A Generic and Automatic Test Strategy for Compiler Testing

Master's Thesis

André Miguel Simões Dias Vieira

A Generic and Automatic Test Strategy for Compiler Testing

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

André Miguel Simões Dias Vieira
born in Loures, Portugal

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

A Generic and Automatic Test Strategy for Compiler Testing

Author: André Miguel Simões Dias Vieira
Student id: 1263161
Email: andre.s.d.vieira@gmail.com

Abstract

Domain-specific Languages (DSLs) are languages specifically tailored for an application or expert domain. These can be implemented as compilers, which check the correctness of an input program and translates it to a target language. Manual testing of compilers is a time consuming and labor intensive task. This motivates the development of approaches to facilitate the quality assurance process.

In this thesis we present an automatic and generic test strategy for the generation of test cases for Spoofax developed compilers. We use a program generator to generate large syntactically correct programs from Syntax Definition Formalism (SDF) grammars. Additionally, we improve the program generator with an expansion of our generation algorithm to use Name Binding Language (NaBL) modules to generate partial name correct programs. We also provide a DSL to define error fixes that are used to attempt the repair of static semantic errors reported after compilation. After program generation we use a partial oracle to automatically detect failures during the invocation of the compiler. Finally, we provide a heuristic to reduce the size of generated programs, whilst preserving their failure inducing behavior.

This test strategy was used to generate test cases for WebDSL, a DSL targeting the domain of developing dynamic web applications with a rich data model. The generated test cases unveiled eleven unique faults in the analysis phase of compilation. These were reported together with the programs reduced by our program shrinking heuristic and they were positively received by the WebDSL development team.

Thesis Committee:

Chair:	Prof. Dr. E. Visser, Faculty EEMCS, Delft University of Technology
University supervisor:	Dr. G. Wachsmuth, Faculty EEMCS, Delft University of Technology
Committee Member:	Dr. M. Loog, Faculty EEMCS, Delft University of Technology

Preface

I am grateful to my supervisor Guido Wachsmuth for his availability, advice and feedback, without whom the writing of this thesis would not have been possible. I would also like to thank Eelco Visser, Lennart Katz, Danny Groenewegen, Vlad Vergu, Elmer van Chastelet, Chris Melman, Gabriël Konat and Christoffer Gersen for the discussions and their help with Spoofax related issues and Kris Pamphilon for the creation of the cover page illustration. Finally, I would like to express my gratitude to my family and friends for supporting me throughout my time at the university.

André Miguel Simões Dias Vieira
Delft, the Netherlands
August 20, 2013

Contents

List of Figures

Chapter 1

Introduction

Domain-specific Languages (DSLs) are languages specifically tailored for an application or expert domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application [30]. DSLs are becoming increasingly popular lately and the number of small customer base DSLs has been growing rapidly.

DSLs can be implemented as compilers, which check the correctness of the input program and translate it to a target language. Figure 1.1 illustrates the traditional compiler pipeline and its three components [1]: the parser, the analyzer and the generator. The parser checks whether the input program is well-formed, that is syntactically correct. If this is the case, the parser transforms the input program into a corresponding tree structure and passes it on to the analyzer, otherwise it reports the detected parse errors. This is called the parsing phase. Parsers are traditionally generated from the language's grammar. In contrast to the two succeeding components, the analyzer and generator, which are traditionally hand written. The analyzer analyzes names and types in the program and checks whether a program

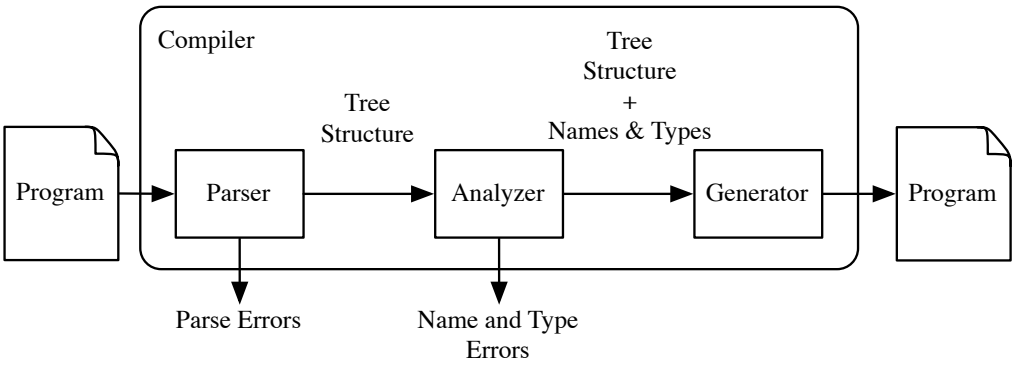


Figure 1.1: The generic layout of a compiler and its three components: parser, analyzer and generator. These are respectively responsible for the parsing, analysis and generation phases of compilation.

is well-named and well-typed. Such a program will be passed on to the generator together with the gathered names and types. Otherwise name and or type errors are reported. This phase is called the analysis phase. The next and last phase is the generation phase. In this phase, the generator translates the input program into a program in the target language.

Compilers are programs and, like any other program of significant size, they are prone to contain faults. These faults jeopardize the quality of the compiler, which would diminish the interest in a DSL. To validate compilers, just as for any other software, engineers resort to two methods: formal proofs and testing. We will focus on the latter of the two, which involves the execution of the compiler with various inputs and in different states with the goal to reveal failures.

Exhaustive testing, that is running the compiler through all possible input-output combinations, is unfortunately impossible for non-trivial compilers due to the infinite domain of possible combinations. Instead, the quality assurance process involves exercising software to the extent of achieving an accepted coverage of input-output combinations. Reaching an accepted coverage without finding failures increases the confidence in the software [3].

Language workbenches have been used to facilitate development of DSLs providing a range of tools to define syntax, semantics and editor features. This enabled smaller development teams to develop such languages. Though these smaller teams, smaller customer bases and lack of testing facilities can lead to a poor quality assurance process. In contrast, general-purpose languages such as Java and wider used DSLs such as HyperText Markup Language (HTML), a DSL for text markup in world wide web pages, have larger development teams, larger testing teams and larger customer bases. Lately the need for better testing of these smaller scoped languages has been recognized and research has been turned towards approaches to enrich these workbenches with testing tools [27, 17].

Since manual testing is often a time consuming and labor intensive task, especially when your program is a compiler [4], we focus on fully automated test case generation to extend testing of DSLs. Automated testing should provide the test suite with additional test cases, strengthening the DSL quality assurance process at a low cost. Building such an automated test case generator for every DSL compiler would increase this cost significantly, invalidating the solution. This is why we need a language independent, that is generic, testing strategy for test case generation.

Generic automated test case generation requires automated generation of input data. This input data can be classified into four embedded subsets [23]:

1. chains of terminal symbols,
2. syntactically correct programs, that is the programs which are well-formed and should pass the parsing phase,
3. statically correct programs, that is the programs that should pass the analysis phase,
4. programs that should pass all phases of compilation.

Due to the staged nature of compilers, each set specifically targets different parts of the compiler. The first set consists of both parsable and non-parsable programs, whereas the second set should only contain parsable programs. Distinction between these is thus

essential for testing parsers. The third set contains programs for which analysis should never report errors. Distinguishing between sets two and three is ideal for testing the analysis phase of the compiler. Finally, set four contains programs which the generator should translate the program flawlessly. These can be used to test the generator.

Parser generators do not suffer from the same testing deficiencies as our target DSL compilers, since they are re-used among compilers, have a larger customer base and are exposed to more intensive testing. This leads us to focus on the two later phases of compilation, making our system under test the analyzer and generator. Consequentially, we focus on the generation of input data belonging to sets two, three and four.

Summarizing, the main goal of this thesis is to research test strategies to generically and automatically test the analyzer and generator of DSL compilers. We present such a test strategy and implement it for languages developed with Spoofax [18], a workbench for language development. Finally, we use the test strategy to automatically generate test cases for WebDSL [38]. WebDSL is a DSL targeting the domain of developing dynamic web applications with a rich data model. WebDSL consists of several sub-languages for data models, web pages, business logic, access control and work-flow. The WebDSL compiler ensures consistency across its sub-languages [9]. The compiler currently consists of approximately 25.000 lines of Stratego code and 15.000 lines of Java code and has been developed on since 2007 with stable Year-Over-Year commits. The work of its small development team amounts to an average of 2.0 fte per year. This small development team and complexity of the language and compiler make WebDSL an ideal test subject for our test strategy.

1.1 Research Questions

In this thesis we aim to answer several research questions, all originating from this main question: How can we generically automatically generate test cases for DSL compilers? Our goal is to test hand written parts of the compiler through generation of input from existing language artifacts. We now discuss these research questions in detail.

RQ1: How can syntactically correct programs be generated generically from language specifications?

Our goal is to generically test hand written components of a compiler and since parsers are traditionally generated from a language's grammar, syntactically incorrect programs are of no use and generation of these is counterproductive. To force generation of syntactical correct programs the generator requires knowledge of the language's syntax. Moreover, the generator should be language independent since we require it to generate programs for different languages using only a declarative specification of the language, readily available. The logical decision for such a specification is the language's grammar, sharing the same source for both parser generation and program generation.

RQ2: How can big programs be generated generically whilst avoiding combinatorial explosion?

Testing involves finding failures and failure finding likelihood is presumed to be related to the size of programs used as test input [40]. Encouraged by this presumption we aim to generate big programs in a random manner. This involves choosing production rules randomly. However, non-trivial grammars contain cyclic/recursive definitions and iteration operators which can lead to exploding recursion during generation. We must thus limit generation to avoid combinatorial explosion whilst generating large programs. At the same time, due to our goal to have a generic approach, generation must be language independent and not embed any language specific heuristics.

RQ3: How can statically semantically correct programs be generated generically?

Our goal is to test all hand written parts of a compiler. The previous questions covered the generation of syntactically correct programs which should always pass the parsing phase, thus reaching the analysis phase. In order to reach the generation phase we require the generation of statically semantically correct programs. If program generation only uses the language's syntax definition, the probability of generating statically semantically correct programs is too low and the compiler's generator will never be tested. Moreover, we argue that generating (partially) statically semantically correct programs would improve testing of the analysis phase too, especially when dealing with languages with a complex analysis phase.

To guarantee generation of statically semantically correct programs the generator requires knowledge of the language's static semantics. Again with language independence in mind we refrain from using language bound heuristics and require the existence of a language artifact describing these static semantics.

RQ4: How can test runs be evaluated to determine success or failure when testing compilers in a generic automated manner?

Automation of testing entails the automation of determining whether a test passes. This requires the existence of an oracle. Such an oracle is composed by a generator for the expected result and a comparator to compare it to the actual result. In the case of a generic automatic test strategy, the oracle would have to be able to generate an expected result for each generated program. The traditional approaches to this problem, differential testing or an extra computational model, are often not applicable. There is typically only one implementation and no computational model is available, due to small developer teams.

RQ5: How can we assist the language engineer in relating failures found automatically to faults in the compiler?

A language engineer should easily be able to understand a failing test and to locate its cause in the implementation. This encourages the use of the test strategy and greatly increases testing productivity. However, our goal to randomly generate big programs will most likely make the failure inducing programs harder to read. This negatively influences the effort required by the language engineer to relate the failure back to the causing fault. The goal to remain generic prevents us from using language specific reductions and forces

us to find generic approaches. This leads us to research a generic program shrinking heuristic for reducing a failure inducing program whilst preserving its failure inducing behavior.

1.2 Contributions

Our contributions consist of a fully automated and generic test strategy for generation of syntactically correct and partial name correct programs using a grammar and a declarative name binding definition, the application of partial oracles to test DSL compiler robustness and data regarding the application of this test strategy to WebDSL. We show that for languages with strong static semantics, generation of even partial name correct programs leads to an higher failure finding likelihood than the generation of syntactically correct programs.

We also contributed to both Spoofax and WebDSL projects with a a code coverage tool for the Stratego java back-end compiler, a program shrinking heuristic that preserves failing behavior, valuable data regarding coverage achieved by WebDSL's existing test suite and reports of new failures for the WebDSL project.

1.3 Thesis Outline

This thesis is organized as follows. We establish the terminology of testing, programming languages and compilers used throughout this thesis and describe Spoofax, the language workbench we use to implement our approach, in Chapter 2. In Chapter 3, we present our automated and generic test strategy for DSL compiler test case generation. In Chapter 4, we describe the implementation of our test strategy for Spoofax developed compilers, together with the coverage measurement tools we developed for grammar and code coverage. We used this implementation to generate test cases for WebDSL and to evaluate the test strategy, we present the results in Chapter 5. In Chapter 6, we give an overview of the related work and we conclude this thesis with conclusions and future work in Chapter 7.

Chapter 2

Preliminaries

Testing is the subject of software engineering covered by this thesis, whereas programming languages and compilers are the domain testing is applied to. This chapter establishes the terminology used throughout this thesis. The last section of this chapter introduces Spoofax [18], a language workbench for the development of DSLs which we used to realize and evaluate the developed strategy,.

2.1 Testing

This section is based on Binders's terminology for testing found in "Testing object-oriented systems: models, patterns, and tools." [3].

Software testing is the practice of executing an implementation using various combinations of input and states attempting to reveal failures. A *failure* is the manifested inability of a system or component to perform a required function within specified limits. A failure can vary from incorrect output, abnormal termination, or unmet time and space constraints. These failures originate from a software *fault*, which is either missing or incorrect code. The implementation being tested or *System Under Test (SUT)* in this case is a compiler, in particular the hand written parts of a compiler. Testing a compiler, or any other piece of software, entails running a *test suite*. A test suite is a collection of test cases, typically related by a testing goal or implementation dependency. The test suite's *effectiveness* is classified by its ability to find a failure, whereas its *efficiency* is classified by the cost, that is the amount of man-hours devoted to create them. The *test cases* that compose a test suite specify the pretest state of the SUT and its environment, the test inputs or conditions, and the expected result. The *expected result* specifies what the SUT should produce from the test inputs. When these are equivalent to the actual results of the SUT's execution, the test case is said to *pass*, otherwise it is a *no pass*. These expected results are provided by an *oracle*, the trusted source of expected results. The oracle can be a program specification, a table of examples, or simply the programmer's knowledge of how a program should operate [31].

In the case of *manual testing*, test cases are created by testers. For compilers, this entails creating programs to run through the compiler, providing the expected output according to the language specification and developing a comparator to verify that this output and

actual results are equivalent. The goal of this thesis is to propose a test strategy to raise the effectiveness of the manually developed test suite by expanding it with failure inducing input through automated test case generation without reducing the efficiency. A *test strategy* is an algorithm or heuristic to create test cases from a representation, an implementation or a test model. In our case the developed test strategy for generic and automatic compiler testing consists of an automatic program generator and an automated oracle.

As the test suite is expanded with failure inducing test cases, developers track down their causing faults and make a decision to correct them or not, depending on various variables. The size of a such a test suite is no indicator on how well the SUT is tested. Since exhaustive testing is almost always impossible for non-trivial SUTs, software engineers agreed to assign confidence to software through achieving acceptable levels of coverage. *Coverage* is an abstraction of the measure of tested input-output scenarios usually represented by a percentage. For compilers the two most observed coverage metrics are compiler code coverage and input domain coverage. Compiler code coverage is a percentage of the compiler code exercised by the test suite. This is a white-box method, since it requires knowledge about the internals of the implementation. The input domain coverage, which is the percentage of possible inputs used in the test suite, is a black-box method requiring no access to the implementation. In the case of compilers an abstraction is used since the input domain, the set of all possible programs, is usually infinite.

2.2 Programing Languages and Compilers

This section is based on Aho's terminology for programming languages and compilers found in "Compilers: Principles, Techniques, and Tools" [1]. The focus lies on textual languages and any reference to a programming language will assume it has a textual representation. In Figure 2.1 we illustrate the relation between language aspects, a compiler, a language workbench and the compiler's source code.

Programming languages are specified through syntax and semantics. The *syntax* of a programming language describes the proper form of its programs, that is what words and in what order they need be written to be accepted as a program of the corresponding programming language. *Semantics* describe the program's meaning. Syntax is traditionally specified by *context-free grammars* or *grammars* for short. These consist of productions, terminals, non-terminals and start-symbols. *Productions* specify the manner in which terminals and non-terminals are combined to represent a non-terminal. *Non-terminals* are syntactic variables that denote sets of words and *terminals*, also called *lexicals*, are the basic symbol from which words are formed, e.g. keywords, delimiters, operators, constants or names. A non-terminal that denotes a complete program is called a *start-symbol*. The following production shows how a non-terminal *Stmt*, to the right of the $\rightarrow$, is defined for an if-then-else statement combining non-terminals and terminals, on the left of the $\rightarrow$:

`"if" "(" **Expr** ")" "then" **Stmt** "else" **Stmt**` $\rightarrow$ `**Stmt**`

The non-terminals **`Expr`** and **`Stmt`**, denote expressions and statements respectively and the terminals `if`, `then`, `else` and parentheses are defined in between quotation marks. Context-free grammars are not sufficient to ensure well-formed programs, since programming lan-

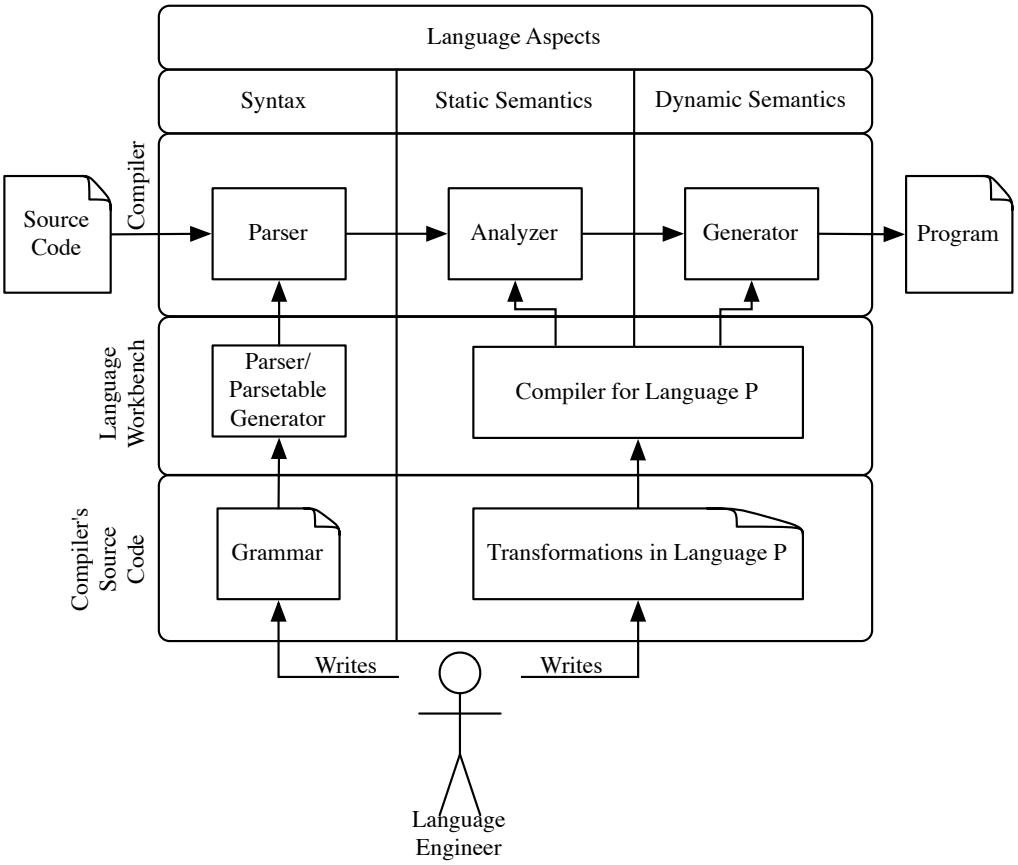


Figure 2.1: An illustration of the relations between language aspects, a compiler, a language workbench and the compiler's source code.

guages have context-sensitive parts such as name binding, types and other constraints. *Static semantics* describes these context-sensitive restrictions on a language. The meaning of the program is defined by the language's *dynamic semantics*. Dynamic semantics can be expressed in different manners, in this thesis the focus lies on *translational semantics*, these translate the input program to a target language. DSLs are usually translated to general-purpose programming languages, such as Java or C.

A *compiler* is a program that reads a program in one language - the source language - and translates it into an equivalent program in another language - the target language. Compilers are implementations of the syntax, static semantics and translational semantics of a programming language. This translation is traditionally performed by three components as illustrated in Figure 1.1, which use transformations to sequentially execute the three corresponding compiler phases: parsing, analysis and generation. A *transformation* is the automatic generation of a target program from a source program, according to a transformation definition. A *transformation definition* is a set of transformation rules that together

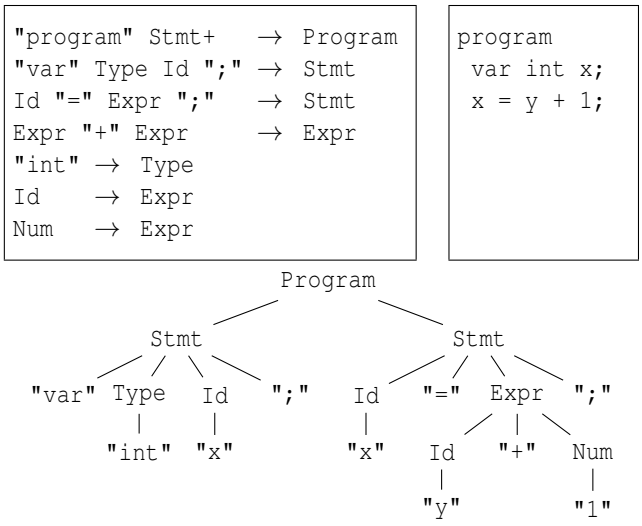


Figure 2.2: Simple context-free productions(top left) for a simple grammar, where the definitions of the `Id` and `Num` symbols are `[A-Za-z0-9_]+` and `[1-9][0-9]*` respectively. And a program in concrete syntax (top right) and corresponding tree representation(bottom).

describe how a program in the source language can be transformed into a program in the target language. *A transformation rule* is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language [21].

The parsing phase, responsible for enforcing syntax, entails checking whether the program is syntactically well-formed or reporting parse errors otherwise. If the program is syntactically well-formed the parser transforms it from its original *concrete syntax*, a sequence of words, into a *Parse Tree (PT)* or a more abstract version, the *abstract syntax tree (AST)*. PTs and ASTs represent the hierarchical syntactic structure of the source program. The difference between the two is that PTs contain complete productions, whereas ASTs abstract over these using constructors to represent a production and omit any context-free terminal, such as keywords and delimiters. This hierarchical representation of the program facilitates future transformations, since tree structures are a more efficient data structure for machines than their textual representation. The corresponding transformation definition of this phase is the language's grammar, where the transformation rules are production rules. These definitions are often used to generate parsers or parse tables, providing the implementation of this phase.

In the top left of Figure 2.2 we show six simple context-free productions. The definitions of the `Id` and `Num` symbols are `[A-Za-z0-9_]+` and `[1-9][0-9]*` respectively. These character sets represent alphanumeric and integer sets. On the top right of Figure 2.2 we show a textual representation of a program belonging to the language defined by this grammar and on the bottom we show the program in its corresponding parse tree representation.

The next two phases, analysis and generation, are responsible for the language's semantics. The first checks whether the tree is statically semantically correct in which case the

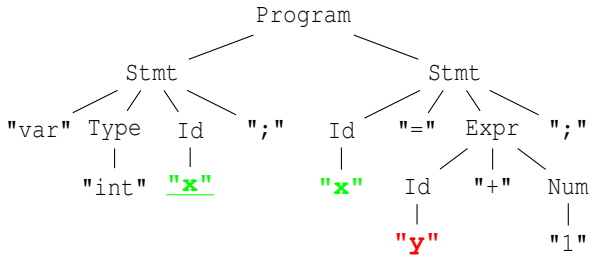


Figure 2.3: The same tree representation of the program used in the previous example in Figure 2.2. Using traditional name binding conventions, names have been color coded to link declarations, which are underlined, to corresponding references. The variable `y` in not defined and a static semantic error would be reported.

tree and gathered semantic information, such as name resolutions and types, is passed on to the next phase, otherwise semantic errors are reported. The second transforms the tree into a program in the target language's concrete syntax. Both transformations are defined by program transformations and are often hand-written by developers. One of the static semantic transformations performed by the analyzer is name resolution. This transformation resolves the implicit static semantic links connecting definitions of names to the nodes that represent their use, yielding a graph with explicit links. However, these connections are more commonly represented in a separate data-structure thus preserving the original tree structure.

In Figure 2.3 we show the tree structure of the program in the previous example and we highlight the names that would be linked by name resolution. The declaration of the variable name `x` is defined in the first statement, where it is underlined, and used by the second statement as a variable reference. The use of variable `y` in the second statement would lead to a static semantic error report, since the variable `y` is never defined.

2.3 Spoofax

Spoofax [18] is a language workbench that provides language engineers with tools to develop compilers for textual DSLs and to develop Eclipse editor plugins with features such as syntax highlighting, error highlighting and content completion. Figure 2.4 illustrates how the different Spoofax components relate to the implementation of compilers, as explained in the previous section. Spoofax encompasses several meta-languages to define DSLs. This includes the Syntax Definition Formalism (SDF) and the transformation language Stratego. Lately, new research towards declarative semantics has led to the development of Spoofax' Name Binding Language (NaBL), a declarative DSL for name semantics. We will now discuss these meta-languages in more detail.

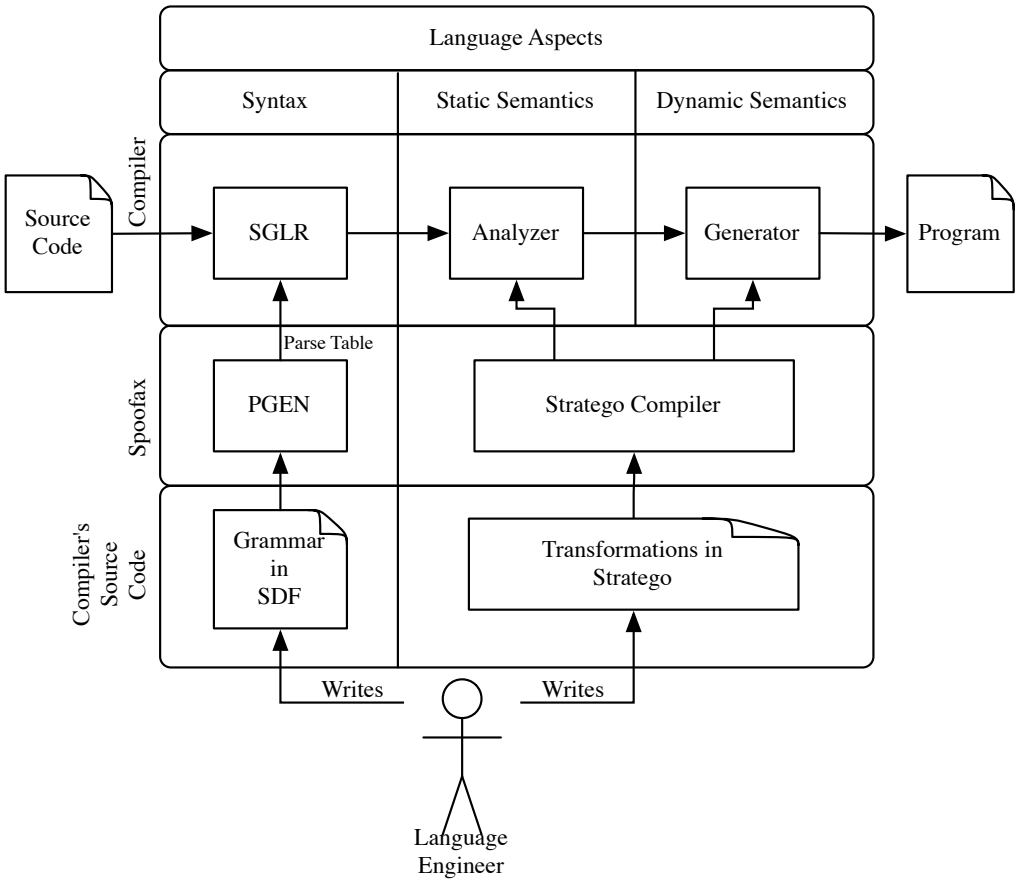


Figure 2.4: An illustration of the Spoofax language workbench in relation to language specifications.

2.3.1 Syntax Definition Formalism (SDF)

SDF [37] enables language engineers to describe syntax in a declarative and highly modular manner, combining lexical and context-free syntax into one formalism. SDF grammars are transformed into parse tables by the parse table generator PGEN and used as input for the Scannerless Generic LR parser SGLR. ASTs resulting from parsing using parsers generated from SDF grammars are composed of Annotated Terms (ATerms). These are representations of productions or terminals and can be composed of: strings, integers, lists, tuples, constructor applications and annotations.

SDF essentially describes syntax using two different syntax types: *lexical* and *context-free*. Lexical syntax describes the allowed format of terminals which upon parsing are included in the AST. Rules describing these terminals have a format that resembles regular expressions, using character sets and ranges. Context-free syntax rules are used to describe non-terminals and how they are combined to represent accepted words. These rules are

composed of symbols representing either terminals or non-terminals, strings representing literals or keywords and structural operators to indicate optionality or iteration. SDF productions take the form `p1...pn -> s { Annotations }` and specify that a symbol `s` can represent a sequence of strings matching symbols `p1` to `pn`. Also, productions can be annotated with extra properties.

In Figure 2.5 we show some parts of the WebDSL's SDF modules on the top followed by three representations of the same WebDSL program. The first is a concrete syntax representation, also often referred to as source code. The second is the corresponding tree representation of the AST and the third is the textual representation of the AST using ATerms. In the last two we show ATerms representing productions in a tree structure. For instance, `ApplicationDefs` with two sub-trees is the abstract representation of the application production in the first module. These production abstractions are called constructors and are described using a simple `"constructor"` annotation in the language's grammar.

SDF also enables declarations of disambiguation rules. These are required to disambiguate over multiple AST representations for the same concrete syntax. Take the expression `1+2*3`, this could be parsed either as `Mul(Add(1,2),3)` or `Add(1,Mul(2,3))`. Such operator precedence is specified in SDF through context-free priorities using the $>$ operator, which is transitively closed, that is if $A > B$ and $B > C$, then $A > C$. In Figure 2.6 we show a part of the WebDSL's SDF grammar for expressions. There we see disambiguation rules that enforce the operator precedence of multiplication and division over addition and subtraction. These would thus disambiguate the previous expression and parsing would yield the correct `Add(1,Mul(2,3))`.

For operators with the same precedence, such as addition and subtraction, parsing order can be determined using operator associativity. SDF defines three associativities:

- left-associativity, which parses left-to-right and is defined with the `left` or `assoc` keywords,
- right-associativity, which parses right-to-left and is defined with the `right` keyword,
- non-associativity, which prohibits associativity and is defined with the `non-assoc` keyword.

In Figure 2.6 we see operator associativity rules for single productions with the keywords as production annotations, and for groups of productions, with the keywords at the beginning of the group. These rules define addition and subtraction as left-associtative in relation to each other and themselves. These rules allow the correct parsing of the expressions `1+2-3` and `1+2+3` as `Sub(Add(1,2),3)` and `Add(Add(1,2),3)` respectively.

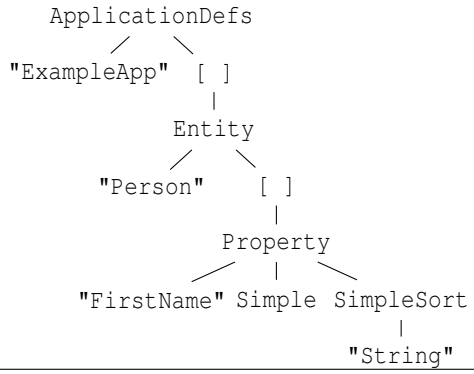
SDF provides more disambiguation means through the `prefer` and `reject` keyword. The first can be used to disambiguate between two possible interpretations, where a production annotated with the `prefer` keyword will take precedence over the other. The second will remove productions from the set of possible interpretations, leaving hopefully a single interpretation.

```
module WebDSL

imports WebDSL-DataModel WebDSL-Lexical
  ...
hiddens context-free start-symbols Unit

exports context-free syntax
  Application -> Unit
  SimpleSort  -> Sort
  Id          -> SimpleSort {"SimpleSort"}

 "application" Id Definition+ Section*
                 -> Application {"ApplicationDefs"}
```

```
module WebDSL-DataModel
...
exports context-free syntax
 "entity" Id "{" EntBodyDecl* "}"  -> Entity {"Entity"}
  Property                         -> EntBodyDecl
  Id PropKind Sort                 -> Property {"Property"}
  Id PropKind Sort ":=" Exp        -> Property {"DerivedProperty"}
  "::"                             -> PropKind {"Simple"}
```

```
module WebDSL-Lexical
...
exports lexical syntax
    [a-zA-Z][a-zA-Z0-9\_]* -> Id
```

```
application ExampleApp
 entity Person {
   FirstName :: String
 }
```




```
Applicationdefs("ExampleApp",
 [Entity("Person",
  [Property("FirstName",Simple(),SimpleSort("String"))])])
```

Figure 2.5: Some parts of WebDSL SDF modules(top), concrete-syntax of an example WebDSL application(second from the top) and its corresponding AST in tree format(second from the bottom) and textual format(bottom).

```
module WebDSL-Action
...
exports context-free syntax
  Exp "+" Exp   -> Exp {"Add", left}
  Exp "-" Exp   -> Exp {"Sub", left}
  Exp "*" Exp   -> Exp {"Mul", left}
  Exp "/" Exp   -> Exp {"Div", left}
  Int           -> ConstValue {"Int"}
  ID            -> Exp {"Var"}

context-free priorities
  {left:
       Exp "*" Exp -> Exp
       Exp "/" Exp -> Exp }
  > {left:
       Exp "+" Exp -> Exp
       Exp "-" Exp -> Exp }
```

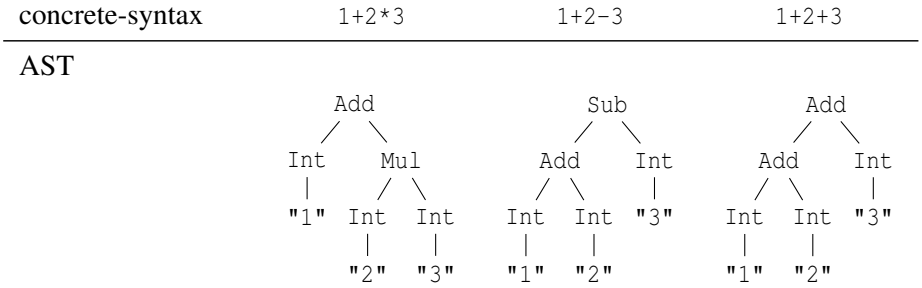


Figure 2.6: An illustration of SDF's disambiguation rules with a part of the WebDSL-Action SDF module(top) and three example expressions and their corresponding correct ASTs (bottom). The first example illustrates operator precedence, the second group operator associativity and the third single operator associativity.

2.3.2 Name Binding Language (NaBL)

Spoofax' Name Binding Language (NaBL) [22] is a declarative language to specify name binding and scope rules of programming languages in a declarative style. Name binding is used to establish relations between a *definition* that binds a name and a *reference* that uses that name. Languages typically distinguish several *namespaces*, that is different kinds of names, such that an occurrence of a name in one namespace is not related to an occurrence of that same name in another. These names live within *scopes*, which restrict the visibility of definitions. Scopes can be nested and name resolution typically looks for definition sites from inner to outer scopes.

A NaBL rule is composed of a term pattern, followed by a list of clauses. The term pattern describes the term the clauses apply to and is composed of variables(`x`) and wildcards(_). A clause specifies either a definition, a reference or a scope and can use variables bound by the pattern.

```
module names

namespaces Entity Property

rules

  ApplicationDefs(a, _ , _) :
    scopes Entity

  Entity(t, body) :
    defines Entity t of type SimpleSort(t)
    defines Property "this" of type SimpleSort(t) in body
    scopes Property

  SimpleSort(t) :
    refers to Entity t
    refers to Entity "Bool"
    refers to Entity "Int"
    refers to Entity "String"
    refers to Entity "Text"
    refers to Entity "Email"

  Property(f, _, t) :
    defines Property f of type t

  DerivedProperty(f, _, t, _) :
    defines Property f of type t

  Var(x) :
    refers to Property x
```

Figure 2.7: A part of the WebDSL NaBL module with name binding rules for the `Entity` and `Property` namespaces.

In Figure 2.7 we show a part of the NaBL module for WebDSL. It shows that the namespace `Entity` is scoped by the `ApplicationDefs` constructor and that the `Property` namespace is scoped by the `Entity` constructor. The `Entity` namespace has the `Entity` constructor as its only definition site and the namespace `Property` can be defined by either the `Entity`, `Property` or `DerivedProperty` constructors. It also shows various references for both namespaces. In Figure 2.8 we show an AST representation of an example WebDSL application. Using the previously mentioned NaBL module we color coded name resolution by giving the same colors to definitions and corresponding references, where definitions are underlined. We see that the `Entity` constructor defines the entity `Person` and contains three constructors that define the following properties: `FirstName`, `Surname` and `FullName`. The last is a derived property and uses references to properties `FirstName` and `Surname`.

```
application ExampleApp
  entity Person {
    FirstName   :: String
    Surname     :: String
    FullName    :: String := FirstName + Surname
}
```

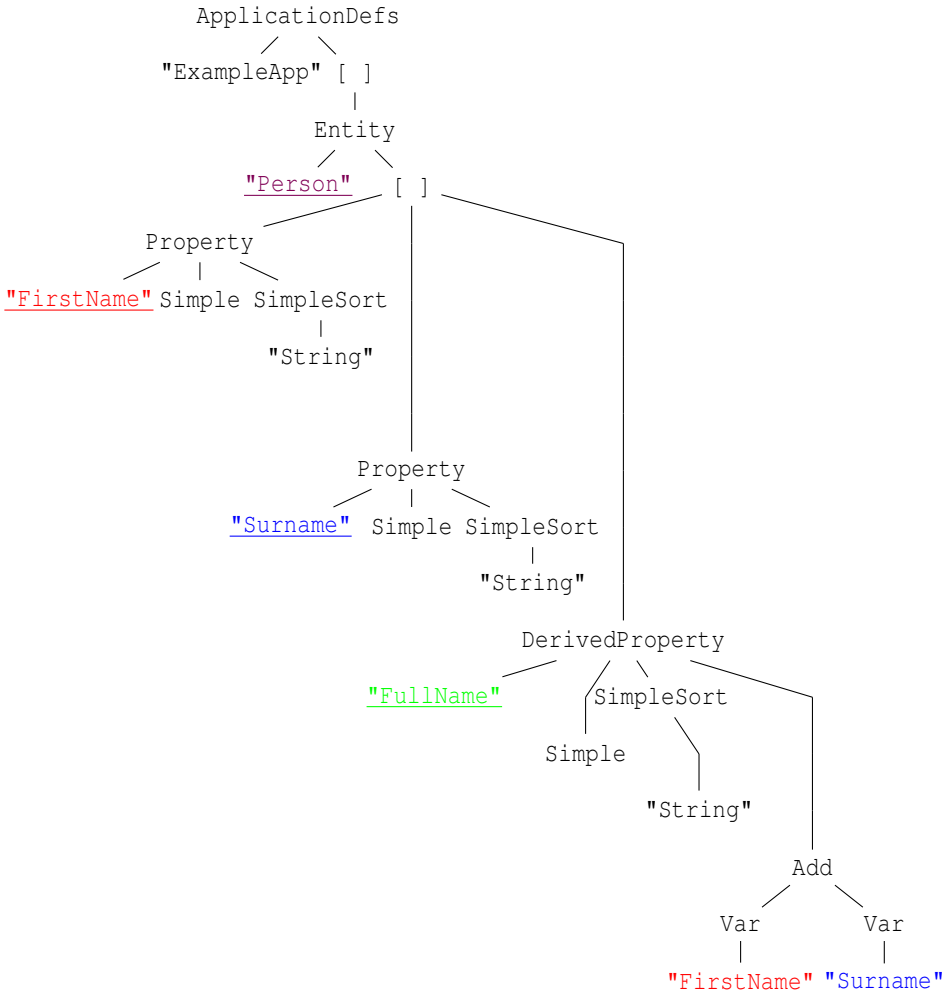


Figure 2.8: An example WebDSL application in concrete syntax(top) and its corresponding AST(bottom), where ATerms involved in name binding are highlighted. Colors link the use of names to their definition sites, which are underlined. The corresponding NaBL module is shown in Figure 2.7.

```
module typechecker
...
rules
 type-of:
   DerivedProperty(pName, Simple(), type, exp) → type
   where
     type' := <type-of> exp
   ; <eq> (type',type)

 type-of:
   Property(pName,Simple(),type) → type

 annotate-type:
   term  →  term{type}
   where
     type := <type-of> term
```

Figure 2.9: Simplified type checking code in Stratego for WebDSL.

2.3.3 Stratego

Stratego [18] is a DSL for program transformation using rewrite rules to specify local transformations and strategies to specify orchestration of local transformations. Stratego supports the development of program transformation infrastructures, domain-specific languages, compilers, program generators and a wide range of meta-programming tasks.

Stratego rewrite rules are composed of a name, a pattern, a rewrite term and clauses. Such a rewrite rule, when invoked using its name, will try to match the input term to the pattern. If this succeeds it proceeds to evaluate the clauses, which in turn are composed of Stratego statements. If all these succeed, the input term is rewritten with the rewrite term. Stratego rewrite rules have two possible outcomes, either success resulting in a rewrite or failure. Stratego allows multiple definitions for the same rewrite rule name, these are applied one at a time, in an undefined order, until one succeeds or if all fail, the invocation of the rule fails.

In Figure 2.9 we show two definitions for the rewrite rule `type-of`. Now assume the following Stratego statement:

```
<type-of> Property("FirsName", Simple(), "String")
```

This statement represents an invocation of `type-of` with the `FirstName` property of the previous WebDSL example program, see Figure 2.8 as input. The execution of this statement would eventually lead to the execution of the second definition of `type-of`. Even if the execution first tries to apply the first definition, it would fail and proceed to attempt to execute the second. The pattern of the second definition of `type-of`, the `Property(pName,Simple(),type)`, matches the input term and binds the variables `pName` and `type` to `"FirstName"` and `"String"` respectively. This definition has no clauses and rewrites the input term with the `type` variable yielding the `"String"` value.

```
type-of:
  |[ e_1 + e_2 ]| → type
  where
    t1 := <type-of> e_1
  ; t2 := <type-of> e_2
  ; <eq> (t1, t2)
  ; type := t1
```

Figure 2.10: An additional definition of `type-of` using concrete object syntax as a pattern match. Assuming meta language is available where `e_INT` is defined as an object of the `Exp` sort.

A Stratego strategy is defined by a strategy name and a list of statements. As an example we show a strategy to try to apply `annotate-type` to all of the input's elements in a topdown manner:

```
type-of-all = topdown(try(annotate-type))
```

If applied to the tree of the example WebDSL program in Figure 2.8, the result would be a similar tree with type annotated `Property` constructors. The `DerivedProperty` constructor would not be annotated, because the clause of the first rewrite rule would fail due to the absence of a `type-of` definition for the `Add` constructor.

Stratego also offers a feature to define concrete object syntax instead of using ATerms. This enables the use of concrete-syntax of the source language to be used for either pattern matches or term writing. In Figure 2.10 we show a `type-of` definition for the `Add` constructor using concrete object syntax `|[ e_1 + e_2 ]|`, which is equivalent to the pattern match in ATerm format: `Add(e_1,e_2)`. The application of the earlier mentioned strategy `type-of-all` would still yield the same result. Though, the `type-of` definition for the `DerivedProperty` constructor would now enter the new `type-of` definition for the `Add` constructor, binding the variables `e_1` and `e_2` to `Var("FirstName")` and `Var("Surname")` respectively. After which the rule would fail when attempting to invoke `type-of` with the term `Var("FirstName")` as input.

Another Stratego feature is the extension of rewriting rules with *scoped dynamic rewrite rules* to achieve context-sensitive rewriting without the added complexity of local traversals and without complex data structures. Dynamic rules are normal rewrite rules that are defined at run-time and that inherit information from their definition context [5]. These dynamic rules can be used to store and retrieve information, using various patterns such as value stores or even key-value pairs.

In Figure 2.11 we show five new rewrite rules for simple renaming of entities, properties and variables for the example WebDSL grammar. Here we see the use of dynamic rules for the storing of entity and property declarations. Now lets assume we define a new strategy called `analyze` in which we will first try to apply `rename` to all the input elements in a topdown manner and then apply the earlier defined `type-of-all` strategy:

```
analyze = topdown(try(rename)) ; type-of-all
```

```
rename:
 Entity(eName, properties) → Entity(eName, properties')
 where
   rules( EntDecl :+ eName → Entity(eName, properties) )
 ; properties' := <map(rename-property(|eName)> properties

rename-property(|eName):
 Property(pName, typeSort, type) →
  Property(pName', typeSort, type)
 where
   pName' := <concat-strings> [eName, ".", pName]
 ; rules( PropertyDecl :+ pName' →
    Property(pName', typeSort, type) )

rename-property(|eName):
 DerivedProperty(pName, typeSort, type, value) →
  DerivedProperty(pName', typeSort, type, value')
 where
   pName' := <concat-strings> [eName, ".", pName]
 ; value' := <topdown(try(rename-var(|eName)))> value
 ; rules(PropertyDecl :+ pName' →
    Property(pName', typeSort, type))

rename-var(|eName):
 Var(vName) → Var(vName')
 where
   vName' := <concat-strings> [eName,".",vName]

type-of:
  Var(x) → type
  where
    property := <PropertyDecl> x
  ; type := <type-of> property
```

Figure 2.11: Stratego code for renaming and typing using rewrite rule parameters and dynamic rewrite rules.

Upon invocation with the AST of the WebDSL example program in Figure 2.8, `rename` will succeed for the entity `Person` in the input tree. This leads to the storing of the declaration of this entity in the `EntityDecl` dynamic rewrite rule with the key `"Person"`. Execution then continues to invoke `rename-property` with the list of properties in this entity as input. Note that here we use the term `eName` as a rewrite rule parameter. These parameters are ATerms that can be given to a rewrite rule as extra input. For all three properties the `PropertyDecl` will store their definitions using the concatenation of the entity name and the corresponding property name separated by a dot as key. The `rename-property` definition for the derived property will further attempt to rename the variables within its definition. Upon completion the property constructors are rewritten with their new names and value, completing the renaming of the program. The next step in execution is the invocation of `type-of-all`, which now will be able to evaluate the type of every typed constructor in the tree. For the `Var` constructor, the invocation of the `PropertyDecl` dynamic rewrite rule will look up the property's declaration and return it. In Figure 2.12 we show the resulting AST with annotated types.

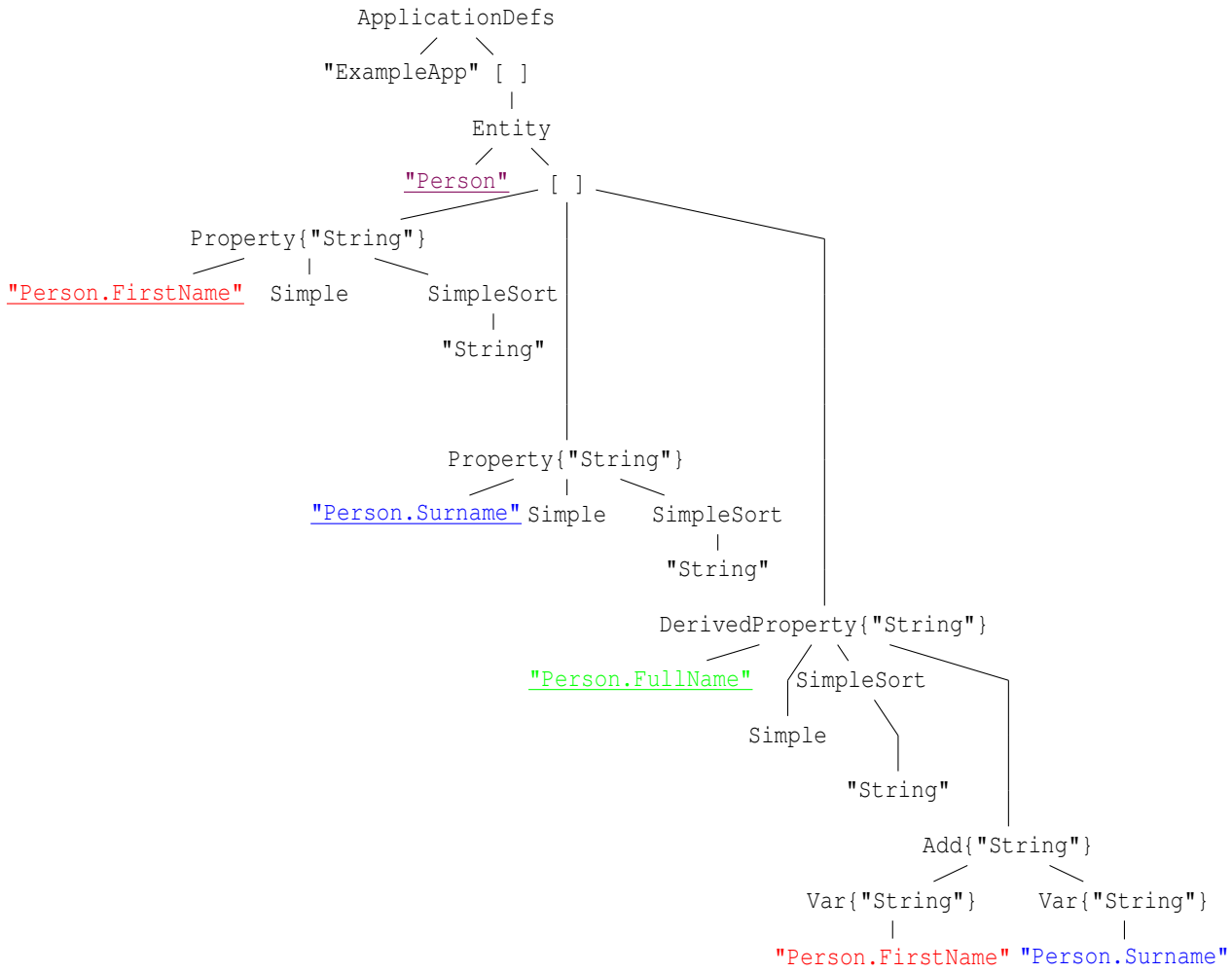


Figure 2.12: The result of the application of the `analyze` strategy to the AST in Figure 2.8. This strategy adds annotations in between curly brackets with the type of the constructor as resolved by the rewrite rules and strategies given as an example throughout this section.

Chapter 3

Generic and Automatic Compiler Testing

Manual testing of compilers is stressful on the development budget of software. To ease the burden and improve the quality assurance process we propose a language independent test strategy to automatically expand existing test suites with failure inducing test cases. This should increase the effectiveness of the test suite without reducing the efficiency, whilst these new found failing test cases may lead to the unveiling of previously unknown faults in the compiler under test.

The basic setup of the test strategy consists of a random program generator using the language's grammar as input and a partial oracle to perform the pass / no pass evaluation. Figure 3.1 shows this basic setup where test case generation begins with the generation of syntactically correct programs using the language's grammar as input. The generated program is then used as input for the compiler in step ①. The outcome of this compilation is forwarded to the partial oracle for a pass/no pass evaluation in step ②. The compilation results together with the generated program and pass/no pass evaluation are stored in step ③ creating the test case.

In Section 3.1 we describe the algorithm for random generation of big syntactically correct programs, in Section 3.2 we describe two approaches that pave the way towards static semantically correct program generation, in Section 3.3 we discuss the partial oracle approach to automated pass/no pass evaluation and in Section 3.4 we describe the program shrinking heuristic to facilitate the understanding of randomly generated failure inducing programs.

3.1 Generating Big Syntactically Correct Programs

In this section we present our algorithm for generation of big syntactically correct programs. First we present the running example used to illustrate the workings of the algorithm then we start to explain the algorithm with the basic version, followed by the expansions made to ensure correctness, termination and an uniform generation.

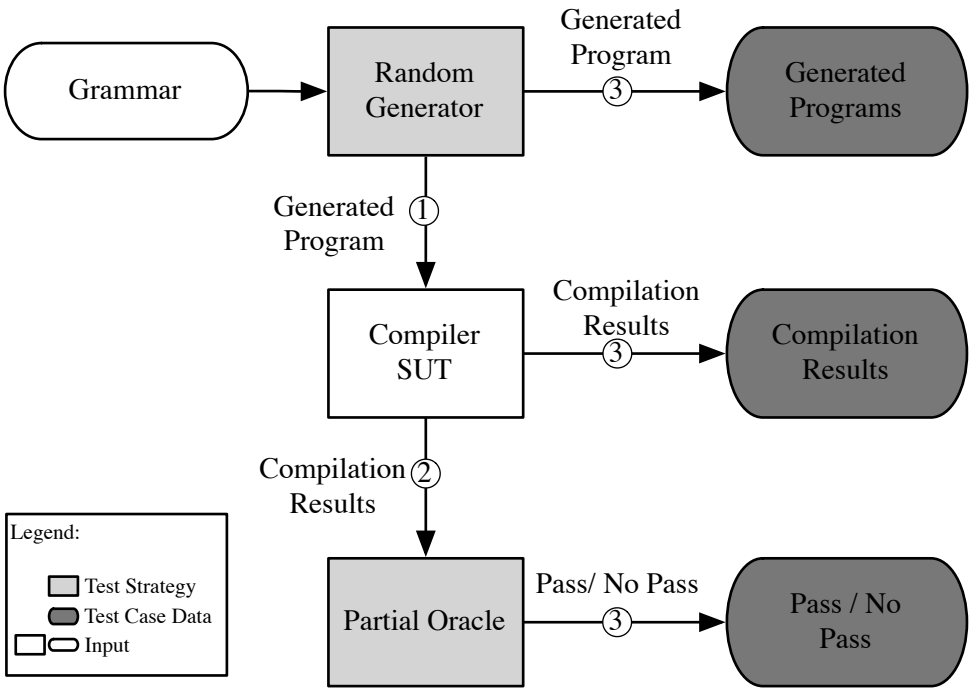


Figure 3.1: Basic test strategy setup for automatic language independent DSL compiler testing. The round corner blocks represent data, input has a white background, test strategy components have a light gray background and the output a darker gray background. In step ① the random program generator generates a large syntactically correct program and invokes the compiler with it. In step ② the compilation results are passed on to the partial oracle for a pass/no pass evaluation. Finally step ③ stores the generated program, pass / no pass evaluation and compiler output, creating the test case.

3.1.1 Running Example

Throughout this section we use the following running example to illustrate the proposed algorithm. The example consists of a simple arithmetic expression language with the terminal symbol `Num`, the non-terminal `Exp` and the productions shown on the left and the priorities and group associativity on the right of Figure 3.2.

The presented algorithm generates programs by first generating a PT and applying the pretty-print operation, transforming it into concrete syntax. Using PTs enables us to ensure the use of brackets where necessary, whereas if ASTs are used a parenthesized pretty-printing transformation has to be available.

Figures 3.3 and 3.4 contain PTs used later on by examples to illustrate the generation algorithm.

```
context-free syntax
 1: Exp "+" Exp -> Exp {"Add", left}
 2: Exp "-" Exp -> Exp {"Sub", left}
 3: Exp "*" Exp -> Exp {"Mul", left}
 4: Exp "/" Exp -> Exp {"Div", left}
 5: Num         -> Exp {"Num"}
lexical syntax
 6: [0-9]+      -> Num
```

```
context-free priorities
{left:
  Exp "*" Exp -> Exp
  Exp "/" Exp -> Exp
} >
{left:
  Exp "+" Exp -> Exp
  Exp "-" Exp -> Exp
}
```

Figure 3.2: Context-free and lexical productions(left) and set of priorities and associativities (right) for the simple arithmetic language.

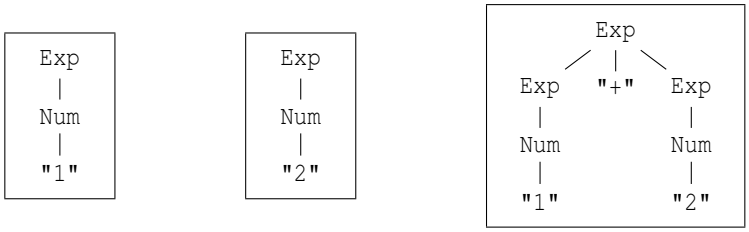


Figure 3.3: PTs belonging to the simple arithmetic language. Note that to generate the rightmost PT, the first two are returned by the generation algorithm's recursive invocations.

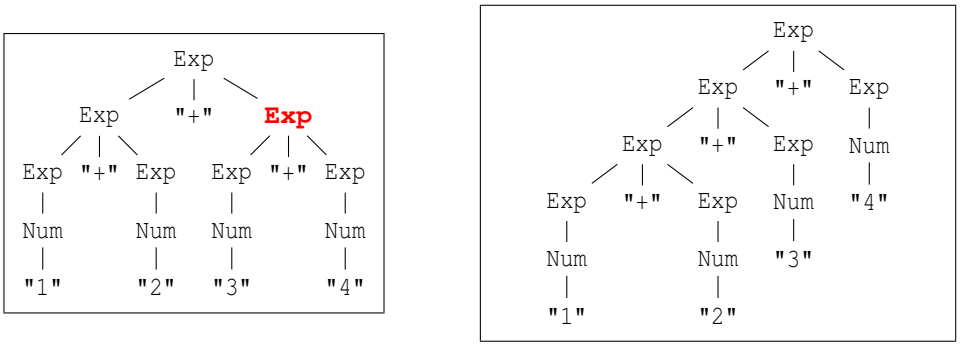


Figure 3.4: Example of tree generation using *generatePT* leading to a conflict with parsing priorities. The leftmost PT could be a result when associativities are not taken into account. A non-parenthesized pretty-print transformation would yield `1+2+3+4` as concrete syntax, which when parsed would yield the rightmost PT, which is not equivalent to the left one.

Input: N : set of all non-terminal symbols.
T : set of all terminal symbols.
P : set of all productions.
S : set of all start symbols.
nTerminals : number of terminals to generate per terminal symbol.
Output: Returns a syntactically correct program according to grammar G.

```
 1: function generateProgram(N,T,P,S,nTerminals)
 2:     for s ∈ T do
 3:         for j = 1..nTerminals do
 4:             t_j := generateTerminal(s);
 5:         end for
 6:         GT(s) := [t_1, ..., t_terminalN];
 7:     end for
 8:     startSymbol := chooseRandomly(S);
 9:     PT := generatePT(startSymbol, N, GT, P);
10:     return prettyPrint(PT);
11: end function
```

Figure 3.5: The base version of the main random generation function that returns a syntactically correct program given the language's grammar and the number of terminals to generate per terminal symbol. The help function *prettyPrint*(PT) is used to transform a PT into concrete syntax by joining its elements, to pretty-print an AST a pretty-print table would have been required.

3.1.2 Basic Algorithm

Our test strategy for automatic language independent compiler testing uses a program generator to provide test input. This generator should yield syntactically correct programs, since syntactically incorrect programs would only test the parser which is traditionally generated. Generating syntactically correct programs ensures the testing of hand-written parts of the compiler, making generation more productive. Generation is also biased towards larger sized programs since we assume these to have an higher likelihood of inducing failures [40].

The program generation starts with the *generateProgram* function, see Figure 3.5. This function has two main inputs, the language's grammar, G, and the number of terminals to generate per terminal symbol, *terminalN*. The function begins program generation by generating these *terminalN* large pools of terminals. Later on during generation any terminal in the PT is chosen from these pools randomly. This promotes name binding links in the generated programs, thus enhancing the coverage achieved of partially static semantic correct programs. The next step is to randomly choose a start-symbol and generate a PT by invoking the function *generatePT*. This function grows a PT from the start-symbol, its root.

Example 3.1.1. For a simple arithmetic grammar the *generateProgram* function is called with $N = [\texttt{Exp}]$, $T = [\texttt{Num}]$, P is the set of production in lines `1` through `6` from Figure 3.2, $S = [\texttt{Exp}]$, its corresponding pretty-print table and $nTerminals = 5$. The function first

Input: s_0 : non-terminal symbol.
N : set of all non-terminal symbols.
GT : collection of generated terminals for every terminal symbol.
P : set of all productions.
Output: PT : a syntactically correct PT with root s_0.

```
1:  function generatePT(s0,N,GT,P)
2:      productions := {p ∈ P | rhs(p) = s};
3:      s1 ... sn → s0 := chooseRandomly(productions);
4:      for i := 1..n do
5:          if si ∈ N then
6:              pti := generatePT(si, N, GT, P);
7:          else
8:              pti := chooseRandomly(GT(si));
9:          end
10:     end for
11:     return constructPT(s0, [pt1, ..., ptn]);
12: end function
```

Figure 3.6: Base version of the algorithm for syntactically correct generation of big PTs. The algorithm generates a syntactically correct PT for the non-terminal input symbol by expanding it with randomly chosen productions. Note furthermore that $rhs(P : \ldots \rightarrow s) = s$, $constructPT(s, [pt_1, \ldots, pt_n])$ constructs a PT with s as root and $[pt_1, \ldots, pt_n]$ as sub-trees and $getTerminal(s)$ will either return a literal, if s describes one, or randomly choose a pre-generated terminal accepted by the regular expression in s.

randomly generates five terminals for the symbol `Num` using its production `[0-9]+` creating $GT(\texttt{Num}) = [\texttt{"1"}, \texttt{"2"}, \texttt{"3"}, \texttt{"4"}, \texttt{"53"}]$. The next step is to choose a start symbol and since S only has one element the choice naturally falls to `Exp`. The function then invokes the *generatePT* with parameters `Exp`, N, GT and P, pretty-prints the result of this invocation and returns it.

The basic version of the *generatePT* function takes four inputs: a non-terminal symbol as the root of the PT, a set of all non-terminal symbols, a dictionary containing sets of generated terminals per terminal symbol and a list of productions. The function starts by collecting all productions from the input grammar that represent the input symbol, that is productions with the input symbol on the right hand side. One of these is randomly chosen to expand the PT by branching the input symbol with the symbols in the left hand side of the chosen production. Next the function recursively invokes itself to construct sub-trees for all non-terminal symbols in the left hand side of the chosen production. Eventually these invocations will return PTs and are used to construct a syntactically correct tree where all leafs are terminal symbols. Note that the generation algorithm assumes the grammar to be complete, such that at any point in the generation productions must exist for any non-terminal used as input for the *generatePT* function. The absence of a such a production is

treated as a smell and reported as a failure.

Example 3.1.2. Continuing the previous example, the *generatePT* function is first invoked with $s_0 =$`Exp`, $N = [$`Exp`$]$, $GT($`Num` $) = [$`"1"`,`"2"`,`3"`,`"4"`,`53"`$]$ and P is the set of production in lines `1` through `6` from Figure 3.2. The first step is to get all productions $p \in P$ for which their right hand side $rhs(p)$ is the non-terminal symbol `Exp`. These are productions `1` through `5` in P and from these a production is randomly chosen, lets assume the function chooses production `1`. The algorithm then loops through the left hand side of the production, $s_1 =$ `Exp`, $s_2 =$ `"+"` and $s_3 =$ `Exp`, to construct $pt_{1...3}$. For s_1 and s_3 the *generatePT* recursively invokes itself with s_1 and s_3 as the new s_0, thus constructing correct sub-trees for these non-terminal symbols. As for s_2 the algorithm uses the fact that $GT($`"+"`$) =$`"+"` and thus $pt_2 =$`"+"`. For the recursive invocations lets assume both choose production `5` and the random choice for $GT($`Num`$)$ falls to `"1"` and `"2"` respectively, this leads to the generation of the left and middle trees in Figure 3.3 respectively, after which the first *generatePT* invocation will return the rightmost tree in Figure 3.3.

3.1.3 Incorporating parsing priorities

Grammar's often contain declarations of parsing priorities for disambiguation. These dictate the order in which to parse nested productions or forbid nesting altogether. A priority declaration is a relation between productions or set of productions and an operator describing this relation as left-,right-, non-associative or higher priority. Note that the higher priority property is transitive, that is if $A > B$ and $B > C$ then $A > C$ where $>$ translates to: higher priority than. Priority declarations have to be taken into account during the generation of a program to ensure syntactically correct generation.

Example 3.1.3. Using the productions of Figure 3.2 the function *generatePT* may lead to the generation of a tree as the left tree in Figure 3.4, which after pretty-printing will yield `1+2+3+4` as concrete syntax. However, due to the left-associative property of addition the parsing of this concrete syntax will yield the rightmost tree in Figure 3.4, which is not equivalent to the previous one. Thus to ensure that generated trees are correct and equivalent to its pretty-print-parsed versions, the generation algorithm must take into account associativities and priorities.

The *generatePT* function is expanded to use such declarations of priority to reject the choice of certain productions for certain branches. Left- and right-associativity forbid productions as direct children of the most right and left branches respectively, whereas non-associativity and higher priority forbid them as direct children of any branch. The function now passes on these rejections which are then removed from the list of productions to choose from. Figure 3.7 shows the expanded *generatePT* function. This expanded version uses the *getRejections* function to acquire a list of lists with rejections per non-terminal branch. All productions with a non-associativity or lower priority relation to the chosen production are rejected for every non-terminal branch; the productions with a left-associativity relation are rejected for the right most branch whereas those with right-associativity are rejected for the most left one. These lists of rejected productions are passed on to the corresponding recursive call of *generatePT* to be filtered from the choice of possible productions.

Input: s_0 : non-terminal symbol.
R : set of rejected productions.
N : set of all non-terminal symbols.
GT : collection of generated terminals for every terminal symbol.
P : set of all productions.
A : set of association and priority properties.
Output: PT : a syntactically correct PT with root s_0.

```
1:  function generatePT(s0,R,N,GT,P,A)
2:      productions := {p ∈ (P \ R) | rhs(p) = s0};
3:      s1 ... sn → s0 := chooseRandomly(productions);
4:      R1 ... Rn := getRejections(s1 ... sn → s0,A);
5:      for i := 1..n do
6:          if si ∈ N then
7:              pti := generatePT(si,Ri,N,GT,P,A);
8:          else
9:              pti := chooseRandomly(GT(si));
10:         end
11:     end for
12:     return constructPT(symbol,[pt1,...,ptn]);
13: end function
```

Figure 3.7: *generatePT* function of Figure 3.6 expanded with priority obeying statements (new code in blue) for syntactically correct generation of large PTs. The new help function $getRejections(p : s_1 \dots s_n \rightarrow s_0, G)$ returns the collection of branch wise rejections for the production p according to the grammars priority and associativity definitions.

Note that again the assumption is made that even with rejections there should always be a production available for expansion of the tree, otherwise this will again be treated as a failure.

Example 3.1.4. Using the earlier example, the *generatePT* function is now also given a set of associativity and priority declarations A. For the simple arithmetic grammar this means that addition and subtraction are left-associative to each other and themselves, the same holds for multiplication and division and the latter have an higher priority. This will cause the choice of an addition production to prompt the generation $R_1 = [production_3, production_4]$ and $R_3 = [production_1, production_2, production_3, production_4]$, which are passed on to the generation of the two branches of the addition. This will lead to the filtering out of productions `1`,`2`,`3` and `4` when generating the rightmost branch of an addition production as the red `Exp` in the left tree of Figure 3.4. Leaving production `5` as the only valid choice, preventing the malformed generation we saw in the previous example.

Input: s_0 : non-terminal symbol.
R : set of rejected productions.
path : set of a symbols generated thus far.
maxRec : maximum recursion.
N : set of all non-terminal symbols.
GT : collection of generated terminals for every terminal symbol.
P : set of all productions.
A : set of association and priority properties.
Output: PT : a syntactically correct PT with root s_0.

1: **function** *generatePT*(s_0,R, *path*, *maxRec*,N,GT,P,A)
2: $path' := symbol \cdot path$;
3: **if** $countDuplicateOccurences(path') > maxRec$ **then**
4: $productions := (TP(s_0) \setminus R)$;
5: $s_1 \dots s_n \rightarrow s_0 := chooseRandomly(productions)$;
6: **else**
7: $productions := \{p \in (P \setminus R)\} \mid rhs(p) = s_0\}$;
8: $s_1 \dots s_n \rightarrow s_0 := chooseRandomly(productions)$;
9: **end**
10: $R_1 \dots R_n := getRejections(s_1 \dots s_n \rightarrow s_0, A)$;
11: **for** $i := 1..n$ **do**
12: **if** $s_i \in N$ **then**
13: $pt_i := generateProgram(s_i, R_i, path', maxRec, N, GT, P, A)$;
14: **else**
15: $pt_i := chooseRandomly(GT(s_i))$;
16: **end**
17: **end for**
18: **return** $constructPT(symbol, [pt_1, \dots, pt_n])$;
19: **end function**

Figure 3.8: *generatePT* function of Figure 3.7 expanded with a maximum recursion control mechanism to ensure termination(new code in blue). The $TP(s)$ function returns a set of minimally depthed syntactically correct PTs with s as the non-terminal root symbol, see Figure 3.9 for the algorithm used to pre-compute these depths and paths.

3.1.4 Ensuring Termination

Grammars with cyclic definitions may lead to recursive use of symbols during generation, especially in cases where the probability of choosing a cyclic production outweighs others. The arithmetic language used in the example, see Figure 3.2, has such a grammar that would probably lead to an infinite generation cycle.

To ensure termination a maximum recursion mechanism is added to the *generatePT* function, see Figure 3.8. This mechanism keeps a branch-wise record of symbols used. If the duplication of symbols reaches a configurable maximum, dubbed *maxRec*, the mecha-

Input: N : set of all non-terminal symbols.
T : set of all terminal symbols.
P : set of all productions.
Output: TP : a dictionary for terminating paths for every non-terminal symbol.
DT : a dictionary for the distance to a terminal symbol from any non-terminal symbol.

```
1:  function computeTPDistance(N,T,P)
2:      ∀s ∈ N | DT(s) := −1;
3:      ∀s ∈ T | DT(s) := 0;
4:      distance := 1;
5:      repeat
6:          toAdd := [];
7:          for ∀p ∈ P do
8:              if {DT(rhs(p)) < 0} & {∀s_i ∈ lhs(p)|DT(s_i) ≥ 0} then
9:                  toAdd := toAdd . p;
10:             end
11:         end for
12:         for ∀p ∈ toAdd do
13:             DT(rhs(p)) := distance;
14:             TP(rhs(p)) := TP(rhs(p)) . p;
15:         end for
16:         distance++;
17:     until toAdd ≠ []
18: end function
```

Figure 3.9: Algorithm to compute distance to a terminal for every production and save the corresponding terminating path.

nism forces termination of the branch by replacing the symbol with a syntactically correct minimal depth sub-tree with the symbol as root. These minimal depth trees are dubbed *terminating paths* and are computed prior to the invocation of the *generatePT* function.

Terminating paths are computed by finding the productions that are the closest to terminal symbols. The distance to a terminal symbol is determined by the amount of non-parallel rewrites required to expand a non-terminal to a sub-tree where all leafs are terminals. A sequence of productions with the shortest distance to a terminal form a terminating path. The distance and their corresponding productions are calculated using the algorithm seen in Figure 3.9. It starts by assigning distance zero to all productions containing only terminals, these are called the level 1 productions. The next step is to raise this level to 2 and loop through all productions to check whether these can be assigned a level. A production can only be assigned a level if it has no previously assigned level and only contains terminals and non-terminals that can be rewritten with a lower leveled production than the current level. These steps are repeated until no new production is assigned a level.

Example 3.1.5. Prior to the generation of a program the random generator runs the *computeTPDistance* function to compute the shortest paths to a terminal from every non-terminal symbol. For the previously used grammar the function will assign production `5` distance 1 in the first loop, since it is the only production with a non-terminal on the right hand side and all symbols on the left for which $DT \geq 0$, thus assigning $DT(\texttt{Exp}) = 1$ and $TP(\texttt{Exp}) = production_5$. During generation the function *generatePT* has a $\frac{4}{5}$ probability of choosing a production with recursive `Exp` symbols. If this happens more than the allowed amount of times for the same branch of generation, the maximum recursion mechanism will detect it and choose a terminating path to terminate the branch. As previously computed, this is production `5` and it will lead to the generation of a sub-tree with depth 1.

3.1.5 Uniform and Realistic Tree Generation

The current random generation algorithm imposes no limits on iteration operators and due to the random production choice, generation is strongly influenced by the grammar's structure. To prevent generation from creating unrealistically and unfeasible large iterations and to distort the grammar's influence on the generation procedure we introduce two new control mechanisms, maximum iteration and maximum size, into the *generatePT* function as seen in Figure 3.10.

The maximum iteration mechanism, lines 12 to 21 of the *generatePT* function in Figure 3.10, is used to prevent the random generation of unrealistic large iterations. Symbols with unbound multiplicity may lead to the generation of for example a function call with tens of thousands of arguments or a program with tens of thousands of functions. The input parameter *maxIt* limits the maximum of the random number chosen for the multiplicity of iteration operators.

The last control mechanism, the maximum size mechanism, limits the size of generated programs with two goals in mind: prevent combinatorial explosion when high recursion and iteration maxima are chosen leading to memory problems and to distort the influence the grammar's structure has on generation. The input parameter *maxSize* denotes the amount of symbols the generation procedure is allowed to generate. The algorithm randomly distributes this number for every branch it generates and decreases it with each symbol it adds to the tree, passing the decreased number to the next recursive call. At the beginning of each call the function checks whether this number has reached zero and if so it returns a terminating path for the current symbol, in the same manner as when the maximum recursion limit is reached.

Apart from the three mandatory control mechanisms, recursion, iteration and size, the strategy offers the possibility to define weights per production to enhance the possibility of it being chosen. This might prove useful for promoting the generation of new language constructs for testing increments of the DSL. These weights are taken into consideration by the *chooseRandomly* function on lines 5 and 8 of the *generatePT* function in Figure 3.10.

Input: s_0 : non-terminal symbol.
R : set of rejected productions.
path : set of a symbols generated thus far.
maxRec : maximum recursion.
N : set of all non-terminal symbols.
maxSize : maximum amount of nodes to add to tree.
maxIt : upper bound for cardinality of symbol iteration.
GT : collection of generated terminals for every terminal symbol.
P : set of all productions.
A : set of association and priority properties.
Output: PT : a syntactically correct PT with root s_0.

1: **function** $generatePT(s_0,R,path,maxRec,maxSize,maxIt,N,GT,P,A)$
2: $path' := s_0 \cdot path$;
3: **if** $countDuplicateOccurences(path') > maxRec$
$V\ maxSize \leq 0$ **then**
4: $productions := (TP(s_0) \setminus R)$;
5: $s_1 \ldots s_n \rightarrow s_0 := chooseRandomly(productions)$;
6: **else**
7: $productions := \{p \in (P \setminus R) \mid rhs(p) = s_0\}$;
8: $s_1 \ldots s_n \rightarrow s_0 := chooseRandomly(productions)$;
9: **end**
10: $R_1 \ldots R_n := getRejections(s_1 \ldots s_n \rightarrow s_0, A)$;
11: **for** $i := 1..n$ **do**
12: $m_i := randomNumber(0, maxSize)$;
13: $maxSize\text{-}= m_i$;
14: **if** $L(s_i) = 0\ V\ (L(s_i) = 1\ \&\ U(s_i) > 1)$ **then**
15: $n := randomNumber(L(s_i), min(U(s_i), maxIt))$;
16: $pt_i := [\,]$;
17: **for** $j := 1..n$ **do**
18: $maxS' := randomNumber(0, m_i)$;
19: $maxSize\text{-}=maxS'$;
20: $pt_i := pt_i \cdot generatePT(s_i, R_i, path', maxS', maxIt, N, GT, P, A)$;
21: **end for**
22: **else if** $s_i \in N$ **then**
23: $pt_i := generatePT(s_i, R_i, path', m_i\text{--}, maxIt, N, GT, P, A)$;
24: **else**
25: $pt_i := chooseRandomly(GT(s_i))$;
26: **end**
27: **end for**
28: **return** $constructPT(symbol, [pt_1, \ldots, pt_n])$;
29: **end function**

Figure 3.10: Expansion of the *generatePT* function to include the maximum iteration and maximum size mechanisms(new code in blue). The help functions $L(s)$ and $U(s)$ return the lower and upper bounds respectively of the symbol's cardinality.

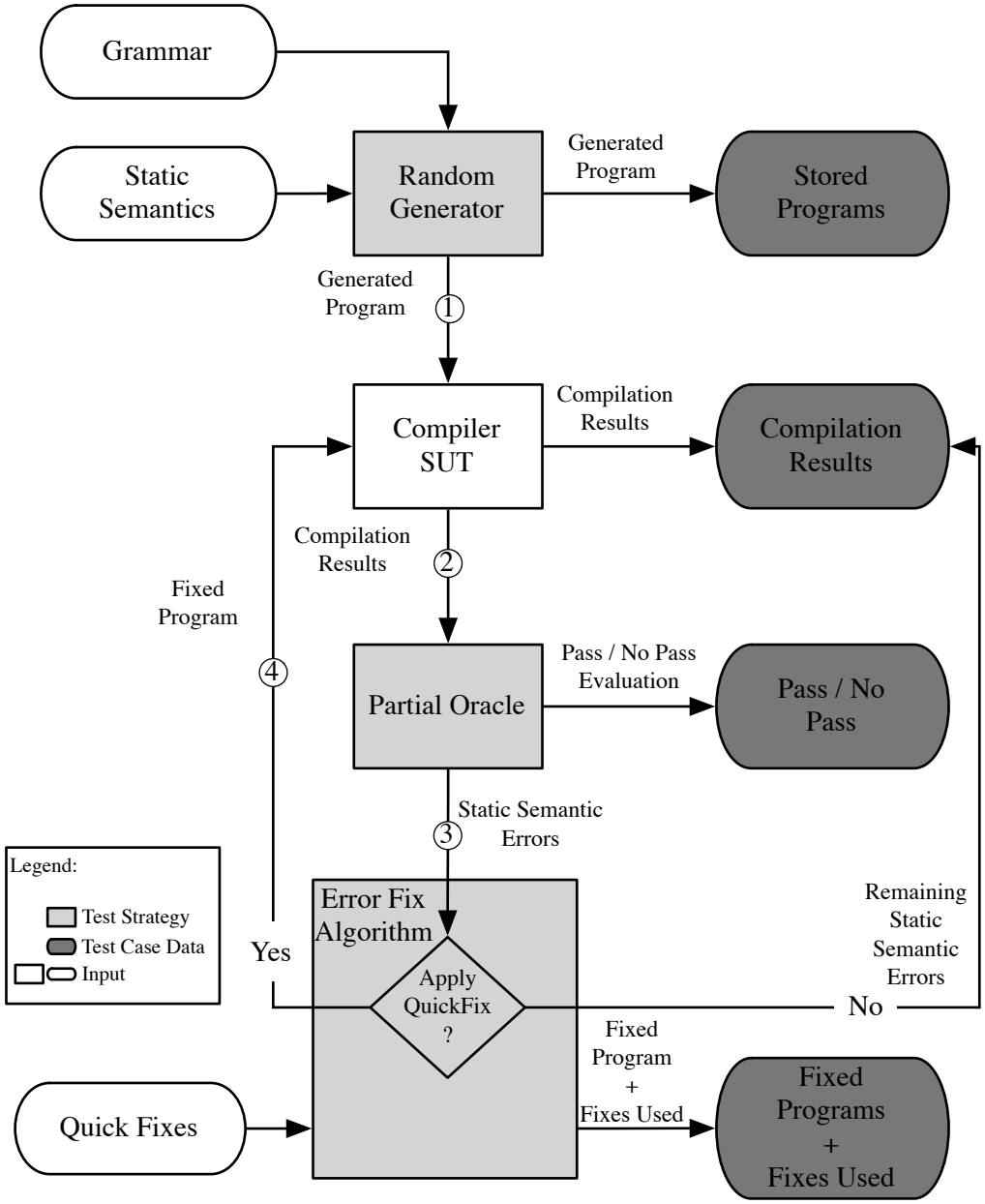


Figure 3.11: Expansion of our test strategy to enable static semantically correct generation. The random generator now generates not only styntactically correct programs but (partially) statically syntactical correct programs and invokes the compiler. Step ① invokes the compiler with this generated program and step ② passes it on to the partial oracle. The partial oracle now does not only store static semantic errors but if present sends them to the Error Fix Algorithm in step ③. This algorithm attempts to fix these static semantic errors and invokes the compiler again in step ④ and the steps ②③④ are repeated exhaustively.

3.2 Generating Static Semantically Correct Programs

Generation of syntactically correct programs ensures the execution of the analysis phase. However, the generation phase of the compiler, also hand written, is only executed when the analyzing phase succeeds and finds no static semantic errors. In other words this phase is only executed when static semantically correct programs are compiled.

This led us to expand our test strategy to enable the generation of (partial) static semantically correct programs programs, which should result in better test coverage. Furthermore, if fully static semantic correctness is guaranteed the partial oracle can be expanded recognizing that statically correct programs should lead to program generation, whereas incorrect programs should lead to static semantic errors.

To achieve static semantic correctness we use two approaches, the first is an expansion of the previous algorithm for syntactically correct generation and the second is applying quick fix suggestions post compilation to mend remaining static semantic errors. Figure 3.11 shows our expanded test strategy to include static semantic correct generation techniques.

In this Section we first present a running example, which builds upon the running example from Section 3.1, then we explain the two approaches mentioned above.

3.2.1 Running Example

To better explain our approach towards static semantic correct generation we expand our earlier running example of Section 3.1 in two ways. First we present a small function definition grammar that uses the simple arithmetic language of the previous example. The expanded language's grammar now includes the terminals: `Type` and `String`; the nonterminals: `FunDef`, `Param` and `Stat`; and the productions in Figure 3.12. Together with the corresponding name binding declarative model, shown in Figure 3.13, this is used to illustrate name binding.

Secondly, we show a set of transformation rules for type-checking and expression evaluation. These are used to illustrate how the name binding language can help induce certain compiler executions and how the error fixing algorithm could be used to generate more static semantic correct programs.

During this running example we will use constructors and ASTs instead of the earlier PT to describe a syntax tree. Moreover, we will show a table representation of scope and name information gathered and used during program generation. Scopes are kept for each namespace and are composed by an URI describing the scoping hierarchy visible for the current node. Examples of such ASTs and corresponding name tables can be found in Figures 3.14 and 3.15.

Figure 3.16 shows transformation rules for type checking and evaluation of the addition rule of the new language.

3.2.2 Random Generation Using Static Semantic Definitions

The first approach is inspired by Hanford's Syntax Machine [11] which uses dynamic grammars during generation. These grammars are dynamic such that they can be shrunk and

```
context-free syntax
 7: Id "(" { Param "," } ")" ":" Type "{" Stat* "}" -> FunDef { "FunDef" }
 8: Type ":" Id         -> Param { "Param" }
 9: Type ":" Id ";" -> Stat { "VarDef" }
10: Id ":" Exp ";"      -> Stat { "VarAssign" }
11: "return" Exp ";"-> Stat { "Return" }
12:  Id         -> Exp { "Var" }
13: String -> Exp { "String" }
lexical syntax
14: "\"" ~[\"\n\\]* "\"" -> String
15: "int"       -> Type
16: "string" -> Type
```

Figure 3.12: Grammar of the language used as the running example for this section, which uses the grammar of the running example of Section 3.1.

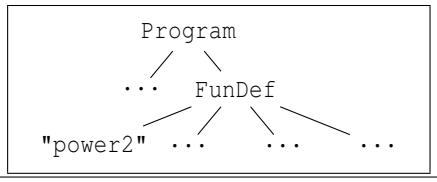
```
FunDef(f, p*, t, _) :   defines Function f of type (t*, t)
                                where p* has type t*
                        scopes Variable

VarDef(t,x)      : defines unique Variable x of type t
Param(t,x)       : defines unique Variable x of type t
Var(x)           : refers to Variable x
VarAssign(x,_)   : refers to Variable x
```

Figure 3.13: Name binding declarations for the language used in this running example. Note that these declarations refer to productions by their constructor, to facilitate notation.

expanded to limit the choice of productions during AST generation such that these obey the language's static semantics. Our approach distinguishes itself from Hanford's whenever the filtering of productions yields an empty list. Where Hanford would backtrack to the previous production, treating the current one as unfeasible, we use an injection procedure. This procedure assumes it is possible to inject a sub-tree that triggers the expansion of the grammar such that productions become available to finish the current sub-tree. This is needed to preserve the original non-backtracking behavior of our generation algorithm, allowing us to detect malformed grammars.

We distinguish between name binding and type semantics and to obey to either the algorithm requires declarative definitions of these. Since during the development of the implementation we only had access to declarative definitions of name binding semantics we will treat these in more detail than type semantics.

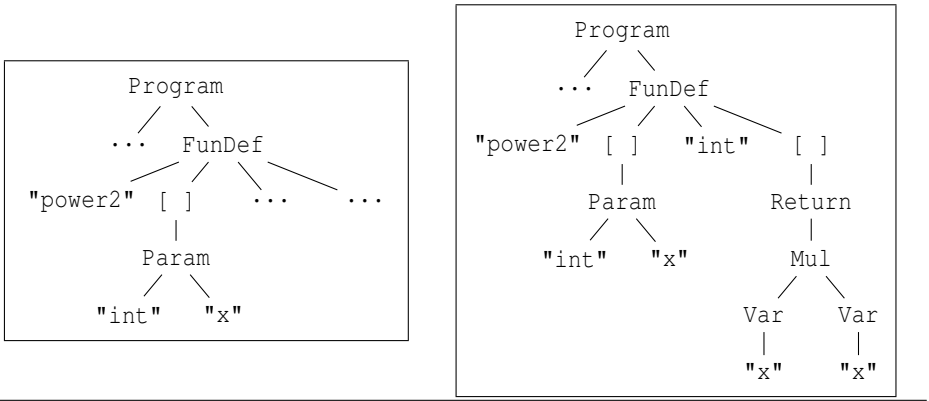
Using the same terminology as Konat et al. [22] we assume declarations of name binding to have the following or equivalent format: `Pattern ":" BindingClause*`. Where `Pattern` represents a production or its corresponding tree node with optional bindings to distinguish identifiers and types and `BindingClause` describes a name binding property of the production, which may be: scoping, referring, defining or importing. Since our test



Scopes
`Function -> Program("test")`
`Variable -> Program("test")/FunDef("power2")`

Values		
NS	Scope	Value
`Function`	`Program("test")`	`"power2"`

Figure 3.14: Intermediate tree (top) resulting of a program generation where a `FunDef` is being generated, thus far only the first child has been constructed and the generation algorithm chose `"power2"` as the function's name. We also show the list of scopes and values (bottom) corresponding to the generation.



Scopes
`Function -> Program("test")`
`Variable -> Program("test")/FunDef("power2")`

Values		
NS	Scope	Value
`Function`	`Program("test")`	`"power2"`
`Variable`	`Program("test")/FunDef("power2")`	`"x"`

Figure 3.15: Intermediate tree (top left) following the generation of the tree in Figure 3.14, where the next element of `FunDef` is generated and the algorithm chose to generate a `Param` with type `"int"` and name `"x"`. The next tree (top right) is later in the generation where the algorithm has chosen to use an existing name for the namespace `Variable` in the current scope in both two branches of the `Mul` production and the choice falls to the only name `"x"`. We also show the updated list of scopes and values (bottom) corresponding to the generation.

```
type-check=
  ?|[ ~x + ~y ]|
; <eq> (<type-of> x , Int())
; <eq> (<type-of> y, String())
; <report-error> ( y, "Incompatible type, expected int")

type-check=
  ?|[ ~x + ~y ]|
; <eq> (<type-of> x , String())
; <eq> (<type-of> y, Int())
; <report-error> ( y, "Incompatible type, expected string")
```

```
eval:
 |[ ~x + ~y ]| →  Int(z)
 where
   <eq> (<type-of> x, Int())
 ; <eq> (<type-of> y, Int())
 with
   x_e := <eval> x
 ; y_e := <eval> y
 ; z := <add> (x_e,y_e)

eval:
  Num(x)  →  x
```

```
eval:
 |[ ~x + ~y ]|  →  String(z)
 where
   <eq> (<type-of> x, String())
 ; <eq> (<type-of> y, String())
 with
   x_e := <eval>  x
 ; y_e := <eval> y
 ; z := <conc-strings> (x_e,y_e)

eval:
  String(x)  →  x
```

Figure 3.16: Type checking rules for the expression language (top). Evaluation rules for the addition or concatenation operator in the expression language (bottom).

```
ErrorFixRule1:
  "Incompatible type, expected int", $_1 → change to Int(1)

ErrorFixRule2:
  "Incompatible type, expected string", $_1 → change to String("something")
```

```
1+"a"
> error: (String("a"), "Incompatible type, expected int")
-- Apply ErrorFixRule1 --
1+1
>2
```

Figure 3.17: Error fix rules (top) to deal with the errors defined in the `type-check` strategy of Figure 3.16. Example expressions and output (bottom) when `type-check` and `eval` are applied after another.

strategy currently does not offer support for multiple file generation we will not treat the case of name importing.

Scopes restrict the visibility of definitions such that referrals may only refer to definitions within the same scope or outer scopes, never inner scopes. To keep track of these incremental scopes we propagate an URI per namespace. Every time a new scope is created for a namespace the URI is appended with this new scope and passed on accordingly. If the scope is horizontal the new scope is propagated to the production's siblings whereas a vertical scope is propagated to the production's children. Note that this requires a change in the generation algorithm in Figure 3.10 such that the loops in lines 11 and 17 have a threaded behavior, passing on new scopes where needed to their siblings.

Productions with defining properties are responsible for introducing names into the corresponding namespace's scope. The binding clause `defines NS id` will trigger the algorithm to add a randomly chosen terminal `id` to a dictionary with a key composed by the namespace and a scope belonging to it. The scope used here is nothing more than the tail element of the URI for the current scope of `NS`. Note that a defining clause may have optional properties that will change this behavior. A name may be required to be unique, triggering the generation algorithm to randomly choose a value from the terminal pool that does not occur in the current scope of the namespace or generate a new terminal. Also, the optional `in subsequent scope` may be used to restrict names from being used before their definitions. This triggers the algorithm to create an horizontal scope, which is passed on to the right siblings of the production and the name is added to this scope.

The referral property is represented by the binding clause `refers to NS id`. This triggers the algorithm to look up all values for visible scopes of `NS` and randomly choose one to replace `id`. The visible scopes here are by default all values in all the elements of the URI for the current scope of `NS`. Again here optional attributes may be added to specify the scope which the production refers to. If no name has been defined for the namespace `NS` in the visible scopes then the algorithm uses the injection mechanism. A definition is thus created, added to a randomly chosen injectable scope and a definition term is queued for injection.

The injection algorithm developed uses the randomly chosen scope of the namespace we wish to inject a definition into to find alternatives for injection. First it finds the AST node responsible for the chosen scope and then using that node and namespace as input it calls the function *getAlternative* in Figure 3.18. This function randomly chooses an injection alternative by traversing the sub-tree of the scoping node in a top-down manner. At every node that corresponds to a list, that is a node with a parent sort which has a cardinality higher than 1, it gets all *definition paths* starting with the same sort and belonging to the namespace at hand. A definition path is a generation path from a sort to a term defining the given namespace, such that the generation never goes through a node that scopes the same namespace. These paths are pre-computed using the algorithm in Figure 3.19.

Example 3.2.1. Using the grammar and name binding declarations in Figure 3.12 lets assume the algorithm during generation chooses to generate a `FunDef` with the randomly chosen name `"power2"`. According to the name binding declarations this will trigger the name binding expansion to add a new scope for the namespace `Variable`. Assuming the ear-

Input: *node* : the current node, starts with the node scoping the scope we will insert into.
namespace : the namespace we are inserting for.
path : the path we need to follow to get to the node we will insert at.
Output: A randomly chosen alternative for insertion containing a *path* to lead the insertion algorithm from the scoping node to the list to insert into and the definition construction term.

```
1:  function getAlternatives(node,namespace,path)
2:      if isList(node) & length(node) < U(node) then
3:          DefPaths := DefinitionPaths(getSort(node),namespace);
4:          for i := 1...|DefPaths| do
5:              Alts(i) := (path·i,DefPaths(i));
6:          end for
7:      end
8:      N := children(node);
9:      for i := 1...|N| do
10:         if not(N(i) scopes namespace) then
11:             Alts := Alts ∪ getAlternatives(N(i),namespace,path·i);
12:         end
13:     end for
14:     return randomlyChoose(Alts);
15: end function
```

Figure 3.18: The *getAlternative* algorithm, used by the injection algorithm to randomly find an alternative for injection. Note that we only inject definitions or trees containing definitions into lists in the tree. The global *DefinitionPaths* is constructed by the algorithm in Figure 3.19.

lier scope was the root it self, the existing scope `Program("test")` will be appended with `FunDef("power2")` thus yielding the new scope `Program("test")/FunDef("power2")`. The corresponding intermediate tree, scope and values lists are shown in Figure 3.14.

Example 3.2.2. Continuing with the previous example the generation algorithm chooses to generate a single `Param` for the second child of `FunDef`. The generated `Param` has the randomly chosen name `x` and type `"int"`, yielding the intermediate tree shown in the top left of Figure 3.15.This generation triggers the algorithm to add a new value to the namespace `Variable` in its current scope, which can later be used when the generation chooses to use an existing name for `Var` in both branches of the `Mul` production, yielding the tree shown in the top right of Figure 3.15.

The approach for type semantics is similar and the component has two responsibilities: keeping track of known static types for names and productions and enforcing static type restrictions. The algorithm is altered to log type information after generation of a type inferring production. Also to enforce correct static typing, each branch of generation is

```
1: function preComputeDefinitionPaths
2:     AllDefinitions := getAllDefinitions();
3:     for ∀def ∈ AllDefinitions do
4:         DefPaths := DefPaths · ([getSort(def)], def);
5:     end for
6:     for ∀(path, def) ∈ DefPaths do
7:         getAndSaveDefinitionPath(path, def);
8:     end for
9: end function
```

Input: *path* : the path from the defining term to the parent sort for which it will be saved.
def : the defining term as declared in the name binding definition of the language.

Output: A globally accessible dictionary containing key-value pairs where $DefinitionPath(s, ns)$ contains all the paths from sort s to the defining terms of namespace ns.

```
1: function getAndSaveDefinitionPath(path,def)
2:     ns := getNamespace(def);
3:     DefinitionPath(path_0, ns) := DefinitionPath(path_0, ns) cot(path, def);
4:     parents := getParentSorts(path_0);
5:     PSorts := PSorts ∪ {∀p ∈ parents | p ∉ path & not(p scopes ns)};
6:     for pSort ∈ PSorts do
7:         getAndSaveDefinitionPath(pSort · path, def);
8:     end for
9:     return DefinitionPath;
10: end function
```

Figure 3.19: The *preComputeDefPaths* algorithm used in the initialisation of the generator to pre compute all the paths from defining terms to any parent sort for every namespace, assuming the path does not cross a term scoping the same namespace. These paths and corresponding defining term are storedin the globally accessible *DefinitionPaths*. Note that the *getParentSorts* takes priority and associativity properties into consideration such that no paths are created that are syntactically invalid.

provided with a list of type restrictions that is passed on accordingly as an extra parameter of the *generatePT* function. For instance, a language might have type semantics that only allow integers to be added to integers and floats to floats. Whenever an addition production is chosen, the generation of the left branch will restrict the generation of the right branch.

3.2.3 Error Fixing Algorithm

Having covered name binding and type semantics, the second algorithm was implemented to further explore the set of (partially) static semantically correct programs. This algorithm exploits the existence of *Quick Fixes* to attempt to fix any static semantic errors reported by the compilation procedure of the generated program. Quick fix suggestions are an often offered feature by IDE's, like in the case of the Eclipse Java plugin [1].

In contrast to the earlier shown setup the partial oracle now forwards the static semantic error reports to the *Error Fix Algorithm* in step ③. This algorithm uses any existing quick fix suggestions to fix the reported errors. These suggestions are expected to be described in two parts, one which should match the error it is supposed to fix and the other the transformation that fixes it. One at a time, the algorithm tries to match the actual set of errors to the matching part of the quick fix and if a match is found it attempts to apply the corresponding transformation. If this transformation succeeds, yielding a syntactically correct program, the algorithm invokes the compiler with the fixed program in step ④. Steps ②③④ are repeated exhaustively during which any static semantic errors, fixes used and fixed programs are stored in the test case.

Example 3.2.3. Take the language described in the running example with a compiler containing the rules shown in Figure 3.16. If the concrete syntax shown in the first line of the bottom part of Figure 3.17 was to be compiled, the results would be the error shown on the second line. This would prevent the execution of the evaluation rules. If the error fixes described in the top part of Figure 3.17 were provided, the algorithm would match the error to the first rule and apply it, resulting in the concrete syntax `1+1`, yielding the value `2` after evaluation. Thus reaching a compiler transformation rule that would not be reached if the error persisted.

3.3 Partial Oracle

Having generated input for testing, the test strategy invokes the compiler with the generated program. We choose to use the concrete syntax to invoke the compiler with, going through the parsing phase instead of bypassing this phase and using the generated PT for the invocation of the analyzer. This guarantees the generated program is indeed syntactically correct and that the parser accepts it. This way, we guarantee that any failure encountered in the analysis phase is not caused by an syntactically incorrect PT.

After invocation the strategy sends the compilation results to the test oracle for a pass/no pass evaluation. The challenge of this evaluation lies in the fact that an automatic oracle is pragmatically unattainable. Traditionally, automated compiler testing solves this problem

[1] Eclipse FAQ: What is a QuickFix? `http://wiki.eclipse.org/FAQ_What_is_a_Quick_Fix%3F`

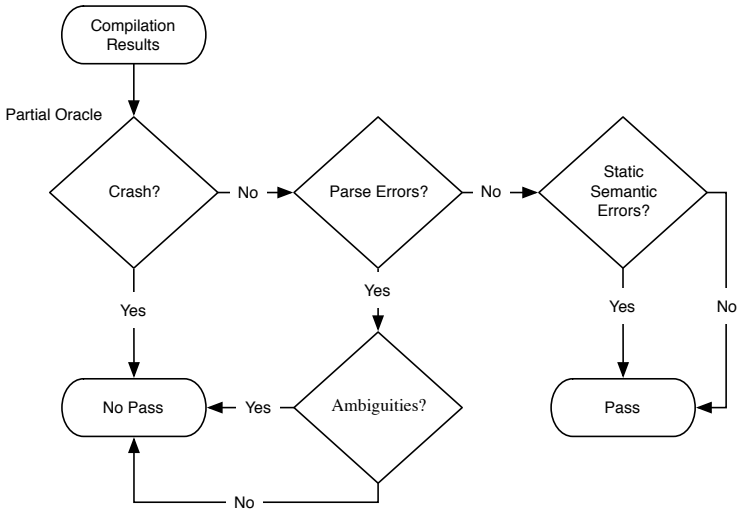


Figure 3.20: The partial oracle and how it disntiguishes between the various recognized compilation results and its pass / no pass evaluation.

through differential testing comparing the execution of two compilers for the same language with the same input. This approach can not be used for most DSLs, since they have only one compiler. Instead we use a partial oracle, this is an oracle that is able to state with assurance that a result is incorrect without actually knowing the correct answer [39].

The partial oracle used distinguishes between five scenarios. The first two detectable scenarios are when the execution of the compiler does not get through the parsing phase. Since we guarantee syntactically correct program generation any parse error must be due to a malformed or incomplete grammar. We distinguish two types of failures during parsing: ambiguities and other parse errors. Ambiguities are caused by missing disambiguation rules, whereas parse errors can be caused by malformations of the grammar or discrepancies between the grammar and the pretty-print table.

The next two detectable scenarios are detected when the compiler execution halts at the analysis phase and are the reporting of static semantic errors or compiler crashes. The first is treated as a pass, whereas the second is naturally treated as a no pass.

The last of the detectable scenarios occurs when the generated program is static semantically correct and passes the analysis phase, thus leading to the generation of a program in the target language and it is treated as a pass. It is however very unlikely that the random syntactically correct generator will yield such programs, however we hope the expansions proposed in Section 3.2 pave the way towards enabling such static semantic correct generations.

Example 3.3.1. Assume the language we are testing is the same as in the running example in Subsection 3.1.1. However, the language developers failed to add the left-associative property to the addition operation. Upon generation of a program with either PTs in Fig-

ure 3.4, the parser would interrupt compilation and report an ambiguity. Our partial oracle would then classify this as a failure and save it as such.

Example 3.3.2. Take the transformation rules described in the running example in Subsection 3.2.1. If the language developers had forgotten to add the conditional where clauses that distinguish between the two functions of the overloaded + operator, the execution of the `eval` rule would depend on which of the two rules it would try to match first, potentially causing a crash for either a concatenation of strings or an addition of strings. Such a crash would be detected by our partial oracle, which would classify it as a failure, save the concrete syntax causing it and the stack-trace.

3.4 Shrinking Heuristic

The generation of big programs may lead to an higher likelihood of finding failure inducing programs but it makes the generated programs harder to understand. This complicates the task of language engineers to trace the failure back to its causing fault. To facilitate this task we offer a program shrink heuristic that attempts to reduce the program's size whilst preserving its failure inducing behavior. Note however that we do not concern ourselves with efficiency.

The heuristic is based on the hierarchical delta debugging algorithm [32] and is shown in Figure 3.21. It begins by parsing the input program and saves the original behavior produced by the compiler when invoked with the input program. Then it traverses the AST in a top-to-bottom manner applying transformations to reduce the size of the sub-tree under each node it passes through. After each transformation the compiler is invoked and the compiler behavior compared to the original behavior. If the transformed program does not yield an equivalent behavior the transformation is undone. We identify the behavior of a failing compiler by the exception and the corresponding stack trace. Due to the top-to-botton approach whenever a node is substituted by smaller sub-tree, all of its m children are no longer treated for shrinking, thus saving m compiler runs. Moreover, the runtime of the heuristic is only N compiler runs, where N is the number of nodes in the original AST, if no shrinking ever takes place.

We propose two transformations for size reduction. The first transformation is used in case the current node is a list and thus corresponds to an iteration in the grammar. The transformation is shown in Figure 3.22 and it attempts to reduce the list's size to its minimum cardinality. For lists with a minimum cardinality of zero the transformation first attempts to remove all items. If this fails or for any other minimum cardinality the transformation removes one item at a time rerunning the compiler and comparing behaviors in between. The second transformation attempts to replace each sub-tree with a smaller, still syntactically correct, tree. It does so by replacing these sub-tree's by terminating paths whenever the depth of such a terminating path is lower than the depth of the sub-tree in question. A terminating path is a minimal depth syntactically correct tree with a non-terminal root and these are computed for every non-terminal in the grammar. The algorithm to compute these is explained in Section 3.1.

Input: *program* : the program to shrink.
P : the set of all productions.
ppTable : pretty-print table.
Output: Returns a program with the same failure inducing behavior as the input program, however it is at least smaller than the input.

```
1:  function shrinkProgram(program,P,ppTable)
2:      AST := parse(program);
3:      originalBehavior := runCompiler(AST);
4:      for ∀node ∈ AST do
5:          if L(AST_node) ≠ U(AST_node) then
6:              AST_node := reduceList(AST, node, originalBehavior, P) ;
7:          else if depthOfTree(AST_node) > DT(AST_node) then
8:              old := AST_node;
9:              AST_node := TP(AST_node) ;
10:             if runCompiler(AST) ≢ originalBehavior then
11:                 AST_node := old;
12:             end
13:         end
14:     end for
15:     return prettyPrint(AST, ppTable);
16: end function
```

Figure 3.21: Algorithm for shrinking an AST whilst preserving its failure inducing behavior. Two transformations are used to reduce the size of the AST in a top-to-bottom approach. If the current node is a list, the algorithm attempts to reduce the size of the list. Whereas if the node is a non-terminal it attempts to replace it with a smaller sub-tree.

```
Data: AST : a tree representation of the program
      originalBehavior : a representation of the original compiler behavior
      node : the list to reduce.
      P : set of all productions, used to determine cardinality.
 1: function reduceList(AST, node, originalBehavior, P)
 2:     if L(AST_node) = 0 then
 3:         AST_node := [ ];
 4:         if runCompiler(AST) ≡ originalBehavior then
 5:             return AST;
 6:         end
 7:         i := 1;
 8:         while length(AST_node) > i & length(AST_node) > L(AST_node) do
 9:             old := AST_node;
10:             AST_node := removeIthItemFromList(AST_node, i);
11:             if runCompiler(AST) ≢ originalBehavior then
12:                 AST_node := old;
13:             else
14:                 i++;
15:             end
16:     return AST;
17: end function
```

Figure 3.22: Algorithm for reducing list size whilst preserving behavior. If the list's minimum cardinality is zero the transformation to an empty list is first attempted, otherwise items are removed once at a time. $length(s)$ returns the length of a list, $removeIthItemFromList(list, i)$ removes the ith item from the $list$ and $L(symbol)$ returns the minimum cardinality for $symbol$.

Chapter 4

Testing Spoofax Developed Compilers

In the previous chapter we presented a test strategy to generically and automatically generate test cases using language artifacts for DSL compilers. These test cases are created using a random generator that yields syntactically correct programs, with the optional ability to adhere to static semantics if declarative descriptions of these are available. A partial oracle is then used to categorize the behavior of the compiler, where program generation or static semantic error reporting is categorized as a pass and crashes or parse errors, including ambiguities, are categorized as a no pass. After execution of the compiler any existing quick fixes are used to fix potentially reported static semantic errors thus yielding a new program and creating a slightly different test case. Post test case generation a shrink heuristic is available to shrink failure inducing programs thus making them potentially easier to comprehend.

To put our test strategy to the practice and evaluate it we implemented this test strategy specifically targeting compilers developed with Spoofax. In this chapter we first describe the implementation specifics in Section 4.1, followed by a description of the coverage metrics we measured and the system developed to measure Stratego code coverage in Section 4.2.

4.1 Implementation Specifics

We implemented the test strategy described in the previous chapter for generic automatic compiler testing specifically targeting DSL compilers developed with the language workbench Spoofax. This restricts the compilers it can accept, however, it is still generic to the point that it can test any compiler implemented with Spoofax. Furthermore, this restriction provides us with a common ground between compilers enabling us to focus on one specific sort of grammar formalism, SDF, a common way of invoking compilers, a common paradigm for error reporting and a declarative name binding language, NaBL.

The test strategy was implemented using Spoofax's transformation language Stratego, enabling the use of the native parsers to parse the language's grammar and name binding declarations, the use of the same native building blocks used by compilers to build ASTs called ATerms, the native pretty-print transformation and the tree traversal and transforma-

tion paradigms which proved to be useful during AST generation. Note that we do not generate PTs since AST generation is easier to implement and traditionally pretty-print-tables are available for Spoofax developed languages. However, we do not assume the existence of a correct parenthesize strategy and generate tree's whilst respecting associativity and priority properties. Also any parenthesizing productions are given a specially created constructor that is later pretty-printed accordingly, including the parenthesizes. Our implementation of the test strategy requires a configuration file which regulates the strategy, its control mechanisms and defines the error fixes, Figure 4.1 shows an example of such a configuration file. It has two main sections, the `automatic test` section containing the main testing inputs and the `error correction` section containing all Error Fixes explained in Subsection 4.1.3. In the `automatic test` section essential settings such as the target language, target strategy, location of jar to invoke, the input required to invoke the jar, optional cache folders that need to be removed and number of generated test cases are set. Control mechanisms can also be adjusted, such as maximum term count per test, maximum recursion depth, maximum iterations per single operator, number of generated terminals per terminal sort and terms to ignore during generation. Note that we also implemented a minimum iteration control, this is to help us force creation of larger programs. The next three subsections describe the specifics involved in the implementation of the random generator, the partial oracle and the error fixing algorithm.

4.1.1 Random Generator

The random generator in our test strategy takes the control mechanism parameters defined in the configuration file and the language's SDF module as main input for program generation. SDF defines a grammar through productions for lexical and context-free syntax and context-free priorities. The lexical productions are used for the terminal generation procedure of the *generateProgram* function in Figure 3.6. The parameter **`terminal pool size`** determines how many terminals are generated per lexical sort. To generate these in a random manner we use a third party Java random generator for regular expressions called Xeger [1]. The lexical productions are translated to regular expressions accepted by Xeger with the restriction that any character generated is in the character set accepted by SDF.

The random generator then proceeds to choose a start-symbol and starts AST generation using the context-free productions, associativities and priorities to yield a syntactically correct tree. We construct these trees using ATerms, Stratego's data format for the structured representation of programs. These can either be constructors representing a context-free production, a string representing a terminal or a list representing a non-single cardinality context-free sort. Note that any productions in SDF marked with a rejected or deprecated property are ignored by our generator and are not included in the generation. Furthermore, our generator supports the use of the `ast` keyword in SDF which is used to manually define constructors for productions, these are parsed and used as normal constructors throughout generation.

[1] Xeger - A Java library for generating random text from regular expressions. `http://code.google.com/p/xeger/`

```
automatic test
 target language: WebDSL
 file extension: app
 target strategy: "webdslc-main"
 invoked jar:
  "file:/home/andre/Documents/Stratego/bin/WebDSL/bin/webdsl.jar"
 invocation input:
  ["test", "-i", @PROGRAMPATH , "--verbose" , "2", "--servlet"]
 cache folders: ".webdsl-parsecache"
 test tag: Test
 nr# runs: 500
 terminal pool size: 50
 max term count: 5000
 max recursion: 30
 max iteration: 50
 min iteration: 0
 test folder: "TestFolder"
 ignore sorts:
  "ExternalScopeVar", "GenericSort", "Module", "Imports"

error correction // for WebDSL

 noRoot:
  "no root page root() defined.", _  →
       insert once DefinePage([],"root",[], None() ,[])
        @ ApplicationDefs(_,$here,_)
    or insert once Section("rootFixSection",
                          [DefinePage([],"root",[], None() ,[])])
                @ Application(_,$here)
```

Figure 4.1: Configuration file for the automated test strategy. `@PROGRAMPATH` is an internal keyword representing the location of the generated program.

The random generator has an extra option to accept NaBL modules to adhere to name binding semantics. NaBL describes name binding by assigning constructors of the language name binding properties, our current prototype accepts three: scoping, defining and referral. This prototype for name binding correct generation also supports the optional unique/non-unique property of names, however no support is offered for horizontal scoping, scope importing or import of scopes and values from other files. Note also that the WebDSL NaBL module used is an old outdated and incomplete version, since during the development of the strategy NaBL was also in its early phases of development.

4.1.2 Partial Oracle

After generating the AST, the random generator uses the native pretty-print transformation to transform the AST into concrete syntax. The first partial oracle step is to parse this concrete syntax using the native parser and the language's parse-table. This step may lead

to three possible outcomes. The first is the reporting of ambiguities, these are caused by missing disambiguation rules. Spoofax' parsers return ambiguities as a list of possible ATerm interpretations of the ambiguous concrete syntax. We store each ambiguity and its corresponding concrete syntax in the test case and log it as a no pass. The second possible outcome is the report of other parse errors, which are caused by inconsistencies between the language's grammar and pretty-print table. Our partial oracle saves both the concrete syntax, original AST and parse error message and logs the test case as a no pass. The last possible outcome is the transformation of the concrete syntax into a syntactically correct AST, green lighting the next step, the invocation of the compiler with the generated program.

To invoke of the compiler with the generated program we developed a custom Java library using Spoofax' Java API. The generated concrete syntax is printed to a file and this location is passed on to the compiler in an ATerm as described by the configuration of the input term. The compiler then either returns an ATerm or a Java Exception. The ATerm may contain either static semantic errors or not, the last indicates the program cleared the analysis phase and generation took place. Either of which are classified as a pass by the partial oracle that stores the compiler console output, the returned ATerm and input program as the test case. Java Exceptions indicate the compiler crashed, this is obviously treated as a no pass and the exception, stack trace, console output and input program are stored as the test case.

4.1.3 Error Fixing Algorithm

Spoofax currently does not offer support for development of a QuickFix suggestion feature. This led us to develop a prototype for an Error Fix DSL to provide testers with the possibility to define quick fixes. The goal of this language is to test our approach in using these to achieve better coverage or find new failures and not to provide Quick Fixes for Spoofax.

The language is composed of error fixing rules, which in turn have an error matching side, to the left of the right arrow($\rightarrow$) and an error correction side to its right. The error matching part is again composed of a message matching pattern and a term matching pattern. The actual error reported by the compiler is also composed by two parts: a message and an ATerm. The message returned to the compiler must contain the words in the defined message pattern and the ATerm has to succeed a Stratego pattern match defined by the term matching pattern in the rule. We also offer the functionality to bind variables in the term matching pattern to use later in the error correcting action.

For this prototype only a set of error correcting actions have been implemented, see figure 4.2 for the implemented grammar of the error correcting actions. The `AddCorrection` actions are designed to insert new ATerms into the AST where the first inserts it at the end of the list and the second at the beginning of the list. The `DeleteFromListCorrection` action is designed to remove an ATerm from any list in the AST. The `ChangeCorrection` action is designed to swap an ATerm for another.

```
context-free syntax

  InsertOperation Fix "@" DestMatch
    -> Action{"AddCorrection"}
  InsertOperation Fix "first@" DestMatch
    -> Action{"AddCorrectionFirst"}
  DeleteOperation { ErrorMatch "or"}+ "from" "list"
    -> Action{"DeleteFromListCorrection"}

  Fix "change" "to" Fix  -> Action{"ChangeCorrection"}
 "change" "all" "to" Fix -> Action{"ChangeAllCorrection"}

  "insert"        -> InsertOperation{"Insert"}
  "insert" "once" -> InsertOperation{"InsertOnce"}
  "insert" "once" "for" "(" ErrorMatch ")"
    -> InsertOperation{"InsertOnceFor"}

  "delete" -> DeleteOperation{"Delete"}
  "delete" "for" "(" ErrorMatch ")"
    -> DeleteOperation{"DeleteFor"}

  "$concat" "(" Fix "," Fix ")" -> Fix{"Fix_Conc"}
  ID "(" {Fix  ","}* ")"        -> Fix{"Fix_Cons"}
  ID                            -> Fix
  STRING                        -> Fix{"Fix_String"}
  "[" "]"                       -> Fix{"Fix_EmptyArray"}
  "[" Fix+ "]"                  -> Fix{"Fix_Array"}
  "$_"INT                       -> Fix{"Fix_Var"}
  INT                           -> Fix{"Fix_Int"}

  ID "(" { ErrorMatch ","}* ")" -> ErrorMatch{"ErrMatchCons"}
  "$_"INT                       -> ErrorMatch{"ErrMatchParamBind"}
  "@_"INT                       -> ErrorMatch{"ErrMatchParam"}
  STRING                        -> ErrorMatch{"ErrMatchString"}
  INT                           -> ErrorMatch{"ErrMatchINT"}
  "_"                           -> ErrorMatch{"ErrMatchWildCard"}

  ErrorMatch -> DestMatch
  "$here"    -> DestMatch{"TargetNode"}
```

Figure 4.2: Grammar for Error Correcting Actions.

4.2 Coverage Measuring

Coverage is the measurement used during software testing to describe to what extent the software has been tested. We will measure two types of coverage to compare our generated test cases to the existing test suite. These two types are code and grammar coverage and they were chosen since they are easy enough to measure and cover the different states of the

```
some_strategy =
    __coverage_point_entry_some_strategy_01__
    statement1
;   statement2
;   ...
    __coverage_point_exit_some_strategy_01__

some_rule :
  patternToMatch  →  patternToWrite
  __coverage_point_entry_some_rule_01__
  where
    statement1
  ; statement2
  ; ...
  __coverage_point_exit_some_rule_01__

some_rule :
  patternToMatch  →  patternToWrite
  __coverage_point_no_condition_some_rule_02__
```

Figure 4.3: Points of instrumentation of Stratego code with Coverage Points.

compiler and input domain respectively.

4.2.1 Code Coverage

Code coverage measures which statements have been executed during execution for the given input. In this case it will log which statements have been executed when analyzing and transforming the program. Spoofax has no such native system thus we developed one to be able to measure code coverage. Our extended Stratego compiler can be given an option to enable coverage instrumentation during compilation. This instrumentation occurs after the parsing phase in the Stratego's compiler pipeline. All strategies and rules in the compiler being compiled are instrumented with *Coverage Points* to log both the *entering* and *exiting* of these rules and strategies.

A coverage point is represented by a special ATerm that is inserted at the beginning and end of a strategy and rule. For a strategy this means a point is inserted in between a strategy's call and first statement, denoting the entering of a strategy, and a point is inserted in between the last statement and return of the strategy, denoting the exiting. As for rules the points are inserted in between the pattern match and optional where/with clauses and in between the end of these clauses and rewriting. If the rule has no clauses these two points are the same and such a point is called a no condition point. See Figure 4.3 for an illustration of these points. Note that a rule or strategy are partially covered when only the entry point is passed and fully covered if the exit or no condition point is passed. All coverage points are identified by an unique identifier and the following information is stored for each identifier:

- **File Path**, the path of the file containing the strategy or rule being instrumented;

```
// code coverage key: type_of_rule_01
type-of:
 |[ ~e1 + ~e2 ]|  →  "int"
 where
   <type-of> e1 => "int"
 ; <type-of> e2 => "int"
// code coverage key: type_of_rule_02
type-of:
 |[ ~x ]|  →  "int"
 where
   <is-int> x
```

```
1+"not a number"
```

Covered points
Entry point for `type_of_rule_01`
Entry point for `type_of_rule_02`
Exit point for `type_of_rule_02`

Figure 4.4: An example of Stratego code coverage. On the left is the code being measured with the keys commented in for better understanding. The first rule is assgiend the `type_of_rule_01` key and the second `type_of_rule_02`. These rules are applied to the simple example program on the top right yielding the coverage of the points in the bottom right table. These indicate that the second rule is completely covered, since both entry and exit points are covered. Whilst the first is only partially covered, since the rule will fail on the second statement in the where clause, never reaching the exit point.

- **Strategy Name**, the name of the strategy or rule being instrumented;
- **Coverage Type**, a predefined constant which categorizes the coverage point as being one belonging to either a strategy or rule and whether its an entry, an exit or a no condition point;
- **Origin Location**, file positioning information indicating where the strategy or rule is defined in the file. Currently not supported due to the unavailability of a Java-based Stratego-to-Java compiler, this function is not supported for C-based compilers.

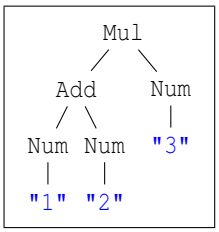
The inserted ATerms denoting the coverage points contain the unique identifier whereas the rest of the information is stored for later use.

During the generation phase of the Stratego compiler the special coverage point ATerms are transformed into Java calls to a method called `beenHere` which is given the unique identifier as a parameter. This method is included in our Java library dubbed `Coverage` and upon invocation increments a zero initialized counter for the corresponding identifier, thus logging which and how often each coverage point is passed. After generation all coverage points and their information are stored in files that are later used to construct code coverage reports. See Figure 4.4 for an example of code coverage.

4.2.2 Constructor Coverage

We measure grammar coverage with an abstraction we dubbed *Constructor Coverage*. This abstraction measures the coverage of SDF constructors, where each constructor with a specific number of children is treated as an unique element. Measuring this coverage for exist-

```
Exp "+" Exp      -> Exp {"Add"}
Exp "-" Exp      -> Exp {"Sub"}
Exp "*" Exp      -> Exp {"Mul"}
Exp "/" Exp      -> Exp {"Div"}
Num              -> Exp {"Num"}
[0-9]+  -> Num
```



Covered Keys
`Mul2`, `Add2`, `Num1`
Coverage
3/5

Figure 4.5: Example of constructor coverage. On the left a simple arithmetic grammar. In the center an AST of a program belonging to the simple arithmetic grammar. On the right the corresponding constructor coverage achieved by the AST in the middle.

ing test suite programs is easier than branch coverage of productions. For constructor coverage it suffices to look at the AST and analyze the constructors it is composed f, whereas production branch coverage would require logging during the parsing phase.

To use this constructor coverage first the grammar is parsed and keys are created for each constructor-number-of-children combination. We simply concatenate the constructor string and the number of children to create an unique key. The collection of these keys represents the set of all constructor coverage keys. During generation of a new or after parsing an existing test case, each constructor in the AST is transformed into its corresponding key and stored. This represents the coverage achieved by the given test case. If this set is equal to the set of all constructor coverage keys then full constructor coverage is achieved. Note that this is not the same as full branch coverage, since productions with no constructors are not measured by the constructor coverage and duplicate construction names with the same amount of children branches share the same key. See Figure 4.5 for an example of constructor coverage.

Chapter 5

Evaluation

We evaluate our test strategy by using it to generate test suites for WebDSL, a Spoofax built DSL. In this evaluation we answer the following questions:

- **EQ1:** What failures do our generated test cases unveil?
- **EQ2:** How frequently can our test strategy generate failure inducing programs?
- **EQ3:** How does the generated test suite cover the grammar and implementation of the language under test?
- **EQ4:** Do the expansions enabling partial static semantic generation improve failure finding likelihood and coverage?
- **EQ5:** Does our program shrink heuristic yield helpful programs?

Throughout this chapter we show our test strategy is able to find ambiguities and lead the WebDSL implementation to crash for valid input; that the generated test suites achieve better grammar coverage than the existing test suite but do significantly worse for code coverage; and that the program shrink heuristic proved to be a very useful tool in identifying the faults causing the induced crashes. We continue as follows: in Section 5.1 we present our research method, in which we explain the variants of the test strategy we evaluated, describe the test subject, elaborate on what data we collected and finally show how it was collected. We then proceed by presenting the results in Section 5.2 and interpret these in Section 5.3. To conclude this chapter we present an analysis of the threats to the validity of the performed evaluation in Section 5.4.

5.1 Research Method

In this section we present how we evaluated our test strategy to answer the questions posed above.

5.1.1 Test Strategy

To evaluate our test strategy we generated a test suite by applying the following variants:

Basic this variant uses the basic generation algorithm to guarantee syntactically correct generation using only the language's grammar as input,

Name this variant uses the name binding expansion to the basic version using both a NaBL module and the grammar as input,

ErrorFix this variant uses the error fixing algorithm post generation and requires both a set of error fixing rules and the grammar as input,

Name-ErrorFix this variant uses both the name binding expansion to the generation algorithm and the post generation error fix algorithm and requires the NaBL module, a set of error fixing rules and the grammar as input.

The differences between these variants allow us to analyze the different effects of the different expansions and aid us in better exploring the domain of possible inputs for compiler testing. Moreover, any variant using the name binding expansion will be used twice, once always adhering to the definitions in the NaBL module and once adhering to these definitions in only 75% of the times. This is done to partially cover the name binding rules, potentially exploring other input-output scenarios of the compiler.

Finally we execute the program shrinking heuristic for a collection of failing test cases to evaluate the shrinking power of our approach.

5.1.2 Subjects

To evaluate our approach we ran the mentioned variants of the test strategy to generate test suites for WebDSL [38]. WebDSL is a DSL targeting the domain of developing dynamic web applications with a rich data model. WebDSL consists of several sub-languages for data models, web pages, business logic, access control and work-flow. The WebDSL compiler ensures consistency across its sub-languages [9]. We generated test suites for two revisions of the WebDSL compiler, revisions r5579 and r5739 found in WebDSL's SVN repository [1].

WebDSL's compiler currently consists of approximately 25.000 lines of Stratego code and 15.000 lines of Java code and has been developed on since 2007 with stable Year-Over-Year commits. Figure 5.1 shows the number of sorts, productions and modules that compose the grammars for both revisions of WebDSL used during evaluation. WebDSL's development team is mainly composed of researchers in the Software Engineering Research group at Delft University of Technology. Masters students often contribute with smaller projects. The combined efforts of both reach an average of approximately 2.0 fte per year.

The NaBL module used for the name correct generation algorithm is based on an early version of NaBL which is a prototype and thus incomplete.

[1] WebDSL's SVN repository: `https://svn.strategoxt.org/repos/WebDSL`.

Revision	Non-terminal Sorts	Terminal Sorts	Context-free Productions	Modules
r5579	250	54	872	22
r5739	251	54	839	19

Figure 5.1: Grammar attributes for the two tested revisions of WebDSL.

5.1.3 Data collection

We apply the different variants of the test strategy to generate test suites for both revisions using a set of control mechanism settings. For each generated test suite we analyze how many and what failures were found, the achieved coverage and the average size of the generated programs.

Our partial oracle can identify three different types of failures: ambiguities, other parse errors and compiler crashes(see Section 3.3 and Figure 3.20). The first occurs when the parser can not distinguish between possible interpretations of the concrete syntax, thus yielding multiple possible ASTs. This is caused due to malformations in the grammar in the form of missing or erroneous disambiguation rules. The partial oracle stores both the concrete syntax and ASTs for such ambiguous programs. Other parse errors occur due to either erroneous generation or discrepancies between pretty-print table and syntax definition. The partial oracle stores the concrete syntax, parse error and generated AST for such programs. Failures originating from the parsing phase are counter-productive to finding failures originating from the analyzer and generator, since compilation never reach these components.

The last possible type of failure is a compiler crash. These occur due to unexpected terminations of the compilation process due to faulty compiler implementations. The partial oracle stores the concrete syntax, AST, error message, stack trace and console output for such programs. Since the probability of our generator yielding programs that reach the generation phase of the compiler is too low, all compiler crashes occur during the static analysis phase. We manually analyze each crash to determine in which stage of the analysis phase it takes place. WebDSL's static analysis phase has three stages:

- the global declare stage, where all global names are resolved and their declaration information saved,
- the local rename stage, where all local names are renamed uniquely and their declaration information saved,
- the type- and constraint-checking stage, where all type- and constraint-semantics are checked.

We measure how each generated test case covers both the input domain and compiler implementation, through constructor coverage and Stratego strategy coverage respectively.

Constructor coverage measures the number of unique constructors that compose the program used to execute the compiler with. Whereas the Stratego strategy coverage measures how many of the Stratego strategies(and rules) are invoked and completed during the execution of the compiler. For more information on these metrics and how they are measured see Section 4.2.

For comparison, we also measure the coverage achieved by the existing WebDSL test suite. This test suite was manually constructed by the WebDSL development team and is used to test each new revision. Within this test suite there is a set of test cases that test the detection of static semantic errors. These tests never reach the generation phase and we shall refer to this set as the WebDSL Fail Tests. To fairly compare the Stratego strategy coverage of the generated test suites to the manually constructed WebDSL test suite we decided to measure the results achieved by this set of failing tests in isolation. These tests never reach the generation phase and are thus a better benchmark for analysis phase coverage.

5.1.4 Analysis Procedure

To answer evaluation question EQ1 (types of failures), we analyze the failures found during generation and determine at which stage of the partial oracle they are detected. There are three types of failures our partial oracle can detect: ambiguities, parse errors and compiler crashes. Additionally, we examine the stack traces in the detected compiler crashing test cases and WebDSL code to determine the stage of analysis the crash-failure occurs in and to try and trace the fault causing the encountered failure.

To answer evaluation question EQ2 (occurrence of failures), we count the number of occurrences of each type of equivalent failure for the generated test suites.

To answer evaluation question EQ3 (coverage) we measure both constructor and Stratego strategy coverage for the generated test suites, for WebDSL's existing manually constructed test suite and for WebDSL Fail Tests, a selection of test cases from the manually constructed test suite that test the static semantic error reporting. This data is then compared and explained using insight and knowledge of the WebDSL language and compiler. Also, we analyze the coverage achieved by the generated test suites to determine whether new parts of the grammar or compiler implementation are covered by these and not by the manually constructed test suite.

To answer evaluation question EQ4 (expansions) we compare the failure finding rate and coverage achieved for the different variants of the test strategy to evaluate their added value.

To answer evaluation question EQ5 (shrinking power), we compare the size of some of the failure inducing generated programs to their corresponding shrunken versions. Furthermore, we use the feedback provided by the WebDSL language developers and time-to-fix after reporting the found failures on WebDSL's issue tracker, YellowGrass[1].

5.2 Results

In this section we present the gathered data from the test suite generation for both WebDSL revisions r5579 and r5739. We reported all encountered crash-failures to the WebDSL lan-

Parameter	r5579	r5739-1	r5739-2
Number of Test Cases	500	500	500
Max Node Count	3000	5000	250000
Min Iteration	0	0	200
Max Iteration	50	50	1500
Max Recursion	15	30	400
Terminal Pool Size	50	50	100

Figure 5.2: The parameters used in the generation of the test suites for the two tested revisions.

guage developers using the project's issue tracker[2]. These issues can be found under a special category tagged as `randomtesting`. In Appendix A we included a small report for every specific failure found during the testing of the two revisions together with its corresponding shrunk program. Figure 5.2 shows the parameters used for test suite generation for both WebDSL revisions.

5.2.1 WebDSL r5579

In this subsection we present the data gathered from the generated test suites for the previously mentioned variants and for the existing test suite.

We counted the occurrences of each type of failure in the test suites generated with each of the test strategy variants, the results are shown in Figure 5.3. Each test suite consists of 500 generated test cases. The detected failures originate from either ambiguous programs, programs containing other parse errors or programs that caused the compiler to crash. We also show the number of fixes applied by the error fixing algorithm and the average size of the generated programs in the test suite.

We detected a total of 363 compiler crashes crashes throughout the generation of these test suites. We identified eight unique crashes and present them here with a clarification. This clarification either explains the fault causing the crash or merely presents a shallow description of the failure if the fault has not yet been traced. These clarifications or descriptions are fruit of interaction with WebDSL developers and our own insight into WebDSL.

F1: Type inference. During the type-checking phase a strategy is used to infer the type of the initialization part of a variable declaration with no defined type. The fault was caused by the code responsible for handling the type inference of untyped expressions such as empty lists and sets. The code responsible for assigning these expressions the type"unknown" was executed too late causing the compiler to crash for not being able to infer the type of such expressions. This failure had been reported earlier in issue #563 and we revived the issue by reporting a shrunken generated application causing the same failure. The fault was fixed

[2]YellowGrass' Issue Tracker page for WebDSL project: `http://yellowgrass.org/project/WebDSL`

	Ambiguous	Parse Errors	Crashes	Fixes	Average Size(kB)
Basic	55	0	42	NA	2,63
Name	299	21	77	NA	91,43
75Name	291	13	75	NA	82,86
ErrorFix	58	1	43	2759	2,45
Name-ErrorFix	311	35	61	777	99,17
75Name-ErrorFix	297	16	65	929	76,15

Figure 5.3: For WebDSL r5579: Number of occurrences of failing test cases in the generated test suites for the different test strategy variants. Each test suite contains 500 test cases. The failing test cases are either **Ambiguous**, exert other **Parse Errors** or cause compiler **Crashes**. We also show the number of **Fixes** applied and the **Average Size** of the generated programs.

in revision r5731 by moving the code responsible for assigning untyped expressions the "unknown" type inside the strategy responsible for inferring types.

F2: Type pretty-printing during error reporting. During the reporting of error messages a strategy is used to pretty-print types. The expected input for this strategy is a type sort constructor. In this case the failure is caused by unexpectedly using a plain string as input for this pretty-print strategy. The failure originates from the execution of code for a constraint check for variable declarations in the constraint check stage of the analysis phase. This failure was reported in issue #463.

F3: Detecting duplicate declarations. During the constraint-check stage a strategy is used to check whether there are no duplicate definitions. The strategy fails because it tries to access a declaration that does not exist. This failure was reported in issue #674.

F4: Local renaming of pre-fetched variables. During the rename stage a strategy is used to rename pre-fetched variables. The failing behavior was caused by faulty code that contained a type check that was too strong. Whenever the pre-fetched variable did not have a type this type check would lead to the compiler exiting unexpectedly. Instead this check was softened, leading to an error report. This failure was reported in issue #729 and fixed in revision r5732.

F5: Template redefinition with undefined arguments. During the constraint check stage the signatures of local template redefinitions are compared to the original template definitions. The failure occurs whenever the arguments in this redefinition are entities that have no declaration. This failure was reported in issue #677.

	F1	F2	F3	F4	F5	F6	F7	F8
Basic	17	14	5	3	1	1	1	0
Name	29	28	2	10	2	1	3	1
75Name	43	22	1	5	1	0	3	0
ErrorFix	22	7	6	1	2	0	0	0
Name-ErrorFix	33	16	3*	6	0	0	3	0
75Name-ErrorFix	40	16	3	2	3	0	1	0
Fixed	✓	✗	✗	✓	✗	✓	✓	✗

Figure 5.4: For WebDSL r5579: Number of occurrences of failures and whether they were fixed before revision r5739.*: these three failures were found after the application of error fixes.

F6: Checking function extensions. During the type- and constraint-checking stages a strategy is used to verify that a function extension extends an existing function or report an error otherwise. The failure was being caused whenever the language construct for function extension was nested in itself, e.g. `extend extend function a( ) ...`. Such a construct was allowed syntactically, but bared no meaning semantically and the strategy did not expect such input. This was reported in issue #731 and fixed in revision r5734 by disallowing the recursive use of the `extend` keyword for function extension.

F7: Cyclic definitions. During the declaration and constraint-check stages various strategies were used to get entity and property declarations. However, these were unable to deal with cyclic definitions of either entities or properties and would enter an infinite loop leading to memory or stack overflows. This was reported in issue #733 and fixed in revision r5735 by including checks for cyclic definitions.

F8: Type resolution of partial function call reference. During the renaming of actions in the rename stage, type resolution occurs and types are stored for later use during type-checking. This failure originates from a strategy that attempts to resolve the type of a partial function call reference. This was reported in issue #739.

We present the number of occurrences of these failing strategies in Figure 5.4. The most occurring failure in the generated test suites is failure F1 followed by failure F2. Whereas failure F8 occures the less often.

In Figure 5.5 we show the coverage achieved by our generated test suite and by the existing test suite for WebDSL. The maximum constructor coverage is 625 and the maximum code coverage is 3887. Furthermore, we show both the number of strategies that have been at least partially covered and the number of those that have been fully covered, any fully covered strategy is naturally included in the partially covered count.

Test Suite	Constructor Coverage		Stratego Strategy Coverage			
			Partially		Fully	
	#	of Total	#	of Total	#	of Total
WebDSL Test Suite	393	63%	3258	84%	2742	71%
WebDSL Fail Tests	174	28%	2440	63%	1821	47%
Basic	311	50%	1207	31%	757	19%
Name	470	75%	1221	31%	785	20%
75Name	474	76%	1216	31%	781	20%
ErrorFix	331	53%	1193	31%	749	19%
Name-ErrorFix	484	77%	1208	31%	776	20%
75Name-ErrorFix	473	76%	1208	31%	785	20%

Figure 5.5: For WebDSL r5579: Constructor and Stratego Strategy coverage achieved by WebDSL's existing manually constructed test suite, a selection of this test suite containing only the test cases targetting static semantic error reporting and all generated test suites using the variants of our test strategy.

All generated test suites combined achieve a constructor coverage of 520 and a code coverage with 828 fully and 1262 partially covered strategies respectively corresponding to 21% and 32% of the total code coverage. We also compared the coverage achieved by the combined generated test suites and the existing test suite and learned that the generated test suites covered 78 coverage points that the existing test suite did not.

5.2.2 WebDSL r5739

For the generation of the test suites for this revision we use the parameters shown in Figure 5.2. The first set of parameters, r5739-1, are used for the generation of the Simple, Name and ErrorFix variants, whereas the second r5739-2 are used for the generation of the larger test suites. We noticed the earlier used parameters were too restrictive to generate large programs for the basic variant. The name variants do not suffer as much due to the terms injected to adhere to name binding properties. These terms are injected regardless of whether the maximum iteration or tree size has been reached. We generated test suites with these parameters for the Simple and Name-ErrorFix variants.

In this revision of WebDSL the faults causing the failures F1, F4, F6 and F7 were fixed. Also, we added some disambiguation rules to the WebDSL's grammar to diminish the frequency of ambiguous generation. Finally, we added a parsing timeout to deal with the excessive parser times due to large amounts of ambiguities. This is categorized as a parse error.

The types and number of occurrences of the failures encountered in the generated test suites are shown in Figures 5.7 and 5.8. Between these failing strategies we identified three new crash-failures and we describe them below.

		Shrunken	
Failure	**Original Size (kB)**	**Size (bytes)**	**LOC**
F1	5,8	123	7
F2	5,9	176	8
F3	5,3	81	5
F4	7,4	172	10
F5	6,2	78	3
F6	2,4	72	3
F7 for strategy `type-of-property`	34,1	78	7
F7 for strategy `extends-check`	7,2	159	13
F7 for strategy `is-property`	3,2	185	10

Figure 5.6: For WebDSL r5579: The original and shrunken file sizes of failure inducing programs. The LOC column shows the number of Lines Of Code in the shrunken program files for an indication of the size of the shrunken programs. These have been manually formatted since pretty-printing is not always pretty and the program was sometimes over-indented.

	Ambiguous	**Parse Errors**	**Crashes**	**Fixes**	**Average Size(kB)**
Basic	39	0	18	NA	3,09
Basic Large	4	29	58	NA	101,97
ErrorFix	49*	1	41	3287	2,63
Name	75	7	297	NA	88,65
75Name	66	8	336	NA	120,50
Name-ErrorFix Large	66	10	352*	192	148,74

Figure 5.7: For WebDSL r5739: Number of occurrences of failing test cases in the generated test suites for the different test strategy variants. Each test suite contains 500 test cases. The failing test cases are either **Ambiguous**, exert other **Parse Errors** or cause compiler **Crashes**. We also show the number of **Fixes** applied and the **Average Size** of a generated program .*: one was found after applying an error fix.

	F3	F4	F8	F9	F10	F11
Basic	2	1	0	15	0	0
Basic Large	8	1	0	43	0	6
ErrorFix	7	3	0	30	1	0
Name	0	1	1	284	11	0
75Name	4	0	0	317	15	0
Name-ErrorFix Large	5*	1	0	334	12	0

Figure 5.8: For WebDSL r5739: Number of occurrences of failing strategies for the different generated test suites. *: a failure occured after the application of an error fix.

F9: Type pretty-printing during error reporting. This is a similar failure to F1, however, the failure now originates from a strategy trying to pretty-print types during the declaration of functions in the declaration stage of analysis. This failure was reported in issue #737.

F10: Renaming of access control rules. During the renaming of access control rules the main rename strategy fails. This was reported in issue #736.

F11: Cyclic definitions. This is the same failure as failure F7. The strategy causing this failure was forgotten when the fix for cyclic definitions was applied. The failure was reported again in issue #740.

In Figure 5.9 we show the coverage achieved for the existing and generated test suites for revision r5739. The maximum constructor coverage is 584 and the maximum Stratego strategy coverage is 3874. Partially covered strategies are strategies that are at least entered and the fully covered strategies are completely executed.

All generated test suites covered 468 constructors, 1244 strategies of which 811 were fully covered. This amounts to 80% constructor coverage, 32% partial and 22% full Stratego strategy coverage. In comparison to the existing WebDSL test suite, the generated test suites covered 65 new coverage points.

5.3 Interpretation

We used the gathered results to answer our evaluation questions.

EQ1: What failures do our generated test cases unveil?

The generated test suites for both revisions were able to identify ambiguities and eleven unique compiler crashes spread through all stages of WebDSL's static analysis phase. The

Test Suite	Constructor Coverage		Stratego Strategy Coverage			
			Partially		Fully	
	#	of Total	#	of Total	#	of Total
WebDSL Test Suite	408	70%	3373	87%	2876	74%
WebDSL Fail Tests	191	33%	2478	64%	1873	48%
Basic	307	53%	1201	31%	762	20%
Basic Large	360	62%	1180	30%	749	19%
ErrorFix	330	57%	1197	31%	784	20%
Name	436	75%	1224	32%	797	21%
75Name	434	74%	1204	32%	824	21%
Name-ErrorFix Large	431	74%	1210	31%	793	20%

Figure 5.9: For WebDSL r5739: Constructor and strategy coverage achieved by the following WebDSL test suites: the existing manually constructed test suite, a selection of all error reporting test cases in the manually constructed test suite and the test suites generated using the mentiond variants.

generated programs never lead the compiler to reach the generation phase of compilation though.

EQ2: How frequently can our test strategy generate failure inducing programs?

As we can observe in Figure 5.3, most failures detected in the first revision are ambiguities. After we implemented several extra disambiguation rules in revision r5739 the generated test suites contained severely less ambiguous programs, see Figure 5.7. More than 83% of generated programs reached the analysis phase and name variants crashed the compiler with a probability between 59% and 70%. Among these, the strategy responsible for pretty-printing types in error messages was the origin of more than 80% of the crashes.

EQ3: How does the generated test suite cover the grammar and implementation of the language under test?

The obtained results show that our program generation achieves at least 50% constructor coverage. In addition, the name variant reaches a constructor coverage higher than 75% which is higher than the constructor coverage achieved by the existing WebDSL test suite. The basic generated test suite for revision r5739 achieves a constructor coverage of 53%, whereas the larger basic test suite achieves a constructor coverage of 62%. We argue that this rise in constructor coverage can be credited to larger program generation. Random generation of larger programs leads to an higher probability of including any specific constructor and thus consequentially of covering more constructors in the same number of generated test cases. The name variant, with an average program size close to that of the

larger basic test suite, achieved a constructor coverage of 75%. We argue that the name variant achieves an higher percentage than the basic generation due to the injection of definitions. By forcing the generation of these terms we raise the probability of generating productions involved in the generation of these terms. Consequentially, the generation of these productions enable the generation to easier reach parts of the grammar that were not reached by the basic generation. We conclude from these findings that the basic generation algorithm has trouble reaching certain parts of the grammar due to the uniform probability of production selection.

The strategy coverage achieved by our generated test suites is significantly lower than the coverage of the existing manually constructed test suite. Even when compared to the coverage achieved by the set of test cases that test error reporting(WebDSL Fail Tests), the generated test suites achieve only half of the existing test suite's strategy coverage. We argue that this low coverage can be explained in two ways. First, the current program generation is not able to generate name correct programs, since the name variant currently does not support all name binding properties and the name binding declarations used as input are incomplete. This causes the generated programs to never lead to the execution of the generation phase nor do they reach all strategies in the analysis phase. Secondly, only 84% of the programs generated with the name variant pass the parsing phase and 74% of those lead to crashes. A crash causes the compilation to be terminated prematurely and this prevents the execution of more strategies and thus contributes to the low strategy coverage achieved.

EQ4: Do the expansions enabling partial static semantic generation improve failure finding likelihood and coverage?

The results we obtained from the generation of test cases for the first revision were not sufficient to properly compare the failure finding likelihood of the different variants. The high rate of ambiguous generation and the small size of the programs generated with the basic generation algorithm prevent this comparison. To lower the rate of ambiguous generation we expanded WebDSL's grammar with new disambiguation rules before testing revision r5739. Additionally, we generated a test suite using the basic generation algorithm but with higher control mechanism limits to enable the generation of programs with sizes equivalent to those generated by the name variant. This enabled us to fairly compare the basic generation to the name variant and conclude that the name variant has an higher failure finding rate than the basic generation algorithm. The generated test suites for revision r5739 have a 13% and 71% crash inducing rates for the basic large and name variants respectively. This shows that for WebDSL, the generation of more name correct programs leads to an higher probability of hitting faulty code during compilation.

For both revisions we did not observe a difference in failure finding likelihood between the full obedient name variants and the 75% ones. We reason that this is due to the fact that the full obedient name variant did not generate fully name correct programs due to its prototypical nature and incomplete name declarations.

We were unable to provide sufficient evidence to justify the use of the Error Fix expansion algorithm. Throughout both revisions it was only able to find five failures after

applying a fix. Moreover, the error fixes were not enough to yield statically semantically correct programs and reach the generation phase of compilation.

EQ5: Does our program shrink heuristic yield helpful programs?

We show that the program heuristic is able to greatly reduce the size of failure inducing programs in Figure 5.6. Additionally, the positive feedback we received from the language developers after posting the shrunk programs on the issue tracker for WebDSL confirm that these were indeed much easier to understand than their original versions. The shrunk programs facilitated the tracing of the faults causing the encountered failures. Four of the first six reported failures were fixed within two days. One of which, failure F1, had been reported earlier by an user and had remained untreated for a year. The renewed interest in the failure and the provided shrunk program proved crucial in discovering the fault causing this failure and in fixing it.

5.4 Threats To Validity

We present our perceived threats to the validity of our results and evaluation. These are presented per type of threat and discussed.

External Validity

Threats to *external validity* threat the generalizability of the results. A major threat to the external validity is that we only applied our test strategy to WebDSL, diminishing our ability to show its generic nature. Though due to WebDSL's composition, being made up of three main types of sub-languages: an action definition language, an UI template definition language and a data model language; we argue that applying our test strategy to WebDSL shows it is capable of generating test cases for different types of languages. Moreover, our test strategy uses no language bound heuristic and the control parameters offer the language developer a comprehensible influence on generation to deal with differently structured grammars. Different efficiency might be experienced with different languages depending on their structure, maturity and size.

Another threat to the external validity is the choice of generation parameters. These are chosen such that the generator is ensured termination without running out of memory and they influence the structure of the generated trees. The parameters are sufficiently transparent in how they influence generation and can be adequately chosen by language developers to fit their language. Our experiences show that the parameters chosen for our evaluation give the generator sufficient freedom to yield large programs. Consequentially, language developers can adequately choose these parameters to fit the language under test, for example functional languages tend to be more recursive in nature and thus an higher recursive limit would be advised.

Internal Validity

Threats to *internal validity* are factors that allow for alternative explanations or interpretations of the result. Detected parse errors threaten the internal validity, as we ensured correct AST generation. The origin of these parse errors has been traced back to discrepancies between the language's grammar and pretty-print entries, parser timeouts and OutOfMemory exceptions caused by the disambiguator. The first should be fixed by the language developers as we believe a malformed pretty-print table may cause harm to IDE features such as pretty-printed error messages or refactoring and is thus correctly categorized as a failure. The second is a smell of a malformed grammar, since ambiguities should not be present altogether and the fact that the parser times out trying to solve them means that the program contained an alarming amount of ambiguities. This should be solved by correcting the grammar to prevent ambiguities. Another threat to internal validity is the random nature of our approach and the size of the generated test suites in number of test cases. The combination of these two raise questions as to whether observed behavior can indeed be explained by assumed logical explanations or is simply a random coincidence. Though, we argue that the chosen number of test cases per test suite is sufficiently large and the observed behavior was explained logically.

Construct Validity

Threats to *construct validity* threat the suitability of the used metrics for the evaluation goal. A major threat to the construct validity of our results is the way the compiler pipeline works and how code coverage is measured. Failure inducing programs either never execute the compiler, in case of ambiguities and parse errors, or terminate execution early, in case of crashes. Hence, the code covered by failing test cases will never be higher than non-failing test cases. This makes the for an unfair comparison between our generated test suites with a fail rate higher than 60% and the the never failing existing test suite. After fixing the faults causing the generated programs to fail, the generated test suites should achieve an higher code coverage than they do now.

Chapter 6

Related Work

In this chapter we relate our work to research in the field of automated compiler testing. We start with our assessment of our test strategy according to Boujarwah and Saleh [4] in Section 6.1. Then we relate our work to stochastic test case generation in Section 6.2 and combinatorial-coverage test case generation in Section 6.3. Stochastic test case generation [29, 40, 28, 15, 33, 7], involves generating programs using a random approach, such generation is biased towards fewer but larger programs. Combinatorial-coverage test case generation [25, 12, 34, 14] focus on generating sets of programs that completely cover a domain or a combination of domains of coverage. This method is biased towards generation of larger sets with smaller programs. Our approach is a generic stochastic generation method biased towards generation of larger programs using only language artifacts. In Section 6.4 we present work related to the test-oracle problem for compiler testing, that is how to automatically evaluate the result of the compiler execution to determine pass or no pass. In Section 6.5 we present approaches to reduce failure inducing programs whilst preserving failure inducing behavior.

6.1 Classification According to Boujarwah and Saleh

Boujarwah and Saleh [4] identified different types of testing methods, techniques for generating test data and developed assessment criteria. We assessed our test strategy using their criteria and the results are shown below

Type of grammar In theory all grammars containing definitions of terminals, non-terminals and productions can be used. The implementation of our test strategy targeting Spoofax uses SDF grammars for program generation. Additional declarative information of static semantic rules and quick fixes can be used to improve semantic coverage. The current implementation supports NaBL and a prototype DSL for error fix definitions.

Data definition Not possible.

Complete Syntax Coverage Not possible since generation yields only syntax correct programs, when provided with a correct grammar. The test strategy achieves high grammar coverage.

Complete Semantic Coverage In theory, provided sufficient static semantic declarations high semantic coverage can be achieved. Though complete coverage is unlikely due to the random nature of the generation.

Extent of automation Fully automated.

Application: Functional vs Procedural Any type of language is supported.

Implementation/Efficiency The test strategy is implementable and was implemented targeting compilers developed with Spoofax. Its application to WebDSL lead to the discovery of eleven crash inducing test cases over two different revisions. The test strategy proved to be efficient in finding failure inducing test cases for WebDSL.

Test Case Correctness Partially guaranteed. The generated test cases do not contain the traditional expected output. Instead a partial oracle is used to determine whether the generated program causes the compilation to end prematurely classifying this as a failure and if not the test case is said to pass. Though further correctness is not checked.

Concurrency No special support to test concurrency properties.

6.2 Stochastic Test Case Generation

There are various stochastic test case generation approaches: some use controlled random generation [20], others use language specific heuristics to guide generation [40] and others use attribute grammars or other forms of static semantic declarations [11, 33, 10].

The "Syntax machine" proposed by Hanford [11] is able to generate syntactically correct programs in a pseudo-random manner. The syntax machine reads grammars in a BNF format and stores those in an internal format to be used by the production algorithm. This algorithm generates pseudo-random input following the grammar of the language by rewriting non-terminals. This generation algorithm is given a recursion limit to ensure termination. The grammar used during generation is dynamic, this means that it can be changed throughout generation. This is required to handle context-sensitive syntax, such as name binding, together with the use of *syntax-generators*. These are appended to production rules to represent context-sensitive rules such as the declaration of a variable. A syntax-generator is activated whenever the production is chosen for generation. A typical syntax-generator for a production defining a variable declaration would expand the dynamic grammar with a production to use the defined variable. To deal with use-before-declarations Hanford uses a *delay qualification*. The writing of terms with a delay qualification is delayed until the terms at its right have been written formalizing the declaration it points to. Such declarations can be used to represent return types of functions. This generation method will fail if the generator can not find a production to choose from. In such a case the generator backtracks

to the last production choice and chooses another. Requiring the declaration of variables before generating productions with variable referrals most probably leads to generation of variable-poor programs.

Guilmette presented TGGS [10], a commercial generator test case generator system. TGGS Hanford's backtracking generation algorithm with their own language for defining syntax and static semantics instead of using attribute grammars. This language should make the definition of these static semantics easier than Hanford's syntax generators.

Klein and Findler [20] present a randomized property-based tester for PLT Redex, inspired by QuickCheck [15]. PLT Redex is a DSL for formalizing operational semantics. This property-based tester allows testers to write predicates to represent properties or facts of their language implemented in PLT Redex. The generator then generates programs in an attempt to falsify these predicates. Their generator uses the defined syntax as input and gradually generates larger programs by relaxing limits on the depth of ASTs, the maximum recursive use of productions, the maximum cardinality of iterations, the character set from which string terminals are generated and the complexity of generated numbers. Generation will eventually bias its production choices by randomly choosing a preferred production for each non-terminal. This gradual increase of program size generation approach does not require predefined control parameters. We propose an approach like this one as future work in Section 7.2.

The generation method proposed by Palka et al. [33] is a variation of Hanford's. First a list of applicable generation rules is generated and one is randomly chosen. If the generation fails, due to malformed rules, then the generator backtracks and another rule is used. To improve Hanford's naïve form of backtracking they introduced special rules to diminish the odds of backtracking by "guessing" types of sub-trees that will be generated.

Yang et al. [40] used stochastic generation in their automated testing tool CSmith. Csmith is a random generator for programs belonging to a subset of C based on Randprog [1]. The two main design goals of Csmith are for all generated programs to be well formed and have a single meaning according to the C standard and to maximize expressiveness while still obeying the first goal. Their generator embeds C specific heuristics and enforces them during random generation. Their generated programs will for instance always include a random number of declarations at the beginning of the generated C program which are saved and used during the generation of the rest of the program.

Our test strategy uses an approach similar to the approach used by Hanford [11], Palka et al. [33] and Guilmette [10]. Though we decided to use an injection algorithm to provide name correct program generation, instead of Hanford's backtracking algorithm. Whenever the generator attempts to write a term that requires a declaration for which none is available, the algorithm writes this term as if such a declaration was available and queues the injection of a definition. After the generation of the program the queued definitions are created and injected accordingly. We argue that our approach will yield programs with more non-terminals involved in static semantics, since our generator does not need to wait for these syntax generators to be triggered and can instead inject the terms that would trigger these

[1] A random program generator by B. Turner.https://sites.google.com/site/brturn2/randomcprogramgenerator

into the tree. E.g. a variable usage needs not to wait for its definition to have been generated and can instead be used throughout generation pending the injection of its definition. Like Yang et al. [40] in CSmith, we bias our generator to large program generation using control mechanisms to limit generation. The main difference is that our test strategy is generic and uses no language bound heuristics. This makes it applicable to any language using its grammar as input. Klein and Findler's [20] gradual increase of program size generation could improve our coverage of the input domain in favor of finding different failures.

Yang et al. also discuss the relevance of failures encountered in randomly generated code. Some of the failures occurred originate from a random structure which the odds of actually ever being written by a human being or generated by a translation or optimization are so remote that fixing the fault behind it would actually be considered counter-productive. Even more so if taken into account that the fixing of a fault usually introduces other faults. Their results however indicate that many previously unknown failures found are indeed useful and lead them to, among others, 25 faults ranked the highest priority for C compilers. We are of the opinion that a detected failure is always welcome, regardless of whether the fault behind it should be fixed. Knowledge of the existence of a fault in an implementation should always lead to a better quality assurance process.

6.3 Combinatorial-Coverage Test Case Generation

Combinatorial-coverage test case generation aims at exhaustively achieving complete coverage of a domain of coverage criteria through generation of many small test cases. Combinatorial-coverage approaches deal with the infinite domain of combinatorial-coverage by using approximations or abstractions of the otherwise infinite domain of input-output scenarios [34, 12] or by using control mechanisms to prune this domain [25, 14].

Purdom [34] presented a sentence generator to test parsers. This generator was capable of generating a small set of short sentences in a short time and for which every production in the grammar had been used at least once. This method may work well for parser testing, though as Lämmel and Harm point out [24] grammar branch coverage is too weak to sufficiently test the analysis and generation phases of compilation. Harm and Lämmel [12] present a two-dimensional approximation combinatorial approach. They propose approximations of coverage for both syntax and semantics and join these. This joined approximation of coverage is then used by their generation algorithm to construct a test suite. The algorithm uses a best-first search aiming to find minimal cases that increase the suite's coverage in the two-dimensional domain. The semantic domain is represented by attribute grammars which are syntax definitions with annotated static semantic declarations. This approach requires static semantic declarations to prune the domain of possible inputs. If declarations of the full static semantics of the language is available this approach can lead to generation of the domain required to exhaustively test a compiler. Though lack of such a complete set of static semantics may lead to the generation of test cases that never test any non-declared static semantic. Furthermore, the absence of any declaration of static semantics will yield a search domain usually too large to exhaustively generate for non-trivial languages.

Lämmel and Schulte [25] extend Purdom's [34] approach with configurable control mechanisms to prune the domain of combinatorial-coverage completeness. These mechanisms control the depth of ASTs, the maximum recursive depth of a term, the balance of ASTs such that their generation is more uniform, the generation of terms which depend on each other for static semantic meaning and finally a mechanism that provides control in the form of correct static semantic generation reminiscent of generation using attribute grammars [11]. This approach was implemented in the C#-based test-case generator Geno. Lämmel and Schulte show that their approach scales well for their baseline grammar which has 21 non-terminals and 34 productions. WebDSL poses a combinatorial challenge with its 251 non-terminals and 839 productions.

Hoffman et al. [14] proposed a similar approach though they define control mechanisms per production or sort using an attribute language.For complex languages this requires a large set of control parameters, though at the same time a fine granularity of influence in generation.

6.4 Test-oracle problem

The test-oracle problem is the question of how to evaluate the execution of a test case to determine pass or no pass. To solve this problem some apply differential testing [29, 40, 25, 6, 14, 10, 36], this relies on implementations of the same specifications and the principle that these should yield equivalent outputs when given the same input. In the case of compilers, this requires multiple language implementations to be available for the language being tested.

Even though there are a few well known DSLs with multiple implementations such as SQL, CSS and HTML, our target DSLs will rarely have more than one compiler. This prohibits the use of differential testing as a viable test validation approach.

Another option is to use an oracle, an extra computational model to verify the outcome of each test. This approach requires either the construction, derivation or generation of an extra model.

Daniel et al. [6] use oracles for Java to check certain invariants before applying differential testing. These oracles vary in complexity from simple oracles, that check that compilers do not crash, to more complicated oracles, that take the language's semantics into consideration. These simple oracles are very reminiscent of Weyuker's *partial oracles* [39]. A partial oracle is an oracle that knows whether an outcome is incorrect without knowing the correct answer, i.e. the results of a partial oracle may not always detect a fault, but if a fault is detected it is definitely present. QuickCheck uses partial oracles in the form of the defined properties for the function being tested. One or more properties can be defined, but they rarely cover the whole behavior of the function under test. Just and Schweiggert [16] tested the applicability of partial oracles in integrated environments, what these are is not relevant, however, they concluded that even though partial oracles were not complete they did prove suitable for automated testing with satisfying results for both Integration and Unit testing. Concluding, it can be useful to know that a test case induces a failure, without exactly knowing what its correct behavior should have been.

We use a partial oracle to verify the generic expected behavior of compilers: given a valid input, a compiler should be able to parse, perform analysis and if no errors are found transform or interpret the source code. We classify a valid input to a compiler as a syntactically correct program, i.e. a parsable program. To verify this behavior we require a flawless execution of the compiler resulting in either the reporting of static semantic errors or a compilable, interpretable or runnable output. Any other behavior is treated as a failure. Our oracle is however unable to detect wrong error reports or erroneous generated code.

Oracles come in many forms and formal proofs are sometimes used to derive such a trusted source of expected behavior. Formal proofs can be used to implement a verified compiler [26], that can be used to produce the correct answer for an input, or they can be used to derive formal models [36, 20, 19], that predict expected behavior.

Leroy's [26] CompCert is a verified compiler for a subset of the C language that translates this subset into PowerPC instructions. Such a verified compiler can be used as a trusted source of output generation. Sirer and Bershad [36] derive certificates from formal proofs over a grammar to validate certain language properties. They also provide a language to define grammar annotations from which more certificates are generated.

As mentioned earlier Klein and Findler [20] present a randomized property-based tester for PLT Redex, inspired by QuickCheck [15]. QuickCheck is Hughes' property-based test generator. QuickCheck provides the tester with a language in which to define a property to test. This property defines input classes and output invariants. QuickCheck then randomly generates test data based on the input's definition, invokes the method and checks whether the output adheres to the invariants. PLT Redex infers these inputs from the language's operational semantics and uses tester defined properties to check invariants. These properties are defined through predicates which too can use the language's operational semantics. Klein et al. [19] used this randomized property-tester to validate operational semantics described in nine research papers. This resulted in faults being found in every of the nine papers.

Additional language artifacts describing semantic properties could be used to enable our oracle to validate test cases according to more properties or invariants. We discuss this in our future work in Section 7.2.

6.5 Program Size Reduction

Reducing the size of failure inducing programs is crucial to facilitate the tracing of the fault causing it, especially when the generated programs causing the failure are very large.

Hildebrand and Zeller [13] first introduced the delta debugging algorithm `ddmin`, this algorithm aims at shrinking the input of a crashing program whilst preserving this crashing behavior. The algorithm reduces the input by partitioning it and running it excluding one partition at a time. If this succeeds, the new and smaller failure inducing partition of the input is again partitioned and reduction is again attempted. If it fails the granularity of the partitioning is increased and these steps are repeated until the granularity can no longer be increased. Misherghi et al. [32] introduced the Hierarchical Delta Debugging(HDD) algorithm, a variation of the Hildebrand and Zellers `ddmin` for structured input. They showed that for input that is highly structured taking advantage of this structure can greatly improve

the efficiency of the algorithm. Regehr et al. [35] further built upon this idea and took advantage of language specific semantics to improve test case reduction for C compilers. For example, whenever an argument is removed from a function definition, its occurrences in the function body is removed and all function calls are adapted. Our program shrinking heuristic is based on the HDD algorithm and uses the language's grammar to aid in the reduction of the program. In contrast to Regehr et al.'s approach our heuristic is language independent and we focus on two grammar specific shrinking opportunities: replacement of sub-trees with minimally depthed trees with an equivalent non-terminal as root and list size reduction.

Chapter 7

Conclusions and Future Work

In Section 7.1 we summarize our answers to our research questions using the evaluation and experiences gained during the development of the test strategy, testing and writing of this thesis. We then present our recommendations for future work in Section 7.2

7.1 Conclusions

RQ1: How can syntactically correct programs be generated generically from language specifications?

In Section 3.1 we present a generation algorithm that, given a grammar and a set of control mechanism limits, generates syntactically correct programs. The results presented in Chapter 5 show that our generation algorithm is able to generate syntactically correct programs.

We also observed the generation of various ambiguous programs, which is counterproductive for our test strategy. Generation of ambiguous programs is credited to ambiguous grammars, that is grammars with missing disambiguation rules. Other grammar deficiencies and discrepancies between it and the available pretty-print table lead to other parse errors. This highlights our test strategy's dependency on a correct grammar.

Our assumption that a parser did not require further testing failed to account for faulty grammars. We advise the use of specific ambiguity detection methods [2] and the fix of ambiguities within the grammar prior to the application of our test strategy. Regarding the issue of discrepancies between pretty-print tables and grammars, we propose a new implementation of the generator to enable generation of PTs instead of ASTs. Since pretty-printing a PT can be done by correctly concatenating all terminals, which yields concrete syntax that is less pretty, but always correct.

RQ2: How can big programs be generated generically whilst avoiding combinatorial explosion?

To handle combinatorial explosion during program generation we put in place a set of control mechanisms as described in Section 3.1. These mechanisms are iteration maximum, recursion maximum and a maximum node count per tree. Whenever these mechanisms detect a configurable limit has been reached they use pre-computed terminating paths to finish the tree in a syntactically correct manner. These terminating paths are minimal depth trees for a given root node, computed for every non-terminal sort in the grammar. In Chapter 5 we show that the name binding adhering generation algorithm was able to generate programs with an average size of approximately 80 kB and the basic syntactical correct generation algorithm was able to generate programs with an average size of 100 kB when given higher control mechanism parameters.

RQ3: How can static semantically correct programs be generated generically?

In Section 3.2 we present an adaption to the generation algorithm to adhere to name binding rules described in a declarative model. Since such a model can be constructed for every language this adaption remains generic. Unfortunately, for other static semantics such as type or constraint checking no declarative models are yet available. Hence, we were unable to research the feasibility of incorporating such restrictions into the generation algorithm. In Chapter 5 we present results that show that expanding the syntactically correct generation algorithm to adhere to name binding properties does raise the failure finding rate for the WebDSL language. With this finding we conclude that the set of partial or full statically semantic correct programs is a valuable generation heuristic for automatic random testing.

RQ4: How can test runs be evaluated to determine success or failure when testing compilers in a generic automated manner?

We implemented a partial oracle to evaluate test runs, by checking that the compiler does terminate compilation prematurely. Our oracle was able to detect 1465 compiler crashes leading to the report of eleven unique compiler crashes, four of which were fixed within two days. This oracle is however unable to detect faulty error reporting.

As mentioned in RQ1, generation yields a large amount of ambiguous programs, when ambiguous grammars are provided. Our oracle detects these and reports them as ambiguous programs before executing the compiler. The oracle does this for all parse errors. The absence of false positives, that is compiler crashes that would never occur in production, is guaranteed since we invoke the compiler with concrete syntax instead of directly using the generated AST as input to the analysis phase.

The use of other test run evaluation methods such as differential testing or the use of computational models is currently not supported. We argue that development of multiple implementations for DSLs with such small development teams is unlikely. Computational models can however be derived from declarative static semantic models, such as NaBL. We

discuss the use of NaBL to provide extra oracle checks in Section 7.2.

RQ5: How can we assist the language engineer in relating failures found automatically to faults in the System Under Test?

Larger generated programs have an higher likelihood of inducing failures than smaller ones, however they also tend to be difficult to read due to their excessive size and random nature. To aid language developers in the tracing of the fault causing the failure we provide a program shrink heuristic that attempts to reduce the size of the generated program whilst preserving the failure inducing behavior.

Language developer's experience with our program shrinker was very positive and thus far the program shrinker was able to reduce failure inducing programs to no more than 15 lines of code. The failures found and their corresponding shrunk programs with our test strategy were reported on the WebDSL issue tracker with Two of the eleven found issues older issues were revisited with shrunk generated programs that induced the same failure, one of which was fixed after a day of posting the shrunk code while it had been open for over an year.

7.2 Future work

In this section we present our recommendations for future work and directions in which the current solution can be further extended.

Following the path towards static semantic correct generation. Further development and research should be done to enable generic and automatic generation of statically semantically correct programs. The following two steps are necessary to enable this:

1. further development of the name variant to support more NaBL features, including support for multiple-file generation, the use of imports, horizontal scoping and any future features of NaBL,

2. a language must be developed to define type and constraint semantics in a declarative manner and the generation algorithm must be expanded to support and enforce these,

Additionally, the ability to guarantee statically semantically correct generation will further enable the addition of new properties to be checked by the partial oracle:

- statically semantically correct programs should reach and pass the generation phase,
- statically semantically incorrect programs should result in error reports,
- if a statically semantically correct program passes the generation phase, the generated code should be either compilable or executable.

Development of such languages to describe static semantics can lead to the generation of the analysis phase of compilation, like parser/parsetable generators are generated for the parsing phase. Testing could then be replaced by formal proofs of such systems and the future of such a test strategy as we describe would shift towards testing of the generation phase and potentially testing of optimizations or features not covered by the formal proofs.

Development of support for a quick fix IDE feature could also help reach statically semantically correct generation. These could be used to replace the currently manually defined error fixes used by the error fixing algorithm proposed for our test strategy.

Automatic gradual increase of program size generation. Reminiscent of Klein and Findler's [20] method to gradually increase program size generation we propose an alternative to our current parametric control mechanism. Instead of predefining the limits for the control mechanisms we could allow generation to gradually raise these, gradually increasing the size of generated programs. The ceiling of these control mechanism limits could be determined by either detection of a too high probability of the compiler running into memory issues, or maybe even the detection of high frequency of equivalent crashes. For the latter we would require an automatic crash comparison method. Currently we compare crashes manually, by analyzing the resulting stack trace, though pattern recognition techniques could be used to determine a level of similarity.

Oracle enhancement of with derived properties. Our oracle could be enhanced by enabling it to accept declarative property definitions or models derived from formal proof systems. These could provide additional checks to evaluate test runs with and potentially expand the type of compiler failures detectable by our oracle. New research could lead to development of property-based testing tools [15] or formal proof systems [20] for Spoofax, motivating such enhancements to our automated oracle.

Lenient Grammar Coverage Driven Generation. Productions that have parent nonterminals that occur with less frequency in other productions have a lower probability of being generated. To counter this syntax definition induced bias we propose the use of a lenient grammar driven generation. Using a set of incremental grammar coverage to raise the weight of productions less covered should lead to more uniform production generation. First a simple branch coverage could be used to slowly reassign weights to productions and later a context-sensitive abstract grammar coverage, as proposed by Harm and Lämmel [12], could be used to improve combinatorial grammar coverage.

Grammar-based White Box Fuzzing. Intrigued by the grammar-based white box fuzzing approach proposed by Godefroid et al. [8] we propose work towards such an approach for Spoofax developed compilers.

Grammar-based white box fuzzing uses a symbolic execution method that keeps track of branching during evaluation. Such a system does not exist for Stratego, the transformation language used in Spoofax, and would need to be developed. Such a system would keep track of the execution path of the compiler whilst making a record of which AST nodes

would be involved in the execution of strategies. Conform coding traditions in Stratego it would be interesting to keep track of what ATerms originating from the tree would match patterns in strategies. These could then be mutated or swapped to generate new test cases. Such an approach could be an alternative to Stratego code coverage driven generation, as we expect such a drive to miss out on faults caused by missing strategies. Whereas mutation of ATerms in the AST that are matched within strategies would probably lead to a better input-strategy combination coverage and thus potentially find such cases of missing code.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Bas Basten. *Ambiguity Detection for Programming Language Grammars*. PhD thesis, Universiteit van Amsterdam, December 2011.

[3] Robert Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[4] Abdulazeez S. Boujarwah and Kassem Saleh. Compiler test case generation methods: a survey and assessment. *Information & Software Technology*, 39(9):617–625, 1997.

[5] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1-2):123–178, 2006.

[6] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 185–194. ACM, 2007.

[7] Boaz Pat El. Automated test generation for Microsoft DSL tools. Master's thesis, Delft University of Technology, the Netherlands, 2010.

[8] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 206–215, Tucson, AZ, USA, 2008. ACM.

[9] Danny M. Groenewegen, Zef Hemel, and Eelco Visser. Separation of concerns and linguistic integration in WebDSL. *IEEE Software*, 27(5):31–37, 2010.

[10] Ronald F Guilmette. TGGS: A flexible system for generating efficient test case generators. 1995.

[11] Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[12] Jörg Harm and Ralf Lämmel. Two-dimensional approximation coverage. *Informatica (Slovenia)*, 24(3), 2000.

[13] Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In *ISSTA*, pages 135–145, 2000.

[14] Daniel Hoffman, David Ly-Gagnon, Paul A. Strooper, and Hong-Yi Wang. Grammar-based test generation with YouGen. *Software: Practice and Experience*, 41(4):427–447, 2011.

[15] John Hughes. Software testing with quickcheck. In Zoltn Horvth, Rinus Plasmeijer, and Viktria Zsk, editors, *Central European Functional Programming School - Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komrno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, volume 6299 of *Lecture Notes in Computer Science*, pages 183–223. Springer, 2009.

[16] Ren Just and Franz Schweiggert. Automating software tests with partial oracles in integrated environments. In *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, pages 91–94, New York, NY, USA, 2010. ACM.

[17] Lennart C. L. Kats, Rob Vermaas, and Eelco Visser. Integrated language definition testing. enabling test-driven language development. In Kathleen Fisher and Cristina Videira Lopes, editors, *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, OOPSLA 2011*, pages 139–154, New York, NY, USA, 2011. ACM.

[18] 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.

[19] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robby Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 285–296. ACM, 2012.

[20] Casey Klein and Robby Findler. Randomized testing in plt redex. In *Workshop on Scheme and Functional Programming (SFP)*, 2009.

[21] Anneke G Kleppe, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[22] Gabriël Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. Language-parametric name resolution based on declarative name binding and scope rules. In Czarnecki and Grel Hedin, editors, *Software Language Engineering, Fourth International Conference, SLE 2012, Dresden, Germany, September, 2012, Revised Selected Papers*, 2013.

[23] Alexander Kossatchev and Mikhail Posypkin. Survey of compiler testing methods. *Programming and Computer Software*, 31(1):10–19, 2005.

[24] Ralf Lämmel and Jörg Harm. Test case characterisation by regular path expressions. In Ed Brinksma and Jan Tretmans, editors, *Proc. Formal Approaches to Testing of Software (FATES'01)*, Notes Series NS-01-4, pages 109–124. BRICS, August 2001.

[25] Ralf Lämmel and Wolfram Schulte. Controllable combinatorial coverage in grammar-based testing. In M. mit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *Testing of Communicating Systems, 18th IFIP TC6/WG6.1 International Conference, TestCom 2006, New York, NY, USA, May 16-18, 2006, Proceedings*, volume 3964 of *Lecture Notes in Computer Science*, pages 19–38. Springer, 2006.

[26] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[27] Ricky T. Lindeman, Lennart C. L. Kats, and Eelco Visser. Declaratively Defining Domain-Specific Language Debuggers. In Ewen Denney and Ulrik Pagh Schultz, editors, *Proceedings of the 10th ACM international conference on Generative programming and component engineering (GPCE 2011)*, pages 127–136, New York, NY, USA, 2011. ACM.

[28] Peter M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990.

[29] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[30] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[31] Edward Miller and William E Howden. *Tutorial, software testing & validation techniques*, volume 138. IEEE Computer Society, 1978.

[32] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical Delta Debugging. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 142–151. ACM, 2006.

[33] Michał H. Pałka, Claessen, Koen, Russo, Alejandro, and John Hughes. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 91–97, New York, NY, USA, 2011. ACM.

[34] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12:366–375, 1972.

[35] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. *SIGPLAN Notices*, 47(6):335–346, jun 2012.

[36] Emin Gn Sirer and Brian N. Bershad. Using production grammars in software testing. In *DSL*, pages 1–13, 1999.

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

[38] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373, Braga, Portugal, 2007. Springer.

[39] Elaine J. Weyuker. On testing non-testable programs. *Comput. J.*, 25(4):465–470, 1982.

[40] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, San Jose, June 2011. ACM Press.

Appendix A

Failures Found

We identified eleven unique compiler crashes and present them here with a clarification. This clarification either explains the fault causing the crash or merely presents a shallow description of the failure if the fault has not yet been traced. These clarifications or descriptions are fruit of interaction with WebDSL developers and our own insight into WebDSL. We reported all encountered crashes to the WebDSL language developers using the project's issue tracker[1].

F1: Type inference

During the type-checking phase a strategy is used to infer the type of the initialization part of a variable declaration with no defined type. The fault was caused by the code responsible for handling the type inference of untyped expressions such as empty lists and sets. The code responsible for assigning these expressions the type"unknown" was executed too late causing the compiler to crash for not being able to infer the type of such expressions. This failure had been reported earlier in issue #563 and we revived the issue by reporting a shrunken generated application causing the same failure. The fault was fixed in revision r5731 by moving the code responsible for assigning untyped expressions the "unknown" type inside the strategy responsible for inferring types.

```
application v75_44M_cW6

section pf  } oqqo
  var GKY_dO := {}

  predicate  UC (  )  {
    GKY_dO
  }
```

[1] YellowGrass' Issue Tracker page for WebDSL project: http://yellowgrass.org/project/WebDSL

F2: Type pretty-printing during error reporting

During the reporting of error messages a strategy is used to pretty-print types. The expected input for this strategy is a type sort constructor. In this case the failure is caused by unexpectedly using a plain string as input for this pretty-print strategy. The failure originates from the execution of code for a constraint check for variable declarations in the constraint check stage of the analysis phase. This failure was reported in issue #463.

```
application uKNw3_RRf9d

section pG  }
  var dN4 := [ false | h7 : Tw5 in global.Tw5 ]

section >qow
  enum eH212_5 { }

  request var j7 : h7
```

F3: Detecting duplicate declarations

During the constraint-check stage a strategy is used to check whether there are no duplicate definitions. The strategy fails because it tries to access a declaration that does not exist. This failure was reported in issue #674.

```
application nx

native class L__bF_3_R24.KD4_I7PP__7 {
  constructor ( )
}
```

F4: Local renaming of pre-fetched variables

During the rename stage a strategy is used to rename pre-fetched variables. The failing behavior was caused by faulty code that contained a type check that was too strong. Whenever the pre-fetched variable did not have a type this type check would lead to the compiler exiting unexpectedly. Instead this check was softened, leading to an error report. This failure was reported in issue #729 and fixed in revision r5732.

```
application eX_B_1_5

section  P%~p pyUd
  session H9VN {
    function I ( ) : Void {
      prefetch-for OE_GB2jpo_ delete from I;
    }
  }
```

F5: Template redefinition with undefined arguments.

During the constraint check stage the signatures of local template redefinitions are compared to the original template definitions. The failure occurs whenever the arguments in this redefinition are entities that have no declaration. This failure was reported in issue #677.

```
application IeLPH__3

define lXq ( ) = g__1c4_JX_(*, g_7CEX2_)
```

F6: Checking function extensions

During the type- and constraint-checking stages a strategy is used to verify that a function extension extends an existing function or report an error otherwise. The failure was being caused whenever the language construct for function extension was nested in itself, e.g. `extend extend function a( ) ...`. Such a construct was allowed syntactically, but bared no meaning semantically and the strategy did not expect such input. This was reported in issue #731 and fixed in revision r5734 by disallowing the recursive use of the `extend` keyword for function extension.

```
application f_Po5__2B8V

extend extend function Kyk_b ( ) : Void { }
```

F7: Cyclic definitions

During the declaration and constraint-check stages various strategies were used to get entity and property declarations. However, these were unable to deal with cyclic definitions of either entities or properties and would enter an infinite loop leading to memory or stack overflows. This was reported in issue #733 and fixed in revision r5735 by including checks for cyclic definitions.

```
application S_5sqX6W_S1

section YaSzpqA
  principal is S_5sqX6W_S1 with credentials S_gGS_2Mvn5

section pq p v[-*
  native class S_5sqX6W_S1 : S_5sqX6W_S1 {  }
```

F8: Type resolution of partial function call reference

During the renaming of actions in the rename stage, type resolution occurs and types are stored for later use during type-checking. This failure originates from a strategy that attempts to resolve the type of a partial function call reference. This was reported in issue #739.

```
application t821NT

extend entity p16S___ {
  validate( function() : Void {
    append  ( pS9QM5, w3_E [  ] {
      var D_6_2g03 := function.uQ_O():Void(*,function.uQ_O():Void());
    } );
  }, I2Q78KX_Ojk)
}
```

F9: Type pretty-printing during error reporting.

This is a similar failure to F1, however, the failure now originates from a strategy trying to pretty-print types during the declaration of functions in the declaration stage of analysis. This failure was reported in issue #737.

```
application ys_UR

access control rules
  rW____c_oP_
  predicate h0qO3V__Vf3 () {
    function ( ) : Void {
      append ( externalscope . wf__H_ , QY___7LVJ with {
        v6 ( ) { define PjqK2 (G : HI, I755_5a_C_ : G ){} }
      });
    }
  }
```

F10: Renaming of access control rules.

During the renaming of access control rules the main rename strategy fails. This was reported in issue #736.

```
application O

access control rules
  yye
  rule action D__6 ( function ( ) : Void { } * ) {
    externalscope . Le8Nfm8
  }
```

F11: Cyclic definitions.

This is the same failure as failure F7. The strategy causing this failure was forgotten when the fix for cyclic definitions was applied. The failure was reported again in issue #740.

```
application cH_Y__5U

enum V8UG { }

section   LpzWyow;
  entity V : g {
  }

section  q+  DoDq
  entity U1_ : V {
  }

section CRou
  entity wKi4fKn__d2 : U1_ {
  }

section
  entity c {
    B7_PZQ <> wKi4fKn__d2 ( ) := false
  }

  entity g : wKi4fKn__d2 {
  }
```

Appendix B

Quick Fixes for WebDSL

```
noRoot:
 "no root page root() defined.", _ →
   insert once DefinePage([],"root",[], None() ,[])
     @ ApplicationDefs(_,$here,_)
  or
   insert once Section("rootSection",[DefinePage([],"root",[], None() ,[])])
     @ Application(_,$here)

bool:
 "expression should be of type Bool", $_1 →
   @_1 change to SimpleSort("Bool")

typeNotDefined:
 "Type not defined:", SimpleSort($_1) →
   insert once for (@_1) EntityNoSuper(@_1,[])
    @ ApplicationDefs(_,$here,_)
  or
   insert once for (@_1)
    Section($concat("typeNotDefSection",@_1),EntityNoSuper(@_1,[]))
    first@ Application(_,$here)

multiDefVar1:
 "defined multiple times.", VarDeclInit($_1,_,_) →
   delete for (@_1)
    VarDeclInit(@_1,_,_) or VarDeclInitInferred(@_1,_) from list

multiDefVar2:
 "defined multiple times.", VarDeclInitInferred($_1,_) →
   delete for (@_1)
    VarDeclInit(@_1,_,_) or VarDeclInitInferred(@_1,_) from list

multiDefSecurityEntity:
 "Entity 'SecurityContext' is defined multiple times.", _  →
   delete AccessControlPrincipal(_,_) from list
```

```
indexNotInt:
 "Index must be of type Int,", $_1 →  @_1 change to Int(1)

wrongPlaceForRefArgument:
 "only allowed in formal parameters of pages, templates, or ajax templates",
 Ref($_1) → Ref(@_1) change to @_1

typeEmptyList1:
 "Type cannot be determined for empty untyped list creation.", $_1 →
   @_1 change to ListCreation([Int("1")])

typeEmptySet1:
 "Type cannot be determined for empty untyped set creation.", $_1 →
   @_1 change to SetCreation([Int("1")])

functionReturn1:
 "Return statement missing in function", Function($_1,_,_,_) →
   insert once for (@_1)
    Return(Int("0")) @ Function(@_1,_,_,Block($here))

functionReturn2:
 "Return statement missing in function", StaticEntityFunction($_1,_,_,_) →
   insert once for (@_1)
    Return(Int("0")) @ StaticEntityFunction(@_1,_,_,Block($here))

entityCapital:
 "Entity name: should start with a Capital", EntityNoSuper($_1,_) →
   change all to $concat("C_", @_1)

derivedPropertyType1:
 "The expression of the derived property should have type",
 DerivedPropertyNoAnno($_1, $_2, SimpleSort($_3),$_4) →
   DerivedPropertyNoAnno(@_1,@_2,SimpleSort(@_3),@_4)
    change to
   DerivedPropertyNoAnno(@_1,@_2,SimpleSort(@_3),ObjectCreation(@_3,[]))

derivedPropertyType2:
 "The expression of the derived property should have type",
 DerivedProperty($_1, $_2, SimpleSort($_3),$_4,$_5) →
   DerivedProperty(@_1,@_2,SimpleSort(@_3),@_4,@_5)
    change to
   DerivedProperty(@_1,@_2,SimpleSort(@_3),@_4,ObjectCreation(@_3,[]))

ExpectedSimpleType1:
 "Expected: Simple type. Encountered:",
 DerivedPropertyNoAnno($_1,Simple(),SimpleSort($_2),$_3) →
   DerivedPropertyNoAnno(@_1,Simple(),SimpleSort(@_2),@_3)
    change to
   DerivedPropertyNoAnno(@_1,Simple(),SimpleSort("String"),@_3)
```

```
ExpectedSimpleType2:
 "Expected: Simple type. Encountered:",
 DerivedProperty($_1,Simple(),SimpleSort($_2),$_3,$_4) →
   DerivedProperty(@_1,Simple(),SimpleSort(@_2),@_3,@_4)
    change to
   DerivedProperty(@_1,Simple(),SimpleSort("String"),@_3,@_4)

ExpectedSimpleType3:
 "Expected: Simple type. Encountered:",
 PropertyNoAnno($_1,Simple(),SimpleSort($_2)) →
   PropertyNoAnno(@_1,Simple(),SimpleSort(@_2))
    change to
   PropertyNoAnno(@_1,Simple(),SimpleSort("String"))

ExpectedSimpleType4:
 "Expected: Simple type. Encountered:",
 Property($_1,Simple(),SimpleSort($_2),$_3) →
   Property(@_1,Simple(),SimpleSort(@_2),@_3)
    change to
   Property(@_1,Simple(),SimpleSort("String"),@_3)

cannotInstantiateString1:
 "Cannot instantiate built-in type 'String'", ObjectCreation($_1,$_2) →
   ObjectCreation("String",@_2)
    change to
   String("Some string!")
  or
   ObjectCreation(@_1,@_2)
    change to
   String("Some string2!")

cannotInstantiateString2:
 "Entity object instantiation syntax is only supported
  for entity types, found type: 'String'",
 ObjectCreation($_1,$_2) →
   ObjectCreation("String",@_2)
    change to
   String("Some string!")
  or
   ObjectCreation(@_1,@_2)
    change to
   String("Some string2!")
```