the second, we walk this tree and resolve all these references to real pointers to other rule objects. We end up with a neat representation that is easy to parse in next steps of the import process.

6.3.2 Flattening Lexer Rules

One of the questions we had to deal with was lexer rule representation. Not minding all the syntactic sugar ANTLR is offering for their definition, in the end, lexer rules are basically just regular expressions used for parsing input source code into tokens. However, after the initial parsing, we end up with some kind of a tree structure that is representing the lexer rule (lexer rules can also be built from alternatives just like parser rules). We would like to take this tree that represents the lexer rule and flattens it into the regular expression. Later in the import process, we would like to assign this expression to some constraint entity representing the rule.

Let’s look at the lexer rule Name that we have in our SimpleXML language, together with its sub-rules:

```
Name         :   NameStartChar NameChar* ;

fragment
DIGIT        :   [0-9] ;

fragment
NameChar     :   NameStartChar
             |   '-' | '_' | '.'
             |   DIGIT
             ;

fragment
NameStartChar:   [:a-zA-Z] ;
```

The regular expression describing the Name rule is following:

```
[:a-zA-Z](([:a-zA-Z])|\-|_|\.|[0-9])*
```

We can notice that we can achieve this by gluing elements of each alternative together and then joining these alternatives with a classic regex OR (|), which is exactly its function in the ANTLR grammar notation. We have put these thoughts into a form of the following recursive algorithm:

```
Flatten($R$):
1) Define $T$ as a list of empty string
2) For each alternative $A$ of the rule $R$:
3)     Define $R$ = ''
4)     For each element $E$ of $A$:
5)         If $E$ is not yet flattened:
6)             Flatten($E$)
7)         $R$.append($E$)
8)         $R$.append($E$.operator)
9)     $T$.add($R$)
10) Build a string S out of elements $t_1$, $t_2$, ... $t_n$ of $T$:
11)     ($t_1$)|($t_2$)|...|($t_n$)
12) Return $S$
```

The output of `Flatten(Name)` then would return a regular expression in the form of:

```
([:a-zA-Z])(([:a-zA-Z])|(-)|(_)|(.)|([0-9]))*
```

We have made some further adjustments, which, for example, escape special characters or remove unnecessary braces. Then we had to add some other minor transformations, because the regular expression notation of ANTLR is not the same as in MPS (Java). Special characters also had to be doubly escaped, because

MPS holds regular expressions in the form of a string and not all escape sequences are allowed.

6.3.3 Subrules

A new feature, called *subrules*[4], was added to the fourth version of the ANTLR notation that makes parsing the structure a little bit more complicated for us. This feature allows inlining some rule expansions directly inside an alternative. Following excerpt from the official XML's ANTLRv4 grammar definition shows this extended syntax:

```
content :   TEXT? ((element? | CDATA | COMMENT) TEXT?)* ;
```

There is a nested block contained in the second part of the rule that has another block nested inside (in-line alternatives). Even though the content rule contains only one top-level alternative, it is possible to build its contents from more variations. There can be any number of levels of these nested blocks and each block can be annotated with its own quantification operator.

This holds a complication for us because we need to parse these subrules too. We decided to solve this by a recursive traversal of the AST and expanding subrules just like they were classic simple parser rules. For each subrule block, we generate a new parser rule and then reference this rule from its original place. The full expanded content rule then looks like this:

```
content :   TEXT? block_1* ;

block_1 :   block_2 TEXT? ;

block_2 :   element?
        |   CDATA
        |   COMMENT
        ;
```

[4]https://github.com/antlr/antlr4/blob/master/doc/parser-rules.md#subrules

6.4 The Structure Aspect

After we have mined the structure of the language out of the grammar file, we can finally start working with MPS. We need to translate our tree structure into the terms of MPS. That means creating concepts inside the structure aspect and linking them together appropriately.

We will describe several different attempts (Sections 6.4.3 and 6.4.4) the author has tried out and show some problems these attempts introduced. We consider it a better approach than just simple description of the final solution because it might prevent others from falling into similar pitfalls such as we discovered ourselves.

6.4.1 Concept Types

Before we start describing how we decided to import our tree structure into MPS, we should remind in more detail which means of expression are available to us. As stated before, concepts are building blocks of all MPS languages and they can be treated similarly as Java classes. They can:

- extend a parent concept
- inherit any number of interfaces (interface concepts)
- have properties (similar to class fields) with a given data type — either a primitive type such as string or integer, or a constraint data type — a string whose value is restricted by a regular expression
- have children concepts (defined as a concept or an interface concept)
- hold references to other nodes (e.g. function call statement would know reference to the function definition)
- have an *alias* and a *description* field for their better identification in auto-completion menus

6.4.2 Common Ground

Some of the import steps are common for both approaches that we have implemented and described in Sections 6.4.3 and 6.4.4.

Firstly, we have noticed that whenever there is a string literal inside a parser rule's alternative (think of the **for** keyword from a Java loop), it will appear only in that rule's projectional editor. There is no need for it to have any function as it is set in stone as an unchangeable part of that alternative's appearance. It might serve its purpose when describing or naming this alternative inside an auto-complete menu, but we will get to that part below.

Secondly, we can take the flattened lexer rules that contain the regular expression, we have built by gluing its parts together in the parsing stage. For each one of these, we create a constraint data type concept, which is exactly what we need. We will be able to later create properties of concepts and set their data type as this constraint data type concept, effectively restricting their value using given regular expression.

Last, we take all literals found in the alternative and glue them together to one string, which we will use as the alias of the concept. We will use the name of the parser rule, this alternative belongs to, as the description text for the concept.

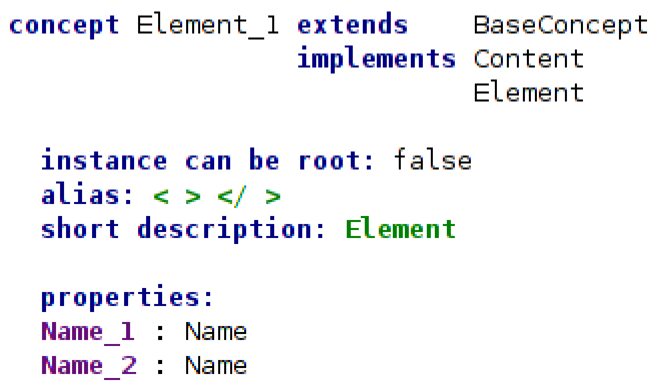
To illustrate what a concept for a sample alternative looks like, consider the element concept that represents the full XML tag with content:

```
element       :   '<' Name attribute* '>' content* '</' Name '>'
              |   '<' Name attribute* '/>'
              ;
```

Because we are creating a new concept for each alternative of the rule, we will be numbering them correspondingly. The concept that will represent the first alternative of the element rule, will therefore be named Element_1 (we will mark concepts green and interfaces purple).

- Since there are two references to the Name lexer rule inside the first alternative of the element rule, the Element_1 concept will contain two properties, whose value will be restricted using the regular expression representing the Name rule. We can achieve this using the *Constraint data type concept*, which we create for each lexer rule.
- Literals are skipped (as explained above using the **for** keyword).
- Parser rules references (attribute and content) will be explained further down the road as they are the part where approaches differ.

Result of this first step can be seen in Figure 6.2.



```
concept Element_1 extends    BaseConcept
                  implements Content
                             Element

  instance can be root: false
  alias: < > </ >
  short description: Element

  properties:
  Name_1 : Name
  Name_2 : Name
```

Figure 6.2: Resulting Element_1 concept

Next, we are going to describe two different approaches — the straightforward approach (Section 6.4.3) and the shortcut approach (Section 6.4.4). The difference between these two lies in the way we create children fields for parser rules and in the way we are going to link them together using interface concepts. Final solution will be described in Section 6.4.5.

6.4.3 The Straightforward Approach

The first attempt was quite straightforward and a little bit explorative, in the way that the author had to discover the MPS API that allows programmatical language generation. Because MPS is still in development, the API isn't that well documented and some features had to be discovered through trial and error or through examination of the PE4MPS project (Section 4.1), which served great aid here.

6.4.3.1 The Algorithm

The main idea behind the first attempt comes from the realization that when a parser rules breaks into more alternatives, we need to create a concept for each alternative. Then we have to somehow mark them as belonging to that parser rule. Consider the content rule from our SimpleXML language:

```
content    :    TEXT
           |    element
           |    comment
           |    CDATA
           ;
```

We will need to have 4 concepts that could appear anywhere the content rule is referenced. That is why we decided that for each rule with more than one alternative, we will create an interface concept. Then for each alternative of this rule, we will create a concept that will implement this interface. So for our content example, we will get following setup:

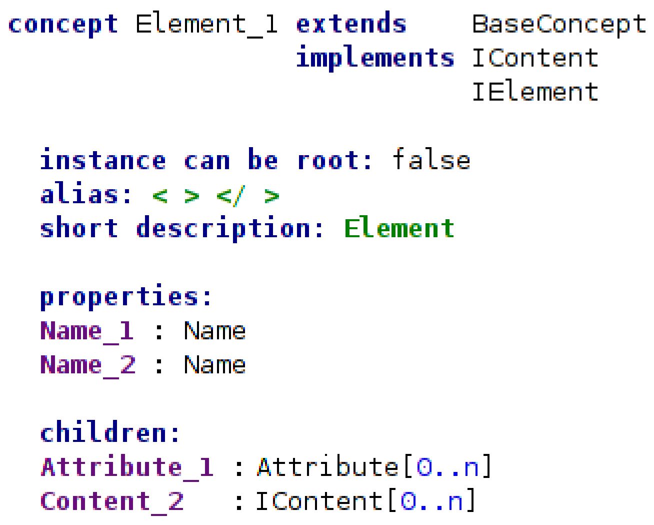
```
IContent   :    Content_1
           |    Content_2
           |    Content_3
           |    Content_4
```

Names of these concepts are derived from the name of the rule, numbers are added to alternatives correspondingly. For rules with a single alternative, no interface is needed. For rules that contain in-line block rules (Section 6.3.3), we have already created artificial parser rules in our tree representation during the parser phase (Section 6.3), which means we do not have to worry about them now.

Now we will describe, how we linked parser rules together. Consider the element rule that is referencing the content rule:

```
element     :   '<' Name attribute* '>' content* '</' Name '>'
            |   '<' Name attribute* '/>'
            ;
```

Following the algorithm mentioned above, there is one IElement interface and two Element_1, Element_2 concepts created. For each referenced parser rule inside a concept, we create a child link and point it to the right interface. In our example, for concept Element_1, there would be two child links pointing to IAttribute and IContent. In Figure 6.3, you can see the full Element_1 concept.



```
concept Element_1 extends    BaseConcept
                  implements IContent
                             IElement

  instance can be root: false
  alias: < > </ >
  short description: Element

  properties:
  Name_1 : Name
  Name_2 : Name

  children:
  Attribute_1 : Attribute[0..n]
  Content_2   : IContent[0..n]
```

Figure 6.3: Element_1 concept's structure aspect

Naming convention of child links also contains numbers, because names have to be unique.

6.4.3.2 The Layer Problem

The algorithm mentioned above will leave us with an MPS language that follows given structure correctly and could be used to represent code in that language inside MPS correctly. Editing the code will not be possible yet, as there is no editor aspect defined so far. There is one problem, however, concerning usability of such language. We will demonstrate this problem on the SimpleXML language, using element and content rules:

```
content     :   TEXT
            |   element
            |   comment
            |   CDATA
            ;

element     :   '<' Name attribute* '>' content* '</' Name '>'
            |   '<' Name attribute* '/>'
            ;
```

When we would be editing code in this language, the MPS auto-complete would give us aid when filling out all properties and children of inserted concepts. Imagine there is a freshly inserted Element_1 concept (a concept representing the full XML element as is stated in the first alternative of the element parser rule). This setup can be seen in the left part of Figure 6.4. Now, we would like to insert some content inside. Let's say we would like to insert a nested XML element inside. We would place the cursor in the content placeholder and press Ctrl+Space to view the auto-completion options.

Following the algorithm mentioned above, we have created a child link of the Element_1 concept of type IContent. There exist 4 concepts that implement the IContent interface as per each alternative of the rule. MPS evaluates this and gives us 4 options inside the auto-complete. This auto-complete is captured in Figure 6.4.

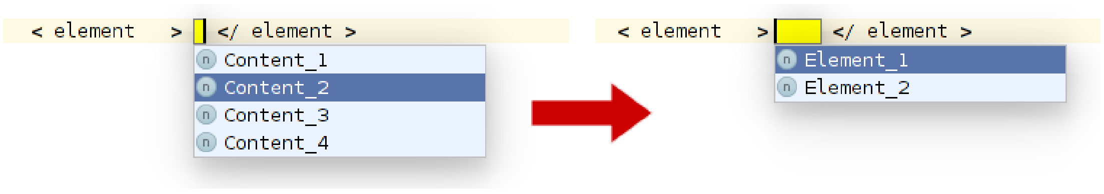


Figure 6.4: Layer problem in auto-completion

In order to correctly insert another nested element inside, we would have to first insert a Content_2 concept inside Element_1 that has an IElement child inside (and nothing else). Then, as a second step, we would call for the auto-complete again and insert either Element_1 or Element_2 inside Content_2. This mean that we have to go through two steps and in the first one either guess correctly (or remember the grammar rule's alternative order), to know, which item of the auto-complete we should go with.

And the problem goes both ways — if we decide to replace the nested Element_1 with, let's say, an XML comment (a Comment concept), we need to delete both intermediary layers before we get back to the original Content_X crossroads. Meanwhile, the user cannot really see, what is happening, since the intermediary level has no appearance or indicator. This leads to confusion on user's part.

We tried to ease this situation up by creating aliases for concepts derived from their content (names such as "Element content" or "CDATA content" instead of Content_N), but the problem goes beyond this. Take into consideration that the SimpleXML grammar is a very simple one. More sophisticated languages may have more than two intermediary layers and writing code would consist of clicking through a large number of auto-completes like shown in the example above. There is also nothing preventing authors of grammars from naming these helper layers in a completely unrelated manner, which would make user's orientation even harder. After all, these layers come into existence when the grammar is being written, so that it is more readable for humans and maybe better maintainable by the author. It has nothing to do with making the AST simple for the ANTLR parser.

Furthermore, using in-line subrule blocks makes this even a bigger problem as the block rule's names are auto-generated by our parser and differ only by numbers, even though the alias assigning process improves the situation by a bit. But from the point of view of the grammar's author, it is an invisible layer of rules nested inside.

6.4.4 The Shortcut Approach

After the problem with layers crossed our path in the first attempt, we took a step back and tried to reevaluate our approach. The second attempt is based on looking at elements of alternatives (parser rule references) and asking: "which concepts could be inserted in this place?". Consider again the content rule and its child rules:

```
content    :   TEXT
           |   element
           |   comment
           |   CDATA
           ;

element    :   '<' Name attribute* '>' content* '</' Name '>'
           |   '<' Name attribute* '/>'
           ;

comment    :   '<!--' TEXT '-->' ;
```

We started building on top of the algorithm mentioned in Section 6.4.3.1. This leaves us again with a concept for each alternative and an interface for parser rules containing more alternatives. We can see that the content rule can ultimately expand into following concepts:

- **Content_1** (TEXT)
- Content_2 → **Element_1**
- Content_2 → **Element_2**
- Content_3 → **Comment**
- **Content_4** (CDATA)

The layer problem (Section 6.4.3.2) dwells in the need of inserting the Content_2 concept before being able to insert one of the Element_1 or Element_2 concepts. We would like to avoid that and offer directly the concepts that are at the end of the chain (in bold). These are concepts we would like to see in the auto-completion menu. They cannot transparently break into more rules and for the sake of text we will call them **end concepts**, or when talking about the grammar rule tree **end rules**.

We will describe an algorithm that will find all end rules for a given parser rule, and later utilize it.

6.4.4.1 The Algorithm

To find end rules for each parser rule, we can recursively scan through the parser tree that we have built before. For each parser rule, we will try to find paths leading to some end rule through its alternatives:

- Whenever we find an alternative that contains only one element, and this element is a reference to another parser rule, we have found an intermediary level that can be transparently hidden from the user of the language. We will continue recursively processing alternatives of this “level” rule (we are not at the end of the chain yet).
- Otherwise, we have found an end rule (recursion stops here).

We have expressed this algorithm using a pseudo-code:

```
FindPathsToEndNodes(R):
1)  Define L as an empty list of list of nodes
2)  Return FindPathsToEndNodes(R, L)

FindPathsToEndNodes(R, L):
3)  Define Q as list of list of nodes
4)  For each alternative A of rule R:
5)      L1 = Clone(L)
6)      If A is a parser rule with only one element E:
7)          Let I be interface/concept representing rule E
8)          L1.Add(I)
9)          P = FindPathsToEndNodes(E, L1)
10)         Q = Merge(Q, P)
11)     Else
12)         L1.Add(R)
13)         Let P be concept representing A
14)         L1.Add(P)
15)         Q.Add(L1)
16) Return Q
```

By appending the rule that is leading to current element (line 12) and then appending that alternative’s element itself (line 14), we will get a path that contains the full path and the target end rule as the last element of the chain. The result of this algorithm, for example for the content rule, equals the listing of the five paths mentioned in the beginning of this Section.

We can do this for all parser rules and for each rule we will get a list of paths that lead from that particular rule to an end node. We will call these paths **shortcuts**, as they can provide a shortcut from the rule to the end of the chain. Now that we have these, we will talk about several ways, how to use them to make our imported language better.

6.4.4.2 Smart Auto-completion

The first attempt on how to use shortcuts (the result of the algorithm in Section 6.4.4.1) was built on top of the straightforward approach, described in Section 6.4.3. There were no structural changes when it came to interfaces or linking concepts together. We imported everything just the same and then added some more functionality.

We were trying to solve the most obvious problem in front of us — the auto-completion. We would like to improve it so that it offers us only end concepts. Luckily for us, MPS gives us the ability to create custom auto-complete menus and use those instead of the built-in one. This means that we are going to be able to construct our own menu containing only end nodes. Unfortunately, it requires us to implement some non-trivial mechanisms.

Defining the Auto-complete

The auto-complete menu is bound to a cell of the projectional editor containing a reference to one of the children of a concept. Let's say we are again talking about the concept that represents the element rule's first alternative. As described above earlier in the process (Section 6.3), we created:

- An interface IContent representing four different alternatives that the content rule can break into
- A child of the Element_1 concept that references the IContent rule

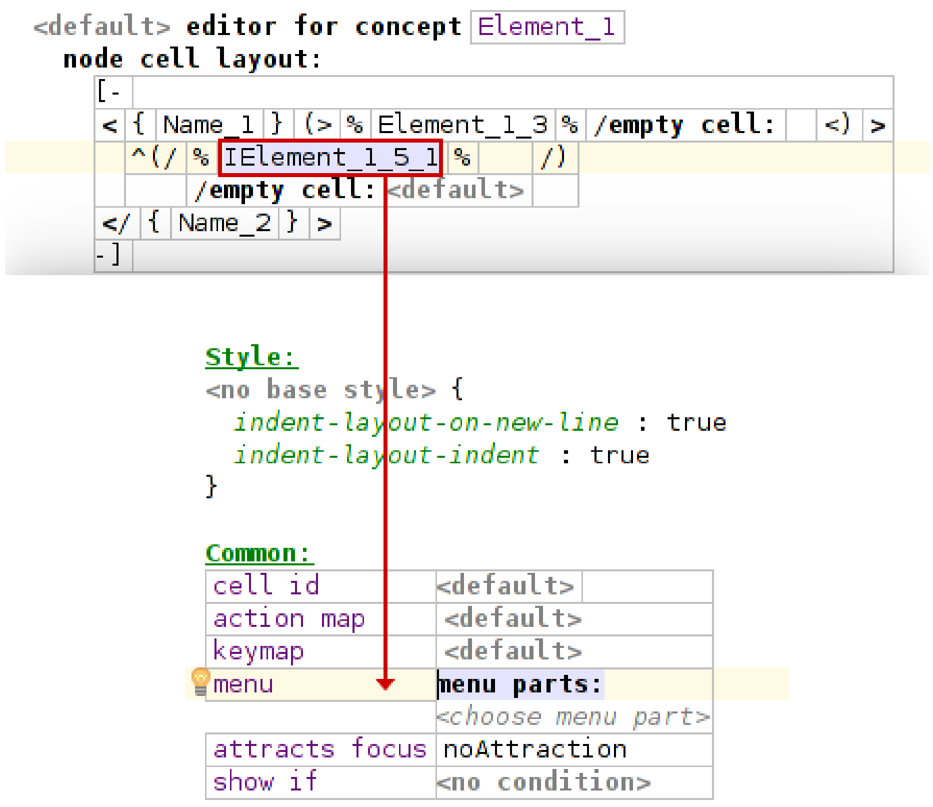
Somewhere in the editor aspect of the Element_1 concept, there will be a cell referencing the concept interface IContent. The cell, together with the auto-complete property can be seen in Figure 6.5.

We would like to create an auto-complete for this cell that would contain following options (end concepts):

- Content_1
- Element_1
- Element_2
- Comment
- Content_4

Because we are inside MPS everything is a concept node. In order to create an auto-complete menu, we need to create some BaseLanguage node and put it inside some AST (referencing the menu). More precisely, we will create an instance of a concept called `CellMenuPart_ReplaceChild_CustomActionConcept`, contained in one of many MPS's core languages.

For this concept, we will define a number of auto-complete options, one for each end concept that we want to offer in the menu. Every option has a **textual description**, **matching text** (so that we can filter through them) and most



```
<default> editor for concept Element_1
  node cell layout:
    [-
    < { Name_1 } (> % Element_1_3 % /empty cell: <) >
      ^(/ % IElement_1_5_1 % /)
        /empty cell: <default>
    </ { Name_2 } >
    -]

Style:
<no base style> {
  indent-layout-on-new-line : true
  indent-layout-indent : true
}

Common:
```

cell id	<default>
action map	<default>
keymap	<default>
menu	menu parts:
	<choose menu part>
attracts focus	noAttraction
show if	<no condition>

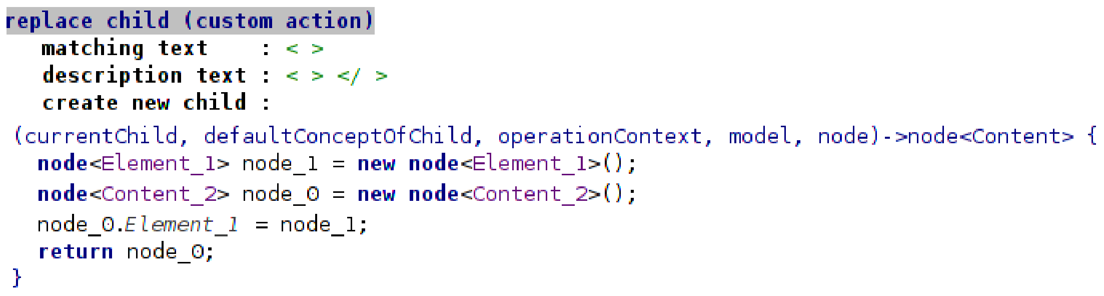
Figure 6.5: Auto-complete property

importantly a **creator method**. If the user selects that particular option, this method will be called with some contextual information in parameters and it is expected to return an instance of some concept that implements the IContent interface.

The complicated part of the process comes now, as we would like to dynamically generate the creator method. We need to create BaseLanguage statements that will instantiate the end concept and return it. There is one small problem, though. Let's say the user selected the second option and decided to insert another Element_1 inside. The problem is that Element_1 doesn't implement the IContent interface. It only implements the IElement interface as per the algorithm of the straightforward approach (Section 6.4.3.1). The shortcut that leads to this end concept leads through the Content_2 concept that has an Element child. This means that we have to follow the whole path and chain individual nodes, beginning with the Content_2 concept. For this particular case it would mean to:

- Create an Element_1 concept and store it in a variable.
- Create a Content_2 concept and store it in a variable.
- Assign the Element_1 node to the right child of the Content_2 node.
- Return Content_2 node.

As we said earlier in Section 6.2, generating BaseLanguage code is a bit more complicated because we need to either use a quotation or create a large number of AST nodes. Using a quotation, in this case, was sometimes impossible as everything is very dynamic. The finished option concept is shown in Figure 6.6.



```
replace child (custom action)
   matching text     : < >
   description text : < > </ >
   create new child :
(currentChild, defaultConceptOfChild, operationContext, model, node)->node<Content> {
  node<Element_1> node_1 = new node<Element_1>();
  node<Content_2> node_0 = new node<Content_2>();
  node_0.Element_1 = node_1;
  return node_0;
}
```

Figure 6.6: Auto-complete action code (Element_1 concept)

For the description text we used concept's alias. We talk about creating aliases in Section 6.4.2. For matching text we used the shortest unique prefix of alias among all other options' aliases in given auto-complete menu. The algorithm for searching shortest unique prefixes in a set of strings is not important from our thesis' point of view, so we decided to not go into further detail.

The Layer Problem Again

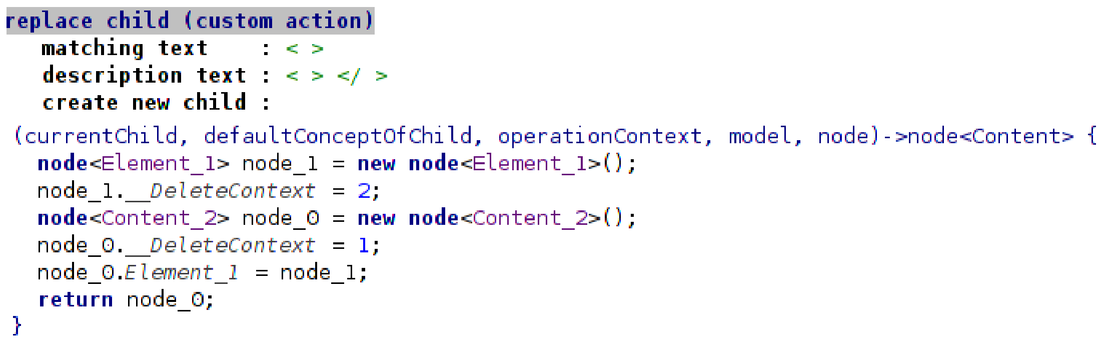
After we implemented the auto-complete menu, coding in the imported language started to be very reasonable and comfortable, since we eradicated intermediate layers. Or did we? When producing code, the language really does what one would expect. However, when the user starts deleting code, the layer problem occurs once more.

What happens here is that when we select the auto-complete option, several layers of concepts might get created and inserted into the cell. Imagine, we just inserted the Element_1 node. According to the creator method, this new node is wrapped inside of the Content_2 node. Now let's say, the user changed his mind, and wants to replace this XML element with a comment. He would press backspace, the XML element would disappear as expected. Then, the user continues by invoking the auto-complete menu again, expecting to have all five options at hand. What happened instead is that the Element_1 node got deleted, but the wrapping Content_2 one remained. The Content_2 concept has only one child of type Element and that is why our user only sees Element_1 and Element_2 as options in the auto-complete.

The Deletion Context

Solution to this resurrected layer problem lies in controlling the deletion event. Once again, MPS' authors have equipped us with tools for doing this. We are able to specify our own handler for the deletion event for any cell of the projectional editor. But what will it look like?

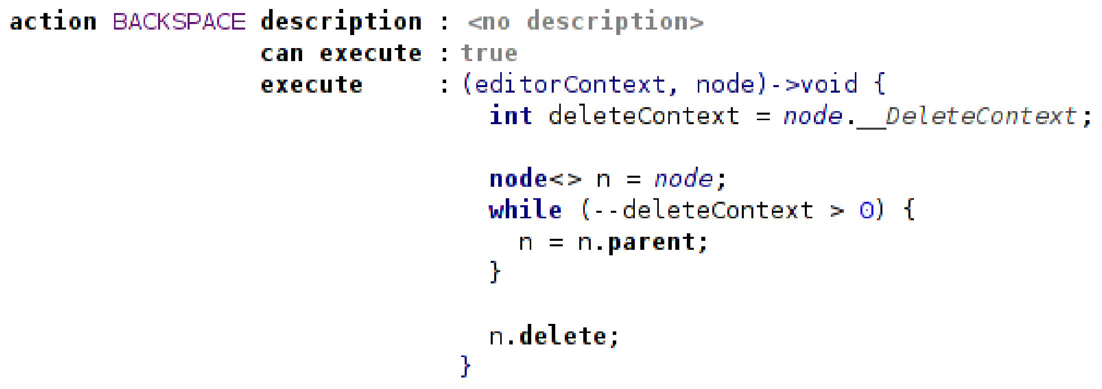
When deleting a node, we would like to remove the whole shortcut path, effectively reversing the effect of the creator method. We decided that it will be the easiest to store the length of the path that is leading to certain AST node, together with the node. So when we are creating nodes, while building the path in the creator method, we tell each node, how deep or far on the path the node is. In order to do this, we created a BaseConcept, a parent abstract concept, from which we will inherit all other concepts. This abstract concept will define a special integer property that will hold our information, effectively making all other concepts inherit it too. We called this property *__DeleteContext* and enhanced the creator method as shown in Figure 6.7.



```
replace child (custom action)
   matching text    : < >
   description text : < > </ >
   create new child :
 (currentChild, defaultConceptOfChild, operationContext, model, node)->node<Content> {
   node<Element_1> node_1 = new node<Element_1>();
   node_1.__DeleteContext  = 2;
   node<Content_2> node_0 = new node<Content_2>();
   node_0.__DeleteContext  = 1;
   node_0.Element_1  = node_1;
   return node_0;
 }
```

Figure 6.7: Auto-complete action extended with deletion context

The last thing remaining, is creating the backspace action. Apart from some problems with referencing the *__DeleteContext* property, which must be reached through the abstract BaseConcept type, it is quite straightforward to generate a BaseLanguage code like shown below (Figure 6.8).



```
action BACKSPACE description : <no description>
                 can execute : true
                 execute     : (editorContext, node)->void {
                                 int deleteContext = node.__DeleteContext;

                                 node<> n = node;
                                 while (--deleteContext > 0) {
                                   n = n.parent;
                                 }

                                 n.delete;
                               }
```

Figure 6.8: Backspace action implementation

We must not forget to pin this handler to every cell that we pin the auto-complete menu to. After we had done this, the layer problem has finally been eradicated. The language became a bit more usable once more.

6.4.4.3 Smart Interfaces

After we have implemented the smart auto-complete, we realized that there might be another way to accomplish almost the same result. It would mean changing the first step concerning interfaces and concept linking.

The main idea behind this is the realization that if for each concept interface there exists a finite set of end concepts, we could just create a special interface and let all these and only these end concepts implement it. It is important that other non-end concepts that are in the shortcut path, will not implement it. Otherwise that would put us right back where we started with the straightforward approach (Section 6.4.3) and the layer problem (Section 6.4.3.2). To give an example regarding our content rule, we would create an IContent interface concept and only following concepts would implement it:

- Content_1
- Element_1
- Element_2
- Comment
- Content_4

There would be no need for the Content_2 concept at all. Concepts would, of course, implement as many interfaces as many shortcuts lead to them. The difference is that the intermediary layers will not, which will prevent the layering.

We still need to find end concepts for each rule, so we will keep the same algorithm for finding shortcuts as before. We, however, do not need to implement our own auto-completion and, subsequently with it, no deletion handlers. The built-in default auto-completion will start to behave exactly the same way our smart auto-completion does.

Cardinality Restriction

So far it looks like the last approach is superior to the auto-completion one. It avoids complicated code generation and even resulting MPS languages are more simple and better performant, as they are not bloated with auto-completion code. There is one small drawback, though that makes this solution slightly suboptimal.

To remind the reader — the cardinality of an element is telling us, how many of each specific child rules can occur inside an alternative. In ANTLR, this was expressed using quantification operators (*,+ and ?). The problem that appears here, can be shown using this modified rule:

```
content       :   element+
              |   comment*
              |   CDATA?
              ;
```

Notice, how we changed the cardinality of elements inside the content rule. This small change will prevent us from creating a shortcut leading through this rule. We are trying to say that shortcuts, that can be used for the smart interfaces approach, can only lead through rules that have cardinality [1..1]. This means through an alternative with a single child element that is a reference to a parser rule AND has the [1..1] cardinality. In the original content rule, all alternatives look like this, so there was no problem. Speaking in terms of the shortcut algorithm (Section 6.4.4.1), we would alter line number 6 and add this restriction to the condition.

Now we will try to explain, why is this restriction necessary. It is caused by the fact that we would be unable to control changing children cardinality on the whole shortcut path. The list of shortcuts that will be generated for this setup is following:

- Content_1+ → Element_1
- Content_1+ → Element_2
- Comment*
- Content_3?

For the content rule, we would create an IContent interface concept. Imagine that some alternative of some other rule contains a reference to this altered content rule, let's say with the [1..1] cardinality. This other rule's projectional editor would need to have a child link holding the IContent type with cardinality [1..1] (one child). The problem is that the cardinality changes inside the shortcut path — each alternative of the altered content rule requires a different number of children to be inserted (e.g. the first alternative has element+). This means that when we would follow the shortcut path and assign the IContent interface to the Element_1 and to the Element_2 concept, we will lose the [1..n] cardinality information since we would only enable the user to insert one child of the IContent child. This conflict is caused, because we created an IContent child link with cardinality [1..1], but the grammar says, there could be more elements inside this child. If we disregarded the cardinality restriction, the child link could ultimately hold only one Element_1 or Element_2 concept. But the content rule says, there can be more ([1..n]) element rules.

The smart auto-completion approach mentioned above could solve this problem by generating lists of children with the corresponding cardinality.

6.4.5 Our Solution

In the end, we have decided to go with the shortcut approach using smart interfaces. As we have stated above, it has several advantages, concerning the solution complexity, compared to the smart auto-completion. It only has one disadvantage regarding the cardinality restriction, described in Section 6.4.4.3. We have analyzed several ANTLR grammars (such as the JavaScript [12]) and we haven't found a single intermediate rule that would be suffering from this restriction. It is quite understandable if you think about how grammars are written. Usually, you create the basic structure (i.e. different kinds of statements) in the simplest way possible and put the quantitative operator rather to an alternative that is referencing the structure.

Based on these observations and some grammar analysis, we concluded that advantages prevail and used the last mentioned approach.

6.4.6 The Structure Aspect and Other ANTLRv4 Features

The ANTLR grammar notation offers more features than just rule definition. We haven't mentioned these earlier because we were focusing solely on the structure of the language. We will mention some features and explain why we have ignored them and why they don't matter to us. We will not spend a lot of time on describing details of these as they are well documented in the official ANTLR reference [7].

6.4.6.1 Modes

For example, just like any other parser/lexer, ANTLR gives us the possibility to switch the parsing context. We are able to create various user-specified modes and then enter these modes when certain rules/tokens are encountered. For each mode, we can define a different set of rules/tokens that can only be applied when the parser is in that particular mode. The syntax is following:

```
// Enter mode when tag opened
OPEN        :   '<'       -> pushMode(INSIDE) ;

mode INSIDE;
// Special rules bound to specific mode
S           :   [ \t\r\n] -> skip ;

// Leave the mode when tag closed
CLOSE       :   '>'       -> popMode ;
SLASH_CLOSE :   '/>'      -> popMode ;
```

We didn't pay any extra attention to modes while dealing with the structural aspect because they don't really influence contents of individual concepts. It is even possible to define concepts that control mode switching and then not including them inside any parser rule. They still get recognized by the lexer and mode is changed.

The reason that this goes beyond our interest here is partially caused by the fact that modes are used on runtime when we are parsing actual code, whereas language structure is more of a static matter. The only time we care about modes is when generating the TextGen aspect, and we will get back to it later in Section 6.6.

6.4.6.2 Actions, Attributes and Semantic Predicates

Actions allow us to append code to rules and this code is then executed every time the parser applies this rule. The code is written in the target language that you are creating the parser for. It is then copied as a string and inserted into the method that is bound to parsing this rule. Again, this is of a little interest to us. Usually, this is used when creating some specific parsers for some particular scenario. We, however, are expecting to parse general purpose languages that contain actions just rarely. And even when they did, we cannot be sure what language will it be in and how to use it.

Attributes allow us to extend some basic predefined set of properties of each rule. We can store some arbitrary information there and later access it for example inside actions using special syntax. From exactly the same reasons as with actions, attributes are of no interest to us.

Semantic predicates tell the parser on runtime, which rules can be applied depending on specified constraints. This is also a runtime matter.

6.5 The Editor Aspect

After we have imported all concepts, their contents (properties) and linked them together (child links), it is time to define, what is the visual representation of these concepts. Without this, we are not able to start using the language inside MPS.

As stated before (Section 2.4.2), MPS uses a cellular system that allows placing concept's properties and children into a table-like arrangement. MPS has a lot of different types of cells that we can use:

- Cells for storing values of properties – the user can enter text inside, which is validated using the type of the property (think XML tag name).
- Cells for storing child concepts – here we can store other parser rule references and build the AST further.
- Cells that have static fixed content, such as constant keyword – we will use these to display literal elements.
- Cells that influence the layout, such as indenting.

Our import plugin has to create these cells and ideally project all of the concept's elements in there.

A part of this thesis' mission was to explore, whether we can also bring some more value into this import step. The problem with grammars is that it serves us no aid when it comes to element layout. The grammar only defines, what the rule breaks up into and which elements (rule references, literals..) are contained inside of each alternative. Since the layout information is missing, we have only two options, how to tackle this problem:

- Get this information from the user by prompting for it somehow.
- Keep everything automatized and generate the information using some heuristic.

It is a very hard problem, though since our plugin doesn't really understand the contents of the grammar on some higher level.

6.5.1 The Interactive Approach

Since the layout information is just missing, we decided to ask the user for it. The first idea on how to tackle this problem was to interactively prompt the user during the import process. We would somehow select rules that we consider important, and give the user several visual options on how we think this rule might be laid out. The user would pick one and we would use this information to create the editor aspect. There are some problems with this, though that led to rejecting this approach.

6.5.1.1 Detecting Interesting Rules

Firstly, we would have to be able to tell, which rules might be worth "discussing" with the user.

The first idea for a heuristic indicating these "interesting" rules was based on a number of elements contained in rule's alternative. For simple rules, which there usually is a big number present in the grammar, we would skip them. For complex rules (let's say 5 elements and more), we would ask the user for help with the layout. The heuristic isn't bad, it would detect complicated rules, such as cycles, branching commands and so on, so it might be a sufficient, yet simple, solution. It, however, has some problems — rules describing a block of statements are usually very simple, example given being the JavaScript one:

```
statementList : statement* ;
```

This rule would be skipped by our heuristics since the alternative has only one element, but in most general purpose languages, this is exactly the kind of statement that we would like to adjust, since normally we put each statement on a separate line. There are dozens of rules that have this form (method parameters, operands, ...), and we have no means of recognizing that this particular one should be a vertical list (meaning its elements should be separated by a new line).

Another heuristic suggestion was detecting pair symbols among alternative's literal elements. Usually, characters such as braces are a good indicator of some indenting. We, however, did not implement any of these heuristics, because, in the end, we decided to abandon the interactive approach completely, as described below in Section 6.5.1.3. That is why we will not go into more detail here.

6.5.1.2 Fixing Rule Layout

Once we have detected a rule that might have some interesting layout, we would ask the user to help us with adjusting it. Consider, for example, the first alternative of the element rule from the XML language:

```
element   :   '<' Name attribute* '>' content* '</' Name '>' ;
```

Let's say, we would prepare few versions of how we think the layout could look like. We would present these options to the user and let them choose. We could ask the user, whether attributes should be spread out horizontally or each on a new line. We could and probably would have to do this for each element since the plugin has no real understanding of the content. We would have to ask for indentation, line breaks etc. because if we were to guess, we would probably guess wrong since there are just too many options. The user would probably end up finalizing the layout himself.

The other option could be in form of some sophisticated smart dialog, where the user would control the layout by dragging the elements around or position them through some text field. This would be probably better, but we would be reimplementing already existing functionality that is already present inside MPS. Again, we are not going into any more detail here, as this approach was rejected because of reasons mentioned below.

6.5.1.3 Approach Evaluation

The author of this thesis came to a conclusion that the results of the interactive approach would be most likely quite suboptimal. It would be hard to recognize rules that are in need of a refactoring. Furthermore, when asking for user's help, we would just duplicate the functionality of MPS's built-in projectional editor. We would hardly mimic all of its functionality and put a lot of effort into something already existent. Moreover, our end user is expected to have knowledge of the MPS editor, since he is importing language there. This means that he probably knows his way around the projectional editor too. It wouldn't make sense to force the user to learn to work with our own interactive dialog, while in the background, this dialog would be just translating the layout back into the terms of the projectional editor. Implementing such mechanism would also probably be very complicated.

To put it a bit differently — we wouldn't be sparing the user from any manual work, we would be only changing the environment, where this work happens. We would be shifting it from the MPS projectional editor designer into our interactive dialog. We would be shifting it from a fully featured MPS environment, the user probably already knows, into a feature-wise way poorer environment, the user sees for the first time. We would be doing this shift for a price of reimplementing already existing mechanisms. We would also need to decide, where to draw a line and which layout features we will cover in our interactive dialog (say line breaks and indenting) and which features we would leave out (code colouring, spacing..). If we didn't draw this line, we would end up reimplementing the whole MPS projectional editor designer. Furthermore, for every feature that we would choose to include, we would have to implement special behaviour inside the interactive dialog and then another logic that would translate settings from this dialog into the editor aspect. And even if we accomplished everything mentioned above, we still believe (and have confirmed that by experimenting) that the language would still need to be adjusted inside the projectional editor designer.

From reasons stated above, we rejected the interactive approach. From exactly the same reasons, we concluded that any approach that would require user input, would be suboptimal to just leaving the user adjusting the layout inside the MPS editor designer.

6.5.2 The Learning Approach

Since we have rejected user-input based approaches, we are left with heuristics. The second approach is more complex and might yield better results. We have, however, not implemented it to a stage that would be presentable, as we met some obstacles. We would like to describe it anyway, so possible follow-up work might take it into consideration.

6.5.2.1 Approach Principle

The learning approach would require the user to, together with the grammar file, supply a set of valid source files written in the imported language.

1. The plugin would automatically generate an ANTLR parser for this language using the ANTLR library, which provides this functionality.
2. It would alter the code of the generated parser and add some additional functionality, described below.
3. It would compile the parser into an executable form.
4. It would use the parser to parse the supplied source code and extract information about the layout of the code.

The benefit of this approach is that the imported language would inherit its code style from the user, as it would learn directly from his code. The quality of the extracted information would depend on the amount of the code supplied. So far, this approach sounds very complicated — extracting layout information sounds like a difficult problem on its own. We have, however, found a simple way, how to mine information very efficiently.

When the ANTLR parser is parsing the code, in its first stage, it reads the input and tries to split it into tokens. In the second stage, it tries to map the token stream onto parser rules of the language and build an AST. Earlier, in step 2, we said that we would change the code of the parser. We would make sure that each parsed token would remember additional information — the number of the line it appeared on. This is an easily accessible property of the parser and the ANTLR framework gives us a lot of room for an adjustment like this.

For example, for our element rule

```
element  :   '<' Name attribute* '>' content* '</' Name '>' ;
```

and following source code,

```
<div>
    ... some content ...
</div>
```

it would yield something like this:

```
TOKEN LINE
----------
<       1
Name    1
>       1
...
// Here, tokens of the content would come
...
</      3
Name    3
>       3
```

From this, we could clearly derive that there is a line break between the first closing *greater-than* bracket, the content and then again when the closing tag starts. Probably, we might also be able to detect indentation, if we would decide to store the column information too, but we haven't explored this possibility.

6.5.2.2 Approach Evaluation

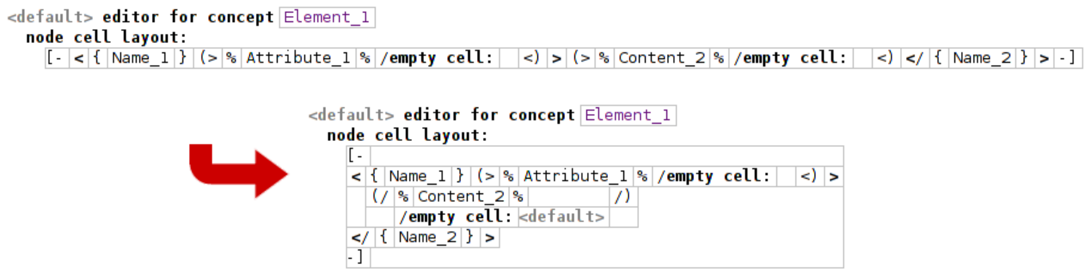
We have started implementing a proof of concept for this approach, so that we can see, whether it is a viable solution. Unfortunately, there were some obstacles, due to which we haven't finished the implementation. The biggest problem that we have identified, was parser generation. We found out that even a small tweaks in the grammar that the user might perform in order to improve the resulting MPS language, can break it enough so that the automatically generated ANTLR parser is not parsing the code correctly. We talk further about this problem in more detail in Chapter 9.4.

Other problems that we have identified, were connected to the environment, where it might not be always possible, to perform actions such as parser generation, parser compilation or dynamic loading of the parser. More problems concerned the algorithm itself, where we would need to be able to match language concepts with all tokens that belong to it. Mapping the parsed ANTLR AST to the MPS one is a complex problem that is also connected to parsing any given text source code and importing it inside MPS. This is considered as an advanced functionality that will probably be subject to a separate follow-up work.

Because of the overall complexity of this approach and because of reasons stated in Section 6.5.3, we have decided to abandon the implementation and only suggest it as a possible follow-up. We, however, concluded that this might be a viable solution for detecting code layout.

6.5.3 Our Solution

When the author of this thesis worked on the plugin, he noticed that the most tiresome and likely error-prone part of working on the editor aspect is incorporating all concept's properties, children, and constant fields into the editor. This means a manual creation of all cells that should appear in the visual representation. We identified this phase as a time consuming but also quite straightforward, as it does not require any major thought on user's part. We have also further noticed that when the plugin would do this heavy lifting, even without layout detection, further adjustments of the editor are very fast since MPS enables doing this very efficiently. We noticed that even the most obvious way of editor generation, such as plain inlining of all of the cells in a single row (shown in Figure 6.9 on top), might be a sufficient solution. In Figure 6.9, we can see an editor adjustment of the Element_1 concept that can be done in just a few clicks, but finalizes the editor to its perfect form, better than any heuristics would come up with.



```
<default> editor for concept Element_1
  node cell layout:
    [- < { Name_1 } (> % Attribute_1 % /empty cell:   <) > (> % Content_2 % /empty cell:   <) </ { Name_2 } > -]
```

```
<default> editor for concept Element_1
  node cell layout:
    [-
    < { Name_1 } (> % Attribute_1 % /empty cell:   <) >
      (/ % Content_2 %          /)
         /empty cell: <default>
    </ { Name_2 } >
    -]
```

Figure 6.9: Projectional editor adjustment of the Element_1 concept

From these reasons, we concluded that for our cause, it might be sufficient, if the plugin only prepared contents of all editor aspects and the end user would take it from there, reaching optimal results in a very short time. We have confirmed this assumption by importing the JavaScript language [12] and manually adjusting all editors that needed it, in a less than hour time. We have described this further in Chapter 8. Further improvements might be a subject for a follow-up work, potentially leveraging the second approach, we have described above in Section 6.5.2.

6.6 The TextGen Aspect

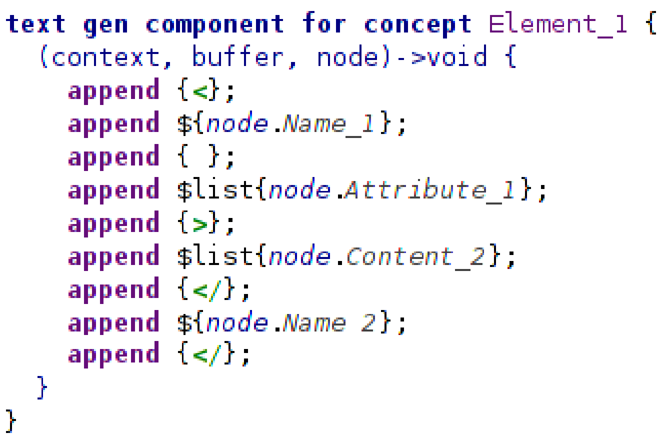
TextGen aspect's purpose is telling each concept, what the real text code generated out of it will be, once we want to turn our MPS code into a real program. Basically, it's a single method definition for each concept of the language. This method has several parameters, such as the currently processed node and some contextual information. It is, however, not returning a string as perhaps expected, but rather manipulates an output buffer/stream using some built-in functions. When generating the code of a program, MPS calls this method for the root concept of the program. It is up to this concept's TextGen method to append its children into the stream by invoking their TextGen methods subsequently.

6.6.1 Our Goal

What we would like to do here, is to create a TextGen aspect for each concept and generate some BaseLanguage code for its TextGen method. The TextGen method is, again, an AST built out of concept nodes as described in Section 6.2. We need to create a root node of the method and to its body child add a list of BaseLanguage statement nodes that represent the code of the method. Let's illustrate this using our SimpleXML example, namely the Element_1 concept that represents the full XML tag with content and is given by the first alternative of the element rule:

```
element    :   '<' Name attribute* '>' content* '</' Name '>'
           |   '<' Name attribute* '/>'
           ;
```

A very basic example of a BaseLanguage code that we would like to generate for this concept, is shown in Figure 6.10.



```
text gen component for concept Element_1 {
  (context, buffer, node)->void {
    append {<};
    append ${node.Name_1};
    append { };
    append $list{node.Attribute_1};
    append {>};
    append $list{node.Content_2};
    append {</};
    append ${node.Name 2};
    append {</};
  }
}
```

Figure 6.10: Example of a TextGen aspect

We can see that we gradually append all literals, properties, and children in the same order as they appear in the grammar definition. This part of the process holds no first-hand complications. We still need to generate BaseLanguage code dynamically, which in some cases might be a challenge. We figured how to overcome those challenges earlier in Section 6.2. There are, however, some hidden problems along the path.

6.6.2 Whitespaces

As you might have noticed, there was no whitespace handling inside of our SimpleXML grammar. For example in the element rule, there is Name the element directly followed by attribute*, but in the XML language we require at least one space as a separator. So how does the parser know how to split tokens, how many whitespace characters are expected, and which characters are whitespace in the first place?

- How does our TextGen generator know that there should be a space in between attribute and Name but is not required between '<' and Name?
- How does it know there is no space between quotes and an attribute value inside the attribute rule?
- And how about multiple attributes — do they need to be separated by spaces?

6.6.2.1 Whitespace Handling in ANTLR

We have left the whitespace handling part out earlier in order to keep things more simple as we were paying attention to the structure of the language in the first place. There are two ways, how we can deal with whitespaces in ANTLR:

- We can define special tokens that don't need to be used inside any parser rule. We mark them with special flag, telling the parser to skip these characters when this token is matched:

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

- The second possibility is to create a similar rule to the one above, but leaving out the skip flag. Then, we explicitly reference this token in every parser rule, where whitespace can occur. This sometimes means hundreds of different positions, so it is generally not a recommended approach. There are however cases, where this is inevitable due to the nature of the ANTLR lexer.

6.6.2.2 The Whitespace Problem

The problem that arises here, is that the skip syntax hides the whitespace information from the grammar. We have no way of tracing this information back. If, on the other hand, we decided to go with the explicit definition, and put WHITESPACE tokens everywhere, we would encounter much bigger problem. From the point of view of the TextGen aspect, it would help us a bit, but not completely, as we still don't know what string represents the rule. Secondly, when creating the structural aspect, we would create a property for each of these token references as per Section 6.4.2. The effect of that would be that the imported MPS language will become unusable. The code will be full of placeholders waiting for whitespace characters (even though they might be unnecessary, i.e. [0..n]).

We could try harder and ask the user himself, whether there are any whitespace detecting rules or try detecting them ourselves. But even this will not solve the problem, as the whitespace can be easily entangled inside non-whitespace tokens as shown below with the chardata rule.

If we look into existing ANTLR grammars, we can see that things get even more complicated. ANTLR, just like as any other parser/lexer, gives us the possibility to switch the parsing context. We are able to enter any user specified mode and then apply a different set of rules depending on current mode, which gives us more flexibility. This means we can handle whitespace both implicitly and explicitly at the same time inside one grammar based on the mode. Secondly, there is the possibility of capturing whitespace together with other characters. Consider the following excerpt from the official ANTLR XML grammar:

```
element     :   '<' Name attribute* '>' content* '</' Name '>'
            |   '<' Name attribute* '/>'
            ;

content     :   chardata?
                ((element? | CDATA | COMMENT) chardata?)* ;

chardata    :   TEXT | SEA_WS ;

TEXT        :   ~[<&]+ ;

SEA_WS      :   (' '|'\t'|'\r'? '\n')+ ;

OPEN        :   '<'              -> pushMode(INSIDE) ;

mode INSIDE;
S           :   [ \t\r\n]        -> skip ;
CLOSE       :   '<'              -> popMode ;
SLASH_CLOSE :   '/>'             -> popMode ;
```

What is happening here, is that the author decided to skip whitespace characters only when we are inside of XML tags (see that the element contains no explicit whitespace tokens). On the outside, they are being cleverly swallowed by the chardata rule (its second alternative) that wraps any content. The content rule is written a bit differently from our SimpleXML version. It is a little bit more complicated, but the shorter structure, showing all the different ways we can express the grammar in.

The problem we find here is that with all the features ANTLR has to offer, there are just too many ways, how we can handle whitespaces and they can get all mixed up together just like the previous example showed. In the worst scenario, from the TextGen point of view, the language is skipping all whitespace characters silently and thus removing the information from the grammar completely.

This leaves us with the original question and the root problem:
How does our TextGen aspect know between which children there must be a whitespace and where it is forbidden so that the produced code is a valid one?

6.6.2.3 Possible Improvements

Following up on the question posed in the previous paragraph, we can ask ourselves a different question. If not our TextGen, how does the ANTLR parser know where whitespaces are due? Surely the parser must know, otherwise, it wouldn't be able to tell wrong code from the right. Well, the answer is that it knows once it starts parsing some code depending on the inner state of the parser. Context modes make this quite simple. They are quite easy to work with on runtime when we are actually parsing some source code, but it is impossible to determine the mode for rules statically. In other words, it is impossible to detect, whether we are swallowing whitespaces silently for given rule or not. This is caused by the possibility to jump between modes in almost any manner and therefore use one parser rule in different contexts.

We do not, however, necessarily need to put different amounts of different whitespace characters in random places. We only need to find places, where we are sure they must be at in order to make the code valid. Then, we can put a single space there. Generally, it doesn't have to be a space character, but since we are aiming for general purpose languages, we will go with the very frequently used space character. We could get some hints from the way the lexer is splitting source code into tokens, which is happening practically statically too. Consider the following setup:

```
element    :   '<' Name attribute* '>' content* '</' Name '>' ;

attribute  :   Name '="' TEXT '"' ;

Name       :   [:a-zA-Z]([:a-zA-Z]|-|_|\.|[0-9])* ;

TEXT       :   ~[<"]* ;
```

From the general knowledge of XML, we know that there is no space needed between the opening bracket of an XML tag and the name of the tag (i.e. **<img**). On the other hand, there has to be one between the name of the tag and the name of the first attribute (i.e. **<img src=...**). But why is that? Simply, because the lexer needs to parse the input source into tokens using available lexer rules (regular expressions). If it encounters a whitespace, it will stop parsing previous token (unless there is space allowed in it), parse the whitespace one (throw it away when skipping) and continue with the next one.

The reason the parser cannot differentiate tag name from attribute name without a separator is caused by their regular expressions. Consider a string **A**, representing the tag name (a string that can be matched by the regular expression of the Name rule). Let string **B** be a string representing the attribute name. Whenever **A** and **B** share some same characters, we cannot clearly state, where one ends and the other starts (when placed together). More precisely, **A** can end with same characters as **B** can start with. Even more precisely, some suffix of **A** can be some prefix of **B**. In our case they are coincidently both given by the Name rule, but in general case we really care about any possible substrings.

Determining whether a suffix of a string matched by one regular expression can be a prefix of a string matched by a different regular expression is not an easy task. We could try to generate strings that satisfy both regular expressions, but probably the only complete solution to this problem would be to construct automata representing both expressions. Then we would need to examine them somehow, in order to detect similarity.

Now what happens, if one of our children that can have zero cardinality, is empty (no element attributes)? What are we going to do about whitespaces wrapping this child? In this case we should probably do the same suffix/prefix comparison with the next-next child on the way, skipping the empty one. Additionally let's not forget that all this logic needs to be contained in generated BaseLanguage code.

We have decided to settle for a more simple heuristics, which is described in Section 6.6.4.

6.6.3 Layout

We have already established that whitespaces are hard to get right. But how about the code layout itself? What if we want to produce more readable code by adding i.e. indentation or line breaks? Can we for example use the same information, we were mining when building the projectional editor? It would seem like the editor and code layout are very close to each other and since we were unable to resolve editor problems fully (Section 6.5), we won't be able to leverage the information here neither.

6.6.3.1 Universal TextGen

One idea, on how to approach layout inside the TextGen aspect, would be creating a really smart universal TextGen method. It would be static (always the same, independently of the grammar) and it would be the same for each concept of the language. It could read the projectional editor definition and then somehow try to mimic this layout on output. This might be quite straightforward, because each cell has its specific function that could be translated into a text layout. The added value of this approach would be that whenever user decides to adjust and improve the projectional editor, it would get automatically reflected inside this universal TextGen. It would also have a drawback that when user would want to change the TextGen definition for some concept, he would have to build it from scratch.

After this proposal was discussed with JetBrains, it was immediately rejected. The main reason for this being that aspects should be independent of each other. This means that when someone would like to introduce an extension of a language and change the editor, it might influence the TextGen in a way that, it would start generating invalid code (e.g. we could for example easily remove the "if" keyword from the projectional part). Sometimes we might also want to define more projectional editors and switch between them, which would also make this dysfunctional. This is probably the reason that JetBrains haven't introduced their own universal TextGen so far (just like they introduced a default projectional editor).

6.6.4 Our Solution

After the analysis, we have come to a conclusion that the problem of TextGen layout is quite similar to the one we have seen with the projectional editor, described in Chapter 6.5. Since we are mostly dealing with text-based languages, their editor representation must be almost the same as their expected text output. Once we would have some information about the layout for generating better projectional editor, we could leverage the same information and use it for TextGen improvement too. We have, however, decided that adjusting the editor manually, after the import is done, is for now the fastest and the most efficient way how to deal with this problem, so we have no information to leverage here.

Adjusting the layout, speaking in the terms of TextGen, means adding line breaks and indentation — practically the same as what we are doing with the editor aspect. Nonetheless, aside from layout, there is still the problem concerning whitespaces that we described above in Section 6.6.2. We decided not to go into full depth and solve the problem using regular expression automata. We have, however, tried to improve the situation using some really simple heuristics which gave surprisingly good results.

We have started with a very basic TextGen that would insert spaces in between every two elements of the concept. Then we started restricting spaces on positions where we thought they are not needed.

1. We have concluded that whenever there is a literal, it is a plain string token defined in the grammar that might, in most cases, get recognized by the parser safely without the need for a whitespace separator around (think **'<'** in XML). So whenever there is a non-alphabetical literal, we omit spaces around it.

2. Next, we were looking at properties (non-literal, but regex lexer rules). These are strings inserted by the user of the language, whose form is constrained by a regular expression (think XML tag's name). If these are neighboring a literal rule, we look at that literal rule's content. We expect that the user will be inputting some alphabetical content (variable/method/class

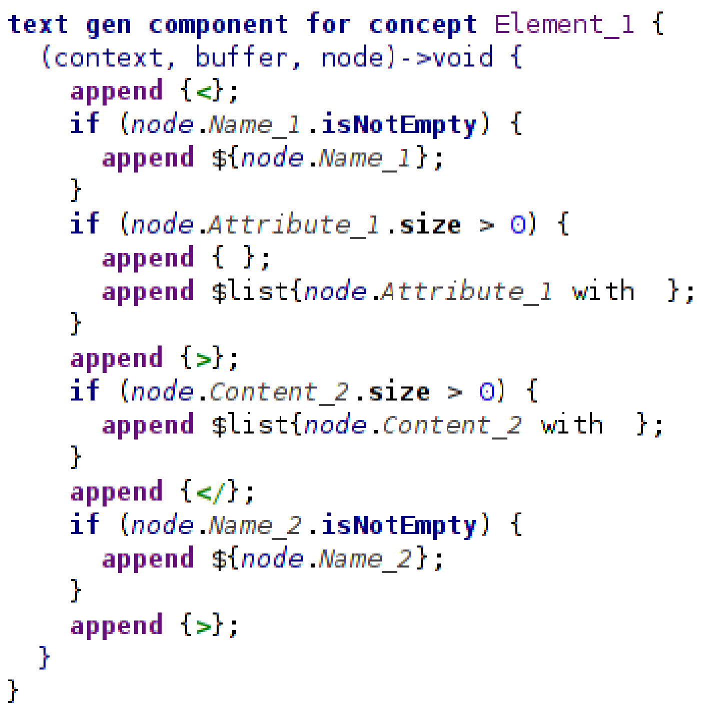
names, identificators, etc.). If the neighboring literal rule ends with an alphabetical character, we insert space, otherwise, we omit it. This will help in cases such as insides of quotes, next to semicolons or around brackets, but on the other hand, it will separate usual language keywords (function, var, in, etc.) from other content.

3. We check for element emptiness (child not present) and do not insert spaces when elements are empty so that spaces do not accumulate.

4. When two children concepts are next to each other, we always insert space.

5. Sequences of children are separated with space. This is the place where we might later want to manually substitute it with a line break. Think repeating content inside an XML tag or list of XML attributes.

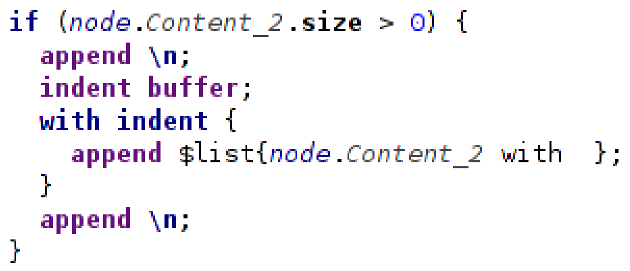
These few simple heuristics have left us with some very nice results (tested mostly on XML, JSON, and JavaScript). Figure 6.11 shows an example of a TextGen aspect for the full SimpleXML element.



```
text gen component for concept Element_1 {
  (context, buffer, node)->void {
    append {<};
    if (node.Name_1.isNotEmpty) {
      append ${node.Name_1};
    }
    if (node.Attribute_1.size > 0) {
      append { };
      append $list{node.Attribute_1 with  };
    }
    append {>};
    if (node.Content_2.size > 0) {
      append $list{node.Content_2 with  };
    }
    append {</};
    if (node.Name_2.isNotEmpty) {
      append ${node.Name_2};
    }
    append {>};
  }
}
```

Figure 6.11: Generated TextGen aspect for the SimpleXML element

If we wanted to manually adjust this generated code, so it generates nice indented XML code, we only need to wrap the Content_2 child with indentation and change the sequence separator to a new line character. This is a very fast and small adjustment, so we concluded that it is right to expect the user to do this kind of adjustments on his own. Resulting adjusted aspect is shown below in Figure 6.12.



```
if (node.Content_2.size > 0) {
  append \n;
  indent buffer;
  with indent {
    append $list{node.Content_2 with  };
  }
  append \n;
}
```

Figure 6.12: Adjusted indentation inside the TextGen aspect

7. Plugin Description

This chapter will give a brief overview of the architecture of the plugin and also a short user manual, describing the import process. The author of this thesis decided to name the plugin **Ingrid**, as a mix of an abbreviation of *Interactive Grammar Import* and a Scandinavian name of a Norse valkyrie, who was bringing dead warriors to Valhalla (as a parallel of bringing a language into MPS).

7.1 Developer's Manual

This section will describe the software architecture of the plugin. The plugin consists of several modules that are dependent on each other. We will describe the overall structure and then each one of these modules.

7.1.1 BaseLanguage and Java

As we have mentioned before in Section 6.2, whenever a user wants to add program parts of his language (or our plugin solution), the BaseLanguage is used. Furthermore, since the whole MPS is Java based and the BaseLanguage is a direct port of Java, JetBrains have made it possible to interconnect Java and BaseLanguage code seamlessly. The user is able to load Java classes (both as .java sources and compiled .class files) inside MPS and the other way around — BaseLanguage produces Java code when compiled into a text source code.

The BaseLanguage is handy when it comes to adjusting MPS languages because it offers a lot of syntactic sugaring and MPS specific extensions for working with languages. It, however, is not an ideal tool when it comes to maintaining larger code base such as the one we need to create for our plugin. Programming in BaseLanguage can be sometimes slow and cumbersome due to the nature of coding inside MPS. During the implementation of the plugin, the author was trying to keep as much code as possible in plain Java and maintain the project using the JetBrains IntelliJ IDEA[1] IDE, which is a free Java (mostly) IDE coming also from JetBrains. There exists an MPS plugin for IntelliJ IDEA and the two can interoperate, but we won't be using this feature.

On the other hand, the BaseLanguage is vital, when it comes to using the MPS language API. This API allows us to programmatically create new languages, concepts, and aspects. The Java code, produced from the BaseLanguage code that is using the API, is very complicated and not human readable. As stated above, the BaseLanguage contains many extensions that hide this complexity, and enable very effective usage of this API. This is the reason, why not all code of our plugin can be written in plain Java and why we needed to create a more complicated code structure.

[1]https://www.jetbrains.com/idea/

7.1.2 Plugin Architecture

We have decided to create one MPS project and one IntelliJ IDEA project. Both projects maintain the same structure, so it is very easy to find your way around the code base. Furthermore, both projects were split up into several modules and, apart from one library module, they were all developed inside the IDEA project in plain Java. We have captured the setup in Figure 7.1.

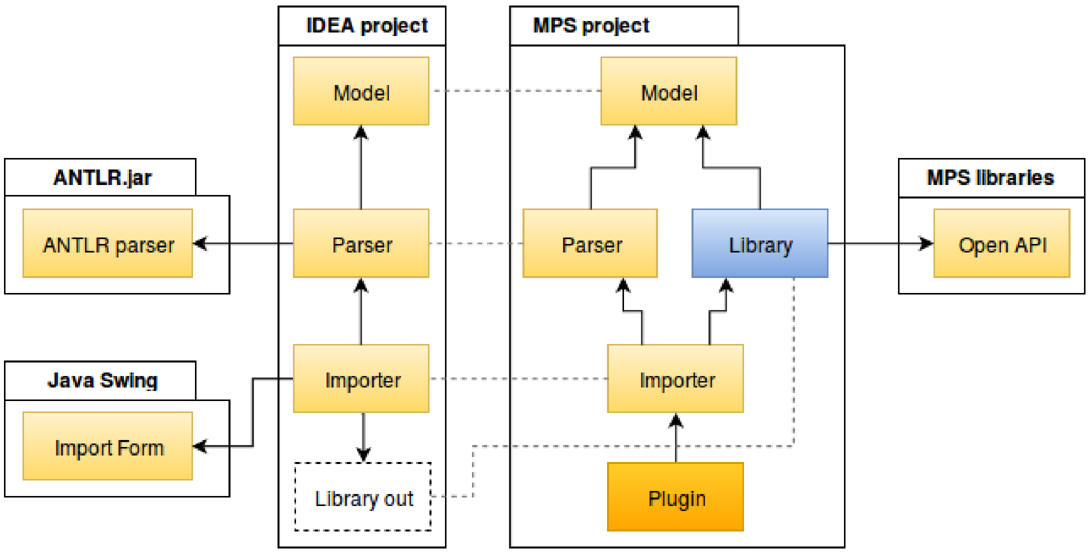


Figure 7.1: Module architecture of the plugin

Each colored box in Figure 7.1 represents a module (or a library). Black solid arrows represent dependency relationship between modules. This relationship is, of course, transitive, but inside MPS it means that all dependencies of dependencies need to be explicitly referenced as well. Using the gray dotted lines, we have expressed references between MPS modules and their IDEA counterparts (dotted gray lines). All MPS modules that are pictured as light yellow, do not contain any BaseLanguage code and only import Java code from the IDEA project into MPS.

Please note, that, from MPS, we are referencing the original Java source code, not compiled .class files. MPS is able to compile these Java sources itself, so for successful plugin compilation, IntelliJ IDEA is not needed at all. It is only used for developing the code, maintaining it, and running unit tests.

7.1.3 Module Description

This section will, with respect to Figure 7.1, describe each module of the plugin.

7.1.3.1 The Model Module

The Model module holds no logic, only a set of Rule classes that are used across the whole plugin. These are the classes, in which we represent the grammar after we parse the ANTLR AST.

The Java code of this module is referenced by the **premun.mps.ingrid.model** solution, which is the MPS counterpart of this module.

7.1.3.2 The Parser Module

Firstly, the Parser module contains a parser of the ANTLRv4 notation that was automatically generated using the ANTLR library. Then, together with the ANTLR library, it is able to read an ANTLRv4 grammar file and construct its AST. Secondly, the module contains some logic that will translate the ANTLR AST into our own data structure, using Rule classes from the Model module. During this process, it simplifies the structure, by performing actions described in Section 6.4.2 and throws away some parts of the ANTLRv4 grammar that we do not need, as described in 6.4.6.

The Java code of this module is referenced by the **premun.mps.ingrid.parser** solution, which is the MPS counterpart of this module.

7.1.3.3 The Library Module

The Library module is not written and maintained in IDEA but is written in BaseLanguage and developed inside MPS. As stated above, usage of the MPS API is practically only possible using BaseLanguage. The Library module contains absolute minimum, necessary to access this API. Basically, it contains a set of helper classes that make an adapter (or a facade) of the MPS API for our Java code.

Upon compilation of the Library module inside MPS, Java code is produced (in Figure 7.1 labeled as "Library out"), but it is a computer generated and not human readable code. We can, however, call methods of this Java code from the Importer module, which is everything we need.

7.1.3.4 The Importer Module

The Importer module is by far the most interesting and complex one. It contains logic for generation of the final MPS language. This logic encompasses almost everything, we have talked about in Chapter 6. It uses the Parser module to mine the grammar representation and then, using helper classes from the Library module, it constructs all the elements (concepts and aspects) of the MPS language. The module also contains the initial import form that the user is shown at the beginning of the import process.

The import process is further divided into several import steps and that is also the way, the code of this module is organized. The Java code of this module is referenced by the **premun.mps.ingrid.importer** solution, which is the MPS counterpart of this module.

7.1.3.5 The Plugin Module

The Plugin module is the final piece that enables us to add a menu item inside MPS and run the code. It is a special kind of an MPS solution — a plugin solution — and can be found under the name **premun.mps.ingrid.plugin**. It adds a menu item called *Import ANTLRv4 Grammar* and places it inside the *Tools* menu. Upon clicking, it only calls the Importer module and begins the import process. It supplies the Import module with important handlers so that the module can later create language inside currently opened MPS project.

7.1.3.6 External Dependencies

There are some external libraries that we used. All of them are captured in Figure 7.1 too. They are:

- **ANTLR.jar** – An external library, the ANTLR parser that allows us to build and walk the AST of any ANTLRv4 grammar.
- **Java Swing** – Swing was used for creating the initial import form that the user is shown at the beginning of the import process.
- **MPS libraries** – The Library module is using the MPS API to create language elements. The Java code that is generated out of the Library BaseLanguage code, can be also compiled inside IDEA, but a set of MPS libraries needs to be referenced from within the IDEA module.

7.1.3.7 The Build Solution

The MPS project also contains a *Build solution*, which enables us to transform the plugin into an installable package. The output of this build process is a package in the form of a zip archive that can be permanently installed inside any MPS instance.

The build solution contains some descriptive XML files and a `.jar` archive for each solution of the MPS project. We will not go into more details here since the build process is only a configuration matter. More information about building plugins can be found in the second volume of the MPS manual [10].

7.1.3.8 Tests

There are also some unit tests, supplied together with the IDEA project. They mostly test parsing of the grammar file, flattening lexer rules and constructing regular expressions.

There are no tests needed for the Model module since there is no logic to test. It was not possible to write tests for the Importer module because a very extensive mocking of MPS API libraries would be needed for this. This presents a space for improvement since it is possible to write tests inside MPS, but the author of this thesis hadn't known that and early structure of the plugin code wasn't even enabling testing. This early plugin structure was much different from the final one. The plugin was heavily refactored in the final stage so that setting up the development environment is easier and building of the IDEA plugin is possible.

7.1.4 Setting up the Development Environment

In case you would like to contribute to the project, change the code, or just build the plugin yourself, you need to set up the MPS and IDEA projects correctly. There are also some technical requirements:

- MPS 3.4 EAP 3 and higher
- Running the MPS using Java 8 (JDK 1.8)
- JDK 1.8 for IntelliJ IDEA

As stated above, IntelliJ IDEA is only necessary when you need to change the Java code. Otherwise, MPS will be sufficient on its own. Setting up dependencies correctly presents the only problematic part. When you open both MPS and IDEA projects, you will be prompted to fill in two path variables:

- **INGRID_HOME** (MPS only) – path to the root of the ingrid repository (contains the `plugin` directory). This is not the root directory of the MPS project, which can be found in `plugin/mps`.
- **MPS_HOME** (IDEA only) – path, where MPS is installed. This is required, so that the library module of the IDEA project can use the MPS API.

After setting these two variables up, it is recommended to restart the IDE.

Next, open the Ingrid build inside of the **Ingrid.build** solution. In **macros** section, set the **mps_home** macro to point to the root directory of your MPS installation (same as the **MPS_HOME** path variable). Unfortunately, a relative path starting in the home directory of the Ingrid MPS project must be used.

To build the plugin, you only need to rebuild the whole project and run the build solution. The build solution can also be run using the Apache Ant library[2] and the **build.xml** file that can be found in the main directory of the Ingrid MPS project. The plugin solution should create a menu item inside of the *Tools* menu. However, we have experienced problems with the EAP (Early Access Program) version of MPS and if errors occur, we recommend to build solutions **one by one** in following order:

1. org.antlr
2. premun.mps.ingrid.model
3. premun.mps.ingrid.parser
4. premun.mps.ingrid.library

[2]http://ant.apache.org/

5. premun.mps.ingrid.importer
6. premun.mps.ingrid.plugin
7. Ingrid.build

The IDEA project has similar structure, but should not present any trouble at all.

7.2 User Manual

This section describes installation and usage of the plugin from the point of view of the end user. In case you wish to change the plugin and build your own version, follow steps in Section 7.1.4.

7.2.1 Installation

The plugin can be installed using a `.zip` package. This package is produced by the build solution, described in Section 7.1.3.7, and it is also attached on the CD included with this thesis. To install the plugin in MPS, you must meet following requirements:

- MPS 3.4 EAP 3 and higher
- Running the MPS using Java 8 (JDK 1.8)
- JDK 1.8 for IntelliJ IDEA

Plugin can then be installed by executing these steps:

1. Open ***File*** >***Settings***.
2. Go to the ***Plugins*** section (in the left part of the ***Settings*** dialog).
3. Click ***Install plugin from disk...***
4. Locate the plugin file **premun.mps.ingrid.zip** (you can find it on the attached CD, see the Attachments Chapter 10).
5. Restart MPS.

7.2.2 Usage

Plugin can be used by clicking on the ***Tools*** >***Import ANTLRv4 grammar*** menu item. In the dialog that is shown afterwards, use the + button to add all grammar files of the language that you wish to import. Continue by clicking on the ***Import*** button. A successful import should end with a dialog containing information about the imported language.

8. Example Imports

Together with the text of the thesis, you should be able to find several attachments. Among these attachments, there is an MPS project named *Examples*, which contains several imported languages:

- **SimpleXML** — Simplified XML that we used in this thesis for explanatory purposes.
- **JSON** — Similar to XML, but this time a full port of the specification.
- **ECMAScript 5.1** — Specification of the language known as JavaScript, dated to the year 2011, which is currently the most frequently adopted version.

For each example language, you can find its original import, exactly in the form as created by our plugin. Then, for each one, there is an adjusted version of this language. This adjusted version was created from the original import and the author spent no more than between 20 to 60 minutes of working on it. The goal was to prove that even though our plugin does not create perfect language, it can be customized very fast into a very usable form. We customized the editor and the TextGen aspect only, structure aspect was left in its original form. We hope that this justifies some of the decisions we have described in Chapters 6.5 and 6.6, proving that we hadn't been trying to avoid implementing complicated approaches, but really had been of the opinion that this is the best approach.

The JSON language was ready for use in no more than 20 minutes of minor adjusting. We chose this language because it doesn't require us to implement complex aspects, such as type checking or data flows. Similarly, the SimpleXML turned out very nice.

On the other hand, we have decided to try to import the JavaScript language, which is exactly the type of a complicated general purpose language that might be of interest to other people. There are already some projects, where a manual port of JavaScript is being done. One of them is the ECMAScript4MPS [13] from the author of the PE4MPS project [4] that we have been talking about in Chapter 4.1. We can deem the result of our import quite good, as we can compare it to this project. Our language is, of course, missing a lot of actions and intentions, that, for example, can help deriving types of expressions and so on. These aspects improve the usability by a lot, e.g. when the user starts writing numbers, a number literal is inserted, whereas in our import the user has to first insert the concept representing the number literal and then proceed with writing numbers. These actions are needed in any MPS language and are expected to be implemented after the import step, as it is not possible to generate them automatically. However, when it comes to language structure, concept aliases, and auto-completion, we think that our plugin did a very good job. The structure aspect is very similar to the ECMAScript4MPS's one, which was created manually over the course of surely a large number of hours. From these reasons, we think that we have achieved the goals of this thesis.

9. Problems with Grammars

As we could see in previous chapters, it is very easy to write a grammar in a way that will cause problems once it is imported into MPS. These problems might occur during the creation of any aspect and they are very hard to mitigate. We have not known about these problems and they occurred during the implementation. In this chapter, we will try to summarize our observations and show a few examples of possible complications.

9.1 Structural Problems

As a first example, we will describe some problematic structures. When writing the grammar, it might happen that we create some patterns that work just fine with ANTLR parser, but pose problems for the usability of the resulting MPS language. For example, we might want to use some syntactic sugar such as subrule blocks, which we talked about in Section 6.3.3. The problem that arises here, is similar to the layer problem, we mentioned in Section 6.4.3.2. It, however, cannot be resolved automatically. Let's look at the content rule of the original XML grammar:

```
content :   TEXT? ((element | CDATA | COMMENT) TEXT?)* ;
```

As we have shown in Chapter 6.6.2, the author of the grammar decided to handle whitespace not by skipping it using ANTLR actions, but rather by wrapping other content with a rule that contains these whitespace characters (the TEXT rule). Then, using the subrule notation, as we can see above, the author made sure that any content might be eventually wrapped in whitespace or text. This will enable the parser to handle mixed content (text, comments, tags or CDATA sections) inside an XML tag.

Now let's look, why this poses a problem for us. in Section 6.3.3, we have described the way, how we handle subrules. We unroll them into a set of parser rules and process them in the same way as if they were expanded from the beginning. The rule above is unrolled to following structure:

```
content           :   TEXT? content_block_1_1* ;

content_block_1_1 :   content_block_1_1 TEXT?
                  ;

content_block_1_2 :   element
                  |   CDATA
                  |   COMMENT
                  ;
```

This transformation just translates the shortened subrule notation into a classic parser rule definition, leaving the semantics intact.

When we import this structure inside MPS, the language will not be pleasant to use. The projectional editor will be full of (most likely) empty TEXT placeholders as shown in Figure 9.1. Sometimes they will contain text, sometimes spaces, but mostly they will distract us and pollute the projectional editor.

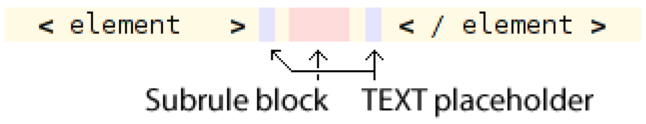
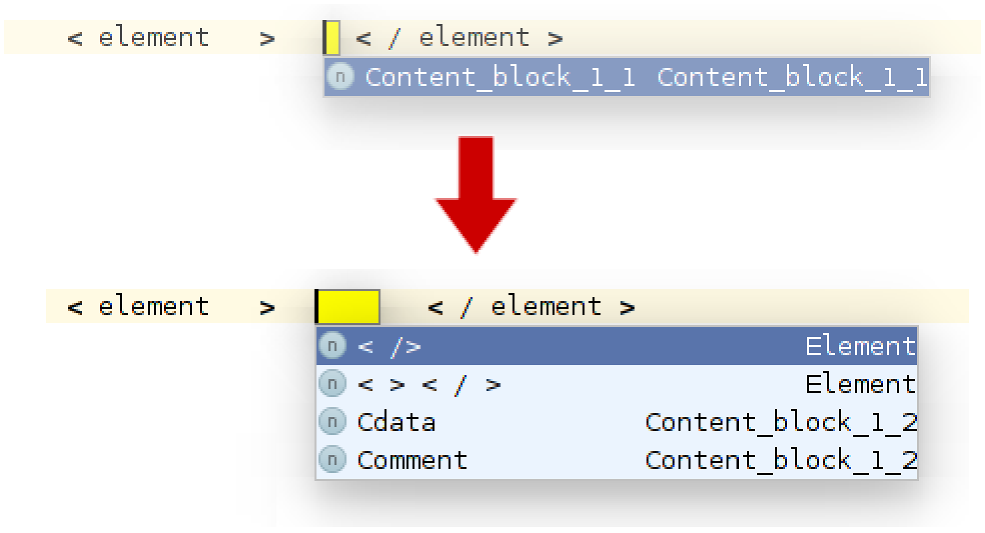


Figure 9.1: Placeholders in the editor

Furthermore, when the user wants to insert some content inside an XML tag, he will be forced through two layers and to choose between a weird set of some weirdly named block rules, about whose meaning he has no idea. We can see the process in Figure 9.2. The auto-completion options represent all end rule alternatives of the artificially created content_block_1_2 subrule.



< element > < / element >
Content_block_1_1 Content_block_1_1
< element > < / element >
< /> Element
< > < / > Element
Cdata Content_block_1_2
Comment Content_block_1_2

Figure 9.2: Subrule effect on language usability

There is, however, no way, how we could detect or prevent this when parsing the grammar, since the plugin has no higher understanding of the grammar. We cannot detect any smart shortcut, like described in Section 6.4.4, skipping the first layer of the auto-complete (content_block_1_1). The reason is that the first block rule contains more elements. Subrules can also be nested inside of each other freely and there can be any setup within multiple levels, additionally with various quantitative operators. We could, of course, try to parse the subrule block differently somehow, but for each possible solution, we can always find a very simple case that will break it.

The easiest solution to this problem is to alter the grammar directly and unroll it manually. We have done something like that with our SimpleXML language. We have restructured the content rule into a more simple version:

```
content     :   TEXT
            |   element
            |   comment
            |   CDATA
            ;
```

It also required us to change the grammar in some other places, but these were just minor changes, mostly concerning cardinality. It made our MPS language just right, but it also had some bad side effects. We talk about these side effects further down this chapter in Section 9.4.

9.2 Other Language Tweaks

There are also some cases, where we do not have a big structural problem per se, but altering the grammar might yield far better MPS language. We will show two examples, where we don't want to improve the structure, but again, the usability of the resulting MPS language. Let's look at the definition of an XML attribute:

```
attribute    :   Name '=' STRING ;

STRING       :   '"' ~["]* '"'
             |   '\'' ~[']* '\''
             ;
```

The original XML grammar has quotes as a part of the value. For the resulting MPS language, it would mean that there would be a placeholder for the attribute value that would expect us to input the leading and trailing quote together with the value too each time. It would also be marked red unless we enter both quotes inside the value since the regular expression checking for quotes will not match. The user might be confused by this and won't be able to tell why his string value is incorrect.

In our SimpleXML language, we adjusted the grammar easily in the following manner:

```
attribute    :   Name '="' TEXT1 '"'
             |   Name '=\'' TEXT2 '\''
             ;

TEXT1        :   ~["]* ;
TEXT2        :   ~[']* ;
```

We turned quotes into literals and they will only appear in the projectional editor as fixed constant cells. We won't have to encapsulate the value in them each time. The user will only have to choose, which attribute version he wants to use (single or double quotes). We could also decide to drop one of the two options, but then we will no longer have a full port of the XML language. Or we could adjust the language further and introduce a special action (for which the intention aspect of the language is used) to switch easily between these two, increasing the speed of coding. It is hard to decide which solution to this problem is the best. Users might have different intentions, requirements or goals for their MPS language and all approaches lead to some results.

As the second example, we will show the ECMAScript[1] language otherwise known as JavaScript. Every statement in JavaScript needs to be either followed by a semicolon, newline, file end or end of the block.

```
eos              : SemiColon
                 | EOF
                 | lineTerminatorAhead()?
                 | _input.LT(1).getType() == CloseBrace?
                 ;

// Example reference of the eos rule
breakStatement : Break Identifier? eos
                 ;
```

Because there are multiple options, our import plugin creates a placeholder at the end of every statement. Every concept representing a statement has one child of the IEos interface type. This placeholder needs to be manually filled in for each statement.

Since the projectional editor has much bigger power over the form of the code, we might want to have each statement on a separate line. Since we can differentiate between statements on the AST level, we don't need an explicit separator between them. This means that we might want to simplify the language and leave the semicolon out, or leave it just as a constant fixed part of the projectional editor, but not as something the user must explicitly fill in. Then we can just put each statement on a separate line as it is usual for JavaScript code, but we don't need the semicolon anymore. This small adjustment is very quick when done inside the grammar. We just change the eos rule to following form:

```
eos : ';' ;
```

Again, we changed the grammar in a way that it won't describe the same ECMAScript language as before, but will definitely make our MPS language more usable. Again, we can ask ourselves, whether it is an intended action — whether we aim for a full port of the language or an MPS alternative.

[1]https://github.com/antlr/grammars-v4/blob/master/ecmascript/ECMAScript.g4

9.3 Adjusting Grammars

Above, we concluded that adjusting the grammar itself might be sometimes the only proper solution to some complex situations. Sometimes it might be a very fast mean of tweaking our MPS language's usability. We also think that the end user of our plugin will be quite educated in this field since they will already be trying to create a computer language. It is also expected that this user will continue on improving the language after the initial import, adding more human touch to it. From these reasons, we concluded that end users of the plugin might also be capable of performing similar grammar adjustments themselves.

9.4 Breaking the Parser

There is one big problem with grammar adjustment that we would like to point out. The problem is that it is very easy to change the grammar in a way that will break the ANTLR parser generated out of it. By breaking we mean that it stops parsing the original language.

9.4.1 Breaking It Is Easy

As stated before, when creating the SimpleXML grammar, we have started off with the original XML grammar[2] and did some adjustments to it. After that, we ended up with a grammar just good enough for our plugin. Nonetheless, we noticed that even though the imported language behaves well enough and mimics the XML language quite nicely, the ANTLR parser generated out of this grammar no longer parses XML successfully. Some changes we have made, such as the attribute adjustment or parser mode omission, broke the grammar down. More precisely, it improved the MPS language, broke down the parser and we haven't even noticed it, because, from the perspective of MPS, the imported language still corresponds to XML.

What we are trying to say is that it is very easy to perform a harmless grammar adjustment that will, at first, seem valid inside MPS, but will break the parser. The problem is that the user performing this change, might and probably won't be aware of breaking it.

The cause of this problem is the way the ANTLR parser is implemented and the quite different purpose we are using the grammar for in this thesis. There are many ways how various parsers deal with, for example, token matching. For instance, ANTLR introduces so called **greedy** and **non-greedy** operators. The greedy way, in which ANTLR matches input on defined tokens and prioritizes their selection, makes some rules very dangerous. When a rule, such as the TEXT rule, matches a wide range of input, it might happen that when parsing, it is prioritized over other rules and swallows a lot more input than the author of it intended. Usually, these dangerous rules are bound to some parser context, which makes them behave well.

[2]https://github.com/antlr/grammars-v4/tree/master/xml

9.4.2 Breaking It Is a Problem

The underlying question is — **do we really care about the parser, being broken?** Our goal here is to create an MPS port of the language. It is also very hard, if not impossible, to verify, whether we have broken it and whether two grammars describe the same language.

The reason why we should care about not breaking it is that if we wanted to proceed with some more complex operations, we might need to automatically generate parsers of that language. Imagine this very real (future) scenario:

1. We have imported the language inside MPS.
2. MPS now knows the structure of the language, we can code in MPS using this language.
3. We would expect MPS to be able to load an existing text source code, written in this language, and import it inside MPS.
4. We would like to use MPS to safely edit this code, together with all features of MPS.
5. We would like to export the code in a text form again and save it back to the source file.

In order to make this happen, several steps must be performed:

1. We must generate the ANTLR parser out of the source grammar that helped us import the language.
2. We must parse the text source file correctly, using the parser from step 1.
3. We must be able to match nodes of the AST coming out of the ANTLR parser to concept nodes of the MPS language.
4. We must build the MPS AST out of the ANTLR AST.

For some of these steps, we would, of course, have to significantly improve some of the parts of the plugin as it is functionality that wasn't subject of this thesis. For example, we would need a support from our grammar parser, so that it would store a mapping between grammar rules and corresponding concepts of the MPS language. We would also need to store this information more permanently and deal with some further problems, such as the user adjusting the MPS language beyond the grammar definition.

But regardless of these problems, when we damage the grammar by adjusting it, we won't be able to create a valid ANTLR parser in the first place. That would leave us stuck at the first step. The user might not even know, why the source code import is broken. It would be hard to tell, whether we did break the grammar or not. Anyway, even if this thesis accomplishes its goal to some extent, advancing further without a valid grammar might make future follow-up efforts very complicated.

10. Conclusion

The thesis dealt with the problem of an automatic language grammar import into the MPS editor. As stated in Section 1.1, it was not expected that this thesis will solve the problem of the grammar import completely. It was expected that some ground research will be done, possibly allowing further follow-up work based on this thesis. Main goals of the thesis (also Section 1.1) were presumably achieved, even though it is quite hard to set acceptance boundaries, since it can be argued, what level of perfection was expected.

Firstly, the author, in Chapters 2 and 4, explored the given environment and made some technological decisions, such as choosing the grammar notation in Chapter 5. These decisions proved to be good ones since they enabled the author to achieve further goals quite efficiently. By *further goals* we mainly mean an MPS plugin implementation enabling automatic import of a given ANTLRv4 grammar as an MPS language. This was, together with some interesting problems that have come up, covered in Chapter 6. The implementation itself also held some problems such as a sparse documentation of the MPS API, complicated dependency setup or a little bit hostile development environment, which was further described in Section 7.1. The resulting MPS language is usable and, after some customization, can be considered as a solid foundation for a full MPS port of this language. Some simple languages, such as JSON or JavaScript, yield very good results, comparable to existing manual ports, but sparing the user from hours of tedious, error prone and time-consuming work.

It is also hard to measure the success, because before the work on this thesis has started, it had been quite unclear, what can actually be achieved. For instance, it was presumed (both by JetBrains and the supervisor) that the biggest part of the research would be devoted to the problem of projectional editor generation. It was expected that we will spend a lot of time figuring out how to derive the code layout from the grammar. However, a major part of the effort was put into the structure aspect, which turned out to hold some important problems on its own. Some of these problems have been overcome and some have not as it turned out they don't necessarily have a perfect solution. The most important problems, that we were unable to solve, concern the structure of the language and are described in Chapter 9.

During the work on this thesis, we have come to a conclusion, that the source of some major problems comes from the fact, that we are using the grammar for a little bit different purpose than it was designed for. Parsing existing source code and devising rules for creating it are two different points of view. This can be seen, for example, in Section 6.6. These problems also present potential obstacles for future follow-up work, so we believe that it is very important, we have identified them and pointed them out.

As per some ongoing discussions with others researching grammar-to-MPS import, it looks like this thesis might become a useful resource for future efforts.

Follow-up work might look more into the problem of code layout detection, possibly exploring suggested approaches such as the one described in Section 6.5.2. This could lead to better results in editor and TextGen aspect generation which could defnitely use some improvement. Another future endeavors will definitely touch the subject of generating parsers that would enable the user to import existing source code into MPS. This could be a very important breakthrough that would help with wider adoption of the MPS editor. This thesis might definitely help in this area since we have brought up some possible issues, such as the ones mentioned in Chapter 9.

Bibliography

[1] JetBrains MPS. www.jetbrains.com/mps/, 2016.

[2] BaseLanguage. confluence.jetbrains.com/display/MPSD34/Base+Language, 2016.

[3] mbeddr. mbeddr.com/, 2016.

[4] PE4MPS project. github.com/mar9000/pe4mps, 2016.

[5] PE project. github.com/mar9000/pe, 2016.

[6] ANTLRv4. github.com/antlr/antlr4, 2016.

[7] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, 2013.

[8] ANTRL_MPS project. github.com/CampagneLaboratory/ANTLR_MPS, 2016.

[9] Fabien Campagne. *The MPS Language Workbench: Volume I*. CreateSpace Independent Publishing Platform, 2014.

[10] Fabien Campagne. *The MPS Language Workbench: Volume II*. play.google.com/store/books/details?id=NPwyBwAAQBAJ, 2016.

[11] ANTLRv4 grammar repository. github.com/antlr/grammars-v4, 2016.

[12] JavaScript ANTLRv4 grammar. github.com/antlr/grammars-v4/blob/master/ecmascript/ECMAScript.g4, 2016.

[13] ECMAScript4MPS project. github.com/mar9000/ecmascript4mps, 2016.

Attachments

Together with the thesis comes a CD disk containing data in following structure:

```
CD/
 ├── examples/ .......MPS project containing imports from Chapter 8
 ├── grammars/ ....................example adjusted ANTLRv4 grammars
 ├── ingrid/ ..............files of the software part of the thesis
 │    ├── idea/ .....................IntelliJ IDEA project (Java part)
 │    ├── lib/ ...............libraries needed for plugin compilation
 │    └── mps/ ............................MPS project of the plugin
 ├── latex/ .......................source LaTeX files of the thesis
 ├── Grammar to JetBrains MPS convertor.pdf .....this thesis in PDF
 ├── premun.mps.ingrid.zip .....................compiled MPS plugin
 └── readme.txt ..................................info about the CD
```

List of Figures