FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

István Satmári

Frege IDE with JetBrains MPS

Department of Distributed and Dependable System

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

Prague 2018

I would like to thank my thesis supervisor RNDr. Pavel Parízek, Ph.D. for his time, valuable advice and remarkable patience while guiding and helping me with writing this thesis. Without him this work would never have been finished.

I would also like to express my gratitude to Václav Pech from the JetBrains Company who helped me with some of the technical challenges I encountered during the course of this work and patiently explained the difficult or otherwise confusing parts of the MPS tool.

Title: Frege IDE with JetBrains MPS

Author: István Satmári

Department: Department of Distributed and Dependable Systems

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

Abstract: Frege is an open-source project which brings the popular functional programming language Haskell to the Java ecosystem. JetBrains MPS is an open-source language workbench which allows users to design a new language and build an integrated development environment with a projectional (structured) editor for the created language. In this work we analyzed Frege grammar and created an IDE based on MPS that assists developers with writing code in the Frege language. Our environment includes a set of intuitive editors for editing Frege syntax, provides a simple type checking and implements code generators for the Frege language. Aim of the Frege IDE is its usability. Additionally, the thesis compares projectional editors with the more common plain-text IDEs, such as Eclipse, and evaluates whether they offer any advantage for editing purely functional programming languages.

Keywords: Frege, Haskell, IDE, MPS, projectional editor

Contents

Introduction **7**
Organization 8

1 JetBrains MPS **9**
1.1 Project Structure 11
1.2 Structure 12
1.3 Editor 15
1.3.1 Editor Actions 18
1.3.2 Transformation Menu Actions 20
1.3.3 Substitute Menu Actions 23
1.3.4 Cell Action Map 27
1.4 Behavior 28
1.5 Intentions 29
1.6 Constraints 31
1.6.1 Scope 31
1.7 Typesystem 32
1.8 Textgen 35

2 Frege **37**
2.1 Hello, World! 38
2.2 Pattern Matching 38
2.3 Types 39
2.3.1 Tuples 40
2.3.2 Lists 40
2.3.3 Custom Algebraic Data Types 42
2.3.4 Type Synonyms 43
2.3.5 Function Types 43
2.3.6 Generic Type 44
2.4 Operators 44
2.5 Currying 45
2.6 Where 46
2.7 Guards 46
2.8 Constant Definitions 47
2.9 Import and Export 48
2.10 Further Reading 50

3 Frege in MPS **51**
3.1 Supported Subset of Frege 51
3.2 Structure Aspect 54
3.2.1 Notation Description 54
3.2.2 Program Structure 55
3.2.3 Definitions 56
3.2.4 Import Statements 57
3.2.5 Function Definition 62
3.2.5.1 Left Hand Side 63

3.2.5.2 Right Hand Side 64
3.2.5.3 Expression 64
3.2.5.4 Grouped Representation 67
3.2.6 Types 67
3.3 Editor Aspect 70
3.3.1 Visual Appearance of AST Nodes 71
3.3.2 Side Transformation Menus 72
3.3.3 Transformation Menu Inclusion Pattern 74
3.3.4 Substitute Menu Actions 76
3.3.5 Wrap Substitute Menu 78
3.3.6 Cell Action Map 79
3.3.7 Seamless Definitions 80
3.3.8 Expression Operators 83
3.3.9 Final Remarks 84
3.4 Code Completion 85
3.4.1 Scope 85
3.4.2 Scope Hierarchy Pattern 87
3.4.3 Import and Export 89
3.4.4 Implicitly Imported Library 92
3.5 Type Checking 92
3.5.1 Types 92
3.5.2 Infix Expressions 94
3.5.3 Function Application 99
3.5.4 Function Definition Type Inference 100
3.5.5 Arguments Type Decomposition 102

4 Evaluation **104**
4.1 Editing Programs in Frege-IDE 104
4.2 Features and Limitations 107
4.3 Comparison with Plain-Text IDE 109

Conclusion **111**
Future Work 112

Bibliography **113**

Attachments **114**

Introduction

Integrated development environments (commonly abbreviated as IDE) are a set of software applications that provide tools and facilities to software developers. They greatly ease the process of development, providing features like intelligent code completion, syntax highlighting, build automation tools, debugger, and many others.

Most IDEs are built as text editors that provide additional features when editing a source code. The editors usually parse the code; generate a parse tree, which allows static code analysis and generic error checking of the written program.

A different approach to designing an IDE can be done via projectional editing. A projectional editor (also known as structured editor) is a document editor that is cognizant of the document's underlying structure. It is usually used to edit hierarchical or marked-up text, computer programs, diagrams, and any other type of content with a clear and well-defined structure [1]. While for the most computer programs, due to their complexity, a conventional text-based IDE may be more suitable, for specific programming languages, especially DSL (domain specific language) a projectional editor might prove to be a more effective tool.

In this work, we intended to create a projectional IDE for the functional programming language Frege to evaluate, whether such approach makes sense and whether projectional editors offer more convenience than regular text-based IDEs when it comes to working with functional languages in general. The application we implemented in this text is often referred to as *Frege-IDE.*

Frege, named after the German mathematician, Gottlob Frege, is a functional language, heavily based on Haskell, trying to bring the language to Java ecosystem. It is considered a Haskell dialect, sometimes called *a Haskell for the JVM* (Java Virtual Machine) [2].

There are several IDEs for Haskell[1]. Most of the known IDEs provide mainly syntax highlighting, macros and project management features, while some also support more advanced functionalities, such as code completion and type checking. Frege, being a relatively new project, does not have as extensive support in IDEs as Haskell, which is one of the reasons why we decided to create an environment specifically for that language.

As an underlying tool for designing our environment, we have chosen an open-source language workbench JetBrains MPS[2]. MPS (standing for *Meta-Programming System*) is a software solution allowing developers and language designers to create a projectional editor, together with advanced features found in many IDEs, such as code completion, syntax highlighting and others. It is primarily used for designing editors for DSLs, for developing new languages and also extending existing ones, when the languages available do not meet the needs of a developer. MPS has a large set of features, allowing for designing editors which closely resemble those from conventional, text-based, IDEs. It allows a language designer to define a structure of AST

[1] Examples available from the WWW [12/07/2018]: <https://wiki.haskell.org/IDEs>
[2] MPS is available at the WWW [14/07/2018]: <https://www.jetbrains.com/mps/>

(abstract syntax tree) to represent the code, editor for manipulating the AST and a text generator to transform the AST into pure text. More about the platform is described in Chapter 1.

Frege, based on Haskell language, has also rather many syntactic and semantic constructs for this work to be able to include them all. We have therefore focused our attention only on the most important features worth examining, such as function declaration and definition, operators and custom data types. Our ideal IDE would have to have a user-friendly editor, which closely emulates writing Frege code in such a way most developers in that language are used to. This should be accompanied by a contextual code completion feature, which would allow referencing already defined functions, operators, variables, and other elements in the correct spots in the code. Last, but not least, we have strived for a type checker that would be able to find small mistakes in the code, such as calling a function with illegal arguments, or infer type of an expression. Section 3.1 describes the supported features of the language in a greater detail.

Organization

The thesis is organized in the following way:

- Chapter 1 is devoted to MPS tool. It describes what MPS is, what it can do and what its limitations are. The chapter introduces a project structure in MPS, how to define an editor for a simple language and how to tackle certain common problems.
- Chapter 2 describes the Frege language. It takes a look into the features of the language and shows their applications on concrete examples.
- Chapter 3 is dedicated to the concrete work implementation. It looks into Frege grammar and shows, how it was transformed into MPS concepts. It explores editor aspect, how it was designed with usability in mind and shows its concrete implementation. Then, code completion feature is explained. The chapter is concluded with type system, where some of the more interesting algorithms used in the work are described.
- Chapter 4 evaluates our decisions and explores the advantages and disadvantages of the projectional editor over a standard, text-based, IDE.
- In the conclusion, a brief summary of the whole work may be found, where we also strived to answer the final question, whether projectional IDEs are actually good for functional languages.

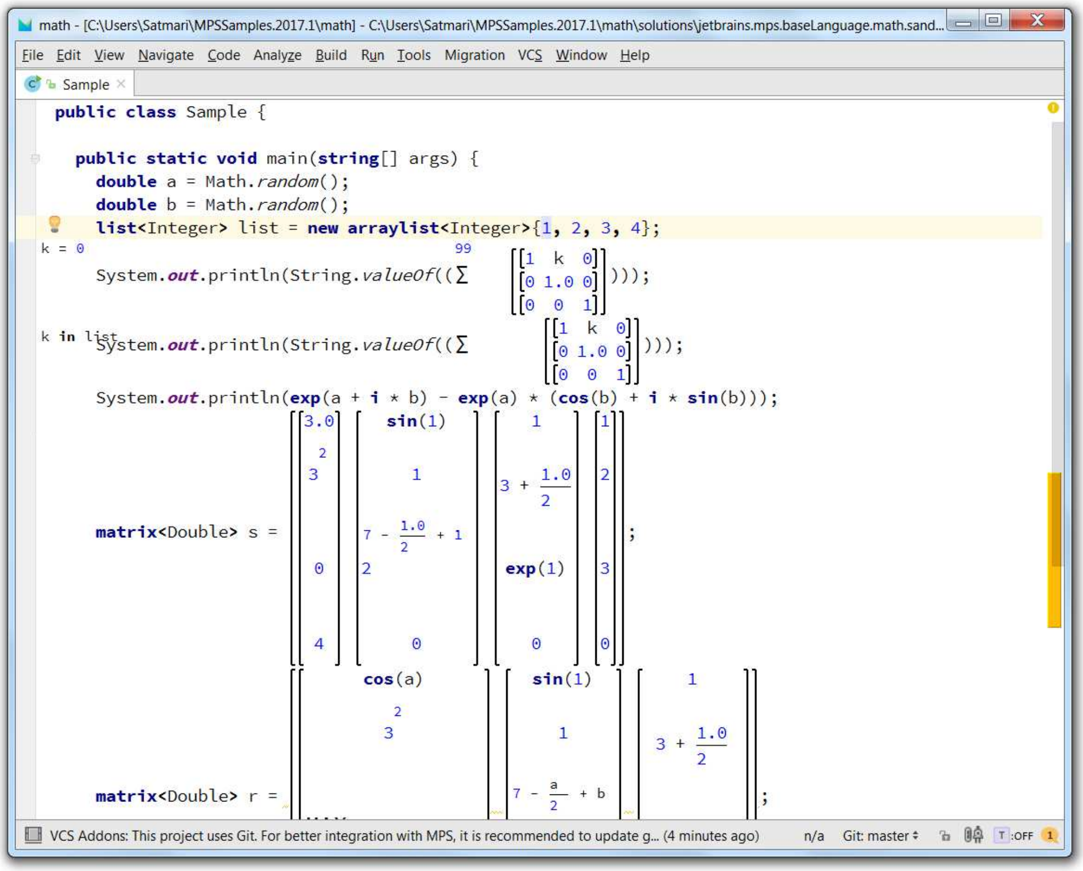
1. JetBrains MPS

JetBrains MPS is an open-source language workbench that focuses on DSLs. It is a tool that helps its users to create a new language and then write other programs in that language.

MPS has a wide range of users. The areas MPS is currently applied in include electrical engineering, data mining, insurance industry and others. The tool can be used to create new languages as well as extending existing ones. Programs written in the defined languages may then be conveniently transformed into pure text in a specific, usually generic-purpose language.

This chapter provides an informal introduction to the MPS tool and describes the usage details later.

MPS is a complex tool built around projectional editing, which means it does not treat the document as a text, but rather as structured concepts. This allows its users to create languages which involve non-parsable notations, such as decision tables, diagrams, and other controls. Additionally, several editors may be specified for a single language, thus allowing users to switch between different visual representations of a document. Figure 1.1 shows an example editor for an extension of Java language with matrices and other non-parsable controls.


math - [C:\Users\Satmari\MPSSamples.2017.1\math] - C:\Users\Satmari\MPSSamples.2017.1\math\solutions\jetbrains.mps.baseLanguage.math.sand...
File Edit View Navigate Code Analyze Build Run Tools Migration VCS Window Help
Sample

```
public class Sample {

  public static void main(string[] args) {
    double a = Math.random();
    double b = Math.random();
    list<Integer> list = new arraylist<Integer>{1, 2, 3, 4};
              99
    System.out.println(String.valueOf((∑      [[1  k   0]
k = 0                                          [0 1.0 0]
                                               [0  0   1]])));
    System.out.println(String.valueOf((∑      [[1  k   0]
k in list                                      [0 1.0 0]
                                               [0  0   1]])));
    System.out.println(exp(a + i * b) - exp(a) * (cos(b) + i * sin(b)));
    matrix<Double> s = ...;
    matrix<Double> r = ...;
```

VCS Addons: This project uses Git. For better integration with MPS, it is recommended to update g... (4 minutes ago)   n/a   Git: master

Figure 1.1: Editor for an extension of Java language with non-parsable controls

Traditional IDEs, on the other hand, involve a similar processing of the code, usually expressed in the form of plain text files, as compilers do. Traditional process of compiling written code involves lexers and parsers to read programs, which are then transformed into tree-like data structures, called ASTs. Figure 1.2 illustrates an example of such an AST for a simple arithmetic expression `(7 + 1) * 2 + 3`. After that, in the process of semantic analysis and code generation, an executable program is created. During these processes a text-based IDE may report and underline any found errors for the user.

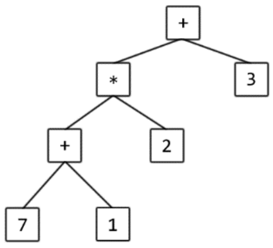


Figure 1.2: AST for a simple arithmetic expression

In contrast, in MPS, the user works with AST directly, therefore completely omitting the process of lexical analysis and subsequent parsing. This brings certain advantages:

- It may be easier to extend an existing language.
- MPS can check for type errors and other mistakes in the code at almost any time.

Extending lexers and parsers to accommodate for the changes in a language requires a certain set of skills and a deeper knowledge of the language's grammar. The process is complicated, since it requires a programmer to keep track of the possible ambiguities that may arise when defining new grammar rules for a parser (a well-known example is the 'dangling else' problem). However, in MPS the process usually only requires defining new concepts that can act as AST nodes and specifying places in the corresponding AST where the new nodes can be created. This also means that in MPS we can combine syntax of several different languages and introduce no syntax ambiguities whatsoever (this, however, may still look ambiguous to the user, if there are several different concepts with the same visual representation).

On the other hand, to check for errors in the code in a traditional IDE, one has to define a specific set of rules to deal with the incorrect syntax. Code being currently typed means, it almost certainly cannot be correctly evaluated by the standard parser for the corresponding language. Therefore, in an example below, we might not be able to tell a user that the integer and string types are incomparable between themselves, until the 'if' expression is properly finished with the required body:

```
if (1 == "")
  // a statement is required here
```

Understandably, there are ways to deal with the illustrated problem, but it requires extra effort. In MPS, this is not an issue, since the code is already ‘parsed’. Even though the body of the `if` expression is not set yet, there is already a node in the corresponding AST associated with the conditional expression inside the `if` brackets. The node then may be further checked and underlined with red color (a well-known technique of many popular IDEs, such as Visual Studio or Eclipse, to report errors). This quality is also useful when designing a smart code completion feature, which requires certain knowledge of the context surrounding the target piece of code.

Working with AST directly also carries some downsides. They mainly include worse code editing. In the example depicted in Figure 1.2, a user would need to define the AST from root to leaves, which at least in case of arithmetic expressions is not very user-friendly. Fortunately, MPS provides several functionalities to allow the language designer to define custom automatic transformations of the AST. The designer can define a transformation for a case, when, for instance, a certain node (or a whole subtree) is deleted, a specific text is written at the end (or a beginning) of a node, and so on. The MPS actions are described in Section 1.3.

We will now describe the MPS platform in a more detail.

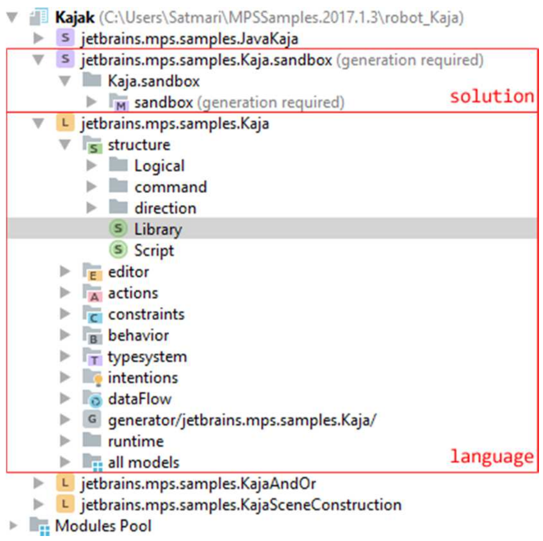
1.1 Project Structure

A project in MPS is divided into two main categories: solutions and languages.

Language is the user defined programming language. It may represent a completely new language or an extension of an existing one. Several different languages may be defined in a single MPS project. They can act as an extension of each other, or be completely independent languages.

Solution, on the other hand, is a part of the project that represents documents (a code) written in one or more of the defined languages. Sometimes, the solution acts only as a runtime support for one or more of the defined languages, to be used, for example, in the code generation process.

Figure 1.3 shows a logical view of the typical project in MPS.



```
Kajak (C:\Users\Satmari\MPSSamples.2017.1.3\robot_Kaja)
  jetbrains.mps.samples.JavaKaja
  jetbrains.mps.samples.Kaja.sandbox (generation required)
    Kaja.sandbox
      sandbox (generation required)
  jetbrains.mps.samples.Kaja
    structure
      Logical
      command
      direction
      Library
      Script
    editor
    actions
    constraints
    behavior
    typesystem
    intentions
    dataFlow
    generator/jetbrains.mps.samples.Kaja/
    runtime
    all models
  jetbrains.mps.samples.KajaAndOr
  jetbrains.mps.samples.KajaSceneConstruction
Modules Pool
```

Figure 1.3: Logical view of the project in MPS

Solution is a set of models. They act as packaging units that make it possible to reference the corresponding set of models from other solutions or languages. The *model* is simply a set of ASTs. We can imagine a single AST as a representation of the single document (an analogy to a source file in the traditional programming paradigm). The model then consists of one or more such documents.

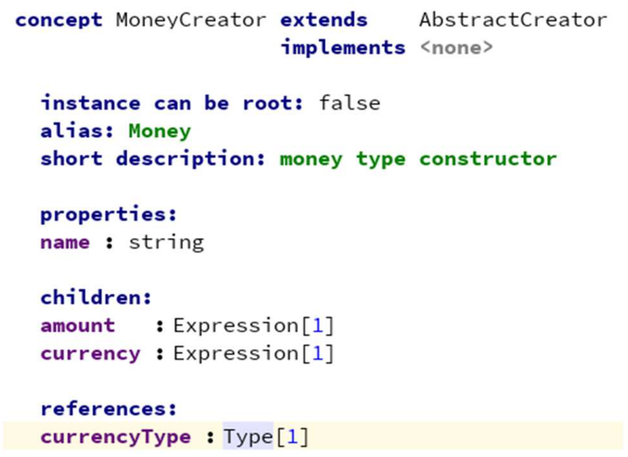
Language describes what types of ASTs can be created with it. It also includes a visual representation of each node, AST transformation actions, syntax and semantic rules together with many other 'settings'. It is divided into several categories, called aspects.

The following is the description of the most important MPS aspects which we have also used in this work.

1.2 Structure

Before we describe the structure aspect, we have to explain the notion of MPS concepts. *Concept* represents a sort of a class of AST nodes. It closely resembles working with classes and instances in many popular object-oriented programming languages, such as Java. In this analogy, the concept is a class, whereas an AST node is an instance of that class. Concepts, in a similar manner as classes, can have defined methods, properties, can extend (inherit from) other concepts or implement interfaces. They can contain fields, which are either valued types or instances of (possibly) different concepts. This way, the language designer can specify a structure of possible ASTs that can be created. A concept may also be declared abstract, in which case no AST nodes may be created directly for such a concept.

There are several different 'points of view' to the concept. The language designer can define methods for them, fields and properties, visual appearance of the AST nodes, and other. These are called aspects. Structure aspect allows to define structure of possible ASTs that can be expressed with the corresponding language. It defines what kind of AST nodes may be used in a program, what properties, children and references they may have [3]. An example of the structure aspect for a concept is shown in Figure 1.4.



```
concept MoneyCreator extends    AbstractCreator
                     implements <none>

  instance can be root: false
  alias: Money
  short description: money type constructor

  properties:
  name : string

  children:
  amount   : Expression[1]
  currency : Expression[1]

  references:
  currencyType : Type[1]
```

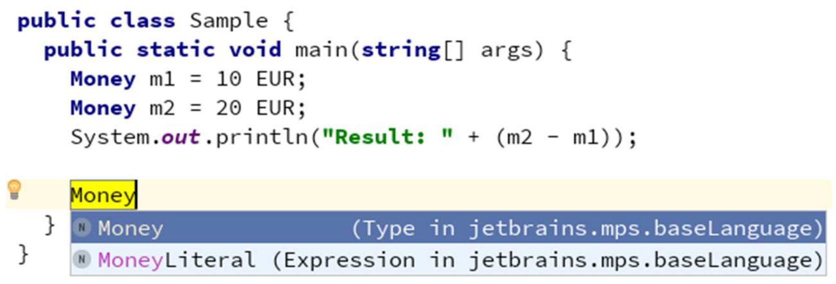
Figure 1.4: Structure aspect for a concept in MPS

A new concept should be named. This is similar to naming a class in languages like Java and must follow a similar set of naming rules. In the example depicted in Figure 1.4, the corresponding concept is named `MoneyCreator`.

The `extends` clause provides a reference to the super-concept. By default, all concepts are created with `BaseConcept` as their super-concept, but this can be changed to a more specific one. Similarly to Java, the clause encodes inheritance (or 'is-a' relationship in UML) and each concept, except the `BaseConcept` itself, directly or indirectly has to extend `BaseConcept` in the formed hierarchy. In terms of MPS, this means that if the concept `A` extends the concept `B`, it indicates that the concept `A` has all of properties, children, references, methods, and definitions from all aspects, as `B`.

Concepts can also implement interfaces by using `implements` clause. *Interface* is in this case a special *interface concept*. It is a mechanism to declare characteristics that can be used across several concept types. Unlike concepts, we cannot define an alias for them (see below) nor can they extend concepts, only other interfaces. They are mostly used for grouping properties that are commonly used together and passing them onto necessary concepts.

Alias acts as a string that triggers a built-in auto-completion menu. An example of such a menu is depicted on Figure 1.5. If the name is unambiguous (i.e. it is not a prefix of another item in the menu), an instance of the concept is immediately created. More about the menu is discussed in Section 1.3 which describes the editor aspect.



```
public class Sample {
  public static void main(string[] args) {
    Money m1 = 10 EUR;
    Money m2 = 20 EUR;
    System.out.println("Result: " + (m2 - m1));

    Money
  }
}
```

Money (Type in jetbrains.mps.baseLanguage)
MoneyLiteral (Expression in jetbrains.mps.baseLanguage)

Figure 1.5: Example of the auto-completion menu in MPS

A concept may be set to act as a root. What this means is that its instances, together with their children, may represent a single unit of a program, a single document or a source code. There should be at least one such concept for the corresponding language to make sense. A concept may be set to act as a root by using the `instance can be root` clause and setting the value to `true`.

An analogy to Java fields is represented by concept properties and children. Properties define concept's custom values (values that are owned by the concept). These are set under the `properties` section. They can be one of the following:

- **Primitive type:** integer, boolean, or string.
- **Enumeration type:** a custom enumeration data type may be created in MPS structure aspect to be used within a concept.

- **Constrained data type:** a custom constrained data type may be created in MPS structure aspect, which is a simple string type validated by a regular expression defined by the language designer.

Note that primitive types can be derived from usage of the other two options.

Children (found under `children` section), on the other hand, resemble aggregation relationship. These are the instances that belong to the instance of the current concept. While there may be references set to these AST nodes from other instances as well, in terms of their lifetime, they strictly depend on the life of the current instance. In case the current instance is removed from the AST, all of its children (and therefore children of their children, recursively) are removed as well.

Children are defined by setting a name, a concept and a cardinality. The cardinality may be one of the following options:

- `[1]`: exactly one instance of the specified concept is required.
- `[0..1]`: there may or may not be one instance of the specified concept.
- `[1..*]`: at least one instance of the specified concept is required. These then form an ordered array.
- `[0..*]`: there may be zero or more instances of the specified concept.

Expressing relationship between the nodes can be also done via references. It is only possible to create a reference to a node if that node already exists in the corresponding AST. Contrary to children, cardinality can take here only two forms:

- `[0..1]`: the reference is optional.
- `[1]`: the reference to an instance of the specified concept is required.

Where would a language designer use a reference? Consider the following piece of code in Frege:

```
f = 7
g = 1 + f
```

The code represents a definition of two constant functions returning an integer number. An (almost) equivalent piece of code could be written in Java in the following way:

```
int f() { return 7; }
int g() { return 1 + f(); }
```

We could express the corresponding AST in many different ways, but let us imagine for the sake of simplicity a root node, representing the source file, consisting of statement nodes. Both `f = 7` and `g = 1 + f` are statements. It is easy to imagine the expression, such as `1 + 2`, as a tree with a node `+` on top having two children, representing the literals `1` and `2`. But in the case of `1 + f`, it is less clear what `f` is. Using a reference here might be helpful. We already have the statement declaring what `f` is in the corresponding AST. In `1 + f` we are only applying an existing function `f`. Therefore, we are referencing the existing function `f` in the node representing the `f` operand. An example of such AST is illustrated on Figure 1.6.

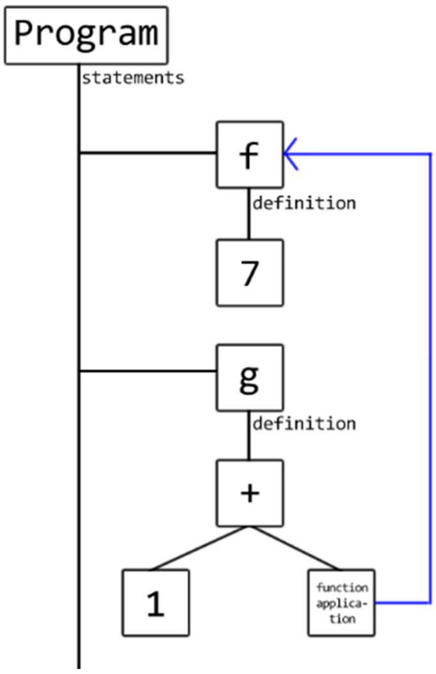


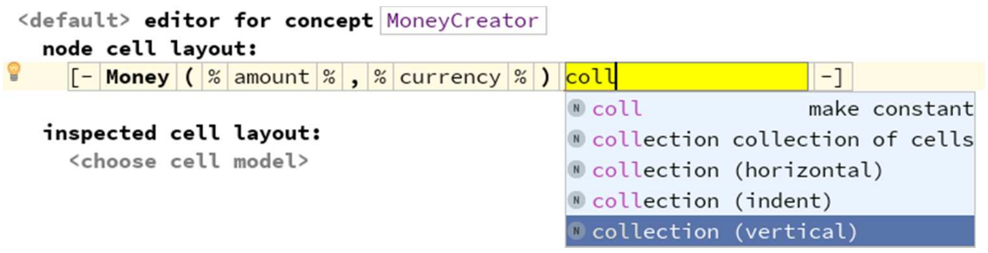
Figure 1.6: Illustration of AST for statements using a reference

1.3 Editor

Editor aspect is responsible for rendering and editing ASTs by the user of the language being created. This includes textual and graphical representation of each AST node and certain AST transformation actions. This aspect is what makes MPS a projectional editor, rather than using lexers and parsers to process the user-written code.

The easiest way to define the editor for the language is to define the editor for each concept (called concept editor). There may be several different editors defined for a single concept, which offers different views of the same concept for different needs. If a concept has no editor defined, the default one will be provided by MPS.

Figure 1.7 shows an example of the concept editor for the `MoneyCreator` concept from the previous section.



```
<default> editor for concept MoneyCreator
  node cell layout:
    [- Money ( % amount % , % currency % ) coll -]

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

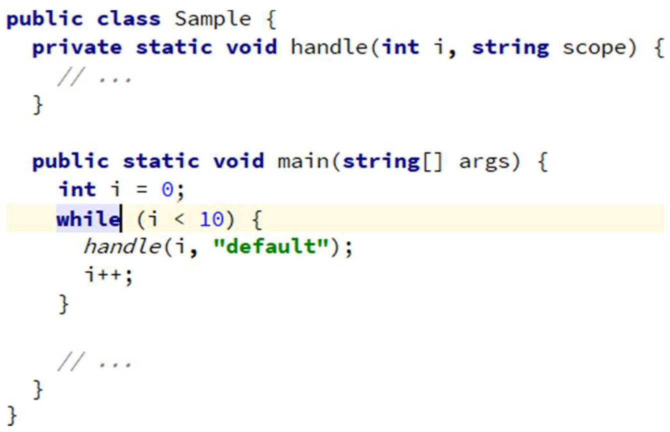
Figure 1.7: Concept editor in MPS

Another way of creating the editor for the language is to create an *editor component*. It is an editor responsible for rendering and editing only a part of an AST node. It does not focus on any single concept and as such may be reused across several concept editors to render certain parts similarly.

A usual representation of an AST node consists of, so called, editor cells. An editor cell is the smallest unit which can be used to render (and possibly edit) a certain portion of the AST node over a rectangular region in MPS editor window. For instance, `MoneyCreator` concept contains a string property called `name`. To show and edit the property value of any instance of the concept `MoneyCreator`, we specify a property editor cell for the corresponding property `name`.

The main types of editor cells include:

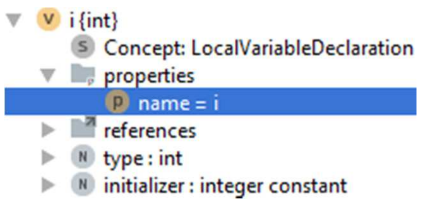
- **Constant cells**: constant cells are used to render keywords and other constant text in editor. Figure 1.8 shows an editor from the user perspective of a demo language, which is an extension of Java. On the example, we use a while-loop, which is an instance of `WhileStatement` concept. The string `while` (blue) is a constant editor cell. In Figure 1.7, the rectangle with the string `Money` also denotes the constant editor cell, but from the perspective of the language designer, who is creating the concept editor for the `MoneyCreator` concept.



```
public class Sample {
  private static void handle(int i, string scope) {
    // ...
  }

  public static void main(string[] args) {
    int i = 0;
    while (i < 10) {
      handle(i, "default");
      i++;
    }

    // ...
  }
}
```

Figure 1.8: Editor for a language that is an extension of Java

- **Property cells**: they render content of a specific property of a concept for which the editor is being defined. Editing such a cell in the editor window for a concrete AST is immediately reflected in the given property of the corresponding AST node. The cell provides automatic binding to the concept's property. On the example above depicted on Figure 1.8, a declaration of the integer variable `i` is an instance of a concept with a string property `name`. The identifier is rendered by the property cell. By invoking a node explorer window (**alt + x**), we can see that the property of the AST node is indeed set to the name we entered. The node explorer is shown on Figure 1.9.



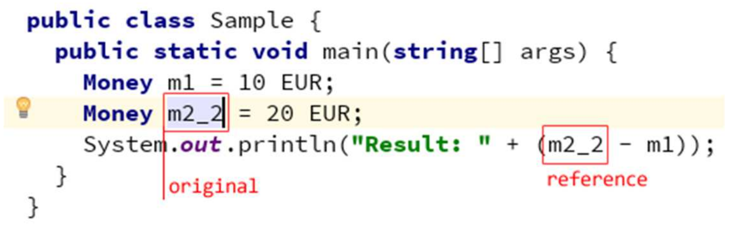
```
▼ i {int}
    Concept: LocalVariableDeclaration
  ▼ properties
      name = i
  ▶ references
  ▶ type : int
  ▶ initializer : integer constant
```

Figure 1.9: Node explorer for the declaration of the integer variable `i`

- **Child cells**: these cells delegate the rendering of a specific concept's child (or a set of children) to their corresponding concept editors. The concrete behavior of such a cell depends on the child's cardinality:
    - `[1]`: the editor cell is always present
    - `[0..1]`, `[0..*]` **or** `[1..n]`: child nodes are bound to their corresponding editors and removing a child in MPS editor window results in removing it from the parent node of the corresponding AST as well
    - `[0..*]` **or** `[1..*]`: the children's corresponding concept editors are separated by a specified textual delimiter

  On Figure 1.8 we can see a method invocation represented by the statement `handle(i, "default")`. The provided two arguments are children of cardinality `[0..n]` of a concept `Expression` and are represented by child cells delimited by a comma.

- **Referent cells**: referent cells are used to display an attribute of the referenced node from the given concept. As in the case of property cells, they are mapped to a certain property of the referenced node in the AST. However, they can only reflect the property of the original node, but not affect it. Figure 1.10 shows an editor with `Money` variable declaration. It has a form of a subtree with an AST node representing the variable's name (originally `m2`). The variable is then referenced in an expression, which prints a subtraction of the variable by another variable into the standard output. The change in the variable's name (`m2_2`) is immediately reflected into the reference. This way MPS support renaming refactoring feature out of the box.



```
public class Sample {
  public static void main(string[] args) {
    Money m1 = 10 EUR;
    Money m2_2 = 20 EUR;
    System.out.println("Result: " + (m2_2 - m1));
  }
}
```

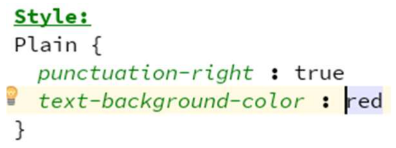
Figure 1.10: Editor for an extension of Java language depicting the usage of referent editor cells

- **Collection cells**: wrapper-like cells to contain other editor cells are called collection cells. They affect visual arrangement of the cells being rendered. There are three main types of collection cells:
    - **Horizontal cells**: cells enwrapped are placed horizontally in row.

- **Vertical cells**: cells enwrapped are placed vertically.
- **Indent cells**: cells enwrapped are placed in a text-like manner.

There are several other types of editor cells. Here we only described the most-used ones from the perspective of this work.

The editor cells may also be rendered in different ways. This can be changed by using *editor styles*. Applying editor style could be described as analogous to applying CSS (Cascading Style Sheets) styles to DOM nodes in HTML and XML documents. This allows the language designer to change the cells' visual properties, such as text color, background color, spacing, padding as well as functional aspects, such as editor cell being editable or read-only and many others. Figure 1.11 shows a usage of editor style for a selected editor cell.



```
Style:
Plain {
  punctuation-right : true
  text-background-color : red
}
```

Figure 1.11: Editor style for a selected editor cell

1.3.1 Editor Actions

So far we have described how we can customize appearance of each AST node. Now we will discuss editor actions, how we can allow automatic transformations of the corresponding AST and how to easily add new AST nodes to the code tree.

A lot of developers are used to write programs in text-based IDEs or just in a plain-text editor. To simulate such a behavior, MPS comes with a notion of editor actions. We have to remind the reader that MPS keeps the code at tree-like data structures at all times. This means that what seems in a text-based editor as a trivial operation (such as adding a new operator and an operand to an arithmetic expression) is a non-trivial AST transformation in MPS.

Let us consider a simple arithmetic expression: `7 - 1 * 2 + 3`. In a plain-text editor, a normal user would write the expression from left to right. In MPS, however, the expression has to be encoded within an AST, and as such has to be entered from root node to the leaves. In this particular example, a user would need to create an instance of the concept representing the `+` operator. This creates a binary tree. The right operand is an AST node representing the literal `3`. The left operand is a new subtree representing the expression `7 - 1 * 2`, which has to be, again, entered from root node to the leaves, starting with the concept representing the `-` operator.

Understandably, the mentioned approach is not very user-friendly. However, we can use MPS editor actions to create the editor where such an arithmetic expression may be entered from left to right. We will show the approach for the concrete expression from the high-level point of view.

1. First, the user types the literal `7`. That is a very simple unary expression and no further work is to be done here.

2. Then, user hits `-`. MPS immediately creates a binary expression subtree, where root is the operator `-`. `7` is put as its left operand and the focus is set on the right operand, so the user may edit that.

3. User types in `1`, which only concludes the editing of the right operand. Figure 1.12 illustrates the AST in its current state.

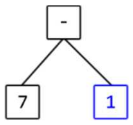


Figure 1.12: Illustration of an AST for the arithmetic expression `7 - 1`

4. Then, however, follows the operator `*`. User is now editing the right child of the AST corresponding to the expression `7 - 1`. MPS, therefore, takes a look at the parent's operator's precedence. It is clear that `-` is less precedent, than `*`. Thus, a subtree for binary operator `*` is created, `1` is put as its left child and a focus on the right child is set. The subtree is placed in the original stead of the node representing operand `1`. Figure 1.13 illustrates an AST after finishing the current step.

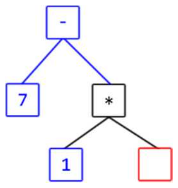


Figure 1.13: Illustration of an AST representing the arithmetic expression `7 - 1 * unset-operand`

5. User types the literal `2`, which concludes the editing of the right operand for the operator `*`.

6. Finally, user types in operator `+`. MPS again takes a look on the parent's operator which is `*` and has a higher precedence. The new subtree, therefore, has to be created elsewhere. The parent of the node representing operator `*` is, however, `-`. While `-` has the same precedence as `+`, all of the operators are left associative which means the new subtree has to be created even on the higher level. MPS creates the subtree, puts the current AST corresponding to the expression `7 - 1 * 2` as its left child and sets the focus on its right child. The current AST is depicted on Figure 1.14.

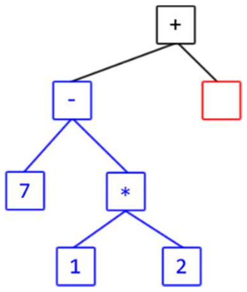


Figure 1.14: Illustration of an AST representing the arithmetic expression `7 - 1 * 2 + unset-operand`

7. Typing `3` only finishes editing of the right child and the expression is concluded.

In the description above, we were always editing a specific editor cell corresponding to a single node in the AST. We always handled an event of writing a specific textual pattern to the right of a certain editor cell. This is just what MPS allows us to do. These type of actions are usually referred to as *transformation menu actions* and we will describe them in a more detail in the following section.

Another important type of actions are *substitute menu actions*, which allow the user to substitute a certain AST node (or a whole subtree) for a different AST node. These actions are usually invoked when a certain text is written in place of an AST node, which we want to automatically substitute for something else. The substitute menu actions are described in Section 1.3.3.

1.3.2 Transformation Menu Actions

Transformations menu actions provide a way to manipulate an AST when a certain textual pattern is entered, usually either left or right of a certain editor cell. They allow us to replace a certain AST node for a different one, change a whole code subtree, or otherwise manipulate the corresponding data structures.

From a certain perspective we could say that the transformation menu actions are specific set of event handlers. The handlers are specified in a general-purpose programming language, which is based on Java (so called *BaseLanguage*). This allows for almost any type of AST manipulation and offers a lot of flexibility.

An example of a usage scenario can be a concept, which represents a certain type of expression enclosed within brackets, for example, `(x1)`. However, the corresponding AST node may be changed to represent either:

- A tuple, which has a form of several expressions within round brackets separated by commas, e.g. `(x1, x2, x3)`
- A list, which has a form of several expressions within round brackets separated by colons, e.g. `(x1 : x2 : x3)`

We want to change the AST node based on the user-entered text. If the expression `x1` is followed by a comma, we will replace the node for an instance of the tuple concept. In case the user enters a colon, an instance of the list concept should be created. Figure 1.15 shows the concrete implementation of the corresponding transformation action, which is also described in a greater detail below.

The actions have to be always associated with a certain editor cell and the corresponding concept. However, the editor actions in general apply only to the following types of cells:

- Constant cells
- Property cells

- Referent cells

In the example above, the editor for the concept representing the expression enclosed within round brackets consists of three parts:

1. A constant cell representing the left round bracket
2. A child cell representing the expression
3. A constant cell representing the right round bracket

This means the transformation action described in the example above has to be created for the concept representing the expression, rather than the whole bracketed expression (we will assume here that the concept representing the expression consists only of one of the three mentioned types of editor cells, for example, the property cell denoting the identifier `x1`).

Transformation menu action can be created as either:

- A default transformation menu for a concept
- A named transformation menu

A default transformation menu is associated with a specific concept. The action is triggered by entering a specific textual pattern either left or right (we can choose either of the two options) of all three types of the mentioned editor cells the corresponding concept editor consists of. For instance, if we created a default right-side transformation menu for the bracketed expression concept, the action would be triggered by entering the specified textual pattern right of both of the bracket symbols, but would not be triggered by entering the pattern right of the expression.

Additionally, every concept is implicitly associated with a default transformation menu. If the language designer does not provide one explicitly, the transformation menu defined for the closest super-concept is assumed. If none are defined, the one implicitly defined for `BaseConcept` is used.

A named transformation menu is an additional action associated with a specific concept. Unlike the default menu, it is not associated with all of the three types of the mentioned editor cells in the corresponding concept editor implicitly. Instead the language designer has to attach the action explicitly to the concrete editor cells he or she likes. However, the same restriction for the editor cell types applies here as well, i.e. the language designer cannot attach the named action to child editor cells, only to the constant, property and referent cells.

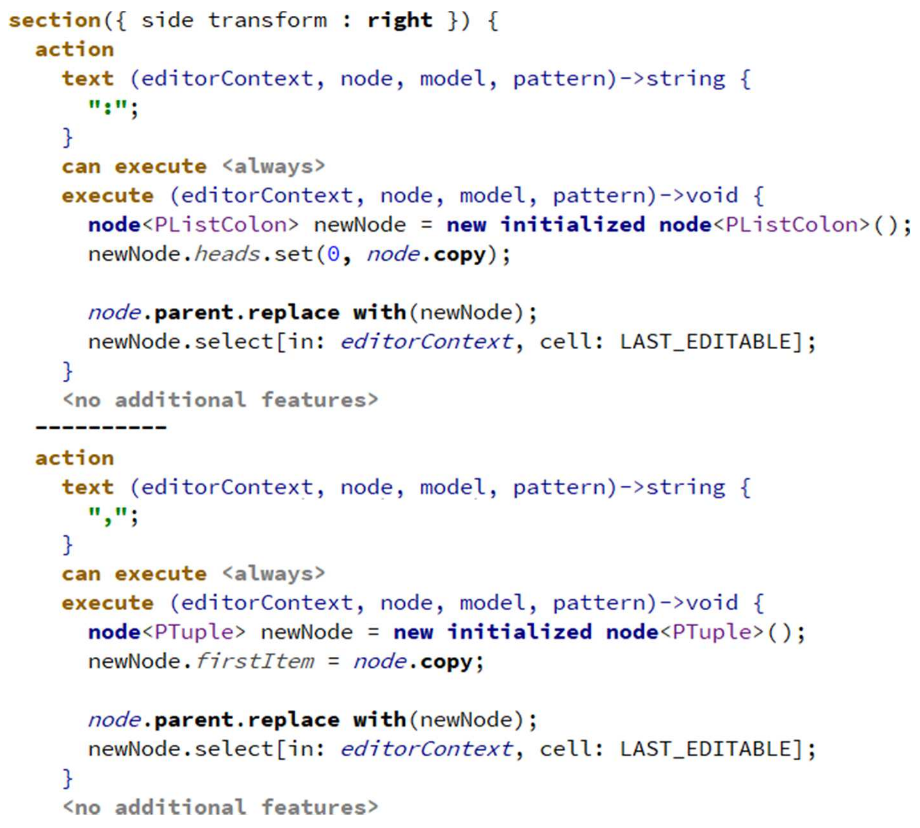
Let us now describe the implementation process of the transformation menu action on the example for the bracketed expression. We will create a default transformation menu for the concept representing the expression. Then we specify the section, i.e. where the transformation should take place. There are several options, but for the purpose of this work, either the action is triggered upon typing a text right of an editor cell, or left of an editor cell. (We chose right. In Figure 1.15 this is represented with the clause `section({ side transformation: right })`.)

Then, we define the `action` from the three main sections:

- **Text:** represents a string that triggers the current action. This is the string a user can type either right or left of the associated editor cells. It can be either a constant, or a piece of code which returns the string that triggers the action.

- **Can execute:** a piece of code that is executed once the action is triggered. If the code returns `false`, the current action is prevented from execution. However, due to how MPS works, it is mostly best to leave the section empty, as returning `true` indicates the possibility to execute the action even if not triggered by the current `Text`.
- **Execute:** the specific handler of the current action, written in a higher-level Java-like language. It specifies the concrete transformation of the AST.

On Figure 1.15 we can see the concrete implementation consisting of two separate actions. Each action performs its own transformation of the bracketed expression node, either to the tuple or the list. The former expression `x1` is copied and placed as their first item. The bracketed expression node is a parent of the current expression, which is why we have to use the statement `node.parent.replace with(newNode)`. The last line of the both handlers denotes setting a focus on the newly created AST node - on its last editable editor cell.



```
section({ side transform : right }) {
  action
    text (editorContext, node, model, pattern)->string {
      ":";
    }
    can execute <always>
    execute (editorContext, node, model, pattern)->void {
      node<PListColon> newNode = new initialized node<PListColon>();
      newNode.heads.set(0, node.copy);

      node.parent.replace with(newNode);
      newNode.select[in: editorContext, cell: LAST_EDITABLE];
    }
    <no additional features>
  ----------
  action
    text (editorContext, node, model, pattern)->string {
      ",";
    }
    can execute <always>
    execute (editorContext, node, model, pattern)->void {
      node<PTuple> newNode = new initialized node<PTuple>();
      newNode.firstItem = node.copy;

      node.parent.replace with(newNode);
      newNode.select[in: editorContext, cell: LAST_EDITABLE];
    }
    <no additional features>
```

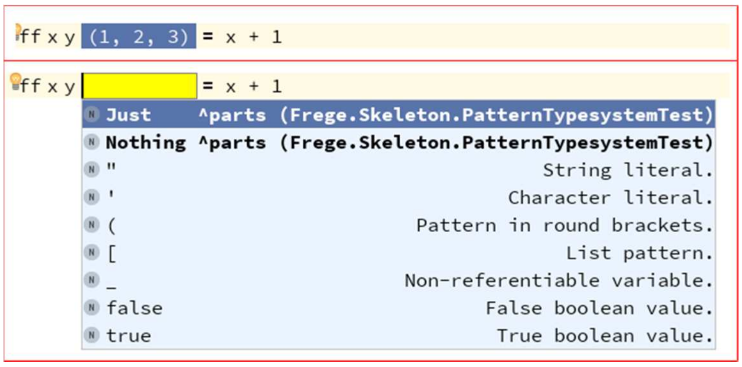
Figure 1.15: The default transformation menu for the concept representing the expression, which is also a child of the bracketed expression concept

There is also a way to reuse transformation menu actions. Instead of specifying an `action`, a language designer may use `include` statement. It includes a specific default or named transformation menu. Furthermore, a transformation menu aimed for a different concept, than the one being currently dealt with, may also be included. Consider the example above and a scenario, where we want to execute the actions defined for the expression concept also when the user types the coma or colon symbol right of the bracket symbols. We may simply create a default transformation menu for the concept representing the bracketed expression and include the default transformation for the expression concept.

1.3.3 Substitute Menu Actions

Substitute menu actions define transformations to some parts of the AST, where one node (or a whole subtree) is substituted by another node (or a whole subtree).

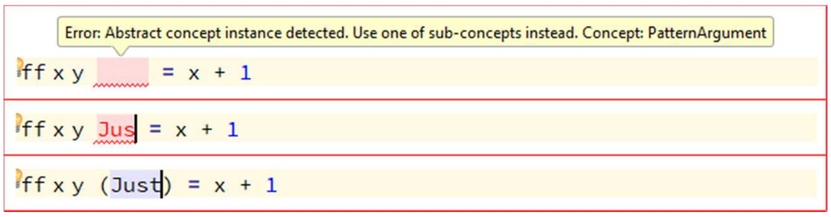
Typically substitute actions are triggered by user when pressing **ctrl + space** in the editor. This invokes the completion menu that contains options that, when selected by the user, will replace the current AST node under caret. Substitute menu actions allow the language designer to add specific items into the completion menu as well as overriding the behavior of the ones included in the menu by default. The default substitute menu is provided by MPS for all concepts, when the caret's position is in front of a node, or the whole node is selected. Figure 1.16 depicts such a scenario in Frege-IDE [4].



```
ff x y (1, 2, 3) = x + 1

ff x y           = x + 1
  Just    ^parts (Frege.Skeleton.PatternTypesystemTest)
  Nothing ^parts (Frege.Skeleton.PatternTypesystemTest)
  "                                     String literal.
  '                                  Character literal.
  (                           Pattern in round brackets.
  [                                       List pattern.
  _                           Non-referentiable variable.
  false                              False boolean value.
  true                                True boolean value.
```

Figure 1.16: The default substitute menu provided by MPS for a selected node

To trigger a substitute action, a user may also simply enter the text in place of an AST node from the completion menu for one of its items. This, understandably, does not work in every case, as not every AST node is completely editable (consider, for instance, an AST node with non-editable constant editor cells). However, instances of abstract concepts, which are created by default for the concepts with children of such abstract concepts, are editable. They are highlighted by reddish rectangle to denote an error and that MPS expects an instance of a concrete concept instead. Figure 1.17 captures the usage scenario. First we have the AST node, which is an instance of an abstract concept. Then we enter a text from the substitute menu, `Just`. Upon hitting the last character, the substitute action is triggered in the same way, as if the user selected the item manually from the menu and pressed **enter** key.



```
Error: Abstract concept instance detected. Use one of sub-concepts instead. Concept: PatternArgument
ff x y       = x + 1

ff x y Jus = x + 1

ff x y (Just) = x + 1
```

Figure 1.17: Using substitute menu actions by entering text directly

The completion menu follows the following scheme:

- All concepts applicable in the given context are displayed in the menu. This follows the structure aspect of the language project. For example, if a concept `A` contains a child of an abstract concept `B` and there are two concrete concepts, which extend `B` – `B1` and `B2`, then `B1` and `B2` are added to the menu. If `B11` extends `B1`, it is also added to the menu.
- Abstract concepts are not included in the menu.
- Concepts, for which their constraints do not allow their presence in the current place in the code, are not included either. More about the constraints aspect is discussed in Section 1.6.
- Smart references are not added to the completion menu, but rather all of the referenceable items are included instead.

Smart reference is a term we use for concepts that contain only a single reference and nothing else. For instance, it may be a concept representing a variable in an expression. In most programming languages, a variable has to be declared first, before it can be used:

```
int i;
boolean b = i > 10; // i is a reference here
```

Such a concept would consist only of the reference to the concept representing the corresponding variable declaration `int i`. MPS then instead of adding the concept itself to the substitution menu adds to the menu all of the referenceable variable declarations. Thus, in the example above, the menu would be populated with the presentation of the AST node `i` (among other visible variables in the given context).

The completion menu may be altered by creating a substitute menu for a concept. If the language designer creates a default, empty, substitute menu for a concrete concept, it will not be populated by that concept, regardless of the context where the menu is invoked. This feature may be used to treat concepts, such as *`EmptyStatement`*. A typical program is usually a series of statements. For the sake of simplicity, let us consider that each statement goes onto a new line. To allow empty lines in the editor, we would create an `EmptyStatement` concept. However, it does not make much sense to allow creating the `EmptyStatement` instances from the completion menu. Instead each line should be an instance of `EmptyStatement` by default and easily rewritten to a different statement. To prevent `EmptyStatement` populating the completion menu, we would do just that - that is, creating an empty default substitute menu for the concept.

The language designer may also specify *substitute actions* and *wrap substitute menus* inside a substitute menu for a concept. We will describe them in a more detail, as they are important and heavily used in Frege-IDE.

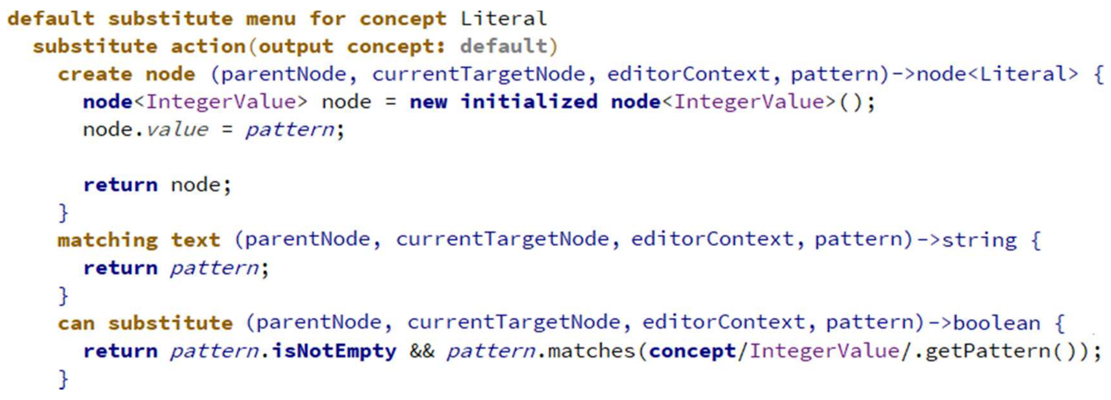
Substitute actions populate the completion menu by a new entry at all places, where the current concept (for which we are defining the substitute menu) would be applicable. A language designer then specifies a custom handler, which has to return a new AST node for the current concept, or a concept which extends, directly or indirectly, the current concept (in the analogy with OOP languages, the new AST node has to have the type of the current concept).

We will demonstrate substitute menu actions on an example. Let us assume an abstract concept `Literal`. We have two concepts which extend `Literal`: `IntegerValue` and `BooleanValue`. What we want is to automatically create a concrete AST node, where a node for `Literal` concept is expected. If the user types an integer number, an instance of `IntegerValue` should be created, whereas if user types `true` or `false`, an instance of `BooleanValue` should be created.

From the point of view of completion menu, in places where an instance of `Literal` concept is expected, there should be three items in the menu available: two for `BooleanValue` (`true` and `false`) and one representing a generic `IntegerValue`. We will handle `BooleanValue` in the following way:

1. Set `BooleanValue` as an abstract concept.
2. Create two concrete concepts, which extend `BooleanValue`, i.e. `TrueValue` and `FalseValue`, representing the corresponding values.
3. Set aliases to `true` and `false` for the corresponding concepts representing the boolean values. This populates the completion menu with the defined aliases instead of the names of the corresponding concepts.

The `IntegerValue` is trickier, because there is no single value to represent the concept with. To solve the problem, we will create a default substitute menu for the `Literal` concept and add a single substitute menu action. Figure 1.18 shows an implementation of such a menu. We describe the details below.



```
default substitute menu for concept Literal
  substitute action(output concept: default)
    create node (parentNode, currentTargetNode, editorContext, pattern)->node<Literal> {
      node<IntegerValue> node = new initialized node<IntegerValue>();
      node.value = pattern;

      return node;
    }
    matching text (parentNode, currentTargetNode, editorContext, pattern)->string {
      return pattern;
    }
    can substitute (parentNode, currentTargetNode, editorContext, pattern)->boolean {
      return pattern.isNotEmpty && pattern.matches(concept/IntegerValue/.getPattern());
    }
```

Figure 1.18: Implementation of a default substitute menu for `Literal` concept

The substitute menu action consists of defining the following sections:

- **Create node**: this is a custom handler of the current substitute action and has to return a new node for the current concept.
- **Matching text:** a string that triggers the current substitute action, when typed. This is also the string that will be displayed on the left side of the invoked completion menu.
- **Can substitute:** a boolean telling the MPS whether the current substitute action may be executed when triggered.

In the case of our example, the `IntegerValue` is a concept with a single property representing the user-entered integer value. Therefore, in `create node` section, we simply create a new node and set its value property to be equal to the user-entered text. The `matching text` section is set to return whatever value the user types. This may make not much sense, but it is important to understand that we cannot represent all integer numbers with a single string. Finally, `can substitute` checks whether the user-entered string actually represents an integer value. It tries to match the string against a regular expression capturing integer values, and if successful, returns `true`.

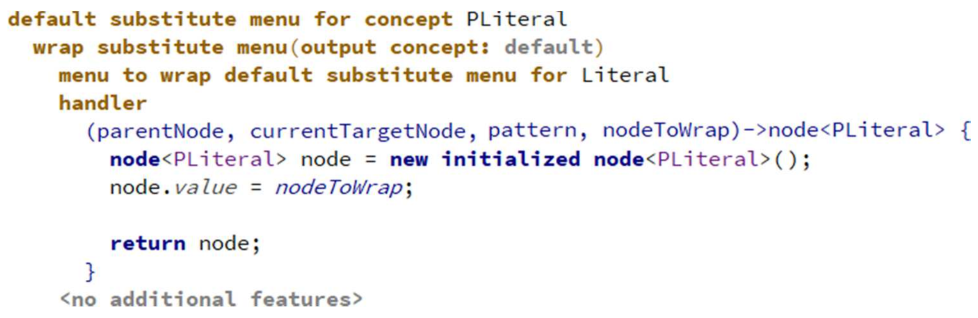
Wrap substitute menu populates the completion menu by a different concept as a 'replacement' for the current concept. The corresponding handler still has to return an instance of the current concept, however, to conform to the defined structure.

Let us consider the following scenario. We have `Literal` concept from the example above, which, according to the structure, extends an abstract `Expression` concept. Then we have another abstract concept, `Pattern`, completely independent from `Expression`. However, we want to be able to use `Literal` also in places, where `Pattern` is expected. Since `Literal` may only extend one of the two concepts, we would need to create a new `Literal` concept, which would extend `Pattern`. Copying the `Literal` together with its sub-concepts would create a lot of code duplicity and the language would quickly become unmaintainable.

A different solution is to create a 'wrapping' concept, let us call it `PLiteral`. The concept extends `Pattern` and has a single child of cardinality `[1]` of type `Literal`. However, we want to preserve everything about the `Literal` concept from the example above, i.e. automatic substitution to `IntegerValue` and `BooleanValue`. In the current state, the user of the language would first need to create an instance of the `PLiteral` and only then would he or she be able to use the defined substitute menu for `Literal` (this is the 'top-down' approach of creating the AST).

The language designer may use, however, the option of defining the wrap substitute menu. He or she would specify that in places where concept `PLiteral` is expected, the completion menu may be populated by the entries from completion menu for `Literal` concept instead. Selecting any of the corresponding entries from the completion menu would create an instance of `Literal`, and then the handler, defined by the language designer, would take the AST node and enwrap it by an instance of the `PLiteral` concept.

Figure 1.19 depicts the implementation details of the wrap substitute menu for the `PLiteral` concept. The language designer selects a concept of which the completion menu should be copied (`menu to wrap default substitute menu for`), then specifies the handler which wraps the original AST node by a new instance of the current concept.



```
default substitute menu for concept PLiteral
  wrap substitute menu(output concept: default)
    menu to wrap default substitute menu for Literal
    handler
      (parentNode, currentTargetNode, pattern, nodeToWrap)->node<PLiteral> {
        node<PLiteral> node = new initialized node<PLiteral>();
        node.value = nodeToWrap;

        return node;
      }
    <no additional features>
```

Figure 1.19: Default substitute menu for the `PLiteral` concept with the wrap substitute menu

1.3.4 Cell Action Map

Cell action map is a custom defined event handler associated with an editor cell. Unlike the previously mentioned types of actions, these allow the language designer to define a handler for events, such as editor cell selection, cell removal, pressing a concrete keyboard key when the editor cell is focused, and so on.

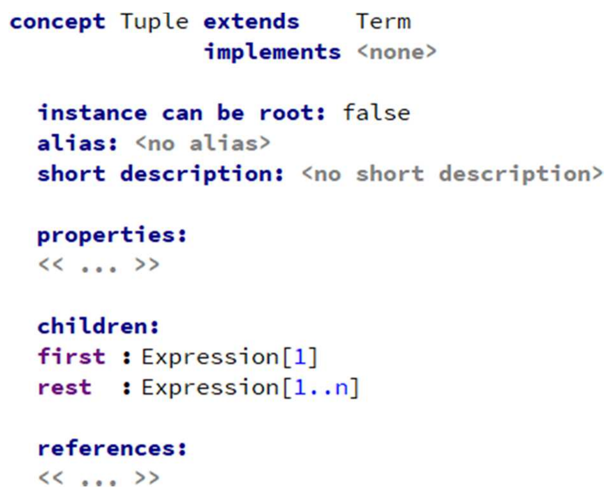
Consider the example from Section 1.3.2. We have these types of concepts:

- A concept representing the bracketed expression, e.g. `(x1)`
- A concept representing the tuple, e.g. `(x1, x2, x3)`
- A concept representing the list, e.g. `(x1 : x2 : x3)`

However, this time, we are faced with the opposite problem – how to change the AST node, representing either a tuple or a list, back to the simple bracketed expression, upon removal of the last item?

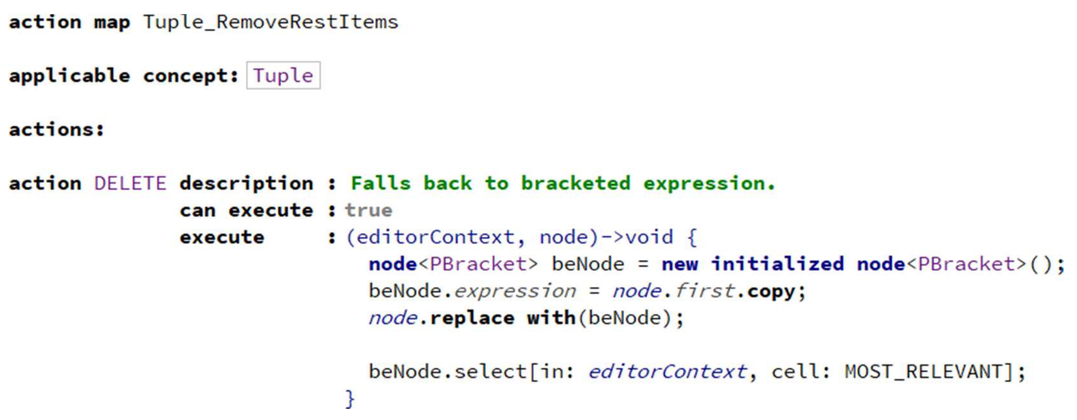
We will demonstrate the usage of the action map on the `Tuple` concept. `Tuple` is a concept containing at least two children of type `Expression`. Figure 1.20 provides an example implementation of its structure aspect in MPS.

In the corresponding concept editor we associate the child editor cells for `rest` children from Figure 1.20 with the new cell action map we named `Tuple_RemoveRestItems`. In the cell action map, we define a new handler for `DELETE` action. The handler itself is relatively simple – we only create a new AST node for the bracketed expression, set the expression between the brackets to be equal to the last remaining item in the `Tuple` AST node and replace the `Tuple` node with the newly created bracketed expression. Finally, we set the focus on the newly created AST node in the editor. Figure 1.21 depicts an implementation of the cell action map.



```
concept Tuple extends    Term
              implements <none>

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

  properties:
  << ... >>

  children:
  first  : Expression[1]
  rest   : Expression[1..n]

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

Figure 1.20: Implementation of structure aspect of `Tuple` concept



```
action map Tuple_RemoveRestItems

applicable concept: Tuple

actions:

action DELETE description : Falls back to bracketed expression.
              can execute : true
              execute     : (editorContext, node)->void {
                              node<PBracket> beNode = new initialized node<PBracket>();
                              beNode.expression = node.first.copy;
                              node.replace with(beNode);

                              beNode.select[in: editorContext, cell: MOST_RELEVANT];
                            }
```

Figure 1.21: Implementation of the cell action map for `Tuple` concept

To conclude the editor section, editor actions provide a flexible way to build a user-friendly editor that can mimic many features of a traditional, text-based, editor. However, it is impossible to allow the completely same behavior, since a user is actually editing the AST data structure and not the text he or she sees. This means that almost every editing feature has to be implemented manually. The language designer should, however, optimize the editor for the most common cases, at least.

1.4 Behavior

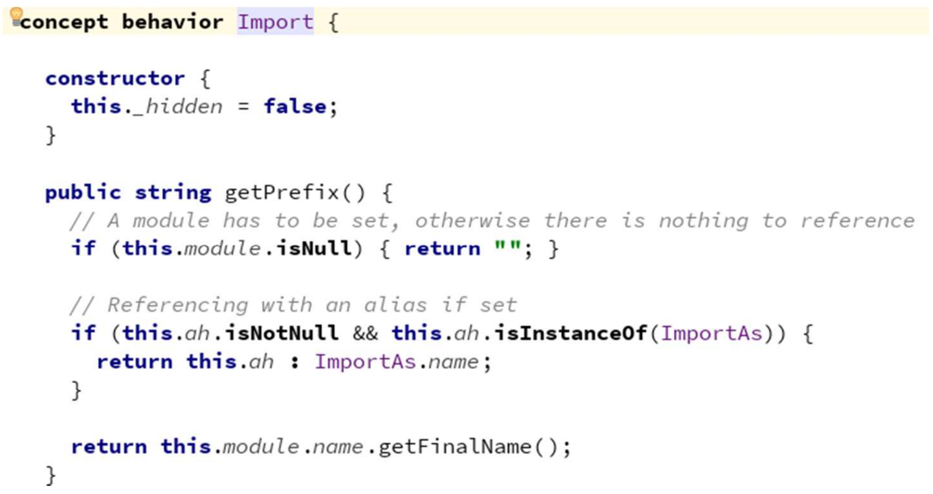
Behavior aspect allows to, simply said, define methods on concepts. If we take the analogy with OOP further, then structure aspect allows a language designer to declare classes and their fields, while the behavior aspect allows to declare and implement their methods, including *constructors*.

In behavior aspect, constructor is a block of code which is executed when a new node of the corresponding concept is created. However, certain exceptions exist, when the constructor is not, in fact, executed. These mainly include creating an instance of the concept by using other means in the MPS BaseLanguage, than the statement `new initialized node<MyConcept>()`.

Similar concept of methods as in OOP languages, such as Java, is present here as well. A concept may be associated with several methods with strictly defined visibility (`public`, `protected`, or `private`). Methods that can be overridden in sub-concepts, have to be marked `virtual`. Static methods exist here too. They are methods not attached to any instance of the concept, but rather have to be called on the concept itself.

Important characteristic of the behavior aspect is that it allows to traverse the AST being created. The language designer can easily inspect parent and children of any node as well as nodes' references.

An example of the concept's behavior aspect is depicted on Figure 1.22. The corresponding `Import` concept, which represents import declaration in Frege, has defined the constructor setting its property to a default value. The `getPrefix` method returns the import's alias. More about the Frege import declaration is discussed in Section 2.9.



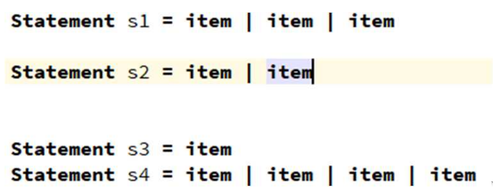
```
concept behavior Import {

  constructor {
    this._hidden = false;
  }

  public string getPrefix() {
    // A module has to be set, otherwise there is nothing to reference
    if (this.module.isNull) { return ""; }

    // Referencing with an alias if set
    if (this.ah.isNotNull && this.ah.isInstanceOf(ImportAs)) {
      return this.ah : ImportAs.name;
    }

    return this.module.name.getFinalName();
  }
```

Figure 1.22: Example of a concept's behavior aspect in MPS

1.5 Intentions

Intentions aspect allows to define special user interface elements (called intentions) that allow executing predefined actions in certain places in the code. They usually perform some modification of the current AST.

Let us assume a program which consists of a series of statements. Each statement is placed on a single line, but some lines may be empty. Statements consist of several items. In this case, when the caret is positioned at the end of a statement, pressing **enter** key is, from the user's perspective, ambiguous. Either it should add a new item to the current statement, or a new line. The scenario is illustrated on Figure 1.23.

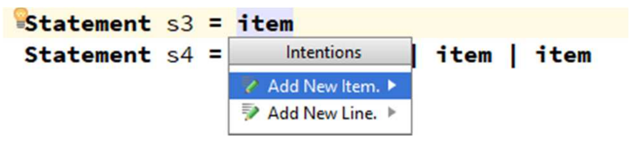


```
Statement s1 = item | item | item

Statement s2 = item | item


Statement s3 = item
Statement s4 = item | item | item | item
```

Figure 1.23: A program in MPS consisting of a series of simple statements

We can approach the problem by letting the user decide. A user would invoke a menu that would let him or her decide whether an item should be added to the current statement, or a new line created below the statement.

The intentions menu is generally invoked by pressing **alt + enter** keys. The menu may contain several items and the selection of a concrete intention is confirmed by pressing the **enter** key. An example of the menu is shown on Figure 1.24.



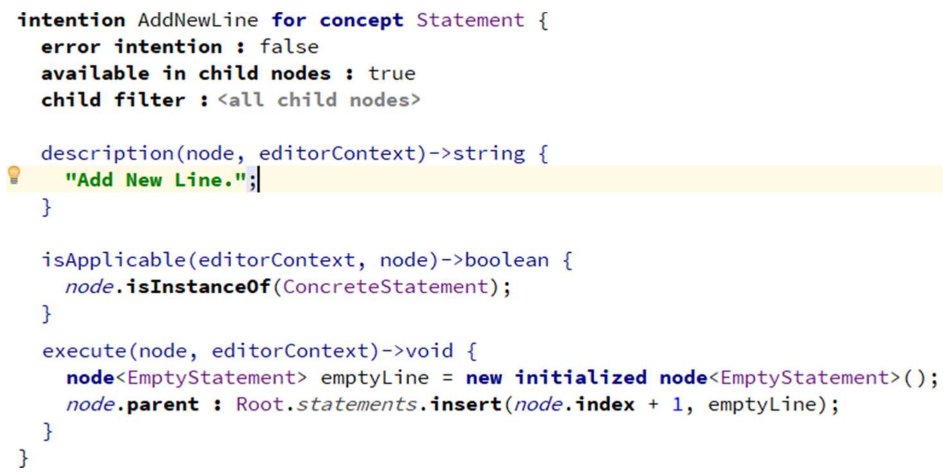
```
Statement s3 = item
Statement s4 = | item | item
```

Figure 1.24: Intentions menu in MPS

Standard type of intention is defined for a specific concept. There is also another type of intention (surround-with type) we have not used in this work and will therefore not describe.

When created, the corresponding intention will be added to all intentions menu invoked on that corresponding concept (this means that the caret has to be positioned on the editor cells associated with the given concept at the time of the menu invocation). The intention may also be executed within the subtree defined by the corresponding concept, if `available in child nodes` is set to `true`.

An example of the intention implementation is show on Figure 1.25. The language designer has to provide the intention's description, which will be shown in the invoked menu, and the handler to specify the required action. Additionally, he or she can define the context in which the intention may be displayed in the menu, by returning a boolean value inside the `isApplicable` section.



```
intention AddNewLine for concept Statement {
  error intention : false
  available in child nodes : true
  child filter : <all child nodes>

  description(node, editorContext)->string {
    "Add New Line.";
  }

  isApplicable(editorContext, node)->boolean {
    node.isInstanceOf(ConcreteStatement);
  }

  execute(node, editorContext)->void {
    node<EmptyStatement> emptyLine = new initialized node<EmptyStatement>();
    node.parent : Root.statements.insert(node.index + 1, emptyLine);
  }
}
```

Figure 1.25: New intention definition in MPS

1.6 Constraints

Constraints aspect lets the language designer declare constraints that help him or her control where nodes of the language are allowed. They also allow to specify and put a set of restrictions on valid values (properties) of AST nodes and to define scopes for referenced nodes.

Scope is an object which defines a list of potential targets that can be referenced. Nodes outside the list may not be set a reference to. Additionally, scope object also helps to locate a suitable target from the given list based on what the user has entered in the corresponding place in the editor. When no scope is defined in the concept's constraint aspect, all nodes of the appropriate type are considered eligible. The auto-completion menu is filled with nodes based on this rule.

Concept constraints are divided into several sections:

- **Can be child:** the section allows specifying a boolean method which returns, whether a node of the current concept can be a child in a specific AST context. If the method returns `false`, then the node will not be suggested in the auto-completion menu. Sections `Can be parent` and `Can be ancestor` work in the similar way.
- **Property constraints:** property constraints allow restricting a set of values of a concept property. In a sub-section `is valid` the language designer may specify a boolean method for checking the corresponding property's value. If the method returns `false`, the value is considered invalid and MPS marks the associated editor property cell with a reddish rectangle.
- **Referent constraints:** in this section the language designer controls how references are established to nodes of the concept. He or she may restrict what nodes will be referenceable from the given concept by specifying the scope object.

1.6.1 Scope

Scope is an object in MPS which defines a list of potential targets that can be referenced. It is an (indirect) instance of the abstract class `Scope` in the BaseLanguage in MPS.

A language designer may specify his or her own implementation of `Scope`. The new class must inherit from the `Scope` class provided by MPS implicitly and implement the necessary abstract methods. Notable methods from the class include:

- **`public abstract sequence<node<>> getAvailableElements(string prefix);`**
  returns all of the nodes from the scope that begin with the string `prefix`.
- **`public abstract node<> resolve(node<> contextNode, string refText);`**
  returns a node, if the entered string `refText` can unambiguously determine a referenceable node from the current scope.

1.7 Typesystem

Typesystem aspect makes it possible to report semantic errors to the user of the language. On its highest level, it could be said it contains mechanisms to check for both non-type related and type related rules.

Non-type related rules are called checking rules. These serve the language designer to implement custom semantic error checks. For instance, consider a type declaration statement in Frege language:

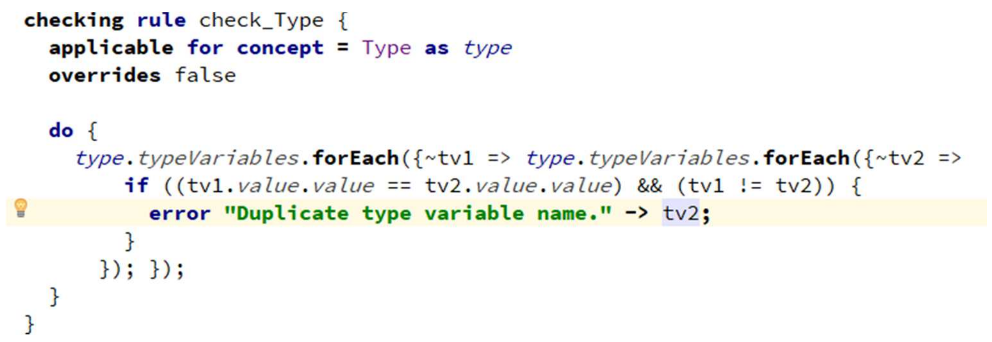
```
type MyType a b = [Int] -> a -> b
```

The statement declares a new type `MyType` with `a` and `b` being type variables. Type variables have to have different names to be distinguishable and thus the following statement is invalid in Frege:

```
type MyType a a = [Int] -> a -> a
```

To check and report the error, the language designer can create a checking rule for the concept representing the type declaration statement. A checking rule is a simple method, associated with a specific concept, which is executed by MPS automatically for each AST node of the concept in the current document. The method is executed mainly when the AST node is changed in any way, or when the document is opened. The error is reported to the user by using `error` statement in MPS BaseLanguage. The language designer specifies a string message and the AST node which caused the error. If the `error` statement gets executed, the MPS underlines the corresponding node with red color in the editor to denote the problem.

Figure 1.26 provides the implementation of the checking rule in our Frege-IDE for the concept representing the type declaration statement. We compare each two type variables between themselves and report an error in case two have the same name.



```
checking rule check_Type {
  applicable for concept = Type as type
  overrides false

  do {
    type.typeVariables.forEach({~tv1 => type.typeVariables.forEach({~tv2 =>
        if ((tv1.value.value == tv2.value.value) && (tv1 != tv2)) {
          error "Duplicate type variable name." -> tv2;
        }
      }); });
  }
}
```

Figure 1.26: The checking rule in Frege-IDE for the concept representing the type declaration statement

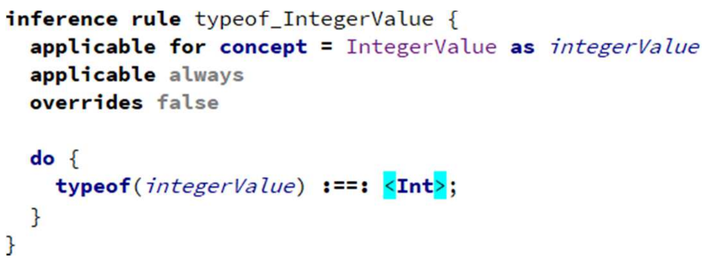
Type related (typesystem) rules offer a declarative way to express rules which support type calculations. The language designer can let MPS calculate types of expressions in the runtime and upon finding inconsistencies, MPS will report errors to the user automatically.

MPS supports several types of declarative rules. We will describe the ones we used in this work, which include inference and subtyping rules.

Inference rules are created to calculate a type of a node for the given concept. These can also be used to enforce a type, i.e. to perform a type check. The rule consists of these sections:

- **Name:** determines name of the current typesystem rule.
- **Applicable for:** used to specify the concept which the rule applies for.
- **Do:** defines the rule for the given concept. The rule is written in a variant of the MPS BaseLanguage, which is extended with statements regarding the typesystem aspect.

Figure 1.27 shows an example of the inference rule for a concept representing an integer literal. The section `Do` contains a single statement, which tells MPS the type associated with the concept. We describe the statement `typeof(integerLiteral) :==: <int>` in a more detail below.



```
inference rule typeof_IntegerValue {
  applicable for concept = IntegerValue as integerValue
  applicable always
  overrides false

  do {
    typeof(integerValue) :==: <Int>;
  }
}
```

Figure 1.27: The inference rule for a concept representing an integer literal

The statement is a special form of an assignment. There are several different operators in MPS the language designer can use:

- `:==:` The operator tells MPS that the type on the left hand side must be the same, as the type on the right hand side. This performs both the check and the assignment.
- `:<=:` This tells MPS that the type on the left hand side is a sub-type of the type on the right hand side.
- `:~:` Usage of the operator tells MPS that the type of operands on either side of it are weakly comparable.

`typeof(integerLiteral)` denotes the type of an instance of the `IntegerValue` concept. The inference rule is executed for each AST node of the current concept.

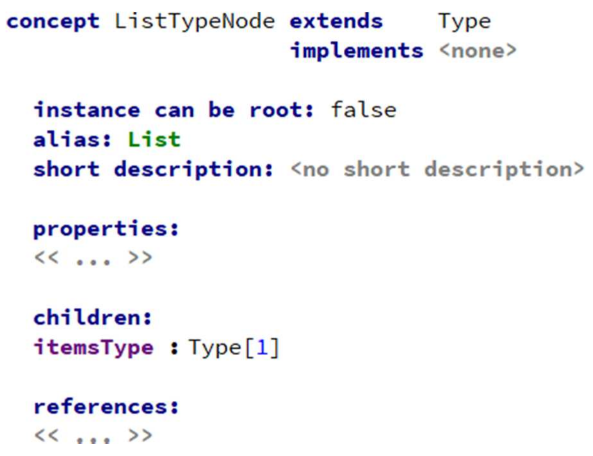
The `<int>` part denotes a *quotation*. It is used when the language designer needs to create nodes of concepts in a language (even the language he or she may be now creating) via MPS BaseLanguage. The equivalent piece of code without the quotation would be as follows:

```
node<IntTypeNode> intTypeNode = new initialized node<IntTypeNode>();
typeof(integerLiteral) :==: intTypeNode;
```

Anything displayed inside the quotations symbols <…> is what the node would look like as if an actual editor of the corresponding language was used to edit the AST. `int` in this case is the textual representation of the concept `IntTypeNode` in its concept editor.

The concept `IntTypeNode` is not a part of the MPS BaseLanguage. It is, however, a concept which extends `Type` concept from the BaseLanguage. The concept may be created in a new language, which is then added to the dependencies of MPS typesystem aspect. It represents a new type.

To represent a more complex type, such as an array of items of a certain type, a similar approach would have to be used. In Frege, the equivalent language construct is called *a list*. The concept representing its type is a new concept extending the `Type` concept from the BaseLanguage. It should contain a single child determining the type of its items. The child is, again, an instance of the `Type` concept. The structure aspect of such concept is depicted on Figure 1.28.



```
concept ListTypeNode extends    Type
                     implements <none>

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

  properties:
  << ... >>

  children:
  itemsType : Type[1]

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

Figure 1.28: Structure aspect of the concept representing the list type in Frege-IDE

When the operator `:==:` is used to compare a type of an AST node, MPS checks the whole tree for equivalence. In the example above, a list of integer items is, according to the operator `:==:`, different from a list of double items.

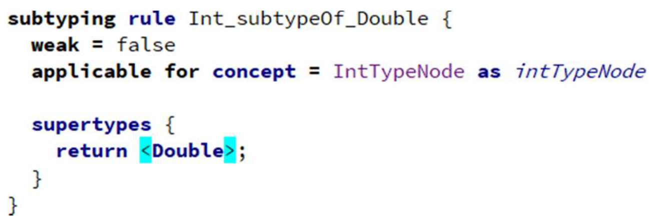
To check whether a list of integer items is assignable to a variable denoting a list of double items, we have to use the sub-typing operator `:<=:`. There is no concrete assignment statement in Frege language, but certain semantic checks of function definition statements against their type annotations behave similarly. For the sake of simplicity, let us consider a concept representing the assignment statement as in languages, like Java, which consists of left and right hand side. For the right hand side to be assignable to the variable on the left, it must be of the same or a 'more concrete' type, than the variable. The following statement illustrates an inference rule of such a concept:

```
typeof(assignment.rightExpression) :<=: typeof(assignment.leftVariable);
```

Let us consider a new concept representing the numeric type double, `DoubleTypeNode`. So far, there is no defined type relation between `DoubleTypeNode` and `IntTypeNode` and an attempt to assign an integer list to a double list variable would MPS underline with red color denoting the error.

To specify that integer is a sub-type of double, we create a *sub-typing rule.*

Sub-typing rule is a simple method which returns a list of instances of `Type` concept that are 'more abstract' than the current type. In the case of `IntTypeNode`, we return the `DoubleTypeNode` instance. Figure 1.29 shows the concrete implementation for our example.



```
subtyping rule Int_subtypeOf_Double {
  weak = false
  applicable for concept = IntTypeNode as intTypeNode

  supertypes {
    return <Double>;
  }
}
```

Figure 1.29: The subtyping rule for `IntTypeNode` concept

By implementing the rule, we have also allowed for sub-typing comparison of the list type. Now, MPS would have no objections against assigning an expression of an integer list to a variable of double list type.

In certain cases, however, the provided mechanisms are not enough to perform type checks. Sometimes a language designer needs to inspect the inferred type of an expression in a more detail. For this, MPS has a notion of `when concrete` block.

Since MPS offers only a strictly declarative way of defining types of AST nodes, it is not certain when a node will have its type inferred. However, a language designer may use `when concrete` statement to surround a block of code which will be executed only when the type of the given node is already known. The surrounded code is executed in a separate thread, which means the rest of the code in the corresponding typesystem rule will continue with its execution independently.

1.8 Textgen

The optional aspect component of a language, textgen, allows to define a mapping from AST nodes to a text. The feature allows the translation of the code written in MPS to a plain text. The user can then compile the code using a standard compiler to an executable program.

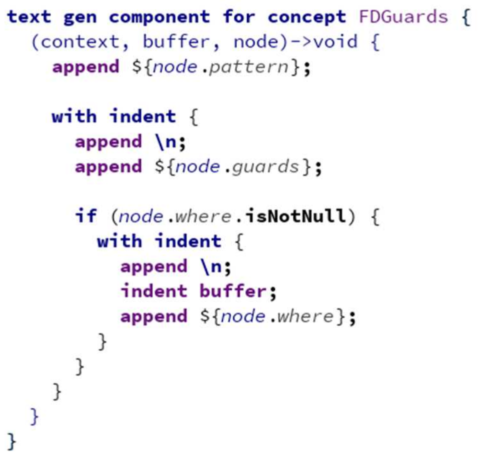
Textgen can be triggered for a specific document by using a **right mouse button** in the editor space and selecting **Preview generated text**.

The language designer has to define the textgen aspect for each concept independently. He or she has to specify a string which is put into a buffer for each AST node encountered. If the corresponding concept contains children, their textgen should be invoked manually as well in a recursive manner, down to the leaf concepts.

Definition of the textgen consists of a single method written in the MPS BaseLanguage. It specifies, what should be outputted to the buffer, by using `append` statement. The indentation of certain portions of the code may be manually increased by using `with indent` block. Last, but not least, it should call the textgen of child concepts, again, by using `append` statement.

Figure 1.30 depicts an example of a usage of the textgen aspect for a concept in Frege-IDE. The corresponding concept `FDGuards`, representing a function definition with guards, does not output anything on its own, but rather calls the textgen of its children. Its `pattern` child is outputted on a single line, then the `guards` are printed starting from the following line, indented. `where` is printed out only if it is actually defined, since the cardinality of the child is set to `[0..1]`.

Note that when using the textgen, an error arises when MPS tries to call the textgen for a concept which does not have this aspect specified. This can be useful when a given AST is currently incomplete or in an erroneous state. That is why abstract concepts usually should not implement it.



```
text gen component for concept FDGuards {
  (context, buffer, node)->void {
    append ${node.pattern};

    with indent {
      append \n;
      append ${node.guards};

      if (node.where.isNotNull) {
        with indent {
          append \n;
          indent buffer;
          append ${node.where};
        }
      }
    }
  }
}
```

Figure 1.30: Example of the textgen aspect for the concept `FDGuards` in Frege-IDE

2. Frege

Frege is a variant of Haskell language, targeted for the JVM platform. It is a purely functional programming language, has a strong static type system with global type inference and non-strict evaluation. The language compiles to Java and runs naturally on the JVM. This way it can be used inside any Java project.

In this chapter, we describe the Frege syntax and what the most common patterns of writing programs in Frege are. We describe differences between Frege and Haskell and mention certain approaches when programming in functional languages in general. It is important to note that Frege is a robust and complex language, which is why we could not include the full set of language constructs in the implemented IDE. Certain features not mentioned in this chapter were omitted.

Frege was designed by Ingo Wechsung, who named it after the German mathematician, logician and philosopher, Gottlob Frege. Its syntax is very close to that of Haskell, with only small differences. The following is a brief description of the main differences.

In general, Frege could be considered a subset of Haskell language. Certain features are missing, such as the foreign function interface, which allows Haskell to interact with code written in other languages [5]. Instead there are language constructs to make Java types and methods usable (all primitive types are simply Java types).

The Frege-Prelude library, an equivalent of Haskell-Prelude library which defines many standard types (for instance, `Maybe`), functions and operators (`<=`, `!=`, `&&`, ...), has many functions, type classes and types known from Haskell. However, Frege uses the Java APIs whenever possible, so certain aspects of the language may feel different. For example, implementation of type classes is incomplete and multi parameter type classes are not supported by Frege at all. Additionally, the support for `newtype` declaration (an algebraic data type with exactly one constructor) is missing as well as `deriving` clause for data type declarations, and a few other keywords. The string value in Frege, unlike in Haskell, is not a list of characters, but an instance of the Java class `java.lang.String` [6]. Furthermore, Frege does not have any operator data constructor other, than colon `:` to separate the head and the tail of a list. This, however, allows the user to define in Frege even such custom operators that begin with the colon character, which is not valid in Haskell (e.g. `:-:` is a valid custom operator function in Frege).

There are several other minor differences between Haskell and Frege. However, they mostly do not affect this work in any way, which is why we do not describe them.

The following sections describe Frege syntax on several small examples as well as overview of certain principles used when programming in a functional language. For the most part, the described syntax is also supported in Frege-IDE. Though recommended to read, people familiar with Haskell or Frege may skip this part.

2.1 Hello, World!

The following piece of code is an example of a ‘Hello, world!’ program in Frege:

```
module Hello where

greeting friend = "Hello, " ++ friend ++ "!"

main args = do
    println (greeting "world")
```

This code would compile to `Hello.class` and `Hello.java` with a regular Java entry point method `main`. Moreover, the `Hello.class` would have the method `public static String greeting(String ...) {...}` that one can call from Java, or any other JVM language.

Just like in Haskell, the function `greeting` is pure, which means it is stateless and does not have any side effects. For the same given input parameters it always returns the same result. This is a great advantage of functional languages that basically allow the results of such functions to be cached. Function `main`, however, is not pure. Since it corresponds to the `main` function in Java language, it may produce side effects, like printing to the standard output, which it actually happens to do so in this concrete example.

2.2 Pattern Matching

An important aspect of programming in both Frege and Haskell is pattern matching. When we define a new function, we may define different variants of its implementation for different input arguments. To elaborate, consider the following definition of the function `charToName`:

```
charToName :: Char -> String
charToName 'a' = "Albert"
charToName 'b' = "Broseph"
charToName 'c' = "Cecil"
charToName _ = "No Name"
```

The main idea behind the function above is to provide the caller with a human name beginning with the given character. (Here we only provide definition for the first three characters of English alphabet, albeit it should suffice for the demonstration.)

The first line of the program tells us that `charToName` is a function accepting a single character argument and outputting a string (we discuss Frege types and type annotations in Section 2.3).

Then, we provide for each character a specific definition. The wildcard underscore character _ matches any input. This way, if we were to call the function `charToName` with an input `'a'`, it would return `"Albert"`, but for `'z'` character we would get `"No Name"`.

The ordering of the definitions is important here. Moving the definition for the pattern with wildcard `_` above the definition for input character `'a'` would result in all calls to `charToName` returning the `"No Name"` string.

Regarding the pattern matching in Frege, it is also important to mention variables. These 'formal parameters' are also patterns; it's just that they never fail to match a value. This is in a certain way similar to the wildcard pattern `_`. However, as a side effect of the successful match, the formal parameter is bound to the value it is being matched against. For this reason, patterns may not contain multiple variables with the same identifier [7].

Based on this, we can create a function returning a second element of any three-item-tuple. A *tuple* is simply an ordered sequence of items of (possibly) different types, separated by comma. The following are examples of constant functions returning tuples:

```
tupleExample1 = (1, 'a')

tupleExample2 = (1 + 1, 2.7, true && false, 'a')

tupleExample3 = (1, 2.7, true, ("hello", 'a'), 'z')
```

To explain the third function `tupleExample3`, the expression `(1, 2.7, true, ("hello", 'a'), 'z')` is a tuple of five items, fourth of which is another tuple of two items.

Our function, returning the second element of a three-item-tuple, would then look like this:

```
second (_, x, _) = x
```

When calling the `second` function with a three-item-tuple argument, the second value is automatically bound to the variable `x`. This is what we then return on the right hand side of the definition. We call this mechanism data deconstruction.

2.3 Types

Frege is a strongly and statically typed language. If the types are not specified by the programmer, they are automatically inferred. To provide types for the function `greeting` from Section 2.1, the user can write:

```
greeting :: String -> String
```

We call this signature a type annotation of the function. Here, we denote that function `greeting` accepts a single argument of the type `String` and returns a result of the type `String` too.

Additional types, which are also supported by Frege-IDE include:

- `Bool`: represents boolean values (`true`, `false`).
- `Char`: represents a single utf-8 character.
- `Int`: represents integer numbers.

- `Double`: represents floating point numbers. It is an equivalent to Java `double` type.
- `Tuple`
- `List`
- Custom algebraic data types
- Function type

2.3.1 Tuples

A tuple, as mentioned in the previous section, is a sequence of items of (possibly) different types, separated by a comma. Here we show the examples of constant functions returning tuples together with their type annotations:

```
tupleExample1 :: (Int, Char)
tupleExample1 = (1, 'a')

tupleExample2 :: (Int, Double, Bool, Char)
tupleExample2 = (1 + 1, 2.7, true && false, 'a')

tupleExample3 :: (Int, Double, Bool, (String, Char), Char)
tupleExample3 = (1, 2.7, true, ("hello", 'a'), 'z')
```

2.3.2 Lists

Besides tuples, there is also another fundamental data structure to hold multiple values in Frege - *a list*. A list is a homogenous data structure (i.e. all of its elements need to be of the same type). An example of a function returning a list is as follows:

```
listExample :: [Int]
listExample = [4, 8, 15, 16, 23, 42]
```

In this specific example we return a list of integer numbers, which is defined by enumeration. For ordinal data types, however, we can also specify a range of values:

```
rangeListExample :: [Int]
rangeListExample = ['a' .. 'z']
```

`rangeListExample` is a function which defines a list of 26 characters from ‘a’ to ‘z’.

Generally, each list can be separated into two parts: the head and the tail. *Head* is a single element at the beginning of the list. *Tail* is the remaining part. If the list contains only one element, the tail is an empty list. A completely empty list has to be represented by `[]` symbols.

This picture of lists is important, because it allows us to pattern-match it against the data constructor operator :. For instance, consider the following example:

```
getTop :: [String] -> String
getTop [] = "No elements"
getTop (x:xs) = x
```

The function above returns the first element from the given list of strings. If the corresponding list is empty, the function returns `"No elements"`. The pattern `(x : xs)` matches the input list in the following way:

- Head of the list is bound to the variable `x`.
- Tail of the given list is bound to the variable `xs`.
- The matching is successful only if the given list is not empty.

The function may be invoked in the following way:

```
frege> getTop ["hey", "hi", "hello"]
hey
```

This principle allows us to work with lists in an actually useful way, otherwise we would need to match them against an exact pre-defined pattern.

The following example shows a definition for a function which joins two lists into a single one:

```
listJoin [] ys = ys
listJoin (x:xs) ys = x : (listJoin xs ys)
```

The implementation follows a recursive approach, which is a common practice in most of the functional languages. The first line of the definition is a trivial join of two lists, first of which is empty. The result in this case is simply the second list. In the second line, however, we say that the result of the join is a new list with first item the same as the first item of the given first list. The tail of the new list is a result of the recursive application of the function on the remaining part of the first list and the whole second list.

A usage example is as follows:

```
frege> listJoin [1, 2, 3, 4, 5, 6] [7, 8, 9]
[1,2,3,4,5,6,7,8,9]
```

There is also another way of declaring a list, using so called *list comprehension*. We can think of this as an analogy to declarative programming, where we define what data we want, input sets, which specify where to get the data from, and condition to filter out the unwanted records.

For example, a function returning a list of Pythagorean triplets can be defined as follows:

```
pt = [(x, y, z) | x <- [1..15], y <- [1..15], z <- [1..15], x < y, y < z,
x*x + y*y == z*z]
```

The part before the vertical line symbol, `(x, y, z)`, denotes the single item of the resulting list. Expression `x <- [1..15]` specifies that the input set for variable `x` is `[1..15]`. Lastly, `x*x + y*y == z*z` denotes a condition so that only the relevant items are included in the resulting list.

A usage example is as follows:

```
frege> pt
[(3, 4, 5),(5, 12, 13),(6, 8, 10),(9, 12, 15)]
```

2.3.3 Custom Algebraic Data Types

Custom algebraic data types allow to create a completely new type in the Frege program. Consider the following example:

```
data Days = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday |
Sunday
```

The statement above introduces the new type, called `Days`, to be used inside the corresponding Frege program. `Monday`, `Tuesday`, and the other parts of the declaration are called *constructors*. They denote the value of the declared data type.

The following function definition shows an example usage of the new type:

```
getNextDay :: Days -> Days
getNextDay (Monday) = Tuesday
getNextDay (Tuesday) = Wednesday
…
```

In this particular example, we have defined only a simple enumeration type. However, we can also wrap additional data as demonstrated by the following example:

```
data Point = Point Double Double
…

movePointX :: Point -> Double -> Point
movePointX (Point x y) _x = Point (x + _x) y
```

We have defined a simple type representing a point in the 2D space and a function moving that point by the given value in x-axis. In this case, our custom data type has only a single constructor. Notice that the name of the data type and the constructor are treated completely independently by the compiler and therefore may be named equally.

However, the constructor arguments do not necessarily have to be of primitive types. Consider the following, more advanced, example:

```
data Shape = Circle Point Double | Rectangle Point Point
…
surface :: Shape -> Double
surface (Circle _ r) = pi * sqr r
surface (Rectangle (Point x1 y1) (Point x2 y2)) = abs (x2 - x1) * abs (y2
  - y1)
```

The data type `Shape` contains additional data of type `Point`, which we have defined earlier in this section. It is used to denote a center of a circle, or an upper-left and a bottom-right point of a rectangle.

Furthermore, we can also construct recursive data structures, as illustrated by the following piece of code in Frege:

```
data Tree = Nil | Node Int (Tree) (Tree)
```

The data type `Tree` represents a binary-tree-like data structure, where each node contains a single integer value and up to two child nodes. `Nil` constructor represents an empty node (a leaf), which does not contain any values.

To provide more flexibility for custom data types, we can also use the notion of type variables. In the example below, `a` represents a type variable. We can use any type in its place. Instead of then having to define several data types for several different functions, we can reuse the type while specifying, what `a` is, for each one:

```
data Maybe a = Just a | Nothing
…

getTopIntList :: [Int] -> Maybe Int
getTopIntList [] = Nothing
getTopIntList (x:xs) = Just x

getTopCharList :: [Char] -> Maybe Char
getTopCharList [] = Nothing
getTopCharList (x:xs) = Just x
```

2.3.4 Type Synonyms

Type synonyms are similar declarations to algebraic data types. Unlike the data types, however, they do not introduce any new types into the program, but only wrap a complex type by a single type name. Additionally, the similar mechanism of type variables as in the data type declarations is supported. The example below demonstrates a usage of a type synonym to declare a new type `Stack`, which is represented by a simple list of items:

```
type Stack a = [a]
…

pop :: Stack Int -> Stack Int
pop [] = []
pop (x:xs) = xs
```

2.3.5 Function Types

In functions, functions can also be used as arguments. Consider, for instance, a list of integer numbers. We may want to apply a certain function to each item of the list and return the new list of integers created this way. The function applied is unknown to us beforehand, but we can still implement the high-level 'mapping' function. We can do it in the following way:

```
map :: (Int -> Int) -> [Int] -> [Int]
map _ [] = []
map ff (x:xs) = (ff x) : (map ff xs)
```

`(Int -> Int)` denotes the function argument. This represents a function accepting a single integer argument and returning a new integer. We then apply this function, bound to the variable `ff` upon a successful match, to each item of the input list and return the new list.

2.3.6 Generic Type

An important aspect of type annotation is providing an interpreter with a ‘generic type’. Consider the example from Section 2.3.5. We may want to implement the mapping function more generally, for all list types, not just the list of integer numbers. Understandably, the implementation of the function is completely equivalent to the `map` function from the mentioned section. What has to be changed, is its type annotation:

```
map :: (a -> b) -> [a] -> [b]
```

What this says is that the function `map` accepts a function that takes an argument of a certain type `a`, and returns an element of a possibly different type `b`. Then `map` accepts as for its second argument the list of items of the type `a` and returns a list of items of the type `b`.

2.4 Operators

Both Haskell and Frege provide a lot of flexibility, when it comes to infix operators. There are several standard built-in operators, such as arithmetic addition `+`, subtraction `-`, comparison operators `==`, `>=`, and so on. It is, however, possible to define almost any custom operators consisting of allowed symbols. These include: `# $ % & * + . / < = > ? @ \ \ ^ | ~ : -`

For example, we can create a custom operator `+++` for adding two integer numbers, while also incrementing the result by `1`. The implementation is as follows:

```
(+++) :: Int -> Int -> Int
a +++ b = a + b + 1
```

The newly defined operator is simple to use in expressions. For instance, the following is a definition of a constant function `ff` returning integer `6`:

```
ff = 2 +++ 3
```

Since operators are basically binary functions, there are no major differences between the two. The type annotation differs only in obligation to wrap the operator inside the brackets.

A user may also specify the custom operator's precedence and associativity. By default, the custom operator is non-associative and has a precedence of 16. There are 16 levels of precedence in Frege (numbered 1 to 16), the higher number denoting the more prioritized operator inside an expression.

The precedence and associativity may be changed by writing the following statement:

```
infixl 5 +++
```

The first part specifies the associativity. Three modes of associativity exist:

- **Left associativity:** the statement begins with keyword `infixl`.
- **Right-associativity:** the statement begins with keyword `infixr`.
- **Non-associative operator:** the statement begins with keyword `infix`. There must not be several non-associative operators used in a single expression in a sequence, with exception of using brackets or changing precedence of a sub-expression in another way.

It is important to note that a combination of several operators with both types of associativity (left and right) with the same precedence in a single expression is not possible and results in a compilation error.

Function application, constructors, and bracketed expressions have all higher precedence, than any operator. For an operator to be used as a function argument, it has to be enwrapped inside brackets. Compare the following two statements:

```
ff = 1 +++ foo 2 3 +++ 7
ff = 1 +++ foo 2 3 (+++) 7
```

In the first statement, we apply the function `foo` with arguments `2` and `3`. In the second statement, we apply the function `foo` with arguments `2`, `3`, operator `+++` and `7`.

2.5 Currying

Currying is the technique of translating the evaluation of a function that takes multiple arguments, into evaluating a sequence of functions, each with a single argument. In Frege, a function may use for its implementation another function, while providing only some of its arguments. The technique may be demonstrated by the following example:

```
multiplyThree :: Int -> Int -> Int -> Int
multiplyThree x y z = x * y * z
multiplyByEighteen = multiplyThree 2 9
```

In the example above, the function `multiplyThree` takes three arguments and returns their product. `multiplyByEighteen` is then implemented by applying the function `multiplyThree` on two out of three possible arguments.

It is interesting that we do not have to provide in the implementation of `multiplyByEighteen` any arguments. This is the main idea behind the technique, otherwise we would need to write the following piece of code:

```
multiplyByEighteen x = multiplyThree 2 9 x
```

Currying may be also used with operators, and also applies in cases of a partial function application, as demonstrated by the following example:

```
max :: Int -> Int -> Int
…
six = (max 4) 6
```

Expression `(max 4)` applies the function `max` only partially, resulting in a function with type annotation `Int -> Int`. This is then applied again for argument `6`, resulting in a constant integer.

2.6 Where

In Frege, it is possible to create local definitions with local scope inside a function definition. Such definitions may be placed inside `where` code block. The scoping rules prevent the functions created this way to pollute the working namespace, which is useful for creating reusable modules.

The following piece of code shows the implementation of a function which describes length of a given list using words:

```
describeListWhere xs = "The list is " ++ what xs
   where
      what [] = "empty."
      what [x] = "a singleton list."
      what ys = "a longer list."
…

frege> describeListWhere [1]
The list is a singleton list.
```

In the example above, in `where` block we define a new function called `what` which accepts a single list argument. It cannot be used outside the function `describeListWhere` where it is defined.

When working with `where` block, keeping a correct indentation is important. In the block, each local definition should be aligned with the `where` keyword, or have a greater indentation.

2.7 Guards

Guards are a mechanism in Frege which allow a user to return an expression based on boolean conditions.

Let us consider a signum function, which is used to describe a sign of a real number by an integer number. The function takes a single argument, which is a real number, and then performs a check of its sign:

- If the given number is greater than 0, the function returns 1.
- If the given number is smaller than 0, the function returns -1.
- If the given number is equal to 0, the function returns 0.

The following example is an implementation of the function in Frege:

```
sign x
   | x < 0 = (- 1)
   | x > 0 = 1
   | otherwise = 0
```

The lines starting with vertical line symbol `|` are called guards. They consist of a boolean expression and the resulting expression to return, if the corresponding condition is evaluated to `true`. The special `otherwise` keyword denotes a condition that always evaluates to `true`. Ordering of the guards is, therefore, important and the conditions are evaluated in the top-down manner.

2.8 Constant Definitions

In Frege, there is a special kind of the function definition where the user can define several constant functions in a single statement. The following example demonstrates definition of the three new constant functions `a`, `b` and `c`:

```
(a, b, c) = (1, 2, 3)
```

The values of the corresponding constants `a`, `b` and `c` are `1`, `2` and `3` in their respective order.

Frege is a relatively flexible language in this manner and the following expression is a valid one as well:

```
[2, f] = [2, 3]
```

In this case, the value of `f` is `3`. Furthermore, from the grammatical point of view, even the following statement is valid:

```
[1, f] = [2, 3]
```

A program with the example above would pass its compilation, however, an exception would be thrown upon an attempt to evaluate the constant `f` in any way. It still allows the user to specify constant functions in a flexible manner. The compiler only checks that the expression on the right hand side of the definition is of the same type as is the pattern on the left. For example, the following is the alternative definition of the constants `a`, `b` and `c` from the beginning of the current section:

```
(a, b, c) = (1, 1 + a, 1 + b)
```

2.9 Import and Export

To use functions, operators and data types from other modules, we must first import them. An import declaration has the following form:

```
import frege.prelude.Math (**, log)
```

The fully qualified name of the module being imported (`frege.prelude.Math`) has to be used. After that, the user may specify which items will be imported into the current namespace of the current module. For all other items the user has to use the qualified name of the item. The full details of the import declaration are described further in this section.

Import declarations are processed in the order they occur in the program text. However, their placement relative to other declarations is irrelevant. Nevertheless, it is considered a good practice to write all import declarations somewhere near the top of the program.

The user may or may not enumerate the items to import into the current namespace. If he or she chooses not to do so, everything from the given module is imported into the current namespace. The declaration in this case has the following form:

```
import frege.prelude.Math
```

The imported module `Math` contains several functions and constants for performing standard calculations. The constant pi may be referenced in this use-case directly in the following way:

```
circumference r = 2 * pi * r
```

In the example at the beginning of this section, there are two explicitly enumerated items: operator `**` and function `log`. Everything else has to be referenced using a qualified name. The constant pi in this scenario has to be prepended by the `Math` namespace:

```
circumference r = 2 * Math.pi * r
```

A user also has an option to enumerate none of the items, but to use the brackets symbols `()` anyway in the import declaration. This forces a programmer to use the qualified name for every item from the imported module.

When using the qualified name of the items, the imported module's name may be aliased by using `as` clause. The following example demonstrates the usage scenario:

```
import frege.prelude.Math as MM ()
…

circumference r = 2 * MM.pi * r
```

Last, but not least, there is an option to use a reverse manner of importing items into the namespace. Instead of enumerating items which we do not have to use the qualified name for, we may specify the only items, we will need to use the qualified names for. This is done by using `hiding` clause, as demonstrated by the following example:

```
import frege.prelude.Math hiding (pi)
…

circumference r = 2 * Math.pi * r
rightTriangleC a b = sqrt (a ** 2 + b ** 2)
```

Among the items that can be imported into the module, constructors are imported into the current namespace independently from the data types where they are defined. Frege additionally supports a special syntax to treat the import of data type constructors and ease their enumeration. Consider, for example, the following data type declaration from Section 2.3.3:

```
data Days = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday |
Sunday
```

The data type `Days` can be imported together with its constructors `Monday` and `Tuesday` into the current namespace using the following syntax:

```
import frege.example.DaysExample as DE (Days(Monday, Tuesday))
```

However, all of the remaining constructors belonging to the `Days` data type need to be referenced using the qualified names (e.g. `DE.Thursday`). To import all of data type's `Days` constructors, a syntax with two dot symbols `..` can be used, as described below:

```
import frege.example.DaysExample as DE (Days(..))
```

Regarding the importing mechanism, all modules by default import the built-in module Frege-Prelude. It contains standard arithmetic operators, such as `+`, `-`, `*` and `/`, comparison operators, many data types, such as `Maybe` and a lot of other definitions [8].

A similar approach to importing modules is also used to export items of the current module. Not exported definitions may not be referenced at all.

To export all of the definitions of the current module, we can use:

```
module Hello where
```

`Hello` denotes the name of the current module. The declaration has to be put at the beginning of the module.

To export only the functions `greeting` and `getTop`, we can write the following piece of code:

```
module Hello (greeting, getTop) where
```

A reader should be aware that based on how the platform MPS works, it is, unfortunately, not possible to work with the Frege modules and libraries written in a plain text directly. These have to be rewritten in a specific MPS project to make them usable. In this work we offer only a limited support of the standard module Frege-Prelude.

2.10 Further Reading

Frege and Haskell include many other aspects that are not mentioned in this work. While the current version of Frege-IDE supports all of the features of the language used in our examples throughout the Chapter 2 (and much more), due to the scale and complexity, a lot was still not implemented. However, for an interested reader we recommend visiting the Frege language specification and Frege Goodness book[3].

[3] Available from the WWW [12/07/2018]: <https://dierk.gitbooks.io/fregegoodness/>

3. Frege in MPS

Frege has rather too many syntactic and semantic constructs for us to cover them all within the scope of this work. We have therefore focused our attention only on the most important features worth implementing, such as function definition and type annotation, operators and custom data types. Our goal was to create an IDE that has a user-friendly editor that emulates normal text editing and writing code in the way that most Frege and Haskell developers are used to. This is accompanied by the context help, sometimes referred to as a code completion, which allows for referencing already defined functions, operators, variables, etc. Last, but not least, we strived for a type checker, which would be able to find small mistakes in the code, such as calling a function with invalid arguments, or check types of expressions.

This chapter assumes knowledge of the reader at least on the level covered in Chapter 2, which describes Frege syntax and features of the language we focused on.

3.1 Supported Subset of Frege

In this work, we focused our attention only on the most important parts of the Frege language, which gained its popularity (or, rather, popularity of Haskell). For the most part, so called 'syntactic sugars' are omitted, as well as monads, which make Frege appear less of a functional and more of an imperative programming language. To also include more advanced features, such as context help and type checking, we had to keep the complexity of the work within reasonable limits, and thus concepts, like type classes and type instances, were omitted as well.

In this section, we review the features of the Frege language that we have implemented from the high-level point of view.

Program structure. A program in Frege has to be properly structured. We expect the Frege module to have a header, depicting its name. We have also tried to emulate recommended practices of writing programs in Frege by, for instance, forcing import declarations to be written at the top of the program. Since a program is then just a series of definitions, we have left the rest up to the user of the IDE.

Import and export. To demonstrate the capabilities of the MPS platform regarding the scoping and code completion area, we decided to implement importing and exporting features of Frege, as they are depicted in Section 2.9, i.e. allowing users to reference functions, operators, custom algebraic data types and their constructors. The imported module may or may not be aliased and the corresponding import declarations occupy the top of the program's code layout.

Comments. Comments should be easily applicable where all of the normal definitions are expected. There are two types of supported comments:

- A single-line comment beginning with two dashes symbols `--`
- A multi-line comment that can be nested, which is surrounded with braces with dashes `{- … -}`

Function definition. The user must be able to define new functions. A function may accept any number of arguments, it consists of the patterns (see Section 2.2 for more details about the pattern matching) and the right-hand side, which provides the implementation for each function pattern.

Type annotation. We allow the user to specify the type of a function. This also plays a role during the type checking and evaluation, where we have an easier job to infer the types of the given functions and their arguments.

Operator definition. Infix operators are functions that strictly accept (at least) two arguments. The operator definition should be able to populate the namespace with new operators, allow these new operators to be used in new expressions and allow the operators to be annotated in the same way the regular functions can be.

Operator precedence and associativity. The statements beginning with the `infix` keyword (`infixl`, `infixr`, `infix`) specify and alter the specified operator's precedence together with its associativity. These have a significant impact on the type checking of expressions consisting of the infix operators.

Custom algebraic data types. The user should be able to declare new algebraic data types, which can be later used in functions. The name of the new data type becomes a new type, whereas the constructors become new values of that type.

Type synonyms. Similar to the data types, these statements introduce a new type inside the module, where they are defined. Unlike the data types, type synonyms only wrap a more complex type and do not introduce any new values.

Standard types and literals. `Bool`, `Char`, `Int`, and `Double` are types that are all part of the standard Frege library. Even though the standard library defines several additional types, such as `Float`, `Decimal`, and others, these were selected due to their prevalence and representative status and are therefore included in Frege-IDE. Additionally, type `String` is also supported with the standard syntax of using quotation marks (e.g. `"Hello, world!"`).

Tuples. Standard tuples are fully supported in Frege-IDE, including their syntax and type checking.

Lists. In Frege there are three main ways of defining a list in an expression:

- **Enumeration:** e.g. `['a', 'b', 'c']`
- **Range:** e.g. `['a' .. 'z']`
- **List comprehension:** e.g. `[x | x <- ['a' .. 'z']]`

We offer only a limited support in Frege-IDE for the list comprehension due to its complexity. Additionally, we have to account for the usage of the constructor operator for deconstructing lists denoted by colon symbol :. The operator can be also used for attaching a single list element at the beginning of another list. The following example demonstrates the usage of the operator in an expression: `ff = [1] : [[2, 2], [3, 3, 3], [4, 4, 4, 4]]`

Function type. The function type is another standard type which covers the type of the function. For instance, a function accepting two integer arguments and outputting a string has the following type:

```
Int -> Int -> String
```

The type must be declarable inside the type annotations.

Where. `where` clause allows to provide additional local definitions with local scope inside a function definition. These are then visible only to the closest outer definition, as demonstrated by the following example:

```
five = 1 + four
  where
    four = 1 + three
      where
        three = 3
```

In the example above, the constant function `three` is not visible in the right-hand side of the function `five`.

Similarly to the export and import feature of the Frege language, we aim to demonstrate the scoping capabilities of our IDE.

Guards. Guards provide an alternative way of the function definition with respect to the standard 'assignment definition' (e.g. `f = "value"`), where each guard contains a boolean condition for whether its branch should execute. This is an analogy to a series of if-else statements.

Additional concepts. From the Frege language we should also include the `if` statement, `case` expression, `let` statement and definition of lambda functions (anonymous functions defined within another definition). These together with the above should cover most of the standard usage of the Frege language, excluding classes and instances.

Type checking and evaluation. A separate aspect of the IDE is a type checker. We have implemented a simple type checking capability into our IDE, which is able to infer types of certain expressions and compare types of function definitions to their type annotations.

Providing a user with a complete type-checking capabilities is not feasible within the scope of this work, and thus we only intended to implement a rather restricted type evaluation, capable of handling only certain scenarios. However, the system should be easily extensible and robust enough to demonstrate the potential of the MPS platform.

3.2 Structure Aspect

Defining the structure aspect in MPS for each language concept is one of the most important part of this work. The concepts are ‘building bricks’ when it comes to working with AST. Every other aspect of Frege-IDE depends on this part, therefore a careful analysis is required to be done here.

Working with structure aspect in MPS to a certain extent resembles defining a grammar of the language for the compiler parser. To implement it correctly, we should understand the Frege grammar and how its different parts relate to the actual features of the language.

In this section, we are going to describe certain parts of the Frege grammar, show what actual features they correspond to and how we transformed them into the MPS concepts. A complete analysis would far exceed the scope of this text, so we will focus only on the most important or otherwise interesting parts we had to deal with.

The section focuses mainly on Frege grammar rules and how they are reflected on structure aspect of the IDE. Nonterminal symbols usually correlate to individual concepts in MPS, but it is not the one-to-one relationship. On the other hand, terminal symbols only rarely need to be represented by individual MPS concepts and are mostly only a part of the editor aspect. This section takes a look at certain Frege grammar rules, describes their high-level meaning and shows implementation of the related concepts from their structure aspect in MPS.

During the analysis, we used materials from Frege official website, namely Frege grammar used in the official Frege compiler and the language reference. The resources are also provided in the attachments.

The grammar uses mainly the EBNF (extended Backus-Naur form) notation, which we often refer to in this work. Explanation of the notation is provided below.

3.2.1 Notation Description

Throughout the Section 3.2, we often use EBNF notation to describe the grammar of certain features of the Frege language. In this section, we briefly describe the notation and explain certain parts.

Grammar of most programming languages is described with rules by which can one replace a nonterminal symbol for a sequence of terminal and nonterminal symbols. For instance, a program in many languages consists of series of statements separated by a semicolon. A rule describing this is provided below:

```
statements ::= statement ';' statements | statement
```

The example above states, that the nonterminal symbol `statements` may be replaced for any number of nonterminal symbols `statement` separated by the terminal symbol semicolon `;`, but at least one `statement` has to be present.

We use a convention that terminal symbols are surrounded with apostrophes, such as the semicolon symbol `';'` in the example above. If there are several options on how to replace a nonterminal symbol, the options are separated with the vertical line symbol `|`. Additionally, we use these regular expression symbols to express certain rules:

- `token*`: symbol `token` repeats in a sequence any number of times
- `token+`: symbol `token` repeats in a sequence at least once
- `token?`: symbol `token` may or may not be present
- `( … )`: symbols inside the brackets are treated as one

It should be noted that in many cases we try to simplify the grammar and omit the irrelevant parts not included in the final project. This is done for clarity and better readability of the corresponding grammar rules. If a major part is omitted, it is mentioned explicitly.

3.2.2 Program Structure

We start defining MPS concepts from the root level. A root concept represents a single document in MPS - a module in Frege.

A high level view of the module definition is described by the following rule:

```
module ::= moduleclause (';' definitions | 'where' '{' definitions '}')
```

In Frege code, the places where a proper indentation is required may be replaced by usage of curly brackets `{` and `}` and a semicolon `;`. Consider the following example:

```
describeListWhere xs = "The list is " ++ what xs
   where
      what [] = "empty."
      what [x] = "a singleton list."
      what ys = "a longer list."
```

The example may be rewritten in the following way:

```
describeListWhere xs = "The list is " ++ what xs where { what [] =
"empty."; what [x] = "a singleton list."; what ys = "a longer list." }
```

In most of the compiler implementations, the parser works with the second variant, while most of the Haskell and Frege programmers use the style with indentation, which we have actually stuck to in this work. The process of converting the first variant to the second is normally done during the lexical analysis.

Knowing this, we will demonstrate the `module` grammar rule on the example from Section 2.1. After we define the module and its qualified name, we may or may not use the keyword `where`, which only visually denotes the separation from the rest of the program. After that the module is just a series of definitions:

1. Part: `moduleclause 'where'`

(We decided to use the variant with the `where` keyword in Frege-IDE. `moduleclause` symbol is usually translated to string `module` followed by the user-entered qualified name, and so on.)

```
module Hello where
```

2. Part: `'{' definitions '}'`

```
definitions ::= definition (';' definition)* ';'?
```

(The rule above states that `definitions` are a series of `definition` symbols separated by the colon symbol, which is translated from separation of the definitions by creating new lines during the lexical analysis. There has to be at least one definition in a module.)

```
greeting friend = "Hello, " ++ friend ++ "!"

main args = do
    println (greeting "World")
```

The implementation of the concept representing the single module in Frege is described in the later subsections, as it is important to look on the symbol `definition` first.

3.2.3 Definitions

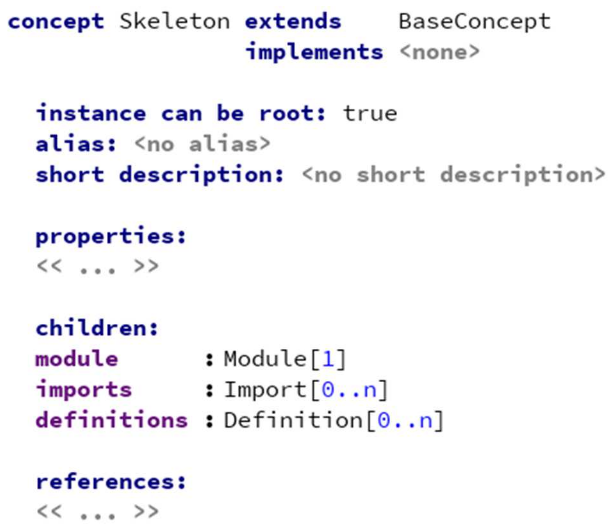
A definition in the Frege grammar is a substitute for one of the following language concepts:

- **Import declaration:** states, what is to be imported into the current module.
- **Fixity:** specifies associativity and precedence of an infix operator.
- **Type declaration:** declares a new type synonym.
- **Data declaration:** allows to create a new custom data type.
- **Class declaration:** omitted in this work.
- **Instance declaration:** omitted in this work.
- **Local definition:** covers function definitions and type annotations.

We have mentioned that it is a good practice to include the import statements in the top of a module definition before any other statements. We can enforce this by not following the exact Frege grammar, but rather implement our own version of the `module` grammar rule. It will have to be a root concept responsible for the overall program structure with the following children:

- `module`: corresponds to the `moduleclause` symbol from the previous section. Represents a name of the module together with additional specifications that usually go onto the first line of a Frege program.
- `import`: corresponds to the import statements, which are part of the `definition` rule. They have to be separated from the rest of the definitions to enforce their placement at the top of the program. Import statements are described in more detail in the following section.
- `definitions`: represents the rest of the statements that are part of the `definition` rule.

A possible implementation of the concept from the structure aspect view is depicted on Figure 3.1. The corresponding root concept is named `Skeleton`.



```
concept Skeleton extends    BaseConcept
                 implements <none>

  instance can be root: true
  alias: <no alias>
  short description: <no short description>

  properties:
  << ... >>

  children:
  module       : Module[1]
  imports      : Import[0..n]
  definitions  : Definition[0..n]

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

Figure 3.1: Structure aspect for the `Skeleton` concept in Frege-IDE

3.2.4 Import Statements

According to the Frege reference, the following rules are associated with the import statements:

```
import ::= 'import' packagename ('as'? namespace)? 'public'? importlist?
```

```
importlist ::= 'hiding'? '(' (importitem (', ' importitem)*)? ')'
```

The `import` rule represents the import statement as it was described in Section 2.9. For instance, the relations between the different parts of the rule are described on the following example:

```
import mps.frege.ExampleTree as ET (Tree(Nil), ->>, traverse)
…

import                              ~         'import'
mps.frege.ExampleTree               ~         packagename
as ET                               ~         ('as'? namespace)?
(Tree(Nil), ->>, traverse)          ~         importlist?
```

Unlike in the official Frege compiler implementation, we did not include the visibility (`public` keyword) and aliasing of the imported items in the final work (however, aliasing of the imported module was included).

The `importitem` symbol represents a function, operator, type or a data type, which is to be imported into the current namespace. The rule from the official reference is as follows:

```
importitem ::= VARID
             | OPERATOR
             | CONID ('(' (member (',' member)*)? ')')?
```

The nonterminal symbols from the rule above are referenced in this work several times. The distinction is as follows:

- `VARID`: represents identifiers beginning with the lowercase symbol (a-z) or with the underscore symbol. It is used to describe names of functions, variables, type variables, etc.
- `CONID`: represents identifiers beginning with the uppercase symbol (A-Z). Describes names of types, data types, constructors, etc.
- `OPERATOR`: it is used to describe the sequence of symbols an infix operator can consist of. The available symbols are mentioned in Section 2.4.

The member enumeration described by the third variant in the `importitem` rule above also allows to enumerate constructors associated with a specific data type. Only the constructors enumerated are to be imported into the current namespace. An example of such an import declaration is as follows:

```
import frege.example.DaysExample as DE (Days(Monday, Tuesday))
```

The example above is from Section 2.9 and imports the data type `Days` into the current namespace together with its constructors `Monday` and `Tuesday`. Rest of the constructors from the data type need to be referenced using their qualified names.

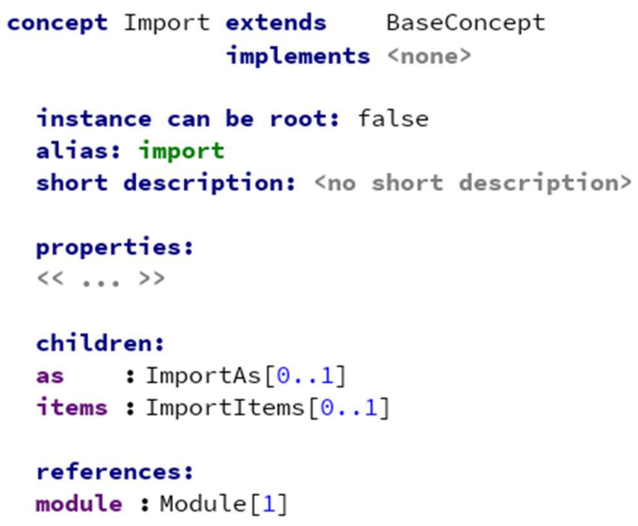
To implement the import statements in Frege-IDE, we can mostly follow the official grammar. We create the concept `Import` with the three main parts:

- Name, which references an existing module's name.

- Optional aliasing part of the import statement (clause `as`).
- Optional list of imported items into the current namespace.

An example of such implementation is captured in Figure 3.2. Note that terminal symbol `import` is not a part of the structure aspect of the corresponding `Import` concept, but rather of its editor. The terminal symbols usually do not affect the structure of the AST in any way and only affect its visual representation.

Additionally, we set the concept's alias to `import`. This tells MPS to create an instance of the concept whenever the user types the string `import` (the set alias) in the correct places of the code.



```
concept Import extends    BaseConcept
               implements <none>

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

  properties:
  << ... >>

  children:
  as    : ImportAs[0..1]
  items : ImportItems[0..1]

  references:
  module : Module[1]
```

Figure 3.2: Structure aspect of the `Import` concept in Frege-IDE

In Figure 3.2., the implementation of the `Import` concept consists of the children concepts `ImportAs` and `ImportItems`. The concept `ImportAs` wraps the `('as'? namespace)?` part of the `import` rule, whereas `ImportItems` relates to the `importlist` symbol.

The reference to `Module` concept correlates to the `packagename` symbol of the `import` rule. This tells MPS that we will allow only importing of such modules that exist and are visible to the current module in Frege-IDE (however, this means we will not be able to import modules and libraries from external sources made outside the Frege-IDE).

As we have mentioned, the concept `ImportItems` correlates to the `importlist` nonterminal symbol. In the Frege-IDE implementation, the concept simply contains an optional child of the concept `ImportHiding`, which represents the optional `hiding` keyword, and `[0..n]` children of the concept `ImportItem`.

`ImportItem` represents one of the three main types of identifiers. In MPS, however, we want to take the advantage of the references to allow the import only of the existing items from the corresponding module that is being imported. From the high-level point of view, we may import only one of the following types of items:

- Functions
- Operators

- Type synonyms
- Data types
- Data type constructors

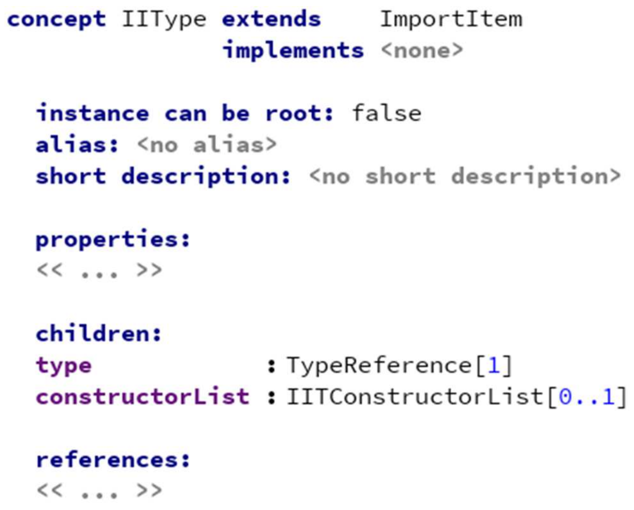
The idea is to represent the import item with an abstract concept and then to create concrete concepts for each type of the import items. The corresponding abstract concept in Frege-IDE is the already mentioned `ImportItem`. The concrete concepts are as follows:

- `IIFunction` for importing functions
- `IIOperator` for importing operators
- `IIType` for importing type synonyms and data types
- `IIConstructor` for importing constructors

Why is it enough to create only one concept as a substitute for importing both type synonyms and data types? The reason is the grammar - there is basically no syntactical difference between the two since they both define a type. Thus, where there is the former, there can also be the latter.

Each of the four concepts needs to contain a reference to the language construct it represents. Since the language constructs are referred from multiple places in the Frege grammar, the implementation is best solved by using smart references. As a reminder, the smart reference is a concept that contains only a single reference. In the case of the Frege language, the four mentioned language constructs have to be separated - for each one we need to create a new smart reference.

Let us take for example the import item `IIType`. It may represent a data type and for this reason, it must contain an optional member enumeration list we named `IITConstructorList`. The type name it references is enwrapped inside a new smart reference concept called `TypeReference`. The implementation of the `IIType` concept in the structure aspect is depicted on Figure 3.3.

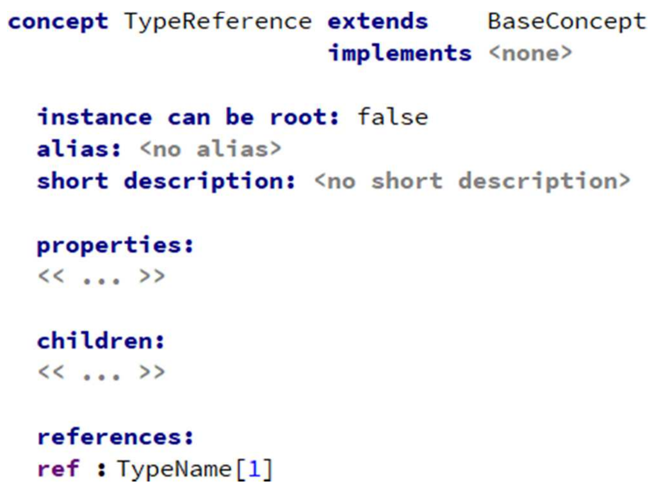


```
concept IIType extends    ImportItem
               implements <none>

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

  properties:
  << ... >>

  children:
  type            : TypeReference[1]
  constructorList : IITConstructorList[0..1]

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

Figure 3.3: Structure aspect of the concept `IIType` in Frege-IDE

The implementation of `IITConstructorList` concept is to a greater extent similar to the implementation of the `ImportItems` concept, since they both represent a sequence of members enclosed within brackets, hence we will not describe it.

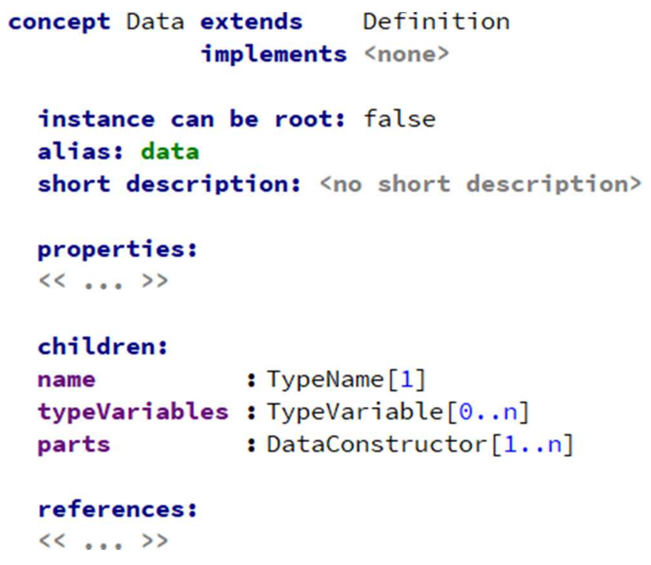
`TypeReference` concept is a smart reference to a type name. Both type synonym and data type declaration are named by the `CONID` variant of the identifier in Frege, which means their names need to begin with an uppercase letter. The two concepts representing the declarations then need to contain the same type of child to represent their name to be referenceable from the `TypeReference` concept. Figure 3.4 shows the structure aspect of the smart reference concept `TypeReference`.



```
concept TypeReference extends    BaseConcept
                      implements <none>

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

  properties:
  << ... >>

  children:
  << ... >>

  references:
  ref : TypeName[1]
```

Figure 3.4: Structure aspect of the concept `TypeReference` in Frege-IDE

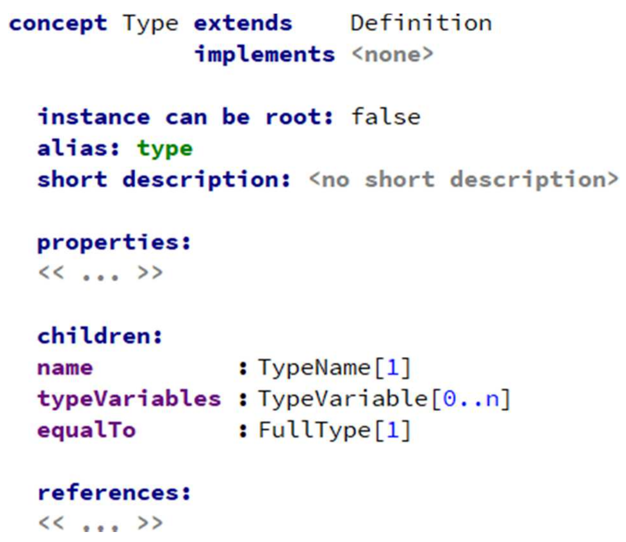
Concepts `Data` representing the data type declaration and concept `Type` representing the type synonym declaration are depicted from their structure aspect in Frege-IDE on Figures 3.5 and 3.6. Their name is represented by a child of the same concept `TypeName`, which is used in the smart reference `TypeReference`.



```
concept Data extends    Definition
             implements <none>

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

  properties:
  << ... >>

  children:
  name          : TypeName[1]
  typeVariables : TypeVariable[0..n]
  parts         : DataConstructor[1..n]

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

Figure 3.5: Structure aspect of the concept `Data` representing the data type declaration in Frege-IDE

We have not yet explained how to allow referencing only the import items actually corresponding to the Frege module being imported. The feature is more advanced and cannot be solved simply by defining the structure of possible ASTs. It is related to the scopes in constraints aspect in MPS and we describe it in Section 3.4.



```
concept Type extends    Definition
             implements <none>

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

  properties:
  << ... >>

  children:
  name          : TypeName[1]
  typeVariables : TypeVariable[0..n]
  equalTo       : FullType[1]

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

Figure 3.6: Structure aspect of the concept `Type` representing a type synonym declaration in Frege-IDE

3.2.5 Function Definition

Function definition covers the area of defining new functions, operators and certain special expressions for defining constants.

On its highest level, we could divide the function definition into two parts: *left* and *right hand side*. When the user is providing a definition, he or she always specifies a pattern and an expression corresponding to that pattern. To demonstrate, let us consider the following example:

```
getTop (x:xs) = x
```

Here, the single statement defines a new function `getTop`, which returns the first item of a non-empty list. In this case, the left hand side consists of the string `getTop (x:xs)`, which describes the name of the function and its arguments. The part `(x:xs)` is usually referred to as *pattern*. On the right hand side, we have the simple expression `x` referencing the argument from the pattern.

A slightly different example is the definition of the signum function as provided below:

```
sign x
   | x < 0 = (- 1)
   | x > 0 = 1
   | otherwise = 0
```

Similarly to the previous case, the string `sign x` is considered to be the left hand side of the function `sign`, whereas the rest of the definition is the right hand side. This corresponds to the actual Frege grammar, where the high-level function pattern binding is defined as follows:

```
binding ::= lhs rhs
```

```
rhs ::= '=' expression ('where' declarations)?
          | guardedExpressions ('where' declarations)?
```

(Symbol `lhs` stands for the 'left hand side' and `rhs` for the 'right hand side'.)

3.2.5.1 Left Hand Side

There are three main ways of defining a function:

- Standard definition, e.g. `multiply x y = x * y`
- Operator definition, e.g. `x :-: y = x + y + 1`
- Any pattern definition, e.g.: `(a, b, c) = (1, 2, 3)`

The grammar for the left hand side is associated with the following grammar rules:

```
lhs ::= VARID patternTerm*
      | patternTerm OPERATOR patternTerm
      | pattern
```

```
pattern ::= listPattern
          | patternTerm
```

```
listPattern ::= patternTerm ':' listPattern
              | patternTerm
```

```
patternTerm ::= VARID                                        (1)
              | '_'                                          (2)
              | literal                                      (3)
              | '[' (pattern (',' pattern)*)? ']'            (4)
              | '(' (pattern (',' pattern)*)? ')'            (5)
              | CONID pattern*                               (6)
```

The grammar rules above are simplified due to the Frege-IDE not supporting certain features, such as word-like operators (e.g. ``x `plus` y = x + y``) or argument capture, while other changes were made for clarity.

From the rules above it is clear that the `lhs` symbol corresponds to the three ways of the function definition mentioned earlier. `patternTerm` symbol allows us to do one of the following:

(1) **Define a new variable:** `factorial n = n * factorial (n - 1)`

(2) **Use the wildcard symbol _:** `charToName _ = "No Name"`

(3) **Use a literal:** `factorial 0 = 1`

(4) **Use a list:** `getTop [] = "No elements"`

(5) **Use a tuple or surround a single pattern inside the brackets:** `second (_, x, _) = x`

(6) **Apply a data type constructor:** `surface (Circle _ r) = pi * sqr r`

Additionally, a list may be matched also by creating a pattern by using the grammar rule for `listPattern` symbol. The example is as follows:

```
getTop (x:xs) = x
```

The part representing the list `(x:xs)` can be created by using the corresponding rule. Additionally, any number of items at the beginning of the list may be specified. The following example demonstrates the usage on a function, which returns a first item of a given list. The given list, however, must contain at least three items, otherwise it will not be matched:

```
getTop (item1 : item2 : item3 : tail) = item1
```

In the final work we made several simplifications to ease us the implementation of the typesystem and editor aspect. First, we require each AST node corresponding to the `listPattern` symbol to be enclosed within brackets. This bears no semantic restrictions and is an actually recommended practice, because it makes the code easier to read. Second, pattern consisting of empty brackets makes no sense in the restricted set of features we provide, therefore it was not included either.

3.2.5.2 Right Hand Side

The right hand side of the function definition can be either a single expression optionally followed by the `where` block, or a series of guards optionally followed by the `where` block. This can be seen on the grammar rule for the symbol `rhs`.

`where` block consists of the function definitions and type annotations. These are described later in Section 3.2.6.

Guards, in contrast to the single expression variant of the function definition, consist of at least two expressions and in their most simple form a grammar for them looks like this:

```
guardedExpressions ::= ('{' guard '}')+
```

```
guard ::= '|' expression '=' expression
```

3.2.5.3 Expressions

The `expression` symbol consists of a series of binary expressions (infix operators with operands) and an optional type, or the `forall` construct. The rule defining the expression is as follows:

```
expression ::= binex ('::' (forall | type))?
```

Even though the `forall` construct is linked to type declaration, it is a part of a more advanced feature in Frege and we have not included it. The `type` symbol, on the other hand, is described later in Section 3.2.6.

`binex` symbol represents a series of operands separated by operators. In Frege-IDE, the concept representing the operands is named `TopExpression`, which corresponds to the symbol `topex` from the official Frege grammar. A single operand may take one of the following forms:

- **Case expression**
- **Let expression**
- **Conditional expression** (if)
- **Lambda**
- **A series of primary expressions separated by a whitespace**

Primary expressions are a subset of expressions that include terms, monads (omitted in this work) and a usage of (qualified) names, for instance application of imported functions or operators [9]. In Frege grammar, terms are literals, lists and tuples, or in other words, values. Additionally, bracketed expression is also a part of the primary expressions.

The last point of the `topex` grammar rule also includes a usage only of a single primary expression. A single primary expression can act as an operand:

```
six = 3 + 3
```

However, for a series of primary expressions separated by a whitespace to be meaningful, it must be some kind of an application - an application of a function, operator, or the data type constructor. Let us consider the following function:

```
six = max 4 (3 + 3)
```

In the example above, `max` is part of the primary expressions. It is a name of a function that is visible in the current namespace. The number `4` is a literal, which is part of the terms. `(3 + 3)` is a bracketed expression. Together they form the application of the function `max` with the arguments `4` and `(3 + 3)`.

A special case of the application is the application of the bracketed expression. In the following example, we enclose the expression `max 4` with brackets, which is the currying technique described in Section 2.5. What remains is the function accepting a single argument, to which we provide the literal `6`:

```
six = (max 4) 6
```

Not every sequence of primary expressions is an application. For instance, a sequence of literals makes no sense when standing alone:

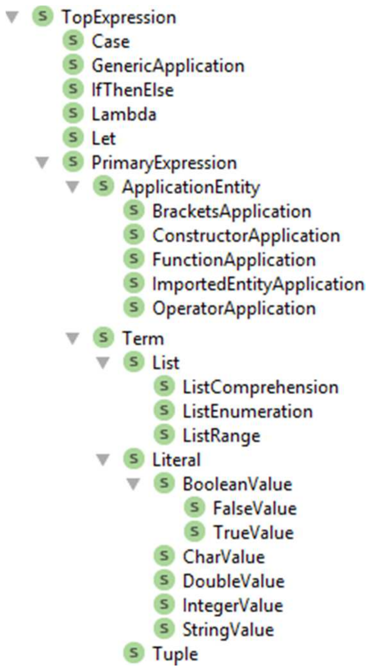
```
invalid = 1 2 3 4 5 6 7
```

We have modified the grammar in Frege-IDE to better distinguish between the different forms of the applications and to allow only the valid applications to occur in the program. As such, the expression from the example above is impossible to type in Frege-IDE. This additionally helped us in the typesystem aspect, where the corresponding type-checking algorithms were easier to implement.

Figure 3.7 shows a hierarchy of concepts in Frege-IDE, starting from the `TopExpression` concept in the corresponding 'inheritance tree'. As we have already described, the `TopExpression` may take several different forms, such as the case expression (concept `Case`) or the conditional expression (concept `IfThenElse`). There is no direct variant corresponding to the series of primary expressions part of the `topex` grammar rule. Instead we introduce two custom concepts, `ApplicationEntity` and `GenericApplication`.

`GenericApplication` represents any kind of application, for instance, application of a function, operator, data type constructor and so on. From a certain point of view, the concept does correspond to the series of primary expressions, but only partially. For cases when we have a single literal in an expression, an instance of the `Literal` concept is used in the corresponding AST instead.

`ApplicationEntity` represents the entity we are applying in the `GenericApplication` concept. It may be a function, operator, but also a bracketed expression.



- TopExpression
  - Case
  - GenericApplication
  - IfThenElse
  - Lambda
  - Let
  - PrimaryExpression
    - ApplicationEntity
      - BracketsApplication
      - ConstructorApplication
      - FunctionApplication
      - ImportedEntityApplication
      - OperatorApplication
    - Term
      - List
        - ListComprehension
        - ListEnumeration
        - ListRange
      - Literal
        - BooleanValue
          - FalseValue
          - TrueValue
        - CharValue
        - DoubleValue
        - IntegerValue
        - StringValue
      - Tuple

Figure 3.7: Hierarchy of concepts starting from the `TopExpression` concept in Frege-IDE

In Section 2.4, we have already mentioned that the applications have higher precedence, than any operator. Now, as we have described the grammar of the expressions in Frege, it is clear they are syntactically bound to act that way.

3.2.5.4 Grouped Representation

A problem with the pattern matching in Frege is that it may seem like we are providing several definitions for the same function. Consider the example:

```
getTop [] = "No elements"
getTop (x:xs) = x
```

In the example above, for different versions of input arguments we are specifying different bodies of the same function. However, in Frege-IDE, we need to reference the existing function name from multiple places, such as import declarations, or the function application. We need to reference an existing node in the corresponding AST representing the current program. If we were to reference the `getTop` function, for instance, from a different function, which one of the two AST nodes would we have to point to? Understandably, the answer should be 'all of them', as they all define the same function. This can be achieved in MPS by creating a new concept, which wraps all of the function definitions that define the same entity.

The 'grouping' concept should contain a single child, which represents the function's name, and `[1..n]` children representing different variants of the function's implementation for its different patterns. The patterns also need to be adjusted so they do not introduce their custom function names, but rather reference the single child of the 'grouping' concept representing the function name. In Frege-IDE, the concept representing the grouped definition is called `FDGrouped`.

Additionally, the grouped representation has to be created also for the operator definitions. However, in the third case of the grammar rule for the `lhs` symbol, there is no grouping necessary – only constant functions can be defined this way and providing different pattern matchers for the constant functions makes little to no sense.

There was also a different way of dealing with the problem, which involved the type annotation to be the AST node which is referenced, instead of the grouped function definition. While this is definitely also an option, in Frege we can have a function without the type annotation whereas the other way around we would get an error during the compilation. We believe the approach we took and described in this section is more appropriate and actually conforms to the Frege specification, which would not be the case in the latter way.

3.2.6 Types

Types are a vital part of the Frege grammar. They play a role in the definition of the type annotation, or when declaring custom data types.

There are several 'native' types, which translate directly to the types from the JVM. These include:

- `Bool`
- `Char`

- `String`
- **Numeric types** (in this work, only `Int` and `Double` are supported)

These types are part of the Frege-Prelude library and are implicitly imported to all Frege modules.

Additional types we included are:

- `Tuple`
- `List`
- Custom algebraic data types
- Function types

Types connected to the monads, exceptions, and the data types that are part of the Frege-Prelude library were not included in this work.

Before we describe the Frege types and how we implemented them in Frege-IDE, it is important to note that by omitting classes and instances, we lose an important aspect of the Frege language, which is called *parametric polymorphism*. This feature allows to apply certain functions and operators on a selected class of types, but not all of them. Consider, for instance, a built-in operator `+`. It allows to add any numeric values together. Its type annotation would look like this:

```
(+) :: a -> a -> a
```

The problem is to specify that the type variable `a` represents a numeric type. This is, unfortunately, not possible without Frege classes and instances. We omitted the feature due to the size and scope of the work and leave it as a possible future extension of Frege-IDE.

Without the type classes, the grammar describing the Frege types is as follows:

```
type ::= (typeApplication '->')* typeApplication
```

```
typeApplication ::= simpleType+
```

```
simpleType ::= VARID                                   (1)
             | CONID                                   (2)
             | '(' type ')'                            (3)
             | '(' type (',' type)+ ')'                (4)
             | '[' type ']'                            (5)
```

On the highest level, we specify the function type, which corresponds to the symbol `type`. Each part of the function type, representing either an argument or the return type, consists of one or more `simpleType` symbols, which is denoted by the `typeApplication` grammar rule. As the name suggests, this rule is responsible for the application of the type on a type variable. Consider, for example, the data type `Maybe`, which consists of a single type variable. A type of the function `getTopIntList` from Section 2.3.3 looks as follows:

```
data Maybe a = Just a | Nothing
…

getTopIntList :: [Int] -> Maybe Int
```

In the example above, the part `Maybe Int` is made possible by the application of the `typeApplication` grammar rule.

The `simpleType` symbol denotes one of the following options:

(1) **Usage of a type variable:** `(+) :: a -> a -> a`

(2) **Usage of a type name:** data type names, together with the built-in types, such as `Int`, or `String`, are part of the grammar rule. `getTopIntList :: [Int] -> Maybe Int`

(3) **Type inside brackets:** semantically the type of the expression is the same, as the type inside the brackets. This allows, for instance, to specify a function type argument of a function, as depicted in Section 2.3.5: `map :: (Int -> Int) -> [Int] -> [Int]`

(4) **Tuple type:** `second :: (a, b, c) -> b`

(5) **List type:** `getTop :: [a] -> a`

Types are used in several places throughout the Frege grammar, but we will mention just two main representatives: type annotation and declaration of new data types.

In type annotation, we provide a name of the function or an operator we want to specify the type for. It is even possible to specify the type for several items at once, as demonstrated by the following example:

```
(+), (-), (*) :: Double -> Double -> Double
```

The right hand side of the type annotation `Double -> Double -> Double` corresponds to the grammar described in this section. It is important here to note the possibility to declare new type variables:

```
map :: (a -> b) -> [a] -> [b]
```

This is an important distinction from the data type declaration, where on the right hand side, only the type variables declared on the left hand side can be used. Consider the following example:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

The left hand side of the data type declaration declares a new type variable `a`. This variable then can be used on the right hand side, where constructors `Nil` and `Node` are defined.

We want to incorporate the mentioned behavior by using references. On the right hand side of the data type declaration, an instance of the concept containing a reference to an existing type variable has to be used. In the type annotation's right hand side, an instance of the concept containing a property has to be used instead. The property, in this case, represents the new type variable's name.

To implement the types for data type declarations and type annotations, we can take one of these two main approaches:

- We can create two sets of concepts for types, each with a different implementation of type variable usage. In the first set, only the declaration of the new type variables would be possible. In the second, type variables could be only referenced.
- We can implement two different concepts that inherit from the concept corresponding to the `simpleType` symbol: one for creating type variables and one for referencing them. Using the constraints aspect in MPS, we are able to allow or restrict the concepts instances in specific places of the AST.

We have chosen the second approach, because it is less time-consuming to implement and poses fewer problems. First, it is syntactically more appropriate as the two different sets of concepts would mean there is a different grammar for the types in the data type declaration and in the type annotation. This is not true as the type variables are simply checked during semantic analysis. Additionally, this solution is easier to maintain if certain changes are to be incorporated.

The remaining part of the types are the built-in native types. In this case, there are also two main implementation approaches:

- We can 'hard-code' the native types into the Frege-IDE, creating a concept inheriting from `simpleType` for each of the native types mentioned at the beginning of this section.
- We can define the types using the data type declaration statements inside a new module. This module will be implicitly imported to all of the other modules a user will create, as in the case of Frege-Prelude library.

While the second option may seem easier to maintain, there are actual benefits to the first approach, which have to do with the typesystem aspect of MPS. Specifically, it would be especially difficult to link the type names to the types of the corresponding literals in case of the latter option. Hence, we opted for the first approach.

3.3 Editor Aspect

The process of creating the editor for Frege-IDE involves defining visual appearance for each AST node and providing the user with a way to manipulate the AST in a user-friendly manner.

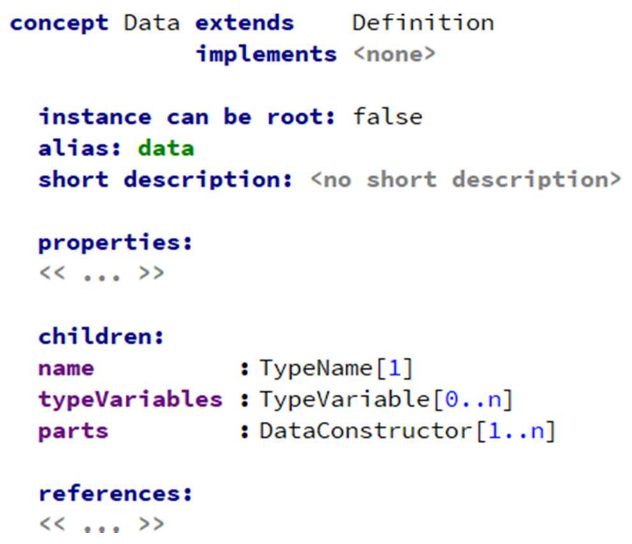
As mentioned before, MPS is a projectional editor and thus it is not possible for the user to work with the code in the text form directly. Every function, name, variable, type etc. is in some way associated with a specific AST node which has to be presented to the user in some way.

In this section, we show how the editor aspect in MPS allows us to tackle many different challenges we came across when implementing the user interface part of our IDE. We also present a couple of non-trivial problems we had to deal with, and what approach we took to solve them.

3.3.1 Visual Appearance of AST Nodes

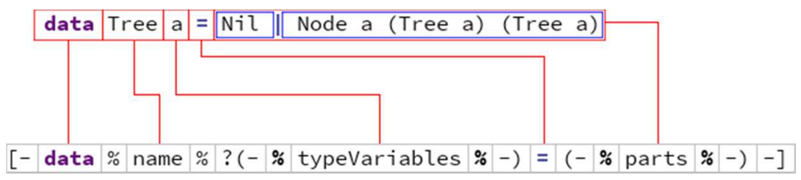
Appearance of the most of the concepts is rather straightforward to implement. We have already analyzed and implemented the structure aspect in the previous section, so it is clearly visible which concept is connected to what language feature. Frege is the text-based programming language, thus creating concept editors involves only specifying the correct set of strings for constant editor cells and the right set of other types of cells mentioned in Section 1.3.

We will demonstrate the implementation of the concept editors on the example for the concept associated with the data type declarations. The corresponding concept `Data` inherits from the `Definition` concept, which was introduced in Section 3.2.3. The structure aspect of the `Data` concept is depicted on Figures 3.5 and 3.8. Its concept editor is depicted on Figure 3.9.



```
concept Data extends    Definition
             implements <none>

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

  properties:
  << ... >>

  children:
  name          : TypeName[1]
  typeVariables : TypeVariable[0..n]
  parts         : DataConstructor[1..n]

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

Figure 3.8: Structure aspect of the `Data` concept in Frege-IDE



```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

```
[- data % name % ?(- % typeVariables % -) = (- % parts % -) -]
```

Figure 3.9: Visual representation of the AST for the data type declaration (top) together with the `Data` concept editor (bottom)

In Frege, the data type declaration has to begin with the keyword `data`. In our editor, it is a constant editor cell with the string `data`. We have additionally applied editor styles to denote the keyword by changing its color and making the text bold. The keyword is then followed by the data type's name. As mentioned in the previous section, the name must not be a property, but rather a child node due to the implementation of the reference concept `TypeReference`. In the editor, we simply use the child editor cell pointing to the concept editor of the `TypeName` concept. Type variable declarations and the set of constructors are both lists of child nodes. For these we had to use the (horizontal) child collection editor cell. They again, as in the case of the data type's name, refer to their corresponding concept editors, which are used when editing their values. The delimiter for the set of constructors is set to the vertical bar symbol `|`, which is added between the child editor cells automatically.

The editor implementation for the concept representing the data type constructor is depicted on Figure 3.10. The data type constructor concept `DataConstructor` consists of the name and an arbitrary amount of type nodes to depict the type of values the constructor accepts. On the example for the `Tree` data type, the constructor `Node` accepts the first argument of type `a` set by the corresponding type variable `a` from the left hand side of the data type declaration. The subsequent two arguments are of types `Tree a`, therefore recursively pointing to the type `Tree` with the same type variable `a`. The editor therefore consists of the child editor cells for the constructor's name and for the collection of type nodes.

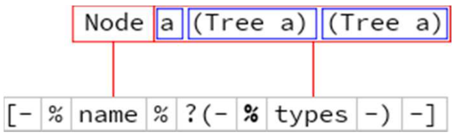


Figure 3.10: Visual representation of the AST for the data type constructor (top) together with the `DataConstructor` concept editor (bottom)

3.3.2 Side Transformation Menus

Side transformations allow the language designer to change the concrete AST in a specific manner when the user of that language writes a certain text either right or left of the given editor cell.

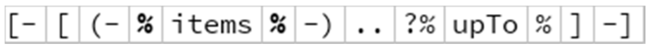
Let us take for example the list expression in the right hand side of the function definition. We have mentioned there are three main ways to define the list:

- **By enumerating elements:** `['a', 'b', 'c']`
- **By specifying a range of elements:** `['a', 'c' .. 'z']`
- **Using the list comprehension:** `[x * y | x <- [1..5], y <- [3..7]]`

For each of the three methods there is a specific concept associated with the list definition. Now we want to enable a seamless transformation from the first to the second method when the user enters `..` right of the last element in the list. To do that, we need to know how both of the concepts are implemented.

Enumeration list concept is a simple term that contains arbitrary amount of children of the `Expression` concept. Its editor consists of two constant editor cells representing square brackets `[` and `]`, and the horizontal collection cell representing the children items of the list.

The range list concept is based on the enumeration list, but it has to have at least one child left of the `..` symbol. Additionally it may or may not contain a child of the `Expression` concept for specifying the upper bound of the range. Figure 3.11 represents the editor implementation for the range list concept. The upper bound child cell `upTo` is preceded by the question mark symbol to denote that the editor cell is to be displayed only if the upper bound is actually specified.

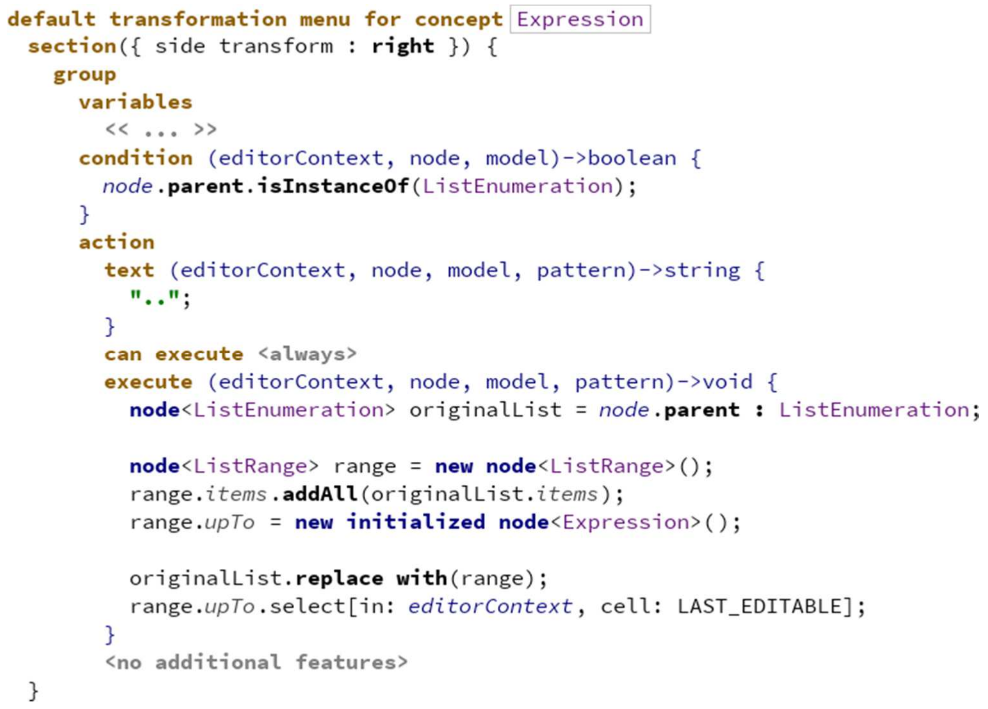


```
[- [ (- % items % -) .. ?% upTo % ] -]
```

Figure 3.11: Editor for the concept representing the range list expression

The implementation of the transformation is rather straightforward and follows a similar pattern as in the example described in Section 1.3.2. We set the default transformation menu for the concept associated with the list item (`Expression` concept), as the only editor cells that can trigger an action are constant, property and referent cells (list item is the child editor cell). We set the action's properties as follows:

- `Text` triggering the transformation should be set to `..`
- The `Can execute` part should be left to `<always>` to prevent the other text patterns from triggering the current transformation menu.
- In the `Execute` part we specify that the new instance of the range list concept should copy all of the items from the former instance of the enumeration list concept. Then it should replace the former instance completely. The items that are copied are the ones that will go before the `..` symbol. In most cases the user actually wants to specify the upper bound of the range list as well, which is why we then create a new instance of the `Expression` concept and set the focus on the new node.



```
default transformation menu for concept Expression
  section({ side transform : right }) {
    group
      variables
        << ... >>
      condition (editorContext, node, model)->boolean {
        node.parent.isInstanceOf(ListEnumeration);
      }
      action
        text (editorContext, node, model, pattern)->string {
          "..";
        }
        can execute <always>
        execute (editorContext, node, model, pattern)->void {
          node<ListEnumeration> originalList = node.parent : ListEnumeration;

          node<ListRange> range = new node<ListRange>();
          range.items.addAll(originalList.items);
          range.upTo = new initialized node<Expression>();

          originalList.replace with(range);
          range.upTo.select[in: editorContext, cell: LAST_EDITABLE];
        }
        <no additional features>
  }
```

Figure 3.12: Default transformation menu for the `Expression` concept

The problem with the current approach is that the concept that triggers the transformation is `Expression`, which is a child of several different concepts, not all of which are list enumeration concepts. We need to include an additional condition where we check that the current `Expression` node is indeed a child of the list enumeration concept and only then allow the action to be executed. Figure 3.12 shows the implementation of the default transformation menu for the `Expression` concept. The described action is enwrapped inside the action group which checks whether the parent of the current `Expression` node is an instance of the list enumeration concept. In the `Execute` part, we replace the parent AST node, which is the actual list enumeration, for the new instance of the range list concept.

3.3.3 Transformation Menu Inclusion Pattern

As described in the previous section, the child editor cells cannot trigger editor actions and therefore the actions are defined on the corresponding child concepts instead. However, certain concepts, such as the `Expression`, are children of several different concepts. The individual actions have to be restricted so that they are applicable only for the relevant parents.

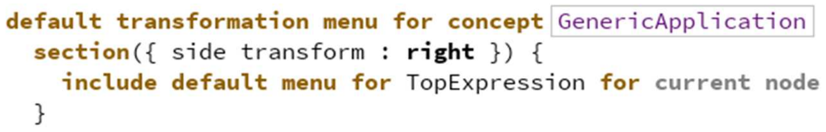
There is, however, another problem with the editor related to the parent-child relationship. In the previous section we described how to transform the list enumeration to the list range upon entering the `..` symbol right of the `Expression` concept. For the sake of simplicity, let us assume the transformation menu was actually defined for the `TopExpression` concept. We want the transformation to also work for the concept `GenericApplication` which is associated with the function application and inherits from the `TopExpression` concept. Consider the following example:

```
ff = [1, 2, getTop [3 .. 100]]
```

The expression `getTop [3 .. 100]` is a text representation of an instance of the `GenericApplication` concept. Upon entering the `..` symbol right of the `[1 .. 100]` expression, the user might expect the list would be transformed to the list range, as demonstrated by the following example:

```
ff = [1, 2, getTop [3 .. 100] .. 7]
```

Unfortunately, this is not the implicit behavior. First, the `GenericApplication` has to have its own default transformation menu defined where it includes the transformation menu for the `TopExpression` concept. The implementation is illustrated on Figure 3.13.



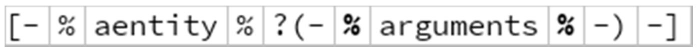
```
default transformation menu for concept GenericApplication
  section({ side transform : right }) {
    include default menu for TopExpression for current node
  }
```

Figure 3.13: Default transformation menu for the `GenericApplication` concept

The `GenericApplication` concept contains the following children:

- A single child of the ApplicationEntity concept, which specifies what is to be applied (e.g. a function, operator or a bracketed expression).
- Arbitrary amount of children of the PrimaryExpression concept, which represent the arguments passed to the application entity.

The GenericApplication editor consists solely of the child editor cells, as depicted in Figure 3.14. As explained in the Section 1.3.2, the transformation therefore cannot be triggered by the GenericApplication, since its editor does not contain the necessary type of editor cells. Instead the transformation menu has to be defined also for all of its children ApplicationEntity and PrimaryExpression concepts.



```
[- % aentity % ?(- % arguments % -) -]
```

Figure 3.14: Concept editor for the GenericApplication concept

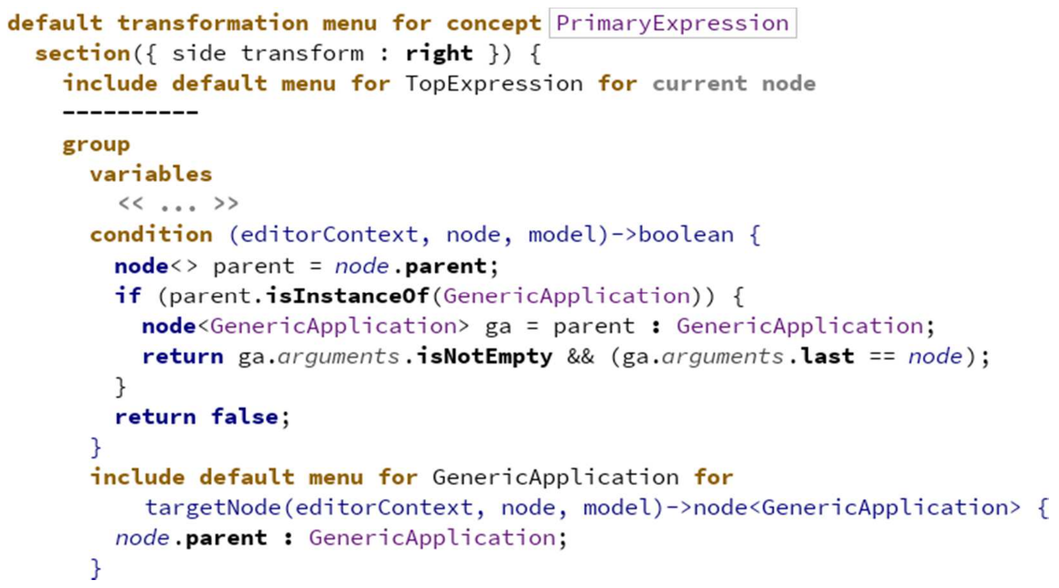
The default transformation menu for the GenericApplication concept should be triggered only if the specific text pattern is written right of the whole node. An instance of the GenericApplication concept may contain none, single, or several children of the PrimaryExpression concept, which means either the transformation menu should be triggered by the last PrimaryExpression child node, or by the single ApplicationEntity child node if the application happens not to contain any arguments at all (i.e. there are no PrimaryExpression children nodes inside the corresponding GenericApplication node).

We, therefore, create the default transformation menu for both of the concepts ApplicationEntity and PrimaryExpression. We enwrap the editor action inside the group, where we check for the following conditions:

- The parent of the current AST node has to be an instance of the GenericApplication concept.
- For the ApplicationEntity default transformation menu, the parent AST node must not contain any children of the PrimaryExpression concept.
- For the PrimaryExpression default transformation menu, the parent AST node's last child of the PrimaryExpression concept must be the current AST node.

The editor action is a simple include action, where we specify that the transformation menu to be included is the default transformation menu from the GenericApplication concept and that it applies to the parent AST node.

The implementation for the PrimaryExpression concept is depicted in Figure 3.15 (a similar approach would be used also for the ApplicationEntity concept). Since the PrimaryExpression concept also inherits from the TopExpression concept, it should include its default menu as well.



```
default transformation menu for concept PrimaryExpression
  section({ side transform : right }) {
    include default menu for TopExpression for current node
    ----------
    group
      variables
        << ... >>
      condition (editorContext, node, model)->boolean {
        node<> parent = node.parent;
        if (parent.isInstanceOf(GenericApplication)) {
          node<GenericApplication> ga = parent : GenericApplication;
          return ga.arguments.isNotEmpty && (ga.arguments.last == node);
        }
        return false;
      }
      include default menu for GenericApplication for
          targetNode(editorContext, node, model)->node<GenericApplication> {
        node.parent : GenericApplication;
      }
```

Figure 3.15: Default transformation menu for the `PrimaryExpression` concept

Understandably, there are also several concepts that inherit from the `PrimaryExpression`, hence they need to include the default transformation menu for the `PrimaryExpression` concept too. The concepts which editors additionally contain the child editor cells also need to follow the similar approach we described in this section.

The main idea behind the pattern is therefore not to define the transformation menus on the concepts which can actually trigger the action (such as `IntegerValue` representing an integer literal which has the necessary type of editor cell), but rather on the higher-level, possibly abstract, concepts (such as `TopExpression`) and to include whatever transformation menus they might have in their child concepts and descendants (descendant in terms of the concept inheritance). We used this pattern throughout most of the implementation of the transformation menus, since a direct implementation would be very difficult to maintain and prone to many mistakes.

3.3.4 Substitute Menu Actions

We use substitute actions whenever we need to allow the user to substitute certain AST nodes for another nodes. In the most cases we need to allow a seamless substitution of an abstract concept instance to a concrete one.

We will now take a look on how literals work in Frege-IDE and demonstrate the substitute menu actions on them.

`Literal` is an abstract concept which inherits from the `Term` concept that represents literals, lists and tuples. `Literal` has the following sub-concepts:

- `StringValue`: contains a single property of the string type. The value is enclosed with quotes.
- `CharValue`: represents a single character. The property is of the custom constrained type which allows only a single character to be typed,

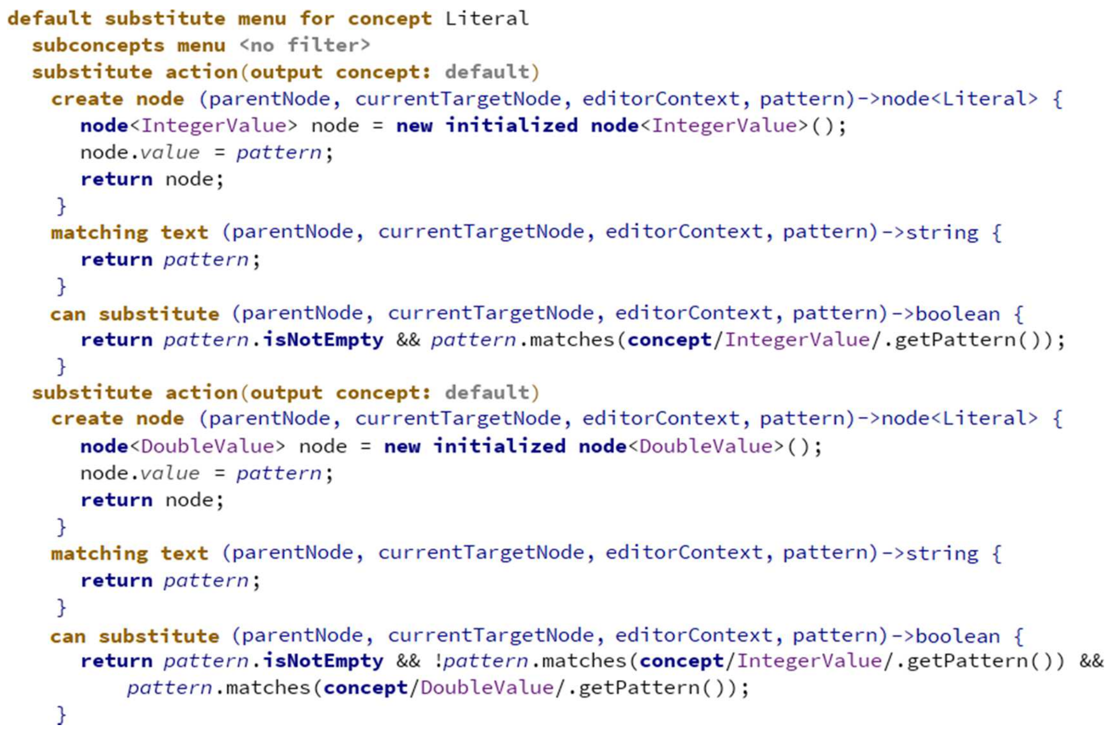
otherwise the node will not be validated correctly. Similarly to the `StringValue` concept, in the editor the value is enclosed with apostrophes.

- `BooleanValue`: it is an abstract concept. The editor consists of the single cell pointing to the content of the concept's alias. The two concrete concepts `TrueValue` and `FalseValue` have then aliases set to `true` and `false` that represents their actual value.
- `IntegerValue`: it is a concept with a single property. Its editor contains the single property editor cell.
- `DoubleValue`: similar to the `IntegerValue`, the concept contains only a single property. However, the property is of a different type to allow the floating-point numeric values to be typed instead of only the integer ones.

Let us assume a situation where there is a focus on an abstract `Expression` AST node. The node should be replaced by an instance of a 'more concrete' `Literal` sub-concept based on the value a user enters. We can think of the following scenarios:

- If a user is about to enter a string value or a character, he or she would most likely enter the corresponding apostrophe or the quote symbol before entering the inner content in the plain text editor. This means that quotation mark can serve as a trigger for entering string values while the apostrophe symbol for entering character literals. This is done by declaring alias of the `StringValue` concept to the `"` symbol and `'` for the `CharValue` concept.
- Boolean values are entered by typing either `true` of `false`. The similar approach with the concept alias works therefore here as well, i.e. we set the alias of the `TrueValue` concept to `true` and the alias of the `FalseValue` concept to `false`.
- When an integer value is entered, the node should be substituted for an instance of the `IntegerValue` concept. To cover all possible integer values, a single value in the concept's alias is not enough and thus the default substitute menu has to be created to handle the cases. We describe the process in Section 1.3.3 where we create the substitute menu for the `Literal` concept. We test the user-entered text for being an integer and if true, the abstract AST node is replaced by a new instance of the `IntegerValue` concept.
- Double values are handled in a similar manner as the integer values. We create a new substitute menu action for the `Literal` concept and check whether the user-entered text is a double value (but not an integer value since an ambiguity would arise). If true, we replace the current node with the new instance of the `DoubleValue` concept.

Figure 3.16 shows the implementation of the default substitute menu for the `Literal` concept. We only need to handle the two cases related to the concepts `IntegerValue` and `DoubleValue`.



```
default substitute menu for concept Literal
  subconcepts menu <no filter>
  substitute action(output concept: default)
    create node (parentNode, currentTargetNode, editorContext, pattern)->node<Literal> {
      node<IntegerValue> node = new initialized node<IntegerValue>();
      node.value = pattern;
      return node;
    }
    matching text (parentNode, currentTargetNode, editorContext, pattern)->string {
      return pattern;
    }
    can substitute (parentNode, currentTargetNode, editorContext, pattern)->boolean {
      return pattern.isNotEmpty && pattern.matches(concept/IntegerValue/.getPattern());
    }
  substitute action(output concept: default)
    create node (parentNode, currentTargetNode, editorContext, pattern)->node<Literal> {
      node<DoubleValue> node = new initialized node<DoubleValue>();
      node.value = pattern;
      return node;
    }
    matching text (parentNode, currentTargetNode, editorContext, pattern)->string {
      return pattern;
    }
    can substitute (parentNode, currentTargetNode, editorContext, pattern)->boolean {
      return pattern.isNotEmpty && !pattern.matches(concept/IntegerValue/.getPattern()) &&
          pattern.matches(concept/DoubleValue/.getPattern());
    }
```

Figure 3.16: Default substitute menu for the `Literal` concept

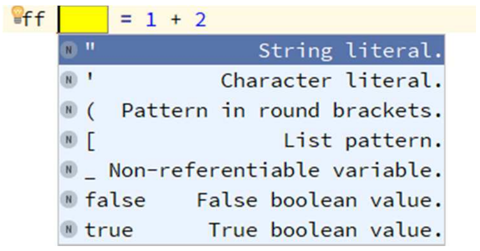
3.3.5 Wrap Substitute Menu

Wrap substitute menu is a special type of the substitution menu to be used when we need to populate the completion menu by instances of a different concept, than we will actually substitute the current AST node for.

Section 1.3.3 demonstrates a usage of the wrap substitute menu on the concept `PLiteral` which is replaced in the corresponding completion menu by the relevant entries related to the `Literal` concept. `PLiteral` concept inherits from the `PatternArgument` concept, which is used in the representation of the left hand side of the function definition (corresponds to the `patternTerm` symbol which is described in Section 3.2.5.1).

The Frege grammar tells us that the symbol `literal` may be created either by replacing the `patternTerm` symbol (part of the patterns of the function definition) or by replacing the `term` symbol (part of the expressions of the function definition). Since MPS does not allow the language designer to create a concept inheriting from multiple concepts at once, a workaround was used in the implementation of Frege-IDE. We created the concept `PLiteral` containing a single child - an instance of the `Literal` concept - and created a wrap substitute menu so that the `PLiteral` behaves similarly to the `Literal` concept.

Figure 3.17 depicts the code completion menu for substituting a function argument in Frege-IDE which is an instance of the `PatternArgument` concept.



```
ff      = 1 + 2
" String literal.
' Character literal.
( Pattern in round brackets.
[ List pattern.
_ Non-referentiable variable.
false False boolean value.
true True boolean value.
```

Figure 3.17: Completion menu for substituting a function argument in Frege-IDE

The completion menu on the example is, among others, now populated by the items discussed in Section 3.3.4 regarding the `Literal` concept (quote and apostrophe symbol, `true` and `false`). The items for creating instances for `IntegerValue` and `DoubleValue` concepts are not shown as they are automatically created upon typing the corresponding literal. The created instances are then automatically enwrapped inside the new instance of the `PLiteral` concept as implemented in the corresponding wrap substitute menu handler.

On a side note, the `Literal` concept can now be a child of the `PLiteral` concept. According to the transformation menu inclusion pattern from Section 3.3.3, the default transformation menu for the `Literal` concept should now include the default menu intended for the `PLiteral` concept as well, otherwise certain transformation actions would not be applicable.

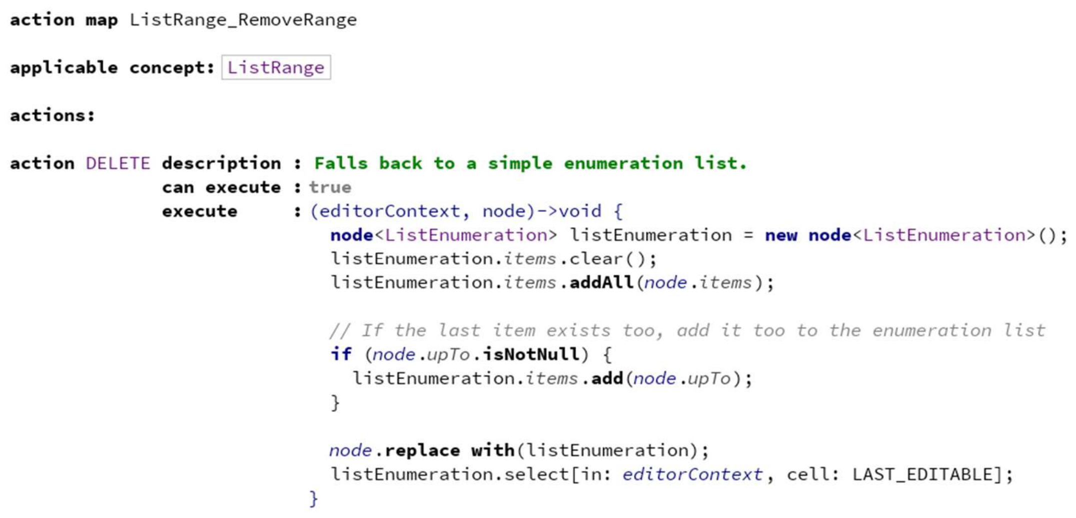
3.3.6 Cell Action Map

Side transformation menus allow us to transform a certain AST node to another when a specific text is written to the right or left of an editor cell. However, in some cases we may also want to put an event handler on a specific editor cell when it is selected, removed, or otherwise manipulated with. For these cases we use the cell action maps.

In Frege-IDE, most of the scenarios, in which we have to use the cell action maps, are simply connected to the editor cell deletion. In Section 3.3.2 we described side transformation menus in Frege-IDE and demonstrated, how we could allow the user to easily transform a list defined by enumeration to the ranged list. To allow the backwards transformation from the ranged list to the enumeration, we have to use the cell action map.

The ranged list's editor consists of constant cells representing the square brackets `[` and `]`, the child cells representing the enumerated items on the left, the constant cell for the `..` symbol and the child cell for the representation of the upper bound of the range. The user might expect to change the range list back to the enumeration upon removal of the `..` editor cell. While this is not the implicit behavior, we can create a handler for the corresponding editor cell.

Figure 3.18 shows the implementation of the cell action map for the aforementioned editor cell. The handler creates a new instance of the list enumeration concept, copies all of the items from the current list range (together with the upper bound if present) and replaces the current node with the new instance.



```
action map ListRange_RemoveRange

applicable concept: ListRange

actions:

action DELETE description : Falls back to a simple enumeration list.
              can execute : true
              execute     : (editorContext, node)->void {
                              node<ListEnumeration> listEnumeration = new node<ListEnumeration>();
                              listEnumeration.items.clear();
                              listEnumeration.items.addAll(node.items);

                              // If the last item exists too, add it too to the enumeration list
                              if (node.upTo.isNotNull) {
                                listEnumeration.items.add(node.upTo);
                              }

                              node.replace with(listEnumeration);
                              listEnumeration.select[in: editorContext, cell: LAST_EDITABLE];
                            }
```

Figure 3.18: Cell action map for the editor cell representing the `..` symbol in the list range concept

3.3.7 Seamless Definitions

Important aspect of Frege-IDE is a user-friendly editor. During the implementation we wanted to emulate the standard process of writing code in the plain text editor the most Frege users are used to. Since a program in Frege consists of a series of definitions, we wanted to focus on this part of the language and provide the best experience for typing new definitions in a Frege module.

In Section 3.2.3 we have discussed what kind of definitions there are and which are supported in Frege-IDE. The subset includes the following:

- Import declaration
- Fixity
- Type declaration
- Data declaration
- Local definition (function definition and type annotation)

Except the type annotation and function definition, all of the above definitions begin with a certain keyword that makes them easily distinguishable, such as `infix`, `type` or `data`. Defining an alias inside those concepts' structure aspect is enough to provide the user with a relatively user-friendly interface - for instance, upon typing `data`, MPS creates the instance of the corresponding data type declaration concept. However, type annotations and function definitions cannot be represented by a single keyword and require a careful analysis. We have made several observations, which will help us:

- When the user begins the definition with the left square bracket symbol `[`, or a constructor's name, or a literal, then it is clear he or she is providing a function or an operator definition. Consider the following examples:

    - `[a, b, c] = [1, 2, 3]`
    - `Just _ *** Nothing = Nothing`
    - `1 +-+ x = x`

- When the user begins the definition with the left bracket symbol `(`, then the step is ambiguous and has two valid outcomes:
    - The user may be trying to annotate an operator. Upon typing any operator symbol right of the bracket symbol, a transformation menu action can transform the corresponding AST node to a type annotation instance. Example: `(+-+) :: Int -> Int -> Int`
    - Upon typing anything else from what is allowed to be inside the patterns, the definition is a function definition as demonstrated by the following examples:
        - `(a, b) = (1, 2)`
        - `(1, c) = (1, 2)`
        - `([a, b], c) = ([1, 2], 3)`
- When the user types a new identifier (excluding the keywords associated with other definitions), it is not yet clear whether the definition will be a type annotation, or a function definition. There must be an ambiguous step in the process represented by a special concept. The outcome is determined by one of the following cases:
    - If the identifier is followed by `::` symbol, it is a type annotation. Example: `idf :: Int -> Int -> Char`
    - If the identifier is followed by a comma symbol, it is a type annotation, as the user is probably adding new items to collectively annotate. Example: `idf1, idf2 :: Int -> Int -> Char`
    - If the identifier is followed by an operator symbol, it is an operator definition. Example: `idf1 -+- idf2 = idf1 * idf2`
    - If the identifier is followed by the `=` symbol, it is either a constant function definition, or an operator definition. For example:
        - `idf = "Hello, world!"` (constant function definition)
        - `idf1 =+= idf2 = 0` (`=+=` operator definition)
    - Upon typing anything else from the set of allowed symbols for the patterns, the user is most probably providing arguments to the function definition. Example: `idf (a, b) [0] = [a .. b]`

The described observations form a rather complex decision tree, therefore we will describe the implementation only briefly. All definitions are concepts that inherit from the abstract `Definition` concept. We created the concept

`FunctionDefinition` which represents the step from the first observation (i.e. what we get upon typing either `[` symbol, or a constructor's name, or a literal). It is not the real function definition that also has to consist of the right hand side which is missing in this concept. Instead it is an 'incomplete definition', considered invalid until properly finished. The `FunctionDefinition` concept is composed of a single child representing the pattern, which has its own editor actions that allow the node to be transformed to the final definition form.

To allow the seamless substitution of the `Definition` node to the `FunctionDefinition`, we defined a wrap substitute menu for all of the three mentioned cases:

- **List pattern:** the menu handler wraps the list pattern concept. Upon typing the input symbol `[` which is normally a trigger for creating a list pattern instance, the handler creates a new instance of the `FunctionDefinition` concept instead and puts the new list pattern as its child.
- `PConstructor`: the menu handler wraps the concept representing a constructor application in a pattern
- `PLiteral`: the menu handler wraps the concept representing a literal in a pattern

By creating the wrap substitute menus, whenever an instance of the `Definition` concept is expected, a user may instead use one of the three options mentioned above and the handler will automatically wrap the corresponding AST node into a new instance of the `FunctionDefinition` concept. Thus, typing, for example, the literal `7` on an empty line would result in creation of the instance of the `FunctionDefinition` concept.

The case for the left bracket symbol `(` was handled in a similar manner. We created a new concept representing the brackets, which are empty at first and are considered to be an 'incomplete definition'. The right-side transformation menus associated with the left bracket then handle both of the possible outcomes mentioned in the second observation. The implementation is as follows:

- We created two transformation menu actions associated with the editor cell representing the left bracket symbol.
- The first action is triggered by typing any character from the set of allowed characters for custom operators (see Section 2.4). The menu handler transforms the current node into an instance of the type annotation concept.
- Upon typing anything else, the node is to be transformed to an instance of the `FunctionDefinition` concept.

The last observation is solved by creating a new concept and defining an appropriate substitute menu for the `Definition` concept to the new concept. The substitute action has to check the user-entered text for whether it actually can act as an identifier and that it does not equate to one of the reserved words (such as `true`, `false`, `type` or `infix`). After that it is necessary to implement all of the cases from that observation by creating the necessary side transformation menus. The approach is no different from the ones described in this section, so we will not go through it.

An interesting problem is to correctly transform the identifier to a reference in case the identifier is followed by `::` symbol, thus transforming the whole AST node to an instance of the type annotation concept. While this problem is related to scoping, if we have the list of available nodes representing the function names, it is sufficient to find the one that is the best match for the user-entered identifier. We pick one and create the new node referencing the picked node.

It is important to mention that the approach described in this section regarding the Frege-IDE editor is just one of many, since there is no exact set of rules for how the IDE should behave. We took this path to closely emulate the common plain text editors and as an experiment which we refer to later in Chapter 4. On the one hand we allow the user to type the Frege code as he or she may be used to, but now we offer only a limited assistance in referencing existing functions and operators when defining type annotations. There are advantages and disadvantages to both approaches.

3.3.8 Expression Operators

Frege language allows users to define custom infix operators with an almost arbitrary precedence. However, the precedence may be changed at any time and handling such an event properly would be complicated to implement. Instead we opted for keeping expressions in linear structures in contrast to a tree that would need to be recomputed and reshaped each time a precedence of an operator is changed. Evaluating type of an expression is therefore left to the typesystem aspect, where a special algorithm must be used to handle various scenarios.

To allow adding operators inside the expressions, we need to obtain a set of available operators first. This topic is described in Section 3.4 that deals with code completion feature and scopes. Once the set is obtained, we can define right-side transformation menu for the operands.

What we want to achieve is to add a new operator and then set focus to the new operand node upon typing an operator the user wants to use. However, there may be a case when the typed operator is a prefix of another. Consider, for instance, the operators `+` and `++`. When the user types an expression `x+`, it is not clear, whether he or she will continue by typing another `+` symbol, or the entered operator is finished. Fortunately, MPS simplifies the scenario by automatically handling ambiguous transformation menu actions once the user-entered text does not conform to any of the triggering `text` parts of the transformation menu anymore. For instance, this means that once a user types `x+y`, it is clear that the text that should have triggered one of the two aforementioned actions was `+` only. Thus, the corresponding action is activated and the operator `+` is added to the expression together with the new operand `y`.

Somewhat unpleasant feature of MPS is that we cannot specify a list of strings in the `text` part of the side transformation menu action, but rather only a single string that can trigger the action. Therefore we have to return the closest operator that begins with the user-entered string. For instance, let us assume that the only operators visible in the given Frege module are `+-+` and `:-:`. When the user types the symbol `+`, we need to return the `+-+` string as that is the closest operator beginning with the `+`. However, if the user tries to type a non-existent operator, such as `+:`, we must return a completely irrelevant string to prevent the action from triggering. This approach is necessary as the number of operators is dynamically changed by the user and thus cannot be each represented by a separate transformation menu action.

However, the described approach does not account for the case where one operator is a prefix of another, such as operators `+` and `++`. If the transformation menu action returned the string `+` as the closest operator for the user-entered text `+`, the action would be immediately triggered as there is no ambiguity. We can force the ambiguity by creating a second transformation menu action with the similar handler, as the first one. In the `text` part, however, we return the second-closest operator beginning with the user-entered text. For instance, given the operators `+`, `++` and `+-+`, the corresponding two menu actions would return the following strings based on the user-entered text:

- If the user types `+`, the first action returns `+` and the second returns `++`.
- If the user types `++`, the first action returns `++` and the second returns an irrelevant string, such as 'illegal pattern'.
- If the user types `+-`, the first action returns `+-+` and the second returns an irrelevant string, such as 'illegal pattern'.

To implement the feature, we have used a custom trie-like data structure. We give the trie the user-entered text and find out if there is a leaf node with the same prefix as the given input. If true, we return the operator, if not, we return 'Illegal pattern'.

3.3.9 Final Remarks

While the editor aspect of MPS provides powerful options to allow the creation of a user-friendly IDE, it takes a considerable effort and resources. Rather than analyzing different usage scenarios of the language, it may be sometimes easier to define a set of executable actions (intentions) which the user may choose from to create the required AST nodes. As it has been mentioned, since the MPS does not rely on lexers and parsers to process the code, we have to take a reverse approach and that is costly.

3.4 Code Completion

An important feature of many great IDEs is the context-aware code completion. This feature speeds up the development process by putting less pressure on the programmer's memorization of the identifiers, members and other language constructs. It usually has a form of a list (see Figure 1.5) which is invoked either automatically or manually from which the user may select the necessary item. A lot of simplified implementations only include a list of members whose name starts with the given prefix or all possible words that were written in the corresponding code. Context-aware code completion differs from such implementations by providing the list of only those constructs that are actually 'visible' in the given context. Consider the following piece of code in Frege:

```
length (y:ys) = 1 + length ys
getTop (x:xs) = x
```

The example introduces two new functions into the given Frege module: `length` and `getTop`. On the left hand side of the function `getTop` we have declared variables `x` and `xs` whereas on the left hand side of the function `length` we introduced the variables `y` and `ys`. Their corresponding right hand sides must contain only their own declared variables from their left hand sides. For example, the user should not be able to pick the variable `y` from the context-aware code completion menu in the `getTop` function's expression (we assume there are not any visible functions named `x` or `y` in the given Frege module).

MPS provides tools to implement the context-aware code completion menu by taking advantage of the constraints aspect. As described in Section 1.6, the constraints aspect allows the language designer to restrict the set of referenceable targets for given concepts by specifying the custom scope object. In this section, we analyze the overall problem with the code completion, discuss possible solutions and provide several examples.

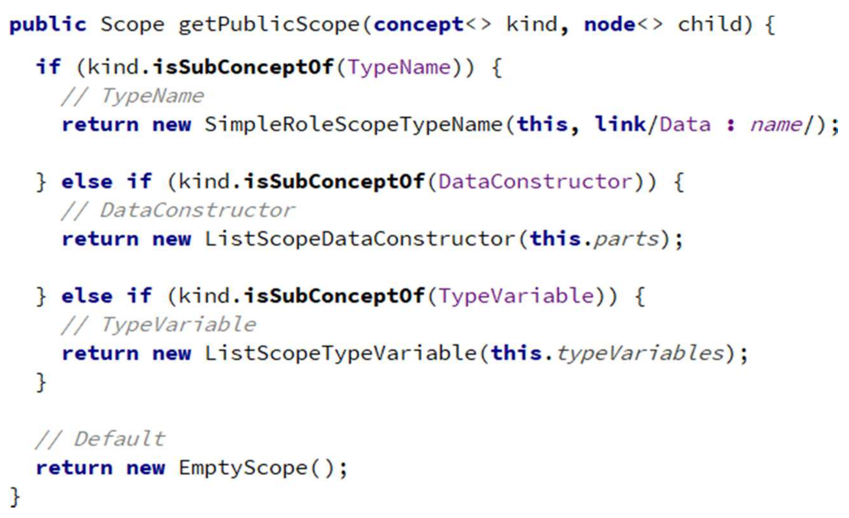
3.4.1 Scope

Scope is an object in MPS which defines a list of targets that can be referenced. The language designer can specify a concrete scope for the concepts that contain a reference in their structure aspect. This can be done by defining a new referent constraint in the constraints aspect of the given concept.

Let us look at the data type and type synonym declarations in Frege-IDE. In Section 3.2.4 we discuss the structure aspect of the corresponding `Data` and `Type` concepts. The type name these language constructs introduce into the given Frege module is represented by the `TypeName` concept. The name is also enwrapped in the smart reference concept called `TypeReference`, which is used, for example, as a child in the concept representing the usage of the type name in the static type declaration. Without specifying any scope for the concept `TypeReference`, the user is able to pick the type name from any of the Frege modules defined in the current MPS model. To restrict the set only to the type names from the current or actually imported modules, we can create a new scope instance where we specify the list of type names that are actually referenceable.

However, specifying the list of referenceable nodes for each relevant concept would be tedious and difficult to maintain. Instead we can take a different approach and proclaim certain concepts to be *scope providers*. What we mean by this is that they contain certain children which are referenced by other concepts. In the example above, both `Data` and `Type` concepts are, according to our definition, scope providers, because they contain the child concept `TypeName`, which is further referenced, for example, by the `TypeReference` concept. Upon request, these concepts should provide a list of AST nodes that are available to them.

Figure 3.19 shows a simplified version of the method that provides the scope of the `Data` concept. The method is defined in the concept's behavior aspect. The `Data` concept consists of the type name, data constructors and type variables. Each of its parts may be referenced, as the data type declaration creates a new type name used in the static type declarations, defines constructors used in the function definitions and optionally specifies type variables which are referenced by its constructors, as in the example of the data type `Maybe`.



```
public Scope getPublicScope(concept<> kind, node<> child) {

  if (kind.isSubConceptOf(TypeName)) {
    // TypeName
    return new SimpleRoleScopeTypeName(this, link/Data : name/);

  } else if (kind.isSubConceptOf(DataConstructor)) {
    // DataConstructor
    return new ListScopeDataConstructor(this.parts);

  } else if (kind.isSubConceptOf(TypeVariable)) {
    // TypeVariable
    return new ListScopeTypeVariable(this.typeVariables);
  }

  // Default
  return new EmptyScope();
}
```

Figure 3.19: Method `getPublicScope` for returning the requested scope in the `Data` concept behavior aspect

In the implementation above, we use custom classes for creating new instances of the scope. We differentiate between the kinds of the scope requested - if the scope for the concept `TypeName` is requested, then that is what the method returns. If the scope for the completely unrelated concept is requested, the method returns an empty scope containing no nodes at all.

The approach with scope providers then forms a hierarchy. The `Data` and `Type` concepts both inherit from the `Definition` and are therefore children of the root `Skeleton` concept. `Skeleton` represents a single Frege module. Upon request for the scope it delegates the call to all of its children. If its child contains another children, it delegates the call to them recursively. In the end, the scopes are merged and returned. This way, in its most simplified form, the `TypeReference` concept can request the scope from the root concept `Skeleton`. The returned scope then contains the list of available type names the `TypeReference` concept can reference.

3.4.2 Scope Hierarchy Pattern

In the previous section we described the notion of scope providers. The concepts in Frege-IDE form a tree and delegate the requests for the scope creation to their child concepts recursively. If a certain concept on the path contains children of the requested type, it is called the scope provider. Upon request it creates the new scope with the list containing the relevant child nodes and returns it to the parent. The concepts that do not provide anything return the empty scope. These scopes are eventually merged on the root level and returned to the requesting node.

However, this hierarchy only gathers all of the nodes of the requested type and returns it in the form of the scope object. To also restrict the nodes based on the requester context, we used a custom pattern.

Let us take for example the function definition. The function definition consists of its left and right hand side. The left hand side specifies the function's name together with the arguments it accepts. As described at the beginning of Section 3.4, the function's name should be visible to all of the sibling definitions whereas the declared variables inside its arguments should be referenceable only from its own right hand side.

We will illustrate the pattern on the following example:

```
f x y = x + y
g = f 1 7
```

Figure 3.20 shows a visualization of the AST for the given two function definitions. Function `f` declares two variables `x` and `y` which are referenced from its right hand side. On the other hand, function `g` references from its right hand side the function `f`. This is valid as the function `f` is its sibling. However, the `g` function's right hand side cannot contain references to the variables `x` and `y` as they are defined in the other function definition.

From the MPS perspective, the concept related to the function definition is the scope provider. It provides the scope either for the function's name or for its declared variables. Based on the grammar, the function's name is indistinguishable from the variables and thus the problem cannot be solved simply by requesting the scope for a different type of the concept. Instead the decision to either include or exclude the declared variables from the returned scope has to be made on the function definition's level. The corresponding concept has to know where the request for the scope comes from. If the request comes from its right hand side, it returns all of its variables together with the function's name. If the request for the scope comes from the 'outside' (e.g. the call to the function definition `f` was made by its sibling function `g`), it returns only the function's name.

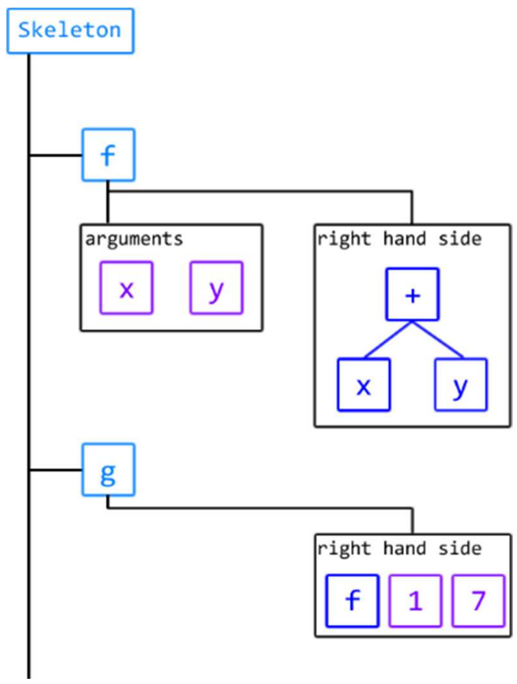


Figure 3.20: A visualization of the AST for a single Frege module with two function definitions

However, this is not enough as the function `f` in the example above is referenced from the function's `g` right hand side which is the actual requester of the scope. The request for the scope therefore gets to the node associated with the `g` function definition and returns the function's `g` name and its variables (which in this particular case there are not any). The function definition for the `g` then has to include its parent scope as well to also cover the reference to its sibling `f`.

Thus, the pattern to implement the scope for referencing concepts is as follows:

- The concept containing a reference requests the scope from its closest ancestor that is a scope provider (provided the given concept is not the scope provider itself). The root concept `Skeleton` is a scope provider and thus such an AST node always exists.
- The scope provider creates the requested scope based on the AST node the request came from.
    - If the request came from its parent, it creates its own scope as necessary and optionally delegates the call to its children (function definition does not delegate the call to the children nodes). It merges its own scope with the scopes from its children and returns it to the parent.
    - If the request for the scope came from one of its children, it creates its own scope as necessary, delegates the call to its parent (which delegates the call recursively up to the root node) and optionally delegates the call to the other children as well. Then the scopes are merged into the single one and returned to the requesting node.

- The root concept Skeleton always delegates the call to all of its relevant children and returns the merged scope.

To help us implement the pattern, we have created the interface DCScopeProvider which extends the built-in interface ScopeProvider. It contains two main methods: one called by the child nodes and one called by parent scope providers. The default implementation, which may be overridden using behavior MPS aspect, is to always include only the parent scope upon a request from children. The scope creation is then delegated recursively up to the root node, which then calls the second method (for calling by parents) on all of its relevant children. The default implementation of that is to, similarly, call the same method recursively on all of the relevant children of the given concept implementing the DCScopeProvider interface.

3.4.3 Import and Export

Section 3.2.4 describes the concepts related to the import and export declarations. Import declaration is represented by the Import concept, which is a child of the root Skeleton concept. The Import contains a reference to an existing Frege module represented by the Module concept which is also a child of the Skeleton. The export declaration is a part of the Frege module definition at the top of the Frege program, thus it is included in the aforementioned Module concept.

According to the scope hierarchy pattern mentioned earlier, the concept Skeleton delegates the requests for the scope creation to all of its relevant children. Thus the responsibility to create the scope for the imported modules is left to the Import concept.

Consider the following example of two Frege modules:

```
ExportExample module:
 -------------------------
module mps.frege.ExportExample (ff, gg) where

ff = 0
gg = 1
hh = 2
```

```
ImportExample module:
 -------------------------
module mps.frege.ImportExample where
import mps.frege.ExportExample as EP (ff)

ii = ff + EP.gg
```

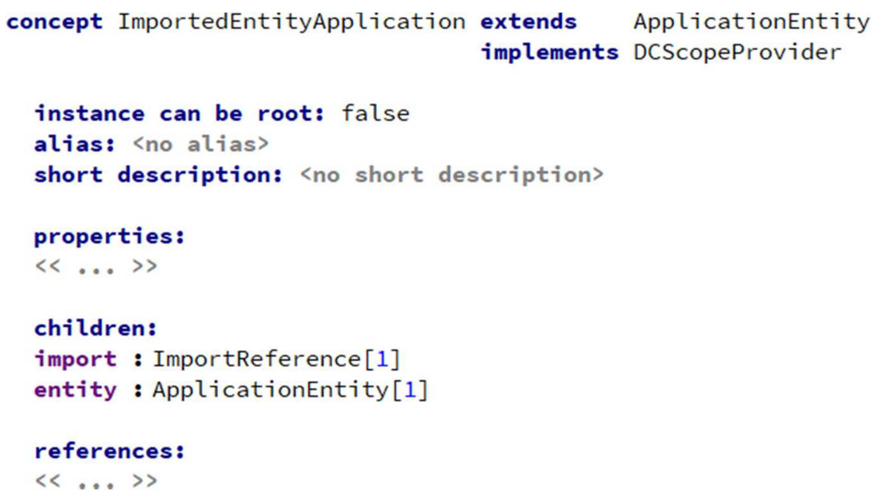
Upon the request for the scope, the Import concept looks into the referenced Module node. In our example, the import declaration import mps.frege.ExportExample references the AST node from the ExportExample module corresponding to the statement `module mps.frege.ExportExample (ff, gg) where`. Then it asks for the scope. There are two possible outcomes for the given request:

- The module declaration exports all of the definitions from the current Frege module (i.e. no brackets are specified in the corresponding declaration). The `Module` node therefore delegates the request to its parent node `Skeleton` which then creates the scope based on the scope hierarchy pattern. However, in this special case it is important not to include the items from the imported modules if there are any, since those are omitted by default.
- The module declaration exports only some of the items from the current Frege module (see the example above). The `Module` node has to iterate through the specified exported items and create the scope containing all of them.

However, the import declaration does not always import all of the items from the referenced Frege module into the current namespace. As in the example above, the module `ImportExample` imports only the function `ff` from the `ExportExample` module while the function `gg` has to be accessed using the qualified name `EP.gg`. This means that the `Import` concept has to be able to provide two different scopes: one for all of the items from the referenced module which are accessed by their qualified names, and the second one for the items that are specified in the corresponding brackets (if the `hidden` clause is used in the import declaration, then the items not specified within the brackets have to be included in the second scope).

The second type of the scope is created simply by filtering the scope from the referenced `Module` node by the items specified within the corresponding brackets in the import declaration. It is the default scope the `Import` concept provides and the items it contains are easily referenceable by concepts such as `VariableReference` used for applying existing functions and variables inside the expressions.

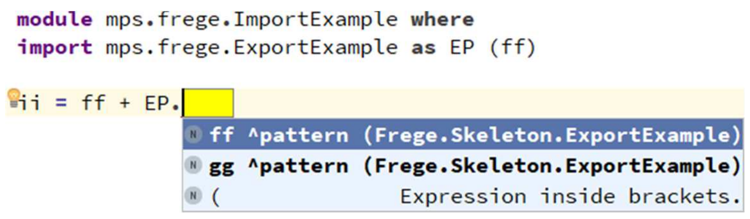
The first type of the scope is provided by the `Import` concept upon the specific direct request. In Frege-IDE, we created special concepts to denote the usage of qualified names. For instance, in Section 3.2.5.3 we described the `ApplicationEntity` concept of which there is also the sub-concept `ImportedEntityApplication`. Figure 3.21 depicts its structure aspect. The concept `ImportedEntityApplication` contains a reference to the import declaration. It also contains the `ApplicationEntity` concept as the child node, which references a node from the imported module. `ImportedEntityApplication` is a special type of scope provider which delegates the scope creation solely to the referenced import declaration. It requests from the `Import` node its first type of the scope that contains all of the exported definitions from the corresponding module and nothing else.



```
concept ImportedEntityApplication extends    ApplicationEntity
                                  implements DCScopeProvider

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

  properties:
  << ... >>

  children:
  import : ImportReference[1]
  entity : ApplicationEntity[1]

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

Figure 3.21: Structure aspect of the ImportedEntityApplication concept in Frege-IDE

On the example at the beginning of the current section, the function ii contains in its right hand side the expression EP.gg. In Frege-IDE this is represented by the AST node of the concept ImportedEntityApplication. EP consists of the editor cell that points to the corresponding Import concept's presentation (in this case the import's alias EP). The child entity (ApplicationEntity concept) is restricted by the scope returned from the corresponding Import node, which contains the function definitions ff and gg and thus these are the only functions or entities that can be applied in the given context. Figure 3.22 depicts the ImportExample module in Frege-IDE. The corresponding code completion menu contains only the aforementioned items ff and gg when using the qualified name starting with EP.



```
module mps.frege.ImportExample where
import mps.frege.ExportExample as EP (ff)

ii = ff + EP.
             ff ^pattern (Frege.Skeleton.ExportExample)
             gg ^pattern (Frege.Skeleton.ExportExample)
             (                 Expression inside brackets.
```

Figure 3.22: ImportExample module in Frege-IDE

Additionally, the Import concept has to provide a very specific scope for its children as well. First, it should allow to import only the other modules and not the one currently being defined. Then, we have to provide the scope for the items to be specified inside the brackets, which will describe, what items will be actually imported into the current namespace. The scope must not include the items from the current Frege module and thus the request for the scope creation from the Import concept's children is not delegated to the Import's parent node. All in all, the scope hierarchy pattern does not fully apply here as the current case is fairly specific.

3.4.4 Implicitly Imported Library

A library with several standard functions and operators has been created to simulate the behavior of the implicit library Frege-Prelude. The module is simply labeled as `Default` in Frege-IDE (`mps.frege.Default`).

Every new module imports the library implicitly upon its construction by searching the visible modules and finding the one with the corresponding name. The module is then imported with the *hidden* flag, which makes it invisible in the editor.

The module contains basic arithmetic operators, such as `+`, `-`, `*`, `/`, `%` as well as boolean comparison operators, for example `<`, `>`, `<=`, `>=` and `==`. The module is not a complete copy of the Frgee-Prelude library and serves only as a demonstration of the capabilities of Frege-IDE. However, it can be easily extended with the new functions, data types or operators.

3.5 Type Checking

To implement a simple type checking capability into Frege-IDE, we used the typesystem aspect in MPS. The current feature has some limitations and supports only type checking of the functions, where a type annotation is provided. In this section, we will go through several examples, describe the problems we encountered during the implementation and discuss the final result.

3.5.1 Types

As discussed in Section 3.2.6, we wanted to include the following types in Frege-IDE:

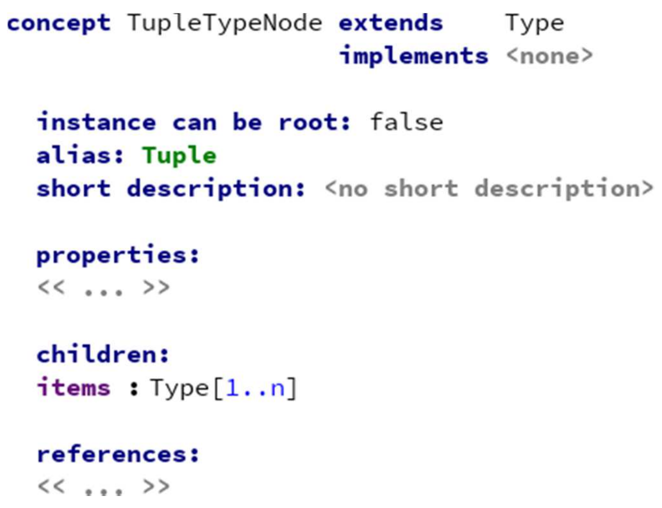
- `Bool`
- `Char`
- `String`
- Numeric types (`Int`, `Double`)
- Tuple
- List
- Function types

Custom algebraic data types were omitted due to the scope of this work. Every other type mentioned, however, has to be modeled by a new concept, as explained in Section 1.7. Additionally, we created a special concept `Unknown` to denote the types of expressions we could not or did not want to infer the types of.

Section 1.7 discusses the implementation of the concept representing the list type. The similar approach is used for the remaining complex types, which include tuple and function types.

The concept representing the tuple type in Frege-IDE is named `TupleTypeNode`. It inherits from the `Type` concept from the MPS BaseLanguage and contains an arbitrary amount of children based on the concrete tuple it represents the type of. The children are also sub-concepts of the `Type` concept.

Figure 3.23 shows the structure aspect of the `TupleTypeNode` concept in Frege-IDE.



```
concept TupleTypeNode extends    Type
                      implements <none>

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

  properties:
  << ... >>

  children:
  items : Type[1..n]

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

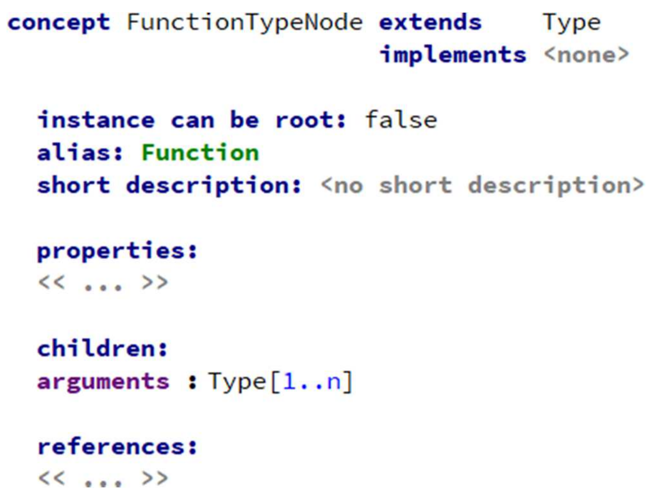
Figure 3.23: Structure aspect of the `TupleTypeNode` concept in Frege-IDE

From the structural point of view, the concept representing the function type in Frege-IDE (named `FunctionTypeNode`) is the same as the `TupleTypeNode` concept. It also contains an arbitrary amount of children representing the types of the function's arguments with the last child representing the return type. The two are, however, not separated due to the currying technique in Frege, which allows the function's return type to be also a function. Consider the following example:

```
multiplyThree :: Int -> Int -> Int -> Int
multiplyThree x y z = x * y * z
curriedMultiplyThree = multiplyThree 1
```

In the example above, the type annotation of the function `curriedMultiplyThree` is `Int -> Int -> Int` which could be considered from a certain point of view to be the return type of the function application `multiplyThree 1`. To ease the implementation, we do not draw any lines between the type of the function's argument and its return type in the Frege-IDE typesystem aspect.

The final implementation of the concept `FunctionTypeNode` in the structure aspect is depicted in Figure 3.24.



```
concept FunctionTypeNode extends    Type
                         implements <none>

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

  properties:
  << ... >>

  children:
  arguments : Type[1..n]

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

Figure 3.24: Structure aspect of the `FunctionTypeNode` concept in Frege-IDE

3.5.2 Infix Expressions

In Frege, the user may define custom infix operators with different associativity and precedence. From the semantic point of view, the operator is simply a function accepting two arguments and returning a certain result. A result of such a function may be again a function accepting additional arguments.

Frege differs from most programming languages such as C# or C++ that allow mostly only operator overloading. This means that in these languages a developer may adjust the behavior of the operators for different input types, but not define new operators or change the precedence or associativity of the existing ones. Thus the syntax tree generated by the parser in these languages always stays the same. However, that is not the case in Frege. In Frege the user may create a completely new operator of an (almost) arbitrary precedence, altering the syntax tree that is generated for the expressions in which the new operator is used.

To ease the implementation of the editor in Frege-IDE, we have decided to keep the expressions in linear data structures instead of keeping them in the form of binary trees regarding the structure aspect of Frege-IDE. The latter would require handling events related to the user changing precedence or an associativity for all of the infix operators and non-trivial reconstruction of the AST. Instead we decided to compute the expression binary tree only in the typesystem aspect.

The user may change the precedence or associativity of a custom operator by entering the following statement in the Frege module:

```
infixl 5 +++
```

The example above sets the custom operator `+++` to be left-associative with the precedence `5`. According to the Frege specification, the statement can be used only in the same module in which the custom operator `+++` is also defined.

In Frege-IDE the statement above corresponds to the concept `Fixity` which inherits from the `Definition` concept. Thus, it is a child of the `Skeleton` concept. Given the operator reference, we can get the `Skeleton` root node in which the given operator is defined. Then we can iterate its children and find the corresponding `Fixity` node that adjusts the precedence and the associativity of the given operator. If no related `Fixity` node is found, then the operator is by default non-associative and has the precedence `16`.

To implement the type checking of infix expressions, we took the following approach:

1. Infer the types of the sub-expressions recursively, such as expression enclosed within brackets, or expressions inside the terms (e.g. tuples or lists).
2. The result of the previous step is an array of operands with the operators in between them. The type of the operands is known from the previous step. The type of the infix operators is given by their corresponding type annotations (if the type annotation for an operator is not provided, then the type of the whole expression is set to `Unknown`).
3. Construct the binary expression tree for the current expression.
4. Check the types of the binary expression tree's nodes. Set the type of the whole expression according to the root node of the binary expression tree.

To demonstrate the approach, let us consider the following expression in Frege:

```
ff = 4 * 5 * 6
```

First, the types of the sub-expressions need to be inferred. The operands `4`, `5` and `6` are simple integer literals and thus their types is `Int`.

In the second step, we find out the type of the operator `*`. Due to the absence of parametric polymorphism in Frege-IDE the type annotation of the operator `*` is set to `Double -> Double -> Double`.

Next the binary expression tree is constructed. The operator `*` is a left-associative operator, which means the expression can be rewritten in the following manner:

```
ff = ((4 * 5) * 6)
```

The precedence does not play any role in the current expression as there are no other operators than `*`. The constructed binary expression tree is depicted in Figure 3.25.

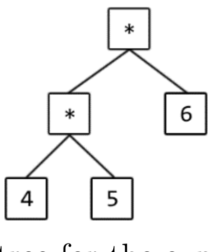


Figure 3.25: Binary expression tree for the expression `4 * 5 * 6` in Frege-IDE

In the last step, the types of the nodes of the constructed expression tree are checked. The types are checked in the bottom-to-top approach, i.e. first the type of the subtree given by the expression `4 * 5` is checked. Both children `4` and `5` are of the type `Int` which are acceptable arguments for the function with the type annotation `Double -> Double -> Double`. The type of the given subtree is therefore `Double`. Similarly the remaining part of the expression tree is checked. The type of the whole expression is then set to `Double`.

Assuming that we can easily infer the types of the sub-expressions and find out the types of the operators used in the given expression we will now describe how the binary expression tree is constructed. We used a derivation of the standard algorithm for translating infix expressions to postfix which uses a single stack. Our algorithm uses two stacks and iterates through the elements of the expression three times in total. The pseudocode of the corresponding algorithm is as follows:

```
function ConstructTree( Expression ):

  create stack S
  lastPrecedence ← 0

  for each element e in Expression (taken left-to-right):
     if e is an operator:
        if precedence of e < lastPrecedence:
           item ← HandlePrioritized( S, precedence of e )
           push item into S

        lastPrecedence ← precedence of e

     push e into S

  return HandlePrioritized( S, MAX_VALUE )


function HandlePrioritized( Stack, Precedence ):

  create stack S
  lastPrecedence ← 0
  lastAssociativity ← none

  for each element e in Stack (taken from top):
     if e is an operator:
        if precedence of e <= Precedence:
           return ConstructTreeFromStack( S )

        if precedence of e = lastPrecedence
           and (associativity of e <> lastAssociativity
           or associativity of e = none):
           error

        if precedence of e < lastPrecedence
           or associativity of e = right:
           item ← ConstructTreeFromStack( S )
           push item into S
        lastPrecedence ← precedence of e
        lastAssociativity ← associativity of e

     pop e from Stack
     push e into S

  return ConstructTreeFromStack( S )


function ConstructTreeFromStack( Stack ):

  root ← top from Stack
  pop top from Stack

  for each two elements (operator, operand) in Stack (taken from top):
     create binary tree:
        tree.root ← operator
        tree.left ← root
        tree.right ← operand

     root ← tree
     pop operator and operand from Stack

  return root
```

The algorithm expects on the input a non-empty expression in the form of an array of operands with operators in between them. The entry function `ConstructTree` iterates the elements of the expression from left to right and pushes them onto the first stack. It keeps the track of the precedence of the last operator and if an operator with a lower precedence is encountered, it lets the function `HandlePrioritized` construct the binary expression tree for the items with the higher priority.

To illustrate the work of the algorithm, let us consider the following expression:

```
x1 + x2 + x3 + x4 * x5 * x6 . x7 . x8 . x9
```

Let us assume the three operators used in the expression have the following attributes:

- `+` is a left-associative operator with the precedence 1
- `*` is a left-associative operator with the precedence 3
- `.` is a right-associative operator with the precedence 2

In the example above we are not actually interested in the types of the variables `x1 - x9` or the type annotation of the operators as the constructed binary expression tree depends only on the two mentioned properties (associativity and precedence).

The first step of the algorithm pushes the items onto the first stack until the operator `.` is encountered. Then it calls the function `HandlePrioritized` to handle the items from the stack up to the operator `+` which has a smaller precedence than the `.` operator. This ensures the subtree for the prioritized part of the expression `x4 * x5 * x6` is constructed first.

In the second step, the function `HandlePrioritized` iterates the items in the stack which means it looks at the items in the reversed order as they came first in the input expression. The function stops either when the first stack is empty or when an operator with a small precedence is encountered, thus handling only the prioritized part of the expression. As the function iterates the items, it can perform the following two important checks:

- There must not be several operators with the same precedence but different associativity in a sequence.
- There must not be several non-associative operators in a sequence.

At this point, the function decides how it creates the expression tree for the given part of the input expression. If the right-associative operator is encountered, the subtree can be created right away as the items of the expression are visited in the right-to-left order. If the left-associative operator is encountered, the function has to wait until the operator with a different precedence is encountered. The algorithms puts the items onto the new stack and when it hits the new operator, it lets the function `ConstructTreeFromStack` construct the expression tree for the postponed items from the new stack.

In the example above, at the time of the first call of the function `HandlePrioritized`, the given stack contains the following items in order from the top: `x6 * x5 * x4 + x3 + x2 + x1`. The function iterates the items up to the operator `+`. The only operator encountered up to the `+` is operator `*` which is the left-associative operator, therefore it pushes the items onto the new second stack. Finally, it lets the function `ConstructTreeFromStack` create the expression tree from the new stack with the following items: `x4 * x5 * x6` (ordered from the top).

The last function `ConstructTreeFromStack` takes the items from the given stack and creates the binary expression tree for them. The function expects the encountered operators to be left-associative with an exception of creating the expression tree for only a single operator with two operands. Figure 3.26 illustrates the created binary expression tree for the input stack with the items `x4 * x5 * x6`.

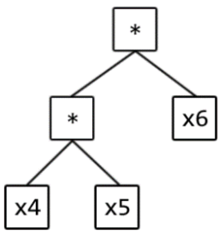


Figure 3.26: Binary expression tree for the expression `x4 * x5 * x6`

At this point, the algorithm continues with the function `ConstructTree`. The first stack contains the following items: `((x4 * x5) * x6) + x3 + x2 + x1` where `((x4 * x5) * x6)` represents the single item in the form of the binary expression tree as depicted above. The function continues to iterate the items from the input expression until there is nothing left. The stack now looks as follows: `x9 . x8 . x7 . ((x4 * x5) * x6) + x3 + x2 + x1` and the execution is passed to the `HandlePrioritized` function.

In the second step, the function `HandlePrioritized` stops only after all of the items from the given stack are processed. It first encounters the operator `.` which is the right-associative operator and thus the binary expression tree is constructed immediately. Then it continues with the items `x3 + x2 + x1` on the input stack where the operator `+` is, again, left-associative and thus the construction of the tree has to be postponed until a different operator with a different precedence is encountered, or the first stack does not contain any more items. The second stack with items `x1 + x2 + x3 + (((x4 * x5) * x6) . (x7 . (x8 . x9)))` is passed to the `ConstructTreeFromStack` function which constructs the final binary expression tree as depicted in Figure 3.27.

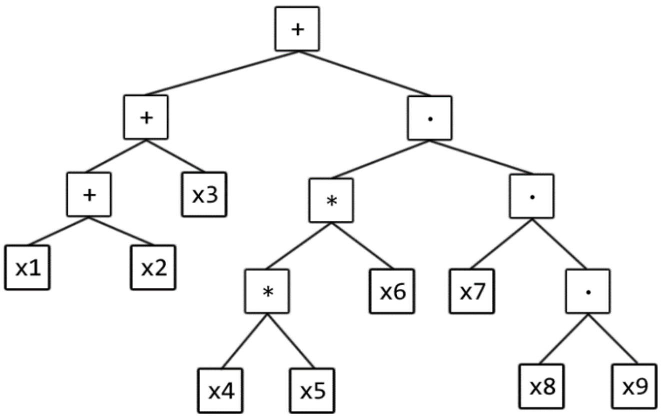


Figure 3.27: Binary expression tree for the expression `x1 + x2 + x3 + x4 * x5 * x6 . x7 . x8 . x9`

The last step of our approach is to check the types of the binary expression tree's nodes. As we have the tree constructed, this is done easily by checking the type of both left and right subtrees recursively and comparing them to the type of the operator for each operator node. The return type of the operator application is also the type of the whole subtree defined by the current node. This step, however, requires the implementation of the function application, which is discussed in Section 3.5.3.

3.5.3 Function Application

In Section 3.5.1 we discussed the overall structure and implementation of the concept related to the function type. The corresponding `FunctionTypeNode` concept is structurally similar to the concept representing the tuple type and contains at least one child of the `Type` concept to represent the types of the function's arguments or the function's return type.

The function application is represented by the `GenericApplication` concept. It contains the single child of the `ApplicationEntity` concept to represent what is being applied (for instance, function, operator or a bracketed expression). In this work, we completely omitted the type checking for the user defined types which includes type synonym and data type declarations. This means we will not be dealing with the constructor application which is normally also part of the corresponding `GenericApplication` concept.

The implementation of the type checking for function application is done in the typesystem aspect of the `GenericApplication` concept. The Frege-IDE first looks at the inferred type of the `ApplicationEntity` child of the corresponding concept. Due to the type inspection the `when concrete` block described in Section 1.7 has to be used.

`ApplicationEntity` may or may not be of the function type. This depends on the expression being applied. Consider the following two examples:

```
six_1 = (max 4) 6
six_2 = (max 4 6)
```

In Frege-IDE, both constant functions `six_1` and `six_2` would contain an instance of the `GenericApplication` concept in their respective right hand sides. `(max 4) 6` is an application of the bracketed expression `max 4` with argument `6`. `(max 4 6)` is an application of the bracketed expression with no arguments. The result of the expression surrounded within the brackets in the former case is a function, whereas in the latter it is an integer value.

The resulting type of the function application has to be properly inferred depending on the `ApplicationEntity` and the arguments provided. Let us demonstrate the process on the following function:

```
multiplyThree :: Int -> Int -> Int -> Int
multiplyThree x y z = x * y * z
```

The function `multiplyThree` accepts up to three `Int` arguments. The following examples of functions apply the `multiplyThree` in two different ways:

```
ff = multiplyThree 1
gg = multiplyThree 1 2 3 + 7
```

The function `ff` depicts the currying technique and the type annotation of the expression `multiplyThree 1` is therefore `Int -> Int -> Int` (i.e. the type of the expression is still a function). However, in the function `gg` the expression `multiplyThree 1 2 3` uses all three arguments, thus leaving us with the resulting type `Int` instead of the function type. Therefore the result of the function application may not be a constant function.

3.5.4 Function Definition Type Inference

There are two main, relatively independent, parts when it comes to the type inference of a function or an operator definition. The return type of either is given by its respective right hand side. On the other hand, types of the arguments the function or an operator accepts are not always unambiguously inferable. For example, consider the following function definition:

```
ff x y = 0
```

While the function ff accepts arguments x and y, its result does not depend on the two. Thus the function can be called with any type of arguments.

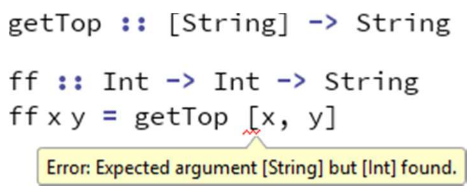
However, in some cases it is possible to deduce the type of the function arguments depending on their usage in the corresponding function's right hand side as demonstrated by the following example:

```
getTop :: [String] -> String
getTop [] = "No elements"
getTop (x:xs) = x
…

ff x y = getTop [x, y]
```

On the example above, the function getTop accepts only a list of the String items. This means that for the expression getTop [x, y] on the right hand side of the function ff to be valid, the x and y must be arguments of the String type too. This approach to the type inference is relatively complex to implement and we did not include it in the final work. Instead we opted for a reasonable compromise:

- If the type annotation is provided for a function or an operator definition, it is considered to be the actual type of that definition. The return type from the type annotation is then checked against the type of the right hand side of the corresponding function or the operator definition.
- If the type annotation is not provided, the types of the function arguments are considered ambiguous and are never checked (the types are set to Unknown). The return type of the function is then based on the type of the expression in its right hand side.

Based on this, Frege-IDE infers the type for the example function ff at the beginning of this section as follows: ff :: Unknown -> Unknown -> Int. If we additionally specify the type annotation for the function ff, then the arguments will have the specified types. Figure 3.28 shows an example of incompatible type annotation for the function definition ff x y = getTop [x, y] at the beginning of this section.



```
getTop :: [String] -> String

ff :: Int -> Int -> String
ff x y = getTop [x, y]
```

Error: Expected argument [String] but [Int] found.

Figure 3.28: Example of an invalid function arguments usage in Frege-IDE

3.5.5 Arguments Type Decomposition

In the previous section we described that we want the types of function arguments to be inferred from the corresponding type annotation, if it is provided.

A certain challenge is posed by the fact that a variable, which we intend to infer the type for, may be included deep within a subtree representing a certain function argument. Let us consider the following example:

```
gg :: [[Char]] -> [Char]
gg [['a'], ['b', 'c'], [x]] = [x, x, x]
```

The type annotation of the function `gg` describes that the first argument of the function `gg` is a list of a list of `Char` items. Thus the pattern `[['a'], ['b', 'c'], [x]]` in the function definition below is also a list of a list of `Char` items, which makes the variable `x` to be of the type `Char`.

To implement the type inference for the pattern variables, Frege-IDE needs to look at the related type annotation. It looks into the corresponding argument's type and tries to decompose it.

We will illustrate the approach on the example above. Frege-IDE knows that the argument `[['a'], ['b', 'c'], [x]]` corresponds to the type `[[Char]]` from the related type annotation. This means that each item of the list has to correspond to the type `[Char]`:

- `['a']` should be `[Char]`
- `['b', 'c']` should be `[Char]`
- `[x]` should be `[Char]`

However, the items of the list are also more complex language constructs created from the simpler ones. Frege-IDE therefore checks each list item individually:

- `['a']` should be `[Char]`, therefore `'a'` should be `Char`
- `['b', 'c']` should be `[Char]`, therefore `'b'` and `'c'` should be `Char`
- `[x]` should be `[Char]`, therefore `x` should be `Char`

The type `Char` cannot be further decomposed and thus the Frege-IDE's work is over.

We have implemented the feature by properly setting up the typesystem aspect in the concepts related to the pattern in the left hand side of the function definition. Most pattern concepts can be associated with a certain type. For instance, the concept representing the list pattern is associated with the list type whereas the `PLiteral` concept for representing the literals inside patterns is associated with the corresponding primitive type depending on the literal used (e.g. `Int`, `Bool`). The concept representing a pattern variable cannot be associated with a specific type. Instead the type of the corresponding variable will be either `Unknown` or the type deduced from the related type annotation.

The implementation follows the following scheme:

- The concept representing the whole left hand side of the function definition looks into the related type annotation. It compares the amount of declared arguments in the type annotation to the amount of arguments defined in the current left hand side. If the numbers are equal, then the current node assigns the type of each argument from the type annotation to each its child.
- The concrete pattern concept compares the type assigned by its parent to the type the current concept is normally associated with. For instance, the list pattern concept checks that the type assigned by its parent is `[t]` where `t` is the type of the corresponding list item. The pattern concept then decomposes the type (if applicable) and assigns the inner types to the corresponding children. In the example with the list pattern, the concept assigns to each its child the type `t`.
- The type of the variable is the type assigned by the corresponding node's parent. If no type was assigned, the variable is of the `Unknown` type.

4. Evaluation

In this work, we have focused on creating a projectional IDE for the functional language Frege. We wanted to see whether projectional IDEs offer more convenience or any other advantage over the regular text-based IDEs in assisting developers with creating programs in functional languages in general. We have included support for code completion, simple error and type checking and refactoring. This chapter looks into the convenience of the usage of such an IDE, how it compares to the classic plain-text IDEs and what are its advantages and disadvantages over the conventional plain-text IDEs.

4.1 Editing Programs in Frege-IDE

Frege-IDE is built on top of the MPS platform which is used for creating projectional editors. This puts several restrictions on how a typical program may be written or edited.

We have already mentioned in this work that projectional editors differ significantly from the plain-text editors. The user is not editing the code in the text form, but rather works with the corresponding AST directly, having a great impact on how the code is further processed. This brings many advantages, such as allowing for an easier extension of an existing language or bringing non-conventional visual elements for representing and altering the program's code.

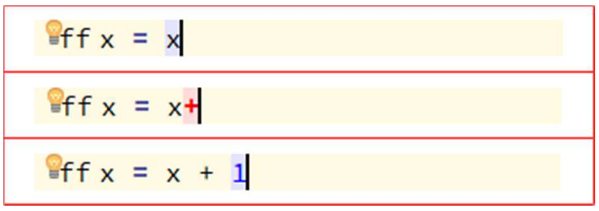
However, using a projectional editor for writing code brings certain limitations, too. As the code is not actually a text, the user is limited in what he or she can alter or type. In Frege-IDE, only certain editor cells are modifiable. The rest is static, resulting in a removal of the whole subtree they are part of when trying to change the text they represent. Additionally, only certain modifications of the corresponding AST are allowed which have to be included by the language designer himself or herself.

The editor is only as good as it is designed. The user of the language may type only certain text in appropriate places in the code defined by the language designer in advance. Every editing feature has to be implemented, even a seemingly trivial functionality such as adding new operator with operands to an existing expression. This puts a considerable amount of work on the language designer, who has to think of multiple usage scenarios of the IDE he or she is developing.

Even though the projectional IDE built on top of the MPS platform will never be as flexible as the editing of the plain text, there are advantages to this approach. They include, for instance, the possibility to force the users of such an IDE to adhere to a specific coding style. The language designer can limit the usage of unwanted features of the language or to prevent the programmers from writing 'unaesthetic' code.

The language designer has several options for how to build the editor for the IDE he or she is creating. Transformation and substitute menu actions can be used to handle events related to writing particular text phrases. Cell action maps are used to handle deletion, selection or other manipulation of concrete editor cells. Last but not least, there is the notion of intentions which are manually-invoked actions designed to be usable in specific places in the code to handle various scenarios.

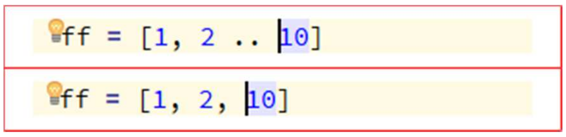
Side transformation menus are actions which are triggered when a certain text pattern is written next to an editor cell. In this work, we used the menus, for example, for adding a new operator to the right of an expression. The usage scenario is depicted in Figure 4.1. First, the user positions the caret to the right of the expression `x`. Then, he or she types the infix operator which is visible in the current namespace, such as `+`. Upon finishing by typing, for example, a literal, the action is immediately triggered, transforming the current AST to include the additional operator with the new operand. The action then positions the caret on the new operand.



```
ff x = x
ff x = x+
ff x = x + 1
```

Figure 4.1: Process of adding a new operator with an operand into an existing expression in Frege-IDE

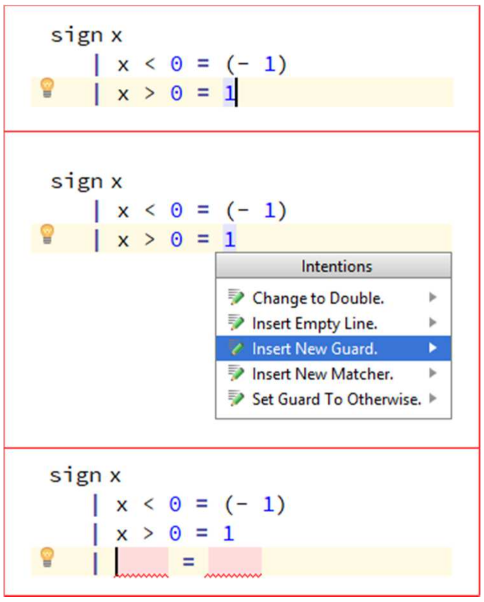
We used cell action maps throughout the work to handle deletion of certain editor cells. As described in Section 3.3.6, an example usage scenario is the range list AST node where we delete the `..` symbol which corresponds to a specific editor cell. The example is depicted in Figure 4.2 in which the user first positions the caret to the beginning of the upper bound literal `10`. Then he or she presses the **backspace** key invoking the `DELETE` event handler for the corresponding editor cell representing the `..` symbol. The handler transforms the node to the enumeration list node and places the former upper bound literal as the enumeration list's new item.



```
ff = [1, 2 .. 10]
ff = [1, 2, 10]
```

Figure 4.2: Process of deleting the range symbol `..` in the range list in Frege-IDE

Certain actions may seem ambiguous from the user's perspective. Pressing **enter** key in certain scenarios adds a different construct in Frege-IDE than he or she might expect. Additionally the user may not know how to create a certain AST or how to execute a specific AST transformation. For the unintuitive cases like these, we used the MPS intentions. As described in Section 1.5, they are special user-interface elements that allow the user to execute predefined actions in certain places in the code. Figure 4.3 shows an example usage in Frege-IDE. In the example, the user tries to add a new guard to the definition of the function `sign`. However, adding the guard requires the transformation of the corresponding AST representing the function definition to include the new child. From the user's perspective it is not clear how to add the new guard. In the plain-text editor, the process would require typing a vertical line symbol with an indentation on a new empty line. Simulating such a sequence of operations is not an easy task in MPS. Instead we defined several intentions to allow the user to execute the necessary action as he or she sees fit.



```
sign x
    | x < 0 = (- 1)
    | x > 0 = 1
```

```
sign x
    | x < 0 = (- 1)
    | x > 0 = 1
```

```
sign x
    | x < 0 = (- 1)
    | x > 0 = 1
    |  = 
```

Figure 4.3: Process of adding a new guard into the right hand side of the `sign` function definition in Frege-IDE

In Figure 4.3 the user of Frege-IDE positions the caret into the right hand side of the definition of the function `sign`. He or she then invokes the intentions menu via key combination **alt + enter** and selects the appropriate item. This invokes the corresponding handler which then transforms the current AST and repositions the caret into the newly created guard.

Overall, while MPS has powerful tools that allow the language designer to create great editors for IDEs, it is still not feasible to simulate the experience of the plain-text editors completely. Editing the plain text provides a lot of flexibility for what the user can do with the written code. On the other hand, projectional editor is cognizant of the document's underlying structure which helps the user greatly in navigating him or her to what can be entered and still considered valid. This allows the language designer to craft an editor which is predictive and saves the user a lot of unnecessary keystrokes. The potential drawback to this is, however, the necessity to learn working with the designed editor as it may come unintuitive to a lot of unexperienced users. We believe this is the case with Frege-IDE as well. While Frege-IDE may seem restrictive at first, given time the user will develop Frege programs in Frege-IDE faster than in the ordinary plain-text editor. The nature of the IDE also limits the potential errors in the code the user can make.

4.2 Features and Limitations

Frege-IDE contains a lot of features to assist Frege developers with writing code. We have already discussed the editor, its nature and high-level design in the previous section. In this section we take a deeper look on the editor, the features Frege-IDE provides and how they can help Frege developers with writing programs.

Most of the Frege language constructs are easily representable by the MPS concepts. A simple reserved keyword or a symbol is usually enough to comprehend what the user wanted to enter. Consider the following statement in Frege:

```
data Maybe a = Just a | Nothing
```

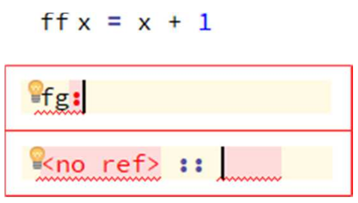
The example above is mentioned throughout the work multiple times and denotes the declaration of a new data type. In this case `data` keyword is enough to comprehend the intention of the user to declare the new data type. However, not all Frege language constructs are representable by a single string.

In Section 3.3.7 we discussed the approach for allowing the user to define new functions or provide type annotations. The most challenging part was to understand from the user input, which one of the two is currently being typed. However, there were several possibilities on how to approach the matter. The one we took and discussed was mainly demonstrating the capabilities of the MPS platform and showing how similar scenarios can be generally handled. The additional considered approaches include the following:

- Intentions aspect can be used to allow the user to choose, what kind of definition he or she plans to create.
- Creating a wrap substitute menu over all defined functions and operators would allow for creating a type annotation immediately once a name of an existing function or an operator is typed. Anything else would be considered to be the function definition.

Ability to invoke intentions menu on an empty line may improve the overall experience with Frege-IDE, since the user would now have several options on how to create the new definition. Choosing a specific item from the intentions menu is also less confusing for the user than typing a text that eventually gets transformed to either of the two aforementioned options. We have implemented this feature in Frege-IDE due to the mentioned reasons as well. However, it is questionable whether leaving only this option of providing the function definition or type annotation would be sufficient. Different users might have different opinions on the matter.

The second approach can be also useful as it is quite convenient to have the code-completion menu populated with the defined functions and operators. During our evaluation, we have encountered situations where we tried creating a type annotation of a function and were unsuccessful due to a small typographical error. The error meant that Frege-IDE was not able to find the definition with the provided name and resulted in an unset function reference. Figure 4.4 captures such a scenario.



```
ff x = x + 1

fg:

<no ref> ::
```

Figure 4.4: An unsuccessful attempt of providing a type annotation for the function `ff` in Frege-IDE

However, this approach has also a few downsides. Typing a new name for a new function definition would not be resolvable until the name was not a prefix of any other, already defined, function. This is due to Frege-IDE not being able to tell whether the user is not trying to type in a name of an existing function instead to provide the type annotation for the corresponding function. This is not resolvable until the name is completely unambiguous. For instance, if the current Frege module contained a definition for a function named `foo`, the user would not be able to type `fo` and provide the definition for the new function `fo` (however, the workaround with selecting an item for creating a new function definition from the intentions menu would work). In the end, we leave the answer to what is the best approach for the given scenario rather unanswered and only speculate over several options, none of which is perfect. The projectional editor while being powerful has a different behavior than a conventional plain-text one and has some limitations.

Additional feature in Frege-IDE that can greatly ease the development process is the context-aware code completion. The implementation of the feature in Frege-IDE was allowed by utilizing the notion of MPS concept references. Similarly to the plain-text IDEs, the completion menu can be populated only by the exiting items in the code. However, consider the following example:

```
five = 1 + four
  where
    four = 1 + 3
```

In the example above, functions `four` is defined within the `where` block of the function definition `five`. Since the expression `1 + four` references the function `four`, it has to be defined after the definition of the function `four` in Frege-IDE. In plain-text IDEs this does not play any significant role, since the code completion menu would simply not contain the item `four` if the user specified the expression `1 + four` first. This limitation is posed due to how references work in MPS. The user needs to keep the correct order of providing definitions in Frege-IDE. Every language construct which references another construct needs to be specified after that construct it references. On the other hand, this provides a powerful refactoring tool where the change of the original AST node is immediately reflected on all of the referencing nodes. This was discussed in Section 1.3 and illustrated in Figure 1.10.

The last major feature we will mention from Frege-IDE is the type checking. The feature infers the type of expressions, checks the correctness of provided functions arguments and compares the return types of the function definitions to their corresponding type annotations. Type checking is restricted only to the built-in types mentioned in Section 3.5.1 and cannot compare the usage of the custom data types or type synonyms. However, this limitation is posed by the scope of this work rather than being somehow related to MPS.

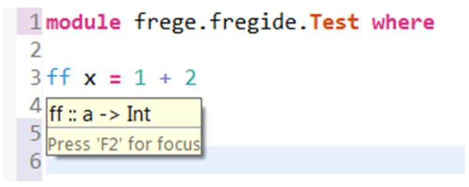
4.3 Comparison with Plain-Text IDE

fregIDE is a plugin for the Eclipse platform which adds, among other things, a support for syntax highlighting, code completion and type checking of Frege programs [10]. It works as an extension of the built-in plain-text editor of the Eclipse platform.

We have tried the plugin to compare it to our Frege-IDE and see, what advantages and disadvantages there are to a projectional IDE over a plain-text based one. We have chosen this specific plugin due to its popularity and extensive support.

fregIDE is a rather robust system and offers many features we could not afford to implement in Frege-IDE. First, it provides a support for the whole Frege language and not just its subset. Its type checking capabilities cover almost all of the cases and do not pose the limitations on existence of the function's type annotation, as we had to make. The code completion feature is similar to ours, however, fregIDE provides the full support for the implicitly included Frege-Prelude library. It also has no problem referencing the items imported from other Frege modules represented as text files. This is a major disadvantage for our Frege-IDE which can work solely with the Frege modules implemented in Frege-IDE itself. This is, unfortunately, given by the fact that MPS cannot work with the text directly and has to keep the code written by the user in tree-like data structures at all times. At present it is not even possible to transfer the documents between different MPS language projects as they are usually represented by completely different ASTs.

An example of fregIDE usage in Eclipse platform is depicted in Figure 4.5. Hovering mouse over a function displays its type annotation.



```
module frege.fregide.Test where

ff x = 1 + 2
```

ff :: a -> Int
Press 'F2' for focus

Figure 4.5: An example of a usage of fregIDE

When trying fregIDE plugin ourselves, we experienced some performance issues with the plugin and difficulties with the code completion menu during testing which was not the case with our Frege-IDE. Frege-IDE keeps the user-written code already processed in the necessary data structures and as such does not suffer from the lexical and syntactical analysis, which takes a certain amount of time to process. However, the issues may have very likely been related to the Eclipse platform rather than the plugin itself and the actual performance of the plugin does not present any real downside when compared to our Frege-IDE.

The fregIDE plugin is based on the plain-text code editing and as such offers a great flexibility on how the user can write and edit the Frege code. On the other hand, in our Frege-IDE, the designed editor takes care of the visual appearance of the defined AST nodes and does not require its user to spend much time pressing unnecessary keys or writing unnecessary symbols. For instance, in Section 4.1 in Figure 4.1 we showed an example of adding operators with operands to an expression. The relevant nodes are automatically visually separated by whitespaces. The written code has a unified structure dependent only on the underlying AST itself rather than counting on users to write the clean code.

Overall, we consider the editor to be one of the most powerful features of Frege-IDE. When used correctly, it saves its users a lot of time and troubles of keeping the code clean. Its predictive abilities also mean the user does not need to type everything - Frege-IDE puts the necessary keywords and symbols based on the context where the new AST nodes are created. This also helps the novice Frege programmers to better navigate in the code and easily understand what language constructs are allowed in what context.

At the same time, for an experienced Frege developer used to programming in a certain coding style, the editor in our Frege-IDE might seem to be limiting. It does not allow for switching between coding styles without changing the editor itself. The code is represented in AST and cannot be arbitrarily changed as in the plain-text editor. Furthermore, only certain editor cells may be copied into the clipboard and pasted elsewhere in the code, which puts additional restrictions on the user. However, there are situations in which these restrictions may be helpful, such as the aforementioned limitation of unwanted features of the language or enforcing a certain coding guideline in programming teams.

In the end, while fregIDE with its much larger support of Frege features is a better pick for most of the serious Frege developers, we believe Frege-IDE still has its place in the development community and in certain cases might prove to be an even more efficient tool than the regular text-based IDE.

Conclusion

In this work, we have analyzed and implemented an IDE infrastructure on top of the JetBrains MPS platform for a subset of the Frege language. The final application, called Frege-IDE, can assist developers with writing, editing, and testing programs in Frege. The IDE includes support for refactoring, code completion and type checking.

Frege-IDE differs from most IDEs in regards that it is a projectional editor, rather than being centered on a code written in a plain text. This brings certain restrictions on how the user may work with the Frege code.

Since all of the source code is represented in the form of AST, the user can enter only the allowed characters in the appropriate places in the code. Certain textual patterns invoke transformations of the underlying AST, others can substitute a specific AST node for another. However, the user is limited by the designed editor in what he or she can do.

While seemingly not as flexible as plain-text based IDEs, it can enforce recommended practices when writing code or restrict the programmers from using undesirable language constructs. The projectional editor only shows what the underlying AST looks like and thus also keeps the textual details of the code, such as whitespaces or indentation, fully automatic and off the developer's mind, so he or she can focus on the development process itself. Additionally, the editor is predictive when used correctly, saving the user unnecessary keystrokes and making the process of writing code faster and arguably more convenient. However, this requires a certain time investment from the user to learn how to work with Frege-IDE in an effective way.

Frege-IDE additionally brings its users support for context-aware code completion and simplified type checking. The features allow the developers to detect the potential errors early in the implementation process thus allowing them to be even more efficient. While during our evaluation the features worked well, it has to be noted that they do not necessarily bring anything new or different from the similar features in their text-based-IDE counterparts.

This work also serves as a demonstration of the capabilities of JetBrains MPS platform and to see, whether it can be used for development of projectional IDEs for functional languages, such as Haskell or Frege. In our experience, we found that the MPS platform is really robust and offers many different ways how to solve typical problems a language designer may come across in his or her work. However, we also lacked some features we needed when developing Frege-IDE, especially the ones related to the editor aspect and had to use a few workarounds. Additionally, the documentation found on the JetBrains official website is rather short and could use more examples. That being said, we were still able to create an IDE we wanted and which, in our opinion, works well for the Frege language.

When speculating whether projectional editors offer more convenience for editing purely functional languages compared to plain-text editors in general, we cannot provide a definitive answer. We believe there are advantages to both approaches. There are many restrictions a projectional editor puts on the user used to work with a certain language in the plain-text editor, who may then feel the environment to be limiting. We feel like the fewer features a language has, the better projectional IDE for the language may be designed. In complex languages, such as Java, or C++, the user has many options how to structure his or her code. This inherently goes against the philosophy of projectional editors, where each language construct should have its own visual appearance and editor, usually not very customizable. All in all, in our opinion, the convenience of a projectional editor is not related to a language being functional or imperative, but rather to the cardinality of the set of the features the language has and, possibly, to the coding style most users of that language are used to.

Future Work

Project Frege-IDE is open for future extensions. These can include extending the set of supported features of Frege language, improving the type checking capabilities and user experience with the environment's editor. Additionally, the built-in Frege libraries were not implemented in this project and could be included in the future work as well.

Bibliography

[1] CAMPAGNE, Fabien. 2015. *The MPS Language Workbench, Volume I.* Second edition. CreateSpace Independent Publishing Platform. p. 19-21, 35-44, 52-54, 130-132. ISBN 9781497378650

[2] WECHSUNG, Ingo. 2016. *Frege Project on Github.* Online; accessed 14 July 2018. URL: <https://github.com/Frege/frege>

[3] MAKARKIN, Aleksey. 2017. *Basic Notions of JetBrains MPS.* Online; accessed 14 July 2018.
URL: <https://confluence.jetbrains.com/display/MPSD20173/Basic+notions>

[4] KOŠČEJEV, Sergej. 2017. *Transformation Menu Language in JetBrains MPS.* Online; accessed 14 July 2018.
URL: <https://confluence.jetbrains.com/display/MPSD20173/Transformation+Menu+Language>

[5] *Haskell Wiki - Foreign Function Interface.* Online; accessed 14 July 2018.
URL: <https://wiki.haskell.org/Foreign_Function_Interface>

[6] WECHSUNG - Ingo, LAUPA - Yorick. 2017. *Differences between Frege and Haskell.* Online; accessed 14 July 2018.
URL: <https://github.com/Frege/frege/wiki/Differences-between-Frege-and-Haskell>

[7] HUDAK - Paul, PETERSON - John, FASEL - Joseph. 2000. *A Gentle Introduction to Haskell - Pattern Matching.* Online; accessed 14 July 2018.
URL: <https://www.haskell.org/tutorial/patterns.html>

[8] *Frege-Prelude Public Interface.* Online; accessed 14 July 2018.
URL: <http://www.frege-lang.org/doc/frege/Prelude.html>

[9] WECHSUNG, Ingo. 2014. *The Frege Programming Language.* Online; accessed 14 July 2018. p. 29-31.
URL: <http://www.frege-lang.org/doc/Language.pdf>

[10] WECHSUNG, Ingo. 2018. *fregIDE Tutorial.* Online; accessed 14 July 2018. URL: <https://github.com/Frege/eclipse-plugin/wiki/fregIDE-Tutorial>

Attachments

The attached CD has the following content:

- `MPS/`
  - Contains the JetBrains MPS (version 2018.1) installation file for Microsoft Windows operating systems
  - In case of using a different operating system, other variants of MPS can be obtained at <https://www.jetbrains.com/mps/download/>
- `Frege-IDE/`
  - The Frege-IDE project as described in this work for JetBrains MPS 2018.1
  - Contains the defined language and a single solution with examples
- `Grammar/`
  - The Frege grammar in EBNF as used in the official Frege compiler implementation
- `Reference/`
  - The Frege language reference in PDF format by Ingo Wechsung
- `Text/`
  - Text of this thesis in PDF format