The MPS Language Workbench

Fabien Campagne

Volume I

http://books.campagnelab.org
# Contents

1 **Introduction** ................................................. 15

1.1 Motivation .................................................. 15

1.2 Language Workbenches ................................. 16

1.3 When Should you Read this Book? ..................... 17

2 **Getting Started** ............................................... 19

2.1 The Abstract Syntax Tree .................................. 21

2.2 The Projectional Editor .................................. 21

2.3 The MPS Language Workbench ......................... 23

2.4 The Project Tab ............................................. 23

2.5 Importing Languages into a Project .................. 27
2.6 Error Highlighting
2.7 Resolving Dependencies
2.8 Meet the Players
2.9 Meet the Projectional Editor
2.9.1 The Module Properties
2.9.2 The Editor Window
2.9.3 Executing the AST Root
2.9.4 Code Generation
2.10 The Intention Menu
2.11 The Inspector Tab
2.12 Book Organization

3 The Structure Aspect
3.1 Overview
3.2 Concepts
3.2.1 The extends clause
3.2.2 The implements clause
3.2.3 Instance can be root
3.2.4 The Concept’s alias
3.2.5 The Concept’s short description
3.2.6 Properties
3.2.7 Children
3.2.8 References
3.2.9 Abstract Concepts
3.2.10 Smart Reference
3.3 Interface Concepts
3.3.1 The extends clause
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>The Editor in Practice</td>
<td>93</td>
</tr>
<tr>
<td>6.1</td>
<td>Editor for the Person Concept</td>
<td>93</td>
</tr>
<tr>
<td>6.2</td>
<td>Authors collection</td>
<td>96</td>
</tr>
<tr>
<td>6.3</td>
<td>Editor for the TitleElement Concept</td>
<td>99</td>
</tr>
<tr>
<td>6.4</td>
<td>Editor for the AuthorRef Concept</td>
<td>100</td>
</tr>
<tr>
<td>6.5</td>
<td>Editor for the BiblioRecord Concept</td>
<td>100</td>
</tr>
<tr>
<td>6.6</td>
<td>Editor for the Bibliography Concept</td>
<td>101</td>
</tr>
<tr>
<td>6.7</td>
<td>Exercises</td>
<td>101</td>
</tr>
<tr>
<td>7</td>
<td>The Behavior Aspect</td>
<td>103</td>
</tr>
<tr>
<td>7.1</td>
<td>Overview</td>
<td>103</td>
</tr>
<tr>
<td>7.2</td>
<td>Concept Behavior</td>
<td>103</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Constructor</td>
<td>104</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Methods</td>
<td>105</td>
</tr>
<tr>
<td>7.3</td>
<td>Internal Concept Declaration Holder</td>
<td>106</td>
</tr>
<tr>
<td>7.4</td>
<td>A baseLanguage Primer</td>
<td>106</td>
</tr>
<tr>
<td>7.5</td>
<td>The SModel language</td>
<td>107</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Traversing the AST with SModel</td>
<td>107</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Node children, descendants and parents</td>
<td>108</td>
</tr>
<tr>
<td>7.5.3</td>
<td>Traversing references</td>
<td>109</td>
</tr>
<tr>
<td>7.5.4</td>
<td>SModel Types</td>
<td>109</td>
</tr>
<tr>
<td>7.6</td>
<td>Learning more</td>
<td>112</td>
</tr>
<tr>
<td>7.7</td>
<td>The Quotation language</td>
<td>113</td>
</tr>
<tr>
<td>7.8</td>
<td>AntiQuotations</td>
<td>115</td>
</tr>
</tbody>
</table>
# Behavior In Practice

8.1 Example Concept ......................................................... 117
8.2 The goodBye method .................................................... 118

# The Intentions Aspect

9.1 Overview ........................................................................ 121
9.2 Intention ........................................................................ 122
9.2.1 For Concept ............................................................... 123
9.2.2 Error Intention ............................................................ 123
9.2.3 Available in Child Nodes ............................................. 123
9.2.4 Description Concept Function ..................................... 123
9.2.5 Is Applicable .............................................................. 124
9.2.6 Execute Concept Function .......................................... 124
9.3 Surround-With Intention .................................................. 124
9.4 Parameterized Intention .................................................. 125
9.4.1 Adding a Parameter .................................................... 126

# The Actions Aspect

10.1 Overview ................................................................. 129
10.2 Understanding Node Substitution ................................... 129
10.3 The Editor Context ........................................................ 131
10.4 Node Substitute Actions ................................................ 133
10.4.1 Substituted Node ......................................................... 135
10.4.2 Description ............................................................... 135
10.4.3 Condition ................................................................. 135
10.4.4 Common Initializer .................................................... 136
13 The TextGen Aspect .................................................. 203
13.1 Overview ......................................................... 203
13.2 Understanding Text Generation ....................... 204
13.3 ConceptTextGenDeclaration ......................... 204
13.4 LanguageTextGenDeclaration ....................... 207
13.4.1 Operations .................................................. 208

14 TextGen In Practice .................................................. 209
14.1 Overview ......................................................... 209
14.2 The TextGen Aspect ......................................... 209
14.3 The Concept TextGen ........................................ 210
14.4 The Language TextGen ....................................... 210
14.4.1 A node for each language concept ............... 210
14.4.2 Bibliography operation implementation .......... 211
14.5 Generating Author Lists .................................... 214

15 The Generator Aspect ............................................... 217
15.1 Overview ......................................................... 217
15.2 Understanding Model to Model Transformations .......... 217
15.3 The Generation Context ..................................... 220
15.4 The Template Macro Language ....................... 220
15.4.1 Property Macros ............................................. 221
15.4.2 Reference Macros .......................................... 222
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.1.2</td>
<td>Define an Intention To Toggle the Annotation</td>
<td>249</td>
</tr>
<tr>
<td>16.1.3</td>
<td>Define an Annotation Editor</td>
<td>250</td>
</tr>
<tr>
<td>16.2</td>
<td>Concept Functions</td>
<td>250</td>
</tr>
<tr>
<td>16.2.1</td>
<td>Function Concept</td>
<td>250</td>
</tr>
<tr>
<td>16.2.2</td>
<td>Function Parameters</td>
<td>252</td>
</tr>
<tr>
<td>16.2.3</td>
<td>Function Return Type</td>
<td>253</td>
</tr>
<tr>
<td>16.3</td>
<td>Importing JAR files</td>
<td>254</td>
</tr>
<tr>
<td>16.3.1</td>
<td>First Class MPS Models</td>
<td>254</td>
</tr>
<tr>
<td>16.3.2</td>
<td>Libraries</td>
<td>257</td>
</tr>
<tr>
<td>16.4</td>
<td>Path Variables</td>
<td>259</td>
</tr>
<tr>
<td>17</td>
<td>Diagrams</td>
<td>263</td>
</tr>
<tr>
<td>17.1</td>
<td>Understanding Diagram Editors</td>
<td>263</td>
</tr>
<tr>
<td>17.1.1</td>
<td>Figure Implementation</td>
<td>265</td>
</tr>
<tr>
<td>17.2</td>
<td>Diagram Cell</td>
<td>266</td>
</tr>
<tr>
<td>17.2.1</td>
<td>Content</td>
<td>267</td>
</tr>
<tr>
<td>17.2.2</td>
<td>Elements Creation</td>
<td>267</td>
</tr>
<tr>
<td>17.2.3</td>
<td>Connector Creation</td>
<td>269</td>
</tr>
<tr>
<td>17.2.4</td>
<td>Palette Declaration</td>
<td>272</td>
</tr>
<tr>
<td>17.3</td>
<td>Diagram Node Cell</td>
<td>272</td>
</tr>
<tr>
<td>17.4</td>
<td>Diagram Connector Cell</td>
<td>274</td>
</tr>
<tr>
<td>17.5</td>
<td>Diagram Port Cell</td>
<td>275</td>
</tr>
<tr>
<td>17.6</td>
<td>Custom Figure Implementations</td>
<td>275</td>
</tr>
<tr>
<td>17.6.1</td>
<td>Creating a Figure Implementation</td>
<td>275</td>
</tr>
<tr>
<td>17.6.2</td>
<td>Creating an ExternalViewFigure</td>
<td>278</td>
</tr>
</tbody>
</table>
Motivation

After contributing three text-book chapters and knowing the effort involved in drafting and finishing each of them, I thought long and hard before deciding to embark on a book project. My key motivation for writing this book was that I needed a reference text that would make it easier to teach others how to use the Meta Programming System (MPS).

MPS represents a new paradigm to programming and information management. Intentional programming [Sim95] and meta programming [Dmi04], were conceived to overcome some of the challenges of traditional language design and development approaches.

I first came across MPS in the fall of 2006 when I downloaded one of the first Early Access Program (EAP) releases\(^1\). The description of the project suggested that the tool could help solve a number of problems I was encountering in the

\(^1\)See [http://icb.med.cornell.edu/wiki/index.php/Language_Oriented_Programming](http://icb.med.cornell.edu/wiki/index.php/Language_Oriented_Programming)
field of bioinformatics and this potential sparked my interest\(^2\). I quickly realized that the version of MPS I experimented with was not for the faint of heart. There was very limited documentation at the time and the only tutorial available with the EAP appeared incomplete. While I spent some time trying to understand this version of MPS, I had to give up when difficulties arose that I could not resolve\(^3\) and when other priorities surfaced. I would have liked then, to find a resource such as I hope this book will be for newcomers to MPS. Of course, the MPS documentation has greatly improved since the version I tested in 2006. I am writing these lines in the winter of 2013. Despite these improvements, I think that there is still a need for a resource that will help beginners become familiar with MPS in a short amount of time. I hope this book will fill this gap and provide both a reference and a gentle introduction to a very unique and promising tool.

### Language Workbenches

Language Workbenches are software engineering tools developed in the last ten years that help their users create new languages and the tools to write programs in these languages.

**Definition 1.2.1 — Language Workbench (LWB).** A Language Workbench is a software tool designed to help its users develop new computer languages.

Some LWBs focus on non-technical users and provide languages to help organize knowledge. Intentional Software (http://www.intentsoft.com/) is a commercial Language Workbench that falls in this category [Sim95; SCC06]. Another prominent language workbench aimed at software programmers is the JetBrains Meta Programming System (MPS) [Dmi04; Voe13; VS10]. MPS

\(^2\)&nbsp;More information about my research in bioinformatics can be found on the laboratory website: http://campagnelab.org

\(^3\)&nbsp;See http://icb.med.cornell.edu/wiki/index.php/MPS/Examples/TextLanguage
1.3 When Should you Read this Book?

If you are reading this book, you are either taking one of my courses or more likely are trying to use MPS to help with software engineering or programming problems that you encounter in your hobbies or in professional life. MPS offers a different paradigm from the traditional programming tools that you may be used to and this section aims to explain what type of problems MPS can help you with.

You will find that MPS is a useful tool if your project requires that you do one or more of the following:

1. Develop a Domain Specific Language (DSL),

   **Definition 1.3.1 — Domain Specific Language (DSL).** A Domain Specific Language is a language designed to meet the needs of a specific application domain[For+04]. Examples of widely used DSLs include HTML and SQL.

2. Develop an Application Specific Language (ASL),

   **Definition 1.3.2 — Application Specific Language (ASL).** An Application Specific Language is a language designed to meet the needs of a specific application program. An example is the PlantUML language used to describe UML diagrams so that the PlantUML program can render them. Some well-designed ASL which are reused by several programs have evolved towards DSLs. An example is the DOT lan-
language, initially used for input to GraphViz, which has evolved towards a DSL for representing graphs and is supported by several programs.

3. Need to automate the generation of source code,
4. Need to automate the generation of configuration files,
5. Need to develop a software framework for a specific application domain. Instead of developing a framework, consider modeling the domain as a set of languages in MPS that generate to the code that you want the framework for.

**Definition 1.3.3 — Software Framework.** A Software framework is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software. [Gac+04]

6. Need to write long boiler-plate code sections as part of usual development that are very repetitive. If you find that the interesting part of your programs are buried among a lot of implementation code, you should consider modeling the interesting part of the program with MPS and generate the implementation part automatically from the short language description.

You can find detailed discussions of the activities and the motivation for using a Language Workbench in the book “DSL Engineering” by Markus Voelter and colleagues [Voe+13]. The book that you are reading now is concerned with teaching you how to perform these activities and helping you quickly become productive with the MPS Language Workbench.
Traditional compiler technology relies on lexers and parsers to read programs expressed as a text files [Lam+06] and transform them into data structures called Abstract Syntax Trees (AST). This process is illustrated on Figure 2.1, top panel.

ASTs are the data structures that a compiler processes to generate executable code. In contrast, in the paradigm used by the Meta Programming System, the user interacts directly with one or several ASTs using a projectional editor. Figure 2.2 presents the relationships between the text representation of an arithmetic expression and a corresponding AST.

The MPS approach works directly with the AST, without the need to express programs as text. This brings several advantages:

1. Language extensions are simple to develop. Extending a lexer and parser for a complicated language requires specialized skills, while extending a language in MPS consists in defining new AST concepts, with their associated editor(s) and semantic. Extending a language consists in describing its structure and creating an editor for the new language concepts. This process is much simpler than
Figure 2.1: Two Paradigms. This figure compares the MPS programming paradigm to a more traditional programming approach. The top panel shows how programmers develop programs with a traditional language environment consisting of a text editor and compiler. The bottom panel shows how a programmer interacts directly with the MPS system.
that needed to extend a compiler for an existing language because the larger the language the more difficult it becomes to develop a text syntax that avoids ambiguities.

2. Different languages can be composed effectively without the risk of introducing ambiguities in the concrete syntax.

The Abstract Syntax Tree

Definition 2.1.1 — Abstract Syntax Tree (AST). An abstract syntax tree (AST) is a data structure traditionally used by compilers to represent and manipulate programs. An AST starts with a root node. The root node can contain child nodes. Nodes in the tree can be of different types, often arranged in one or more concept hierarchies. Figure 2.2 presents an example of an AST.

Definition 2.1.2 — AST Concept Hierarchy. The AST Concept Hierarchy describes types of the nodes that make up possible ASTs, and their relationships to other AST nodes concepts. The AST concept hierarchy is analogous to a data schema since its structure defines which ASTs can be constructed.

In the MPS tool, the AST Concept Hierarchy can be defined in the Structure Aspect, which will be presented in Chapter 3. In contrast to the traditional approach shown at the top of Figure 2.1, MPS implements a projectional editor.

The Projectional Editor

Definition 2.2.1 — Projectional Editor (PE). A projectional editor is a user interface that makes it possible to create, edit and interact with one or more ASTs.

Good user interfaces are usually difficult to develop, but MPS makes it easy to create a robust projectional editor for new languages. This book describes how to develop a PE in Chapter 5.
Figure 2.2: Understanding the AST. This figure presents the relationship between a program text view for an arithmetic expression (left), the corresponding Abstract Syntax Tree (middle) and the associated fragment of the node concept hierarchy (right). The expression shown adds three integer constants ($1 + 2 + 3$). The root of the AST is a BinaryOperator node (operator: $+$). Lines connecting nodes of the AST indicate aggregation relationships. For instance, the root note has a left child of type BinaryOperator, and a right child of type IntegerConstant, with value 2.
Figure 2.3 presents a snapshot of the MPS workbench. This snapshot should help you orient yourself around the MPS user interface. In the next section, we will look at the Project Tab in detail.

![The MPS Language Workbench](image)

The Editor

The Project Tab

Figure 2.3: **The MPS Language Workbench.** The snapshot presents the user interface of the MPS Workbench. When starting to work with MPS, it is useful to focus on the editor and Project tab. The editor is shown to the right and the project tab to the left. Note that you need to open an existing project, or create a new one to see this view.

**The Project Tab**

A key user interface element of MPS is the Project Tab. This tab gives you access to a logical organization of the languages included or imported in your
Figure 2.4: The Project Tab Explained. Solutions and Models. Orange icons with a S inside them denote Solutions. The name of the Solution follows the orange square. Solutions contain models, which themselves contain ASTs.

Language and Language Aspects. The yellow square icons with an L in them represent languages. The name of the language is shown to the right of the icon. Solution and language icons are prefixed with a little triangle. Click on the triangle to open the solution or language. Click again to collapse the solution or language. Opening a solution presents the models in the solution (ExecuteCommand is a model in the org.campagnelab.NYoSh.sample solution). In contrast to solutions, opening a language exposes a list of language aspects. Several aspects will be described in detail in the following chapters. See Table 2.1 for a brief description of the function of these aspects.

Modules Pool. Shown in green at the bottom is the Modules Pool. Note the other Languages and Solutions shown under Modules Pool are not part of the project, but provided by the platform or by some active MPS plugin. It is a great place to look for languages that you might wish to use in your own languages or solutions.
When working with MPS, you will often use the Project Tab to modify properties of languages or solutions that make up an MPS project. Let’s take a moment to become familiar with this MPS user interface element. A blown-up Project Tab is shown in Figure 2.4 to show details and explain the meaning of the icons included in the tab.

Developing a new language in MPS consists in adding and configuring nodes in these various language aspects. Note that some aspects are not created by default when creating a new language. You can add new aspects to a language by right clicking on a language name and selecting *New* Aspect (see Figure 2.5).

Figure 2.5: **Create New Aspect.** Some language aspects are not created by default for new languages and can be created as shown in this snapshot.

The function of the various MPS languages aspects shown in the Project Tab are explained in Table 2.1.
<table>
<thead>
<tr>
<th>Language Aspect</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structure</strong></td>
<td>Defines the AST Node Concept Hierarchies.</td>
</tr>
<tr>
<td><strong>Editor</strong></td>
<td>Define mini-editors for each concept of the structure aspect.</td>
</tr>
<tr>
<td><strong>Actions</strong></td>
<td>Provide means to customize the editing behavior of the editor.</td>
</tr>
<tr>
<td><strong>Constraints</strong></td>
<td>Provides means to restrict which AST nodes can be child or parents or other nodes and help define scopes.</td>
</tr>
<tr>
<td><strong>Behavior</strong></td>
<td>Provides method declarations and definitions for concepts of the structure aspect.</td>
</tr>
<tr>
<td><strong>Typesystem</strong></td>
<td>Makes it possible to implement a typesystem for the language.</td>
</tr>
<tr>
<td><strong>Scripts</strong></td>
<td>Holds AST migration scripts to evolve the language during its life-time.</td>
</tr>
<tr>
<td><strong>Intentions</strong></td>
<td>Defines language intentions: context-dependent AST transformations that the user can activate on demand.</td>
</tr>
<tr>
<td><strong>DataFlow</strong></td>
<td>Defines the language data flow rules which govern which statements are reachable, which variables are read from or written to.</td>
</tr>
<tr>
<td><strong>Test</strong></td>
<td>Provides the means to test languages by developing unit tests to check type calculations and error detection.</td>
</tr>
<tr>
<td><strong>Refactorings</strong></td>
<td>Provides the means to offer refactoring tools for the language to end-users [Ele+01].</td>
</tr>
<tr>
<td><strong>TextGen</strong></td>
<td>Provides means to transform AST nodes to text.</td>
</tr>
<tr>
<td><strong>Generator</strong></td>
<td>Provides means to transform AST nodes to other languages.</td>
</tr>
</tbody>
</table>

Table 2.1: Function of the most common MPS Language Aspects.
It is useful to realize that MPS aspects are implemented as MPS models, which contain ASTs and are created and edited with the same mechanisms that are available for languages developed with MPS. For instance, the structure aspect for the language `org.campagnelab.Swift` is implemented as a model stored under `languages/org.campagnelab.Swift/languageModels/structure.mps` and contains ASTs expressed in the `jetbrains.mps.lang.structure` language.

### Importing Languages into a Project

When you create an empty MPS project, you might find it useful to import other languages into the project alongside the languages that you are developing. For instance, you could open the sample projects shipped with MPS as standalone projects, but it is often more convenient to import some languages from these projects into your own project. You then become able to switch back and forth between languages in the same workbench window and are able to copy and paste across languages.

Let’s assume you created an empty MPS project (or a project with one solution and/or language). You can import languages into this project by right clicking on the name of the project, at the very top of the Project pane and selecting ‘Project Paths’. This opens the dialog shown in Figure 2.6. Try it now by creating a new project, and adding the Complex language to your new project.

### Error Highlighting

After adding the language following the instructions of Figure 2.6, you should notice that the `jetbrains.mps.samples.complex` language is highlighted in red. MPS uses this cue to indicate an error in the language. See Figure 2.7 to learn

---

1The complex language should be located under `~/MPSSamples.3.0/complexLanguage/languages/complex/jetbrains.mps.samples.complex.mpl`, where `~/` indicates your home directory.
(1) Right click over Project Name
(2) Select Project Paths
(3) Press + in the Project Properties dialog
(4) Locate MPS sample complex language
(5) Press OK

Figure 2.6: Adjusting Project Paths. You can import new languages and solutions into a project by following these instructions. Notice the names of the user interface dialogs as we will refer to them by name in the future.

Figure 2.7: An Example of Error Message. Browsing over the name of the language highlighted in red reveals the error message. In this case, MPS indicates that the runtime solution associated with the complex sample language cannot be found in the project. To fix this error, you can add the complex runtime solution to the project following the approach you used to add the complex language.
how to find the cause of this error.\textsuperscript{2} Once you have followed similar steps to those shown in Figure 2.6 and added the runtime solution, you should notice that the error has been resolved. Dependencies encoded in languages and solutions must be resolved explicitly within a project for solutions and languages that are not available under Modules Pool.

**Resolving Dependencies**

Modules pool contain languages shipped with the MPS platform or bundled with plugins that are active in the MPS workbench. You can inspect the plugins active in your session by opening MPS Preferences and locating the Plugins section. MPS can download plugins through a remote repository, or you can install plugins manually by installing plugin distributions under the plugins directory of the MPS distribution. Follow the installation instructions given with each plugin.

**Meet the Players**

An MPS projects can contain a number of parts, organized in a hierarchical fashion. Typical parts are shown below, organized starting with Languages and Solutions which are the parts of MPS Projects:

- Languages
  - Structure Aspect
    - Concepts
    - Concept Interfaces
    - Enumeration Types
    - Constrained Data Types
    - Primitive Data Types

\textsuperscript{2} The complex solution should be located under ~/MPSSamples.3.0/complexLanguage/solutions/jetbrains.mps.complex.runtime/jetbrains.mps.samples.complex.runtime.msd, where ~/ indicates your home directory.
- Other Aspects [See aspect chapter for description of parts]
  - Solutions
    - Models
      * AST Roots
        - AST Nodes

In this hierarchical structure, parts are organized from parent to descendants, where descendants are contained in a parent. For instance, Solutions contain Models that contain AST Roots, which contain AST Nodes and their descendants.

MPS Project parts can also be connected across this hierarchy. For instance, Solutions contain models described in specific Languages.

**Definition 2.8.1 — Language.** An MPS Language describes what types of ASTs can be created with the language. A language includes a set of AST Node Concepts, semantic and behavior, represented in one or more MPS Aspects. A language is a kind of MPS Module.

**Definition 2.8.2 — Solution.** An MPS Solution holds a set of models. Solutions are convenient packaging units that make it possible to reference a set of models as a unit from other Solutions or Languages. A solution is a kind of MPS Module.

**Definition 2.8.3 — Model.** In MPS, a Model is a container for a set of AST Roots. Models are serialized to disk in XML (.mps extension) or binary format (extension .mpb).

**Definition 2.8.4 — AST Root.** An AST root is an AST node that can be used as an immediate descendant of a Model. AST Roots are analogous to source files in a traditional programming paradigm.

The specific set of languages associated with a given Solution is recorded in a dimension orthogonal to the part hierarchy. Each solution includes references
to the set of languages needed to write ASTs in the Solution.

Remember that MPS Projects are organized as a large AST (whose root node is the project node). Links between Solutions and Languages are an example of reference across an AST. AST References will be described in Chapter 3.

Meet the Projectional Editor

Figure 2.8: Create A New Project with a Solution. This snapshot provides directions to create a new project, starting at the window displayed when you first start MPS. If you already have one or more projects opened, close these projects to see this dialog. Alternatively, you can select File > New Project from the menu in any project to bring up the New Project Dialog (steps (2-4)). Create the project as instructed to continue learning how to use the editor.

Now, import the MPS Sample math language into the new project. We described how to import languages in a project in Section 2.5. You will need to import the complete set of dependencies shown below to resolve errors:
1. jetbrains.mps.baseLanguage.math
2. jetbrains.mps.baseLanguage.math.pluginSolution
3. jetbrains.mps.baseLanguage.math.runtime

The Module Properties

In order to use these languages in the Solution MeetTheEditor, we need to declare that the solution uses the math language. To do this, you need to know about the Module Properties dialog. You can bring this dialog up by right clicking on the Solution name in the Project Tab, and selecting the last item of the popup-menu: Module Properties. In the rest of this book, we will use a short-hand notation to denote interactions with the MPS user interface. The previous action will be represented as \text{solution name} \rightarrow \text{right-click} \rightarrow \text{Module Properties}. Follow the instructions in Figure 2.9 to declare that the MeetTheEditor Solution uses both the jetbrains.mps.baseLanguage.math and jetbrains.mps.baseLanguage languages. Note that we add jetbrains.mps.baseLanguage because the sample math language does not provide a root AST concept. The math language provides a baseLanguage extension that can be used inside baseLanguage statements.

Definition 2.9.1 — Used Languages. Each MPS module has a set of Used Languages. Only languages that are declared under Used Languages can be used to write ASTs in this module.

Let’s try the math and baseLanguage languages by creating a model under MeetTheEditor solution. Right-click on the solution name and select \text{New} \rightarrow \text{Model} in the popup menu. The New Model Dialog appears. Type LetsStartHere.Math in the Model Name field and press OK. You are then presented with a Model Properties Dialog for the LetsStartHere.Math model. The dialog has three tabs: Dependencies, Used Languages and Advanced. Enter math and baseLanguage in the Used Languages tab in a similar manner to that used for the Solution. See Figure 2.10 for directions.
2.9 Meet the Projectional Editor

Figure 2.9: Adjusting Used Languages. This snapshot provides directions to adjust the set of Used Languages for a solution.

The Editor Window

After you have created the class AST Root node, you will see in the editor tab the content shown in the top left corner of Figure 2.12. Follow the directions in this figure to enter two complex variable declarations.

If you are familiar with the Java programming language, you will recognize the class language construct, but may be puzzled by the complex variable type. This type appears in the editor as if it was a primitive type of the language, but the Java language does not offer such a type [Gos+05]. The explanation is that we are not editing Java code in a text editor, but are editing an AST in MPS by composing two languages. The baseLanguage (language) provides most of the Java language syntax and behavior, while the jetbrains.mps.baseLanguage.math (math) extends baseLanguage with features to make it easier to work with mathematical abstractions. The complex primitive type is offered by the math language. Similarly, typing the I letter in the editor enters the imaginary number i and makes it possible to use this complex number in expressions.
Figure 2.10: **Model Properties.** Follow these directions to specify model properties for the MeetTheEditor.Math model. The set of used languages should look as shown when you are done. The language names are greyed out because they are currently not used anywhere in the model. When the list of used languages grows over time and as and projects evolve, greying out languages that are not used helps remove languages from a model that are no longer used.

Figure 2.11: **Create an AST Root.** This snapshot provides directions to create an AST Root node. Select a Class Root node from `baseLanguage`. If you open the Model Properties dialog again, you may notice that `baseLanguage` is no longer greyed out (the language is now used in an AST node of the model).
2.9 Meet the Projectional Editor

Figure 2.12: **Walkthrough the math language Editor Example.** Immediately after creating a class AST Root node, follow these directions to learn how to edit the AST in the editor. In step 11, notice how the editor presents two possible completion choices after you have typed the “*” character.

Well-designed MPS languages present a smooth editor experience and include notations that feel natural to the intended end-users of the languages. In the previous walk-through the complex data type is well-integrated with baseLanguage: (i) it is possible to declare new variable declarations of type complex, as if the complex type was a primitive type of the language, (ii) the complex instance \(i\) can be used inside expressions of the language. This book aims to describe how to develop languages with MPS that feel natural to use for end users of the language.

### Executing the AST Root

2.9.3

Some MPS languages are designed to generate code that can be executed. This is the case of the math language because it generates to baseLanguage and baseLanguage generates to Java source code. MPS make it possible to generate and run Java code directly within the workbench. Figure 2.13 shows how to execute the program that we created in Figure 2.12.
Getting Started

Right-click when mousing over (1) the main method name

Running HelloWorld displays results

Figure 2.13: Running the Complex Math Example. Follow these steps to generate, compile and execute the complex example. (1) Right click on the main method. Select Run ‘Class HelloWorld’ in the popup menu. Execution occurs in the Run tab (3), at the bottom of the MPS workbench. Comparing the output to the program shows a seamless integration of the complex type in baseLanguage.

Code Generation

2.9.4

What kind of code does MPS execute when you select Run ‘Class HelloWorld’? The answer is code generation. The math language includes a Generator Aspect that transforms the concepts of the math language into concepts of baseLanguage (model to model transformation is described in Chapter 15). Because MPS ships with a Build System capable of compiling Java code seamlessly, and because baseLanguage classes with a public static void main method can be executed by the workbench you can seamlessly run the generated code in the workbench. Let’s have a look at the generated code. You can do this with solution nameAST Rootright-clickPreview Text Output. See Figure 2.14 for detailed steps.

The Generator Aspect is explained in Chapter 15 and describes how to
Figure 2.14: **The Complex Math Example Generated to Java Source Code.** To generate this output from the editor window, locate the HelloWorld AST Root, right click and select ‘Preview Generated Text’. An alternative way to preview the generated code is to right click on the Model or AST Root node in the Project Tab. Select ‘Preview Generated Text’. Note that models may generate multiple output text files if they contain several AST roots. Compare the generated output to the content of Figure 2.13.

```java
package LetsStartHere.Math;

/*Generated by MPS */
import jetbrains.mps.baseLanguage.math.runtime.Complex;

public class HelloWorld {
    public static void main(String[] args) {
        Complex a = new Complex(2, 3);
        Complex b = new Complex(2, 3);

        System.out.println("a*b=" + a.mul(b));
        System.out.println("i*i=" + Complex.I.mul(Complex.I));
    }
}
```
generate text output or to transform an AST expressed in one language into another one. While many approaches have been developed over the years to generate code, including StringTemplate[Par] and Velocity Template[Con], MPS provides a principled and very effective platform to generate code from concise language notations. MPS Generators scale easily as projects grow and benefit from full editor support in the templates. This is a strong advantage that has no equivalent in existing code generation tools based on templates.

2.10 The Intention Menu

Intentions are a convenient way to interact with the AST when there is no simple way to map an action to a word\(^3\) or set of keys\(^4\). An intention menu is available on nodes that provide intentions. In MPS, the intention menu is shown as a yellow light bulb (💡) in the left margin of the editor when the cursor is immobile on a node for a few seconds. You can also trigger the display of the intention menu with the keyboard. Use option+→ on mac, or ctrl+→ on Windows or Linux. When the intention menu appears, select the intention that you want to run, or navigate to the code that runs the intention by using the intention submenu.

2.11 The Inspector Tab

The Inspector Tab is located at the bottom right of the main MPS window.\(^5\) It can be opened by clicking on the icon. The Inspector Tab shows the Inspected Cell Layout for a node. The Inspected Cell Layout can be defined when developing a custom AST node editor (see Chapter 5 Section 5.2.4).

---

\(^3\)Section 5.3.1 describes how to run actions when some text is typed.

\(^4\)Section 5.3.2 describes how to run code when a key combination is pressed.

\(^5\)In MPS 3.1+, you may need to enable the inspector tab before you can open it. You can enable/disable tabs by clicking or browsing over the icon in the lower left corner of MPS 3.1 window.
For existing MPS Concepts, the Inspector Tab will often provide a means to edit properties associated with a node which are not shown in the main editor window. This can be done to avoid clutter or to provide a complementary editing dimension when working with some kinds of AST nodes. This is particularly useful when working with the Editor and Generator languages.

**Book Organization**

This book is organized in a succession of reference chapters interleaved with chapters that present applications of the concepts covered in the reference chapters. Reference chapters focus on clear definitions and exhaustive explanations of MPS capabilities. Application chapters highlight some of the most important points through a worked example.

**Conventions**

This book uses the following typographical conventions:

1. italic identifiers refer to MPS languages (e.g., `baseLanguage` or `jetbrains.mps.baseLanguage`). Note that I also use italic to refer to concept interface names, but there should be no ambiguity with language names.
2. Keystrokes are shown as follows: `option + SPACE`. This means that you should press the option key together with the SPACE key on the keyboard.
3. Menus that you should select to perform some action are shown as `Solution right-click Model Properties`. Note that MPS has a menu bar, but attaches menus over several types of user interface elements.
4. The 💡 symbol represents the icon that appears in the left margin of the editor and makes it possible to execute an intention or navigate to the code of the intention.


$CALL$, 225
$COPY_SRCL$, 224
$COPY_SRC$, 224
$IF$, 225
$INCLUDE$, 225
$INSERT$, 225
$LABEL$, 226
$LOOP$, 226
$MAP_SRCL$, 227
$MAP_SRC$, 226
$SWITCH$, 227
$TRACE$, 227
$VAR$, 227
$WEAVE$, 227

ASL, 17
AST Concept Hierarchy, 21
AST Root, 30
Auto-completion Menu, 44, 60, 86

baseLanguage, 105
Bootrapping dependencies, 115
Build System, 36

CALL, 225
Cell Action Map, 69
Cell Key Map, 70
Cell Menu Component, 86
Cell Menu Parts, 88
Color, 85
come from, 174
CompositeScope, 171
Computable, 132
Concept Editor Context Hints, 80
Concept functions, 80
caption// expression, 110
caption<Concept>, 110
concepts menu, 139
ConceptTextGenDeclaration, 204
Connector Creation, 269
container, 268
Content, 267
Content Node, 232
COPY_SRC, 224
COPY_SRCL, 224
Create Template Fragment, 233
Custom Figure Implementations, 275

DelegatingScope, 171
Dependent Computation Item, 201
Diagram Cell, 266
Diagram Connector, 269
Diagram Connector Cell, 274
Diagram editors, 263
Diagram Element, 267
Diagram Node Cell, 272
Diagram Port Cell, 275
Diagrams, 263
Domain Specific Language, 17
Downcast operator, 77
DSL, 17

EAP, 15
Editor Cell, 68
Editor Component, 82
Editor Styles, 83
editorContext, 131
Element Menu, 72
Elements Creation, 267
EmptyScope, 170
ensure, 192
enummember<Enum Data Type>, 112
executeCommand(), 132
Extends Clause, 42, 49
ExternalViewFigure, 278

Figure Implementation, 265
Figure parameters, 273
FilteringByNameScope, 172
FilteringScope, 172
Find usage, 101
genContext, 220
Generator Descriptor, 236
generic query, 143
GlobalScope, 166
Greying out, 34

IF, 225
Implements, 42
INamedConcept, 60, 93
indent-layout-new-line-children, 100
Inline Template, 222
inputPorts, 273
INSERT, 225
Style Sheets, 84
Substitution menu, 130
Super-concept, 42
Surround-With Intention, 124
SWITCH, 227

Target node, 130
Template Declaration, 231
Template Macro Language, 220, 233
Template Switch, 233
The Constraints Aspect, 159
The Editor Language, 68
The Generation Context, 220
The Meta Programming System, 15
TRACE, 227
Transform Menu Actions, 144
Traversing the AST, 107
typeof, 192
unordered, 46
Used Languages, 32, 33

VAR, 227

Weak sub-typing, 186
WEAVE, 227
when concrete, 192
wrapper, 142