1 Grammar structure

1.1 Problem introduction

NOTE: Here we assume that Rascal/MPS are already introduced

In this section we will introduce the problem of communicating the core "structure" of the textually defined grammar to a projectional editor. The core "structure" referenced here refers most closely to the abstract syntax tree of the grammar in question, i.e the non-terminals and their production rules, stripped of any layout information and literals. For example, lets take the production as shown in Figure 1. At this stage, all we are concerned with is:

- There is a non-terminal (Program).
- With a labeled production rule (prog).
- The rule contains a single "Declarations" non-terminal labeled "decls".
- The rule contains 0 to n "Statement" non-terminals labeled "body".

We can visualize this structure as a tree as in figure 2. Note that any purely visual information, such as the literals ("begin", ";" and "end") are lost. These will be handled under the Editor section of this thesis.

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

Figure 1: Textual grammar production rule

Now that we have a clear idea of what elements of the textual grammar we want to recreate in the projectional editor, we can start defining the sub-problems of how to do so. These goals will be as follows:

1 Extract the AST from the textual grammar definition.

2 Transfer this AST to the projectional editor.

3 Recreate the projectional model using the AST data.

We will now describe how our solution solve each of these defined sub-problems and how these put together solve the overall problem of recreating the core "strucure" of the grammar as a projectional model. First we will show a general overview of the solution and it's parts. Then we will take a closer look at each part in itself and show how it solves one of the sub-problems.

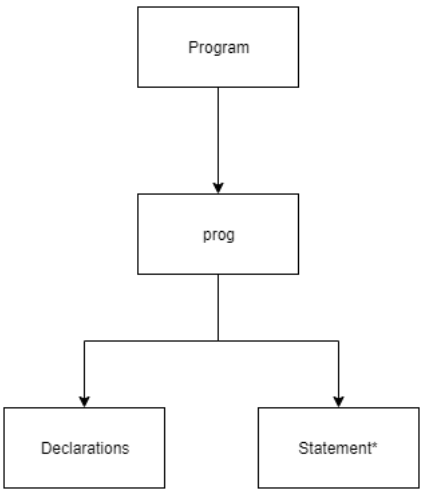

Figure 2: AST visualized

1.2 General overview

The general architectural overview of the solution in context of the Rascal and MPS worlds is presented in Figure 3. There are two main components, which exchange information through an intermediary format.

- Rascal2XML generates an the AST representation of a given Rascal grammar and outputs it as a XML file.
- XML2MPS takes a XML file representing a Rascal grammar AST, creates a new MPS Language and populates the Structure Aspect of said MPS Language by constructing Interfaces and Concepts based on the given AST.

Both parts are implemented within their own respective world: Rascal2XML consists of multiple Rascal modules and is thus called from the Eclipse editor. XML2MPS is implemented as multiple MPS solutions and a MPS plugin, and is correspondingly called from within MPS. The approach is therefore a two-step process: First one exports the XML-version of an AST representing a Rascal grammar. Next, one opens MPS with the XML2MPS plugin loaded and imports this XML file.

An alternative to this two-step approach would have been to implement the solution in such a manner that the abstract syntax tree can be communicated directly between both worlds, without the need for a file-based intermediary format. Such a solution was considered and tested early on in the project. This solution took advantage of the fact that Rascal itself is implemented in Java, and MPS has native support for importing and calling

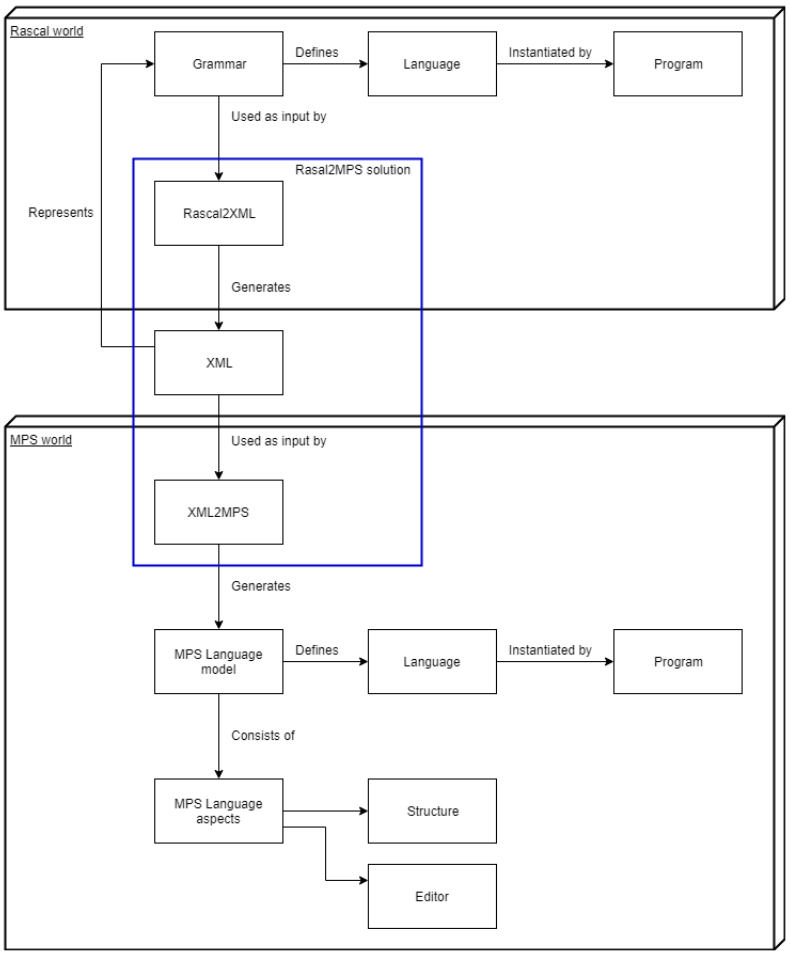

Figure 3: Rascal2MPS architecture and context

Java JAR files. The idea was to implement functions within Rascal, create a JAR of the entire Rascal runtime environment, including the implemented methods for creating an AST from a Rascal grammar file, and call these functions from within MPS. This would bypass the need for an external file-based data storage, as it would give us direct access to Rascal data types and functions. Unfortunately, this approach ran into technical problems.

NOTE: TODO More detailed explanation of the technical problem

Solving these problems would have required extensive research into both Rascal and MPS internal code. In the end, there were enough advantages to the two-step approach using a file-based intermediary format that it was chosen as for our solution. More about the choice of XML as our intermedi-

ary file-based format and the potential advantages a file-based intermediary format can bring us is presented in the "Intermediary format" section of this chapter.

Next we will take a closer look at each of the components of the solution and how these relate to the given sub-problems.

1.3 Intermediary format

1.4 Rascal2XML

First we will explore the problem of generating an abstract syntax tree from a given Rascal grammar file and storing this data in our file-based intermediary format, in this case XML.

To generate the abstract syntax tree of a given grammar file, we can reuse existing Rascal functionality in the # operator. In Rascal, calling NonTerminal where NonTerminal is the name of some Rascal syntax construct will generate a tree-like structure representing the grammar following this non-terminal. Thus calling this operator on the root of the grammar will generate a kind of "dummy" parse tree containing information about all the different alternatives for each non-terminal encountered recursively from the starting rule. In Rascal, starting symbols of a grammar are signified by the "start" keyword before a "syntax" declaration, as can be seen in Figure 1. In this case, we can obtain the representative parse tree of this grammar using "#Program".

This Rascal Tree structure however is quite large contains much more information than we require for our core "structure" as described in the introduction to this chapter. It contains for example layout information, literals and lexical definitions. To extract only the information we require we perform a top-down tree walk over the generated tree, filtering out the core structure, using pattern matching on the desired structures. There is one important distinction made by the algorithm here, namely between non-terminals and their production rules and lexicals. Lexicals in Rascal are a way for the language engineer to constrain user input using regular expressions. Exactly how lexicals are handled by Rascal2XML and XML2MPS in discussed in a later section. For now, it suffices to say that they are registered during the tree walk, since they appear in child nodes of production rules.

Besides selecting the required data from the Rascal parse tree, there is also another purpose to the tree walk. While we are doing th tree walk, we simultaneously construct DOM nodes using the Rascal standard DOM library. The constructed DOM encodes the tree that is to be stored in XML

format and later used by the XML2MPS component. It thus must contain any relevant information we wish to move between the worlds.

On the core structure level, the DOM is quite simple:

- There is a single top-level root node
- Containing 0..n "nonterminal" nodes.
- Each "nonterminal" node has:
  - A "name" child node with the non-terminal name
  - 0..n "production" nodes; one for each production rule of the non-terminal.
  - Each "production" node contains:
    - A "name" child node with the production rule label
    - 0..n "arg" nodes containing the references to other grammar structure.
    - Each of which contains:
      - "name" child node containing the reference label
      - "type" child node containing the reference non-terminal name
      - "card" child node containing the cardinality of the reference, either 1..1, 0..n or 1..n.

For example, if we take the our production rule in Figure 1 and put it through the algorithm, we end up with a corresponding XML file as shown in Figure 4. This is all the information we need to start creating a MPS Language and populate the Structure Aspect in the XML2MPS component.

1.5 XML2MPS

The next task will be to take the XML file generated by Rascal2XML, create a new MPS Language and populate the Structure Aspect of said MPS Language. To this end we must decide how to handle

1.6 Lexicals

1.7 Limitations

NOTE: This subsection might be better as its own chapter instead. There are several limitations to the current approach that place restrictions on the source grammar. The current restrictions are:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
 <nonterminal>
    <name>Program</name>
    <production>
      <name>prog</name>
      <arg>
        <name>decls</name>
        <type>Declarations</type>
        <card>1</card>
      </arg>
      <arg>
        <name>body</name>
        <type>Statement</type>
        <card>*</card>
      </arg>
    </production>
  </nonterminal>
</root>
```

Figure 4: XML representation of a Rascal grammar rule structure

- All production rules must be labeled. This concerns the label of a production rule, for example the "prog" for the production rule shown in Figure 1. Rascal does not actually require this label to be present. A production rule declaration as in Figure 5 is perfectly valid. However, when creating our MPS Language Structure Aspect, we use this label as name for the constructed MPS Concept. One approach to this problem is for there to be some automated renaming scheme for when the rule is unlabeled. However this run into several other problems: Concept names must be unique and users will use the concept name to select the element in the projectional editor. Thus the name must be descriptive; having several versions of "dummy" does not tell the user what element they are actually inserting. Renaming also becomes a problem in one of the other major problems we are trying to solve, namely the program importing, where name matching is used to determine the structure of the model. In the end, we chose to restrict the source grammar to require labeled production rules, since this does not alter the expressiveness of the grammar in any way while allowing us to create

- All non-terminal and production label names must be unique. MPS does not allow duplicate names of Concepts and Interfaces. Since the non-terminal and production label names must be unique

```
syntax AnnoElemDec
  = semicolon: ";"
  | ClassDec
  | ConstantDec
  | InterfaceDec
  | annoMethodDec: (Anno | AbstractMethodMod)* Type Id "(" ")" DefaultVal? ";"
  ;
```

Figure 5: Syntax declaration using unlabeled production rules