Redesigning the Spoofax Testing Language

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

V. Lanting
born in Gouda

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl/`

Redesigning the Spoofax Testing Language

Author:	V. Lanting
Student id:	1513273
Email:	volkerlanting@gmail.com
Thesis defense:	29-01-2020

Abstract

The Spoofax Testing Language (SPT) is the existing solution for testing in the Spoofax language workbench. It allows developers of domain specific languages to write their test cases declaratively. As it aims to be implementation agnostic, developers don't need to concern themselves with the details of the artifacts generated by Spoofax, and can write their tests before implementing their language. However, the previous implementation has become slow and unusable for larger test suites and can not be executed programatically. This means it can't be used for continuous integration and automated regression testing. As Spoofax was redesigned to become more robust and platform independent, the previous SPT is no longer compatible. We took this opportunity to redesign SPT.

In this thesis we will discuss the benefits of a testing approach like SPT, how far along it is on the path of testing any language, and what is required to make it usable by modern day developers. We will analyze the problems that SPT had to tackle and how it solved them, and which problems still remain. Finally, we present and evaluate our new design and implementation to solve some of these remaining problems. We created a platform independent, real-time performant, easily extendable architecture that allows SPT to be used for automated tasks such as continuous integration and the automated grading of students' domain specific languages.

Contents

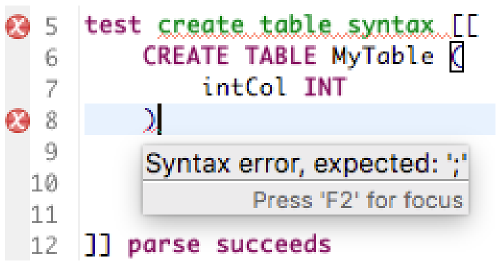
Chapter 1

Introduction

Testing is an important tool to gain confidence in the correctness of a software system. According to a survey by the National Institute of Standards and Technology software failures are estimated to cost the US economy around 59.5 billion dollars each year [1]. The NIST estimated that 22.2 billion of these losses could be saved by the correct use of existing software testing approaches throughout the development process. This illustrates not only the importance of software testing, but also the importance of making it easier for developers to apply software testing approaches. For example, by providing tools for writing and executing tests.

The Spoofax Testing Language (SPT) does exactly that for the field of language engineering [2]. It is a domain specific software language (DSL) in which language engineers can declare test cases for the software language they are developing. By focusing on the field of language engineering, SPT can offer abstractions specific to the testing of software languages. These domain specific abstractions greatly reduce the effort of writing test cases and enable the use of test driven development [3].

Figure 1.1 shows a test case for the syntax of the MiniSQL language. It illustrates some of the powerful abstractions of SPT. First, the program that is being tested is not just treated as text, but is a first class citizen of the language, allowing us to show syntax highlighting and errors directly in the input program. This representation of an input program is called a fragment in SPT. Second, it illustrates how test cases in SPT are declarative. A simple `parse succeeds` is enough to indicate that the input program should be parsed and the parsing is expected to succeed. SPT provides a predefined set of these test expectations, which can be used to test the different aspects of the language (like its parser). Finally, it shows how tests are executed and the results are displayed directly inside the editor, as the developer is writing the test, shortening the feedback cycle as much as possible. This short feedback cycle and declarative syntax of SPT, make it very suitable for test driven development.



```
test create table syntax [[
    CREATE TABLE MyTable (
        intCol INT
    )


]] parse succeeds
```

Figure 1.1: An example of a test case in SPT.

SPT is one of the tools offered by the Spoofax language workbench [4]. A language workbench is a collection of tools for the development of a software language, which are integrated into a single environment [5]. Language workbenches are basically Integrated Development Environments (IDE) specifically made for the development of software languages. They allow the creation of a software language implementation with significantly less code, improving the maintainability of the implementation [6].

As Spoofax was being used to develop more languages, new use cases and requirements arose for Spoofax and its tools. In response to this, the core of the Spoofax language workbench has been rewritten to be more robust and to break away from its tight coupling with the Eclipse IDE to become platform independent. This major change meant that many of the tools provided by Spoofax (including SPT) had to be ported to the new implementation.

The previous implementation of SPT is tightly coupled with both the old Spoofax internals and Eclipse. Due to this, porting it would require quite a lot of changes. At the same time, new use cases have emerged for SPT, which its previous design does not support. These new use cases are integrating SPT in a continuous integration system, and using SPT test suites for automated grading of student submissions for a compiler construction course, where students learn to make their own language implementation. As the previous SPT does not allow programmatic execution of tests, these use cases are not yet supported. There is also a significant performance degradation as test suites get larger, and the implementation does not allow easy addition of new test expectations. Therefore, we decided to use this opportunity to reevaluate SPT and identify its shortcomings, so a new version could be created to tackle these problems and support the new use cases.

In this thesis we will analyze the previous implementation of SPT to find the main problems, and analyze the new use cases to create a list of requirements. Then, we propose a new design that:

- works with the new Spoofax,
- does not suffer from the same performance problem,
- allows the easy addition of new test expectations,

- allows programmatic execution of test cases,
- and meets the requirements for the new use cases.

We implemented this new design and evaluated it by using it for both existing and new use cases. The new implementation was used for test driven development of the MiniSQL language. It was integrated into the Maven build system to allow its use during continuous integration, and it was used to create an automated grading system for the Compiler Construction course at the Delft University of Technology. Finally, we benchmarked the execution time of test suites of different lengths and compared those against the execution times of the previous SPT.

The rest of this thesis is outlined as followed. In chapter 2 we provide more background information about the field of language engineering, language workbenches and specifically Spoofax, and the ideas behind and benefits of SPT. Then, we move on to the analysis of the previous implementation of SPT in chapter 3, where we give an overview of the most pressing problems and present a list of requirements for the new design. In chapter 4 we present our new proposed design and discuss how it tackles the problems and meets the requirements outlined in chapter 3. The implementation of this new design will be evaluated in chapter 5, where we will briefly discuss how it was used for all use cases discussed in chapter 3.3 and evaluate if it meets the requirements. Next, we'll discuss how this work fits in the current landscape in chapter 6, and in chapter 7 we will discuss what more can be done to improve SPT.

Chapter 2

Background

2.1 Testing in Language Engineering

The field of language engineering is concerned with the creation of computer languages, languages designed to communicate instructions to a computer. Examples of computer languages are programming languages like C, Java, or JavaScript, but also query languages like SQL, and markup languages like HTML.

When talking about languages, we make a distinction between the language specification and the language implementation. Most languages have a textual specification which describes the syntax of the language (i.e. what you are allowed to write) and the semantics of a language (i.e. what it means). However, to execute programs written in the language a software system is required. This system is called the language implementation. The relation between the specification and implementation is illustrated in figure 2.1. It usually consists of some, or all of the following components:

- Parsing – the parser transforms the program from its initial representation, usually a textual representation, to a format that can more easily be reasoned about. Usually the output is some form of Abstract Syntax Tree (AST), a tree representation of the syntactical information of the program. Parsing checks if the syntax of a program is valid.
- Analysis – after parsing, the AST can be used to reason about the static semantics of the program. This includes things like generating errors when a variable is used before it is defined, and type analysis. Analysis checks the static semantics of the program.
- Transformation – if a program has valid syntax and static semantics, it can be transformed to a different format. For example, a compiler would transform the AST to machine code that can be executed on the computer. A transpiler would transform the AST to a representation of another programming language.

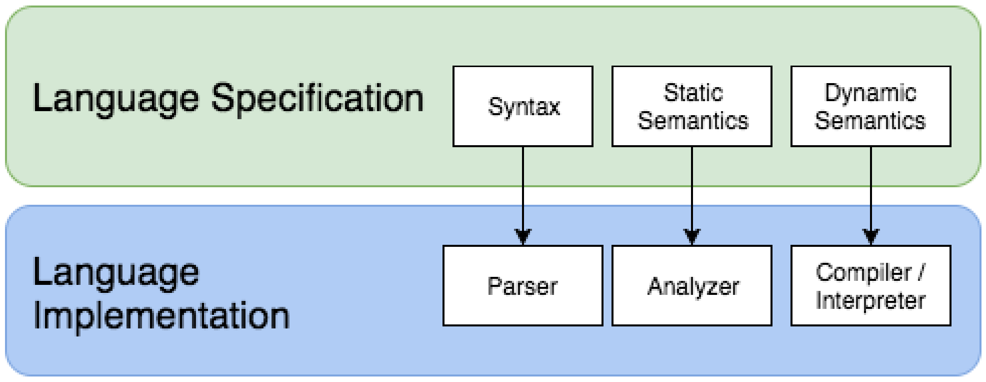


Figure 2.1: The language specification describes the behaviour of the aspects of a language, like what constitutes a valid program and what should happen when it is executed. There can be any number of implementations that provide this behaviour.

Any errors in the language implementation can affect all programs that are executed with it, so the proper testing of a language implementation is crucial. To assist with the creation of the language implementation and its tests, we need proper tool support.

There is already extensive support for the development of generic software systems in the form of Integrated Development Environments (IDEs). An IDE integrates a set of tools into a single environment, allowing a developer to enjoy their benefits in an intuitive way. This same principle is being applied to the development of languages. Both new and existing tools for language development are integrated into a single environment (usually into an existing IDE) for the language developer. Such an environment is called a Language Workbench. Examples of language workbenches are XText and Spoofax [4]. These language workbenches allow the developer to formally specify (parts of) the language specification, and will generate (parts of) the language implementation from this specification.

When testing languages, this distinction between the language specification and the language implementation is important (see figure 2.1). The intended behaviour of the language is specified in the specification, and the implementation should implement this behaviour. This means that any tests concerned with the observable behaviour of the language (so called black-box tests [7]), can be written against the specification, instead of against the implementation. This has two main benefits. First, it abstracts over the implementation specifics, making it faster and less complex to write the tests. This is especially important when the tools of a language workbench are used to create the language. These tools often generate artifacts for the language implementation, like how a parser generator generates a parser from a grammar specification. This means the language engineer writes the specification, not the implementation. Therefore, writing tests that require knowledge about the generated artifacts is a bigger hurdle than when the implementation is written by hand.

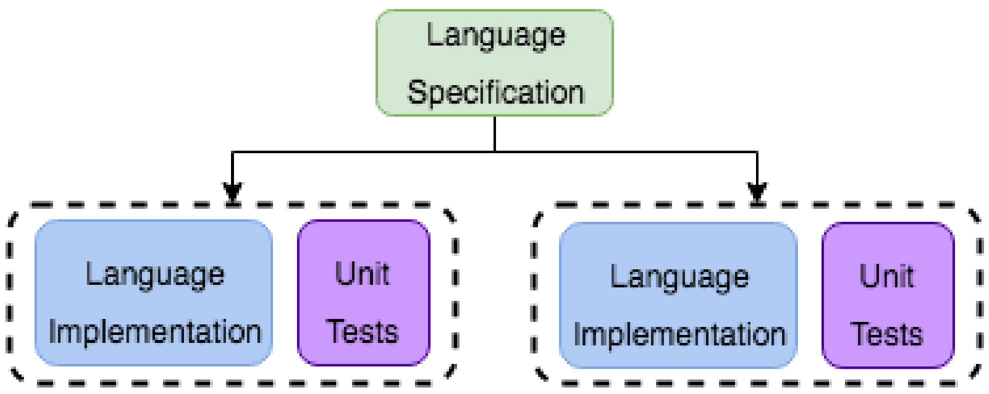
Second, it allows the sharing of test cases. Let's look at an example: the EcmaScript standard is the specification for the JavaScript language. Some well known language implementations for this language are the V8 and SpiderMonkey JavaScript engines. When multiple implementations of the same specification are created, they all have to be tested. In this case we would like to reuse as many tests as possible, with the least amount of effort.

When it comes to testing of generic software systems, one of the main tools is a unit testing framework. There are unit testing frameworks for most general purpose languages, allowing developers to write tests for software written in that specific general purpose programming language (GPL). These frameworks are often based on the xUnit family of frameworks [8], which provides a general architecture for unit testing frameworks. Keeping the terminology and general architecture of frameworks consistent across the implementations for different GPLs makes it easier for developers to start writing tests if they are already familiar with a different framework of the xUnit family. Unit tests written in such a framework can only be used to test the behaviour of a system written in the programming language supported by the framework. For software systems written in multiple programming languages, multiple unit testing frameworks would have to be used.

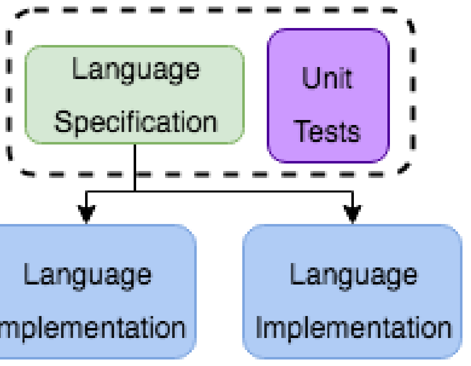
Using this standard approach to unit testing, we would test against the implementation. So each language implementation would have their own test suite, using the unit testing framework for the language in which the language implementation was written. Testing against the specification, instead of the implementation, would result in a test suite per language specification, that can be reused for all language implementations of the language. The difference between these testing setups is illustrated in figure 2.2.

The problem is how to abstract over the language implementation and write tests against a language specification. One solution is to write the tests in the language that is being tested. For example, the Ecma TC39 committee has a test suite of test cases written in JavaScript that can be used to test JavaScript Engines [9]. This approach only works if the language is expressive enough to test itself, which is not the case for many domain specific languages (DSLs), especially declarative ones. Another limitation of this approach is that it generally does not allow negative test cases, like verifying that wrong syntax is indeed rejected, as that would result in non-compiling tests. This problem can be solved by creating a program to execute these tests and check if negative tests actually failed, but such a program would need to be created for each language implementation.

In [2] a solution to this problem was proposed: the use of a declarative testing language, called a Language Parametric Testing Language (LPTL), that allows the user to specify black-box test cases for the domain of language



(a) Traditional unit testing approach.



(b) Unit testing at the level of the language specification.

Figure 2.2: Unit testing setups for testing languages.

engineering. It is called language parametric, because it can be used to write tests for any textual language. The proposed LPTL comes with a range of tools and abstractions to ease the creation of unit tests. For example, the programs that serve as test inputs are treated as first class members of the LPTL, allowing syntax highlighting and content completion within these program texts, and intuitive selection of elements within the program text.

A subset of this LPTL was implemented for the Spoofax language workbench. This implementation was called the Spoofax Testing Language (SPT). In the next section we will give more background information about the Spoofax language workbench. After that, we will go into more detail about SPT and its benefits.

2.2 The Spoofax Language Workbench

The implementation of SPT as described in [2] was used as a tool to facilitate testing of languages created with the Spoofax language workbench [4]. Before we discuss this version of SPT and its benefits, we will first give an overview of what Spoofax is and what if offers.

Like any language workbench, Spoofax offers a set of tools for the development of the different components of a language implementation, and integrates them into a single environment. The environment of choice for Spoofax is the Eclipse IDE. The tools that Spoofax offers are a set of declarative DSLs in which the language developer can express each component of their language. These DSLs are each specific to a specific domain within language engineering. For example, the Syntax Definition Formalism (SDF) [10] can be used to express the syntax of the language, and the Name Binding Language (NaBL) [11] can be used to specify the scoping and reference resolution rules for the analysis of the language.

By using these DSLs, the language developer can specify each component of their language in a declarative way, without having to deal with the details of how it is implemented. This approach shields the developer from a lot of the accidental complexity involved in the creation of a language implementation, and allows them to focus on what the language should be doing. The specifications written in these DSLs together describe the language that is being developed, and are called the language definition. From the language definition, artifacts are generated, resulting in the language product. Spoofax can then use the artifacts of the language product to provide the services they describe, like a parser, a compiler, or an editor providing syntax highlighting and code completion. This combination of the language product and the Spoofax framework forms the language implementation. For example, the SDF grammar specification is part of the language definition. It produces a parse table as part of the language product. The parse table can then be used by Spoofax's parser framework to parse programs of the language. The combination of the parser framework and the parse table is part of the language implementation. These concepts are illustrated in figure 2.3.

Spoofax offers editors for all its DSLs, with editor services like syntax highlighting, content completion, and real-time error reporting inside the editor. We will now give a quick example of how Spoofax's DSLs can be used to develop a language. To illustrate how a language can be developed, we will use the development of MiniSQL as an example. MiniSQL is a small subset of the SQL query language. We will discuss how each aspect of the language (parsing, analysis, transformation, and editor services) can be specified in Spoofax.

Parsing

To specify the syntax of a language, the Syntax Definition Formalism (SDF) [10] can be used. Let's look at the grammar definition of our example language MiniSQL. It allows the definition of tables using the `CREATE TABLE` statement,

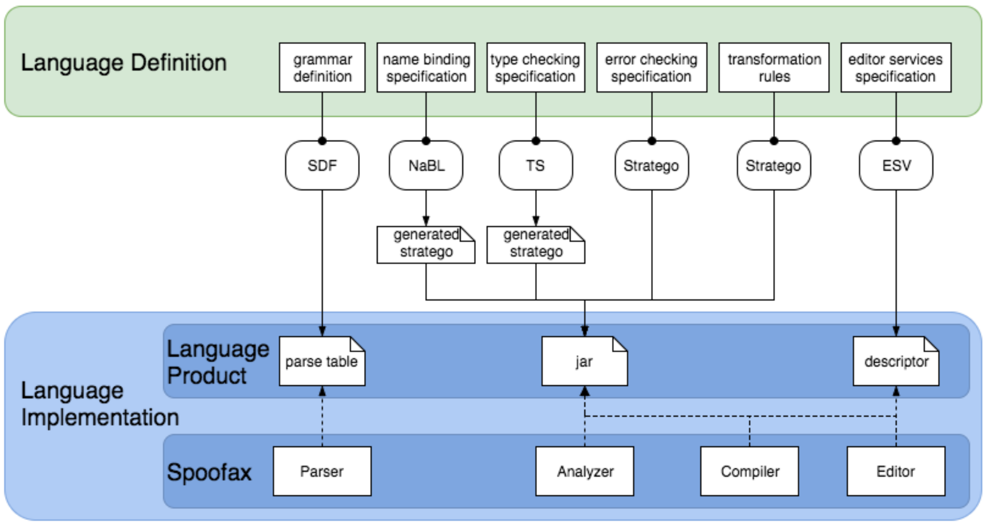


Figure 2.3: A language definition in Spoofax is written using several DSLs. From this language definition, artifacts are generated that, together with Spoofax, form the language implementation.

and simple queries with a `SELECT`, `FROM`, and optional `WHERE` clause. An example program of this language would look like this:

```
CREATE TABLE Tbl (
    intCol int,
    strCol VARCHAR
);

SELECT t.intCol
FROM Tbl t
WHERE t.intCol = 5 AND t.strCol = "some string";
```

The grammar definition for this language is written in SDF3 using production rules and templates. Production rules specify what program text each non terminal can produce. If a program can be produced from any of the language's starting non terminals it is a valid program. Apart from the syntax of the language, the production rules also specify what the abstract syntax tree (AST) that results from parsing should look like. Each production rule starts with the name of the sort (non terminal) followed by a dot and the name of the constructor of the AST node that should be created for this production. Here is the production rule for the `CREATE TABLE` statement of our language:

```
Statement.TableDef = <
    CREATE TABLE <ID> (
        <{ColDef ",\n"}*>
```

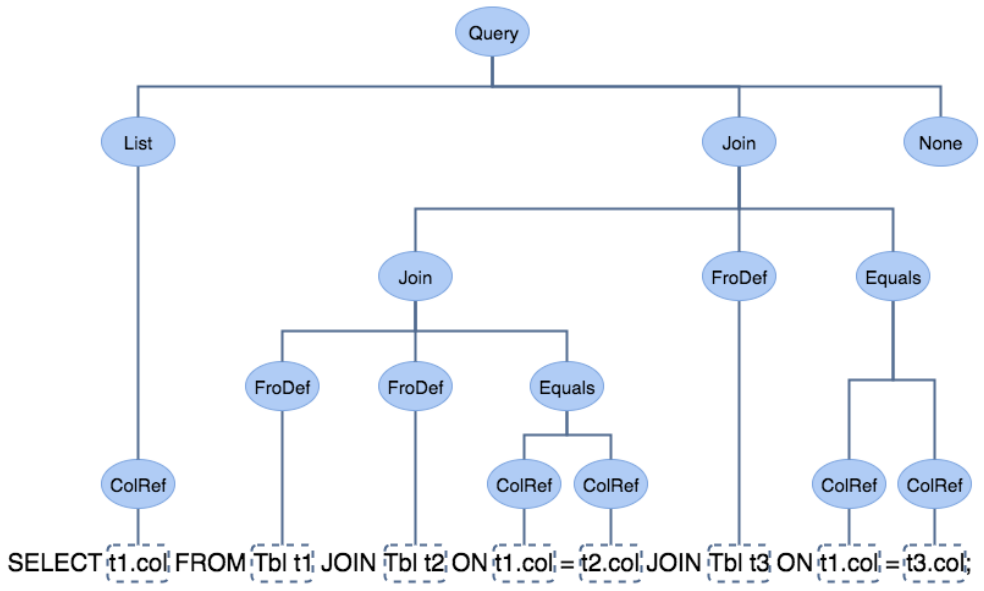

Query
List
Join
None
Join
FroDef
Equals
FroDef
FroDef
Equals
ColRef
ColRef
ColRef
ColRef
ColRef
SELECT t1.col FROM Tbl t1 JOIN Tbl t2 ON t1.col = t2.col JOIN Tbl t3 ON t1.col = t3.col;

Figure 2.4: The AST of a sample MiniSQL statement.

```
    );
>
```

This rule specifies that the sort (non terminal) `Statement` can produce the literal strings (keywords) `CREATE` and `TABLE`, separated by whitespace (called layout), followed by the sort `ID` and any number of `ColDef` sorts, separated by a comma. When a string like this is found in a program, it will be parsed to a `TableDef(id, coldefs)` AST node, where the first child is the `ID` node representing the name of the table, and the second child a list of `ColDef` nodes representing the column definitions of this table.

From production rules like these a parse table is generated that allows Spoofax's parser to parse a program written in our language to an AST. An example of such an AST is given in figure 2.4.

Analysis

After a program has been parsed to an AST, it can be analyzed. The analyzer checks the AST and returns any error messages, warnings, or notes that should be produced. The specification from which this analyzer will be generated usually consists of three parts: name analysis, type analysis, and any other analysis.

The name analysis specification specifies what kind of named entities exist in the language, what constructs define them, how they are scoped, and what constructs refer to them. MiniSQL has three different kinds of named entities: tables, columns, and fro's. Tables and columns are quite self explanatory. A fro is the entity represented by an expression in the `FROM` clause of a query. In normal SQL this could be a view, a reference to a table, or a table alias. MiniSQL only allows table aliases.

To specify where these Tables, Columns, and Fro's are defined, we can use the Name Binding Language (NaBL) [11].

```
    TableDef(tname, coldefs) :
        defines Table tname
        scopes Column

    ColDef(cname, typeref) :
        defines Column cname of type typeref

    FroDef(tname, alias) :
        defines Fro alias
        imports Column from Table tname
        scopes Column
        refers to Table tname
```

These rules specify that if a `TableDef(tname, coldefs)` node is found in the AST, it will be a definition site of a Table with the name in the `tname` node, and it creates a new scope for Columns. When a `ColDef(cname, typeref)` node is found, it will be a definition site for a Column. We can also specify any properties of this column like its type. These types can be used later in the type analysis specification. The final rule specifies that if a `FroDef(tname, alias)` node is found, representing a table alias in the `FROM` clause of a query, it is a definition site of a Fro with the name of the node `alias`. It also specifies that this Fro creates a scope for columns and imports all columns from the scope of the table with the name of the node `tname`. Finally, it also specifies that the `tname` node is a use site of the Table with the name of the `tname` node.

From these name analysis specification rules, Stratego code will be generated. When unresolved references or duplicate definitions are found, an error message will be generated by this code. This generated Stratego code is included in the generated code for the analyzer (see figure 2.3).

The second part of the analysis specification is type analysis. Type analysis can be specified using the Type System language (TS).

```
ColRef(tname, cname) : ctype
    where definition of cname : ctype

Int(i) : INT()

String(str) : VARCHAR()
```

```
Equals(e1, e2) : BOOL()
    where
        e1 : e1type
        and e2 : e2type
        and e1type == e2type
        else error $[Expected [e1type], not [e2type]] on e2
```

The first rule specifies that a `ColRef(tname, cname)` node has the same type as the definition that the `cname` node refers to. This links in with the name analysis to find the Column it refers to and get the type that was recorded for that definition. Next we specify the types for ints and strings. Finally we specify the type of the equals expression, and generate an error message on the `e2` node if the types on both sides of the equals sign don't match. For example, when comparing a string and an int typed column.

Just like the NaBL code, TS will generate Stratego code that can be included in the analyzer.

The final part of the analysis specification is written in Stratego [12]. For example, we can generate a warning on the name of a table if it doesn't start with a capital letter.

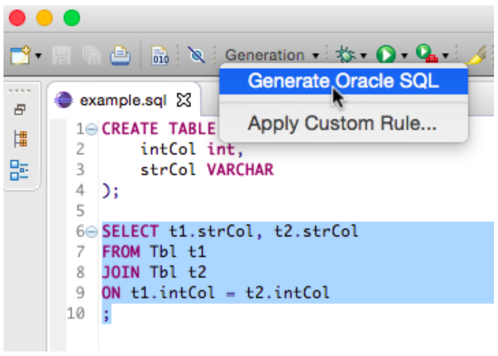
```
constraint-warning:
    TableDef(tname, cols) ->
        (tname, "Table names usually start with a capital.")
    where
        [first-char | other-chars] := <explode-string> tname;
        <is-lower> first-char
```

A Stratego rule matches an AST term on the left of the arrow and its result on the right of the arrow. Additional computation can be done in the where clause. The analysis framework of Spoofax will then traverse the AST and call these Stratego rules to collect all errors, warnings, and notes. As the example illustrates, Stratego is not as declarative as TS and NaBL. In return, this allows the developer to implement any analysis they want.

The Stratego code generated by the NaBL and TS specifications, and the manually written Stratego code are used by Spoofax's analysis framework to process the AST and gather all error messages, warnings, and notes. Together, the framework and the Stratego files form the analyzer that is part of the language implementation.

Transformations

The next part of the language specification are the transformation rules. These rules can be used to transform the program into something else. For example, transformation rules can be used to transform the `ANSI SQL JOIN ON`



```
CREATE TABLE
    intCol int,
    strCol VARCHAR
);

SELECT t1.strCol, t2.strCol
FROM Tbl t1
JOIN Tbl t2
ON t1.intCol = t2.intCol
;
```

Figure 2.5: An example of a menu entry in the editor.

expressions of MiniSQL to Oracle `SQL` style predicates in the `WHERE` clause. Transformations are implemented with Stratego rules.

Editor Services

The final part of a language specification are the specifications for several editor services. These specifications are written in ESV, and are used by the editor created by Spoofax to initialize the editor's services.

An example of such an editor service is a menu entry. Menu entries can be specified using ESV.

```
menu: "Generation" (openeditor) (realtime)

  action: "Generate Oracle SQL"  = gen-ora-style-builder
```

This specification means the editor will contain a menu labeled 'Generation', with a menu entry called 'Generate Oracle SQL'. When the menu entry is pressed, the Stratego rule `gen-ora-style-builder` will be executed on the selected part of the AST. Figure 2.5 shows an example menu entry for this transformation.

Other editor services that can be specified through ESV files are syntax coloring, content completion, code folding, and an outline view. Spoofax will generate sensical defaults for these services, so the language developer does not have to specify them.

2.3 The Spoofax Testing Language

Now that we have seen how Spoofax offers DSLs for each component of a language, it should come as no surprise that their testing solution is also a DSL. The Spoofax Testing language (SPT) is a DSL that can be used to specify test cases for the behavior of a language [2]. Just like the other DSLs that are a part of Spoofax, SPT offers domain specific abstractions to increase the expressiveness and productivity of the language developer. The domain of SPT is black-box testing of textual languages. In black-box testing (or behavioural testing [7]) the behaviour of an object is tested. A black-box test consists of an object under test, an input for that object, and the expected output. It feeds the input to the object under test and compares the obtained output with the expected output. The advantage of black-box testing is that it only requires knowledge of the requirements of the object, not of the actual implementation details. As a language developer using Spoofax only wrote the language definition, from which a language implementation was generated, it is a great benefit to be able to write tests without needing to know any details about the generated language implementation.

Spoofax has an editor for SPT test specifications which offers editor services like syntax highlighting, content completion, and real-time error reporting. The tests that the developer specifies in SPT are executed as the test is being written. Any failing test cases are marked with an error. This was done to make the cycle of test driven development as short as possible [3]. In test driven development, the developer first writes tests for a new feature. They then watch them fail, to ensure that the tests do indeed capture the new behaviour introduced by the new feature. Only after this will the feature actually be implemented. After the implementation is complete, the developer reruns the tests to check if they do indeed pass. If so, they now have more confidence in the correctness of the implementation. By executing the tests as they are being written and reporting the errors in real-time SPT facilitates this process.

In this chapter we will give an overview of the previous SPT, as it was used in Spoofax versions below 2.0, before the major rewrite of the Spoofax architecture.

Test Cases

A test case in SPT corresponds to a subtest as described in [7]. It is the smallest unit of testing and consists of the component under test, the initial state for that component, the input, and the expected output. In this section we will discuss how SPT uses domain specific abstractions to make it easier to specify test cases.

The behavior of a language depends on the program that it gets as input. Therefore, when designing black-box tests for a language, the input of a test case is always a program text. This program is either faulty and the test should ensure that the fault is discovered (a negative test case), or it is a correct program and the test should ensure that it is not erroneously marked as faulty (a positive test case). We call such an input program text a *fragment*.

Let's look at an example. Throughout this thesis we will be using examples of test cases for the MiniSQL language. MiniSQL is a very small subset of SQL that only allows the creation of tables with integer (`INT`) and string

```
test table creation [[
  CREATE TABLE T (
    intCol INT
  );
]] parse succeeds
```

(a) A positive test case for the syntax of the `create table` statement.

```
test mandatory whitespace [[
  CREATETABLE T (
    intCol INT
  );
]] parse fails
```

(b) A negative test case to ensure that whitespace is mandatory between `CREATE` and `TABLE`.

Figure 2.6: Example test cases for the syntax of the MiniSQL language.

(`VARCHAR`)columns, and simple select, project, join queries. An example of both a positive and a negative test case for the syntax of the `create table` statement of MiniSQL is shown in figure 2.6. These test cases illustrate some of the basic abstractions that SPT offers.

The expected output of a test case depends on the component of the language that is being tested. For example, given a certain program, a parser will produce an abstract syntax tree, an analyser can produce semantic error- and warning messages, and a runtime can execute the program to obtain some result. As the format of the expected output depends on the component under test, SPT combines the two in a so called *test expectation*. In the example of figure 2.6 the `parse` expectation is used. It declares that the parser is the component that should be tested. It is followed by either `succeeds`, to indicate that the expected output of this test is a successful parse (i.e. no errors occurred during parsing), or `fails`, to indicate that the expected output of the parser should contain at least 1 error message.

The syntax of a test case in SPT is as follows: A test case begins with the `test` keyword, which is followed by the *description* of the test case, the input *fragment*, and one or more test *expectations*. We will first describe how the input of a test case can be specified in a fragment, and what the benefits of fragments are. Then, we will discuss the set of test expectations offered by SPT and how they can be used to specify the component under test and the expected output. Finally, we will discuss the initial state for the component under test.

Fragments

As discussed above, the input of a black-box test for a language is always a program, written in that language. In generic testing frameworks, the program would be represented as text (e.g., a string). However, because SPT is domain specific, we can treat the fragment as an actual program, and invoke the editor services of the language under test if they are available. This means that the full range of editor services (e.g., syntax highlighting or content completion)

```
test resolution of tablenames [[
  CREATE TABLE [[MyTable]] (intCol INT);
  SELECT t.intCol FROM [[MyTable]] t;
]] resolve #2 to #1
```

Figure 2.7: A test to check if the reference `MyTable` in the `FROM` clause resolves to the correct table definition.

could be available to the developer as they are writing the program text. Also, any errors or warnings produced by failing test cases can be shown directly inside the program text to make it easier to discover why a test failed. This makes the creation of programs as test inputs just as easy as writing the program in the actual editor for the language under test.

An example of how fragments allow the use of editor services can be seen in figure 1.1. This is an example of an erroneous test case. The developer wanted to create a positive test case for the syntax of the `CREATE TABLE` statement, but forgot the semicolon at the end of the statement. Because SPT executes tests as they are being written it is immediately visible that the test fails (the red error on the test description) and because of fragments it is possible to report the error in the input program text, allowing the placement of an error on the missing semicolon.

The previous SPT only invokes the parser and analysis of the language under test. It does not invoke any of the other possible editor services of the language under test. It does, however, support a predefined syntax highlighting on the program text of the fragment. The syntax highlighting on the fragment is specified by SPT itself, based on token information supplied by the parser of the language under test, so it uses the same highlighting as the rest of the SPT code.

For some tests, a bit more information is required than just the program text. For example, to test if a reference resolves to the correct definition, we need to know which reference should be resolved and which definition it should resolve to. To accomplish this SPT allows the *selection* of parts of the program within a fragment. To select a part of the fragment, the same delimiters are used as those that delimit the fragment (usually `[[` and `]]`). Selections can then be referred to by the order in which they appear in the fragment. They are numbered from left to right, from top to bottom, starting at 1. An example of the use of selections in a fragment to test name resolution for the MiniSQL language can be seen in figure 2.7. Here we declare that we expect the reference at the second selection to resolve to the definition at the first selection.

Test Expectations

The expectations of a test case specify the component under test, and what the expected output is. The set of test expectations supported by SPT is grouped based on components that are common to most language implementations. For example, we've already seen the `parse` expectations which are used to test the parser, and very briefly touched upon the `resolve` expectation which is used to test the name analysis.

We will now give an overview of all the expectations offered by SPT.

Testing the Parser

To test the syntax of the language under test SPT invokes its parser. All test expectations related to testing the parser begin with `parse`. They only differ in the expected output. SPT expects a parser to produce an AST if the input program has valid syntax, or one or more errors if it does not. We've already seen the most basic way to test the syntax of an input program in figure 2.6: `parse succeeds` and `parse fails`. When the parser produces one or more errors, the parsing is considered to have failed. If it produces no errors, it is considered to have succeeded.

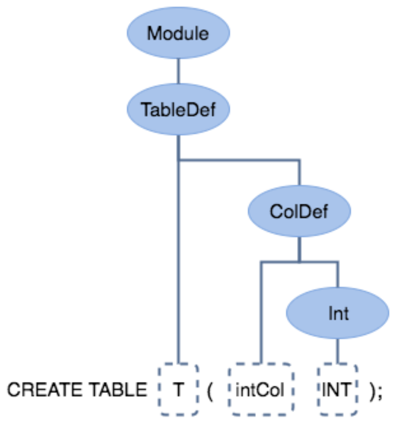
In case of a successful parse, the developer may want to reason about the structure of the produced AST. To do so SPT offers 3 more ways to express the expected output. The first way is to express exactly what the AST should look like. To stay implementation independent the Annotated Terms (ATerms) format [13] can be used. The ATerms format offers a platform- and language independent way to represent tree-like data structures. An example of how the ATerm format can be used to specify an expected AST can be seen in figure 2.8. When specifying expected output as an ATerm, an underscore can be used in place of any node as a wildcard, to indicate that you don't care about the output in that specific part of the data. This is useful when you don't know, or don't care about, the exact value of a part of the data. For example when testing the output of a transformation that generates unique names which can't be known beforehand.

As the input program gets bigger or more complicated, the AST can get quite big. The second way of expressing the expected output was made to tackle this problem. The `parse to file <filename>` test expectation can be used to separate the ATerm representation of the expected AST from the rest of the SPT file. When evaluating this expectation, the ATerm representation will be read from the specified file and compared to the output of the test.

The third way to express the expected output is to use an output fragment. Just like the input fragment of a test case, an output fragment is a piece of program text written in a different language, for which SPT will try to provide editor services. This type of expected output is also known as concrete syntax, as opposed to abstract syntax that is expressed using ATerms. At the end of

```
test produce the given AST [[
  CREATE TABLE T(intCol INT);
]] parse to Module([
  TableDef(
    "T",
    [ColDef("intCol", Int())]
  )
])
```

(a) Check if the fragment parses to the given AST in ATerm format.



(b) Graphical representation of the AST.

Figure 2.8: Using the ATerm format to specify the expected output.

```
test precedence [[
  4 + 5 * 6
]] parse to [[
  4 + (5 * 6)
]]
```

(a) Concrete syntax

```
test precedence [[
  4 + 5 * 6
]] parse to Add(
  Int("4"),
  Mul(Int("5"), Int("6"))
)
```

(b) Abstract syntax

Figure 2.9: Using concrete- or abstract syntax to test operator precedence.

this section we will discuss how the language of the output fragment can be specified. The output program will be parsed with the specified language, or with the language under test if no other language was specified for the output fragment, and the resulting AST is compared against the output of the test case. A typical example where concrete syntax is useful is to test operator precedence or associativity. Although such a test could be specified by providing the expected AST in ATerm format, it is more concise and convenient to use concrete syntax. Especially because in such a test we don't care about the actual values in the AST, we only want the precedence and associativity to be correct. Both the abstract syntax approach and the concrete syntax approach for such a test are illustrated in figure 2.9.

Next, we will discuss the expectations that can be used to test the analysis of a language.

```
test duplicate tables not allowed [[
  CREATE TABLE t (intCol INT);
  CREATE TABLE [[T]] (intCol INT);
  CREATE TABLE [[T]] (intCol INT);
]]2 errors
  1 warning
   /Duplicate table/
```

Figure 2.10: Testing for error messages on duplicate table names, and a warning on a table name that isn't capitalized.

Testing the Analysis

The analysis uses the AST of the program to reason about its meaning. Analysis is usually used to signal problems or possible problems in the program to the developer, but it can also be used to provide editor services such as tooltips with type information when hovering over expressions or following a reference to its definition by clicking on it. In its most basic form, SPT expects the analysis to return a set of error and warning messages. For example, the analysis of MiniSQL returns an error when comparing incomparable types, and it returns a warning on table names that don't start with a capital letter. The expected output for errors and warnings can be expressed as `n errors` and `n warnings` respectively. The number `n` here can be any non-negative integer. SPT will check if exactly `n` errors or warnings were returned. As `1 errors` is not proper English, SPT also offers the `n error` and `n warning` expectations. They are just an alias for their plural counterparts. Figure 2.10 shows a test case that uses all of SPT's test expectations related to testing errors and warnings. This test case expects 2 errors, because duplicate table names are not allowed (one error on each duplicate table name). It expects 1 warning, because a table name without a starting capital letter is used (`t`). The third test expectation specifies that one of these error and warning messages should contain the string "Duplicate table". As long as at least one of the errors or warnings contains that string, the expectation is met. Finally, it illustrates how selections inside the fragment can be used to specify stronger test expectations. When the `n errors` or `n warnings` test expectations are present, SPT checks if all marked selections in the input fragment contain an error or warning message. So in this example we ensure that the error messages are indeed located on the duplicate table names.

SPT also offers dedicated test expectations for name analysis. The core of name analysis is to determine to which definition the references in a program refer. Most languages that do static name analysis do not allow references for which no definition can be found, as both their value and type can't be known. To check if the name analysis managed to resolve a reference to any

```
test reference resolution [[
  CREATE TABLE S (intCol INT);
  CREATE TABLE [[T]] (intCol INT);
  SELECT t.intCol FROM [[T]] t;
]] resolve #2 to #1
```

Figure 2.11: Checking if a reference to a table resolves to the correct definition.

definition, the `resolve` test expectation can be used. The reference in the input fragment should be selected, and the `resolve` keyword should then be followed by the reference to that selection. To check if it resolves to a specific definition, the definition can be selected as well and the `resolve` expectation can be followed by the `to` keyword and the reference to the selected definition. An example test case for the name analysis on table names can be found in figure 2.11.

Testing Transformations

Part of the language implementation generated by Spoofax is an editor for the language that is being developed. As discussed in section 2.2 transformations can be made available to users through the menu of this generated editor. In Spoofax versions before 2.0 transformations that were made available this way were called *builders*. To test these transformations SPT offers the `build` test expectations.

Let's take the `gen-ora-style-builder` builder from section 2.2 as an example. To test if a builder succeeds a simple `build builder-name` is enough. To test if a builder fails, we can just add the `fails` keyword, just like we do with the `parse` test expectation. If we also want to test the output of the transformation, the same options are available as for the `parse` test expectation: an ATerm representation of the output, or an output fragment. Figure 2.12 shows how we can test the output of the `gen-ora-style-builder` builder. Note that we can mark a selection within the input fragment (in this case the `SELECT` query). The builder will be executed as if the user selected the marked text before executing the builder. This example illustrates the situation from figure 2.5;

Testing other components

We have now discussed all of the test expectations that SPT offers to test the behavior of components that are common to most languages (parsing, analysis, transformations). However, some areas are not yet covered. For example, editor services like syntax highlighting and content completion can not be tested with the expectations we have discussed so far.

```
test join predicate transformation [[
  CREATE TABLE Tbl (
          intCol int,
          strCol VARCHAR
  );
  [[
  SELECT t1.intCol, t2.strCol
  FROM Tbl t1
  JOIN Tbl t2
  ON t1.intCol = t2.intCol;
  ]]
]] build gen-ora-style-builder to "
  SELECT t1.intCol, t2.strCol,
  FROM Tbl t1, Tbl t2
  WHERE t1.intCol = t2.intCol;
"
```

Figure 2.12: Testing the transformation from MiniSQL's ANSI style joins to Oracle style join predicates.

To cover the testing of any component, there is the `run` expectation. This expectation allows the developer to run something on either the entire input fragment or on a selection within the fragment. In Spoofax, the Stratego language is used to glue all the different components together and to extend them with functionality that could not be expressed in one of the other meta languages [12]. The run expectation can be used to run a piece of Stratego code (called a strategy), allowing the developer to test arbitrary components of their language implementation. An example of the run expectation can be seen in figure 2.13. Here we check the type of a selected expression by invoking the `get-type` strategy. This strategy interfaces with the type system that the developer defined in the TS meta-language. To specify the expected output of the strategy, the same format can be used as for the `build` expectation: `fails`, an ATerm specification, or an output fragment.

Finally, SPT offers a freeform test expectation to test arbitrary pieces of Stratego code without even requiring an input fragment. By using this freeform expectation, SPT kind of turns in to a unit testing framework for Stratego code. An example of the freeform expectation can be seen in figure 2.14, where we specify some test cases for the `sum` strategy of Stratego, which is supposed to sum up all the elements in a list.

```
test check types [[
  CREATE TABLE T (intCol INT);
  SELECT t.intCol
  FROM T t
  WHERE [[t.intCol > 5]];
]] run get-type to Boolean()
```

Figure 2.13: Running a Stratego strategy to test the type of a comparison operation.

```
test sum works on lists
<sum()> [1,2,3] succeeds

test sum ONLY works on lists
<sum()> 1 fails

test does sum work?
<sum()> [1,2,3] => 6
```

Figure 2.14: Some freeform test cases for the `sum` strategy of Stratego.

Test Suites

Test cases are grouped into so called test suites. Each SPT file is a named test suite that contains zero or more test cases. Organizing test cases into test suites does not only allow the developer to organize their tests, it also offers possibilities for the configuration of all the tests in a test suite at once.

A test suite starts with the `module` keyword, followed by the name of the test suite. This declaration is followed by one or more configuration headers, that provide runtime configuration for all the tests in the test suite. After these headers, test cases and `setup` blocks can be declared. This `setup` block is how *test fixtures* can be defined in the previous SPT. They provide a way to set up and tear down the initial state for all the tests in the test suite.

First, we will provide an overview of the available configuration headers. Then, test fixtures will be discussed.

Configuration Headers

The most important configuration header is the `language` header. This header specifies what the language under test is for all the tests in the test suite. It starts with the `language` keyword, followed by the name of the language under test. This header is optional as long as the test suite contains no test cases. As soon as the test suite contains a test case, the language under test has to be specified through this header. The language that is specified using this

```
module expressions
language MiniSQL

test expression syntax [[
  CREATE TABLE T(
    intCol INT
  );
  SELECT t.intCol
  FROM T t
  WHERE
    t.intCol > 4 + 5 * 6;
]]
```

(a) Wrapping an expression in a valid MiniSQL program.

```
module expressions
language MiniSQL
start symbol Exp

test expression syntax [[
  4 + 5 * 6
]]
```

(b) Using the start symbol header to reduce boilerplate.

Figure 2.15: The start symbol header allows parsing to start from a different non terminal to reduce the size of fragments.

header will be used for all the input fragments and for evaluation of the test expectations of all the test cases in the test suite. To specify a language to use with the output fragments of the test cases inside the test suite, use the `target language` header. All output fragments, like those from `parse`, `build`, or `run` expectations, will be parsed with the specified language.

The next header is the optional `start symbol` header. This header specifies from which start symbol the syntax tests should operate. A start symbol is simply a non terminal of the grammar definition that the language developer wrote in SDF. It can be used to test the syntax of parts of the language, thereby reducing the amount of boilerplate code that has to be written. For example, let's take another look at figure 2.9 where we test the operator precedence in MiniSQL. The input fragments of these tests are just expressions, and not valid MiniSQL programs. If the MiniSQL parser would be executed on such a fragment, it would fail, causing the tests to fail. To test the syntax of expressions we would have to put the expression inside a real MiniSQL program. For example, by wrapping it in a predicate of a query, but that would require declaring the full program for every test case for the syntax of expressions.

Figure 2.15 shows the difference in boilerplate that the use of the start symbol header can bring. These test suites only contain a single test case, but the difference becomes even more apparent as more test cases are added to the test suite, because the start symbol header applies to all test cases in the test suite.

```
module expressions
language MiniSQL

setup schema [[
  CREATE TABLE T (
    intCol int,
    strCol VARCHAR
  );
]]

test select syntax [[
  SELECT t.intCol
  FROM T t;
]] parse succeeds

test join syntax [[
  SELECT t1.strCol
  FROM T t1 JOIN T t2
  ON t1.intCol = t2.intCol;
]] parse succeeds
```

Figure 2.16: Using test fixtures to declare initial state for all test cases.

Fixtures

We already discussed how the start symbol header can be used to reduce boilerplate. This only works if the parser is capable of parsing from a specified non terminal. If this is not possible, or to specify other initial state of the component under test, test fixtures can be used. Test fixtures are textual fragments that will be combined with the input fragment of each test case to create the actual input program for that test. As such, they can be used to extract text common to all test cases in the test suite. Test fixtures can be expressed with `setup` blocks. For example, in figure 2.16 a test fixture is used to create the schema that is used by both test cases.

Chapter 3

Requirements Analysis

Last chapter we discussed how behavioural test cases for languages can be concisely and declaratively specified using SPT. In this chapter we will look at how the previous SPT works and what its shortcomings are. First, we will briefly touch upon the Language Parametric Testing Language (LPTL) [2]. The LPTL is the generalized idea behind SPT: to make a testing language which can be used to test any other language (the language parameter of the LPTL). We will discuss how the previous SPT implements the ideas and goals of the LPTL, and where it falls short. Then, we will give an overview of the architecture of the previous SPT and discuss some problems resulting from this architecture. Finally, we will discuss the use cases for SPT. Both the use cases for which it was designed and the use cases that emerged over time as it was being used in practice. For each use case we will look at if, and how, the previous implementation of SPT can be used to fulfill it.

3.1 The ideas behind SPT

When SPT was designed, the goal was to make a testing language that could be used to test any textual software language. The term for this language was a Language Parametric Testing Language (LPTL) [2]. SPT is the Spoofax specific implementation of a subset of the design goals of this LPTL. We will discuss the LPTL in a little more detail, outline these design goals, and look at how the previous SPT tries to meet them.

An LPTL is a testing language for computer languages, which is completely agnostic to the language it is testing. Ideally, the LPTL should be agnostic to both the language specification and the language implementation of the language it is testing. When it is agnostic to the language specification, it means that tests can be written for any language specification. When it is agnostic to the language implementation, it means that tests can be executed against any implementation of the language specification. The specific language specification and implementation that are being tested can be seen as the language parameters of the testing language.

The novelty of an LPTL is that it can use abstractions specific to the domain of testing languages, to offer tool support for *writing* tests. By treating input programs for tests as a first class citizen of the testing language, the LPTL can offer editor services like real-time error reporting, syntax highlighting, and content completion within this input program. It also allows easy selection of parts of the program. The use of test expectations specific to testing language components (like the parser or analysis) allows the developer to concisely declare what should be tested. Instead of implementing a test case, they merely have to declare it. This focus on supporting the developer when writing test cases is the main benefit of an LPTL.

The second benefit of an LPTL is that it allows the developer to write tests against the language specification, not the implementation. As illustrated in figure 2.2 this allows the reuse of test cases, which is useful for languages with many different implementations (like JavaScript or SQL). This way of testing allows the use of test driven development, as the developer can write tests that capture the desired behaviour before they implement it. It is also particularly well suited for testing languages created by using a language workbench, as the language implementation is generated for such languages, and the developer does not need to have in-depth knowledge of the details of the generated implementation to test it.

The third benefit is that an LPTL allows the testing of any language. This means that once the developer is familiar with the LPTL, they can use it to read and write tests for many different languages.

To offer these benefits, an LPTL would have to meet some requirements. The main requirements are outlined as design goals of an LPTL in [2].

- agnostic to the language specification
- agnostic to the language implementation
- allow instantiation for a specific language under test
- offer test fixtures

Language specification agnostic

The first design goal is to be language specification agnostic. This means it should be possible to test any textual language. To do this, we should offer a way to express any textual input program as a first class citizen of the LPTL.

As we have already seen in chapter 2.3 the previous SPT uses fragments to express input programs as first class citizens. However, due to its architecture and implementation, these fragments do not allow the expression of all program texts. There are 2 limitations on the language specifications it can test.

```
test double selection marker [[
  a = [[5]]
  print a[0][0]
]] 0 errors
```

(a) In this case, we would get an error, as the variable a is assigned the value 5. The double square brackets around the 5 are interpreted as an SPT selection.

```
test triple selection marker [[[
  a = [[5]]
  print a[0][0]
]]] 0 errors
```

(b) In this case the test would succeed. The double square brackets are not considered a selection, as the triple squared brackets are used as delimiter of the fragment.

Figure 3.1: Using different markers to not conflict with the program text in the fragment. These test cases use Python as the language under test.

The first limitation is mostly a theoretical limitation. To be able to recognize selections inside a fragment, the program text inside the fragment should not contain the selection markers as part of the program itself. If it does, SPT will parse it as a selection marker instead, possibly causing faulty program syntax in the input program. This issue is illustrated in figure 3.1a, where we are testing a simple (and contrived) Python program. In this program we assign a list, containing a list, containing the value 5 to the variable a. The syntax for this introduces two square brackets directly after each other, which also happens to be the syntax for an SPT selection marker. The test case in figure 3.1a would fail, as the double square brackets would be parsed as SPT tokens and removed from the program text, causing the variable a to have the value 5 instead of a list with a list with the value 5. As we already mentioned, this is hardly a problem in practice, as SPT offers multiple markers to work around this problem. In figure 3.1b we can see how we can use the triple square brackets as delimiters for the fragment, causing SPT to use these same triple square brackets as a selection marker. The double square brackets would no longer be parsed as a selection marker, and there would no longer be an issue. Another easy solution for this problem would be to add a space between the two subsequent square brackets in figure 3.1a. This is still valid Python syntax, with the same semantics, but since the square brackets no longer directly follow each other, they are not parsed as an SPT selection marker.

The second limitation is a bigger problem. Due to its architecture and implementation for showing error messages inside the fragment, the previous SPT does not support all layout sensitive language specifications. A layout sensitive language is a language where whitespace characters (e.g., tabs, spaces, or newlines) actually matter for the syntax and semantics of the program. The problem is caused by the way the previous SPT handles the parsing of test cases. It replaces all SPT specific characters in the file with spaces, and leaves only the text inside the fragment and test fixtures intact. We will

```
test double selection marker [[
  def main():
    [[x]] = "hello world"
    print [[x]]

  main()
]] resolve #1 to #0
   0 errors
```

Figure 3.2: The SPT selection markers will be replaced by whitespace, causing the 'print' statement to be incorrectly indented.

discuss why SPT does this in chapter 3.2, where we will look into the details of the architecture and implementation of SPT. The problem with this approach is that it adds whitespace characters to the program text, which may alter the meaning of the program or cause syntax errors. An example of this can be seen in figure 3.2. Here we have another test case for the Python language. Python is layout sensitive, because it expects blocks of statements to start at the same column (character offset from the start of the line). As the double square brackets would get replaced by spaces, the assignment of variable x will be indented by 2 extra spaces. This would cause the next line (the print statement) to not be on the same column and therefore the program would raise an error. A possible workaround for this specific problem would be to indent the print statement by another 2 spaces. It would solve the indentation problem, but would require taking these implementation details of SPT into account while writing tests, which defeats the main purpose of SPT: to make writing tests easier.

Language implementation agnostic

Next is the design goal to be agnostic to the language implementation. There are two parts to meeting this requirement. First, the LPTL should allow the developer to specify tests for their language without having to worry about implementation details. Second, these tests should be executed against an actual implementation.

For the first part, we need to allow the developer to reason about the behavior of the implementation, without needing to specify any details of the implementation. The proposed LPTL of [2] accomplishes this by offering a set of test expectations for common components of language implementations. However, this set of expectations can not cover all components of any implementation. For example, there are no test expectations for things like the type system or testing tooltips that pop up when you hover over elements of the program. To allow testing of such components that are not covered by the LPTL's main test expectations, there is the `run` expectation, which runs something against (part of) the input program and checks the result. This is implementation agnostic in the sense that the developer doesn't have to specify how this thing should be executed or where it can be found.

The previous SPT offers test expectations for parsing, errors and warnings resulting from semantic analysis, and reference resolution. It also implements the `run` expectation, and therefore fully covers this first part.

The second part of being agnostic to the language implementation is executing the test cases. It requires executing parts of the language implementation. To do so, either the LPTL should have knowledge about the language implementation or the language implementation must adhere to some interface defined by the LPTL. Both of these solutions mean we exclude some language implementations, making it impossible to be truly agnostic to the language implementation. Therefore, the LPTL should limit the amount of work one has to do to make the tests executable against a new implementation. For example, by requiring the implementation to expose its behavior through a specific API.

As SPT is the Spoofax specific implementation of an LPTL, this design goal to be language implementation agnostic can be limited to language implementations generated by Spoofax. However, the previous SPT does not call the components of the language implementations through a generic API. Instead, it directly uses some of the generated artifacts. The biggest problem here is that it uses the generated parse table for the language under test to parse test cases. However, Spoofax allows a language to register a custom parser, which can do additional tasks on top of just parsing a file with the generated parse table. For example, the previous SPT has such a custom parser. This custom parser takes care of parsing the test cases and merging the results into the output of the parsed SPT syntax. As the previous SPT does not support these custom parsers, it can not be used to test itself. Even though the syntax of fragments allows SPT to test itself (language specification agnostic), this direct usage of the parse table means SPT can not be used to execute test cases against itself.

Instantiate the language under test

The third design goal is to be able to instantiate the LPTL for a specific language under test. This means that it should be possible to specify which language to use, and the implementation for that language should be loaded. That way the tests can be executed against that implementation and the results can be displayed directly in the editor. It is not acceptable to have to compile a different version of the LPTL for each language we want to test it with. This means the instantiation should happen at runtime.

The way this is done in the previous SPT relies on the architecture of Spoofax before version 2.0. As these versions were tied into the Eclipse IDE, it meant that SPT was only used inside this environment. Therefore, it could use the IMP language registry which was part of all Spoofax installations, to access all Spoofax languages registered in the current Eclipse IDE. Based on the name in the language header of the test suite, SPT pulls the required artifacts from this language registry and uses them to execute tests.

The limitation here, is that it requires a running instance of Eclipse to be able to execute tests. Which makes programmatic execution of test cases a problem. It also means that in order to test against a different implementation for the language under test, this new implementation has to be loaded into the language registry. Doing so requires manual work and a restart of Eclipse.

Test fixtures

The final design goal for the LPTL is to offer test fixtures which can be used to factor out boilerplate code from multiple test cases. Test fixtures are a part of an input program, which will be combined with the fragment of each test case to form the input program which will actually be tested.

We have already shown an example of test fixtures in section 2.3. What's important to note here, is that the test fixtures of the previous SPT are slightly different than the proposed fixtures of the LPTL [2]. There can be only one LPTL test fixture per test suite. The test fixture is located in one place and defines the place where the program text of the test cases' fragment should be inserted with the following marker `[[...]]`. Test fixtures in SPT are called setup blocks. Each test suite can contain multiple setup blocks, and they have no markers to insert the program text of the fragment. Instead, the setup blocks are located around the test cases and when a test case is parsed, all SPT specific syntax is replaced by spaces, and only the text of the setup blocks and the test case's fragment remains, which is parsed with the language under test. This difference is illustrated in figure 3.3, which shows the same test case as in figure 2.15 with test fixtures from the LPTL and with setup blocks from SPT.

Setup blocks are more powerful than test fixtures, but less readable, because they can be inserted between test cases. This would cause some test cases' fragments to be 'inserted' at other places than other test cases' fragments. In practice this power is seldom needed and only confuses the reader of the test suite. This is illustrated in figure 3.3b where we need to define a separate setup block at the end of the test suite just to insert the semi colon. The test fixture in figure 3.3a is a lot more readable, as it is in the same place. The only reason for the setup blocks of the previous SPT is the fact that it was required to allow the use of the whitespace technique we discussed before. As mentioned before, we will discuss why SPT needs this whitespace technique in chapter 3.2.

In this section we have discussed the design goals behind SPT and some of the problems that have to be solved to meet them. These problems are

```
fixture [[
  CREATE TABLE T(
    intCol INT
  );
  SELECT t.intCol
  FROM T t
  WHERE
    t.intCol > [[...]];
]]

test expression syntax [[
  4 + 5 * 6
]]
```

(a) Test fixtures of the LPTL.

```
setup header [[
  CREATE TABLE T(
    intCol INT
  );
  SELECT t.intCol
  FROM T t
  WHERE
    t.intCol >
]]

test expression syntax [[
  4 + 5 * 6
]]

setup footer [[
  ;
]]
```

(b) Test fixtures of the previous SPT.

Figure 3.3: Test fixtures between SPT and LPTL.

summarized in table 3.1.

3.2 Previous SPT architecture and implementation

As we discussed in the previous section, the idea behind SPT was to create a single language that allows:

1. the declarative specification of tests for any textual language,
2. instantiating SPT with a language implementation,
3. the execution of these test cases using this language implementation,
4. reuse of boilerplate code between tests by offering test fixtures.

Each of these goals provides their own problems that SPT had to tackle. In this section we will discuss the architecture of the previous SPT, the problems that had to be solved, and how the previous implementation has tackled them.

The main architecture for the previous SPT was determined by the goal to have a single language to test any other language created with Spoofax. In the Spoofax ecosystem, most of the tools that are offered are languages which

Category	Discussion	Solved	
Specification agnostic	dynamic markers		1.1
		selections in program text	1.2
Implementation agnostic		extensible set of expectations	1.3
		allow custom parsers	1.4
		error reporting in fragment	1.5
	syntax highlighting		1.6
	other editor tools		1.7
Fixtures		declarative fixtures	1.8
Instantiation		reuse tests for multiple implementations	1.9

Table 3.1: Summary of the problem areas where the previous SPT does not (fully) meet the goals of an LPTL, as we discussed in section 3.1.

are themselves made with Spoofax. The decision was made to do this for SPT as well. To do so, it meant the parsing, analysis, and editor services of the language under test had to be integrated with those of SPT itself, and the results had to be merged with those of SPT itself in such a way that Spoofax could handle them.

As explained in section 2.2, each Spoofax language consists of the following main aspects: parsing, analysis, transformation, and editor services. We will now discuss for each aspect, how the previous SPT tackled the problem of combining its own results with those of the language under test.

Parsing Tests and Fragments

During the parsing phase both the SPT syntax and the fragments of the test cases are parsed. To accomplish this, SPT first needs to be instantiated with the language under test, so it knows how to parse the fragments. There are two ways to accomplish this. The first is merging the grammar of SPT and the language under test. The second is parsing the fragments separately from the SPT syntax.

Merging the grammars is not an easy task, as the SPT selections syntax can be used inside the program text of the fragment and faulty syntax is allowed inside fragments as well. For this reason, SPT uses the second approach and parses the fragments separately from the SPT syntax.

Parsing the fragments separately requires several steps. First, the SPT syntax is parsed, so we have access to the configuration headers, the raw text inside the fragments, the selections, and the test expectations of each test. Then, we need to instantiate SPT and get the parser for the language under test. Then, the parser needs the fragment text as input, without any characters that are part of SPT selections. With the use of test fixtures, the fragment text needs to be created from the combination of these test fixtures and the fragment itself. Finally, the parse results of the fragments have to be merged with the parse result from the SPT syntax. This includes the ASTs of fragments that parse successfully and the errors of fragments that failed to parse.

This approach to parsing does not fit well within the Spoofax paradigm. In Spoofax, the language developer writes a grammar definition, from which a parse table is generated that can be used with Spoofax' scannerless generalized LR parser (SGLR) [14]. The problem with this is that it does not allow the developer to perform any custom actions like the ones required to parse the fragments. To work around this problem, Spoofax allows languages to provide a custom parser, which can perform additional actions, as long as it returns an AST and a token stream. We will now discuss how the custom parser for SPT parsed the fragments of the test cases and combined their results with the results of parsing the SPT syntax.

Instantiate SPT To parse the fragments of the test cases, the custom SPT parser has to know where it can find the parse table for the language under test. Spoofax was tightly coupled with the Eclipse IDE at the time and it used the IDE Metatooling Platform (IMP) for Eclipse to store information for all Spoofax languages that were installed. The SPT parser uses the IMP language registry directly to find the parse table for the language under test and then invokes the SGLR parser to parse the fragment.

This approach has two main drawbacks. First, the dependency on the IMP language registry means the SPT parser requires an Eclipse environment. When Spoofax was redesigned and became platform independent in version 2.0, SPT could not be easily migrated, because it still had direct dependencies on Eclipse and IMP. Second, the explicit use of the SGLR parser means SPT can only be used to test languages that work with the standard Spoofax parser. Any language that requires a custom parser, like SPT itself, can not be tested with this approach.

Obtaining fragment text Fragments of test cases can contain selections. Selections are delimited on the left and right by markers which are part of the SPT syntax instead of the fragment. These selection markers need to be removed from the fragment to obtain the fragment text. The previous SPT simply replaces these markers with spaces. The reason for doing this, is that it keeps the location of the characters within the fragment text, defined by the line number and character offset, in sync with the location of these characters in the test specification. This is important when merging the parse result of the fragment with the parse result of the SPT syntax, which we will discuss later.

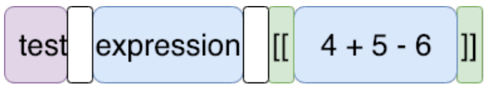
For test fixtures, the text of the fragment has to be combined with the fixture. We have already explained how the test fixtures of SPT are different from those of the proposed LPTL. Instead of a single test fixture, SPT requires the use of multiple setup blocks. The location of the setup block relative to the fragment determines how it should be combined with the fragment text. All text of all setup blocks before the fragment should come before the fragment text, and all text of all setup blocks after the fragment should come after the fragment text.

This approach of using multiple setup blocks was chosen to keep the locations of characters in the resulting program text in sync with their locations in the test specification. If the LPTL test fixture would have been used, it means that there can be characters (like the semicolon of figure 3.3a) which appear before the characters of the test's fragment in the test specification, but appear after the test fragment in the combined program text of the fragment and the test fixture. Because the characters no longer have the same location within the test specification and within the program text constructed from combining the fixture with the fragment text, it becomes harder to merge the token streams. We will look at token streams, why they need to be merged, and what problems this brings in more detail when we discuss the merging of the parse results.

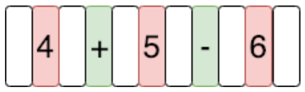
Just like with selections inside the fragment, any character that is not part of the setup blocks' text or the fragment text is replaced with a space. The resulting text (with lots of whitespace) is given to the SGLR parser with the parse table for the language under test, to obtain a parse result for the fragment. This step is repeated for every fragment in the test specification.

For larger test suites, all this excess whitespace slowed down the parsing a lot. We benchmarked this on a simple test suite that can be generated when you make an example Spoofax project. This test suite contains 7 tests and a total of 56 lines and the fragments were parsed in less than a tenth of a second. When copying these 7 tests 11 times, we get a total of 77 tests on 516 lines, and the parsing of fragments slowed down to more than 34 seconds. To allow direct feedback to the developer, parsing is triggered every time the developer edits the test suite. In such a case, waiting for more than a couple of seconds to get any feedback is a problem.

Merging parse results Finally, the parse results of the fragments have to be combined with the parse result of the SPT syntax. The main result of parsing with the SGLR is either an AST or one or more parse er-



(a) The token stream produced by parsing an SPT test case.



(b) The token stream produced by parsing the fragment of this test case with the parser of the language under test.

Figure 3.4: Token streams from parsing a test case and its fragment. Each block represents a token in the stream.

rors. Next to this result, the parser also returns a token stream. In the standard approach to parsing, a lexer (or scanner) first turns a program text into a token stream, and then the parser parses the token stream. However, the SGLR is *scannerless*, which means it directly parses the program text and *produces* a token stream. The tokens in this token stream represent syntactic elements like keywords, delimiters, or text, which makes them useful for services like syntax coloring. The nodes of the AST are linked to one or more tokens in the token stream. Each token is linked to one or more characters in the program text, so tokens can be used to find the location of the characters that represent the AST node. When a program text fails to parse, an error token is used to represent the characters that could not be parsed.

In figure 3.4 we can see an example of such token streams. First, we see the token stream produced by parsing only the SPT syntax (figure 3.4a). Each block represents a token, and the color of the block represents the type of the token. In this figure tokens representing layout (i.e., whitespace) are white. Tokens representing keywords are purple. Green is used for operator tokens. Blue represents text, and finally red represents numbers. We can see that as far as the SPT syntax is concerned the entire fragment is just text. The other token stream (figure 3.4b) is what we would get from parsing the fragment text with the parser for the language under test. Here we see that this token stream does provide further syntactic meaning to the fragment by recognizing numbers and operators.

To merge the parse results of the fragments with the parse result of the SPT syntax, both the ASTs and the token streams have to be merged. Merging the ASTs is achieved by annotating the SPT AST node representing the fragment, with the AST resulting from parsing the fragment. If an error occurred during the parsing of the fragment, the SPT AST node is annotated with an error node. This way, the analysis of SPT can access the parse result of the fragment and use it to execute the test. This situation is illustrated in figure 3.5.

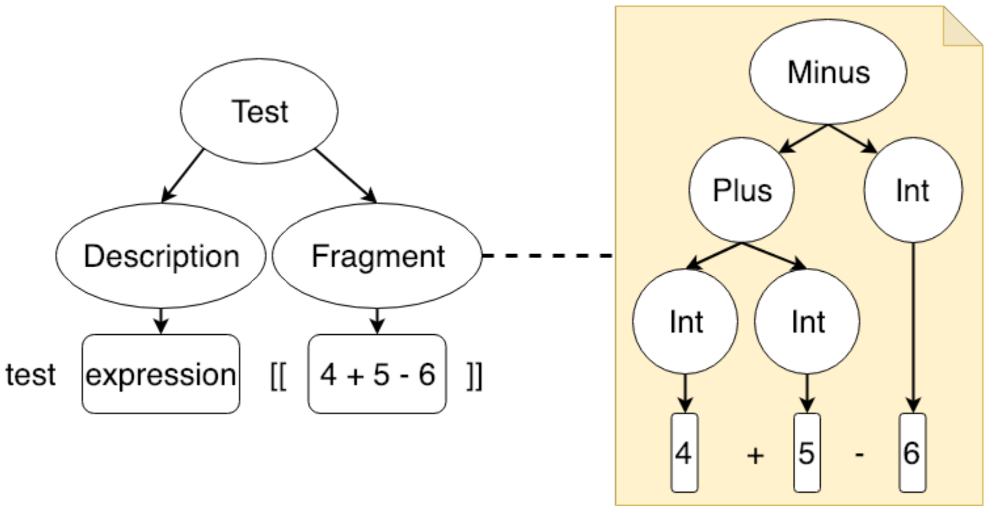


Figure 3.5: The AST of the parsed fragment (in yellow) is added as an annotation on the AST of the SPT syntax.

Merging the token streams is a bit more complicated. Each AST is linked to a single token stream and each node in that AST is linked to a set of tokens from that stream. Editor services like clicking on a reference to jump to the definition of that reference use this link to know where to jump to. For services like these, the character locations of the tokens of the parsed fragment need to match those of the original test specification. The Spoofax editor uses error tokens in the token stream of the AST it gets from the parser to mark any parse errors. For a positive test case, where no syntax errors are expected, any parse errors should be reported in the fragment. To do so, these unexpected parse errors in the token streams of the parsed fragments have to be merged into the token stream of the AST returned by the custom SPT parser.

We call the token stream obtained by parsing the SPT syntax the SPT token stream (see figure 3.4a). The token stream of the parse result of the fragment text will be called the fragment token stream (see figure 3.4b). The custom parser creates a new token stream and copies over all tokens from the SPT token stream, up to the first token that is linked to a character in a fragment text. Then, if a parse failure is expected for this fragment, any error tokens in the fragment token stream are converted into other tokens. This prevents SPT from showing parse errors on fragments of test cases where such a parse error was expected. Next, it copies over the tokens from the fragment token stream that contain characters inside the fragment text.

As described above, the character locations of the parsed fragment text are consistent with their location in the original test specification due to

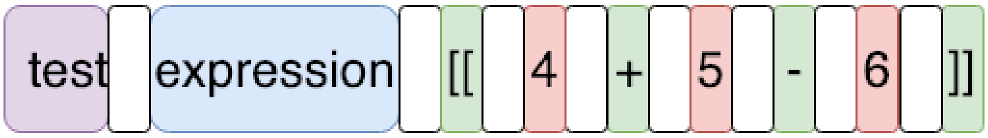


Figure 3.6: The token streams from figure 3.4 merged into a single token stream.

the whitespace technique that the custom SPT parser uses. Therefore, the tokens from the fragment token streams work seamlessly with those from the SPT token stream and can just be appended to the new stream. If the offsets would have been different we would have to determine if a token was part of the fragment text, or of a test fixture. If it's part of the fragment text, the character offsets linked to the token should be mapped to match the actual offsets within the test specification. Although this is feasible, it does make things a lot more complicated. Therefore, the previous SPT uses its whitespace technique to avoid these problems.

Next, it skips all tokens containing characters from the fragment text from the SPT token stream. This process repeats until all fragments have been processed. The end result is a new token stream where the tokens of the fragment token stream are used for each fragment. All the other tokens in the new token stream come from the SPT token stream. For the example token streams from figure 3.4, the resulting merged token stream is shown in figure 3.6. Here we can see that the blue text tokens representing the fragment in the SPT token stream have all simply been replaced by the tokens from the fragment token stream.

During this merging of the token streams, we lost the tokens from the SPT token stream representing the fragments (the blue tokens from figure 3.4). Any AST nodes linked to these old tokens now have to be linked to the new tokens (the ones copied over from the fragment token stream) so they are still linked to the correct locations in the file. After this is done, all AST nodes of the merged SPT AST and fragment ASTs are linked to tokens in the new token stream. This means that the merged AST and token stream can safely be passed back to Spoofax.

The returned token stream is used by the Spoofax editor to determine the syntax highlighting for the file. This means that the syntax highlighting of SPT is applied to the tokens from the parsed fragments. The result is a form of syntax highlighting as defined by SPT, instead of the highlighting as defined by the language under test. As Spoofax does not allow the language developer to dynamically determine the highlighting rules, this was the only possible approach to getting some form of highlighting. This means it is a limitation in both SPT and in Spoofax.

Analysis and executing tests

Whenever changes are made to a program in the Spoofax editor, the new program is parsed and analyzed. Problems with the syntax of the program are detected during parsing. We already discussed how this works for SPT in the previous section. Analysis is responsible for finding any problems with the semantics of the program, like type or name errors. To enable interactive test execution the previous SPT executes tests during the analysis stage and reports any failing test results like semantic errors.

For Spoofax, the input for the analysis stage is the AST obtained from the parsing stage. The result should be a new AST with more information, like types and information about definitions and references, and a collection of messages. The new AST's nodes are linked to nodes of the input AST and therefore they are also linked to a location in the program text. The messages consist of a location within the program text, a message text, and a severity (i.e. errors, warnings, and notes).

Just like how the SPT parser parses both SPT syntax and the fragments, the analysis of SPT should also analyze both itself and the fragments. An example of an analysis error in SPT itself would be if a test expectation references a selection, but there are no selections in the corresponding fragment. This analysis of SPT itself always happens. Analysis of the fragment itself is only required if a test expectation that does not test the parser is present. For example, to test reference resolution, we need the information from the new AST returned from the analysis of the fragment. After analyzing the fragment, the expectations can be checked. Based on the evaluation of the test expectations some messages from the analyzed fragment have to be propagated to the output of the SPT analysis. For parse expectations, the analyzer simply checks the parse result of the fragment to see if it was an error or not.

Analyzing the fragment Just like the SPT parser obtained the parse table for the language under test from the IMP language registry, the SPT analyzer can obtain the analyzer for the language under test. Although the previous SPT uses the parse table of the language under test to parse the fragments, thereby losing the possibility of testing languages with a custom parser, the analysis in Spoofax is always triggered from a Stratego strategy. So when it comes to analysis, SPT supports all languages created with Spoofax.

The analyzer is executed and the result is stored, as all actions required for the evaluation of test expectations require this result.

Evaluating test expectations To evaluate test expectations related to messages (i.e. errors, warnings, notes), the analyzer simply checks the messages it got from analyzing the fragment. As the output AST from the analysis of the fragment is linked to the input AST that was obtained from parsing, the locations of the messages already correspond to locations in the test specification file. This makes it trivial to check both the number of messages, and if the location of the message matches that of a selection inside the fragment.

For reference resolution expectations (i.e. `resolve` expectations), the IMP language registry is consulted to get the reference resolution strategy. This strategy is executed on a node of the AST obtained from analysis of the fragment. Running reference resolution on an AST node representing a reference returns the AST node corresponding to the definition of that reference. First, the reference node needs to be obtained, based on a selection in the fragment. SPT traverses the AST until it finds a node node with the same location as the selection. However, more than one node can have the same location. For example, both the `List` and the `ColRef` nodes on the left side of figure 2.4 correspond to the same characters in the program text, and therefore have the same location. In the case of reference resolution, SPT always gets the innermost node (in this case the `ColRef`) as that is where the analysis stores the reference resolution information. To check if the output of reference resolution is correct, we simply compare the location of the returned AST node with the location of the selection.

The final type of test expectations are transformations (i.e. `build` expectations), running Stratego strategies (i.e. `run` expectations), and freeform Stratego test expectations. The `build` and `run` expectations first select what part of the analyzed AST to run on, either the entire AST or the node corresponding to the selection if a selection was made. Just as with reference resolution, there may be more than one such node. In this case SPT selects the outer most node that matches the location of the selection. Now the specified builder or strategy can be executed on this input and the result can be compared with the expected output. For the freeform expectations, there is no input fragment at all. The Stratego code inside the expectation is simply evaluated and the result compared with the expected output.

Each test expectation is evaluated separately. If any expectation fails, an error message is generated on the description of the test. A test expectation can fail because the action (e.g., reference resolution) failed to execute properly or because the output does not match the expected output. Although the expectations are evaluated separately, they use the same input fragment and the same selections. This can make the semantics of a test case unclear, when multiple expectations share the same selections. For example, when you test for error or warning messages, all selections in the fragment are expected to contain an error or warning. This means it is not possible to test a builder in the same test case, as the selections of the error or warning expectation will also be used by the build expectation. Only in the rare case where these selections are at exactly the same spot can we combine the two expectations.

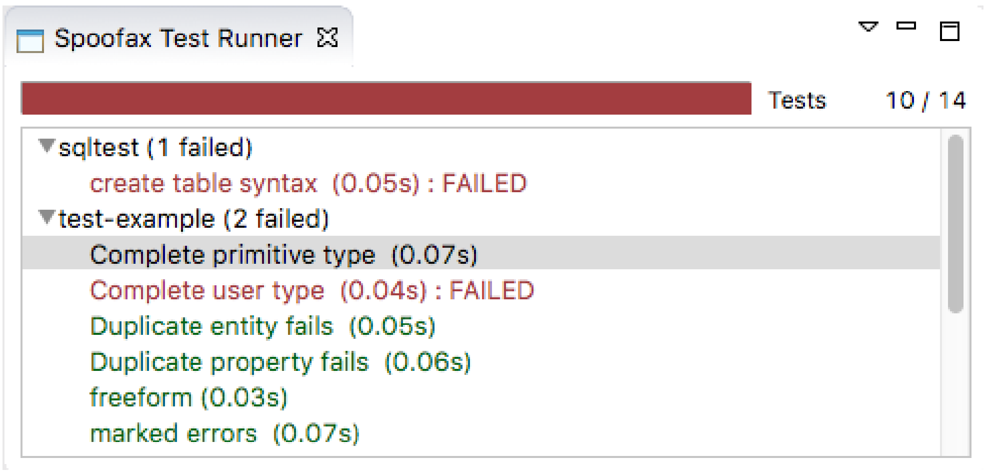
propagating messages When a test expectation fails, new error or warning messages are generated to describe what went wrong. However, some of the original messages from analyzing the fragment may be important for the language engineer to see why the test failed. For example, when a test expectation specifies it expects an error at a certain location, but the error is actually somewhere else, it would be good to show that error in the editor. In cases such as this, these messages are collected and also returned as part of the output of the analysis phase of SPT. These messages are simply copied over, because their location is already in sync with the location of the test specification. The downside of this approach is that a message that spans from an element in a test fixture to an element in the input fragment of the test case will be displayed in its entirety by the editor. So the editor will also mark all the SPT characters in between the element in the test fixture and the element in the input fragment as part of the message, because no care is taken to prevent that. Another problem is that the test fixtures are combined with all input fragments of the test specification. So any messages inside the fixture can be propagated by each failing test case, resulting in duplicate messages.

Transformations for SPT

There are two transformations for SPT in the menu of the SPT editor. Both transformations are batch test runners which can be used to get a more concise overview of test results. The first transformation executes the tests of the test specification that is currently open in the editor. The second transformation executes all tests of all test specifications in the same project as the test specification that is currently open in the editor. The usual way to see if tests were executed correctly is to manually check for errors in the editor. However, the batch runner provides a clear and concise overview of the results of the execution of all tests of one or more test specifications.

These transformations are a good illustration of how broad the idea of a transformation is. They are no source to source transformation, but instead parse and analyze all the appropriate test specifications and provide a graphical representation of the output. Figure 3.7 shows the batch execution of two test specifications, `sqltest` and `test-example`. For each test case in these specifications it shows the description (i.e., name) of the test case and a color to indicate if it passed or failed.

The batch runner consists of two parts. The first part is the transformation implementation in the Spoofax project. This part determines which test specification files it should execute on and parses and analyzes those files.


Spoofax Test Runner

Tests 10 / 14

```
▼sqltest (1 failed)
    create table syntax  (0.05s) : FAILED
▼test-example (2 failed)
    Complete primitive type  (0.07s)
    Complete user type  (0.04s) : FAILED
    Duplicate entity fails  (0.05s)
    Duplicate property fails  (0.06s)
    freeform (0.03s)
    marked errors  (0.07s)
```

Figure 3.7: The batch runner that executed all tests of two different test specifications.

Category	Discussion	Solved	
Parsing		test custom parsers	2.1
		platform independent	2.2
		real time feedback	2.3
Analysis		clear semantics of test expectations	2.4
		errors contained in fragments	2.5

Table 3.2: Summary of the problems with the previous SPT implementation we discussed in section 3.2.

While it does that it calls into the second part of the batch runner, the so called test listener. This test listener is connected to a graphical UI element in the Eclipse workspace. By separating the batch runner into two parts it is possible to use a different test listener if applicable. The transformation and the test listener are linked to each other at runtime through the use of Eclipse's extension points API. Although this test runner gives a nice overview of the results of multiple test suites at once, it can only be manually executed from within the Eclipse IDE.

The problems we discussed in this section are summarized in table 3.2.

3.3 Use Cases for SPT

We have already discussed some of the problems SPT faces in trying to meet its design goals and doing what it was created for: to enable test driven development of languages with the Spoofax language workbench. However, as SPT was being used, new use cases emerged. We will give an overview of both the main use cases for which SPT was designed, and the use cases that arose while SPT was in use. For each use case we will look at how well the previous SPT can be used to meet that use case, and identify any potential problems.

Interactive Test Case Design

Interactive test case design is concerned with supporting the creation of test cases. Instead of only offering a way to run test specifications, the developer should be provided with interactive services when writing the test specification. For example, when writing a test specification in SPT, there is syntax highlighting available and errors are reported inside the fragment. Test cases are executed as soon as they are written and any test failures are reported directly inside the editor. These editor services make it easier to write the test specification and facilitate test driven development. This was one of the main use cases for which SPT was developed.

To aid the language engineer with writing tests, the previous SPT offers syntax highlighting, content completion, real time error reporting, and hover tooltips for the SPT syntax. For example, the content completion can generate templates for a test case, or complete a test expectation. Hovering over a fragment of a test case will show a tooltip with the result of parsing the fragment.

The provided editor services for the program text inside the fragments and test fixtures are a bit more limited. Ideally, the language engineer would have syntax highlighting, content completion, real time error reporting, and hover tooltips available while writing the fragment. However, the previous SPT only supports real time error reporting and a limited version of syntax highlighting within fragments and test fixtures. We already discussed when and how syntactic and semantic errors are propagated to the SPT editor in the previous section. We also discussed the perfomance limitations on larger test suites. This means that, for larger test suites, the error reporting is no longer real time. The syntax highlighting within fragments and test fixtures is also a little tricky. Instead of using the syntax highlighting as defined by the language under test, SPT uses its own highlighting rules. Languages developed with Spoofax can define their own syntax highlighting rules using the Editor SerVice language (ESV). It is possible to define syntactic colors and styles based on the name or type of an AST node. Unless a custom syntax highlighting specification is present, a language developed with Spoofax will use the default syntax highlighting based on the tokens in the token stream. The previous SPT applies this default token based highlighting to the text within fragments and test fixtures.

Regression testing

When using test driven development, tests are written for each feature of the language implementation. When a feature is completed, the tests for that feature will become part of the regression tests. Regression tests are tests used to verify if a software system still has the correct behaviour after it has been changed. So when a new feature is implemented, the tests that were written for the other features will also be executed again to verify that the behaviour has not changed. Even when test driven development is not used, regression tests are often still used. They can help verify that any changes made during maintenance do not introduce new bugs.

The set of regression tests can become pretty large, making it impractical to run them by manually inspecting each test case of each test suite inside the SPT editor. This use case creates the need for a way to run multiple test suites and check the results in a convenient overview. If the set of regression tests becomes really large, a subset of tests is often selected as the quick regression tests. These are the most important tests and should be executed after every modification. The full set of regression tests can then be executed less often to reduce the time spent running tests.

Regression testing was also part of the use cases for which SPT was developed. To help give an overview of test results from either one test suite, or all test suites in a project, SPT has a batch runner (see figure 3.7. Although there is no way to specify which test suites should be part of short regression tests, and which ones shouldn't, it does allow the developer to quickly check the results of all test suites of the project.

Continuous Integration

For bigger software projects, where multiple developers work on the same system, the changes of each developer can be made simultaneously and the resulting differences in the system have to be integrated with each other. The more the two systems diverge, the harder it will be to integrate them. The solution for this problem is continuous integration [15], where developers regularly integrate their changes with a controlled version of the software system. To further reduce the manual effort, a continuous integration server can be used. A continuous integration server will usually build the software system and run the regression tests whenever a developer integrates his changes. If the system fails to build or some tests fail, the change can be rejected, causing the controlled version of the software system to stay in a working state.

As Spoofax was being used in bigger projects, the need for automatic execution of SPT tests by a continuous integration system arose. The continuous integration server usually wants to reject a change as quickly as possible if a problem was found, causing it to stop test execution as soon as one regression test fails (fail fast strategy). However, some systems prefer to run all tests and give a more detailed and comprehensive list of the problems with the change, to make integration easier for the developer. Therefore, both of these execution strategies have to be supported for this use case.

In the previous SPT the only way to execute test cases is to open them in the editor, or to run the batch runner. Both approaches are tightly integrated with the Eclipse IDE and can only be executed manually. Another problem is the fact that the results of the test cases are not displayed in a machine processable way. This means that a continuous integration system can not execute the tests, and even if it could, it could not check the results.

Automated Grading of Student Tests and Language Implementations

During the Compiler Construction course at the Delft University of Technology, students learn about both traditional and new techniques for the creation of compilers. The course is accompanied by a lab, where students create their own language implementation, using Spoofax, of a subset of the Java programming language, called MiniJava. The lab is split up into several steps, each covering a certain component like the parser, the type system, name analysis, other static semantic checks, and the compiler that transforms the MiniJava program to Jasmin bytecode.

Test driven development is used for this lab, where the students first submit an SPT test suite for a new feature that they will be implementing. The test suite is graded by running it against a set of slightly wrong reference implementations with regards to the feature that is supposed to be tested by their test suite. Their grade depends on how many of these errors are caught by their test suite. After that, they get to implement and submit their own implementation, using Spoofax, which is graded by running our own test suite on their implementation.

This lab both illustrates the capability of SPT to be used for test driven development, as well as its capability to write tests against the language specification, not the implementation. This second point is illustrated by the fact that students' test cases can be executed on a set of slightly different language implementations, as well as the fact that our own test suite can be executed on the students' language implementations.

To make automated grading of both the students' test suites and their implementations possible, some way of programmatic execution of SPT test cases is required. The grading of a student's language implementation is quite straightforward, we simply run an existing test suite against their implementation and detract points for every failing test case. The only requirements this poses are the automatic execution of tests and having the test run return a machine processable result.

The grading of a student's test suite is a bit trickier. First, a set of slightly wrong language implementations has to be created. Each test case should then

Category	Discussion	Solved	
Interactive design		editor services for SPT	3.1
	editor services for fragments		3.2
		real time feedback	3.3
Regression testing		select test suites to run	3.4
Continuous integration		test execution strategies	3.5
		programmatic execution	3.6
		machine processable output	3.7
Automated grading		filtering test cases	3.8

Table 3.3: Summary of the problems we have to tackle to make SPT suitable for the use cases we discussed section 3.3.

be executed first against a correct reference implementation of the language. If it fails against the correct implementation, the test case no longer needs to be executed against the slightly wrong implementations, as its failure does not mean anything. The test cases that did indeed pass against the correct implementation are then executed against each slightly wrong implementation. If any of them fail, the test suite discovered the bug represented by that implementation and the rest of the test cases no longer need to be executed on this implementation. The student then gets points for every wrong implementation that his test suite discovered. This requires more instrumentation of test execution. It requires access to the actual test cases to filter out those that failed against the correct reference implementation, as well as a fail-fast mode of execution for the remaining test cases, where test execution stops as soon as a failing test case occurs.

We have looked at the two use cases for which SPT was created, and found some problems with the previous approach. For the two new use cases we also identified some missing features. Although some of them overlap, we summarize the results in table 3.3.

Chapter 4

Proposed solution for SPT

The main problems with the previous SPT that sparked this thesis were the fact that it isn't compatible with the new Spoofax, and that it could not be used for the new use cases: continuous integration and automated grading (see chapter 3.3). After analysis of the previous implementation we also found another big problem (see chapter 3.2): the whitespace method used by the implementation to keep error messages at the right location in the test suite caused test execution of larger test suites to grind to a halt. It also meant that the implementation had to use different, less declarative, test fixtures than the orignal design of [2] (see chapter 3.1).

Solving these three problems is the main goal for the new design of SPT. But since we were doing an overhaul anyway, we also decided to tackle some other design issues. As discussed in chapter 2.3 and chapter 3.1, SPT gains a lot of its declarative power from concise test expectations to test common components of languages. However, these expectations can not capture every conceivable component of every language (hence the need for the `run` expectation). Therefore, we can expect to need more test expectations in the future. So we also aim to make the new design extensible with regards to test expectations.

This brings us to the following list of problems to solve:

1. work with the new Spoofax 2.0
2. support the use cases of chapter 3.3
3. move more towards an LPTL
4. extensible with regards to test expectations

To work with the new Spoofax, we identified which parts of SPT depended on the old Spoofax internals, Eclipse, or IMP. For supporting projects (more than 1 test suite) SPT used Eclipse specific APIs. For loading the language under test it used IMP. To access components of this language (parse table,

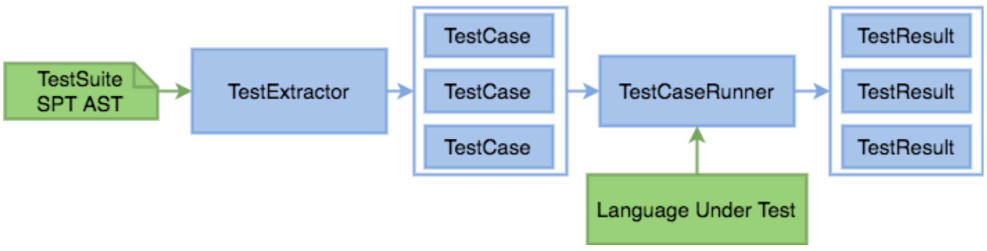


Figure 4.1: The architecture of the new SPT core API. The TestExtractor extracts the test cases from an SPT specification. The TestCaseRunner can then be used to execute test cases.

analysis, transformations) it used the old Spoofax internals. The new Spoofax version offers APIs for all these things. The previous SPT is a monolith, so although getting SPT to work with the new Spoofax is a straightforward port, it's a tedious process.

The other problems require a bit more changes to the setup of the previous SPT. To fix these problems we split SPT into multiple parts to create a more modular system. The main parts are SPT Core, the SPT grammar, and the interactive Spoofax language for SPT providing the editor support. The idea behind this split is that the core of SPT (to execute test cases) should be used by machines (required for continuous integration and automated grading and regression testing) and people (interactive test case design). However, the machines are not interested in the editor support for SPT.

We will now discuss SPT Core and its architecture. Then, we will list the changes we propose for the SPT grammar, to move more towards the goals of an LPTL and make the language clearer. Finally, we describe the process of adding new test expectations to SPT.

4.1 SPT Core

The main functionality of SPT Core is to execute test cases defined in an SPT test specification. In the simplest use case you just pass a test specification to SPT Core and get the test results back. However, the new use cases require more control over test execution, like a fail-fast execution strategy or cherry-picking which test cases from the specification should be executed and which shouldn't be. Therefore, we decided to go with the same architecture that has been used for the xUnit family of unit testing frameworks [8]. This architecture represents every test case as a single unit, which can be executed with a test runner. The benefit of this approach is that the user has complete control over which test cases they want to run.

Figure 4.1 illustrates how a list of TestCase objects can be obtained from an SPT test suite specification. These TestCases contain all the information that is required to execute the test case, like the fragment text and the list of test expectations. The TestCaseRunner can then be used to execute one or more of these TestCases. The result is a TestResult object, which contains information about the execution of the test case, like whether it passed or not, and if not, a collection of error messages. We leave it up to the user of the API to decide how to represent this information. For example, the SPT editor will show any error messages at the right location in the fragment, and a regression tester will give a more concise overview with just the test case name and whether it passed or not.

Extracting Test Cases

To extract the test cases from a test suite, the SPT syntax needs to be parsed. As discussed in chapter 3.2, the grammar of SPT syntax can not be combined with the language under test into a single grammar. The main reason for this is that SPT has to allow both instantiation for any language under test, and negative test cases with faulty syntax in the fragments. Therefore we still use the same approach of first parsing the SPT syntax, and then parsing the fragments later, after SPT has been instantiated.

We decided to delay instantiation of SPT until the test case is executed, instead of doing it during test case extraction. The main reason for this design decision was to support the use case of automated grading of student test cases. These tests have to be executed multiple times against different reference implementations, to see if they catch the mistakes in these implementations. By delaying the instantiation until the test is executed, we only have to extract the test cases once. An added benefit of leaving the parsing of the fragments until test execution, is that a simple parser generated by Spoofax can be used to parse the SPT syntax, instead of the custom parser that the old SPT used.

As illustrated in figure 4.1 the resulting AST from parsing the SPT syntax is the input for the extractor. We also allow passing the test suite file instead of the parse result, for convenience. In that case the extractor parses it before the extraction starts. The extractor then retrieves the configuration headers and their values from the AST, like the module name of the test suite and the name of the language under test. It also processes each test case's AST node. For each test case it extracts the description, and converts the AST node representing the fragment into a Fragment object that can be queried for its text and any selections inside the text.

The fragment text is combined with the test fixture if the test suite has one and is represented as pairs of a starting character offset and a string. The character offset is the offset of the first character of the string in the test specification. An example of the fragment text for the test case of figure 3.3a can be seen in figure 4.2. Here we see the text of fragment and the test fixture. Each piece of text is preceded by the start offset of the first character of the piece of text in the actual file. After each piece we can see the range of offsets

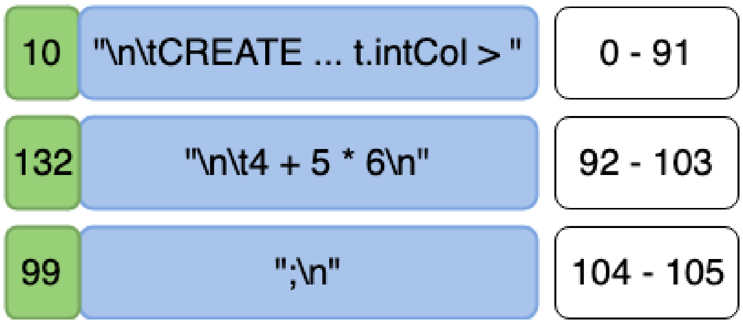


Figure 4.2: The representation of the fragment text of figure 3.3a as stored in SPT Core. It stores the text as tuple of the start offset of the text in the test suite and the text itself. To the right is the offset range that the text would span when all texts are appended.

this piece of text would span if we only considered the fragment text. With this information we can keep track of where each piece of the fragment text is located and map any errors to the correct location in the test specification file without using the whitespace trick we discussed in chapter 3.2. We will get into this in more detail when discussing the execution of tests.

The only remaining part of the test case are the test expectations. SPT Core has expectation providers, which can be registered through dependency injection. For each AST node representing a test expectation, this expectation providers are searched to find one that can convert this AST node to a TestExpectation object. The resulting expectations are added to the TestCase object. By using dependency injection to convert test expectations' AST nodes to an intermediate format, it becomes easy to add new test expectations to SPT.

With the description of the test case, the program text of the fixture and fragment, and the test expectations we have everything we need to represent a test case. The next step is to execute this test case.

Executing Test Cases

To execute a test case, we pass the test case object and the language under test to the TestRunner. The runner will execute the test and return whether the test case was executed and passed all expectations. If any errors occur, or expectations aren't met, the result will also contain error messages mapped to the correct location in the test suite.

The first step of executing the test case is to check what should happen. The runner searches through all registered expectation evaluators to find an evaluator for each of the test expectations of the test case. These evaluators can be registered through dependency injection, just like the expectation providers we already discussed. An expectation evaluator is responsible for checking if a test expectation is met or not. Once we have all the expectation evaluators, we check if they require a parse result or an analysis result. This lets us avoid doing unnecessary work.

Depending on the needs of the expectation evaluators, the program text of the test case (the fragment and fixture text) is parsed and possibly analysed using the given language under test. Each evaluator then gets the parse or analysis result and returns whether the test expectation passes or not. If the expectation is not met, the result also contains a list of error messages.

The final job of the evaluator is to specify which selections inside the fragment it uses to determine if the test expectation is met. This way we can give a warning if a test case contains any unused selections.

Once we have the results of each test expectation, we know if the test case passed or not.

Although the evaluation of test expectations is left completely to the test expectation evaluator, we do have some utilities for tasks performed by multiple expectations. As shown in figure 2.9 some test expectations have output fragments. These fragments also have to be parsed or analysed, either with the language under test, or with a different language. In the previous SPT, this is called the target language and can be declared with a configuration header as discussed in chapter 2.3. We argue that the language for the output fragment should be specific to the output fragment, and should therefore not be configured for all output fragments in the test suite. We will discuss this in more detail in the next section. What is important to know is that these target languages have to be obtained somehow. The language under test is explicitly passed to the test runner to be able to rerun test cases with different language implementations for the automated grading use case of chapter 3.3. For now, we don't have a use case that requires the swapping of these target languages as well, so they are loaded the standard way. The new Spoofax offers a language registry to register languages by name. As SPT Core runs within the Spoofax framework, we use this language registry to retrieve the target language by name. Once the target language is obtained, the output fragment can be parsed with the FragmentParser utility. This is the same utility that is used to parse the input fragment and keep the locations of messages in sync with the location of the fragment in the test suite.

We made two different implementations for this fragment parser. One uses the same whitespace trick that the previous SPT uses. It replaces all characters of the SPT syntax with spaces, leaving only the fixture and fragment text in tact and at the same location (character offset) as it has in the actual file. This trivially ensures the messages are mapped to the right location in the test suite, but is extremely slow and limits the ability of SPT to test layout sensitive languages. The other implementation simply concatenates the text of the fixture and fragment, without adding any whitespace. It then keeps the locations of messages in track by mapping them back to the correct location in the test suite. Each fragment's text can be represented as a set of tuples of a start offset and a piece of program text. This is illustrated in figure 4.2, where we see the pieces of fragment text of the test case from figure 3.3a. Each consecutive piece of program text is accompanied by the offset of the first character of that piece of text. In figure 4.2 this start offset is located to the left of the program text. On the right the we can see the range of character offsets it spans if we concatenate the pieces. So any location at offsets between 0 and 91 will actually be 10 characters further in the actual test specification. Any location between 92 and 103 will actually be 40 characters further (132 − 92). And any location between 104 and 105 will actually be 5 characters back in the actual test specification. This method requires a bit more work to keep the locations of AST nodes in track with the position of the corresponding text in the test suite. But by doing so SPT can test layout sensitive language and each test case can be executed in time proportional to the length of the program text of the test case, as opposed to the file length of the test suite it was in. Another benefit of this approach is that the declarative LPTL style test fixtures discussed in chapter 3.1 can now be used, as the fragment no longer needs to be in between the start and end of the test fixture in the actual test suite.

The methods described above are not only used to keep messages in the correct location in the test suite, but also to get the AST nodes corresponding to a selection in the fragment. For these selections we have the location in the test suite, and need to map that to the right location used by the parser of the fragment. As discussed in chapter 3.2 the AST nodes produced by the parser are linked to their corresponding text by using character offsets. The fragment parser also needs to make sure these links are still correct with regards to the actual location in the test suite, so we can easily obtain the AST nodes corresponding to a selection.

We also have a utility that can be used to compare AST nodes, taking the wildcards of expected output in consideration (see chapter 2.3).

This way the evaluator can easily parse and analyze fragments, compare output, and get AST nodes corresponding to the selection in the test case. The specifics of what to do with these building blocks is left to the evaluator.

4.2 Changes to the language

Apart from changes to the architecture of SPT, we also propose changes to the language itself. These changes are meant to make SPT more declarative and easier to understand. One of the problems we discussed in chapter 3 was that the behaviour of some of the test expectations of the previous SPT were not clear and not documented. Especially the behaviour of multiple test expectations on the same test case, and the selections within them was not clear.

The first change was to remove the `target language` configuration header.

A target language determines which language should be used to work with the output fragments, the fragments of some test expectations, like the `parse to` expectation. By specifying this target language in a configuration header, the developer is forced to group their test cases in test suites based on the target language of the test expectations of the test cases. We believe test cases should be grouped into test suites based on which part of the language they test, not based on the specifics of test expectations that are used. Therefore, the `target language` header has been removed. Instead, the developer can specify the target language for each output fragment in the test expectation. If no target language is specified, the language under test is used for that fragment. This allows the developer to group test cases as they wish, and also makes it a lot clearer what the test case means, as the target language is stated right at the test expectation that uses it.

We also replaced the setup blocks the previous SPT uses by the more declarative test fixtures from [2]. This change was made possible by removing the whitespace trick and keeping track of the location of fragment text in a different way as discussed in the previous section.

Finally, we made some changes to the test expectations. Some of these changes were made to make the test expectations more clear or legible. Others simply add more functionality.

One of the most unclear set of expectations were the ones for testing analysis messages (errors, warnings, and notes). A clear example can be found in figure 2.10. Here we have 3 test expectations to check some assertions about error and warning messages. The semantics of the first 2 test expectations seems quite straightforward: we expect exactly 2 errors and 1 warning. However, the presence of selections in the fragment makes it a lot less simple. As discussed before, if any of the `n errors`, `n warnings`, or `n notes` test expectations are present, each selection in the test case must contain an error, warning, or note. This makes it hard to combine these test expectations with other test expectations that use selections. For example, if we have a `resolve to` expectation as well as a `1 warning` expectation, we now have to ensure the selection used by the `resolve to` expectation has an error, warning, or note. Another problem is that you can't specify what you expect at the selection. Even if you only have a `n errors` expectation with a selection, it will still pass if there is only a warning or note at the selection. Both of these problems can be fixed by making the expectation a bit more specific. For these reasons, we updated these test expectations to ignore selections, unless they are specifically mentioned in the expectation. This gives us far greater control over what we test, makes the semantics of the expectation clearer, and therefore allows easier combination of test expectation. The example of figure 2.10 can now be written as shown in figure 4.3. We can ensure the errors are located at the duplicate table names, and that they are actual errors not warnings. We can also ensure the warning is where we expect it: on the table name starting with a lower case letter. Finally, we can now also be more specific when testing the

```
test duplicate tables not allowed [[
  CREATE TABLE [[t]] (intCol INT);
  CREATE TABLE [[T]] (intCol INT);
  CREATE TABLE [[T]] (intCol INT);
]] 2 errors at #2, #3
  1 warning at #1
  error like "Duplicate table" at #2
  error like "Duplicate table" at #3
  warning like "should start with a capital" at #1
```

Figure 4.3: Testing for error messages on duplicate table names, and a warning on a table name that isn't capitalized.

contents of errors, warnings, and notes. The old syntax only checks if there is any error or warning with contents that match the expectation. With the new syntax we can specify if we expect an error, or a warning, or a note with the given contents. We can also optionally specify a selection to ensure the message is located at a specific spot. As can be seen in figure 4.3 we can now fully test the messages and their locations and contents for this example.

Another change we made to these message expectations is to less specific if one chooses to. For example, in the test case of figure 4.3 we currently check if there are exactly 2 errors. What we would really like to check in this example, however, is only if the duplicate table name errors are present. If, for whatever reason, there would also be another error, we still want this test to pass. As long as the duplicate table errors are present and at the right location. To allow this kind of testing we added an optional operator (`=`, `<`, `<=`, `>`, `>=`) in front of the expectation. So for figure 4.3 we can write `>= 2 errors at #2, #3`. A change like this would have been difficult to add in the previous architecture for SPT, where the evaluation of all test expectations was intertwined. However, with the new approach, the evaluation of each test expectation is isolated in it's own evaluator, making it easy to add or change functionality. We will discuss this new approach to test expectations in more detail in the next section.

We made a similar change to the `run` expectation. This expectation runs and Stratego strategy on either the entire fragment, or on the AST node corresponding to the first selection in the fragment. As with this test expectations for checking analysis messages, this implicit use of selections makes it harder to combine it with other test expectations that also use selections. Therefore, we decided to allow the developer to specify on which selection the strategy should be run. The syntax for this is `run my-strategy on #1`.

Finally, we change the `build` test expectation. This expectation runs a transformation, which was called a builder in the old Spoofax versions. In the previous SPT it is executed the same way as any other Stratego strategy. The input of a builder is a tuple with some specific information regarding the AST of the full program and the currently selected AST node (if there is any). This knowledge of the expected input of a builder could be considered knowledge of the internals of Spoofax, instead of a public API. As the new Spoofax has a public API specifically for executing transformations, we decided we should rely on that, instead of the previous way of running transformations. This new API, however, does not work based on the name of the transformation (the name of the Stratego strategy), but based on the name under which this transformation is registered. Let's look at the example of figure 2.12. Here we see a test for the transformation that is defined as shown in figure 2.5. The previous SPT would construct the specific input tuple for builders and execute the `gen-ora-style-builder` strategy. The new Spoofax's public API for transformations, however, needs to be passed the name under which it is registered: "Generate Oracle SQL". As this name is not necessarily unique, one may have to also specify the menu's in which the transformation is nested. In this case, that would be "Generation -¿ Generate Oracle SQL".

Because this is such a drastic change, and to keep the terminology consistent, we decided to rename the `build` expectation to `transform`. The only difference is the name of the builder, and that instead of specifying the strategy name, the name (or hierarchy of names) under which it is registered should be passed. The expectation of figure 2.12 would now be `transform "Generation -> Generate Oracle SQL to ..."`.

4.3 Adding new test expectations

Although the previous grammar of SPT supports test expectations for content completion and refactoring, these are both broken. There is also no support for the testing of analysis messages of the `note` severity. These test expectations will have to be implemented at some point, and it's very likely that more new test expectations could be added in the future. A possible example would be a `has type` test expectation to test the type system. Therefore, we separated the evaluation of each test expectation from the others, making it a modular system. This makes it easier to add new test expectations, or change the behaviour of existing test expectations, without affecting others.

Each test expectation has 4 components:

1. A grammar component. The expectation has to be recognized by the SPT parser so it can generate an AST from it.

2. A TestExpectationProvider. This class can take the AST node of the expectation and extract its data into a data model.

3. A TestExpectation data model. This model will hold all the information required to evaluate the test expectation. It will be a part of a TestCase.

```
test check types [[
  CREATE TABLE T (intCol INT);
  SELECT t.intCol
  FROM T t
  WHERE [[t.intCol > 5]];
]] #1 has type Boolean()
```

Figure 4.4: A possible new test expectation to test the type system.

4. A TestExpectationEvaluator. This evaluator will be given the TestExpectation and the results from parsing or analysing the test case's input fragment. It should then determine if the expectation passes or not, and report the results.

To add a new test expectation, these 4 parts have to be implemented. Let's say we wanted to add a new test expectation to test the type system, a `has type` expectation. Consider the current way we test the type system, as illustrated in figure 2.13. This depends on the existence of a `get-type` Stratego strategy that returns the type of the AST node at the first selection in the fragment. As we did with the other test expectations that use selections, we would want to make the actual selection to use explicit. An example of our desired expectation can be seen in figure 4.4.

As discussed before, we split the main grammar for SPT from the rest of the Spoofax language. This way we can not only use just this main grammar for parsing an SPT test suite, before passing it to the test case extractor, but we can also simply extend the grammar to add a new test expectation. Of course we do need to take care that this new test expectation's syntax does not conflict with the existing test expectations. Apart from that, no other knowledge of the SPT grammar is required. Let's use the example of a test expectation to test the type system. We could either change the original SPT grammar, or create a new grammar which imports the original grammar. We just need to add the following grammar rule:

```
Expectation.HasType = <<SelectionRef> has type <ATerm>>
```

The next step is to create a provider for our test expectation. If we decided not to extend the original SPT grammar, but create a new one, we'd have to parse our test suite (see figure 4.4) with this new grammar, and pass the result to the test case extractor as described before. The extractor will find the `HasType(SelectionRef(1), Boolean())` node and ask all registered providers if they can turn this AST node into a data model. Again, we have the option of extending the actual SPT implementation, or simply extending it by adding a new Guice Module. Guice is the dependency injection framework used by the new Spoofax. The SPT Core program is also implemented as a Guice Module that can be loaded into Spoofax. So we can extend this module, or just create a new module and make sure that is also loaded into Spoofax when we run the test case extraction.

The next step is the actual data model for this new test expectation which implements the ITestExpectation interface. It would be a simple data object to store the region of the test expectation's AST node (for purposes of error reporting, as required by the ITestExpectation interface), the number of the selection reference, and the expected type. The extractor would hand the AST node to the provider we just created, which would hand back our new data model. Let's say we called this class HasTypeExpectation. This object would be added to the resulting TestCase's list of expectations.

The final step is the evaluator. During test execution, the test runner will look at the expectations of our test case and try to find the evaluator that was registered for our new HasTypeExpectation, using Guice. The resulting evaluator will be given the HasTypeExpectation object along with the parse and analysis result of the input fragment and the TestCase itself, and be asked to evaluate the test expectation. The evaluator would then ask the number of the selection reference from the HasTypeExpectation, and get this selection from the TestCase. It will use Spoofax's public API to retrieve the AST node corresponding to the location of this selection from the AST of the the analysis of the input fragment. The next step would be to determine the type of this AST node. At the moment Spoofax has no public API for that yet, which is why we still use the `get-type` strategy. We can then compare the resulting type with the expected output from the HasTypeExpectation. If it matches, we report success to the test runner. If anything along the way failed, we report failure, and one or more messages describing what went wrong.

As we can see, the only part of this process where we have to worry about other test expectations is in writing the grammar extension. Further more, all parts can either be implemented separately from our new SPT, or added to it. This allows us to add test expectations specific for some languages in separate modules. The user of SPT can then determine if they want to use these modules with new test expectations or just keep to the standard SPT. We hope this flexibility in a modular setup will make the new SPT a good base to further develop the language as Spoofax gains more and more features.

For example, once Spoofax gets a public API for obtaining the type of an AST node, we can follow the process outlined above to add this new test expectation, without disturbing the rest of the system.

Chapter 5

Evaluation

We have implemented the proposed solution from chapter 4 and evaluated if it indeed fixed the problems we discoverd in chapter 3.

First we will go over the three tables that summarize the problems we found in chapters 3.1, 3.2, and 3.3. For each of the problems we will discuss if they are solved or not. For those we consider solved, we will explain why we do so. For those that are not yet solved, we will discuss the difficulties we encountered for the problem and why we did not solve them. In the next chapter we will discuss the main shortcomings and look at the next steps to improve upon our new SPT with future work.

Then, we will discuss each of the use cases of chapter 3.3 and explain how we used the proposed solution in this setting.

5.1 Requirements of chapter 3.1

In chapter 3.1 we examined the idea behind SPT and which problems SPT has to tackle to get as close to this idea as possible. These problems were summarized in table 3.1:

1.1 dynamic markers The markers SPT offers for selections in fragments restrict the languages which can be tested with SPT, as these markers may not be used in the program text of the fragment. In practice this is not a real problem, as SPT offers 3 different markers, and most languages allow the use of whitespace to break up such sequences in their program text.

Due to this limited impact, we did not attempt to solve this problem and it's still present in both the previous and the new SPT. In the next chapter we will discuss a possible approach to solving it as future work.

1.2 selections The problem here was that the previous SPT replaces all SPT syntax (including the selection markers) with spaces. This means

```
test selections where whitespace isn't allowed [[
    CREATE TA[[]]BLE T();
]] parse succeeds
```

Figure 5.1: Selections in the fragment should not influence the program text that is given to the parser of the language under test.

that it can not be used to test layout sensitive languages where these extra spaces change the semantics of the program.

This behaviour can be illustrated and evaluated with the test case of figure 5.1. Here we can see a selection in the middle of a keyword. The previous SPT would replace that marker with spaces, causing a parse error. Our proposed solution uses a different way to keep the locations of the fragment text in track with where they really are in the test suite file. Thanks to this, we no longer need to replace the marker with spaces. Instead, the new SPT just takes out the marker's syntax, resulting in a valid MiniSQL program, causing the parsing to succeed.

As the new SPT's selections no longer affect the program text, we can now also use them when testing layout sensitive languages. Therefore, we consider this problem solved.

1.3 extensible expectations As described in chapter chap:requirements the previous SPT was implemented as a standard Spoofax language. The evaluation of test expectations was done in Stratego, and it was done in such a way that the evaluation of one type of expectation intertwined with those of other test expectations. The proposed solution fully separates the evaluation of test expectations, and allow new expectations to be added to the language without disturbing the others. Therefore, we consider this problem solved.

1.4 allow custom parsers The problem here was that the previous SPT uses the parse table of the language under test directly to parse the fragments. If the language under test has a custom parser, it will simply be ignored. As the new Spoofax offers a public API for parsing text with a given language, this problem was solved for us. All we had to do was use this API. The benefit is that SPT can now be used to test more complex languages.

A very nice moment in the implementation of the proposed solution was when we were able to test SPT specifications with SPT itself.

Evaluating whether or not custom parsers work in the new Spoofax is outside of the scope of this paper.

1.5 error reporting This describes the problem where SPT has to keep the locations of AST nodes of the parsed fragments in track with the location of the corresponding text in the test suite file. The problem is that the fragment text that is parsed is not the actual text in the file. The file also contains SPT syntax.

Although the previous SPT does indeed report errors inside the input fragment, any errors that span from within the fragment's text to the text in a test fixture will cause the error to stretch over all test cases in between the fragment and the offending setup block of the test fixture. We solved this problem in our proposed solution by trimming messages to always fit inside the fragment. If the message was completely inside the test fixture, we can't trim it to be contained in the fragment, and will simply place it on the description of the test case instead. This is a slight improvement on the error reporting in the previous SPT.

1.6 syntax highlighting This is the problem of providing syntax highlighting in the fragment itself. Ideally, we'd use the syntax highlighting defined by the language under test to highlight the program text in fixtures and fragments. This was not possible, both in the old Spoofax and in the new Spoofax, so the problem was, and remains, unsolved. It is the most important focus for future work, as there is much to be gained in terms of usability.

There is however one problem with our proposed solution that the previous SPT does not have. The previous SPT parses fragments and merges their token streams into the SPT token stream. This allows the Spoofax editor to use the token-based syntax highlighting defined for SPT itself on the program text in the fragments as well. Although it's not the highlighting defined by the language under test, at the very least it is still some highlighting, which helps greatly in the readability of test cases.

Our proposed solution moved the parsing of fragments to execution time (i.e. the analysis phase) instead of parse time. This makes it possible to reuse test cases and greatly simplifies the SPT parser, but it also means that the token streams can only be merged at execution time. Spoofax only does syntax highlighting during parse time, so the merged token streams after analysis do not affect syntax highlighting. The result is that the new SPT doesn't have any highlighting on fragment text anymore.

The solution to this problem would be to change Spoofax to offer an API for syntax highlighting so we can use the highlighting of the language under test. As the new SPT is meant as a basis on which SPT can be further developed while Spoofax is further developed, we decided not to try to incorporate the hacky way the previous SPT provides some highlighting to fragment texts. We would rather see Spoofax extended with a public API for semantic highlighting and use that for SPT.

Until that time though, the lack of any kind of highlighting in fragments is a big unsolved problem for our proposed solution. Much more so than it is in the previous SPT.

1.7 other editor tools Syntax highlighting in fragments got it's own mention, as it was partially solved in the previous SPT, but not in our proposed solution. However, there are many more editor tools we would like to offer inside fragments. Some examples of these would be content completion or tooltips with type information that pop up as you hover over AST nodes.

These editor services are not supported in the previous SPT and they are not supported in our proposed solution either. Therefore we mark this problem as not solved, but there is a clear path to a solution by extending Spoofax. We will discuss this in more detail in chapter 7.

1.8 declarative fixtures This is the problem of the setup blocks used by the previous SPT, as opposed to the test fixtures described in the paper. The setup blocks were invented so any text of the test fixture that comes after the text of a fragment, also comes after the fragment text in the test suite. This was required to make the whitespace trick work. As we implemented a new way to keep locations of AST nodes in track with the actual location of text in the file in our proposed solution, we can now use the more declarative test fixtures as described in the paper.

We used the test case of figure 3.3a to verify that the new type of fixture works. This test parses and executes as expected with the new SPT.

1.9 reuse test cases This is the problem of when SPT is instantiated with a specific language under test. The previous SPT does this at parse time, making it impossible to reuse test cases to execute them against different languages under test. The proposed solution instantiates SPT at analysis time, which does allow reuse of test cases. This is further facilitated by the SPT Core data model, allowing the user control over individual test cases.

As part of our implementation of the proposed solution, we included a test runner which can be executed from the command line. This test runner first extracts test cases, and then executes them with the given language. We used this runner and two similar language implementations to make sure we could execute the same test against both languages.

5.2 Requirements of chapter 3.2

In chapter 3.2 we discussed the architecture of the previous SPT and identified some problems with its implementation. These problems were summarized in table 3.2:

2.1 allow custom parsers This requirement is the same as requirement 1.4 discussed above. The problem of not supporting custom parsers was caused by the lack of an API to obtain such a parser based on the language name. This was solved for us by the new Spoofax, which does offer an API for parsing.

As this was solved by Spoofax itself, and not by our new design, evaluation of this feature is outside of the scope of this thesis.

2.2 platform independent A platform independent version of SPT can be used programmatically on development machines, as well as continuous integration servers. The previous SPT only works within an Eclipse instance with Spoofax loaded, making it hard to execute programmatically. The new Spoofax is platform independent as it is written in Java. Several projects have already been started to integrate it into other integrated development environments like NetBeans and Emacs. The proposed solution is a module within this Spoofax framework and is therefore also platform independent.

We evaluated this by using the command line test runner that is part of our new implementation on both MacOS and Windows machines.

2.3 real time feedback One of the biggest usability problems with the previous SPT, is that it is too slow to provide real time feedback on larger test suites. As the test execution takes place as the developer writes their tests, slow execution means the developer has to pause while writing their tests. While the test execution is running, none of the other editor tools were available. We discovered that this slow down is caused by the whitespace trick employed in the previous SPT. As the test suite grows in size, so does the amount of whitespace that the parser of the language under test has to handle. On larger test suites performance is so bad that the user has to wait several minutes before they get feedback when writing tests. By removing the need for the whitespace trick in the new SPT, the execution time no longer explodes as a test suite gets larger. This leaves us with a fast enough response time for real time feedback. We benchmarked the previous SPT and our implementation of the proposed solution to verify that this problem was indeed fixed. The benchmarks were done on a Macbook Air with a 1.7 GHz Intel Core i7 processor and 8GB of ram.

We started the benchmark with the standard set of test cases that is part of the example project of Spoofax. This baseline test is included below.

```
module test-example

language Runtime

setup Common [[
  module test
]]

test Simple entity [[
  entity User {
    name  : String
    other : User
  }
]] 0 errors

test Resolve type fails [[
  entity User {
    name  : [[Strin]]
    other : [[Use]]
  }
]] 2 errors

test Duplicate entity fails [[
  [[entity User {}]]
  [[entity User {}]]
]] 2 errors

test Duplicate property fails [[
  entity User {
          [[name : String]]
          [[name : String]]
  }
]] 2 errors

test Resolve type [[
  entity [[User]] {}

  entity Owner {
    owns : [[User]]
  }
]] resolve #2 to #1
```

Parsing the fragments of these test cases took around 8 to 16ms depending on the test case. We duplicated these test cases to create larger test suites, one with about 500 lines, and another with about 1000 lines. The resulting parse times are recorded in table 5.1. Note that this was just a single run for each test suite, because the actual values down to the millisecond are not what we are interested in. We simply care about the order of magnitude. We can see that parsing the same test case in a test suite of about 500 lines is an order of magnitude slower than parsing it in the baseline test suite of about 50 lines. The larger test suite, of about 1000 lines, increases the parse time of each test case by another order of magnitude.

We can see that with the previous SPT the parse time for a single test case becomes longer as the size of the test suite grows. It is also important to note that a longer test suite usually has more test cases than a smaller one, all of which will take longer to parse. The result is that for the 500 line test suite we had to wait 34 seconds until all test cases were parsed, and for the larger test suite of 1000 lines we had to wait 346 seconds. This is not fast enough for real time feedback.

Test description	baseline (50 lines)	500 lines	1000 lines
Simple entity	12ms	719ms	4815ms
Resolve type fails	16ms	609ms	3982ms
Duplicate entity fails	10ms	581ms	3813ms
Duplicate property fails	8ms	568ms	3925ms
Resolve type	8ms	555ms	3770ms

Table 5.1: Parse times from the previous SPTfor the fragments of the test cases from the baseline test suite, and larger test suites made by simply copying the test cases from the baseline test suite.

Our proposed solution no longer uses the whitespace trick, which means that the length of the program text of a test case no longer depends on the size of the test suite. We ran the same tests as we did for the previous SPT and the results can be found in table 5.2. The results show that this problem is now indeed solved, the medium test suite was parsed in 155ms, and the long one in 281ms. Each test case has a consistent parse time that is fast enough for real time feedback.

2.4 clear semantics of test expectations The behaviour of test expectations in the previous SPT is not always clear. Especially when multiple expectations are combined, or when selections are present in the fragment. The changes discussed in chapter 4.2 fix these problems. Each test

Test description	baseline (50 lines)	500 lines	1000 lines
Simple entity	29ms	14ms	15ms
Resolve type fails	4ms	4ms	2ms
Duplicate entity fails	4ms	3ms	2ms
Duplicate property fails	3ms	2ms	3ms
Resolve type	4ms	3ms	2ms

Table 5.2: Parse times from our proposed solution for the fragments of the test cases from the baseline test suite, and larger test suites made by simply copying the test cases from the baseline test suite.

```
fixture [[
    CREATE TABLE T(
            stringCol VARCHAR
    );
    SELECT t.stringCol FROM T t WHERE
    t.stringCol = 3 + [[...]];
]]
test selections where whitespace isn't allowed [[
    5
]] 0 errors
```

Figure 5.2: This test case fails, as the expression

$$3 + 5$$

is not of type String. The error will only be shown on the test case, not on the fixture.

expectation is now self contained and combining multiple test expectations on a single test case does not influence their individual semantics. We documented the available test expectations and their semantics. This documentation can be found on the Spoofax documentation website: http://www.metaborg.org/en/latest/source/langdev/meta/lang/spt/index.html.

2.5 errors contained in fragments When an error in the test case starts in a setup block, and ends in the fragment text, or vice versa, the error spans the full length between setup block and fragment. This means that it spans over all test cases in between as well. This problem was fixed in our proposed solution by simply cutting off errors, and other analysis messages, to stay within the fragment itself. If an analysis message is fully outside of the fragment, it is relocated to the description of the test case.

This behaviour can be evaluated with a testcase like that of figure 5.2.

Here we have an expression of type Int being compared to a column of type String. In the previous SPT (using setup blocks instead of the fixture) the error about the wrong type would span all the way from the 3 in the test fixture to the 5 in the fragment. All characters between the 3 and 5 would be marked as an error, including the SPT syntax. In the new SPT the error is cut off at the start of the fragment.

5.3 Requirements of chapter 3.3

In this chapter we listed the main use cases for SPT. Both the ones for which it was designed, and the new use cases that emerged as SPT was being used in practise. We will first discuss each of the problems SPT encounters for these use cases, as summarized in table 3.3. Then, we will give an example of how our proposed solution is used in each of these use cases, to further illustrate why we believe our proposed solution is suited for these use cases.

3.1 editor services for SPT To support the developer in writing tests, the editor in which they write their tests should offer editor services like syntax highlighting, content completion, and error reporting. For the SPT syntax, these editor services are all available, both in the previous SPT and in our proposed solution.

3.2 editor services inside fragments We would also like to provide the editor services mentioned above for the language under test inside the fragments. As we have discussed for requirements 1.6 and 1.7, the editor support inside fragments is still lacking.

3.3 real time feedback This requirement is the same as 2.3 and is discussed in the previous section.

3.4 select which suite to run In the previous SPT tests are executed manually by either opening them in the Eclipse editor, or by executing all tests of a project through the batch runner discussed in chapter 3.2. We maintained this batch test runner in our proposed solution, but also added a command line test runner, where the user can specify which test suites to execute. Of course the SPT core API can also be used programmatically.

This gives the user complete control over which test suites to execute.

3.5 test execution strategies The idea is that we need more control over how tests are executed. In the previous SPT we can only execute no tests, or all tests of a test suite. Sometimes we want to stop running tests as soon as the first test fails to save time (fail-fast).

As the SPT core API gives the user complete control over which test cases they want to run, these strategies can be implemented. Therefore, we consider this problem solved.

3.6 programmatic execution As we mentioned above, the previous SPT only allows manual execution of tests. We added a command line test runner to execute test suites, and programs can also use the SPT core API directly to execute tests. This means that any program that can interface with Java or the command line can execute tests with our proposed solution.

3.7 machine processable output The results of executing test cases in the previous SPT are either displayed directly in the editor, or in a graphical overview in the batch runner. Neither of these are well suited for a program to interpret. This issue is solved by the SPT core API, which returns data objects with a boolean to indicate whether the test passed, and a set of messages to give further feedback as to why it failed if it did.

3.8 filtering test cases to execute This was a new requirement posed by the Automated Grading use case. Here students submit test suites and we need to determine automatically how good these test suites are. To check if a test suite is good, we want to check if it would catch certain types of errors in the language implementation. The first step, however, is to ignore all test cases that fail even on a correct implementation. This is not possible with the previous SPT as it can only execute all the tests in a test suite.

The SPT core API does allow the user to execute a single test case. Allowing them to filter out test cases if they want. We will discuss the details of the grading approach when we evaluate the Automated Grading use case.

Now that we have briefly discussed the main problems that have to be solved to use SPT for the use cases of chapter 3.3, we can see that all of them are solved by our proposed solution. We believe this solution is suitable for all use cases that we have discussed. To further prove this point, the implementation of our proposed solution is already being used in practise. We will now discuss how our proposed solution is used in each of these use cases.

Interactive test case design This use case is about supporting the developer as much as possible as they write their test cases. The previous SPT can already be used for this use case, as it was designed for it. Our proposed solution adds more declarative test fixtures, has more specific test expectations, contains errors inside the fragments, and is fast enough for real time feedback even for longer test cases. These changes all help the developer with writing test cases in a declarative manner, to make writing tests as easy as possible.

One big downside of the proposed solution is the total lack of syntax highlighting in the fragment texts. As discussed above, even if it isn't the syntax highlighting of the language under test itself, at least the previous SPT has some form of syntax highlighting inside fragments.

Currently, our proposed solution is shipped with the new Spoofax. This means that it can be used by anyone who develops their language with such a Spoofax version. It is used to test the GreenMarl language at Oracle, and is widely used in the programming languages research group at the Delft university of technology.

Regression testing Regression testing in the previous SPT requires manually executing tests in the Eclipse batch runner. We still offer this same approach with our proposed solution, but also added more detailed error reporting to help debug any failing test cases. Apart from that, regression testing can now also be done from the command line, using our command line test runner.

In practise, regression testing is usually done as part of continuous integration, which we will discuss next.

Continuous integration As the previous SPT does not allow programmatic test execution, it can not be used for continuous integration. Continuous integration was one of the new use cases for SPT that we aimed to make possible with our new design. Our proposed solution can be used to execute tests programmatically, by using the SPT core API. Gabril Konat has made a Maven plugin which executes all test suites of a language project as part of the build process using SPT core. By using this plugin in a language project, we can make a build fail when a test case fails. This is both useful for regression testing and continuous integration.

Automated grading This is the second new use case for which the previous SPT can not be used because it can't be used programmatically. For the compiler construction course at the Delft university of technology, an automated approach to grading of student test suites and language implementations was wanted.

Now that we can execute test cases programmatically it's quite trivial to test a student's language implementation. You simply run a few test suites against it to find errors in the implementation, and detract points for every failing test case or test suite. As SPT core gives control over specific test cases, you could even assign different weights to each test case. That way you can get more fine grained grading.

The hard part is the grading of a student's test suites. To check if a test suite is any good, we need to check if it would catch a problem in the language implementation. For this a set of slightly wrong language implementations was created. Each implementation has a specific error in it. By executing the student's test suite against the slightly wrong implementation we can see if the test suite captures that particular error. For every slightly wrong implementation that the test suite caught, we can award points to the student.

The first problem is creating this set of slightly wrong language implementations. There is nothing SPT can do to make that easier.

The second problem is knowing if a test suite captured the error in the implementation. For this, we first have to ensure the test suite passes on a right implementation. As SPT core gives complete control over which test cases are executed we can run the test suite on the correct language implementation first and discard all test cases that fail on this implementation. This leaves us with a set of test cases which don't fail on a correct implementation, so if they fail on a wrong implementation, that means they caught the error in the implementation.

Next, we just need to execute these tests on each of the slightly wrong implementations. SPT core allows us to reuse the test case and execute it on a different language under test. This saves us the time required to extract the test cases again or swap the language under test inside Spoofax's language registry. We can also stop test execution as soon as the first test case fails, because that means the error was captured. As both the set of student test suites and the set of slightly wrong language implementations are quite large, anything we can save on performance is quite welcome.

The final step is to award points for each error that the student's test suite has captured.

This automated grading approach has been implemented by Martijn Dwars, using our proposed solution. It is currently being used for grading some of the lab exercises of the compiler construction course.

Chapter 6

Related work

As we have discussed in this thesis, the main power of SPT is to use domain specific abstractions to allow language engineers to write declarative test cases against their language's specification. SPT combines the language under test and the test cases themselves in the same editor, to further support the developer in writing tests. This concept of combining tests and the language, or program, being tested, is not new. For example, Pyret ([16]) allows developers to write their tests around their implementations. Although this combines the program under test and the test in the same editor, it is not specific for language engineering, nor does it enable declarative tests against the specification instead of the implementation.

We will discuss other related work in three different categories. First, we will look at other language workbenches and how they support the language engineers in testing their language. Next, we'll discuss how some languages that have been developed without a language workbench are being tested. Finally, we'll take a look at some work in the field of test case generation. The declarative nature of SPT tests makes it easy to generate an SPT specification once the hard work of generating an input program is done.

6.1 Testing in language workbenches

Spoofax is not the only language workbench that is available. Here we will briefly discuss how testing is done in 4 other major language workbenches, and compare their benefits to SPT: XText, Rascal, Racket, and MPS.

XText – most of the language engineering in this language workbench is done in an extension of Java, called XTend [17]. This language takes away some of the boilerplate of Java, allowing the developer to write more concise programs. As most programming is done in this extension of Java, XText encourages developers to write their tests in JUnit, a well known and widely used testing framework for Java. This means that the

```
...

@Inject extension ParseHelper
@Inject extension ValidationTestHelper

@Test
def void testLowercaseName() {
    val model = "entity foo {}".parse
    model.assertWarning(DomainmodelPackage.Literals.ENTITY, null,
            "Name should start with a capital")
}
```

Figure 6.1: An example of testing if a program has a warning in XText (`http://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#testing`)

test cases are focussed on testing the generated Java artifacts, instead of the specification. However, XText provides some useful libraries which can be used to make the test case more declarative. See figure 6.1 for an example of a very declarative test case for parsing and analysis.

The biggest difference between this approach and SPT is that in XText's testing approach input programs are just strings. This means the developer does not have any editor services available inside the input program text and it's harder and less readable to reference parts of the program text, as there are no markers available. The benefits of using JUnit, however, are that regression testing, continuous integration, and programmatic execution of test cases is very easy. As JUnit offers all of this out of the box. In this thesis we redesigned the architecture of SPT to also offer these benefits.

Racket – a programming language with a strong focus on language oriented programming [18]. This means they encourage developers to write their own domain specific languages to solves the problem they are facing. Therefore, all programming required to make a language using Racket, is done in the Racket language. Any testing also happens in this language.

There are multiple ways to approach testing in Racket. First, there is the OverEasy test engine. This is a test engine for Racket, allowing the developer to group test cases into test suites. Another approach would be to use RackUnit. RackUnit is a library for Racket that also allows the developer to group test cases into test suites, but also provides an easy way to set up and tear down the state of the component under test.

Tests are just code in Racket, which means they can be interspersed between the actual implementation. This allows the developer to work on the language under test and the test in the same editor. The down side is that, although there are a lot of asserts and comparing functions provided, there is not a lot of support specifically for testing languages. This means the tests will be focussed on the implementation, and won't be reusable for other implementations of the language. Another consequence is that test cases are not declarative, unless the developer writes their own abstractions to make their tests declarative. The benefit of SPT is that it offers these abstractions for you.

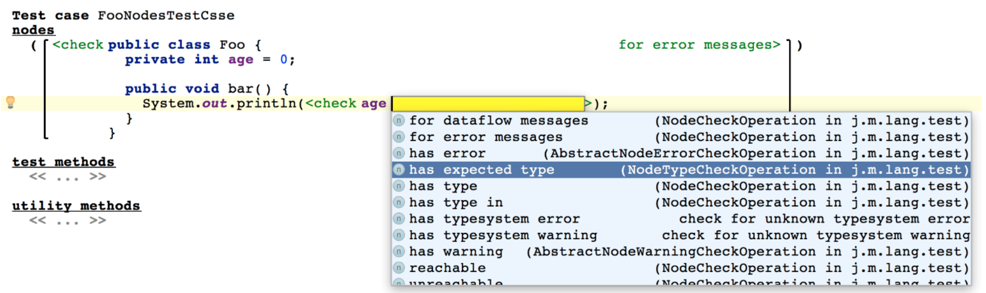
Rascal – this is a programming language with abstractions specifically for working with source code [19]. These abstractions allow developers to more easily create a language with Rascal, that with a programming language without these abstractions. Just like with Racket, all the implementation of the language is done in the Rascal language itself, meaning tests will also have to be written in this language.

Testing in Rascal can be done by creating functions with the `test` keyword. These functions are written in the Rascal language, meaning you have all the same abstractions available when writing tests that you do when creating your language. This means tests can be more declarative, without having to write the abstractions yourself. The downside is the same as with Racket: you're still writing tests against the implementation, which means they can't be shared between between two implementations of the same language. Another problem is that there is no way to group test cases into test suites for easier regression testing, or built in support for continuous integration. Such tools, however, can be built.

Another interesting feature of testing within Rascal, is that when a test function is given parameters, the test engine will generate some random values for these parameters. This is an interesting feature that is closely related to test generation. Which we will discuss in the next section.

MPS – the MetaProgrammingSystem (MPS [20]) is an interesting language workbench. It differs from the other language workbenches we discussed, because it uses a projectional editor as opposed to textual editors. This means it's possible to create languages with input programs that are not just textual, but have different graphical views. It also means that the only way to create a program in such a language, is with the projectional editor of that language.

As the programs in the language under test have to be written in the MPS projectional editor, the tests too need to be able to provide this editor. To do this, MPS allows the developer to specify their input program for the test using the projectional editor for the language under test. This input program can then be annotated with certain test expectations which will be verified when the test is executed. The interesting



```
Test case FooNodesTestCsse
nodes
  ( [<check public class Foo {                                            for error messages> ])
              private int age = 0;

              public void bar() {
                System.out.println(<check age                    >);
              }
            }

test methods
  << ... >>

utility methods
  << ... >>
```

```
for dataflow messages          (NodeCheckOperation in j.m.lang.test)
for error messages             (NodeCheckOperation in j.m.lang.test)
has error         (AbstractNodeErrorCheckOperation in j.m.lang.test)
has expected type          (NodeTypeCheckOperation in j.m.lang.test)
has type                       (NodeCheckOperation in j.m.lang.test)
has type in                    (NodeCheckOperation in j.m.lang.test)
has typesystem error              check for unknown typesystem error
has typesystem warning          check for unknown typesystem warning
has warning     (AbstractNodeWarningCheckOperation in j.m.lang.test)
reachable                      (NodeCheckOperation in j.m.lang.test)
unreachable                    (NodeCheckOperation in j.m.lang.test)
```

Figure 6.2: An example of testing the type system in MPS (`https://www.jetbrains.com/help/mps/testing-languages.html#nodetests`)

thing is that such tests can only be created once the projectional editor of the language under test is created. This makes test driven development impossible, but it does maximize the integration of the editor services of the language under test and the editor for the tests. An example of this can be seen in figure 6.2, where the user is about to add an annotation to test the types of a Java input program.

6.2 Testing outside of language workbenches

Not all languages are created in a language workbench. In this section we will take a look at two well known general purpose programming languages and how they are defined and tested: JavaScript and Java.

JavaScript – as we have mentioned in chapter 2 JavaScript has a test suite written in the language itself [9]. The benefit of this approach is that the test suite can be used to test any JavaScript engine, not just 1 implementation.

The test programs are parsed and evaluated and pass if no error occurs. Without negative test cases, this means that an engine that does nothing would trivially pass the tests. To allow negative test cases, they use special comments with metadata. The test runner would need to interpret this metadata and evaluate the test accordingly. This metadata can be compared to the test expectations of SPT.

This approach to testing only works if the language is expressive enough to evaluate itself. To do so, it would need things like type checks to check its own types and value comparisons to test the dynamic semantics and name resolution. Not all domain specific languages have these features, and even if they do, the test runner that interprets the metadata would still need to be implemented. SPT can be seen as the test runner for languages created by the Spoofax language workbench. Which means developers don't need to write their own test runner.

Java – the process for testing Java implementations works quite similar to that of JavaScript. First, they make a specification for the language (in the form of Java Specification Requests). Then, they create a Technology Compliance Kit (TCK [21]). This is basically a set of test suites and a harness to execute those tests.

The harness can be configured, but it does impose some constraints on the Java implementation to allow the execution of the tests. The test cases themselves provide metadata about the test in the comments above the test case, and the harness interprets this to know how to execute it.

6.3 Test generation

In the previous section we saw how some languages are tested by running manually written tests cases against a language implementation. It is up to the developers who wrote these test cases to ensure they cover certain properties of the language implementation that need to be verified. Another approach to testing properties is to generate inputs instead of manually crafting them and then check to see if the property holds. An example of this can be seen in QuickCheck [22]. This is a Haskell tool that allows the developer to specify properties of their implementation. QuickCheck will then generate inputs and check if the property holds for these inputs.

If we want to apply such a technique to testing in language engineering, we need to know which properties to test and what inputs should be generated. This is pretty clear in SPT: a test case consists of an input program and test expectations. The test expectations would be the properties to test, e.g. is the syntax valid (parsing) or are the static semantics valid (analysis). The input that needs to be generated is the input program. The most basic example of a generated test would be to generate random strings and see if they can be parsed. This wouldn't be very helpful as a test, because the generator wouldn't know if the generated string is supposed to be valid syntax or not.

The main question for test generation in language engineering is to generate input programs of which we know certain properties. For example, Csmith [23] can generate C programs with valid syntax and semantics. So C compilers can be tested by seeing if they can compile the generated programs.

To bring this research to the Spoofax language workbench the Spoofax Generator (SPG [24]) was created. This tool can generate syntactically valid input programs based on the SDF grammar and can also generate well-typed input programs based on the NaBL2 type definitions.

Chapter 7

Future work

In the evaluation we noted there are still two important unsolved problems with both the previous and the new SPT. These are support for dynamic fragment markers, and editor services (most notably syntax highlighting inside fragments). These problems are excellent candidates for future work, as they require both work on SPT and on Spoofax itself. In this chapter we will describe these problems in a bit more detail. We also add another interesting topic for future work to this list: the sandboxing of test execution. This topic was discovered after the new SPT had been used in several actual language projects.

7.1 Dynamic markers

Support for user-defined markers to delimit fragments is mainly a theoretical problem. The two different markers SPT offers (`[[[` and `[[[[`) are sufficient to test most languages. However, if we truly want to approach the goal of an LPTL and become language specification agnostic, we need to allow the user to define what delimiters they want to use, so any conflict with the syntax of their language can be avoided.

To be able to solve this problem, we need add support for parameterized grammars to Spoofax. To do so, we would need to extend the SDF meta DSL as well as the parsing algorithm.

7.2 Editor services

As discussed in the previous chapter SPT has never properly supported the editor services of the language under test, but with the new SPT we took a step back in terms of syntax highlighting of fragments. This issue has high priority for the users of SPT as it makes the writing of test cases a lot less enjoyable.

To be able to add syntax highlighting to fragments again we would need Spoofax to allow syntax highlighting to change after the analysis phase. This would be required for us to change the highlighting of the fragments, as their highlighting is currently determined during the analysis phase of SPT. An added benefit would be that highlighting during analysis actually allows semantic highlighting, instead of just syntax highlighting.

The approach to using the highlighting of the language under test would be the same as using the other editor services. To be able to add these, we would need a public API from Spoofax to access these editor services from the language under test. The second step would be to provide a custom implementation for these APIs for SPT itself. For example, Spoofax currently uses the language's grammar and parse table to provide content completion for a language. We would need to override this for SPT, as content completion requests within fragments would have to be delegated to the language under test, while requests outside fragments would have to be handled using the default parse table approach.

For content completion this public API to call the service for the language under test exists, but the second part, overriding its behaviour for SPT, would require quite some changes to the new Spoofax.

7.3 Sandboxing

When SPT was used in practise, it was used to test a language that performed a lot of side effects during the analysis phase. It generated several custom artifacts and reused them if they already existed. As SPT currently has no way to track these side effects, it also can't undo them after a test has executed. This caused some test cases to exhibit different behaviour based on the order in which they were executed and which custom artifacts already existed and which did not.

To combat this problem, we would like to execute tests within a sandbox. That would not only allow us to undo changes to the environment after each test case, but would also provide a layer of security for our automated testing approach.

Chapter 8

Conclusion

We have noted some of the problems with the previous SPT in chapter 3, and proposed a solution for these problems in chapter 4. This proposed solution solves many of the problems we identified, and can serve as a good basis for the further development of Spoofax and SPT. It is already being used in practise for interactive test case design, regression testing, continuous integration, and automated grading.

Overall we met all of the goals that we started out with. The new solution works with Spoofax versions above 2.0, has an extensible and modular architecture, and meets the requirements for the new use cases. We did, however, run into some problems and limitations while designing this new SPT.

The first problem which remains unsolved is editor services inside test fragments. For these editor services we'd need both a public API from Spoofax to execute those editor services on the language under test, and we'd need to be able to override the default implementation for SPT. If the editor service is performed outside of a test fragment, the default service can be used, otherwise we need to delegate it to the language under test. Future work would be to research how this can be added to the new Spoofax architecture. The most important editor service to focus on would be syntax highlighting, or preferably even semantic highlighting.

Another problem is sandboxing of test execution. SPT creates a different representation of reality for test execution, as the text in the test suite file is not what the parser and analyser of the language under test are given. For this reason, we need to map these locations between the 'realms' of SPT and the language under test. We could take this one step further and fully separate the execution environments, allowing us to sandbox the test execution. This would be very helpful for language implementations that produce side effects during the analysis phase.

We believe our proposed solution offers a good basis to build this future research on.

Bibliography

[1] NIST. The economic impacts of inadequate infrastructure for software testing. `http://www.nist.gov/director/planning/upload/report02-3.pdf`, May 2002.

[2] Lennart C. L. Kats, Rob Vermaas, and Eelco Visser. Integrated language definition testing: enabling test-driven language development. In Cristina Videira Lopes and Kathleen Fisher, editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 139–154. ACM, 2011.

[3] K. Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[4] Lennart C. L. Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In William R. Cook, Siobhn 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.

[5] Martin Fowler. Language workbenches: The killer-app for domain specific languages? 2005.

[6] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The state of the art in language workbenches. In Martin Erwig, Richard F. Paige, and Eric Van Wyk,

editors, *Software Language Engineering*, pages 197–217, Cham, 2013. Springer International Publishing.

[7] Boris Beizer. *Black-Box Testing : techniques for functional testing of software and systems.* Wiley, 1995.

[8] Paul Hamill. Chapter 3: The xunit family of unit test frameworks. In *Unit Test Frameworks.* O'Reilly, 2005.

[9] Ecma TC39 committee. Official ecmascript conformance test suite. `https://github.com/tc39/test262`, 2018. [Online; accessed 2-August-2018].

[10] Jan Heering, P. R. H. Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism sdf - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[11] Gabril D. P. Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In Krzysztof Czarnecki and Grel 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.

[12] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[13] Mark G. J. van den Brand, H. A. de Jong, Paul Klint, and Pieter A. Olivier. Efficient annotated terms. *Software: Practice and Experience*, 30(3):259–291, 2000.

[14] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[15] Martin Fowler. Continuous integration, 2006. `http://www.martinfowler.com/articles/continuousIntegration.html`, May 2006.

[16] Pyret. `https://www.pyret.org/index.html`. Accessed: 2019-12.

[17] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

[18] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113–128, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[19] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Easy meta-programming with rascal. In Joao M. Fernandes, Ralf Lmmel, Joost Visser, and Joo Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer, 2009.

[20] JetBrains. Mps: The domain-specific language creator by jetbrains. `https://www.jetbrains.com/mps`, 2000. [Online; accessed 28-December-2019].

[21] Oracle Corporation. Tck tools and documentation. `https://jcp.org/en/resources/tdk`, 2019. [Online; accessed 29-December-2019].

[22] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[23] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[24] Martijn Dwars. Random term generation for compiler testing in spoofax. 2018.