Extending Spoofax with a REPL

Final Report

Gerlof Fokkema
Jente Hidskes
Skip Lentz

Delft University of Technology

TU Delft

Extending Spoofax with a REPL

Final Report

by

Gerlof Fokkema
Jente Hidskes
Skip Lentz

To obtain the degree of Bachelor of Science in Computer Science
at the Delft University of Technology,
to be presented July 1st, 2016 at 9:30.

Committee:	Prof. dr. Eelco Visser,	Client, TU Delft
	Ir. Hendrik van Antwerpen,	Supervisor, TU Delft
	Otto Visser	Bachelor Project Coordinator, TU Delft

TU Delft

Preface and Acknowledgments

This report comprises the end of the TU Delft Computer Science Bachelor Project; the final, compulsory project at the Delft University of Technology to attain the Bachelor of Science in Computer Science.

The client during this project was the PL group at the TU Delft, who *"conduct research into concepts and techniques for programming language design and implementation. The flagship product of the group is the Spoofax Language Workbench"*. The Spoofax Language Workbench was to be extended with a Read-Eval-Print Loop that would allow language designers and users to interactively work with the domain specific languages generated with Spoofax.

The purpose of this report is to describe the project as it was carried out over the course of ten weeks. It describes the process and aims to inform the reader about the problem, the design and the implementation of the solution to this problem. Finally, recommendations are made to the client to improve the delivered product.

Acknowledgments

We would like to thank Eelco Visser for the project proposal and the opportunity to do our project under his supervision, Hendrik van Antwerpen for being an excellent coach, Vlad Vergu for his collaboration and assistance with DynSem and Gabriël Konat for answering several of our questions.

Summary

This report comprises the end of the TU Delft Computer Science Bachelor Project; the final, compulsory project at the Delft University of Technology to attain the Bachelor of Science in Computer Science.

"The Spoofax Language Workbench supports the definition of all aspects of textual languages using high-level, declarative meta-languages. From a language definition using these meta-languages, Spoofax generates full-featured Eclipse and IntelliJ editor plugins, as well as a command-line interface."

A feature that Spoofax is lacking is a Read-Eval-Print Loop (REPL). A REPL is an interactive programming environment that takes single expressions, evaluates them and prints the result(s). REPLs facilitate exploratory programming and debugging and are therefore a nice addition to Spoofax.

Two contributions have been made. First, a functioning REPL has been created, comparable in features to those of popular programming languages such as Python and Haskell. Second, changes have been contributed to Spoofax that have been integrated into the main repository.

To successfully complete the project, extensive knowledge needed to be gained of the conceptual ideas behind programming language implementations, and of the Spoofax API implementing these concepts.

Despite having faced significant challenges during the start of the project, the most important goals as set forth in the problem description have been achieved. Therefore, the project team is still satisfied with the end result: while not all requested features have been implemented, it has been shown that it is possible to create a functioning REPL for any language defined in Spoofax, requiring only additional configuration to a language definition.

Contents

1 Introduction

General purpose programming languages such as Java and C have historically been the most prevalent in the field of computer science. Nowadays a wide variety of domain specific languages is also available, each of which is designed to efficiently solve particular domain specific issues. *"Domain-specific languages (DSLs) provide high expressive power focused on a particular problem domain. They provide linguistic abstractions and specialized syntax specifically designed for a domain, allowing developers to avoid boilerplate code and low-level implementation details"* [11]. Well known examples of such DSLs include HTML for creating websites and SQL for dealing with relational data.

At the Delft University of Technology, the Programming Languages group researches and develops domain specific languages. To facilitate prototyping and creation of DSLs, the Spoofax Language Workbench was developed and first published in 2010 [11]. Since then Spoofax has had several releases and will soon be arriving at version number 2.0. The description below is taken from the Spoofax website: *"The Spoofax Language Workbench supports the definition of all aspects of textual languages using high-level, declarative meta-languages. From a language definition using these meta-languages, Spoofax generates full-featured Eclipse and IntelliJ editor plugins, as well as a command-line interface."*

A feature that Spoofax is lacking is a Read-Eval-Print Loop (REPL). A REPL is an interactive programming environment that takes single expressions, evaluates them and prints the result(s). REPLs facilitate exploratory programming and debugging and are therefore a nice addition to Spoofax.

This report details the implementation of such a REPL as part of a TU Delft Computer Science Bachelor Project. This report is structured as follows. An overview of the required background needed to understand the problem domain is given in chapter 2. In chapter 3 the problem definition is then framed in the context of this background. The design and implementation of the final product are discussed in chapter 4 and chapter 5. Chapter 6 evaluates both the final product and the process leading up the final product. The whole project is reflected upon in the discussion in chapter 7. Recommendations regarding the final product are made in chapter 8. Chapter 9 concludes this report.

2 Background

This chapter gives the background information required to understand the rest of this report. If the reader is already well-versed in Spoofax, familiar with the concept of a REPL and understands literate programming, this chapter can be skipped.

The first section, section 2.1, briefly explains what Spoofax is and what services it provides to the language designer. Section 2.2 gives a quick overview of what a REPL is and what advantages it offers to programmers. Lastly, section 2.3 discusses literate programming.

2.1 The Spoofax Language Workbench

Spoofax is a platform that allows for giving a completely *declarative* definition of a programming language and accompanying integrated development environment (IDE) support [11]. Such a platform is called a *language workbench*.

In discussing programming languages, it is useful to distinguish the following three concepts: an *aspect* of a language, the formal *specification* of that aspect, and the actual *implementation* of that aspect that adheres to the specification. Any programming language can be divided into multiple distinct aspects. In figure 2.1, four common aspects of programming languages are shown, along with the relation between the formal specification and implementation for each aspect. An example of an aspect is the *syntax*: its formal specification is done with grammar rules, and its implementation traditionally consists of a tokenizer and a parser.

A key property of Spoofax is that it allows a language designer to give a complete specification of each aspect of their language, and automatically derives the implementations of those aspects from their respective specifications. Spoofax provides a range of high-level, declarative *meta-languages* to declare language specifications.

This section goes over the aspects that come into play with the development of a language and how Spoofax tackles each of these aspects with its meta-languages. First, the section goes over some of the elements that make up the specification of each language aspect[1]. A language commonly consists of the following aspects, shown also in figure 2.1,

[1]This section follows the structure of the language specification portion of the compiler construction course at the TU Delft. The slides can be found here: `http://tudelft-in4303.github.io/lectures/specification/`.

of which a selection is discussed in more detail throughout the rest of this section:

1. **Syntax:** The first aspect concerns what textual representations of a program are syntactically valid. Its specification consists of a grammar specified with context-free grammar productions. A parser provides an implementation of this specification, by mapping a textual representation of a program to an abstract syntax tree (AST) representation. In Spoofax, the syntax is declared with a domain specific language (DSL) called SDF3.

2. **Static semantics:** A parsed AST then optionally goes through static analysis (type checking, name binding and variable scoping), to test if the program is well-formed. Static analysis is the implementation of an aspect called *static semantics*. The static semantics of a language can be formally specified by *type and/or name binding rules*. Spoofax provides two DSLs that can specify the two distinct parts of the specification: the TS Type Specification language and the NaBL name binding language.

   This language aspect is not discussed in this section, but it is explained in the research report in appendix C.1.2.

3. **Program transformation:** A well-formed AST can then be transformed, for example for desugaring or optimization. This execution step falls under the aspect of *program transformation* and can be formally specified by *term rewrite rules*. For the specification of this aspect, Spoofax provides a DSL called Stratego.

   This aspect is also not discussed in this section. The relevant section from the research report can be found in appendix C.1.3.

4. **Dynamic semantics:** Next the transformed AST is either compiled or interpreted, thereby providing a means of execution. The aspect of *dynamic semantics* defines what the behavior is of a program upon execution. The formal specification of this aspect can be done with *reduction rules*, although there are other ways to accomplish this. For example, interpretation can be seen as the transformation of a program to a single value. In Spoofax, the dynamic semantics can be defined with either Stratego or a DSL called DynSem.

After explaining the language aspects, this section concludes with a discussion on the other part of a language: its IDE. Spoofax provides IDE support by means of its *editor services*.

2.1.1 Syntax

The first part of the specification of a language is its syntax. The syntax of a language is often specified by means of a *lexical grammar* and a *context-free grammar*, as can be seen in the specification of, for example, Standard ML [16]. The lexical grammar is most often defined using regular expressions. It defines the individual words made up of characters, such as identifiers and numeric constants. The context-free grammar then defines syntactically valid sentences made up of words.

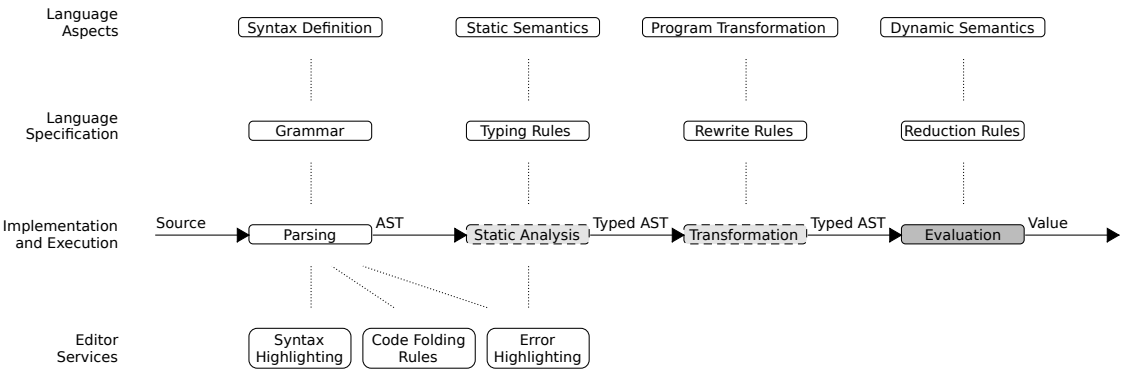


Figure 2.1: The relations between the aspects of a programming language, their specifications and their implementations. The dashed boxes in the bottom row indicate optional steps.

SDF3: syntax definition in Spoofax

To specify a syntax definition declaratively in Spoofax, a DSL called SDF3 [26] is used. SDF3 is the third generation of the *Syntax Definition Formalism* (SDF) [6]. It uses only context-free grammar productions for the specification of both the lexical syntax and the context-free syntax, a feature that was introduced in SDF2 [24].

The declarative nature of SDF3 allows for thinking in terms of the structure (the *what*), instead of in terms of parser algorithms (the *how*) as is the case with many current parser generators such as ANTLR and YACC [13]. The syntax definition is used to make parsers that parse a textual representation of a program into its AST and pretty-printers for mapping ASTs back to text. However, due to its declarative nature, SDF3 is not limited to generating parsers and pretty printers: it can also be used for error recovery rules [1], syntax highlighting rules and folding rules for editors (see figure 2.1 and section 2.1.3).

The Spoofax API gives access to the generated parser through the `SyntaxService`. The results obtained by executing the parser results in, among other information, an AST represented as an internal data structure. This data structure is briefly highlighted in the following section.

ASTs in Spoofax: Stratego terms

In Spoofax, an AST is represented as a Stratego term [11]. Terms in Spoofax comes with a textual format called the ATerm Format [22]. As an example, a term representation of the arithmetic expression $2 + 2 - 4$ is shown in listing 2.1.

```
Sub(Add(Int("2"), Int("2")), Int("4"))
```

Listing 2.1: An example ATerm representation of an arithmetic expression.

Terms can however represent any tree structure, not just ASTs. For example, a term may just as well represent a value. As such, when using Spoofax programmatically through its API, one sees terms used as an internal representation of the data going through the execution pipeline as depicted in figure 2.1.

2.1.2 Dynamic semantics

Dynamic semantics refers to how a program written in some language behaves [27]. There are many approaches to formally specify the dynamic semantics of a programming language (for an extensive treatment, see "The Formal Semantics of Programming Languages: An Introduction" [27]). One way, for example, is to specify the compilation to a different language of which the dynamic semantics are already known.

DynSem: rule-based dynamic semantics

Aside from Stratego, the Spoofax team has developed an additional method to declare the dynamic semantics of a language, namely a DSL called DynSem [23]. DynSem allows for an operational semantics specification from which a Java-based AST interpreter can be automatically derived.

DynSem uses an approach called *operational semantics*. The rest of this section discusses how DynSem (and operational semantics) is used to specify the dynamic semantics of a language in Spoofax.

Reduction rules In DynSem and operational semantics, reduction rules are used to specify the dynamic semantics of a language. The syntax of a DynSem specification is similar to the formal syntax as shown in equation (2.1), which will be discussed later in more detail. Listing 2.2 shows the syntax of a reduction rule as specified in DynSem.

```
1   Rs |- <term1> :: RWs --> <term2> :: RWs'
2   where <premise1>;
3         <premise2>.
```

Listing 2.2: The syntactic structure of a reduction rule in DynSem.

A reduction rule consists of a set of premises (lines 2 and 3) and a conclusion (line 1). Put simply, the conclusion of a reduction rule holds if all of its premises hold [10]. Thus, the rule shown in listing 2.2 is read as follows: given that the premises `premise1` and `premise2` hold, the conclusion in line 1 holds.

The left-hand side of the conclusion, the terms to the left of the arrow, represents a pattern match on terms, matching variables that can be referenced in other places of the rule. The right-hand side of the conclusion represents an instantiation of terms, and is called the result of the reduction [23].

Semantic components The variables `Rs`, `RWs` and `RWs'` shown in listing 2.2 are called *semantic components*: they represent the context in which the reduction takes place. An example of a semantic component is that of an environment, which maps variable names to their values. Another example is that of a store, which maps addresses of values to the values themselves.

There are two kinds of semantic components: *read-only* semantic components and *read-write* semantic components. An environment is a read-only semantic component: a variable is bound only within a certain scope, and any other reduction rules that are applied outside of that scope do not see this environment. Read-only semantic components therefore are only propagated downwards into the premises: the changed read-only semantic

component does not appear in the result of the premise, and therefore cannot be propagated again in the next premise. A store is a read-write semantic component: changes in the store are visible everywhere, as it represents mutable state. In contrast to a read-only semantic component, a read-write semantic component does appear in the result of a rule and can thus be threaded from one premise to the next.

Specifying function application in DynSem An example of a reduction rule specified in DynSem is shown in listing 2.3. The rule specifies the semantics of function application by using an environment. Next to it, in equation (2.1), the same rule is shown in a formal syntax, to highlight the similarity of DynSem's syntax with that of a formal one.

```
1 rules
2   App(ClosV(x, e, E), v1) --> v2
3   where
4     E  |- bindVar(x, v1) --> E';
5     E' |- e --> v2.
```

Listing 2.3: Specifying function application through the use of an environment.

$$\frac{(x \mapsto e_1) \circ E \longrightarrow E' \qquad E' \vdash e \longrightarrow v}{(\lambda x.e, E)\ e_1 \longrightarrow v}$$

Equation 2.1: The same rule in a formal specification. Here, the premises are shown on top.

The rule first introduces a pattern-match on a term representing function application on a closure (i.e. a function together with its enclosing environment `E`). In line 4, the environment `E` (a semantic component) is extended with a variable binding `x` $\mapsto$ `v1` to form a new environment `E'`. Then, in line 5, the function body `e` is reduced to the value `v2`, within the context of the newly extended environment `E'`.

2.1.3 Editor services

This section concludes with a brief description of editor services, which provide the IDE support for languages defined in Spoofax. Examples of such services include an outline view, menus in which one can bind actions to menu buttons (see figure 2.2), but also syntax highlighting, syntactic code completion and code folding rules[2].

The Spoofax API provides the editor services with similar naming. For example, the outline can be retrieved from the `OutlineService`, the syntax highlighting can be accessed through the `StylerService` and syntactic code completion is accessed with the `CompletionService`. The defined menus for a particular language can be retrieved with the `MenuService`, from which the menu actions can be retrieved and used.

Editor services are defined using a DSL called ESV, shown in the bottom window of figure 2.2. In the case of menus, their actions are specified using Stratego. Since Stratego supports native strategies, these actions can also be specified in Java. As such, Spoofax allows for defining arbitrarily complex IDE actions.

Many of these editor services such as syntax highlighting and code folding rules can be derived from the syntax definition [12] and can be further customized if needed. Taken together with the language definition, the editor services provide a language with a complete and state-of-the-art IDE experience [11].

[2]More services are listed on the Spoofax website: `https://web.archive.org/http://www.metaborg.org/spoofax/editor-services/`

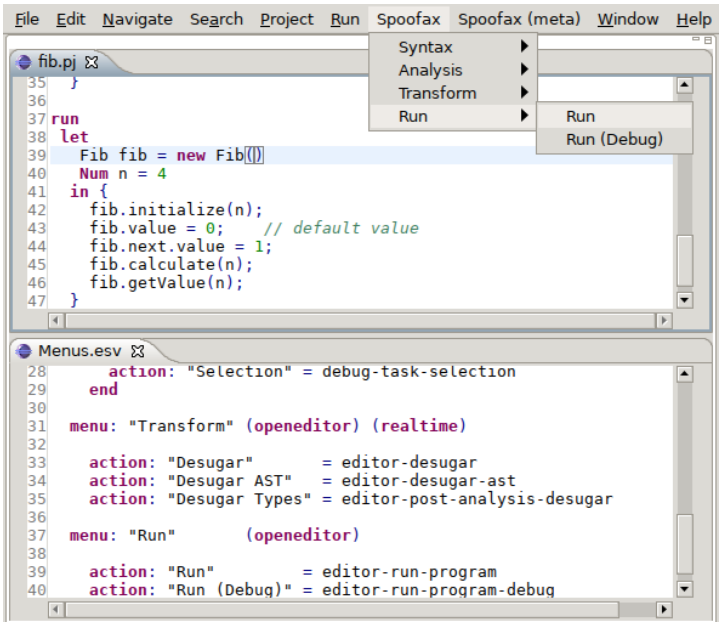


Figure 2.2: A menu action for the paplj language defined using Spoofax. The bottom window shows the menu definition, the top window shows a program written in paplj.

2.2 Read-Eval-Print Loops

Many programming languages come with an interactive environment. This interactive environment is an interface to the programming language's execution engine. One common form of such an environment is an interface in which expressions in a programming language are typed by the user, after which the results of that expression are printed back to the user. Such an environment is called a Read-Eval-Print Loop (REPL), although different names are known, including but not limited to *language shell*, *command-line interpreter* or *interactive interpreter*. In this report the term REPL is chosen, because it conveys the notion of such an interactive environment well.

2.2.1 Origin of REPLs

The Lisp programming language is one of the first programming languages offering such an interactive environment [17]. The name REPL comes from the Lisp functions that implement it:

1. The `read` function takes a user's input, which often is just one or several expressions as opposed to a complete compilation unit. It then parses this input and creates an AST.

2. The AST created in the previous step is then passed on to the `eval` function, which evaluates it.

3. The result yielded by the previous function is then printed out to the user by the `print` function.

4. After having printed the result, the environment needs to `loop` back to the read state.

Assuming the individual functions listed previously exist, a REPL can be created in a single line of code simply by combining the functions:

```
(loop (print (eval (read))))
```

Lisp has a property called "homoiconicity": Lisp's syntax is similar to its internal representation, resulting in the ability to infer a program's or data object's state simply by reading its textual representation. Syntax and AST are thus isomorphic, allowing data and code to be accessed and transformed interchangeably.

In Lisp REPLs, therefore, arbitrary data objects yielded from a previous expression can be used directly as input to the next expression. In programming languages that do not belong to the Lisp family, homoiconicity is an unusual feature. Interactive environments for these languages therefore often require additional steps to read and evaluate expressions. This is part of the semantic differences between the different names as mentioned in the introduction

2.2.2 Advantages of REPLs

REPLs provide the ability to program interactively. Programming interactively has multiple advantages.

When creating software solutions for an as of yet not well understood domain, it is often not clear which data structures and algorithms are required. In such cases, interactively developing and debugging software is an advantage over the (oftentimes much slower) edit-compile-run-debug development style. This kind of programming is called exploratory programming [2]. Related to this kind of exploration, programming interactively also provides a means for rapid prototyping and bottom up programming [4].

The explorative and interactive features of a REPL also make it an excellent tool for programmers to learn a new programming language. REPLs are also combined with what is called literate programming to offer notebooks or language playgrounds, as discussed in section 2.3.

2.2.3 Execution model

Every programming language defines an execution model, which specifies how programs written in that language are executed. Among others, it specifies what an indivisible unit of work is (a *compilation unit*) and in what order these units are executed. The implementation of an execution model is a compiler and/or an interpreter. The execution model of a REPL as shown in figure 2.3 has a couple of differences with respect to that of an interpreter or compiler.

One difference is what is considered an indivisible unit of work: a REPL might accept singular expressions that a regular interpreter or compiler might not. Usually the program is executed as a whole, without the need for smaller blocks of execution. In contrast, when using a REPL, it might be desirable to be able to execute only a single statement as one unit of execution. The AST output of the parsing step, as shown in figure 2.3, can therefore be a smaller unit than an AST found in the ordinary execution model.

Another difference between a compiler or an interpreter and a REPL is that a REPL needs to dynamically maintain an environment such that each new expression can be analyzed and evaluated within the environment of previously executed expressions. For

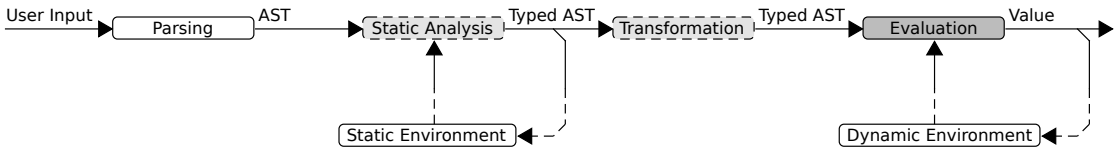


Figure 2.3: The execution model as shown in figure 2.1, adapted for a REPL. The analysis and evaluation is done within the incrementally built environment of previously executed expressions.

	Python	R	Common Lisp	Haskell (GHCi)	Swift
Execution in context	✓	✓	✓	✓	✓
Input history	✓	✓	✓	✓	✓
Automatic binding of previously yielded values	✗	✗	N/A	✗	✓
Persistent input history	✓	✓	✗	✓	✓
Multiline input editing	✓	✓	✓	✓	✓
Redefining identifiers	✓	✓	N/A	✓	✓
Error reporting	✓	✓	✓	✓	✓
Semantic code completion	✓	✗	N/A	✗	✓
Help or documentation system	✓	✓	✓	✗	✗
Additional commands to the REPL	✗	✗	✓	✓	✓
Nested REPLs to enable debugging	✗	✗	✓	✗	✗

Table 2.1: A feature comparison of several well-known REPLs.

every expression entered, it thus needs to apply the outlined steps again and update the environment it maintains in memory with the new results. This could result in a different order of evaluation than when for example an interpreter executes a program as a whole.

2.2.4 Functionality

Every REPL provided with a programming language has its own set of functionality. However, a core set of functionalities, shared between all REPL implementations, can be identified. To reach this core set of features, well-known REPLs have been investigated and their features have been compiled into a matrix as seen in table 2.1. These features are shortly discussed below.

Execution in context Expressions that are entered are executed within the context of previous executions.

Input history REPLs keep a history of inputs, such that previously entered expressions can be retrieved.

Automatic binding of previously yielded values When an expression has been evaluated, the yielded result is bound to an automatically created identifier, such that it can be

reused easily in future expressions.

Persistent history The input history as discussed previously can be recorded into a file (either per project or globally) to enable a persistent history of input.

Multiline input editing Some constructs in a programming language naturally span multiple lines. REPLs therefore provide multiline input editors that recognize incomplete code and promptly switch to a multiline environment when required.

Redefining identifiers When using a REPL in an exploratory manner, it is not uncommon to want to redefine an identifier's type or to completely reimplement a method. In this way, a REPL can be different than its host language, especially if the host language is a functional language that does not allow variables' values to change once initiated.

Error reporting A REPL typically has the same error reporting functionality as an interpreter or a compiler, meaning it prints the error message accompanied by the corresponding section of the source code.

Semantic code completion Semantic code completion is a helpful tool to provide an overview of the APIs a developer works with, adding to the explorative nature of a REPL. Note that this is a restriction of syntactic completion, which is offered by all the studied REPLs.

Help or documentation system The exploratory nature of a REPL means that one will often see new methods. Some REPLs (most notably Python's REPL with Python's docstrings) offer a documentation system, so that the developer does not have to exit the REPL to look up documentation.

Additional commands to the REPL Some REPLs offer additional commands to inspect the environment or to control their behavior. These commands are often not in the syntax of the host language and are highly diverse between REPL implementations. A notable example of a REPL offering such commands is Haskell's GHCi [21].

Nested REPLs to enable debugging A notable feature of (mostly) Lisp REPLs is that in case of an error, a new REPL is spawned inside the context of this error. This REPL then has additional commands (see the previous feature) to enable debugging and inspection of the error state. When the user has resolved the error, the nested REPL exits and the user is returned to the parent REPL. This can go to arbitrary depths.

2.3 Literate Programming

Just as with REPLs, the concept of literate programming is implemented in various forms under various names. Therefore, this section starts with an explanation of what literate programming is based on a few implementations. Afterwards, the IPython implementation of literate programming is explored in more detail.

Literate programming, as defined by Donald Knuth [14], introduces the ability to annotate source code with natural language. According to Knuth, better documentation of programs is essential to make further progress in the state of the art of programming. To achieve this he proposes to write programs not with the intention to explain the computer what to do, but with the intention to explain to humans what the programmer wants a computer to do [14] by mixing documentation and source code in a single file. This idea of literate programming was realized in its original form as the "WEB" language developed during Knuth's research at Stanford University.

Even though the idea was conceived over thirty years ago, implementations are not common. However, in recent years the idea seems to gain popularity again. A recent implementation of literate programming is Apple's Swift playgrounds [8]. Swift playgrounds are interactive documents or "notebooks" in which code is executed as it is typed, in contrast to the non-interactive style of the "WEB" language in which TEX and PASCAL were combined into one language: TEX served to document the program and PASCAL to produce a machine-executable program.

In recent years, there has also been a particular focus on reproducible research. While literate programming primarily aims to add documentation to code, reproducible research focuses on adding code to documentation. More specifically, reproducible research refers to the idea that scientific papers should be augmented with the computer code used to carry out the research [20]. Examples of recent projects claiming to support both reproducible research and literate programming include the IPython project [19] and Emacs Org-mode [20].

IPython with Jupyter notebooks

IPython, together with Jupyter notebooks, supports both reproducible research and literate programming. IPython was partially inspired by other scientific tools already offering notebook-like functionality, such as Matlab or Mathematica. Since its inception, the project has been split off into IPython, which provides an interactive REPL and a kernel that runs the user's code, and Jupyter notebooks, which provide the notebook format and web application. Like Swift playgrounds, Jupyter notebooks allow for REPL-style interactive editing; documentation and code can be edited live and blocks of code can be reevaluated, printing their updated results. Jupyter notebooks also allow for more complex graphical elements such as 3D-plots. See figure 2.4 for an example of an IPython notebook.

As explained, IPython and Jupyter notebooks have become more or less separate projects, to the extent that Jupyter notebooks can use different kernels. Nowadays there are kernels for over forty languages that can be used in these notebooks. This illustrates that in Python's case literate programming is more or less an extension to the interactive IPython REPL. Since Jupyter notebooks reuse the IPython REPL, the execution model used for Jupyter notebooks is essentially the same as it is for IPython [9].

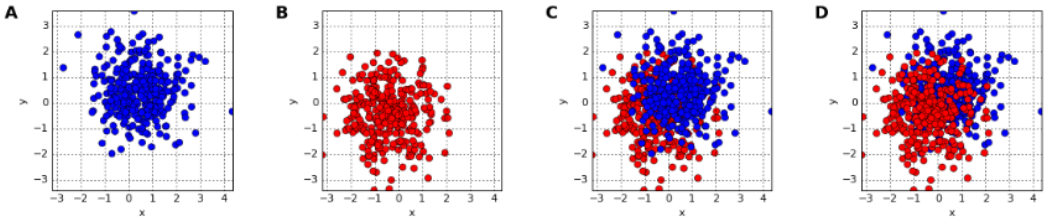
1. Overplotting¶

Let's consider plotting some 2D data points that come from two separate categories, here plotted as blue and red in **A** and **B** below. When the two categories are overlaid, the appearance of the result can be very different depending on which one is plotted first:

In [2]:
```
def blues_reds(offset=0.5,pts=300):
    blues = (np.random.normal( offset,size=pts), np.random.normal( offset,size=pts), -1*np.ones((pts)))
    reds  = (np.random.normal(-offset,size=pts), np.random.normal(-offset,size=pts),  1*np.ones((pts)))
    return hv.Points(blues, vdims=['c']), hv.Points(reds, vdims=['c'])

blues,reds = blues_reds()
blues + reds + reds*blues + blues*reds
```

Output: Out[2]:



Plots **C** and **D** shown the same distribution of points, yet they give a very different impression of which category is more common, which can lead to incorrect decisions based on this data. Of course, both are equally common in this case. The cause for this problem is simply occlusion:

In [3]:
```
hmap = hv.HoloMap({0:blues,0.000001:reds,1:blues,2:reds}, key_dimensions=['level'])
hv.Scatter3D(hmap.table(), kdims=['x','y','level'], vdims=['c'])
```

Figure 2.4: A plot from data in an IPython notebook.

3

Problem Definition and Analysis

The previous chapter provided the necessary background knowledge of the problem domain. This chapter discusses the problem itself. Section 3.1 gives the problem definition. Section 3.2 then analyzes this definition. Section 3.3 lists the requirements that are extracted from the previous sections.

3.1 Problem Definition

Spoofax offers a wide variety of tools to develop DSLs and accompanying IDE support. Precisely because Spoofax is concerned with developing DSLs, rapid prototyping of syntax and grammar is a convenient addition to the product. Section 2.2 showed that REPLs provide this rapid prototyping ability. The client has expressed their interest in a REPL that works with languages designed with Spoofax. The main problem is thus:

How can a REPL be created that can be used with languages defined with Spoofax?

3.2 Problem Analysis

This section describes the sub-problems that need to be solved while incorporating a REPL within the larger context of Spoofax.

The general theme of the sub-problems that arise from the problem defined in the previous section, is that the REPL should work with any language defined with Spoofax. Languages differ in what properties they have and how they are specified. For example, a functional programming language with no mutability is specified differently from an imperative and mutable programming language. Finding a generic way to make the REPL work for all of these languages, without any configuration from the part of the language designer, is hard, and may even be impossible.

The solution to many of the problems is to identify which parts of the problem can be solved generically, and which parts warrant configuration by the language designer. When the REPL requires configuration by the language designer, enough flexibility should be provided to do so. Each of sections 3.2.1 to 3.2.3 discusses a more concrete instance of this general problem.

3.2.1 Supporting different execution pipelines

Different Spoofax languages have different ways of executing a program. For example, some languages have an analysis step, whereas others do not. Moreover, specifying the dynamic semantics of a language can be done in multiple ways (see section 2.1.2): one can use DynSem, Stratego, and some languages even implement a Java backend[1].

Furthermore, a REPL maintains an execution environment for the duration of the REPL session, but precisely what an execution environment looks like differs among languages.

3.2.2 Language-specific commands

Another problem is when some additional command can be useful for a REPL of a particular language, but would not make any sense within the context of other languages. For example, some languages such as Python allow for loading modules, which brings all of the definitions inside of these modules into scope. An additional command to load a module could in that case be useful, but would not make any sense in languages that have no concept of definitions that can be imported.

This problem could be solved by the language designer by extending their language with reflective capabilities. However, the language designer might not want to extend the language with reflective capabilities outside of the context of a REPL. It is therefore clear that a different approach should be considered.

3.2.3 REPL-specific semantics

Sometimes a property of some language stands in the way of the rapid prototyping capabilities that are desirable for a REPL. For example, a language like Haskell does not allow for redefining a function's implementation in its normal semantics, whereas in the context of a REPL this is usually what one wants.

This poses a similar problem as in the previous section: adding REPL-specific semantics to the language should not affect the original semantics of the language. Therefore, requiring the language designer to change the original semantics of the language is an inadequate solution.

3.3 Requirements Analysis

Defining requirements upfront is important for several reasons: it is a contract between the developers and the client, it guides the product development, it enables the client to track the progress and finally it allows for validation of the deliverable. The requirements are listed in this section.

3.3.1 Design goals

The client has expressed some high-level requirements, which are listed in this section as design goals in order of priority. The design goals serve as an important guideline when defining and implementing the individual requirements. As such, they can be considered the bounds within which all requirements must fit.

[1] See IceDust: `https://github.com/MetaBorgCube/IceDust`

Language-agnostic The REPL should not make any assumptions about its host language: it must work with all languages defined with Spoofax. This also means that a language-agnostic configuration interface must be provided, in order to allow the language designer to configure and implement the language-specific parts of the REPL. Note that this does not mean that the REPL is expected to work out of the box for any language.

Minimal configuration The REPL should require only additional configuration. This means that the REPL allows the language designer to reuse existing components of their language definition in its configuration. Moreover, the language designer should not have to configure areas of the REPL that can instead be derived automatically.

Maintainability Spoofax is an existing open source project, managed by several people. When the product is delivered, these people will take over the ownership. Therefore, it is important that the code is maintainable. This means that the code should be well-documented, with low coupling and high cohesion in the code's modules.

IDE-agnostic Current stable releases of Spoofax integrate solely with Eclipse. An effort is underway to make Spoofax IDE-agnostic and to provide separate modules to tie Spoofax to IDEs. The REPL should keep this in mind from the start and not tie itself to any IDE.

Performance Spoofax's developers already focus on performance: both the generation and the use of the services are performant. This should be no different for the REPL.

Modify Spoofax's existing codebase as little as possible The product should be an extension to Spoofax, which means the changes made to the existing Spoofax codebase should be as small as possible. Preferably, the REPL should be a standalone module.

3.3.2 Requirements

Under the guidance of the design goals listed in the previous section, the requirements compiled from the feature matrix discussed in section 2.2 and meetings with the client are discussed below using the MoSCoW method.

Must have

Requirements listed under "must have" are of critical importance to the usability and success of the deliverable. Without these, the product is not in a workable state and is not likely to be accepted by the client. *Must* can also be considered an acronym for the Minimum Usable SubseT.

Interactive REPL Per the definition of a REPL given in section 2.2, the REPL has to be interactive. It should evaluate single statements and expressions typed in by the user and print the results back to them.

Works with any language defined in Spoofax Every service in Spoofax operates from language definitions. It is evident that the REPL should work with these definitions if it is to fit within Spoofax. As for the parts of the REPL that do require configuration, the REPL should provide a flexible and generic interface for that.

Input history Users should be able to retrieve previously typed expressions and statements to support the explorative and interactive nature of a REPL.

Automatic binding of previously yielded values In the same vein, previously yielded results should be implicitly bound to automatically generated identifiers to make their values available in future expressions.

Multiline input editing Multiline input editing is a crucial feature for user satisfaction. The user should be able to enter expressions spanning multiple lines.

Error reporting To support the interactivity of the REPL, error reporting should be available in two ways: reporting errors while typing an expression and reporting errors after entering the expression. While typing a statement, on-the-fly error reporting should indicate wrongly typed parts of the statement. After entering the statement, any errors that occurred in the execution pipeline should be displayed, whether they are parse errors, errors found during static analysis or evaluation errors.

Syntax checked expressions Supporting the above requirement, all input should have its syntax checked on the fly.

Syntax highlighting All expressions and statements (whether they are currently being entered, displayed as previously entered input or displayed as previously yield results) should be syntax highlighted.

Integration with Eclipse As a first implementation, the REPL should integrate with Eclipse to provide an interface to the users.

Should have

Requirements listed under “should-have” are important, but not required for a working product.

Ability to redefine identifiers As explained in section 2.2.4, REPLs provide the ability to explore unknown problem domains. It is not a far-fetched idea that a developer would want to change function implementations or the types of certain variables. To support this, a REPL should allow users to redefine identifiers. This might mean that the REPL needs different semantics than the language it operates on, contrasting the design goal that the REPL should be language-agnostic.

Environment inspection The exploratory and interactive nature of REPLs calls for the ability to inspect the current environment. This is to replace the files with source code that a developer could otherwise inspect and an initial step towards offering debugging features in the REPL.

Save and load REPL state Often, developers want to save the current state of their IDE and return to it later. As such, the REPL should allow their state to be saved and restored.

Could have

Requirements listed under "could-have" are desirable, but not necessary. These requirements often improve usability or customer satisfaction and are included only if time permits.

Code completion The multiline input editor should ideally function just like an IDE's editor. Semantic code completion is not yet implemented in Spoofax, so the REPL should provide syntactic code completion instead.

Hover over variables to see value, type and others Another step towards debugging would be the ability to hover variables with the mouse in order to inspect their value, type and other known information. The difference between this feature and the previously mentioned environment inspection is that this feature works per variable.

Literate programming As explained in section 2.3, literate programming offers the advantage that code and documentation go hand in hand. This allows developers of languages in Spoofax to document and illustrate their language simultaneously with the development: documentation and examples can never be outdated, because outdated example code will halt the execution.

Integration with other IDEs (IntelliJ) Generating Spoofax's editor services for IntelliJ is currently a work in progress and it would be nice if the REPL works in IntelliJ from the start.

Won't have

Requirements listed under "won't-have" are not to be implemented during this project. They have been identified as possible features, but are outside of the scope of this project and listed only as possible suggestions for further work.

GDB-style debugging and nested REPLs Spoofax currently does not generate any debugging features, which would be required for the REPL to offer such functionality as a built-in feature. Writing a debugger is outside the scope of this project and thus offering debugging capabilities inside the REPL is something to consider for a later version.

3.3.3 Minimal viable product

The design goals and requirements listed in the previous sections give a good idea of what the deliverable should look like. It is possible, however, that due to unforeseen problems, not all the listed requirements and design goals can be met. It is important to therefore define a minimal subset of the design goals and requirements that *must* be present in the final deliverable: the "must have" requirements have to be implemented, whilst adhering to the first two design goals.

4

Design

As detailed in the previous chapters, the Spoofax Language Workbench already offers a wide variety of services to assist in developing new domain specific languages. However, not all exposed functionality can be used easily from within the context of a REPL, where a user wants to be able to type in some expressions and see their results.

An example is the various transformation goals made available to the user by a language designer. There is no such thing as a uniform "evaluation command" that works across all languages. Instead, each language defines its own transform goals, of which one is an evaluation goal. Furthermore, the sequence of processing steps needed to go from source to parsed source to a transformed result also varies between languages. One of the main goals that the design had to accomplish, therefore, was to expose a uniform interface for the various transformation stages to REPL frontends.

Section 4.1 gives an overview of the product design and its components. Afterwards, the individual components are explained in more depth. Section 4.2 addresses the presentation of a uniform interface to frontends by encapsulating operations in commands. Section 4.3 details how commands have been decomposed into smaller functions, which allows frontends to build their own transformation pipeline. Section 4.4 explains how transformation results are returned to the REPL frontends. Section 4.5 discusses how the strategy pattern is applied to allow future support for other evaluation backends, such as Stratego.

4.1 Overview

As stated in section 3.3.1, one of the design goals during the project was to keep the implementation of the REPL IDE-agnostic. To achieve this goal, the product has been split into a backend and several frontends. The backend interacts with the Spoofax services on behalf of the frontends. An overview of its design can be found in figure 4.1. A diagram of the complete design can be found in appendix D.

To request and receive results from the backend, a frontend can invoke commands by sending the command name and its parameters to the `ICommandInvoker` in the backend. The `ICommandInvoker` will then execute the `IReplCommand` corresponding to the command name. Every executed command returns an `IResult`.

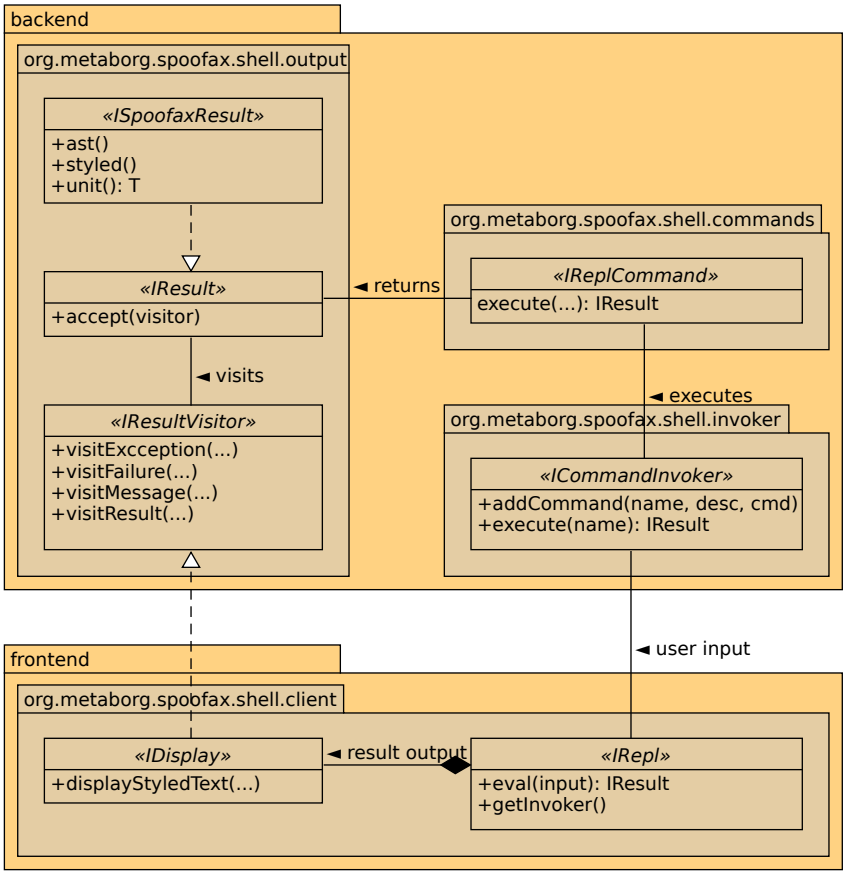


Figure 4.1: An overview of the most relevant components of the backend.

To execute a command, the frontend thus only has to implement a way of querying the user for input, after which it can send this input to the `ICommandInvoker`. Since execution of different commands can have varying types of results, the actual returned type of `IResult` can vary. The returned `IResult` can be visited by the frontend, thereby dispatching results based on their specific types.

4.2 REPL Commands

To separate the implementation of user operations into logical units, each user operation is encapsulated within a class implementing the `IReplCommand` interface. See figure 4.2 for the design of the `commands` package. After acquiring an invoker instance, the frontend has two commands available by default:

:load `LanguageCommand`, loads a language from a file path;

:help `HelpCommand`, prints descriptions of all available commands.

Frontends can define additional commands by implementing the `IReplCommand` interface.

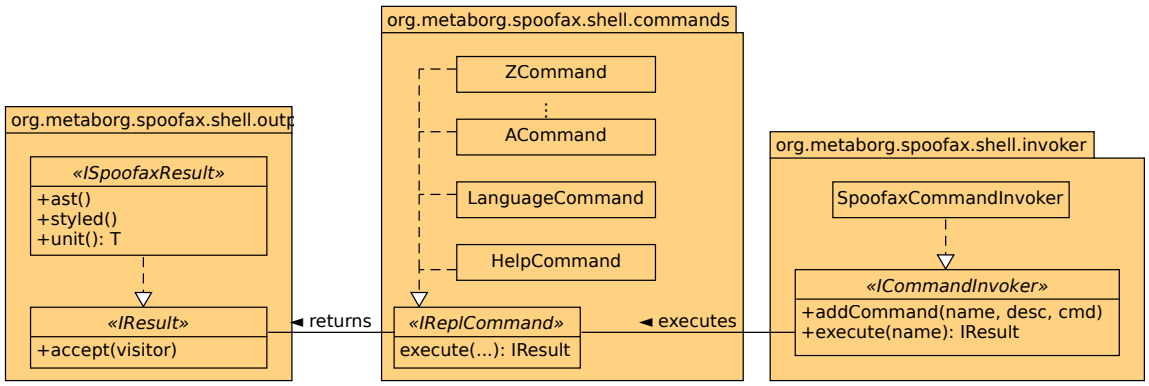


Figure 4.2: UML of the commands frontends can execute and the corresponding result interfaces.

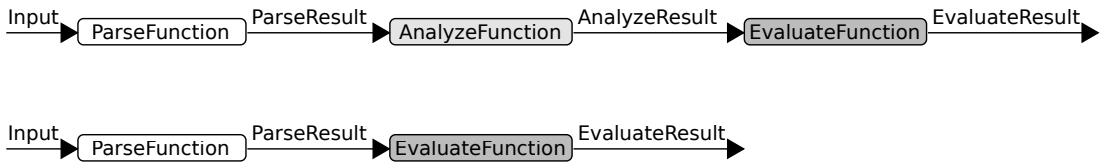


Figure 4.3: Example composition of execution steps. The top shows the execution model of a language with static analysis, the bottom shows one without static analysis.

As explained in the introduction of this chapter, there are only a few commands that work across all languages, since every language can define its own transformations or evaluation strategy. Therefore, the `LanguageCommand` tries to determine several properties of the loaded language and adjusts the set of loaded commands accordingly.

By splitting operations in small logical units the design ensures command classes adhere to the Single Responsibility Principle, which makes all REPL commands less complex and therefore easier to maintain. The logical units that make up the commands also ensure a flexible design that can be modified as the Spoofax architecture develops further on.

4.3 Function Composition

There was a need for a more fine-grained and flexible way of defining commands for those that deal with the execution model as described in chapter 2. This is because the specific execution steps that are needed for the evaluation of a program in one language can differ from those of other languages. For example, some languages have no analysis stage, whereas some do. The `LanguageCommand`, as explained in the previous section, should thus create different commands depending on whether the loaded language has an analysis step. See figure 4.3 for a schematic example.

To accommodate for this, commands dealing with the execution model are built by composing functions using a `CommandBuilder`, a class following the builder pattern. In this context, a function represents a step in the execution model, whereas the composition represents the execution model as a whole. The builder has methods for composing

functions, and a method for building a command like the one described in the previous section out of the composition. A UML diagram detailing this design is shown in figure 4.4.

This design decision brings several benefits:

1. It is a direct mapping from the problem domain: functions represent execution steps and their composition represents the execution model.
2. The functions are reusable, since each function only assumes the types of its input and output.
3. It allows for composing and defining commands at runtime.

One problem remains in this design, which is how to handle failures that occur somewhere in the execution pipeline. For example, analysis should not be done when parsing has failed. Using Java's exceptions to solve this problem is not desirable, as one would like to treat failures as any other result of an execution step. The solution to this problem is to apply monadic composition, as opposed to regular function composition.

4.3.1 Monadic composition

The function composition model takes inspiration from the `Either` data type that is known to many functional languages such as Haskell. It is a data type that represents two alternatives, for example *either* a success or a failure. As any step in the execution model of the REPL can result in either a failure (for example a syntax error) or a success, this data type is a good candidate for solving the problem of intermediate errors.

The `Either` type is used as follows: Any individual function in the execution model returns an `Either` type, representing the alternatives of success or failure. When a function is composed with another function, the latter function is only executed when the former function returns an `Either` representing a successful result. Otherwise, the latter function is not executed at all and the same `Either` is returned instead, thereby propagating the cause of the failure to the end of the pipeline whilst discarding any other functions further down this execution pipeline. This kind of function composition is an example of what is known as *monadic composition*[1].

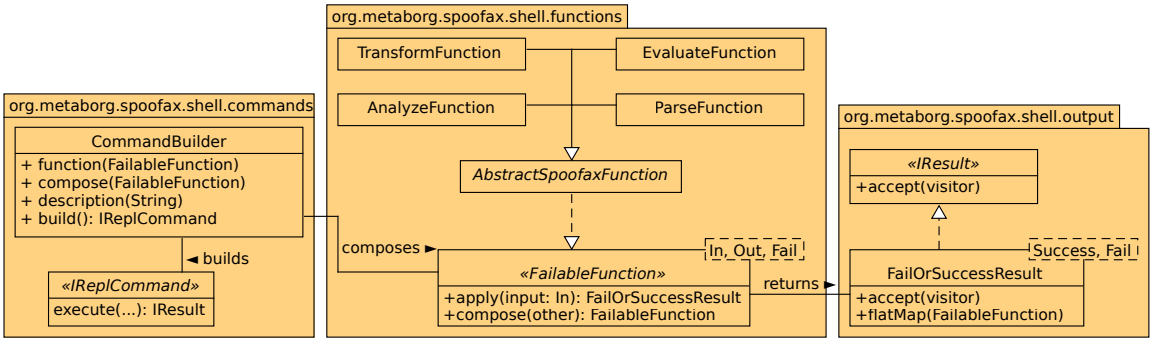


Figure 4.4: UML of the `CommandBuilder` and function composition interfaces.

[1]Monads are a more abstract concept than just this specific case. In fact, the `Either` data type is just one type of monad.

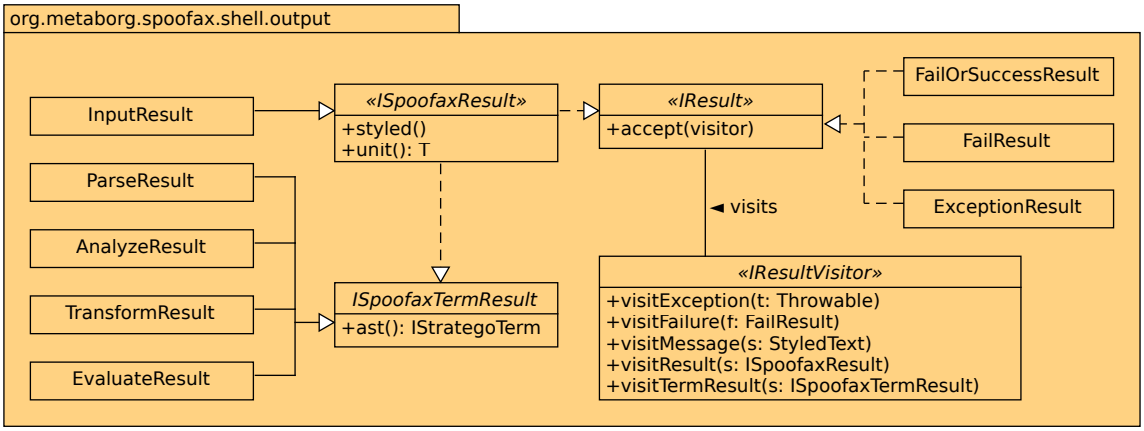


Figure 4.5: UML of the concrete results and the result visitor.

The `Either` data type has been adopted in the design of the REPL as a class named `FailOrSuccessResult`. The class is more specific to the domain of the execution model of the REPL. It implements the `IResult` interface just like any other result class, thereby allowing it to be treated in the same way generically. The functions representing the execution steps bear the name `FailableFunction`, and contain a method for composing it with another `FailableFunction`.

4.4 Returning Results to the Frontend

As mentioned in section 4.1, every command implementing the `IReplCommand` interface returns an instance of a class implementing the `IResult` interface. It is up to the frontend to determine how to handle the result. Since commands can result in a failure or an exception (see section 4.3), it is not always clear what kind of `IResult` the command returns.

To illustrate this, it is possible that a user tries to evaluate an input that passes the parsing step but fails with an exception at the analyze step (see figure 4.3). The expected result is an `EvaluateResult`, but because an exception was thrown the corresponding wrapped `AnalyzeUnit` cannot be created. The command that did the evaluation then returns a `FailOrSuccessResult` containing the corresponding `ExceptionResult` that caused the failure. However, the caller of the command has no way of telling that the returned `IResult` is in fact a `FailOrSuccessResult`.

The visitor pattern has been adopted to solve this problem. As can be seen in figure 4.5, every `IResult` can be "visited" by classes implementing the `IResultVisitor` interface. Each subclass of `IResult` decides what visitor method is called, and the `IResultVisitor` gets a concrete subclass of `IResult` as argument by means of double-dispatch. Furthermore, the `FailOrSuccessResult` delegates the visitor to its wrapped `IResult`.

Implementations of the `IResultVisitor` can then decide per result type how to handle the returned information. Since all commands can always return a result with the original exception or the failed `ISpoofaxResult` by returning a `FailOrSuccessResult`, the frontend is always provided with error messages or the results of earlier processing steps.

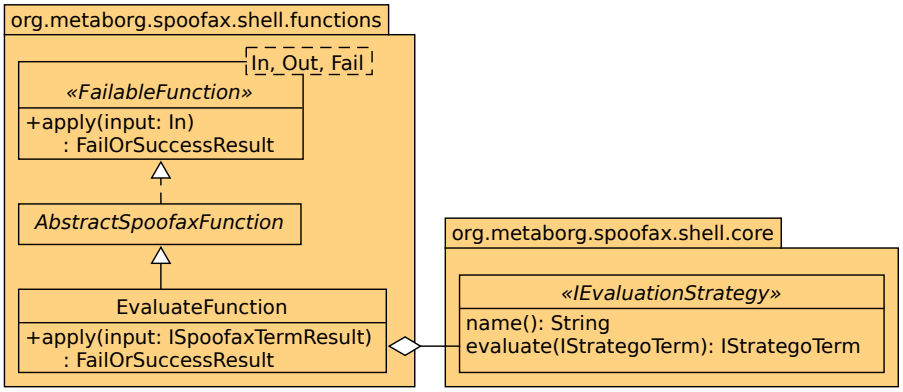


Figure 4.6: UML of the `IEvaluationStrategy` interface and its collaborators.

The frontend has the responsibility to decide how to present the returned data to the user, resulting in a lot of flexibility. However, this flexibility is not always required. It is not necessary, for example, for a text-based console REPL, although it can be used to easily highlight errors or print stack traces. When integrating the REPL in multithreaded graphical user interface (GUI) toolkits, however, this flexibility is useful.

4.5 Evaluation Strategies

As explained in section 3.2.1, there are multiple ways for specifying the dynamic semantics of a language. Interpretation can for example be performed through an interpreter generated from a DynSem specification, or by implementing the interpreter in Stratego or Java. However, ultimately all of these methods do the same thing. That is, they accept an AST as input and somehow transform it into an output. In Spoofax, both the AST input and the output value can be represented as a Stratego term. This is due to the fact that the format for terms in Spoofax supports all of the basic types that Java has: integers, real numbers, strings, lists and named constructor applications for encoding objects [22]. It is only the internal representation of an interpreter's evaluation context and the implementation of the interpreter that differs.

The strategy pattern is a pattern for encapsulating computation tasks that have the same types of input and output, but a different implementation. This is precisely the case for the different methods of evaluation: the method or "strategy" of evaluation is therefore put behind an interface called `IEvaluationStrategy`. Figure 4.6 shows a UML diagram of this part of the design.

An implementer of the `IEvaluationStrategy` is responsible for performing evaluation on an AST representation of a program within the evaluation context that it maintains throughout successive invocations. In this way, maintaining the evaluation context is encapsulated, and no assumptions are made of what the evaluation context looks like. This is a desirable property since the representation of an evaluation context varies across languages and evaluation methods.

5 Implementation

The previous chapter described the design of the product. During the implementation of this design, issues surfaced that had not been foreseen. This chapter discusses these issues and the changes made to the design to resolve them. Section 5.1 illustrates the implementation of the evaluation strategy for DynSem. Section 5.2 explains the extensions that have been made to Spoofax's editor services to enable the language designer to configure and extend the REPL for their language. Section 5.3 highlights the use of dependency injection via the Guice framework. Finally, Section 5.4 discusses the implementation of the console and Eclipse frontends.

5.1 DynSem Evaluation Backend

An implementation for the `IEvaluationStrategy` as discussed in section 4.5 is made for languages that specify their dynamic semantics using DynSem. The implementation class, called `DynSemEvaluationStrategy` (see figure 5.2), evaluates REPL-specific DynSem rules and maintains the environment in which evaluation takes place.

This section first goes over the configuration interface for DynSem that is provided by the REPL. After that, the implementation of DynSem and of the `DynSemEvaluationStrategy` are discussed in more detail.

5.1.1 The configuration interface

To evaluate a program in their language within the REPL, the language designer has to make two kinds of configurations for the REPL in their DynSem specification. The first is a rule for initializing the execution environment for the REPL, shown in listing 5.1. The second are the rules for implementing the REPL-specific semantics as discussed in section 3.2.3.

Environment initialization The initialization rule shown in listing 5.1 is evaluated upon the initialization of the evaluation strategy. It instantiates the semantic components that form the execution environment for the REPL: an environment with an initial variable binding of $x \mapsto 4$, and an empty store. The evaluation strategy uses this in its successive evaluations, and updates the execution environment after each result.

```
1 rules
2   // Initialization of shell state: an environment with "x" bound to 4,
3   // and an empty store.
4   ShellInit() -init-> ShellInit() :: Env { "x" |--> NumV(4) }, Store {}.
```

Listing 5.1: The initialization rule for the semantic components.

REPL-specific semantics The second kind of configuration are the rules for the REPL-specific semantics. These can be seen as entry points for the REPL to the interpreter. The rules are all named "shell", so that they are distinct of the ordinary semantics. Listing 5.2 shows an example of such a rule. The rule implements a construct that is only valid when evaluating within the REPL: it implements binding the result of an expression to a variable. With the specification of this rule, the bound variable can be used in successive evaluations done by the user.

Note that the environment *E* is passed as a read-write component, instead of a read-only component as is explained in section 2.1.2. This is because in this case the environment *should* be writable, since the resulting environment after execution should be available to the REPL. Note also that in line 5 of the rule, the rest of the specification is recursively invoked. This shows that much of the existing specification can be reused when implementing the REPL-specific semantics.

```
1 rules
2   // let x = 2
3   Let(x, e) :: E -shell-> v :: E'
4   where
5     E |- e :: Store {} --> v :: Store _;
6     E |- bindVar(x, v) --> E'.
```

Listing 5.2: A rule specifying semantics specific to the REPL.

5.1.2 The implementation

This section discusses the implementation of the evaluation strategy. However, before doing this, an explanation of the interfaces provided by DynSem and its generated interpreters is in order.

DynSem's generated interpreters As explained in section 2.1.2, DynSem is able to generate interpreters in Java from a dynamic semantics specification written in the DynSem meta-language. DynSem uses the Truffle language implementation framework for its generated interpreters [7]. By using the Truffle framework, the generated interpreter can benefit from the performance optimizations of the Truffle virtual machine (VM) [28], as opposed to writing one's own optimizations instead.

The evaluation of a program is done by invoking a reduction rule with as its arguments the program in the form of an AST, and the semantic components. A reduction rule can be found by performing a lookup with the rule's signature as the lookup key, through Truffle's foreign object access interface [5]. That is, the foreign objects are precisely the rules such as the one shown in listing 5.2 (in the form of an internal representation).

For this reason, the interpreter generated by DynSem can be seen as a *meta-interpreter*: it interprets a DynSem specification, and allows access to rules as value objects through Truffle's foreign object access interface. The rules themselves (or the "values" interpreted by the meta-interpreter) can then be seen as access points to the interpreter of the language, as they can be invoked with an AST and semantic components. The relation between a meta-interpreter and an ordinary interpreter is captured in the diagram shown in figure 5.1.

The implementation of the evaluation strategy The UML diagram of the evaluation strategy is shown in figure 5.2. The `DynSemEvaluationStrategy` uses an `IInterpreterLoader` to load the interpreter as generated by DynSem. It initializes the Truffle VM, called the `PolyglotEngine`, with the interpreter generated by DynSem, by evaluating the DynSem specification through the `eval` method provided by the `PolyglotEngine`.

The `PolyglotEngine` provides the `findGlobalSymbol` method, allowing the evaluation strategy to lookup the reduction rules of the interpreter represented as a `Value` object. The returned `Value` object has an `execute` method, which provides the interface for invoking the reduction rule with the program AST and the semantic components. This way, the configuration interface as outlined in section 5.1.1 can be implemented: the initialization rule and the REPL-specific rules can simply be looked up by their rule signature.

5.2 ESV Extensions

REPL-specific extensions have been made available to the ESV configuration language (see section 2.1.3) to configure language-specific features. Currently, the backend used to specify the dynamic semantics of a language (see section 4.5) and the start symbols valid within the context of a REPL can be configured. The newly added configuration settings are illustrated in listing 5.3. They are available through Spoofax's API as a `ShellFacet`, after loading a language that defines these additional settings.

```
module editor/SL-Shell

shell
    evaluation method : "dynsem"
    shell start symbol : Expr
```

Listing 5.3: Configuring language specific settings.

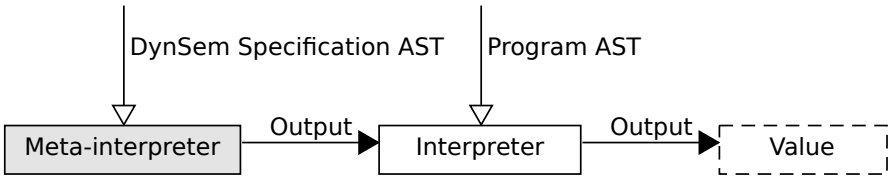


Figure 5.1: The relation between a meta-interpreter and an ordinary interpreter.

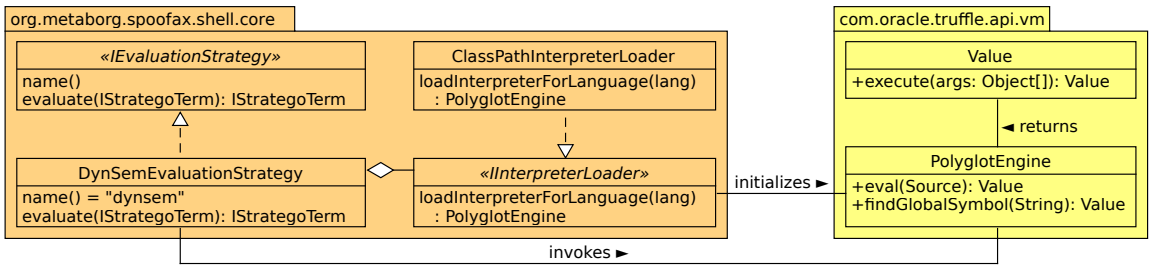


Figure 5.2: UML diagram of the `DynSemEvaluationStrategy` and its collaborator `IInterpreterLoader`.

This ESV configuration can be extended in the future to allow the specification of language-specific additional commands, as discussed in section 3.2.2.

5.3 Dependency Injection

The design of Spoofax makes extensive use of dependency injection, specifically by using the Guice framework[1]. To achieve as much interoperability as possible, the final product uses dependency injection in the same manner as Spoofax. Guice had a substantial impact on the design and implementation of the final product, since it nearly eliminates the need to create factories for complex datatypes.

Guice requires the programmer to bind implementations of dependencies in its module classes, thereby separating behavior and dependency resolution. Complex datatypes can then accept their dependencies as constructor arguments, allowing Guice to resolve and inject an instance of a bound type.

In the final product dependency injection is also used to supply classes with a default configuration. An example is the default list of commands available to a user as described in section 4.2. The list of commands is created from a Guice module by instantiating a `MapBinder` with predefined bound commands. Child modules can then append extra entries to the `MapBinder`, which makes them directly available to the existing architecture.

Using dependency injection has made it easier to create a modular product, centered around smaller interfaces interacting with each other. All these interfaces can easily be bound to new implementations, thereby extending the functionality of the REPL.

5.4 Frontends

Chapter 4 showed that the backend places few restrictions on the frontends. A good example of this fact is the visitor pattern (see section 4.4), which allows the frontends to schedule the processing of the results and to display them in many different ways. To illustrate this flexibility, two frontends have been developed: a single threaded, text-based console frontend and a multithreaded, graphical plugin to the Eclipse IDE. Both of these are discussed next.

[1] `https://github.com/google/guice/wiki/Motivation`

5.4.1 Console frontend

The console frontend provides the user with a text-based user interface. For users already accustomed to the command line, this is perhaps the most familiar looking interface, see figure 5.3.

```
Spoofax REPL, version 0.0.4

Loaded language impl. org.metaborg:simpl:0.1.0-SNAPSHOT

[In ]: let x = 40
[...]:
NumV_1(40)@7db534f2

[In ]: let addToX = a -> a + x
[...]:
ClosV_3(a,Plus_2(Var_1(a)@2de366bb,Var_1(x)@3f093abe)@61a002b1,{"x" NumV_1(40)@7db534f2})@23a9ba52

[In ]: addToX(2)
[...]:
NumV_1(42)@4108fa66

[In ]: addToX(2
[...]:
addToX(2
Syntax error, expected: ')'

[In ]: :exit
```

Figure 5.3: The console frontend.

A REPL running in the console requires a means of getting user input from the standard input stream and printing results and error messages to the standard output and error streams.

A BSD-licensed library that features these requirements is JLine2. JLine2 is a library for handling console input, similar to GNU readline and BSD editline. Most of the editing features present in these two libraries are also present in JLine2.

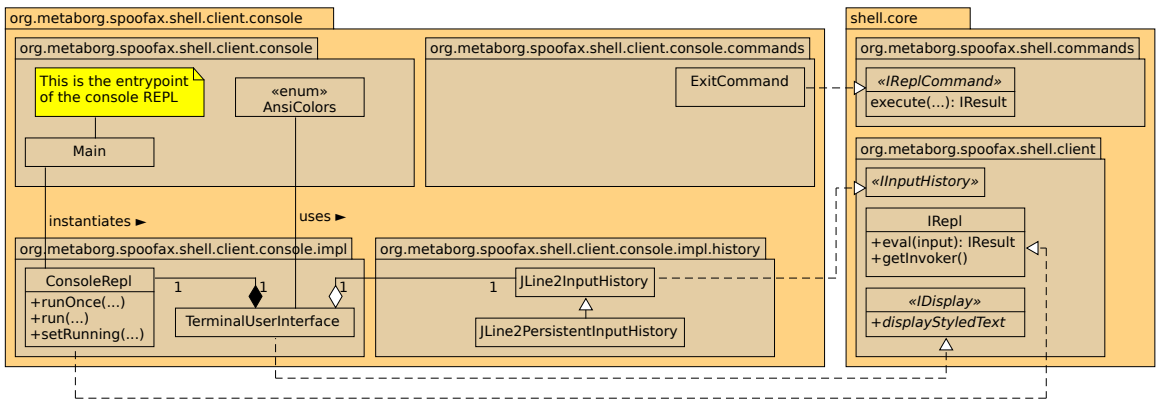


Figure 5.4: UML diagram for the console frontend.

As can be seen in figure 5.4, a frontend has to implement only a few interfaces. The `ConsoleRepl` class is the central element in the console frontend. It is instantiated first

and controls the lifecycle of this frontend. Its `run` method maps to the `loop` step in Read-Eval-Print Loop and thus contains the `read`, `eval` and `print` steps.

The `read` and `print` steps are provided the by `TerminalUserInterface` class, which, as the name implies, is responsible for the user interface. This class has connections to the standard input, output and error streams. It uses JLine2 to provide a blocking input editor. Because the input editor is blocking, the console frontend can run in a single thread, thereby greatly simplifying its implementation.

A straightforward algorithm to provide multiline editing capabilities is used in combination with JLine2's input buffer. JLine2 does support input history, which means only an adapter class, `JLine2InputHistory`, had to be made to interoperate with the history interface that is offered by the backend. An extension to this class is provided to maintain persistent history.

To provide syntax and error highlighting, ANSI color codes are used. These color codes are obtained through the `AnsiColors` enumeration, which maps colors as returned by the `IResult` interface to their closest ANSI equivalent.

Lastly, an `ExitCommand` is provided to halt the `run` method in `ConsoleRepl`.

5.4.2 Eclipse plugin

Spoofax currently features extensive integration in the Eclipse IDE. The client expressed their interest to have this same level of integration for the REPL. A plugin to the Eclipse IDE has been developed, which exposes the REPL functionality through a graphical user interface that is familiar to anyone working in Eclipse, see figure 5.5.

As is customary for a graphical user interface toolkit, the toolkit offered by Eclipse uses multiple threads. One such thread is designated the "UI thread", or user interface thread. This thread is responsible for processing user-generated events (such as mouse clicks) and updating the graphical representation of the widgets. All tasks that perform long running calculations are supposed to be run in a background thread, such that the UI thread is free to process incoming events. Instead, if a long running computation is run in the UI thread, the widgets on the screen stop responding to the user and the program appears to be in a frozen state. For this reason, the backend has to perform its evaluations in a background thread, whilst all the widgets and user interaction takes place in the UI thread. The following discussion refers to the UML as depicted in figure 5.6.

Again, as can be seen in the UML, only a few interfaces have to be implemented. The `EclipseRepl` class is the central element in the Eclipse plugin. It is instantiated by the `ReplView` class, which in turn is instantiated by Eclipse when the user requests the REPL to be opened. This `ReplView` class also instantiates the input editor widget, `EclipseEditor` and the `IDisplay` implementation, `EclipseDisplay`. In a multithreaded graphical user interface environment, asking text entry widgets for the entered text is rarely a blocking process. This is no different in Eclipse. For this reason, the `EclipseRepl` cannot simply loop as the console frontend does. The solution is to use reactive programming through RxJava: the `EclipseRepl` is registered as an observer to the `EclipseEditor`. When the user presses the Return key, `EclipseEditor` notifies `EclipseRepl` with the contents of its text buffer. `EclipseRepl` in turn launches a background job in which the evaluation takes place, so as to not block the UI thread. Since the backend simply returns an implementation of the `IResult` interface (see section 4.4), the `EclipseRepl` can schedule the processing of this result by the `EclipseDisplay` class on the UI thread. This processing

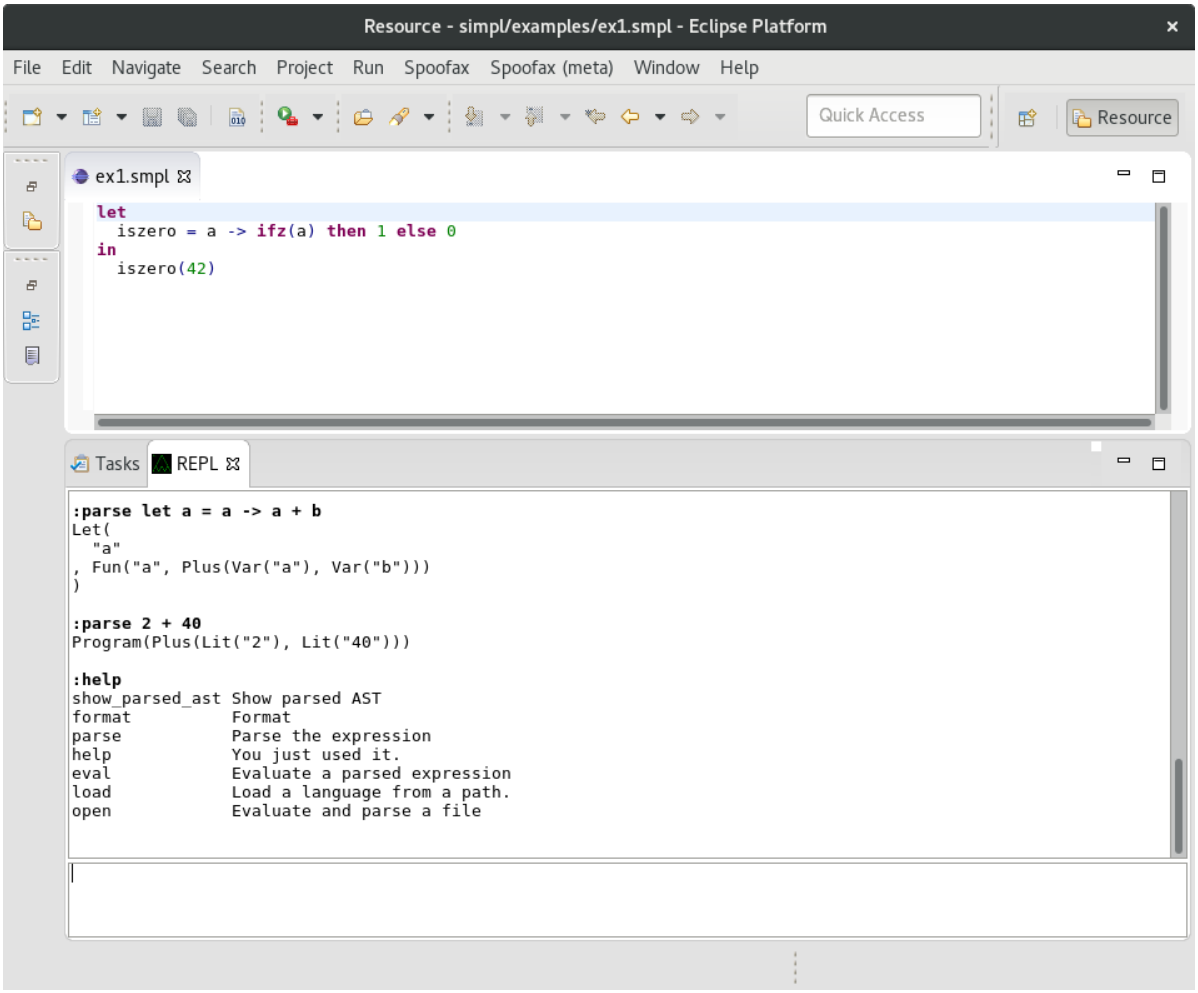


Figure 5.5: The Eclipse plugin.

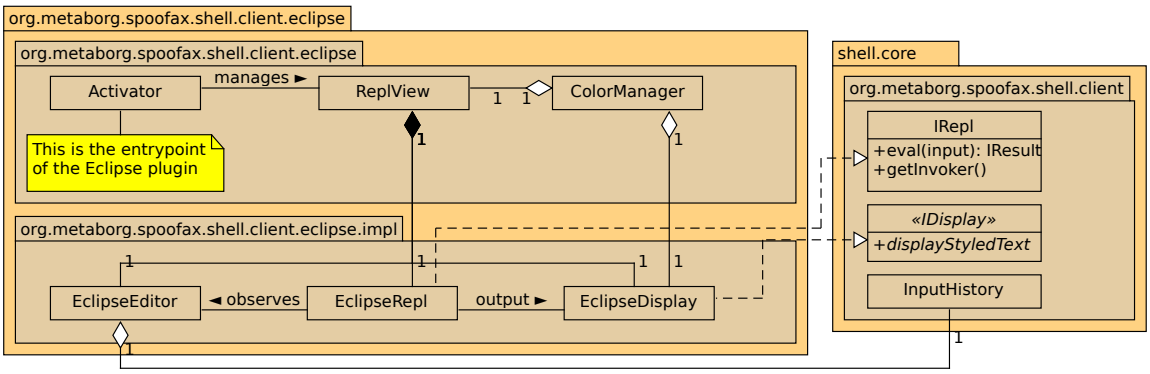


Figure 5.6: UML diagram for the Eclipse plugin.

has to be done on the UI thread, because otherwise the `EclipseDisplay` cannot update the widget it maintains to display results. As one can see, the Eclipse plugin does not implement the `loop` step of a REPL. There is no need to do so, due to the use of multithreading and reactive programming.

6 Evaluation

Two evaluations are made in this chapter. The first section of this chapter, section 6.1, evaluates the product. This is done firstly by looking at the requirements and secondly by looking at the design goals, both as described in section 3.3. The end of this section concludes with an evaluation of the product as a whole in section 6.1.3. The second section of this chapter, section 6.2, evaluates the process by which the product came to be. It discusses the development methodologies used and the feedback as received by the Software Improvement Group.

6.1 Product Evaluation

Section 3.3 describes the requirements and the design goals that were agreed upon with the client at the start of the project. This section evaluates how these are met by the product. Section 6.1.1 discusses each requirement and decides whether or not it has been met and how. Section 6.1.2 does the same for the design goals. Finally, this section concludes with a short discussion to see if the minimal viable product as defined in section 3.3.3 has been delivered.

6.1.1 Evaluation of the requirements

This section discusses all the requirements as set out in section 3.3.2. All the “must have”, “should have” and “could have” requirements are listed in table 6.1. Per requirement, the table lists whether it has been met, including an explanation as to how it has been met or why it has not been met.

6.1.2 Evaluation of the design goals

In this section the design goals as set out in section 3.3.1 are discussed. Each design goal is listed below, along with an explanation as to how it has been met.

Language-agnostic This design goal stated that the REPL should work for any language. Additionally, the language designer should be provided with a language-agnostic interface to configure and implement the language-specific parts of the REPL.

Requirement	Met?	Explanation
Interactive REPL	Yes	Two interactive REPLs have been created: one for the console and another in Eclipse.
Works with any language defined in Spoofax	Yes	The backend retrieves the information it needs from Spoofax's services. A language developer is able to define REPL-specific configuration in an ESV file, as well.
Input history	Yes	An interface with a corresponding frontend-agnostic implementation has been created in the backend, allowing frontends to either use the implementation or implement the interface as they see fit. Both the console and the Eclipse frontend provide input history.
Automatic binding of previously yielded values	Yes	The language designer can specify these semantics in DynSem.
Multiline input editing	Yes	The way in which input is given is not enforced by the backend. However, both the console and the Eclipse frontend implement multiline input editing.
Error reporting	Yes	Both the console and the Eclipse frontend print errors to the user. The characters to which these errors belong are highlighted in red. Error reporting is not yet enabled during typing.
Syntax-checked expressions	Yes	The backend parses the input with the services provided by Spoofax. As such, syntax-checked input was given for free. However, this feedback is not yet enabled during typing.
Syntax highlighting	No	A working prototype has been implemented. Due to the nature of console-based user interfaces, the console frontend cannot support this feature. The Eclipse frontend has most of the required functionality.
Integration with Eclipse	Yes	A REPL has been implemented in Eclipse, allowing the user to give input and see output.
Ability to redefine identifiers	Yes	The language designer can specify these semantics in DynSem.
Environment inspection	No	This is impossible to do in the current DynSem implementation, because the environment is captured in an object of type `Object` and is therefore unprintable.
Save and load REPL state	No	Due to limitations in time, this feature has not been implemented.
Syntactic code completion	No	Spoofax's code completion services do not yet work with all languages and are in process of a rewrite. As such, code completion has not been attempted.
Hover over variables to see value, type and others	No	Due to limitations in time and the fact that the environment is captured in an object of type `Object`, this feature has not been implemented.
Literate programming	No	An attempt has been made at writing an IPython frontend for the REPL backend. However, due to limitations in time and the pressure of finishing more important features, this work has not been finished.
Integration with other IDEs (IntelliJ)	No	The Eclipse frontend is not entirely done yet, which was a requirement before starting on implementing a frontend for IntelliJ. As such, this feature has not been attempted.

Table 6.1: Evaluation of the requirements, separated per MoSCoW category.

An extension of Spoofax's editor services has been created, allowing language designers to configure certain parts of the REPL. Additionally, language-specific changes, such as the ability to redefine identifiers, can be made through DynSem. Currently, however, only DynSem-based interpreters are supported, as opposed to Java- and Stratego-based interpreters. Despite this, care has been taken to make this extendable. This design goal, therefore, is met.

Minimal configuration This design goal stated that the REPL should require only additive configuration, by reusing existing components of the language definitions and automatically deriving other settings.

The extension of the editor services discussed in the previous design goal is only additive. Three settings can be configured at the time of writing, all of which have sane default settings. Furthermore, because the existing DynSem specification can be reused, only new DynSem rules have to be written to specify REPL-specific semantics. Finally, the SDF3 configuration is reused to specify which start symbols are valid in case the language designer wants to allow different start symbols. Therefore this design goal is met.

Maintainability This design goal stated that the code should be maintainable, because ownership will be transferred to the people already working on Spoofax.

Various static analysis tools, up-to-date documentation, proper software design and a received 4,5 out of 5 stars during the first round of feedback from the Software Improvement Group (see section 6.2.2 below), are good indications that this design goal is met.

IDE-agnostic This design goal stated that the REPL should place no assumptions on the environment it runs in, since an effort is underway to do the same for Spoofax itself.

The backend is agnostic to any frontend, no assumptions are made whatsoever: console or graphical, blocking or unblocking user input, single- or multithreaded, et cetera. This design goals is met.

Performance This design goal stated that the REPL should be performant, because care is taken to ensure that Spoofax itself is performant.

Launching the REPL is near instantaneous. Commands and expressions are evaluated in real time, without noticeable delay. This design goals is thus met.

Modify Spoofax's existing codebase as little as possible This design goal stated that the REPL should be an extension to Spoofax, which means that the existing codebase should be modified as little as possible.

The REPL is developed into a standalone repository. It uses the same API exposed by Spoofax as any other extension would use. Minimal changes have been made to Spoofax itself: the only large change that went upstream is the ability to configure the REPL in an ESV file. However, DynSem had to be modified after its rewrite to support the use case of a REPL. These changes were made in cooperation with its maintainer, and most were features that had to be implemented regardless. As such, this design goal is also met.

6.1.3 Evaluation of the product

This section evaluates whether the minimal viable product, as defined in section 3.3.3, has been delivered. The minimal viable product has been defined as all the "must have" requirements (see section 3.3.2), including the design goals of being language-agnostic and requiring only additive configuration.

The previous section concluded that all the design goals have been met. Section 6.1.1 and table 6.1 show that all the "must have" requirements have been implemented, except one: syntax highlighting. The backend is ready to provide this functionality, and a working prototype has been developed.

It is currently impossible to provide syntax highlighting in the console frontend. The text-only environment is the limiting factor: once text has been printed, it cannot be changed. This means that text would have to be printed twice: once as it is typed in by the user, and again with syntax highlighting applied.

The Eclipse frontend has the required functionality to provide syntax highlighting. The backend, however, has no way of telling which widget (see figure 5.5) to send the resulting text to. This means that as of yet, only text in the output buffer can be highlighted.

As one might conclude, much of the code to implement syntax highlighting is present. Finishing this work would not take much more effort. As such, this must have requirement is close to being met.

The client has accepted the product after an acceptance test. The absence of this feature thus does not influence the acceptance of the product. The minimal viable product is therefore deemed to be delivered.

6.2 Process Evaluation

This section gives an evaluation of the process by which the delivered product came to be. The methodologies used during development are discussed in section 6.2.1. Section 6.2.2 shows the feedback received from the Software Improvement Group and discusses the changes that were made to the code accordingly.

6.2.1 Development methodologies

In the beginning of the project, agreements were made on the development methodologies that were to be used. Throughout the duration of the project, some of these methodologies turned out to work well, whilst others were less effective than anticipated. This section discusses these agreements.

The Scrum methodology was chosen to manage the product development. Previous experience showed that one week sprints are often too short, so for this project biweekly sprints were decided. This worked well: there was enough time to make informed decisions and to evaluate or rework certain things. Biweekly sprints also allow for an easier revision of the sprint plan if changes need to be made due to unexpected issues.

Sprints were evaluated with a meeting with the client, during which the progress was demonstrated. The feedback received during these meetings was valuable and taken into account to guide the next sprint.

The client mandated the use of Trello to manage the sprints, which has been a positive experience as well. Trello makes it easy to manage deadlines and to know what has to be

done at what time. Managing a backlog of user stories and moving them around from, for example, "developing" to "testing", is straightforward and helps to keep everyone, including the client, updated on the progress. The pull-based development model worked well, as was anticipated from previous experience. The continuous integration, in combination with a set of strict static analysis tools, ensured that the code review process had a positive influence on the quality of the code.

Naturally, there were also things that could have gone better. It is easy to underestimate the amount of work when putting together a sprint plan, which has been a source of problems. The solution for this issue is to deliberately plan less: it is easier to move new user stories to the sprint plan than it is to decide which user stories not to complete.

A second issue concerns the unstable development environment provided by Eclipse. Omitting all the details, several hard to track issues occurred which took a considerable amount of time to resolve. To add to this, on several occurrences the development of the Eclipse plugin had to wait for changes to make it into Spoofax.

The biggest issue that occurred during the development is the fact that Spoofax is a moving target. Halfway through the allotted time for development, DynSem was rewritten. After this rewrite, the functionality required for a REPL was no longer present. To resolve this, considerable effort has been spent to add these features to DynSem in cooperation with DynSem's maintainer. All of this work has been accepted upstream and has now been published in the latest snapshot releases.

To conclude, the process by which the product came to be went predominantly well, despite some issues that can be improved during future projects.

6.2.2 Software Improvement Group

The code was sent to the Software Improvement Group[1] for review on May 27th. The following feedback was received on the third of June:

> [Analyse]
> De code van het systeem scoort 4,5 ster op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Interfacing.
>
> Voor Unit Interfacing wordt er gekeken naar het percentage code in units met een bovengemiddeld aantal parameters. Doorgaans duidt een bovengemiddeld aantal parameters op een gebrek aan abstractie. Daarnaast leidt een groot aantal parameters nogal eens tot verwarring in het aanroepen van de methode en in de meeste gevallen ook tot langere en complexere methoden.
>
> In jullie geval heeft de constructor van `Data` een enorm aantal parameters. Er zijn grofweg twee manieren om dat te addresseren: de eerste is het groeperen van velden in sub-types. Een flauw voorbeeld is het groeperen van `width` en `height` in een type `Dimensions`. Dit voorbeeld is natuurlijk triviaal, maar je ziet wel vaak dat studenten aan het begin wat huiverig zijn om kleine classes voor alleen een type te introduceren, terwijl dit op de lange termijn

[1] `https://www.sig.eu/`

> vaak wel degelijk zin heeft. De tweede manier is niet meer alle parameters in de constructor meegeven, maar via setter-methodes. Dat leidt tot meer code, maar maakt het intialiseren van een object wel leesbaarder.
>
> Een ander voorbeeld van dezelfde situatie is de constructor van `TransformCommand` (zo te zien krijgen jullie daar ook al een waarschuwing vanuit CheckStyle). Dit geval is wat moeilijker op te lossen, aangezien jullie hier van dependency injection gebruik maken en de methode daardoor wat moeilijker kunnen refactoren.
>
> Het is goed om te zien dat jullie naast productiecode ook veel testcode hebben geschreven. De verhouding tussen de hoeveelheid productiecode en testcode is met bijna 1 op 1 ook zeer gezond. Hopelijk lukt het jullie om dat vast te houden tijdens het vervolg van het project.

This analysis says that the code scored bad on unit interfacing, because there were classes that had an above average amount of constructor parameters. The reason that this came to be is the fact that dependency injection is used, as opposed to the traditional way of managing dependencies. This fact is also mentioned in the analysis itself.

Two suggestions were given as a possible solution to this problem: create new classes that encapsulate parameters, or use `setter` methods to pass parameters this way. However, neither solution was satisfactory as-is: data classes are often considered to be a code smell, and `setter` methods risk not fully initializing an object. The classes to which the criticism applied have been refactored into smaller, more manageable units. Ultimately, this not only led to resolving the only negative feedback made by the Software Improvement Group, it also led to a better architecture.

On June 28th, the second round of feedback was received:

> [Hermeting]
>
> In de tweede upload zien we dat de omvang van het systeem flink is gestegen. Er is ook een lichte stijging te zien bij de score voor onderhoudbaarheid.
>
> Die stijging komt voornamelijk door de verbetering van Unit Interfacing. Dit is zeer positief, aangezien dat Unit Interfacing het meest kritische punt was bij de vorige evaluatie. De score voor Unit Interfacing is op dit moment marktgemiddeld.
>
> We zien ook stijging in het volume van de testcode. De verhouding tussen nieuwe geschreven productiecode en testcode is bijna 1:1, wat optimaal is.
>
> Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie zijn meegenomen in het ontwikkeltraject.

This analysis says that the size of the codebase has increased significantly. The maintainability score has improved slightly, primarily because the score for Unit Interfacing increased. The ratio between production and test code is almost one to one, which is an optimal score. From these observations, the Software Improvement Group concluded that the recommendations made in the first round of feedback have been taken into account.

7 Discussion

Chapter 6 discussed how the final product satisfied the client's requirements. In this chapter, additional open issues are discussed, as well as some obstacles that were met during the development.

7.1 Partial Program Evaluation by Wrapping

During the research phase of the project, the investigated solutions to the problem of partial program evaluation were primarily focused on "wrapping" an incomplete program such that it becomes a valid program as a whole, as the OpenJDK team has done for JShell (see appendix C.5). A solution using templates for various language constructs was proposed to keep this approach generic across all languages supported by Spoofax. Figure 7.1 shows a diagram which describes this idea. The diagram shows the wrapping of both the environment definitions and the incomplete program AST. The environment definitions, in this case, refer to the language-specific constructs such as class definitions, function definitions and variable declarations.

However, as the project developed and the entry point to DynSem was refactored (see section 7.2), wrapping turned out to be an unnecessary and inadequate solution for two reasons. Firstly, the ability to invoke a rule corresponding to any AST, both complete and incomplete, automatically allows for evaluating incomplete programs without wrapping them first to form a complete program. Secondly, due to the ability to pass the semantic components as arguments to a reduction rule, the need of wrapping the environment

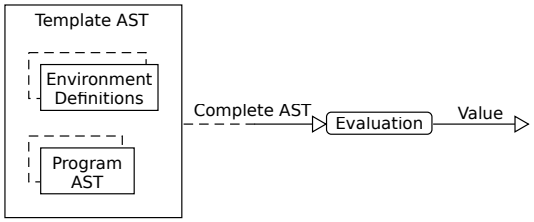


Figure 7.1: Wrapping the environment definitions and an incomplete program AST inside of a template to form a complete and valid program AST.

definitions (such as functions and variables) became unnecessary too.

The custom start symbol as described in section 5.2 together with the REPL-specific rules used to obtain and pass execution contexts to DynSem then already results in a fully functional REPL, without the need to wrap partial programs inside predefined templates.

7.2 Refactoring DynSem's Entry Point

The implementation of DynSem changed during the project: instead of letting the generated interpreters be defined on their own, they now use the Truffle language implementation framework for their implementation [7]. Due to this change, the interface of the entry points of the generated interpreters changed as well.

Initially, the only available rule was the "main" rule for evaluating complete programs, rather than any rule. Furthermore, the evaluation of a program was tied to the parsing of that program's original source text, even if the parsing step had already been executed before by the REPL. Lastly, the entry point did not allow for the evaluation in an existing context, an important requirement for the REPL. The new entry point therefore required a refactor, so that firstly rules could be looked up and invoked individually in an existing evaluation context, and secondly so that evaluation could be done on ASTs rather than only program source text.

The refactoring consisted of separating parsing from evaluation and allowing all of the rules to be accessed through Truffle's foreign object access interface [5]. These changes alone also allowed for invoking the rules with an existing context. The changes were approved by the author of DynSem, and have been integrated into the upstream repository.

7.3 Eclipse Multithreading Issues

The implementation of the Eclipse frontend has been a source of exposing shortcomings in the initial designs. These shortcomings and how they have been resolved are discussed in this section.

7.3.1 Single- versus multithreading

The initial design assumed that the frontends and the backend would run in the same thread. For a console based REPL, this assumption holds and greatly simplifies the design. However, this assumption does not hold when the frontend uses a multithreaded GUI toolkit. This assumption resulted in two problems, which are listed separately in the next sections. The solution and changes made to the design are discussed afterwards.

7.3.2 Blocking- versus non-blocking input

In a multithreaded environment, asking graphical text entry widgets for the entered text is rarely a blocking process. The REPL backend at the time, however, assumed that getting the user's input was always a blocking operation. Therefore, when a conceptual Eclipse frontend was made, the REPL spun into an infinite loop trying to execute empty expressions.

7.3.3 Evaluating on the UI thread

As explained in section 5.4.2, multithreaded graphical user interface toolkits designate one of their threads the "UI thread", or user interface thread. This thread is responsible for processing events (such as mouse clicks) and updating the graphical representation of the widgets. All tasks that perform long running calculations are supposed to be run in a background thread, such that the UI thread is free to process incoming events. Instead, if a long running computation is run in the UI thread, the widgets on the screen stop responding to the user and the program appears to be in a frozen state. This is exactly what happened when the backend assumed to be run in the same thread as the frontend: whilst the backend was evaluation expressions, Eclipse appeared to be frozen due to this evaluation taking place in the UI thread from which the execution was started. This issue would be worse in case blocking input were to be used.

7.3.4 Accustoming to multithreaded frontends

As indicated in the previous sections, the only solution to these problems is to allow multithreaded frontends. This meant that the assumption of every operation running in the same thread no longer held. As a result, several changes had to be made:

1. Initially, a `Repl` class was present in the backend to centralize a REPL implementation. This class made too many assumptions on behalf of the frontends, among which blocking user input and the entire main loop of a REPL. The alternative is the `IRepl` interface (see section 4.1), which defines the only method each frontend has in common: the `eval` method to evaluate input. It is now entirely up to the frontend on how to implement the read, print and loop steps, allowing for much more freedom.

2. The initial design provided "hooks" to the frontend to process results or errors. A frontend would register itself to process certain hooks, after which the registered method would be called. The registered methods would be called immediately after the evaluation was done, and on the same thread. In the conceptual Eclipse frontend, these hooks updated the widgets running in the UI thread. This then led to widgets being updated from a thread other than the UI thread. The solution, discussed in section 4.4, was to return the result to the frontend, allowing it to process the result whenever and however it needs.

Not only did these changes resolve the threading issues, they also made for a cleaner architecture overall. An additional advantage is that a much wider range of possible frontends is now supported.

7.4 Language Interpreters on the Classpath

In section 5.1 the implementation of the DynSem evaluation strategy was outlined, together with its backend, the Truffle VM. From the DynSem specification of a language an interpreter can be generated, which uses Truffle's `PolyglotEngine` for its implementation.

As any other Truffle language, the language of the interpreter has to be registered with Truffle by annotating it correctly[1]. DynSem generates this annotation for languages defined in Spoofax, after which the `PolyglotEngine` finds and registers all annotated languages when it is instantiated. However, according to the Truffle specification, the `PolyglotEngine` will *"search for languages registered in the system class loader and make them available for later evaluation[2]"*.

As a consequence, interpreters for languages to be used inside the REPL need to be on the classpath of the system class loader before instantiating a `PolyglotEngine`, which implies that a REPL user should specify the path to the desired language interpreter before starting the REPL. Since Truffle enforces the use of the system class loader, attempts to load the `PolyglotEngine` using a custom class loader unfortunately did not solve this issue. It is suspected that this issue will become apparent in Spoofax as well once the new DynSem implementation will be integrated more tightly.

7.5 DynSem Data Types for Rule Results

Currently, the results of invoking reduction rules are of type `Object`. The actual type of the semantic components is unknown, as a semantic component can be of any type defined in the DynSem specification.

As such, it is hard to perform operations such as printing on the returned data. One could simply call the `toString` method, but often more control is needed. For example if one wants to provide a graphical view of the environment in the form of a tree structure, this solution is inadequate.

Furthermore, there is at the time of writing an open issue in DynSem to support conversion of its internal term representation to a Stratego term representation. The result of an evaluation can then be treated just as any other Stratego term, giving more control to the frontend in deciding how to present the result to the user.

[1] See: `http://lafo.ssw.jku.at/javadoc/truffle/latest/com/oracle/truffle/api/TruffleLanguage.Registration.html`

[2] See: `http://lafo.ssw.jku.at/javadoc/truffle/latest/com/oracle/truffle/api/vm/PolyglotEngine.html`

8

Recommendations

This chapter makes several recommendations to the client for future work on the delivered product. First, recommendations are made to improve the backend itself. Secondly, suggestions are made on how to improve the Eclipse frontend. Finally, a recommendation is made to provide literate programming functionality.

8.1 Extending the Functionality of the Backend

The delivered backend is a solid product. Due to time constraints, however, some desirable features have not been implemented. This section shortly highlights these features.

8.1.1 More extensive history functionality

The current input history implementation provides a means of iterating the previously entered expressions in a linear way: a user can scroll back and forth through the old entries. It would be nice if the history was searchable, too. When implementing this, one could turn to existing command-line shells for inspiration. The GNU readline library, for example, has two ways of searching through the input history: via a keyboard shortcut, which when pressed allows the user to enter a word that they remember is in the entry they are looking for, or via an always-on setting that allows the user to enter the beginning of the expression which in turn filters the linear iteration over the history to only those entries starting with the entered input.

Another limitation of the current history implementation is the fact that it is recorded per session. This means that when a user uses several languages in the same session, the history contains entries from all those languages. Instead, history should perhaps be kept not only per session, but also per language.

Related to the above, is implementing persistent history. The Eclipse frontend currently only provides volatile history, which means that history is thrown away when Eclipse is closed. The console plugin does provide persistent history, but this implementation also leaves things to be desired: all history is saved into the same file, resulting again in a mixed history. The recommendation that is made here is to provide the ability to have separate, per language persistent history. If the frontend is an IDE, these files can perhaps be saved inside the language project's directory structure.

Improving the history implementation should result in a smoother and faster way to interact with the REPL, enhancing the explorative nature.

8.1.2 Analysis in context

Section 7.2 explained how DynSem was refactored to allow passing in an existing context in which REPL commands are evaluated. As explained in section 2.1, some languages also define rules for static analysis. While support for evaluating an expression in a given context has been added to DynSem, Spoofax's `AnalysisService` currently does not support this use case.

Providing analysis in the context of previous expressions is a desirable feature that would enhance the REPL. Currently, however, Spoofax does not provide the means to support this.

8.1.3 Adding support for alternative evaluation strategies

As discussed in section 4.5, languages developed with Spoofax are not limited to a single interpreter. Instead, there can be different strategies for evaluation. While this has been taken into account for the design of the product (see section 5.2), only a DynSem evaluation strategy has actually been implemented.

Languages with a Java backend (such as IceDust[1]) therefore do not currently work with the REPL. To add support for specific interpreters, a language should provide the REPL with a named implementation of the `IEvaluationStrategy`. The language designer can register alternative evaluation strategies with the REPL by extending the Guice module of the backend and overriding the `bindEvalStrategies` method.

8.2 Integrating Evaluation with DynSem in Spoofax

Most of the execution steps as outlined in section 2.1 have a corresponding API in Spoofax to provide programmatic access to those aspects. For example, for parsing there exists a `SyntaxService`. However, currently Spoofax lacks an `EvaluationService` to perform evaluation with DynSem's generated interpreters. Such an interface could hide the implementation details of Truffle and class loaders (see section 7.4). Instead it can provide an API for performing evaluations in context. The existing evaluation backend for DynSem could then be adapted to use the `EvaluationService` instead.

8.3 Improvements to the Eclipse Frontend

The Eclipse frontend that is delivered with the product has been discussed in section 5.4.2. The plugin provides interaction with the REPL backend, as discussed in chapter 4.

The delivered Eclipse frontend is just that: a frontend to the REPL backend. Integration with Eclipse and Spoofax Eclipse are not yet present. This section provides several recommendations to better integrate this frontend into Eclipse and Spoofax Eclipse.

[1] https://github.com/MetaBorgCube/IceDust

8.3.1 Building languages and projects

To start a REPL session in the delivered project, the language designer has to build the language, start the REPL and issue a "`:load`" command to load the language. It is desirable if all this could be done automatically at the press of a button. Such a *"Run in REPL"* button can be added to the "Spoofax (meta)" submenu and inside the project and package explorer context menus.

When the REPL gains the ability to load existing files, such a button can also be added to the "Spoofax" submenu to assist a language user.

An obstacle in implementing this feature is the classpath issue as described in section 7.4.

8.3.2 Interaction between Spoofax Eclipse and the Eclipse plugin

The REPL backend supports loading individual files to bring the definitions inside these files into scope. This functionality, however, is not yet provided by the Eclipse plugin by means of menu items or buttons, nor is there a deeper integration between the Eclipse plugin and Spoofax Eclipse.

Deeper integration improves the interaction between Spoofax Eclipse and the Eclipse frontend. For example, hovering over variables could indicate where they are defined, or changes in a currently loaded file could be automatically picked up. Changes in the other direction are interesting, too: when a function definition is overridden inside the REPL, an option could be provided to apply this change to its originating file as well. This could even be extended to write (parts of) the current environment to a new file.

8.3.3 Improving the UI

The user interface currently offered by the Eclipse frontend does not resemble a typical REPL: the input and output views are separated. To improve the user experience, these two views could be merged into one. This would require the use of a widget or text representation that can be *"semi-read-only"*, as the previously entered text and results should not be editable, while the current input should be.

8.4 An IPython Kernel for Jupyter Notebooks

The client expressed their interest in literate programming functionality for Spoofax, as explained in section 2.3. Implementing an IPython kernel on top of the core shell module should not take much effort, however due to the problems discussed earlier priority was given to a solid REPL and an Eclipse plugin. Therefore, the product currently does not deliver an implementation of literate programming.

IPython kernels provide a solid framework for literate programming that has proven itself with many different languages[2]. An IPython kernel is a daemon with network sockets representing the input and output streams of the frontend application. Several message

[2]See: `https://github.com/ipython/ipython/wiki/IPython-kernels-for-other-languages`

types are sent over these sockets in JSON format, such as requests to evaluate a certain string or requests regarding the kernel state[3].

Since the invoker class from the backend accepts any string given to it and then resolves the command itself, implementing an IPython kernel should be straightforward. Most of the required work would be in creating an adequate messaging framework capable of understanding all defined JSON messages. When the messaging framework is in place, user input can be sent directly to the invoker after which the results can be returned via network sockets by visiting the `IResult` interface, similar to any other frontend implementation.

All functionality offered by Jupyter notebooks as shown in figure 2.4 will be available to the Spoofax REPL at once by implementing an IPython kernel (if the language supports these features). This directly results in the ability to live edit code interspersed with documentation, while also allowing more complex graphical elements.

[3]See: `https://jupyter-client.readthedocs.io/en/latest/kernels.html`

9 Conclusion

Over the course of the last quarter the Spoofax Language Workbench has been explored and a Read-Eval-Print Loop (REPL) has been created that operates with any language defined in the Spoofax.

Chapter 2 gave the required background knowledge needed to understand the problem domain. Chapter 3 framed the problem definition in the context of this background. Chapter 4 and chapter 5 discussed the design and the implementation of the final product. Chapter 6 evaluated both the final product and the process by which it came to be. Chapter 7 reflected upon the project, after which chapter 8 made recommendations to improve the final product.

To successfully complete the project, extensive knowledge needed to be gained of the conceptual ideas behind programming language implementations, and of the Spoofax API implementing these concepts.

Once the required knowledge was attained, two contributions have been made. First, a functioning REPL has been created, comparable in features to those of popular programming languages such as Python and Haskell. Second, changes have been contributed to Spoofax that have been integrated into the main repository.

Despite having faced significant challenges during the start of the project, the most important goals as set forth in the problem description have been achieved. Therefore, the project team is still satisfied with the end result: while not all requested features have been implemented, it has been shown that it is possible to create a functioning REPL for any language defined in Spoofax, requiring only additional configuration to a language definition.

Bibliography

[1] Maartje de Jonge, Lennart C. L. Kats, Eelco Visser, and Emma Söderberg. Natural and Flexible Error Recovery for Generated Modular Language Environments. *ACM Trans. Program. Lang. Syst.*, 34(4):15:1–15:50, December 2012. ISSN 0164-0925. doi: 10.1145/2400676.2400678. URL http://doi.acm.org/10.1145/2400676.2400678.

[2] Peter Fritzson. Systems and tools for exploratory programming overview and examples. Technical Report R-86-36, 1986. Also presented at the Workshop on Programming Environments - Programming Paradigms, Roskilde, Denmark, October 22-24, 1986.

[3] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[4] Paul Graham. *On Lisp*. Prentice Hall, 1993. ISBN 0130305529.

[5] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-performance cross-language interoperability in a multi-language runtime. *SIGPLAN Not.*, 51(2):78–90, October 2015. ISSN 0362-1340. doi: 10.1145/2936313.2816714. URL http://doi.acm.org/10.1145/2936313.2816714.

[6] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF - Reference Manual. *SIGPLAN Notices*, 24(11):43–75, November 1989. ISSN 0362-1340. doi: 10.1145/71605.71607. URL http://doi.acm.org/10.1145/71605.71607.

[7] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing ast interpreters. *SIGPLAN Not.*, 50(3):123–132, September 2014. ISSN 0362-1340. doi: 10.1145/2775053.2658776. URL http://doi.acm.org/10.1145/2775053.2658776.

[8] Apple Inc. Swift playgrounds, 2014. URL http://devstreaming.apple.com/videos/wwdc/2014/408xxcm26svis12/408/408_swift_playgrounds.pdf.

[9] IPython. Execution semantics in the ipython kernel, 2014. URL http://ipython.readthedocs.io/en/stable/development/execution.html.

[10] G. Kahn. *Natural Semantics*, pages 22–39. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987. ISBN 978-3-540-47419-7. doi: 10.1007/BFb0039592. URL http://dx.doi.org/10.1007/BFb0039592.

[11] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*,

pages 444–463, Reno/Tahoe, Nevada, 2010. ACM. ISBN 978-1-4503-0203-6. doi: http://doi.acm.org/10.1145/1869459.1869497.

[12] Lennart C. L. Kats, Karl Trygve Kalleberg, and Eelco Visser. Domain-Specific Languages for Composable Editor Plugins. volume 253, pages 149–163, 2010. doi: http://dx.doi.org/10.1016/j.entcs.2010.08.038.

[13] Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 918–932, Reno/Tahoe, Nevada, 2010. ACM. ISBN 978-1-4503-0203-6. doi: http://doi.acm.org/10.1145/1869459.1869535.

[14] Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[15] Gabriël D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2012. ISBN 978-3-642-36089-3. doi: http://dx.doi.org/10.1007/978-3-642-36089-3_18.

[16] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. Introduction. In *The Definition of Standard ML: Revised*, chapter 1. MIT Press, 1997. ISBN 9780262631815.

[17] James L. Noyes. *Artificial Intelligence with Common Lisp: Fundamentals of Symbolic and Numeric Processing*. D. C. Heath and Company, Lexington, MA, USA, 1992. ISBN 0-669-19473-5.

[18] OpenJDK. jshell: The java shell (read-eval-print loop), 2016. URL `http://openjdk.java.net/jeps/222`.

[19] Fernando Pérez and Brian E Granger. Ipython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007.

[20] Eric Schulte, Dan Davison, Thomas Dye, Carsten Dominik, et al. A multi-language computing environment for literate programming and reproducible research. *Journal of Statistical Software*, 46(3):1–24, 2012.

[21] The GHC Team. GHCi commands, 2015. URL `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci-commands.html`.

[22] Mark GJ van den Brand, HA De Jong, Paul Klint, and Pieter A Olivier. Efficient annotated terms. *Software Practice and Experience*, 30(3):259–291, 2000.

[23] Vlad A. Vergu, Pierre Neron, and Eelco Visser. DynSem: A DSL for Dynamic Semantics Specification. In Maribel Fernández, editor, *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*, volume 36 of *LIPIcs*, pages 365–378. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. ISBN

978-3-939897-85-9. doi: 10.4230/LIPIcs.RTA.2015.365. URL http://dx.doi.org/10.4230/LIPIcs.RTA.2015.365.

[24] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[25] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer, 2001. ISBN 3-540-42117-3. doi: 10.1007/3-540-45127-7_27. URL http://link.springer.com/chapter/10.1007/3-540-45127-7_27.

[26] Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. Declarative Specification of Template-based Textual Editors. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 8:1–8:7, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1536-4. doi: 10.1145/2427048.2427056. URL http://doi.acm.org/10.1145/2427048.2427056.

[27] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-23169-7.

[28] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509581. URL http://doi.acm.org/10.1145/2509578.2509581.

A

Project Infosheet

General Information

Title of the project: Spoofax Shell
Name of the client organization: PL group, Delft University of Technology
Date of the final presentation: July 1st, 2016
Description: Create a command-line shell interface for the Spoofax Language Workbench

Description

The TU Delft PL group conducts research into concepts and techniques for programming language design and implementation. The flagship product of the group is the Spoofax Language Workbench.

The deliverable for this project was to create a REPL for the Spoofax Language Workbench. The challenge was to figure out how such a REPL fits within the larger context of Spoofax, and how to create a generic REPL that does not make assumptions on its host language. In order to integrate the REPL, the students not only had to learn what Spoofax is and how it works, but also how it is build and how the different parts fit together. The existing architecture of Spoofax directed the architecture of the product.

The development process was managed with the Scrum software development framework. The source code was managed using Git, using the pull-based development methodology.

The largest unexpected challenge faced during the project is the rewrite DynSem halfway through the project. This new version of DynSem did not expose the required functionality for a REPL. Collaborating with its developer, these features were implemented over the course of two weeks.

The delivered product consists of a backend, providing a means of parsing, analyzing and evaluating expressions using Spoofax's services. Two frontends were delivered: one running in the console and the other inside the Eclipse IDE. Both the frontends and the backend are tested with unit- and integration tests.

The client accepted the product after an acceptance test. Recommendations to further enhance it have been made to the client. For example, more interpreter backends can be provided and the Eclipse frontend can integrate more with Spoofax Eclipse.

Members of the project team

Gerlof Fokkema's interests are Operating Systems, networking and programming languages. Gerlof primarily worked on the backend, specifically the Spoofax commands available to the user from within the REPL.
Jente Hidskes' interests are Operating Systems and programming languages and the collaboration between the two. Jente mainly worked on the frontends of the REPL and the communication with the backend.
Skip Lentz's interests lie in the field of programming languages, both object-oriented (Smalltalk) and functional (Haskell), which provide novel and more efficient ways for reasoning about program logic. Skip mainly worked on the backend, specifically the interoperability with DynSem's generated interpreters.

Client:

Eelco Visser, PL group TU Delft

Coach:

Hendrik van Antwerpen, PL group TU Delft

Contact:

Gerlof Fokkema, gerlof.fokkema@gmail.com
Jente Hidskes, hjdskes@gmail.com
Skip Lentz, skippetie@gmail.com

The final report for this project can be found at: `http://repository.tudelft.nl`.

B

Project Description

Create a command-line shell interface for the Spoofax Language Workbench

From Spoofax's website:

> Spoofax is a platform for developing textual domain-specific languages with full-featured Eclipse editor plugins.

A feature that Spoofax is lacking is a Read-Eval-Print Loop (REPL). A REPL is an interactive programming environment that takes expressions, evaluates them and prints their results. REPLs are a popular tool for programming because they facilitate exploratory programming and debugging. Common examples include command-line shells such as Bash and Python's REPL.

The deliverable for this project, then, is to create such a REPL for the Spoofax Language Workbench.

C

Research Report

Introduction

Spoofax is a language workbench developed by Delft University of Technology over the course of several years. During those years it has grown to be *"a language workbench for efficient, agile development of textual domain-specific languages with state-of-the-art IDE support"* [11].

When developing new domain-specific languages using Spoofax, the ability for rapid prototyping using a read-eval-print loop (REPL) would be very convenient. This report describes the initial research phase for this project: extending Spoofax with a REPL that works with all languages defined in Spoofax.

The first three sections of this research report give an overview of the problem domain. Appendix C.1 gives the scope of Spoofax and each of its meta-languages. In appendix C.2, REPLs are explained in more detail. After that appendix C.3 introduces literate programming.

The remaining four sections concern the problem and the project. In appendix C.4, a short problem definition is given. Appendix C.5 then gives an analysis of this problem definition in the context of the services that Spoofax already provides, to clarify potential problems and their solutions when creating a REPL that works with every language defined in Spoofax. Next, in appendix C.6, a list of design goals will be formalized to guide the development of the deliverable. From these design goals and the problem definition and analysis, a list of requirements for the product will be compiled. Lastly, appendix C.7 gives an account of the development frameworks and tools to be used during the development of the project.

C.1 The Spoofax Language Workbench

Spoofax is a platform that allows for giving a completely *declarative* definition of a programming language and accompanying IDE support [11]. Such a platform is called a *language workbench*. The definition of a programming language is done using high-level *meta-languages* for each aspect of the programming language.

To define a language declaratively means that one uses the meta-languages to specify *what* the properties of a language are, not *how* these properties are implemented. For

example, instead of asking *"How do I implement a tokenizer and parser for my language?"*, one asks *"What is the syntax of my language?"*. From such a description in a meta-language, the tokenizer and parser can be derived without the designer of the language ever having to care about its implementation.

This section goes over the aspects that come into play with the development of a language and how Spoofax tackles each of these aspects. First, the section goes over the elements that make up the specification of a language[1]. A language specification consists of the following conceptual steps:

1. Syntax Definition: The first step defines what textual representations of a program are syntactically valid. A parser provides an implementation of this definition, by mapping a textual representation of a program to an abstract syntax tree (AST) representation. In Spoofax, the syntax is declared with a domain specific language (DSL) called SDF.

2. Static Semantics: The AST then goes through static analysis (type checking, name binding and variable scoping), to test if the program is well-formed. Static semantics describe the rules for the static analysis step. Spoofax provides two DSLs that can specify the two distinct parts of this step: the TS Type Specification language and the NaBL name binding language.

3. Term Rewriting and Program Transformation: Optionally, a well-formed AST can then be transformed, for example for desugaring or optimization. Spoofax provides Stratego for this step.

4. Dynamic Semantics: Next the optionally transformed AST is either compiled or interpreted, thereby providing a means of execution. Dynamic semantics define what the behaviour is of a program upon execution. In Spoofax, the dynamic semantics can be defined with either Stratego or a DSL called DynSem.

This section concludes with a discussion on the other aspect of a language: its integrated development environment (IDE). Spoofax provides IDE support by means of its Editor Services.

C.1.1 Syntax Definition

The first part of the specification of a language is its syntax. The syntax of a language is often specified by means of a *lexical grammar* and a *context-free grammar*, as can be seen in the specification of, for example, Standard ML [16]. The lexical grammar is most often defined using regular expressions. It defines the individual words made up of characters, such as identifiers and numeric constants. The context-free grammar then defines syntactically valid sentences made up of words.

[1]This section follows the structure of the language specification portion of the compiler construction course at the TU Delft. The slides can be found here: `http://tudelft-in4303.github.io/lectures/specification/`.

SDF3: syntax definition in Spoofax

To specify a syntax definition declaratively in Spoofax, a DSL called *SDF3* [26] is used. SDF3 is the third generation of the *Syntax Definition Formalism* (SDF) [6]. It uses only context-free grammer productions for the specification of both the lexical syntax and the context-free syntax, a feature that was introduced in SDF2 [24].

The declarative nature of SDF3 allows for thinking in terms of the structure (the *what*), instead of in terms of parser algorithms (the *how*) as is the case with many current parser generators such as ANTLR and YACC [13]. The syntax definition is used to make parsers that parse a textual representation of a program into its AST and pretty-printers for mapping ASTs back to text. However, due to its declarative nature, SDF3 is not limited to generating parsers and pretty printers: it can also be used for error recovery rules [1], syntax highlighting rules and folding rules for editors (see appendix C.1.5).

The Spoofax API gives access to the generated parser through the `SyntaxService`.

AST-based rules

The meta-languages that will be discussed in the coming sections all have one property in common: all of them use *rules* based on the AST in order to specify one of the parts of a language definition. The rules are said to be *syntax-directed*: the specification for one AST node (whether it be a static semantics, rewriting or dynamic semantics specification) is done by the specification of the children of that AST node [27].

C.1.2 Static Semantics

Static semantics refer to the meaning of what a well-formed program is for a particular language [16]. This imposes more constraints than syntax definition, such as name binding, scoping rules and type checking. These cannot be specified by a syntax definition alone and are thus considered separately.

Declarative static semantics specification in Spoofax

In Spoofax, all the static semantics as well as the dynamic semantics used to be specified with the *Stratego* transformation language (which is discussed in appendix C.1.3). Nowadays, two high-level DSLs exist for specifying static semantics declaratively: NaBL and TS. The two DSLs can work together: for instance, the type of a variable can be set with NaBL, so that TS can be used to make assertions on the type of that variable.

The static analysis step of a language is exposed through the Spoofax API by the `AnalysisService`.

NaBL: the Name Binding Language

With *NaBL* (pronounced *enable*), name binding and scoping can be specified declaratively using AST-based rules [15]. Here is an example of name binding and scoping rules for a class, from the *paplj* language.[2]

[2] paplj is used as an exercise language for the "Declare Your Language" book, which is a work-in-progress at the time of writing. More information can be found here: `https://github.com/MetaBorgCube/declare-your-language`

```
1  namespaces Program Class Field Method Variable
2  // ...
3  binding rules
4    Class(c, _, _, _) :
5      defines Class c of type ClassT(c)
6      // Declare new scope
7      scopes Field, Method, Variable
8      implicitly defines Variable This() of type ClassT(c)
9
10   Extends(c) :
11     // Import namespaces from superclass
12     imports Field, Method from Class c
```

The most important concept to take away from this example is the way the rules are specified on the AST: new scopes for names can be defined on the level of an AST node, and can be imported again by referring back to the scope definition.

As can be seen from line 8, it can also associate type information with names to interplay with TS. The type annotations can also be used for instance when desugaring or rewriting with Stratego (see appendix C.1.3).

TS: the Type Specification language

Type checking can be done by specifying typing rules with the *TS* DSL. Again an example of the paplj language:

```
1  type rules
2    Class(c1, Extends(c2), _, _) :-
3      where store ClassT(c1) <sub: ClassT(c2)
4
5    x@This() : t
6      where definition of x : t
7  // ...
8  type rules
9    Add(e1, e2) : NumT()
10     where e1 : NumT() else error "number␣expected" on e1
11       and e2 : NumT() else error "number␣expected" on e2
```

This example shows how in TS, the rules are syntax-directed: The typing rule of the Add node is specified by the types of its children e_1 and e_2, on which the typing rules will be applied recursively.

Again, in line 6, interplay can be seen between TS an NaBL. Here the type of a variable can be accessed, which is set in the NaBL specification (see the previous section, appendix C.1.2).

C.1.3 Term Rewriting and Program Transformation

Sometimes the AST needs some form of transformation before it is to be compiled or executed, for example to transform it to a canonical form, or to perform optimizations such

as constant folding. Program transformations are specified by *term rewrite rules*: The left-hand side of a rule introduces a pattern (for example $x + x$), and the right-hand side specifies a replacement for it (e.g. $2 \cdot x$).

Rewriting using Stratego

Spoofax offers a DSL called *Stratego* for specifying program transformation with rewrite rules [25]. Stratego can be seen as the most general part of Spoofax: before NaBL and TS, Stratego was used for specifying the static semantics. Moreover, being a program transformation language, it can also serve as a compiler and can thus be used to specify the dynamic semantics.

An example of a rewrite rule for the paplj language is given below.

```
1 rules
2   desugar-let :
3         Let([], e) -> e
4
5   desugar-let :
6         Let([b1, b2 | bs], e) -> Let([b1], Let([b2 | bs], e))
```

This desugars a `let` expression with multiple bindings into multiple nested `let` expressions each having just one binding. Again it can be seen that these are syntax-directed rules, from the way the rules are specified using the AST.

To construct the main algorithm of the program transformation, Stratego has the notion of *strategies*. A strategy is used to specify where and in what order the rewrite rules are applied to an AST. Another example from paplj is given below:

```
1 strategies
2   pre-desugar =
3     innermost(desugar-let <+ desugar-do)
4
5   post-desugar =
6     innermost(desugar-do <+ desugar-get <+ desugar-set);
7     resugar
```

The strategy `innermost` in this example is used to apply the strategy given as parameter (a composition of rewrite rules) in a specific traversal order on the AST nodes.

The Spoofax API provides the `TranformService` for performing program transformation. Internally the `TransformService` accesses the Stratego runtime, which is retrieved from the `StrategoRuntimeService`. The same holds for the `AnalysisService` of the previous section: it too uses the Stratego runtime.

Stratego furthermore has support for *native* strategies, which are specified in Java instead. Therefore the interface is bidirectional: Stratego can hook into Java, and Java can use the Stratego API.

C.1.4 Dynamic Semantics

Dynamic semantics refers to how a program written in some language behaves [27]. There are many approaches to formally specify the dynamic semantics of a programming lan-

guage (for an extensive treatment, see [27]). For this section only one sort of approach called *operational semantics* is relevant.

DynSem: rule-based dynamic semantics

Aside from Stratego, the Spoofax team has developed an additional method to declare the dynamic semantics of a language, namely a DSL called *DynSem* [23]. DynSem allows for an operational semantics specification from which a Java-based AST interpreter is automatically generated.

In DynSem, like other meta-languages in Spoofax, the dynamic semantics are specified by means of syntax-directed rules. To show how rules can define the dynamic semantics of a language, consider the classic example of the β-reduction, which defines function application in the lambda calculus. The rule replaces all the occurences of the parameter x with the argument e_2, within the expression e_1:

$$(\lambda x.e_1)e_2 \rightarrow e_1[x := e_2] \tag{C.1}$$

In a similar way, dynamic semantics can be specified in DynSem using a syntax very similar to the formal syntax used in the literature. Take here the example of defining the behaviour of some boolean operators in paplj:

```
1 rules
2   And(BoolV(false), _) --> BoolV(false).
3   And(BoolV(true), e)  --> e.
4
5   Or(BoolV(true), _)  --> BoolV(true).
6   Or(BoolV(false), e) --> e.
```

The example applies the standard rules for boolean operators, and is sufficient to specify the behaviour of these operators. The rules are recursively applied to the expression e on the right-hand side of the rule until it eventually converges.

DynSem generated interpreters can be accessed through the same APIs as those of Stratego, because the interpreter is a native Stratego strategy. Therefore, alternatively, the generated interpreter can also be accessed directly from Java provided that one has the AST of the program to interpret.

C.1.5 Editor Services

This section concludes with a brief description of editor services, which provide the IDE support for languages defined in Spoofax. Examples of such services include an outline view, menus in which one can bind actions to menu buttons (see figure C.1), but also syntax highlighting, syntactic code completion and code folding rules[3].

The Spoofax API provides the editor services with similar naming. For example, the outline can be retrieved from the `OutlineService`, the syntax highlighting can be accessed through the `StylerService` and syntactic code completion is accessed with the `CompletionService`. The defined menus for a particular language can be retrieved with the `MenuService`, from which the menu actions can be retrieved and used.

[3]More services are listed on the Spoofax website: `http://www.metaborg.org/spoofax/editor-services/`

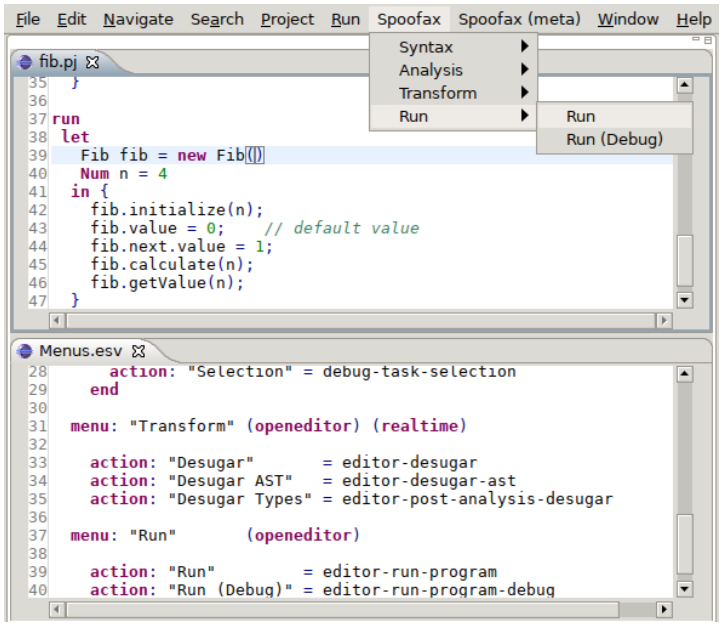


Figure C.1: A menu action for the paplj language defined using Spoofax. The bottom window shows the menu definition, the top window shows a program written in paplj.

Editor services are defined using a DSL, shown in the bottom window of figure C.1. In the case of menus, their actions are specified using Stratego. Since Stratego supports native strategies, these actions can also be specified in Java. As such, Spoofax allows for defining arbitrarily complex IDE actions.

Many of these editor services such as syntax highlighting and code folding rules can be derived from the syntax definition [12] and can be further customized if needed. Taken together with the language definition, the editor services provide a language with a complete and state-of-the-art IDE experience [11].

C.2 Read-Eval-Print Loops

Many programming languages come with an interactive environment. This interactive environment is an interface to the programming language's execution engine. One common form of such an environment is an interface in which expressions in a programming language are typed by the user, after which the results of that expression are printed back to the user. Such an environment is called a Read-Eval-Print Loop (REPL), although many different names are known, including but not limited to *language shell*, *command-line interpreter* or *interactive interpreter*. There are subtle differences between these names and the name REPL. These are, however, mostly of semantic value. In this report the term REPL is chosen, because it conveys the notion of such an interactive environment well.

C.2.1 Origin of REPLs

The Lisp programming language is one of the first programming languages offering such an interactive environment [17]. The name REPL comes from the Lisp functions that implement it:

1. The `read` function takes a user's input, which often is just one or several expressions as opposed to a complete compilation unit. It then parses this input and creates an AST.

2. The AST created in the previous step is then passed on to the `eval` function, which evaluates it.

3. The result yielded by the previous function is then printed out to the user by the `print` function.

4. After having printed the result, the environment needs to `loop` back to the read state.

Assuming the individual functions listed previously exist, a REPL can be created in a single line of code simply by combining the functions:

```
(loop (print (eval (read))))
```

Lisp has a property called "homoiconicity": Lisp's syntax is similar to its internal representation, resulting in the ability to infer a program's or data object's state simply by reading its textual representation. Syntax and AST are thus isomorphic, allowing data and code to be accessed and transformed interchangeably.

In Lisp REPLs, therefore, arbitrary data objects yielded from a previous expression can be used directly as input to the next expression. In programming languages that do not belong to the Lisp family, homoiconicity is an unusual feature. Interactive environments for these languages therefore often require additional steps to read and evaluate expressions. This is part of the semantic differences between the different names as mentioned in the introduction

C.2.2 Advantages of REPLs

REPLs provide the ability to program interactively. Programming interactively has multiple advantages.

When creating software solutions for an as of yet not well understood domain, it is often not clear which data structures and algorithms are required. In such cases, interactively developing and debugging software is an advantage over the (oftentimes much slower) edit-compile-run-debug development style. This kind of programming is called exploratory programming [2]. Related to this kind of exploration, programming interactively also provides a means for rapid prototyping and bottom up programming [4].

The explorative and interactive features of a REPL also make it an excellent tool for programmers to learn a new programming language. REPLs are also combined with what is called literate programming to offer notebooks or language playgrounds, as discussed in appendix C.3.

C.2.3 Execution model

Every programming language defines an execution model, which specifies how programs written in that language are executed. Amongst others, it specifies what an indivisable amount of work is (a *compilation unit*) and in what order these units are executed. The

	Python	R	Common Lisp	Haskell (GHCi)	Swift
Executes single expressions	✓	✓	✓	✓	✓
Executes statements	✓	✓	✓	✓	✓
Input history	✓	✓	✓	✓	✓
Automatic binding of previously yielded values	✗	✗	N/A	✗	✓
Persistent input history	✓	✓	✗	✓	✓
Multiline input editing	✓	✓	✓	✓	✓
Redefining identifiers	✓	✓	N/A	✓	✓
Error reporting	✓	✓	✓	✓	✓
Semantic code completion	✓	✗	N/A	✗	✓
Help or documentation system	✓	✓	✓	✗	✗
Additional commands to the REPL	✗	✗	✓	✓	✓
Nested REPLs to enable debugging	✗	✗	✓	✗	✗

Table C.1: A feature comparison of several well-known REPLs

implementation of an execution model is a compiler and/or an interpreter, often accompanied with a runtime system to provide features such as garbage collection.

The execution model as implemented by a REPL differs only slightly to that of an interpreter or compiler. A difference might be as to what is considered an indivisable unit of work: a REPL might accept singular expressions that a regular interpreter or compiler might not.

A compiler or an interpreter reads the files that make up a program's source code, after which it will go through the steps as outlined in appendix C.1. Usually the resulting program is executed as a whole, without the need for smaller blocks of execution. In contrast, when using a REPL, it might be desirable to be able to execute say only a function body as one unit of execution. However, breaking up a program into smaller blocks of execution might be quite challenging.

Another contrast between a compiler or an interpreter and a REPL is that a REPL needs to dynamically maintain an environment such that each new expression can be evaluated within the environment of previously executed expressions. For every expression entered, it thus needs to apply the outlined steps again and update the environment it maintains in memory with the new results. This could result in a different order of evaluation than when say an interpreter executes a program as a whole.

C.2.4 Functionality

Every REPL provided with a programming language has its own set of functionality. However, a core set of functionalities, shared between all REPL implementations, can be identified. To reach this core set of features, well-known REPLs have been investigated and their features have been compiled into a matrix as seen in table C.1. These features are shortly discussed below.

Input history REPLs keep a history of inputs, such that previously entered expressions can be retrieved.

Automatic binding of previously yielded values When an expression has been evaluated, the yielded result is bound to an automatically created identifier, such that it can be reused easily in future expressions.

Persistent history The input history as discussed previously can be recorded into a file (either per-project or globally) to enable a persistent history of input.

Multiline input editing Some constructs in a programming language naturally span multiple lines. REPLs therefore provide multiline input editors that recognise incomplete code and promptly switch to a multiline environment when required.

Redefining identifiers When using a REPL in an exploratory manner, it is not uncommon to want to redefine an identifier's type or to completely reimplement a method. In this way, a REPL can be different than its host language, especially if the host language is a functional language that does not allow variables' values to change once initiated.

Error reporting A REPL typically has the same error reporting functionality as an interpreter or a compiler, meaning it prints the error message accompanied by the corresponding section of the source code.

Semantic code completion Semantic code completion is a helpful tool to provide an overview of the often many APIs a developer works with, adding to the explorative nature of a REPL. Note that this is a restriction of syntactic completion, which is offered by all the studied REPLs.

Help or documentation system The exploratory nature of a REPL means that one will often see new methods. Some REPLs (most notably Python's REPL with Python's docstrings) offer a documentation system, so that the developer does not have to exit the REPL to look up documentation.

Additional commands to the REPL Some REPLs offer additional commands to inspect the environment or to control their behavior. These commands are oftentimes not in the syntax of the host language and are highly diverse between REPL implementations. A notable example of a REPL offering such commands is Haskell's GHCi [21].

Nested REPLs to enable debugging A notable feature of (mostly) Lisp REPLs is that in case of an error, a new REPL is spawned inside the context of this error. This REPL then has additional commands (see the previous feature) to enable debugging and inspection of the error state. When the user has resolved the error, the nested REPL exits and the user is returned to the parent REPL. This can go to arbitrary depths.

C.3 Literate Programming

Just as with REPLs, the concept of literate programming is implemented in various forms under various names. Therefore this section starts with an explanation of what literate programming is exactly based on a few implementations. Afterwards, the IPython implemention of literate programming is explored in more detail.

Literate programming, as defined in [14], introduces the ability to annotate source code with natural language. According to Donald Knuth, better documentation of programs is essential to make further progress in the state of the art of programming. To achieve this he proposes to write programs not with the intention to explain the computer what to do, but with the intention to explain to humans what the programmer wants a computer to do [14], by mixing documentation and source code in a single file. This idea of literate programming was realized in its original form as the "WEB" language developed during Knuth's research at Stanford University.

Even though the idea was conceived over 30 years ago, implementations are not very common. However, in recent years the idea seems to gain popularity again. A very recent implementation of literate programming is Apple's Swift playgrounds [8]. Swift playgrounds are interactive documents or "notebooks" in which code is executed as it is typed, in contrast to the non-interactive style of "WEB" language in which TEX and PASCAL were combined into one language: TEX served to describe the program and PASCAL to produce a machine-executable program.

In recent years, there has also been a particular focus on reproducible research. While literate programming primarily aims to add documentation to code, reproducible research focuses on adding code to documentation. More specifically, reproducible research refers to the idea that scientific papers should be augmented with the computer code used to carry out the research [20]. Examples of recent projects claiming to support both reproducible research and literate programming include the IPython project [19] and Emacs Org-mode [20].

IPython with Jupyter notebooks

As explained in the introduction, IPython (together with Jupyter notebooks) supports both reproducible research and literate programming. IPython was partially inspired by other scientific tools already offering notebook-like functionality, such as Matlab or Mathematica. Since its inception, the project has been split off into IPython, which provides an interactive REPL and a kernel that runs the user's code, and Jupyter notebooks, which provide the notebook format and web application. Like Swift playgrounds, Jupyter notebooks allow for REPL-style interactive editing; documentation and code can be edited live and blocks of code can be reevaluated, printing their updated results. Jupyter notebooks also allow for more complex graphical elements such as 3D-plots. See figure C.2 for an example of an IPython notebook.

As explained, IPython and Jupyter notebooks have become more or less separate projects, to the extent that Jupyter notebooks can use several kernels. Nowadays there are kernels for over 40 languages that can be used in these notebooks. This illustrates that in Python's case literate programming is more or less an extension to the interactive IPython REPL. Since Jupyter notebooks reuse the IPython REPL, the execution model used for Jupyter notebooks is essentially the same as it is for IPython [9].

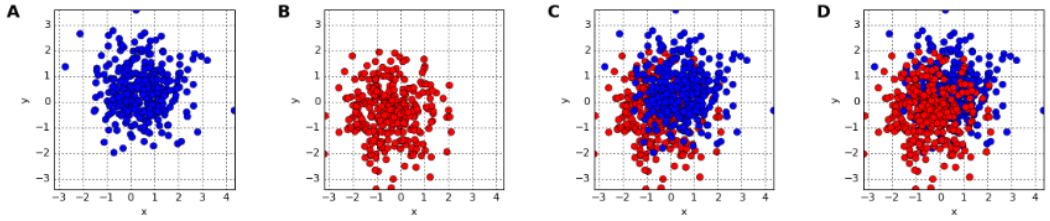
1. Overplotting¶

Let's consider plotting some 2D data points that come from two separate categories, here plotted as blue and red in **A** and **B** below. When the two categories are overlaid, the appearance of the result can be very different depending on which one is plotted first:

In [2]:
```
def blues_reds(offset=0.5,pts=300):
    blues = (np.random.normal( offset,size=pts), np.random.normal( offset,size=pts), -1*np.ones((pts)))
    reds  = (np.random.normal(-offset,size=pts), np.random.normal(-offset,size=pts),  1*np.ones((pts)))
    return hv.Points(blues, vdims=['c']), hv.Points(reds, vdims=['c'])

blues,reds = blues_reds()
blues + reds + reds*blues + blues*reds
```

Output: Out[2]:



Plots **C** and **D** shown the same distribution of points, yet they give a very different impression of which category is more common, which can lead to incorrect decisions based on this data. Of course, both are equally common in this case. The cause for this problem is simply occlusion:

In [3]:
```
hmap = hv.HoloMap({0:blues,0.000001:reds,1:blues,2:reds}, key_dimensions=['level'])
hv.Scatter3D(hmap.table(), kdims=['x','y','level'], vdims=['c'])
```

Figure C.2: A plot from data in an IPython notebook.

C.4 Problem Definition

The previous sections provided the necessary background knowledge of the problem domain. Now that the background knowledge has been explained, this section will discuss the problem definition of the client.

Spoofax offers a wide variety of tools to develop DSLs and accompanying IDE support. Precisely because Spoofax is concerned with developing DSLs, rapid prototyping of syntax and grammar is a convenient addition to the product. Appendix C.2 showed that REPLs provide this rapid prototyping ability for other languages like Lisp and Python. Therefore, the client has expressed their interest in a REPL that works with all language definitions created in Spoofax. Such a REPL would aid both the developer of the language and its end-user.

Since a lot of language services are already provided by Spoofax, the REPL should try to hook into existing services as much as possible. Besides exposing already existing Spoofax services in a REPL, the product should also expose additional features geared towards interactive exploration of programs in Spoofax defined languages. Examples include displaying and editing the contents of current program context, saving and loading program context and keeping a history of executed expressions.

When a REPL is realized in time, the product could be further extended to add sup-

port for literate programming, allowing for rapid prototyping and documentation at the same time [20]. This would allow developers of new DSLs to document and explain their language in an interactive manner that directly allows experimentation.

C.5 Problem Analysis

This section gives an analysis of the problem defined in the previous section. More specifically, this section describes the problems that are expected and that will have to be solved while incorporating a REPL within the larger context of Spoofax.

A language can have many different language constructs. The REPL for Spoofax, however, should not be constrained to any particular language constructs. Rather, it should operate on any defined language. For this reason the implementation and particularly the interaction with the user needs to be carefully considered.

Execution model of the REPL

As illustrated in appendix C.2.3, a major difference in the execution model of a non-interactive interpreter and a REPL is concerned with what an indivisable unit of execution is. The build process as implemented in Eclipse uses a "BuildInputBuilder" to transform and build a (valid) program as a whole. While this is quite a convenient way to quickly create a compiled program; for a REPL to really offer an interactive experience to the user, the REPL should be able to execute smaller units of work than a complete program.

An example implementation that solves this problem is the Java Shell (JShell) REPL [18] currently under development for OpenJDK. The OpenJDK team introduces a concept called "wrapping" to achieve this: *"if X is an input that JShell accepts (as opposed to rejects with error) then there is an A and B such that AXB is a valid program in the Java programming language."*. More specifically, this says that when a user types in an expression like `int x = 2 + 2;`, JShell generates the surrounding class and method body needed for this expression to be valid.

This technique cannot be directly applied to Spoofax, since the language constructs needed to sucessfully "wrap" other constructs are not known in advance. A first approach to solve this problem could be to require that the language designer provides a program template in an esv file, in which they can indicate the various types of language constructs that are valid from within the REPL and where they would be inserted in this template. The REPL should then be able to distinguish the language constructs defined in the esv file as the user types them. A template for a complete paplj program could look like this:

```
program
  $classes
run
  $expressions
```

The illustrated solution could later be extended to eliminate the need for manual specification, by reusing the SDF syntax definitions for the language: given a partial program as typed in by the user, the REPL should then try to determine whether it is valid somewhere within some syntactically valid AST resulting from the language definition. By doing a breadth-first search, starting from the context-free start symbols of the language, the REPL could prepend symbols from the syntax definition until the program is valid.

When a compiler or an interpreter is invoked, the program is a complete and static unit of source code that will not change during execution. In contrast, when a REPL is invoked, often the user wants to alter or add code to previously executed expressions. This means the REPL needs to form "programs" in memory by combining all the entered expressions. Thus the REPL should have a way of combining previously entered expressions to form a program. JShell does this by keeping the wrapped source and generated class files in memory and importing previously defined classes [18]. The concept of imports is however very language specific and it cannot be assumed to exist. Therefore, another way of combining entered expressions into one program needs to be found.

Detecting unfinished expressions for multiline editing

To support multiline editing, the REPL should detect that the expression or statement is unfinished when the user presses the "Return" key for the next line, instead of trying to parse and execute it. An obvious part of Spoofax that is relevant for this problem, is the syntax definition in SDF3 (see appendix C.1.1).

Language specific additional commands

Another problem is when some additional command can be useful for a REPL of a particular language, but would not make any sense within the context of other languages. For example, some languages such as Python allow for loading modules, which brings all of the definitions inside of these modules into scope. An additional command to load a module could in that case be useful, but would not make any sense in languages that have no concept of definitions that can be imported.

This problem could be solved by the language designer by extending their language with reflective capabilities. However, the language designer might not want to extend the language with reflective capabilities outside of the context of a REPL, for example when the language is a DSL. It is clear therefore that a different approach should be considered.

One possible solution is to allow for language specific configurations that are loaded with the language definition of that language. This can be done by extending the editor services discussed in appendix C.1.5 with REPL commands. Similar to menu actions, where one can define menu buttons such as "Run" to run a program, one may then even bind the "load-file" command to an action.

Redefining terms bound to names

When the user is prototyping methods inside the REPL, they would likely want to be able to redefine that method to be of a different implementation. However, this poses a similar problem as in the previous section: for some languages it may not be possible nor desirable to do so outside of the context of a REPL. Thus requiring the language designer to extend the language with such abilities is again an inadequate solution.

One could propose the same solution as in the previous section, namely to define an additional command for redefining a class or method. Another possible and maybe more adequate solution for redefining a term bound to a name, is to allow the user to give the name and the new term. The name can then be used to find the old term, so that it can be replaced with the new one.

It should be noted, however, that implementing this can slightly change the semantics of the language when it is run inside the REPL.

C.6 Requirements Analysis

Defining requirements upfront is important for several reasons: it is a contract between the developers and the client, it guides the product development, it enables the client to track the progress and finally it allows validation of the deliverable. The requirements are listed in this section.

C.6.1 Design goals

The client has expressed some high-level requirements, which are listed in this section as design goals in order of priority. The design goals serve as an important guideline when defining and implementing the invidual requirements. As such, they can be considered the bounds within which all requirements must fit.

Language-agnostic The REPL should not make any assumptions about its host language: it must work with all languages defined with Spoofax. This also means that a language designer should not have to do any additional work to get a REPL for their language. However, if required, the language designer should be able to provide REPL-specific configuration.

Autogeneration The REPL should be automatically provided, just like all of Spoofax's other services. This means that providing access to the REPL should be integrated into Spoofax's build system and not require any additional steps from the user.

Maintainability Spoofax is an already existing, open source project managed by several people. When the product is delivered, these people will take over the maintainership. Therefore, it is important that the code is maintainable. This means that the code should be well-documented, with low coupling and high cohesion in the code's modules.

IDE-agnostic Current stable releases of Spoofax integrate solely with Eclipse. An effort is underway to make Spoofax IDE-agnostic and to provide separate modules to tie Spoofax to IDEs. The REPL should keep this in mind from the start and not tie itself to any IDE.

Performance Spoofax's developers already focus quite a bit on performance: both the generation and the use of the services are performant. This should be no different for the REPL.

Modify Spoofax's existing codebase as little as possible The product should be an extension to Spoofax, which means the changes made to the existing Spoofax codebase should be as small as possible. Preferably, the REPL service should be a standalone module.

C.6.2 Requirements

Under the guidance of the design goals listed in the previous section, the requirements compiled from the feature matrix discussed in appendix C.2 and meetings with the client are discussed below using the MoSCoW method.

Must have

Requirements listed under "must have" are of critical importance to the usability and success of the deliverable. Without these, the product is not in a workable state and is not likely to be accepted by the client. *Must* can also be considered an acronym for the Minimum Usable SubseT.

Interactive REPL Per the definition of a REPL given in appendix C.2, the REPL has to be interactive. It should evaluate single statements and expressions typed in by the user and print the results back to them.

Works with any language defined in Spoofax Every service in Spoofax operates from language definitions. It is evident that the REPL should work with these definitions if it is to fit within Spoofax. An optional feature is to allow language-specific configuration and language-specific commands, defined in an esv file.

Input history Users should be able to retrieve previously typed expressions and statements to support the explorative and interactive nature of a REPL.

Automatic binding of previously yielded values In the same vein, previously yielded results should be implicitly bound to automatically generated identifiers to make their values available in future expressions.

Multiline input editing Multiline input editing is a crucial feature for user satisfaction. The input editor should start in single line editing mode and recognise when an expression or statement is part of a multiline construction. When it recognises this, it should automatically switch to its multiline input editing mode, wherein the prompt character indicates this new mode. This mode's behavior (e.g. keyboard shortcuts) is different from the single line editing mode.

Error reporting To support the interactivity of the REPL, error reporting should be available in two ways: while typing an expression or a statement, on-the-fly error reporting should indicate wrongly typed items. The interpreter that evaluates syntactically correct input should also be able to display its error messages to the user in case of an error during the dynamic semantics phase.

Syntax checked expressions Supporting the above requirement, all input should have its syntax checked on the fly.

Syntax highlighting All expressions and statements (whether they are currently being entered, displayed as previously entered input or displayed as previously yieled results) should be syntax highlighted.

Integration with Eclipse As a first implementation, the REPL should integrate with Eclipse to provide an interface to the users.

Should have

Requirements listed under "should-have" are important, but not required for a working product.

Ability to redefine identifiers As explained in appendix C.2.4, REPLs provide the ability to explore unknown problem domains. It is not a far-fetched idea that a developer would want to change function implementations or the types of certain variables. To support this, a REPL should allow users to redefine identifiers. This might mean that the REPL needs different semantics than the language it operates on, contrasting the design goal that the REPL should be language-agnostic.

Environment inspection The exploratory and interactive nature of REPLs calls for the ability to inspect the current environment. This is to replace the files with source code that a developer could otherwise inspect and an initial step towards offering debugging features in the REPL.

Save and load REPL state Often, developers want to save the current state of their IDE and return to it later. As such, the REPL should allow their state to be saved and restored.

Could have

Requirements listed under "could-have" are desirable, but not necessary. These requirements often improve usability or customer satisfaction and are included only if time permits.

Code completion The multiline input editor should ideally function just like an IDE's editor. Semantic code completion is not yet implemented in Spoofax, so the REPL should provide syntactic code completion instead.

Hover over variables to see value, type and others Another step towards debugging would be the ability to hover variables with the mouse in order to inspect their value, type and other known information. The difference between this feature and the previously mentioned environment inspection is that this feature works per variable.

Literate programming As explained in appendix C.3, literate programming offers the advantage that code and documentation go hand in hand. This allows developers of languages in Spoofax to document and illustrate their language simultaneously with the development: documentation and examples can never be outdated, because outdated example code will halt the execution.

Integration with other IDEs (IntelliJ) Generating Spoofax's editor services for IntelliJ is currently a work in progress and it would be nice if the REPL works in IntelliJ from the start.

Won't have

Requirements listed under "won't-have" are not to be implemented during this project. They have been identified as possible features, but are outside of the scope of this project and listed only as possible suggestions for further work.

GDB-style debugging and nested REPLs Spoofax currently does not generate any debugging features, which would be required for the REPL to offer such functionality as a built-in feature. Writing a debugger is outside the scope of this project and thus offering debugging capabilities inside the REPL is something to consider for a later version.

C.6.3 Minimal viable product

The design goals and requirements listed in the previous sections give a good idea of what the deliverable should look like. It is possible, however, that due to unforeseen problems, not all the listed requirements and design goals can be met. It is important to therefore define a minimal subset of the design goals and requirements that *must* be present in the final deliverable: the "must have" requirements have to be implemented, whilst adhering to the first two design goals.

C.7 Frameworks and Tools

The previous section analyzed the requirements for the product as set in appendix C.6. The purpose of this section is to discuss the frameworks and tools used during the development of the product.

C.7.1 Development frameworks

Spoofax is an already existing project. In order for the product to be as easily integrated as possible, most of the tooling used by Spoofax will be reused. This means that the Java programming language is to be used, together with Maven for the build environment and JUnit for unit tests. On top of this, Spoofax uses a few open source Java libraries as can be seen in the list below. It is most likely that these will be used in the product as well.

The only exception to this is the fact that TravisCI is used as opposed to Jenkins, because Jenkins is self-hosted whilst TravisCI is available for free online.

- Guice, a lightweight dependency injection framework;
- Guava, provides general utilities missing in the JDK;
- RxJava, a library for composing asynchronous and event-based programs using observable sequences;
- Apache VFS, a library for accessing various different file systems;
- EhCache, a standards-based cache that boosts performance;
- Apache's logging library.

If new libraries are to be used during the development of the project, it is important that these are under a license compatible with the Apache license under which Spoofax is licensed. Compatible licenses include, but are not limited to, the LGPL, MIT, BSD and Apache licenses.

C.7.2 Development tools

To manage the source code, the git version control system is used. Pull-based development [3] is the paradigm chosen, in order for each member of the team to work on their assigned tasks without interfering with each other. To facilitate this, the GitHub platform is used because if offers an easy interface to pull-based development: it automatically discovers the commits to be merged, it facilitaties code review and discussion, new commits are automatically added to the pull-request so that the discussion can continue and GitHub automatically verifies whether commits can be merged. On top of this, GitHub also provides an integrated issue tracker.

Furthermore, the tools FindBugs and PMD will be used to assist in delivering quality code. Checkstyle will be used to guarantee a coherent style throughout the code. These three static analysis tools will help ensure the design goal of maintainability, as highlighted in appendix C.6.1.

C.8 Conclusion

The purpose of this research report was to explore the problem domain, to define and analyse the problem which the product is to resolve and to analyse its requirements.

The first three sections explored the problem domain; the Spoofax Language Workbench, read-eval-print loops and literate programming were investigated. These three separate domains come together when working towards a viable product.

The next section defined the problem, which is to bring rapid prototyping and exploratory programming to users who are defining programming languages in Spoofax. This problem was then analyzed in the following section, wherein it was outlined how the product will fit into the existing Spoofax codebase. Several issues were foreseen and discussed together with possible solutions. From the problem definition and analysis, a list of requirements was compiled. These requirements are structured using the MoSCoW method. The “must-haves”, together with outlined design goals, form a minimal viable product. Finally, the last section discussed the development frameworks and tools to be used.

With the research phase now complete, an overview has been formed as to what the product should do and how it should be implemented.

D

Complete UML Diagram of the Final Product

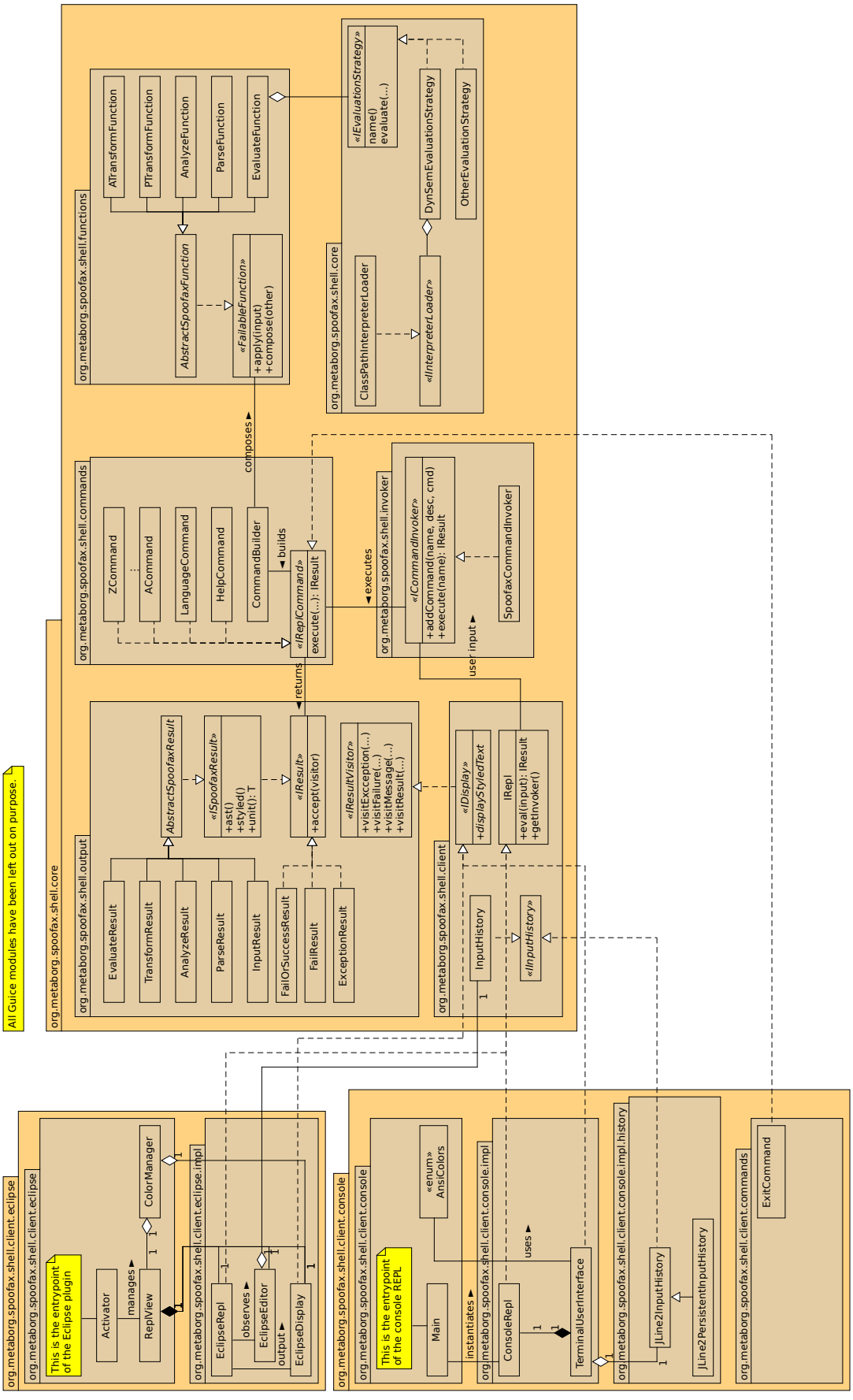


Figure D.1: Complete UML Diagram of the final product.