

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Master Thesis

Introducing Static Semantic
Analysis to Templates
Using JastAdd

by
A.J. Peeters

Supervisors:

prof. dr. M.G.J. van den Brand
drs. ing. B.J. Arnoldus

Eindhoven, October 9, 2009

Abstract

Code generators based on templates provide ease-of-use for the developer, but a way to ensure syntactic and semantic safety of the output is desired. A framework consisting of two existing tools, Repleo and JastAdd, is presented to guarantee that safety. Repleo is a code generator that provides the syntactic safety, and in this thesis a methodology is described to extend Repleo with static semantic checking using JastAdd.

Contents

Contents	ii
Introduction	v
1 Templates	1
1.1 Source Code with Meta-Gaps	1
1.2 Meta Language Constructs	2
1.2.1 Substitution Placeholder	3
1.2.2 Conditional Placeholder	3
1.2.3 Iterative Placeholder	5
1.2.4 Match-Replace Placeholder	6
1.2.5 Subtemplates	7
1.3 Processing Input Data	8
1.3.1 Querying	9
1.3.2 Analyzing Structure	9
1.4 Semantics of Templates	11
1.4.1 Language Independent	11
1.4.2 Language Dependent	12
1.4.3 Bad Smells	12
1.5 Syntactic Ambiguities	13
1.5.1 Ambiguous Parsing	13
1.5.2 Approaches	14
1.5.3 Applying Disambiguation	16
2 Tooling	17
2.1 Introduction to Repleo	17
2.1.1 Syntactic Safety	17
2.1.2 Components	17
2.1.3 Placeholder Grammar vs. Target Grammar	18
2.1.4 The Evaluator	19
2.2 Introduction to JastAdd	19
2.2.1 Reference Attribute Grammars	19
2.2.2 Using JastAdd	20
2.2.3 Weaving Aspects	21
2.2.4 JastAddJ	21

3	Connecting SGLR and JastAdd	23
3.1	Boolean Evaluator	24
3.1.1	Traversing the Parse Tree	24
3.1.2	Evaluating the Booleans	27
3.2	JastAddJ	27
3.2.1	Language Differences	27
3.2.2	Implementing the Connection	29
3.2.3	Extending JastAddJ with the Meta-Language	29
4	Implementing the Static Semantic Checker	31
4.1	Constructing the Meta-Language	32
4.2	Syntactic Ambiguities	34
4.2.1	Using the Context	34
4.2.2	Prioritizing Placeholders	35
4.3	Static Semantic Checker	35
4.3.1	Language Error Detection	36
4.3.2	Bad Smell Detection	36
4.3.3	Placeholder Error Detection	37
4.4	Collecting Placeholders	38
5	Validation	40
5.1	Examples	40
5.1.1	Example 1	40
5.1.2	Example 2	41
5.2	Metrics	42
5.2.1	Conclusion	44
6	Future Work	45
6.1	Subtemplates	45
6.2	Checking Using the Input Data	45
6.3	Translate Language Definitions	45
7	Related Work	46
7.1	Code Generators	46
7.1.1	Templates	46
7.1.2	Staged	46
7.1.3	Certification	47
7.2	Static Semantic Checkers	47
7.2.1	Extensible Attribute Grammar Systems	48
7.2.2	Extensible Compiler Frameworks	48
8	Conclusion	49
	Bibliography	50
A	Pico Unparser	53

CONTENTS

B PicoJava	55
B.1 PicoJava.sdf	55
B.2 PicoJava.ast	57
B.3 Placeholders.ast	58
C TML.ast	60
D Abstract Syntax Tree	61

Introduction

The underlying ambition of this project is to improve the quality of code generators, especially those based on templates. Therefore the primary research question is:

Can static semantic checking be used in template-based code generators using off-the-shelf tooling.

This thesis presents two tools which can be used to answer this question. First Repleo[1], a syntax-safe template engine, which will form the basis of the project. Secondly JastAdd[10], a Java-based compiler compiler system, that is used to perform the static semantic checking. Proceeding with these tools more direct questions can be formulated:

RQ1 *Can for this purpose a framework be created using off-the-shelf tools?*

- *Can JastAdd be coupled to Repleo?*
- *Is the resulting coupling scalable to a large commercially used language?*

RQ2 *How can static semantic checking be performed on templates?*

- *How to implement this using JastAdd?*

The structure of the thesis is as follows; In Chapter 1 templates will be introduced. Furthermore the meta-language from Repleo (Sections 1.2 and 1.3) and the semantics of templates will be discussed in this Chapter (Section 1.4). Chapter 2 will give an introduction to the tools, i.e. Repleo and JastAdd. The answers to the research questions are described in the chapters that follow. Chapter 3 will answer RQ1 and Chapter 4 will describe the solution to RQ2. Finally in Chapter 5 the implementation of the methodology is demonstrated using examples and metrics.

Chapter 1

Templates

A template is an architectural pattern that can be used to implement a code generator. Most of the template-based code generators are text-based. Gaps are left within a piece of code, which later will be filled by text stored in input data or model. The difference with other code generators is that a template is not a black box, but is an exemplar of the resulting code. A main advantage of using templates for code generators is that they are very intuitive to use. The programmer can get an idea how the eventual output code will look like from reading the template code. The most simple template is a piece of code without gaps and therefore equal to its output code.

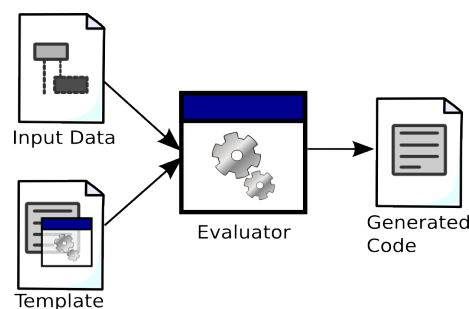


Figure 1.1: Template based generator

A common area where these template systems are used are web-based applications. For instance the creation of web-pages based on a content management system. But templates can and are used in a more expressive form than a simple text-based approach, in which input data is pasted into the template as strings.

1.1 Source Code with Meta-Gaps

For generation of code a text approach is often not expressive enough and there exist template systems that are extended with generative properties. An example of this is Apache Velocity (<http://velocity.apache.org/>).

The example in Listing 1.1, taken from the user guide of Velocity, illustrates the use of iterative and conditional commands that are added to the template system. These commands

Listing 1.1: Example of Apache Velocity

```

<HTML>
<BODY>
Hello $customer.Name!
<table>
#foreach( $mud in $mudsOnSpecial )
  #if ( $customer.hasPurchased($mud) )
    <tr>
      <td>
        $flogger.getPromo( $mud )
      </td>
    </tr>
  #end
#end
</table>

```

can be seen as a special form of gap, a meta-gap. Not a piece of data is inserted into the code, but the body of the meta-gap is inserted according to certain conditions related to the input data.

A meta-gap is defined by its syntax and semantics. The syntax of a meta-gap consists of several core elements:

- Lexical bounds to indicate the start and end of a meta-gap.
- Reference to input data.
- A body to place in the output code (in case of generative commands).

The semantic side of a meta-gap is defined by two separate entities. Obviously, a meta-gap is defined by the properties of the command it represents. Furthermore a meta-gap receives properties from the place of occurrence in the language of template, the target language. A meta-gap that is located on the place of an expression has the properties of an expression, for instance type information.

1.2 Meta Language Constructs

A way to express meta-gaps in code is by the use of placeholders, the main constructs in the meta-language presented in Repleo. For the necessary expressiveness a few variants of placeholders have been created; a substitution, a conditional, an iterative, and a Match-Replace placeholder. Each of these has their own local semantics and implications on the language. To be able to identify the impact of placeholders on a target language the details of the placeholders will first need to be explained.

The notation for a placeholder is:

```
<: "data-reference" :> ...
```

The symbols <: and :> are called *hedges*, these mark the beginning and end of each piece of meta-language within the target language. The data-reference is the link towards the input

data that will be used to fill the placeholders during the evaluation of the templates. This expression refers to the exact location within the input data or is part of a query to the specific location. A more detailed description can be found in Section 1.3.

To explain the grammar of the placeholders SDF[24]¹ is used. This provides a way to express the concrete syntax of the language and will also be used further on as part of the implementation. The placeholders will be defined as alternative production rules on the non-terminals described in the grammar of the target language. A placeholder cannot represent a lexical symbol or a terminal in language.

1.2.1 Substitution Placeholder

The substitution is the most straightforward of the placeholders. The analogy of gaps within a piece of code directly applies here. The placeholder represents all possible data that can be inserted in the specific location of its occurrence. Therefore it can be deduced that the placeholder contains exactly the same semantic properties as the non-terminal that it is replacing. In the structure shown in Figure 1.2 X represents the non-terminal the placeholder

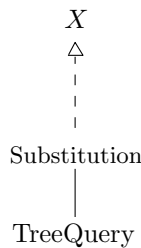


Figure 1.2: Substitution Placeholder

is replacing. The relation between the placeholder and the non-terminal is similar to inheritance. The placeholder acquires all properties the non-terminal possesses and adds some specific properties that are part of the placeholder.

In SDF the substitution is defined as follows:

```
"<:" TreeQuery ">:" -> PlaceholderSubstitution[[Sort]]
```

Based on the sort of non-terminal in the grammar that it will be replacing the placeholder will become an alternative for that non-terminal. The `TreeQuery` is the reference to the specific value from the input data that is to be inserted at that position. Further information on the functioning of this `TreeQuery` can be found in Section 1.3.

1.2.2 Conditional Placeholder

Other than the substitution placeholder the conditional does introduce new semantics. The structure is shown in Figure 1.3. The placeholder is again a derivative of the non-terminal it is replacing and acquires all of its properties. It also adds placeholder specific properties in the form of the `then` and `else` part.

In SDF this results in the following definition:

¹This is also the formalism the placeholders are defined in.

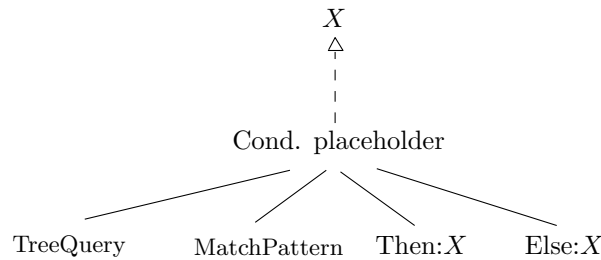


Figure 1.3: Conditional Placeholder

```

"<:" "if" TreeQuery "==" MatchPattern "then" ">"
  Sort
"<:" "else" ">"
  Sort
"<:" "fi" ">"    -> PlaceholderIfThenElse[[Sort]]
  
```

As the syntax already indicates, the functionality of this placeholder is closely related to the if-then-else construct that can be found in most imperative languages. Depending on the condition either the **then** or the **else** part will be inserted into the code. The condition is given by comparing the value found in the input data (via the `TreeQuery`) to pattern described by the `MatchPattern`. Further details on the `MatchPattern` can also be found in Section 1.3.

```

boolean <: if x == "true" then :> b <: else :> c <: fi :>;
  
```

In this example the input data retrieved using `x` must match the String literal `"true"`. To express that any input data suffices (in other words) that regardless of the input data the **then** part is selected, one should use:

```

boolean <: if x == _ then :> b <: else :> c <: fi :>;
  
```

The “`_`” represents a match to any structure or data. It is described in further detail in Section 1.2.4.

Different variants of this placeholder are available. These variants describe whether or not an **else** part may be used, or if the **then** or **else** part may contain a list. For instance when applied to an identifier or an expression in Java, there must be an **else** part present and the placeholder cannot produce a list. On the other hand a statement is in Java used as part of a list and therefore allows the omission of the **else** part and supports lists as part of the **then** or **else**. By making these explicit in the syntactic definition the parser deals with these. As an example, consider the definition of the conditional placeholder that supports lists with separators.

```

"<:" "if" TreeQuery "==" Matchpattern "then" ">"
  { Sort ";" }*
"<:" "else" ">"
  { Sort ";" }*
"<:" "fi" ">"    -> PlaceholderIfThenElseSepStar[[Sort]]
  
```

1.2.3 Iterative Placeholder

The iterative placeholder is comparable to the conditional placeholder and closely related in form to its imperative counterpart. It is used to iterate over a list in the input data, selecting an element of the list in every iteration. The formal structure is again similar to previous placeholders, but with a few exceptions. Since an iteration can be performed any number of times, the placeholder must be able to produce a list. This means that the placeholder may only be derived from a non-terminal that is (only!) used as part of a list. This fact is reflected in the structure in Figure 1.4.

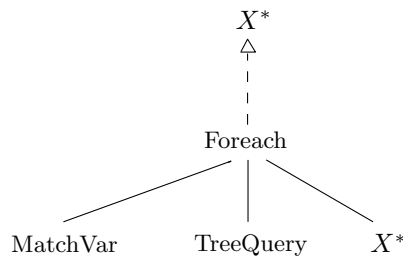


Figure 1.4: Iterative Placeholder

The definition as used in SDF is as follows.

```

"<:" "foreach" MatchVar "in" TreeQuery "do" " :>"
  Sort*
"<:" "od" " :>"  -> PlaceholderIteratorStar[[Sort]]
  
```

There is new construct, the `MatchVar`, which represents a meta-variable. Basically it is a special variable used only by placeholders. The meta-variable can contain any element or subtree from the input data. Consider the following example:

```

int i = 0;
<: foreach $x in X do :>
  i = i + <: $x :>;
<: od :>
  
```

From a location `X` in the input data, which should be a list, the `foreach` selects an element and stores it in the meta-variable. Implicitly it will never select the same element twice, although it will remain implementation specific in which order the elements are selected². The scope of the meta-variable `$x` will be confined to the body of the iterative placeholder. The resulting code of the example will for the input data `X([1,2,3])` be

```

int i = 0;
i = i + 1;
i = i + 2;
i = i + 3;
  
```

²In the implementation of Repleo the items are selected in sequence from head to tail of the list

1.2.4 Match-Replace Placeholder

The latest addition to the placeholder arsenal is the Match-Replace placeholder. The easiest analogy to make for this placeholder is that it is a pattern matcher that can express recursive calls. This makes it able to express both the iterative and the conditional properties, with the addition of checking multiple cases and handle tree structures (instead of only lists). Although this makes the conditional and iterative placeholder obsolete, these two are kept to let the placeholder language be more accessible. The fact that they are closely related to imperative languages makes them intuitive in use.

As in the previous subsections the structure of the placeholder is given. The placeholder again derives from the non-terminal it is replacing, but it has an more complex internal structure compared to the other placeholder, as shown in Figure 1.5.

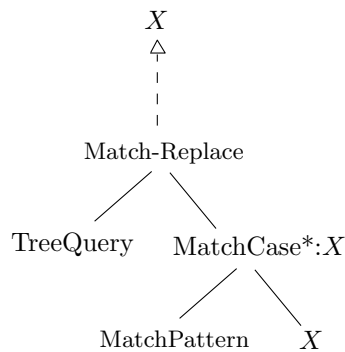


Figure 1.5: Match-Replace Placeholder

The definition in SDF is as follows:

```

"<:" "match" TreeQuery? " :>"
  MatchCase[[Sort]]+
"<:" "end" " :>"          -> PlaceholderMatchReplace[[Sort]]

"<:" MatchPattern "=" " :>"
  Sort                    -> MatchCase[[Sort]]

ATerm                    -> MatchPattern
MatchVar                 -> ATerm
"_"                     -> ATerm
  
```

The placeholder consist of two non-terminals. First the main construct, which has the `TreeQuery` which functions the same way as in the previous placeholders, except it is now optional. Implicit behavior is given when no `TreeQuery` is present, namely that the selected subtree is the root of input data for the template. Inside the Match-Replace there is a list of a new non-terminal called `MatchCase`. This non-terminal is also a descendant of the non-terminal `X`. This is not shown in the SDF definition due to the effect it would have on the parser, but semantically it contains the same properties.

As an example of the use of the Match-Replace placeholder the example of the previous section is repeated, only now in the form of the Match-Replace. Both will produce the exact same output code.

```

int i = 0;
<: match x :>
  <: [ $e, $t ] = :> i = i + <: $e :>; <: $t :>
  <: [ ] = :>
<: end :>

```

The subtree of the input data retrieved by `x` will be matched to two cases, the empty list and the non-empty list. The matching is based on implicit behavior where the list matcher is of the form `[<elem> , <list>]`. This takes the first element `$e` and provides a tail list `$t`. The element is placed in the code as a substitution, whereas the tail list marks the point of recursion. The general rule for the difference between the `$e` and the `$t` is given by the fact that the `$e` is an element, in this case a literal in the input data and can be directly inserted in the code. Whereas the `$t` is a list, which is for the input data concerned still a structure and cannot be placed into the code directly. Therefore the `$t` is a point where the match needs to be re-applied. Although the Match-Replace may select all kinds of components in the input data, it can only (semantically) process components for which it has described a `MatchPattern`. For input consisting of lists this implies that the general case and the default empty list needs to be described. For function labels this implies that all labels that can occur in the selected input are present as a `MatchPattern`.

1.2.5 Subtemplates

A template is defined as code to which input data can be assigned and is written as follows:

```

template( ...code fragment... )

```

This notation is also used to extend a template with subtemplates. The *root* template or the template that uses the full input data is given the name `template`. Templates that are used as subtemplates can have any name (with the exception of `template` and all reserved words of the meta-language). In the example in Listing 1.2 the templates are called by their name and a parameter, just like calling a function. The parameter is the relative root of input data to be used by the template. The root of *the* template is the root of the input data, the root for a subtemplate is the top of the tree selected by its argument. This makes it possible to create an unparser like shown in Listing 1.2. It takes parsed code as input and reconstructs the code. The full unparser is shown in Appendix A.

Listing 1.2: Unparser of the Pico language

```

template(
<: match :>
  <: program( decls($decls), $stms ) =:>
    begin
      declare <: decls( $decls ) :>;
      <: stms( $stms ) :>
    end
<: end :>
)

decls(
<: match :>

```

```

    <: [ $head, $tail ] =>
      <: idtype($head) :>, <: $tail :>
    <: [] =>
  <: end :>
)

stms(
  <: match :>
    <: [ $head, $tail ] =>
      <: stm($head) :>;
      <: $tail :>
    <: [] =>
  <: end :>
)
...

```

1.3 Processing Input Data

In the previous section two different constructions were used to form the connection with the input data, the `TreeQuery` and the `MatchPattern`. The first is used for extracting the specific subtree from the input data, the second is used to analyze the structure of the selected subtree. When describing a connection with input data, discussing certain aspects of the input data becomes unavoidable. A tree is the most rudimentary data structure that support forms of selection and of which the structure can be described fairly elegantly. The tree structure used in the examples has three components:

- **Literal**
Denoted by "...".
- **Functionlabel**
Denoted by `funname(item1 , item2 , ...)`, where the `funname` is the label given to the specific function.
- **List**
Denoted by `[item1 , item2 , ...]`. Items are separated by commas and the empty list is given by `[]`.

These three components can be retrieved via `TreeQuery` and the structure of those components can be analyzed via the `MatchPattern` described earlier. Trees can be formed in two ways; either via nesting of lists, or via nesting of functionlabels.

- i [["foo", "bar", [" foo "]] , "bar"]
- ii a(b("1"), c(d("2") , e("3")))

The difference between the use of these notations is the way they can be addressed from the template. The literals in i cannot be selected directly via an `TreeQuery`, but can only be accessed through selection in a `MatchPattern`. Furthermore this notation supports iteration over all of its subtrees. The properties of the latter are the opposite; direct selection is possible, but iteration is not.

The syntax of the matching expressions is based on the ATerm grammar [23] which has been extended with the meta-variable and the “_” (called the “blank”). The blank can represent any pattern and is used whenever it is important to indicate that there is a pattern, but nor the structure of the pattern nor the subtree in the input data it represents is of any importance.

1.3.1 Querying

The `TreeQuery` is used to select a subtree from the input data using the functionlabels. It is composed of one or more interconnected functionlabels via which the input data can be traversed. The subtree selected by the `TreeQuery` will be given by the contents of the functionlabel. In case of lists in the input data, the functionlabel will be located within the list. For instance take the input data:

```
[ a("foo"),b("bar"),c(d("1")) ]
```

And the following template:

```
int <: a :>;
<: a :> = <: c/d :>;
```

This will be evaluated to:

```
int foo;
foo = 1;
```

A few restriction can be found concerning the input data. The substitution placeholder can only place literals into the code, and therefore may only select those from the input data. The iterative placeholder may only select a list from the input data, it cannot iterate over a literal or a functionlabel. For the Match-Replace and the conditional placeholder no restriction are given.

For the substitution placeholder an extension is given for the `TreeQuery`. Which is the ability to concatenate multiple queries and custom strings. Given the same input data as before:

```
int <: a + "_" + b :>;
```

Will be evaluated to:

```
int foo_bar;
```

The queries given `a` and by `b` are handled separately after which the results are concatenated together.

1.3.2 Analyzing Structure

The structural analysis of the selected input data follows the three components of the input data; the literals, the functionlabels and the lists. In the section concerning the Match-Replace placeholders (Section 1.2.4) an example of matching on lists is shown. The analysis of the structure of input data is nothing else than tree pattern matching. In the field of pattern matching a lot of research has been done on the underlying theory[2, 3], but in this project the focus will reside solely on the use of pattern matching, leaving the underlying algorithms as implementation details. An example on the use of pattern matching is TOM[17].

Literals

The most rudimentary pattern is that of the literal. For the literals a Match-Replace placeholder will be similar to the switch/case statement from imperative languages. Consider the following:

```
String s;
s = <: match x :>
  <: "0" = :> "zero"
  <: "1" = :> "one"
<: end :>;
```

The quotes indicate the use of the literal which will be matched to the data selected by the Match-Replace.

Functionlabels

When matching functionlabels, the matching occurs on the label. The structure of the contents of the functionlabel can be examined and placed within a meta-variable as shown in the following example.

```
int i;
i = <: match x :>
  <: plus($lhs,$rhs) = :> ( <: $lhs :> + <: $rhs :> )
  <: minus($lhs,$rhs) = :> ( <: $lhs :> - <: $rhs :> )
  <: times($lhs,$rhs) = :> ( <: $lhs :> * <: $rhs :> )
  <: divide($lhs,$rhs) = :> ( <: $lhs :> / <: $rhs :> )
  <: inc($e) = :> ( 1 + <: $e :> )
  <: negate($e) = :> - <: $e :>
<: end :>;
```

This example shows the embedded tuples in the input data can be traversed. An example of input data selected by `x` is:

```
plus( negate("1"), times( inc("2") , inc("3") ) )
```

Giving the program:

```
int i;
i = ( -1 + (( 1 + 2 ) * ( 1 + 3 )));
```

Lists

The pattern of a list is indicated by the use of brackets (i.e. “[” and “]”). In which the elements of the list are separated by commas. Consider the following example:

```
int i = 0;
i = <: match x :>
  <: [$e, _] = :> <: $e :>
  <: [] = :> 0
<: end :>;
```

The resulting code will be either that `i` becomes the first element of the list, or in the case of an empty list 0.

1.4 Semantics of Templates

As mentioned in Section 1.1 the semantics of meta-gaps (or placeholders) are defined by their generative properties and the place of occurrence in the code. When meta-gaps are applied in a template the semantics of the language of the template also changes. In addition to the original language semantics two new semantics are introduced:

- **Language Independent**

The semantics of the placeholders independent to the target language.

- **Language Dependent**

The semantics that concerns itself with the effect the generative properties have on the target language.

The nature of the language dependent semantics are somewhat different from the semantics of the target language and the language independent semantics, which can be checked purely statically. The language dependent semantics describe the effects the meta-language has on the target language. These effects are highly dependent on the input data. Checking these semantics will require a different approach, for which the term *Bad Smells* is introduced.

1.4.1 Language Independent

The independent part concerns itself only with the meta-language. The most checks concern with meta-variables. This is illustrated by the following example.

```
int i;
<: match x :>
  <: [$e,$t] = :> i = i + <: $e :>; <: $t :>
  <: [] = :>
<: end :>
```

The meta-variables are defined within the Match-Replace and used in the corresponding body of the `MatchCase` as a substitution. For the meta-variables several rules can be applied, such as:

- i Any used meta-variable, needs to be declared.
- ii A meta-variable must be unique within its own scope. The scope of a meta-variable is given by the placeholder in which it is defined.

For the Match-Replace placeholder the first rule can be implemented by the following check. Consider the template:

```
int i = 0;
i = <: match x :>
  <: [$e,$t] = :> <: $e :> + <: $t :>
  <: [] = :> 0
<: end :>;
```

The scope of the meta-variables is within the case of the Match-Replace. The meta-variables are declared as part of the matching pattern (i.e. `[$e,$t]`), and used in the body of the corresponding case. Verification is performed by collecting all meta-variables in the declarative part (i.e. the `MatchPattern`) and in the imperative part (i.e. inside the `MatchCase`). If all

elements in the bag of the imperative part are contained within the bag of the declarative part the code is correct in accordance with the first rule. The collected bags (of the declarative part) are also used to verify the second rule, the uniqueness constraint.

1.4.2 Language Dependent

The use of placeholders will cause disturbances within the semantics of the target language. Due to the fact placeholders will be derived from non-terminals, they will adopt all properties of the non-terminal. Take for example the use of an identifier. An identifier has as property its name, i.e. the representation within the code. A placeholder derived from an identifier has indeed a name as property but does not have the actual representation in the code. For resolving a property, the way it is used in the static semantic checker is important. In case of the name of an identifier, it is used by lookup routines for checking that a variable that is used is also declared. Consider the following example;

```
int <: b :>;
<: b :> = true;
```

In this case it is possible to replace the name property with the expression within the placeholder. Doing so will make it possible to use the checker of the target language to show that this will result into a typing error.

The influence of the target language on the placeholder language also works the other way around. Look again at the example. Notable is the different ways the same placeholder is used. Although both describe a variable name, the first is used as an declaration, whereas the second is used as a reference to a declaration. The placeholders are therefore derived from two different non-terminals and are as such two different things. In this case the same placeholder is allowed to be used in both contexts. In other cases this is not always allowed. Only if the intersection of the sets of syntactically allowed input (input for which the placeholders can produce valid programs) is not empty this is permitted. Consider the example:

```
<: a :> int foo() {};
```

```
int <: a :>;
```

The placeholder <: a :> is used first as a modifier, then as variable name. As input the first accepts any reserved word that is a modifier, the second accepts any string starting with a letter, with the exception of reserved words (including modifiers). The intersection of the set of allowed input is therefore empty.

1.4.3 Bad Smells

In several cases errors can be detected via semantic checks, although often the checks performed for placeholders can only detect possible errors. Consider the following template:

```
<: foreach $x in X do :>
  int i;
<: od :>
```

This can produce correct code if the list defined by X in the input data consists of at most one element.

The difference between possible errors (induced by the placeholders) and certain errors (caused by the semantic checker of the target language) needs to be made clear. Therefore the errors produced by the static semantic checker for the language dependent part will be

indicated by the term *Bad Smells*. There remains a thin line between what can be considered a bad smell and what is not. The “possible” in the definition is an arbitrary notion. Since it is always possible to construct input data which will cause incorrect code, a border has to be set what is considered a bad smell and what is not. The border is set as follows:

Possible errors that are related to properties of placeholders are considered bad smells, whereas restrictions given only by the input data are not.

Consider the following example:

```
<: foreach $x in X do :>
  int <: $x :>;
<: od :>
```

This will produce correct code if (and only if) all elements of the list selected by **X** are unique, and will not be considered a bad smell. The first example is a bad smell considering this is related to a property of the iterative placeholder; i.e. it must be able to iterate $n + 1$ times.

1.5 Syntactic Ambiguities

When introducing meta-gaps to a fragment of code, it can happen that the same code fragment could be parsed in many different ways. The parser is able to apply different production rules to the same fragment of code. This is called a syntactic ambiguity³. Consider the following example:

```
public class A {
  <: a :> <: b :>() {}
}
```

There are two different production rules that can be applied here:

- <: a :> is a modifier and <: b :> an identifier. In this case it would be recognized a constructor.
- <: a :> is a type and <: b :> is a identifier. This produces a method declaration.

The semantics of a placeholder or derived from the location of their occurrence. Therefore when syntactic ambiguities show up, you are faced with multiple production rules and multiple interpretations for the placeholder.

1.5.1 Ambiguous Parsing

Syntactic ambiguities are a parsing problem. If the parser is unable to deal with it, static semantic analysis will not be possible in any case. Normal LL and (LA)LR parsers cannot deal with with such ambiguities. They will either produce a subtree based on the first applicable production rule or will produce a syntactic error. There exists a group of parsers designed to deal with ambiguities named Generalized LR parsers[22]. An implementation that belongs to this group is the SGLR[29] parser, which is used in all following practical implementations.

³The target language is assumed not to be ambiguous. In other words, all ambiguities that are found will be the result of the extension by the meta-language

This SGLR parser creates, when confronted with an ambiguity, a new node marked as ambiguous which contains a list containing all possibilities. If one of the possibilities contains a syntactic error this subtree will be pruned. When only one possibility remains, the ambiguity is resolved.

1.5.2 Approaches

When applying static semantic checking to templates these ambiguities are a problem. Each option of an ambiguity has semantic consequences on the checking of a template. The implications ambiguities might have on the rest of the abstract syntax tree cannot be predicted due to the inability to predict how and when the ambiguity might arise. Therefore the only way to ensure the correctness is to verify the entire program independent for each of the options in the ambiguity. This is an undesirable effect. On the syntactic level, the SGLR parser can resolve an ambiguity with the assistance of syntactic errors. Unfortunately not all ambiguities can be resolved syntactically, but on a semantic level it is also possible to resolve ambiguities.

Two approaches for disambiguation can be used for the placeholders. Using the semantic context to choose an option (described in [26]) and giving preference or prioritizing a placeholder (described in [25]).

Using Semantic Context

A generic approach is to try to find out if the context can be used to reduce the number of options in the ambiguity. The static semantic checker can assist in this process. If errors, warnings and bad smells can be collected for a specific subtree, the static semantic checker can be used to verify the separate options of the ambiguity. For each option can be verified whether there are any conflicts, warnings or errors that arise from the context. If an error has been found the options will be pruned from the ambiguity node. Consider again the example, which was slightly altered:

```
<: a :> i = 0;
public class A {
    <: a :> <: b :>() {}
}
```

This produces the same ambiguity as before, but can be resolved using the context by noting that <: a :> must be a type.

Prioritizing Placeholders

“Placeholders can occur at any place in a piece of code.” This principle implies that every non-terminal in the grammar can be replaced by a placeholder. An ambiguity always occurs when in the grammar the design pattern of specialization is used. Which is when a non-terminal has one or more production rules that contain only one non-terminal and no syntactic elements.

For instance an expression similar to the one in Java. An expression can be an operator, a variable-name or a literal.

```
Var "=" Exp    -> AssignStmt
Op              -> Exp
Var             -> Exp
```

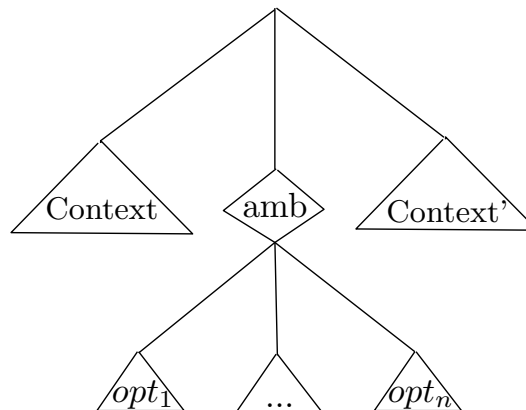


Figure 1.6: An ambiguity in relation to its context

Lit \rightarrow Exp

Consider the following fragment:

```
int i;
i = <: a :>;
```

This will produce the ambiguity shown in Figure 1.7. The placeholder `<: a :>` can be read as a placeholder derived from the `Exp` non-terminal itself, or one of the three non-terminals that are derived from the expression.

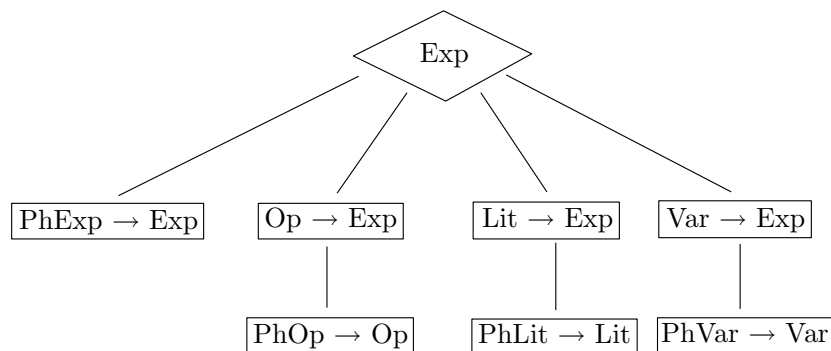


Figure 1.7: Ambiguity on Exp

The input data that can be used to replace the placeholder during evaluation can be the following:

- A variable, e.g. `i`
- An operator, e.g. `Exp "+" Exp`
- A literal, e.g. `5`

Consider the options of the ambiguity. The placeholder on the expression is the most permissive, and will accept all of the above data. The other options can only deal with their own piece of data. In this case one should choose the placeholder `PhExp`. If one of the others would be chosen this could result in false or unnecessary errors or bad smells.

In Figure 1.7 it is shown that the most permissive placeholder in this case is the placeholder that is a direct child of the ambiguity node. Since all children of the ambiguity node are syntactic equivalent, all other subtrees will result in a placeholder. Furthermore it will not contain additional branching or lexical symbols. This would negate the syntactic equivalence (unless the branching is due to another ambiguity, which will be of the same form and the same properties as this ambiguity). These properties would imply that you would just have to check if one of the options is a placeholder, and select that placeholder.

1.5.3 Applying Disambiguation

The goal of resolving an ambiguity is to choose an option that is correct and as specific as possible. Prioritization provides a way to select an option, but will always choose the most general option. However it might still be possible to ascertain from the context that only a more specific option can lead to correct output code. Therefore to choose the most specific option, the disambiguation via context is used first, the prioritization second.

At the end of the disambiguation the situation will be as follows:

- **Only one option remains**

This indicates that the ambiguity was successfully reduced.

- **No options remain**

All options contain errors.

- **More than one option remains**

Unfortunately the ambiguity could not be resolved using the context nor with prioritization. A warning will be given which indicates this as a bad smell.

The inability to resolve an ambiguity has been labeled a bad smell. The static semantic checker is not able to verify all options to their full extent and no guarantees can be given on the full static semantic correctness of the template.

Chapter 2

Tooling

In this chapter the tools will be introduced. First Repleo[1], a syntax-safe template engine, which forms the origin of this project. The meta-language described in Section 1.2 comes from this project. Second is JastAdd[10], a Java based compiler compiler system. This tool will be used to define the static semantic checker.

2.1 Introduction to Repleo

Repleo[1] has been presented as a generic system to build template evaluators, with as main purpose to add tool support for creating code generators. The basic principle of such a system is to have data and a template as input and as output correct code. The generator or evaluator is independent of the content of the code that is to be produced. To be generic the evaluator must also support multiple languages for both input and output. Repleo does indeed support all these things. Via SDF[30] definitions of languages it can be expanded to support any language, with as only requirement the existence of a grammar. The key selling point of Repleo is the fact that it only can produce syntactic correct output code.

2.1.1 Syntactic Safety

The guarantee that code can be correctly parsed is called syntactic safety. For a code generator that aims to deliver correct output code this is an important guarantee. Repleo delivers this guarantee by not just pasting the input data into the output code, but by parsing the data into the parse tree using the grammar of the object language. This enables Repleo to detect any syntactic errors during evaluation, making the output code syntactic safe.

2.1.2 Components

The inner workings of Repleo can be expressed as a pipeline of components. As first input there is the template, which is parsed by the SGLR[29] parser. The parser uses both the grammar of the target and meta-language to construct a parse tree. The SGLR parser is a component originating from the ASF+SDF Meta-Environment[24] and is a Scannerless Generalized LR parser. Scannerless means that not only the context-free syntax is used to parse a language but also the lexical part. The term Generalized refers to the fact that the parser finds *all* possible derivations for an input string. The effect of these properties is that the parse tree includes the full template after parsing, including whitespace, character-symbols

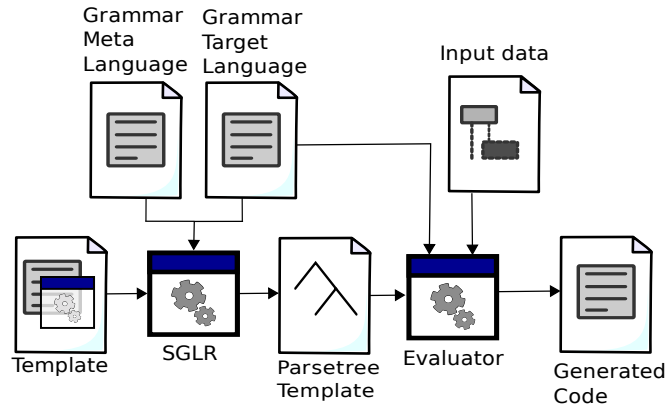


Figure 2.1: Implementation pipeline of Repleo

and locations. The generalized property is quite valuable when introducing meta-language constructs because they easily introduce syntactic ambiguities. The parse tree is subsequently evaluated, in which the meta-language constructs are being replaced by values from the input data.

2.1.3 Placeholder Grammar vs. Target Grammar

When using Repleo two different grammar definitions will be used. On the one hand the grammar of the target language, the language that forms the basis of the template language and is also the output language, and on the other hand the grammar of the meta-language. The grammar of the latter contains the form of the placeholder but not yet the application to a specific non-terminal. For reducing ambiguities Repleo was designed to not immediately include all non-terminals from the target language as placeholders. It is required to manually specify which non-terminals are replaced by which placeholder. In Listing 2.1 the form of such a specification of the connection is shown.

Listing 2.1: Merging target-language with placeholders

```
module TargetGrammar_with_Placeholder

imports TargetGrammar
imports repleo/syntax/PHNonTerminal [NonTerminal1]
imports repleo/syntax/PHNonTerminal [NonTerminal2]
```

For extending a target language, a new module in SDF needs to be created that imports the target language. In that module, for each non-terminal an `include` is added with the name of the non-terminal (here designated by `NonTerminal1` and `NonTerminal2`). The import provides a standardized way to connect a placeholder to a non-terminal. For the substitution placeholder the import will add the following production rules.

```
PlaceholderSubstitution[[NonTerminal]] -> NonTerminal
"<:" Expression ">" -> PlaceholderSubstitution[[NonTerminal]]
```

2.1.4 The Evaluator

The evaluator of Repleo is, although worthy of mentioning, not of importance for this thesis. The parts of it that are related to this research are before mentioned in Section 1.2 in the form of the described behavior of the separate placeholders, especially the implicit behavior. The evaluator is the part of Repleo that takes the parse tree, placeholders and the input data, and transforms the whole into output code. The evaluator traverses the parse tree and parses the input data into the placeholders.

2.2 Introduction to JastAdd

JastAdd[10] is a Java based system for the construction of compilers based on reference attribute grammars[9]. Like Repleo JastAdd can be made to support any language that is supported by a parser generator linkable to JastAdd. JastAdd is not equipped with a parser but it is claimed that JastAdd can be used with any parser generator. Tests have been performed using several LL and LALR based parser generators.

As shown in Figure 2.2, JastAdd takes as input a language definition and the reference attribute grammar to create Java classes. For each non-terminal in the language definition a class is generated where the attributes are woven into. This weaving is part of the aspect oriented approach used by JastAdd. Attributes and also methods can be declared outside of the class they will belong to, enabling a modular approach for creating a static semantic checker.

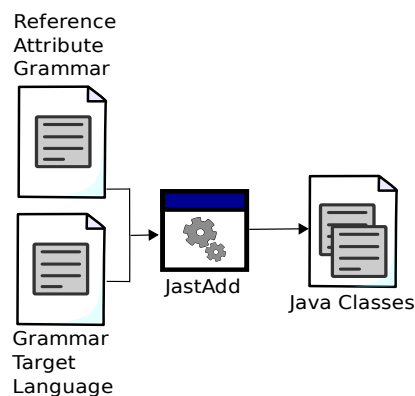


Figure 2.2: JastAdd

2.2.1 Reference Attribute Grammars

Canonical attribute grammars were introduced by Knuth[15] as a formalism to describe context-sensitive properties of constructs in a language in a declarative manner. An attribute is a property assigned to a non-terminal in the grammar and can contain values dependent

on its context in the syntax tree¹. A major problem with these canonical attribute grammars is that non-local dependencies are difficult to express. Semantic checkers often require information from nodes far away in the tree, for instance checking for duplicate declarations in a scope. Reference attribute grammars (RAGs) were introduced by Hedin[9] as a formalism that extended the canonical attribute grammars. Besides being able to store values in the attributes, RAGs also allow references to nodes in the syntax tree to be stored. For example, this makes it possible to store the reference to the declaration of a variable in an attribute attached to the use of the variable. Thus giving access to typing information from the declaration.

2.2.2 Using JastAdd

The grammar of the target language serves as input for the JastAdd component. The grammar is defined not as a concrete syntax definition, like the SDF format, but is defined by an abstract syntax definition that is a variant on the EBNF syntax which has been named AST². This relates to the fact SDF is used also as a parser definition whereas JastAdd has been made to be parser independent. An example of an AST is shown in Listing 2.2.

Listing 2.2: An example of the format of AST

```

abstract BinaryNumber ;
IntegralNumber: BinaryNumber ::= IntegralPart: BitList ;
RationalNumber: BinaryNumber ::=
    IntegralPart: BitList FractionalPart: BitList ;
abstract BitList ;
SingularBitList: BitList ::= Bit ;
PluralBitList: BitList ::= BitList Bit ;
abstract Bit ;
Zero: Bit ::= ;
One: Bit ::= ;

```

The way to read this formalism is as follows:

```
<non-terminal name> : <inherits from> ::= <production rule> ;
```

The non-terminal name is the new class which is optionally inherited from another class. The inheritance is used for specializations of non-terminals and for grouping certain non-terminals based on shared properties (e.g. the fact they possess a result type). The production rule (if present) is the rule that the non-terminal will instantiate. The production rule is of the form:

```
... ::= <label1>:<non-terminal1 > <label2>:<non-terminal2 > ;
```

The label is a name that can be given to such a link. It is not required by the language unless there are multiple non-terminals of the same sort. This is necessary since the order of the non-terminals is not important and no conclusions can be made from their relative positions.

The reserved word **abstract** in Listing 2.2 is used to indicate that the Java class derived from the non-terminal is an abstract class. In the case it has a production rule the non-terminals derived from the abstract will inherit the items of the production rule, a form of

¹Whenever the term tree or syntax tree is used, this refers to the instantiation of a grammar. Not the definition

²Although this stands for abstract syntax tree, these are definitions of a grammar and not instantiations thereof

explicit forwarding of properties within the language description. This inheritance is not only possible from an abstract non-terminal but from any non-terminal. The inherited production rule is placed at the beginning of the new production rule.

The grammar described in the `.ast` file is transformed by JastAdd to Java classes. For each non-terminal in the language definition a class is generated, each in its own file and for the contents of the production rules *get* and *set* methods are created.

2.2.3 Weaving Aspects

The RAG attributes are described separately from the grammar as aspects. The implementation of an attribute is similar to a function. The function can be defined as a value that can be set somewhere, or it can be an equation that depending on its context (the surrounding nodes in the tree) computes a result value. As mentioned, the grammar of JastAdd uses inheritance to link non-terminals, even passing on production rules down to their children. The same happens with attributes. An attribute defined for a non-terminal is passed down to all its descendants, although in case of the equations often their children will resolve the value of the attribute differently depending on their context.

When creating classes from the grammar, these attributes are incorporated into the generated classes. The system used for this resembles the *introduction* feature of the language AspectJ. These files are called Jrag modules. They consist of the name of the aspect to which the attributes relate and for each non-terminal class the definition of its attributes. There are three main modifiers for these attributes. The `inh` modifier is used for passing information down the tree, the `syn` modifier is used to pass information up the tree. The former will be defined by equations (`eq`) defined on non-terminals that contain the non-terminal for which the inherited attribute is defined. Whereas the latter, being synthesized, will be defined by an equation defined for that the same non-terminal or all its derived non-terminal separately. An example of this is show in Listing 2.3. Also normal Java code and constructs can be included into the jrag's. An example of this is the use of "+" and "-" operators as well as the use of `java.lang.math.pow()`.

Listing 2.3: An example of the format of jrag

```
aspect BinaryNumberValue {
  // Attributes
  syn double Bit.value();
  inh int Bit.scale();
  ...
  // Equations for Zero and one
  eq Zero.value() = 0;
  eq One.value() = java.lang.Math.pow(2.0, scale());
  ...
  eq PluralBitList.getBit().scale() = scale();
  eq SingularBitList.getBit().scale() = scale();
  ...
}
```

2.2.4 JastAddJ

As a case study to validate the usefulness of the JastAdd system, a Java compiler called JastAddJ[6] was developed. It supports the Java 1.4 and Java 1.5 versions. For the 1.4

specification a static semantic checker is available that includes name analysis and type analysis. Included in JastAddJ are a scanner and a parser created by respectively JFlex (<http://jflex.de/>) and Beaver(<http://beaver.sourceforge.net/>). JFlex is a scanner based on LEX [16] and Beaver is a LALR parser and variant of YACC[14].

Chapter 3

Connecting SGLR and JastAdd

Repleo to JastAdd are connected using the setup described in Figure 3.1. As shown in the figure, the challenge is to use the output of the SGLR parser to instantiate the classes generated by JastAdd. A template will be parsed by the SGLR parser based on the grammar definitions of the meta- and target language resulting in a parse tree. This parse tree will be in the AsFix[4] format. On the JastAdd side, the same grammar definition (although in an other format) will be run through the JastAdd system together with the attribute files containing the definition for the static semantic checker. This results in a set of Java classes, one for each non-terminal in the grammar definition. Connecting both can best be described as a traversal function that traverses through the parse tree created by Repleo and matches nodes in the parse tree with the classes created by JastAdd. The matching results in creating a new node as an object and continuing the traversal if possible (or required).

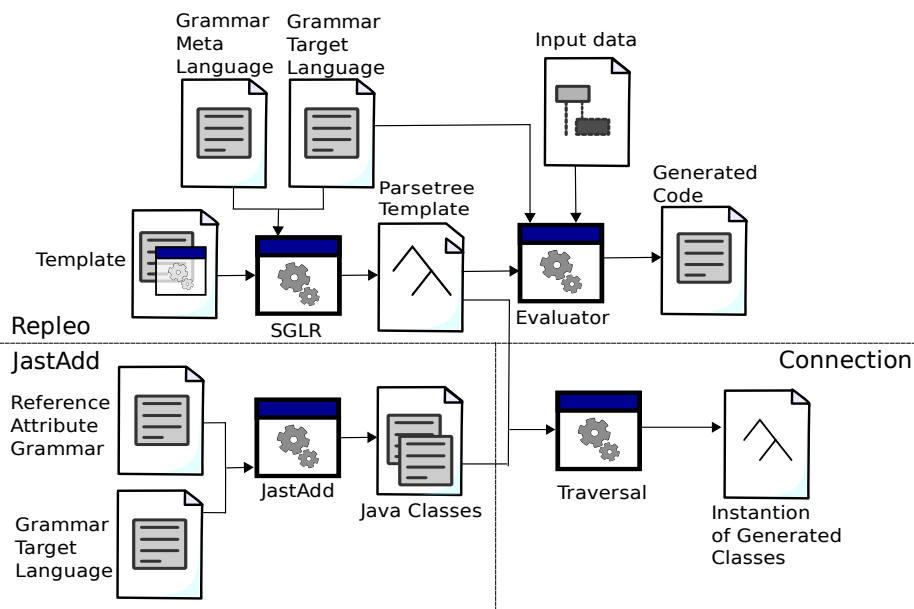


Figure 3.1: Combining JastAdd and Repleo

This chapter will answer the first of the specific research questions posed in the intro-

duction. A boolean evaluator created with the SGLR parser and JastAdd will show the possibility of the connection. The scalability of the connection to the level of commercially used languages is shown by replacing the parser provided with JastAddJ by the SGLR parser.

3.1 Boolean Evaluator

The feasibility of this approach is shown using the language of the booleans with the corresponding evaluator. The SDF definition is obtained from the Meta-Environment, which was slightly adapted to avoid naming conflicts within JastAdd. Since JastAdd generates Java classes for all of its non-terminals using reserved words of Java as non-terminals is problematic. The AST definition has been created to match.

Listing 3.1: The grammar definition of booleans in SDF

```
context-free syntax
 "(" BoolExp ")"          -> BoolExp {bracket, cons("bracket")}
 "not" "(" BoolExp ")"    -> BoolExp {cons("not")}
 lhs: BoolExp "&" rhs: BoolExp -> BoolExp {left, cons("and")}
 lhs: BoolExp "|" rhs: BoolExp -> BoolExp {left, cons("or")}
 BoolCon                  -> BoolExp {cons("constant")}
 "true"                   -> BoolCon {cons("booltrue")}
 "false"                  -> BoolCon {cons("boolfalse")}
```

Listing 3.2: The grammar definition of booleans in AST

```
BoolRoot ::= BoolExp;
abstract BoolExp;
bracket   : BoolExp ::= BoolExp;
not       : BoolExp ::= BoolExp;
and       : BoolExp ::= lhs: BoolExp rhs: BoolExp;
or        : BoolExp ::= lhs: BoolExp rhs: BoolExp;
constant  : BoolExp ::= BoolCon;
abstract BoolCon;
booltrue  : BoolCon ::=;
boolfalse: BoolCon ::=;
```

In Listings 3.1 and 3.2 the similarities and differences between the two formalisms are presented. The main differences can be traced to the difference between concrete syntax and abstract syntax. An important similarity is the relation between the names of the non-terminals of the AST and the annotation connected to the production rules in the SDF of the form `cons("...")`. These can be used to directly instantiate the classes created by JastAdd by means of reflection.

3.1.1 Traversing the Parse Tree

For bridging the gap between SGLR and JastAdd a small Java program is created that traverses the parse tree created by SGLR and instantiates the classes generated by JastAdd. This should be a one-to-one translation from the constructors in the SDF grammar (i.e. the annotation contained in the `cons("...")`) to the class names of JastAdd. The tool *ImplodePT* from the ASF+SDF Meta-Environment[24] is used to extract the constructor information from the SGLR parse tree. This tool takes a parse tree in AsFix[28] format and

produces an abstract syntax tree in ATerm[23] format that only contains the constructors. The tool *ImplodePT* in fact implements the transformation from concrete syntax to abstract syntax. A boolean expression of the form `true & false`, shown in Figure 3.2

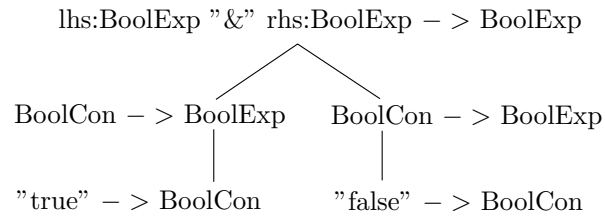


Figure 3.2: `true & false`

Will be represented (in String format) as:

```
and(constant(booltrue),constant(boolfalse))
```

The structure of this approach is shown in Figure 3.3.

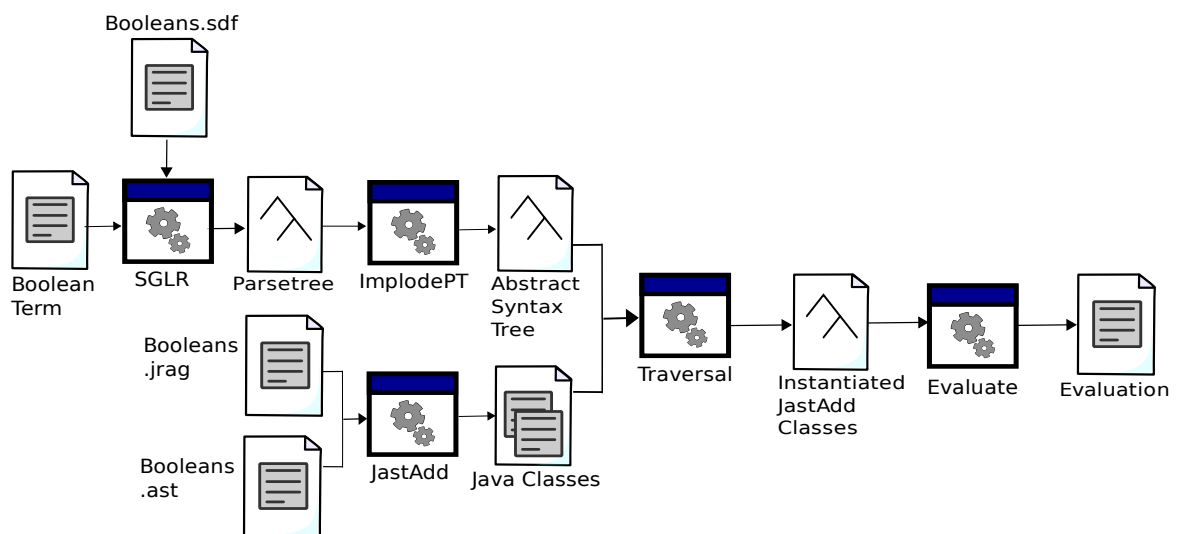


Figure 3.3: Structure of the implementation

The corresponding JastAdd-created classes are instantiated via reflection. The transformation from parse tree to JastAdd is a straightforward tree-traversal function shown (partially) in Listing 3.3. Based on the name extracted from the abstract syntax tree the corresponding class is retrieved using reflection.

Listing 3.3: Implementation of the traversal

```

private Object ParseNode(ATerm term){
    ...
    if (term_name != "") {
        try {
            Object new_node = null;

            // for the non-abstract classes find the constructors
            // and their parameters
            Class<?> Node_Class = Class.forName(term_name);
            Constructor<?> cnst [] = Node_Class.getConstructors();

            // Build the parameter list
            Object Params [] = new Object[term.getChildCount()];
            for (int i = 0; i < Params.length; i++) {
                // create for each child a new node
                Params[i] = ParseNode((ATerm) term.getChildAt(i));
            }

            // Pick the constructor based on the arguments
            for (int i = 0; i < cnst.length; i++) {
                Class<?> ParamTypes [] = cnst[i].getParameterTypes();

                Boolean b = Params.length == ParamTypes.length;
                if (b) {
                    for (int j = 0; j < ParamTypes.length; j++){
                        // for each of the parameters check equality
                        // with the constructor types
                        b = b &&
                            (Params[j].getClass().toString().equals(
                                ParamTypes[j].toString())
                                || Params[j].getClass().getSuperclass()
                                    .toString().equals(ParamTypes[j].toString()))
                    }
                    if (b) {
                        // the correct constructor has been found.
                        new_node = cnst[i].newInstance(Params);
                        break;
                    }
                }
            }
            return new_node;
        } catch (Exception e) { ... }
    } else { ... }
} else { ... }
}

```

3.1.2 Evaluating the Booleans

In order to build an evaluator with JastAdd, it is necessary to define value attributes and equations for the non-terminals. The jrag¹ for the boolean evaluator is shown in Listing 3.4. For the classes in the AST a synthesized attribute is created with a boolean type that represents the evaluated value of its subtree. The attribute value is calculated by the equations for the non-abstract non-terminals. Consider as example the equation for the `and`. The `getlhs()` and `getrhs()` will be generated from the AST by JastAdd and are exactly what they claim to be, the getters of the objects contained in respectively the right and left hand side of the production rule. The values of those two nodes are combined using the Java operator `&&`, and are stored in the value attribute of the `and`-node. The attribute is obtained for that node by inheritance from the abstract node `BoolExp`. These equations are woven into the classes created from the AST. To retrieve the evaluated boolean value from the abstract syntax tree, only the value attribute of the top-node needs to be read. Retrieving this value will apply the equations woven into the tree.

Listing 3.4: The evaluator definition for the booleans

```
aspect Booleanvalue {
  syn boolean BoolRoot.value();
  syn boolean BoolExp.value();
  syn boolean BoolCon.value();

  eq BoolRoot.value() = getBoolExp().value();

  eq and.value()      = getlhs().value() && getrhs().value();
  eq or.value()       = getlhs().value() || getrhs().value();
  eq not.value()      = !getBoolExp().value();
  eq bracket.value()  = getBoolExp().value();

  eq booltrue.value() = true;
  eq boolfalse.value() = false;
}
```

3.2 JastAddJ

For the Java 1.4 language specification² JastAdd has an implementation for which a static semantic checker has been created. Furthermore a complete SDF language definition exists for Java 1.4. Unfortunately these are not the same. In the following section will be explained what those differences are and how these were tackled.

3.2.1 Language Differences

Both language definitions are not the same. This is not due to the lack of compliance with the Java 1.4 language specification, but to design decisions made in the implementation of both SDF and JastAddJ. The differences between the language definitions can be split into two basic categories.

¹the file-format for the attribute grammar definition in JastAdd

²The specification can be found at http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html

- Rewriting during parsing.
- Instantiation of the Static Semantic Checker.

Rewriting During Parsing

In some places rewriting is performed within the parser provided with JastAddJ (i.e. Beaver³). The definition of the AST does not closely follow the Java specification anymore but combines two production rules which are still visible in the parser definition. The definition given in Listing 3.5 comes from the specification used by Beaver.

Listing 3.5: Rewriting of MethodDecl

```
MethodDecl method_declaration =
    method_header.m method_body.b    {: m.setBlockOpt(b); return m; :}
;

MethodDecl method_header =
    modifiers.m? type.t IDENTIFIER LPAREN formal_parameter_list.l? RPAREN
    dims.d? throws.tl?
    {: return new MethodDecl(new Modifiers(m), t.addArrayDims(d),
        IDENTIFIER, l, tl, new Opt()); :}

| modifiers.m? VOID IDENTIFIER LPAREN formal_parameter_list.l? RPAREN
    throws.tl?
    {: return new MethodDecl(new Modifiers(m),
        new PrimitiveTypeAccess("void"), IDENTIFIER, l, tl, new Opt()); :}
;
```

In this example the `method_header` and the `method_body` are in the AST definition combined into a single `MethodDecl`. The reason for this design decision can be found in the properties of JastAdd. The mechanisms in JastAdd support communications between parent and child nodes via the attributes, dependent on the direction of the communication via a synthesized or inherited attribute. These imply that no direct communication between sibling nodes is possible and requires an extra step or traversal. In the example there are parameters defined in the header which are used in the body. To save the extra step they chose to do rewriting on the grammar during parsing.

As can be expected, these rewrites cause problems when making the connection from SGLR to JastAdd. There are several ways to deal with these issues. For the first rewrite, the header-body combination, this was handled by changing the SDF according to the AST definition. The array dimension are a larger problem, in parsing with SGLR subtrees cannot just disappear. To deal with this neatly a rewrite would be necessary in JastAddJ, which would lead to a partial rewrite of the static semantic checker. Since this construction is already advised against in the specification, it was decided not to perform these alterations.

Instantiation of the Static Semantic Checker

Consider the following example;

```
Access qualified_name =
```

³A YACC based parser generator

```

    name.n DOT simple_name.i    {: return n.qualifiesAccess(i); :}
;

```

An additional function is called on the newly created name `n`. This function performs some pre-ambule for the resolving of names. Although this is not a problem with descendants of YACC, this is problematic for SGLR. In SGLR the separation between the parser and the static semantic checker is absolute. No additional functions on nodes can be defined in SGLR. In the implementation of the connection the traversal function deals with such cases.

A few solutions can be found to deal with this without having to compromise the language independence of the connection. First is to define an extra traversal within the static semantic checker to deal with such pre-ambule. An other is to create a special kind of annotation. In SDF it is possible to define annotations to each production rule, examples of such annotations are the constructor information(`cons()`), location information(`posinfo()`) and the placeholder details. These can be used to contain snippets of code to be performed on the node created by the production rule. But eventually these have the same effect as customizing the traversal. All are attempts to work around the strict separation between the parser and static semantic checker.

3.2.2 Implementing the Connection

The connection is implemented using the same approach (and mostly the same code) as the example of the booleans. The difference in code does not lie with the approach or the traversal but in extra functionality and a more elaborate framework to support an incremental way of working. As explained earlier the differences between the AST and the SDF definition called for a incremental working process involving a step-by-step process to increasing the amount of language supported. The extra functionality can be found in adding position information to the parse tree. This is not part of SGLR but is added via an extra tool, called `AddPosInfo`, adding position information as annotation to the parse tree created by the SGLR parser. This position information is required for error messages created by the static semantic checker. The resulting setup is shown in Figure 3.4.

3.2.3 Extending JastAddJ with the Meta-Language

The previously mentioned language differences and transformations on the tree during the traversal phase make `JastAddJ` unsuitable to be extended with placeholders. For the introduction of placeholders to a language the grammar definitions of both the SDF and AST need to facilitate a one-to-one transformation of the parse tree. A transformation during the traversal phase will result in transformations on the occurrence of the placeholders. Since the location in the parse tree is of utmost importance to the semantic properties of a placeholder, any transformation will result in additional dependencies on the placeholders.

This does not imply that extending Java with placeholders is impossible. This merely indicates that there some design decisions made in `JastAddJ` that are not beneficial to the extension of the grammar with placeholders, with as key issue the non-strict separation between parser and semantic checker.

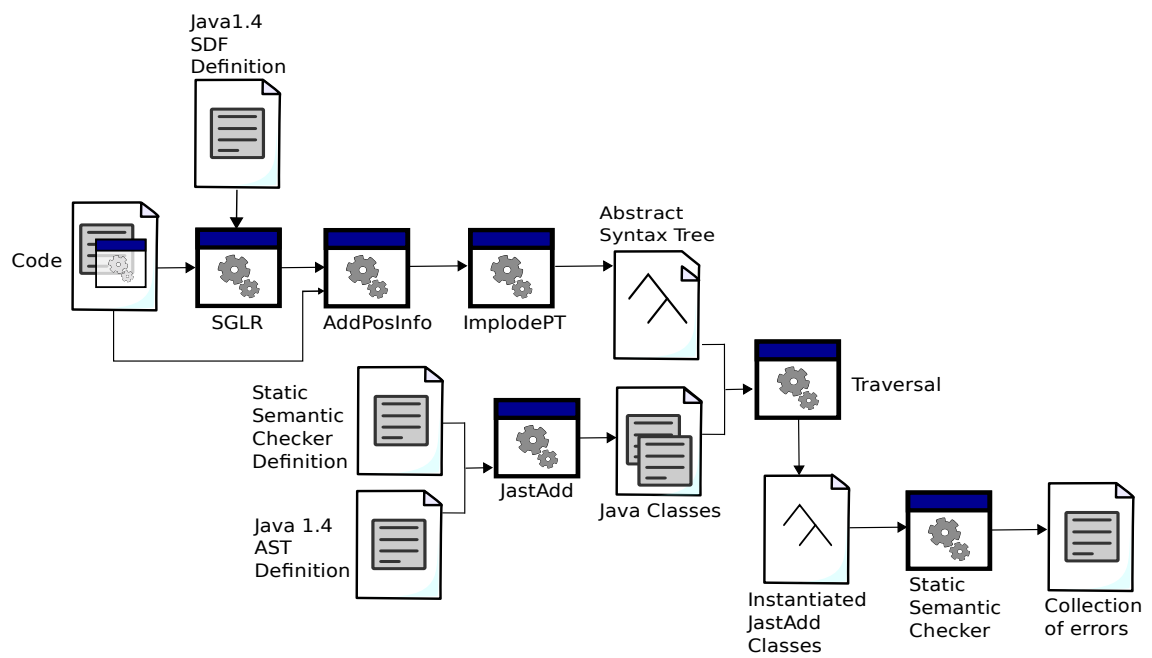


Figure 3.4: Implementation pipeline for Java

Chapter 4

Implementing the Static Semantic Checker

In Figure 4.1 the pipeline is shown how the complete system with static semantic checker would operate. Central in the pipeline is the connection defined in Chapter 3. This connection will deliver an abstract syntax tree consisting of instantiated classes. The code for the static semantic checker has been woven into these classes by JastAdd. The operations remaining in the figure are calls to methods within the top node of the tree that start the disambiguation, the semantic checking and the collecting of the placeholders.

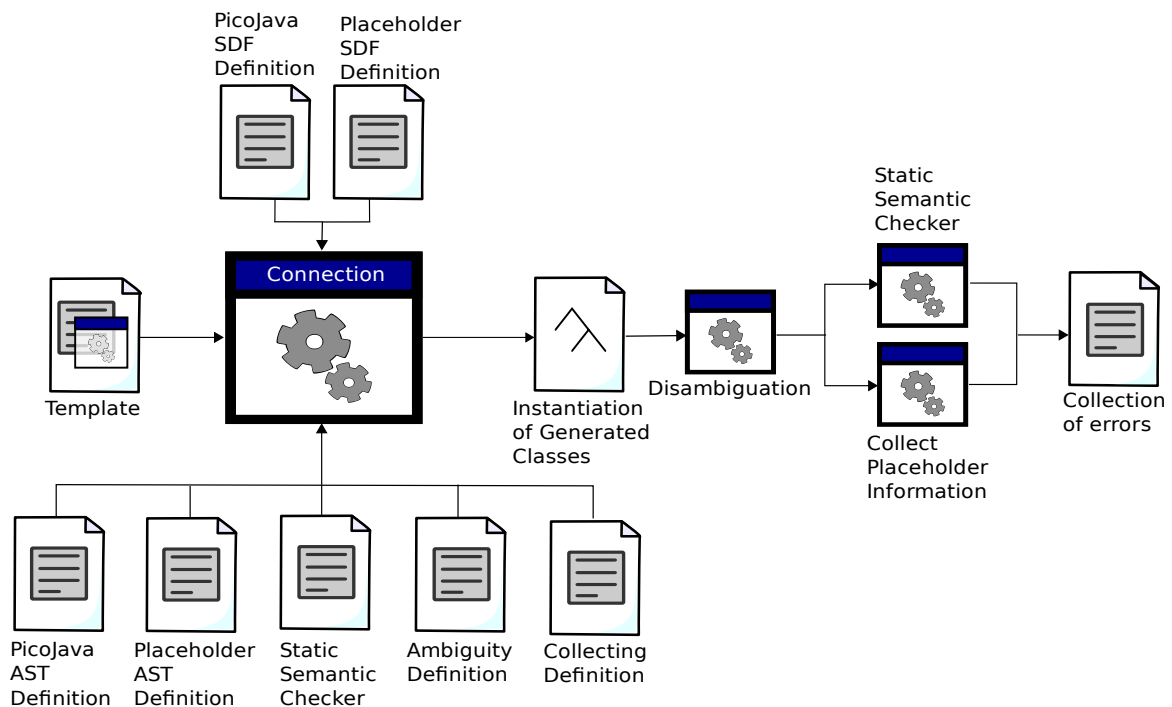


Figure 4.1: Pipeline for the static semantic checker

This chapter describes how these operations are defined using JastAdd. As object language for this demonstration a subset of Java is chosen, called PicoJava. The PicoJava language is an example that is available for JastAdd. The example comes with a fully working static semantic checker implemented with JastAdd. The smaller language gives an effective and manageable platform to demonstrate and implement the concepts designed as a part of this project. The original PicoJava language was very small, but has been extended to be able to create more interesting examples. The extensions consist of the addition of a few more primitive types, binary and unary operators, and a conditional statement.

Appendix B.1 shows the SDF definition of the PicoJava language. This definition was created to match the definition of JastAdd. The non-terminals in JastAdd can be mapped to the constructors (i.e. the annotation `cons("...")`). There is one special type of constructor which is the `Skip`. The `Skip` signifies a parsing step where no new JastAdd non-terminal needs to be created. This occurs at places where the production rule consist of a single non-terminal that does not refer to lexical syntax. In the JastAdd definition, shown in Appendix B.2, these skips relate to the use of abstract non-terminals and the inheritance between non-terminals without production rules being attached. An alternative for using the `Skip` is: omitting the constructor when no class needs to be created. Having no constructor is considered a bad practice in SDF and this alternative was rejected.

The static semantic checker that is provided by JastAdd including the language files can be found on <http://jastadd.org/jastadd-tutorial-examples/picojava-checker>. The functionality the checker contains is:

- Basic name resolution.
- Dot name resolution.
- Inheritance name resolution.
- Type analysis.

Although no claims are made towards the completeness of the semantic checker, the checks are extensive enough to create interesting examples.

4.1 Constructing the Meta-Language

As mentioned in Section 1.4 the properties of the placeholders are defined by the place of occurrence. Therefore placeholders can be derived from the non-terminal that they are replacing. This also happens to be the exact way to implement this in JastAdd. As shown in Listing 4.1, the placeholders are derived from the non-terminals they are replacing. The complete AST definition of the placeholders can be found in Appendix B.3. A fact to remember is that JastAdd creates Java Classes for each of the non-terminals and that these derivations are manifested as inheritance in the classes. All properties are propagated down to its descendants by means of the attributes that are also passed down. There are some subtle differences between the SDF and the AST. The `MatchCase` is part of the `Match-Replace` placeholder and is derived from the same non-terminal as its `Match-Replace` placeholder. Also the `MatchCase` does not contain an `ATerm` but a non-terminal called `MatchPattern`. The reason for this change is the fact that it causes naming conflicts within the traversal program. As shown in Section 3.2 the traversal program is based on the `ATerm` representation used by the SGLR parser.

Listing 4.1: Excerpt from the AST definition of the placeholders for PicoJava

```

...
// Identifier
PhIdentifier      : Identifier ::= TreeQuery;
PhIfIdentifier    : Identifier ::= TreeQuery MatchPattern
                    Then:Identifier Else:Identifier;
PhMRIdentifier    : Identifier ::= TreeQuery
                    Cases:PhMRIdentifierSort*;
PhMRIdentifierSort : Identifier ::= MatchPattern Identifier;

// Access
PhAccess         : Access      ::= TreeQuery;
PhIfAccess       : Access      ::= TreeQuery MatchPattern
                    Then:Access Else:Access;
PhMRAccess       : Access      ::= TreeQuery Cases:PhMRAccessSort*;
PhMRAccessSort  : Access      ::= MatchPattern Access;

// Exp
PhExp            : Exp         ::= TreeQuery;
PhIfExp          : Exp         ::= TreeQuery MatchPattern
                    Then:Exp Else:Exp;
PhMRExp          : Exp         ::= TreeQuery Cases:PhMRExpSort*;
PhMRExpSort     : Exp         ::= MatchPattern Exp;

// BlockStmt*
PhBlockStmt     : BlockStmt   ::= TreeQuery;
PhIfBlockStmt   : BlockStmt   ::= TreeQuery MatchPattern
                    Then:BlockStmt* Else:BlockStmt*;
PhForeachBlockStmt : BlockStmt ::= MatchVar TreeQuery BlockStmt*;
PhMRBlockStmt   : BlockStmt   ::= TreeQuery
                    Cases:PhMRBlockStmtSort*;
PhMRBlockStmtSort : BlockStmt ::= MatchPattern BlockStmt*;
...

```

For the `TreeQuery` and the `MatchPattern` `JustAdd` definitions are created, which can be found in Appendix C. These are a direct translation of the SDF definition.

To let the existing static semantic checker use these placeholders, some attributes need to be defined for the newly created placeholder classes/non-terminals. This means for the `Access` that it is required to extend the inherited attribute `lookup` and to define the equation for the `decl()` attribute.

```

inh Decl PhAccess.lookup(String name);
eq PhAccess.decl() = lookup("<: " + getTreeQuery().value() + " :>");

```

The reason for extending these attributes is that the normal semantic checker can use the placeholders as if they were part of the target language.

The full definition of the placeholders is shown in Appendix B.3. Although, not for all non-terminals placeholders have been created. Placeholders have only been created for the non-terminals that have been used in the production rules part of the AST definition given in Appendix B.2. These are the only non-terminals for which placeholders shall be used. There are two reasons for this:

- i Placeholders will only be instantiated on non-terminals within a production rule.
- ii Although the non-terminals in production rules can have descendants, the disambiguation via the prioritization rule from (Section 1.5.2) will prevent any of those descendants to be used.

This implies a further refinement on the previously made statement that is possible to replace any non-terminal by a placeholder. More precise would be that every non-terminal that is (directly) part of a production rule¹ can be replaced by a placeholder.

4.2 Syntactic Ambiguities

Before starting the static semantic checker the ambiguities first need to be removed from the abstract syntax tree. For this, PicoJava needs to be expanded to deal with the ambiguities. For each non-terminal node in the language a new node is created that represents the ambiguity for that node. Each node is derived from the corresponding non-terminal and has a list named `Options` with nodes of the same type. Like the placeholders, attributes also need to be extended. These will ensure that normal type-checking routines will remain operational.

```
// ambiguities that arise at the abstract nodes of the language
AmbBlockStmt : BlockStmt ::= Options:BlockStmt*;
AmbStmt      : Stmt      ::= Options:Stmt*;
AmbDecl      : Decl      ::= Options:Decl*;
AmbTypeDecl  : TypeDecl  ::= Options:TypeDecl*;
AmbExp       : Exp       ::= Options:Exp*;
AmbAccess    : Access    ::= Options:Access*;
AmbIdUse     : IdUse     ::= Options:IdUse*;
AmbUse       : Use       ::= Options:Use*;
AmbIdentifier: Identifier ::= Options:Identifier*;
AmbOperator  : Operator  ::= Options:Operator*;
AmbBinOperator: BinOperator ::= Options:BinOperator*;
AmbUnOperator: UnOperator ::= Options:UnOperator*;
```

This is the definition of the ambiguities for the PicoJava language. It is not necessary to create an ambiguity node for all non-terminals in the AST definition. If a non-terminal has no non-terminal descendants (effectively making it a terminal) no ambiguity can arise. Using these definitions both solutions described in Section 1.5.2 can be implemented.

4.2.1 Using the Context

As mentioned in Section 1.5.2 this method uses the error-checking routines to remove all incorrect ambiguous options. To be able to apply these routines first the whole program needs to be instantiated, including ambiguities². For the ambiguities the new nodes will be used. The ambiguity nodes function like placeholders, using the properties of the non-terminal it is replacing. Before the error checking routines will be used on the entire parse

¹This goes for the JastAdd definition of the production rule. In SDF terms this would restrict to rules that either contain lexical symbols or contain more than one non-terminal. In other words, any rule that is not a form of specialization.

²For most imperative programming languages this is not necessary, for those only the context preceding the ambiguity is required. Other languages might also be use the subsequent context. For instance in PicoJava a variable can be declared after its use.

tree a separate pass is made over the tree to attempt to resolve the ambiguities (if present). When an ambiguity is encountered, for each of the options the error- and the placeholder checker is deployed. How to deal with the results of those checks has been explained in Section 1.5.2. Due to the fact the ambiguous nodes have a list of options, the pruning of erroneous options is easily done by removing them from the list. If there is only one option left, the ambiguity node will be replaced by the remaining option.

4.2.2 Prioritizing Placeholders

The prioritization is performed if the context could not be used for the disambiguation. It simply selects the option of the ambiguity node that contains a placeholder as its top node and replaces the ambiguity node with the selected option.

The prioritization could also be applied during the traversal phase. By detecting the ambiguous node in the abstract syntax tree and checking the children the rule can be applied. In the tree an ambiguity is marked by a function symbol `amb()` with as its child a `List` of possibilities. The placeholder node will be detected by inspecting its class name, and verify that it matches the naming convention used for the ambiguities. The advantage of this lies in the performance. Using the context is costly, whilst prioritizing during the traversal takes only a single check. The disadvantage is the possibility that benefits of choosing a more specific option are lost and potential errors are missed.

The paper by van den Brand et.al.[26] describes a similar mechanism for prioritization which has been included into the ASF+SDF Meta-Environment[24] and the SGLR parser. Adding `prefer` to the annotation of the SDF production rules of the placeholders would have a similar effect as the solution in the traversal phase. Unfortunately the `prefer` contains some additional functionality (varying between different versions of the SGLR parser) which makes its operation less predictable than desired.

4.3 Static Semantic Checker

The checking of a template is split up into three parts:

- Target Language.
- Placeholder language independent.
- Placeholder language dependent.

These parts can be traced back into the code as three separate traversal functions³ which traverse over the instantiated `JastAdd` parse tree. For the routines the inheritance within `JastAdd` is used. For the abstract node class `ASTNode` the traversal itself is created.

```
public void ASTNode.collectErrors(Collection c) {
    for(int i = 0; i < getNumChild(); i++)
        getChild(i).collectErrors(c);
}
...
public void BinOperator.collectErrors(Collection c) {
```

³These can be combined to reduce the number of actual traversals. They have been kept separate for demonstrating purposes

```

    super.collectErrors(c);
    if ( !getlhs().type().isSubtypeOf(getrhs().type()) &&
        !getrhs().type().isSubtypeOf(getlhs().type()) )
        error(c, "The types of the lhs and rhs do not match");
}
...

```

This way the `collectErrors` traverses over all nodes, whilst the individual tests can be added for each separate node, as shown in the example for the binary operator. The test checks whether the types of the left-hand-side matches that of the right-hand-side.

4.3.1 Language Error Detection

This check implements the semantic rules of the target language. For the placeholders this is still of influence. As described in Section 4.1 the placeholders inherit attributes from their parents, but a number of these attributes still require equations to provide values. The ways these attributes are filled depends on the checks that use the attribute. A separation is made between attributes, those who are only locally important for the checker and those who depend on the context. The first deals with properties that is receives from its context. For instance:

```
boolean <: b :>;
```

The attribute that stores the type information is not dependent on the placeholder itself and can be processed by this error checking routine. The second part of the separation are properties that are dependent on the placeholder. Consider the following:

```
boolean <: b :>;
<: c :> = true;
```

In this case the normal type checker would indicate that the placeholder `<: c :>` would be undeclared. This is not necessarily the case since `<: b :>` might be the same as `<: c :>` or might contain a previously declared (non-placeholder) variable name. Therefore this will be considered a bad smell and belongs to the next section. A note to be made here is that the attributes must be filled to make sure that no error will be given at this point.

4.3.2 Bad Smell Detection

The bad smell detection deals with the checking of all semantic rules that exist between the target language and the placeholder language. Creating the checks that are part of this routine have to be created for each language extended with placeholders. Due to similarities between for instance imperative languages some checks can be reused but most have to be rewritten. The way to create these checks within JastAdd is to use the attributes and functions defined by the regular semantic checker, with new attributes added where needed.

Consider again the example from Section 1.4.3 where the definition of Bad Smell was explained using the iterative placeholder. The template used was:

```
<: foreach $x in X do :>
    int i;
<: od :>
```

This example was deemed a bad smell since the resulting output code would be incorrect if the iteration was performed more then once. The check, shown in Listing 4.2, searches in the

body of the iterative placeholder for variable declarations. If one is present, and the variable declared is not a placeholder, an error is given.

Listing 4.2: Checking routine for the iterative placeholder of BlockStmt

```
public void PhForeachBlockStmt.collectPhErrors(Collection c) {
    super.collectPhErrors(c);
    for (int i = 0; i < getNumBlockStmt(); i++) {
        if (getBlockStmt(i) instanceof VarDecl) {
            if (!(((VarDecl) getBlockStmt(i)).getIdentifier()
                instanceof PhIdentifier))
                error(c, "Variable declarations in a Foreach may
                    only be about placeholders");
        }
    }
}
```

4.3.3 Placeholder Error Detection

This routine concentrates on the checks that are specific for the placeholder language and check the semantic properties that have been defined. As an example on how checks are added, the implementation of the checks described in Section 1.4.1 is given.

The collecting of meta-variables either declared or used is done by adding a collection node in the MatchPattern definition.

Listing 4.3: Collecting meta-variables

```
...
syn lazy Collection MatchPattern.matchvars();
eq MatchPattern.matchvars() {
    Collection coll = new ArrayList();
    for (int i = 0; i < getNumChild(); i++) {
        if (getChild(i) instanceof MatchPattern) {
            coll.addAll( ((MatchPattern) getChild(i) ).matchvars() );
        }
    }
    return coll;
}
...
eq MatchVar.matchvars() {
    Collection coll = new ArrayList();
    coll.add( getcon() );
    return coll;
}
```

The collection of meta-variables is constructed bottom-up (via the `syn` attribute) by letting the collection of meta-variables be the union of the collections of its children. The node representing the meta-variable, the `MatchVar`, will return itself as a collection. Not only the `MatchPattern` is extended with the collection attribute but also the rest of the language. So the following check can be devised:

Listing 4.4: Comparing collections of meta-variables

```
public void PhMRExpSort.collectErrors(Collection c) {
```

```

super.collectErrors(c);

if (! getMatchPattern().matchvars().containsAll(
    getExp().matchvars())){
    Collection remainder = getExp().matchvars();
    remainder.removeAll(getMatchPattern().matchvars());
    error(c, "the meta-variables "+ remainder.toString() +
        " have been used that were not declared");
}

if (! getExp().matchvars().containsAll(
    getMatchPattern().matchvars())){
    Collection remainder = getMatchPattern().matchvars();
    remainder.removeAll(getExp().matchvars());
    error(c, "the meta-variables "+ remainder.toString() +
        " have been declared but were not used");
}
}

```

This check is defined for the `MatchCase` in the `Match-Replace` placeholder of the `Exp` non-terminal. It retrieves the collection in the declarative part (the `MatchPattern`) and the collection in the imperative part (the `Exp`) and performs the comparison.

4.4 Collecting Placeholders

By collecting all placeholders with information about the non-terminals they replace, as well as general information such as typing, placeholders can be compared and connected to each other to see if certain placeholders do not admit any valid input. For instance for expressions (and derived non-terminals) the typing is important. Consider the following example:

```

int <: i :>;
<: i :> = <: j :>;

```

For all placeholders the typing information is retrieved from the parse tree. The output given by the semantic checker is shown in Listing 4.5.

Listing 4.5: list of placeholders in output

```

—> List of PlaceHolders
(Query)   (Non-terminal)   (Type)   (Inferred Type)
i         PhIdentifier   int      $unknown
i         PhVariableUse int      $unknown
j         PhVariableUse $unknown int

```

The information gathered is the *type* and the *inferred type*. The type is the typing information used by the semantic checker of the target language. The inferred type is an addition that represents one extra derivation step. The placeholder `<: j :>` is unable to derive its type from the assignment statement it is in, but the placeholder `<: i :>` receives its type from the variable declaration. By taking one more step the type of `<: j :>` also can be derived, designated here as the inferred type. In the implementation only step has been taken, but for a fully derivation of all implied types of placeholders one could refer to techniques such as fixed-point iterations.

An other advantage of gathering information about the placeholders in this manner is that the information can be used for comparison with the input data or the design of the input data (see Section 6.2).

Chapter 5

Validation

The purpose of this chapter is to show the work that has been done implementing the presented methodology. Examples are given for the checker implemented as part of the project and described in Chapter 4. Furthermore some metrics are given to indicate how much work implementing a static semantic checker for a given language is.

5.1 Examples

5.1.1 Example 1

The first example is chosen to show the semantic checks. Therefore it contains an error for each of the checking routines.

Listing 5.1: Template 1

```
1 template(
2 {
3   <: if y == $e then :>
4     boolean i;
5     i = <: match x :>
6       <: [$e, $t] = :> <: $e :> + <: $t :>
7       <: [] = :> 0
8     <: end :>;
9   <: fi :>
10  i = 0;
11 }
12 )
```

In Listing 5.1 the errors are:

- Type error between the boolean variable `i` and the integers assigned to it.
- Duplicate declaration of the meta-variable `$e`.
- The variable `i` is declared within an conditional placeholder but used outside of it.

The template is syntactically correct and is parsed without a problem. The (pretty-printed) abstract syntax tree of the template is shown in Appendix D. The resulting output is shown in Listing 5.2.

Listing 5.2: Output for Template 1

```

—> Resolving ambiguities
6:23  Ambiguity found: Exp
      Ambiguity resolved using prioritization
6:34  Ambiguity found: Exp
      Ambiguity resolved using prioritization
Done

—> Language Errors
10:2   :  Can not assign to a variable of type boolean
       a value of type int

—> Placeholder Errors
6:6    :  duplicate declaration of meta-variables [$e]

—> Placeholder Bad Smells
7:6    :  inferred type of placeholder does not match
       the type of it's declaration
10:2   :  Variable i might not be declared, dependent on [y]

--> List of PlaceHolders
$e      PhExp          $unknown      $unknown
$t      PhExp          $unknown      $unknown
x       PhMRExp       $unknown      boolean
y       PhIfBlockStmt $unknown      $unknown

```

The errors that are inserted into the template are all found by the static semantic checker. Notable are the bad smells given in the result. The first one uses the inferred type to detect the typing difference between the variable outside of the Match-Replace placeholder and the literal 0 inside of it. In the second bad smell the relation is shown between the declaration of the variable and the condition laid upon it by the placeholder.

Due to the placeholder error, the error of the duplicate declaration of the meta-variable, the evaluator will not be able to produce output code.

5.1.2 Example 2

This second example in Listing 5.3 is similar to the previous example, with the exception that the duplicate declaration is removed. This makes it possible to show the resulting code from the evaluator. The meta-variable **\$e** is replaced by the literal value **true**.

Listing 5.3: Template 2

```

1  template(
2  {
3  <: if y == "true" then :>
4    boolean i;
5    i = <: match x :>
6      <: [$e, $t] = :> <: $e :> + <: $t :>
7      <: [] = :> 0
8    <: end :>;
9  <: fi :>
10 i = 0;

```

```
11 }
12 )
```

As input data is chosen:

```
x([1,2,3,4,5]),y("true")
```

This results in the code in Listing 5.4

Listing 5.4: Generated code for Template 2

```
{
  boolean i;
  i = 1 + 2 + 3 + 4 + 5 + 0;
  i = 0;
}
```

In the resulting code the errors and bad smells shown in the previous section are visible. The typing error is still there in both line 3 and line 4. The second bad smell is not in the resulting code. The input data selected by `y` matched the pattern and no semantic error arose in the code. If the input data was:

```
x([1,2,3,4,5]),y("false")
```

The resulting code would have been:

```
{
  i = 0;
}
```

The variable `i` is, as the bad smell indicated, not declared anymore.

5.2 Metrics

Several files with code were creating in the process of implementing the methodology described in this thesis. In the tables are shown the filename, the lines of code¹, if they are newly created for this project, and whether it is reusable when implementing a different target language. The tables are split in the following categories based on the manner they are presented in this thesis.

Table 5.1: Coupling

For the connection described in Chapter 3 the traversal through the abstract syntax tree that instantiates the generated classes is created. This traversal is independent from the target language, and can be reused. The file `main.java` is the main project file that calls all tools in the right order with the correct arguments.

Table 5.2: PicoJava

PicoJava is presented in Chapter 4. This language was already implemented in JastAdd including static semantic checker. The only new file is the SDF file. The files were altered during the implementation due to extensions made to the language to make PicoJava large enough more interesting examples. The reusability column is left empty since these files will be the ones replaced when implementing a new language.

¹including whitelines and comments

File	loc	new	reusable
main.java	72	yes	no
traversal.java	492	yes	yes
TOTAL	564		

Table 5.1: Metrics of the coupling

File	loc	new	reusable
picojava.ast	49	no	-
picojava.sdf	109	yes	-
ErrorCheck.jadd	161	no	-
NameResolution.jrag	145	no	-
NullObjects.jrag	23	no	-
PredefinedTypes.jrag	41	no	-
TypeAnalysis.jrag	112	no	-
PrettyPrint.jadd	75	no	-
TOTAL	606		

Table 5.2: Metrics of PicoJava

Table 5.3: Meta-language

In Section 4.1 the implementation of the meta-language introduced in Section 1.2 is discussed. The file `placeholders.ast` is both the coupling between the meta-language and the target language as well as the abstract syntax for the meta-language. This makes it impossible for further reuse. The remaining files describe the properties of the matching language, i.e. the language independent part of the meta-language that focuses on the connection to the input data (Section 1.3).

File	loc	new	reusable
placeholders.ast	54	yes	no
TML.ast	35	yes	yes
ATerm.jrag	12	yes	yes
TML.jrag	301	yes	yes
TOTAL	402		

Table 5.3: Metrics of the meta-language

Table 5.4: Static Semantic Checker

These files contain the definition for the attributes as well as the routines used for the checking (described in Section 4.3). Also the collecting of the placeholders is defined within these files. Since they describe routines that are defined for nodes in the target language or placeholder

derived from them, none of these files are reusable.

File	loc	new	reusable
PhErrorCheck.jadd	534	yes	no
PhAnalysis.jrag	112	yes	no
PhNameResolution.jrag	310	yes	no
TOTAL	956		

Table 5.4: Metrics of the static semantic checker

Table 5.5: Ambiguities

The ambiguities described in Sections 1.5 and 4.2 are implemented in these files. The ambiguity nodes are defined for the target language and are not reusable. The disambiguation on the other hand is defined for the generic node and does not refer to language dependent ambiguity nodes.

File	loc	new	reusable
ambiguities.ast	18	yes	no
ambiguity_attributes.jrag	40	yes	no
amb_errorcheck.jadd	60	yes	no
TOTAL	118		

Table 5.5: Metrics of the ambiguities

5.2.1 Conclusion

The lines of code are comparable to writing lines for a condensed form of Java, as a rough estimate of the work creating them. Although JastAdd provides an easy and fast way to create a static semantic checker for a language, this does not mean creating one is a trivial matter. In JastAdd creating the checks themselves is not difficult, as can be deduced by looking at the size of the listings in Chapter 4. But the hardest work lies in the creation of the attributes. This requires some skill in envisioning the tree structure of the grammar and insight in the the target language. Though writing the attributes down can be fast and efficient.

In the tables several files, beside the PicoJava and semantic files, were indicated not to be reusable. This is because attributes need to be directly defined for the non-terminals that form the connection between the target and the meta-language. The definition of these attributes is composed largely of repetitions to apply the same properties to all nodes in the tree. Therefore it might be possible that for a large part these files can be generated based on the language definition.

Chapter 6

Future Work

6.1 Subtemplates

Although subtemplates are part of the approach for templates, no semantic rules or checks have been constructed. Subtemplates can be used for recursion as a tree structure, as shown in the example in Appendix A. This implies that dependencies between templates can be described for which checks can be devised.

6.2 Checking Using the Input Data

As mentioned in Section 4.4 information about possible input data for each placeholder can be derived from the templates. This enables a whole range of new possible checks.

- **Use the taxonomy or design of the input data**

The restrictions put into the design of the input data can be used as additional checks on the placeholders. For instance if the input data is in the form of a database the properties of the attributes can be used. Such restrictions could be typing, range or uniqueness.

- **Check the input data based on the template**

It will also be possible to turn everything around. Take the template as leading source and use the template to check the (probable) validity of an input data set. This approach could be useful when a structure for the input data is used for which little or no restrictions can be made on the input.

6.3 Translate Language Definitions

In the methodology described in this thesis all language definitions are defined twice. First as SDF format for Repleo (SGLR) and second as AST, the input format for JastAdd. The difference between the two formalisms is that SDF is a concrete syntax definition whereas AST is an abstract syntax definition. Ideally only one language definition needs to be created, which means that the other has to be extracted from the single definition. It should be possible to extract an abstract syntax definition from a concrete one. Such a transformation has been described in [33].

Chapter 7

Related Work

As related work context and alternatives for the tools are given. First Repleo is positioned in respect to the field of code generators and second alternative approaches for the semantic checking now performed by JastAdd are discussed.

7.1 Code Generators

There are many approaches for code generation, as well described in the book by Jack Herrington [11]. Three categories of code generators are listed here; templates, staged languages and certificate systems. Although there are other categories of code generators, these are chosen because work has been done on semantic safety in projects belonging to them.

7.1.1 Templates

What templates are has already been described in Chapter 1. However in relation to other code generators the main selling point of templates is the ease-of-use. The WYSIWYG¹ approach to creating code generators gives the developer a direct view of how resulting code will look like, since templates represent concrete object code.

7.1.2 Staged

Staged languages are programming languages in which the programmer is able to specify each stage of the process from code to execution explicitly. In the context of program generation there are three distinct stages; generation, compilation, and execution. The staged languages allow programmers to have a more direct control over the stages and get the ability to perform analysis at each stage. Typically these are extensions of general-purpose languages and are only applicable for those languages.

Multi Stage Languages MetaML[21] is a multi-staged extension to the language ML, a general purpose functional language. MetaML ensures the type safety of the resulting code. MetaPhor[18] is a project inspired by MetaML and is a meta-language that uses reflection to add the theory devised in MetaML to object-oriented languages such as of Java and C#

¹What You See Is What You Get

Two Stage Languages DynJava[20] is an extension to Java that presents a two-stage approach for adding dynamic code generation² to Java. It has a static type system that prevents errors in the generation. Another example is a code generating dialect of Cyclone[12] which has similar properties. Both are less expressive than MetaML.

Comparison

In contrast to the language independent approach given in this thesis staged code generators are extensions to a general purpose language. Thus they are only usable with the language they extend.

7.1.3 Certification

An other approach to verifying the correctness of the generator is to extend a language with assertion-like commands. The assertions will be used as certificates of the output, indicating the compliance to rules that have been stated. This approach combines the checking with the generation process. As a result checking cannot be performed prior to the generation process.

Certifiable Program Generation[5] This paper concerns itself with defining the technique of delivering certificates, as well as providing a proof on concept implemented in the AutoBayes[7] and AutoFilter[32] program generators.

SafeGen[13] is a meta-language with which a code generator can be defined that has as input a Java program or class and as output a well-formed Java program. The created generator encompasses both the generated commands and the static code that will be part of the result. When a generator is created it is tested with help of a static semantic checker. The SafeGen checker is a combination of a traditional Java type checker and an automated theorem prover which provides the certificate. The theorem prover used is SPASS[31], a prover created by the Max-Planck-Institute.

Comparison

The checking of the code generation cannot be performed prior to the generation process³. The approach in thesis allows full static semantic checking prior to the evaluation, which makes the development of code generators easier.

7.2 Static Semantic Checkers

In this thesis JastAdd has been used to create the static semantic checker. The key point for choosing a tool for a static semantic checker for templates is its extensibility. The template meta-language can be seen as an universal extension to any general purpose language. As such, a tool designed for extending or extended languages would be perfect for a further extension.

²Dynamic code generation produces and executes code during run-time

³SafeGen can provide some feedback, but cannot check the generator fully.

7.2.1 Extensible Attribute Grammar Systems

There are several projects that are, like JastAdd, based on extensions of attribute grammars. An example of this is:

Silver[27] Silver uses extensions for referencing other parts of the parse tree similar to JastAdd. Furthermore an extension to attribute grammars called forwarding is introduced to implement extensions to a language in a more cost-effective manner.

Comparison

Of the projects named here Silver is the one that is closest to JastAdd in terms of usability and functionality. From the information available on the project it would seem that this tool could be used instead of JastAdd. The deciding factor for choosing JastAdd is the availability of detailed tutorials, examples and manuals for JastAdd, thus making the learning curve of the tool less steep. Furthermore the existence of JastAddJ provided the possibility to also extend Java with the meta-language.

7.2.2 Extensible Compiler Frameworks

There are besides attribute grammar systems other options to implement an extensible system. Examples of these are:

Eli[8] A compiler construction system created by using a collection of off-the-shelf tools controlled by a central system designed for managing user requests.

PolyGlot[19] A compiler construction framework focused around creating extensions or modifications for Java.

Comparison

Both systems provide ways to define extensible compilers, but contain much more than the need posed in this thesis. Furthermore, PolyGlot is focused solely on Java, which would go against the language independent approach presented here.

Chapter 8

Conclusion

In this thesis a methodology is presented for performing static semantic analysis on templates. Considering the entire framework formed by Repleo and JastAdd, you see a complete methodology that provides an easy-to-use and powerful way of developing a code generator. The templates provide the ease-of-use for the developer, Repleo guarantees the syntactic safety, and JastAdd the semantic safety. In the introduction the research question was split into two parts; creating a framework using off-the-shelf tools, and performing the static semantic checking.

RQ1

The presented framework is created using only off-the-shelf tools, except for a small Java program implementing the connection between both tools. In the framework, Repleo provides the parsing and the evaluation of the templates and JastAdd is used to provide the static semantic checking. They were coupled by means of a single traversal on the parse tree. The factor that allows this combination is the grammar definition that forms the basis of both tools. If the two grammar definitions are identical the coupling is reduced to a one-to-one linking.

RQ2

What the semantics of templates are is explained in Section 1.4. Semantics can be given to meta-gaps in the code according to the place of occurrence and their own generative properties. The way to check them using JastAdd is described in Chapter 4. The choice for JastAdd was a good one. The extensible nature of JastAdd makes it perfectly suited for extending a language with the template meta-language. Furthermore it provides a concise and effective way to describe semantic properties on non-terminals in the grammar.

Bibliography

- [1] J. Arnoldus, J. Bijpost, and M.G.J. van den Brand. Repleo: a syntax-safe template engine. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 25–32, New York, NY, USA, 2007. ACM.
- [2] L.G.W.A Cleophas, M.G.J. van den Brand, B.W. Watson, and C. Hemerik. *Tree algorithms: two taxonomies and a toolkit*. PhD thesis, Technische Universiteit Eindhoven, 2008.
- [3] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: <http://www.grappa.univ-lille3.fr/tata>, 2008. release November, 18th 2008.
- [4] M. de Jonge, E. Visser, and J. Visser. Definition of the syntax definition formalism SDF and its parse tree format AsFix. *Technical report, CWI*, 2000.
- [5] E. Denney and B. Fischer. Certifiable program generation. In *Generative Programming and Component Engineering*, volume 3676/2005 of *Lecture Notes in Computer Science*, pages 17–28. Springer Berlin / Heidelberg, 2005.
- [6] T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming*, October 2007.
- [7] B. Fischer and J. Schumann. AutoBayes: a system for generating data analysis programs from statistical models. *Journal of Functional Programming*, 13(03):483–508, 2003.
- [8] R.W. Gray, S.P. Levi, V.P. Heuring, A.M. Sloane, and W.M. Waite. Eli: a complete, flexible compiler construction system. *Commun. ACM*, 35(2):121–130, 1992.
- [9] G. Hedin. Reference attributed grammars. *Informatica*, (24):301–317, 2000.
- [10] G. Hedin and E. Magnusson. The JastAdd system - an aspect-oriented compiler construction system. *Science of Computer Programming*, (47):37–58, 2003.
- [11] J. Herrington. *Code Generation in Action*. Manning Publications Co., Greenwich, CT, USA, 2003.
- [12] L. Hornof and T. Jim. Certifying compilation and run-time code generation. *Higher-Order and Symbolic Computation*, 12(4):337–375, December 1999.

- [13] S.S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with SafeGen. In R. Glück and M. Lowry, editors, *Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 309–326. Springer-Verlag, 2005.
- [14] S.C. Johnson. YACC – Yet Another Compiler-Compiler. *Technical Report CS-32*, 1975.
- [15] D.E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.
- [16] M. E. Lesk and E. Schmidt. LEX – a lexical analyzer generator. *NIX Programmers Supplementary Documents*, (1), 1986.
- [17] P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer, 2003.
- [18] G. Neverov and P. Roe. Towards a fully-reflective meta-programming language. In *ACSC '05: Proceedings of the Twenty-eighth Australasian conference on Computer Science*, pages 151–158, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
- [19] N. Nystrom, M.R. Clarkson, and A.C. Myers. *Polyglot: An Extensible Compiler Framework for Java*, volume 2622/2003 of *Lecture Notes in Computer Science*, pages 138–152. Springer Berlin / Heidelberg, 2003.
- [20] Y. Oiwa, H. Masuhara, and A. Yonezawa. DynJava: Type safe dynamic code generation in Java. In *JSSST Workshop on Programming and Programming Languages*, March 2001.
- [21] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.
- [22] M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.
- [23] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Softw. Pract. Exper.*, 30(3):259–291, 2000.
- [24] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 365–370, London, UK, 2001. Springer-Verlag.
- [25] M.G.J. van den Brand, A.S. Klusener, L. Moonen, and J.J. Vinju. Generalized parsing and term rewriting : semantics driven disambiguation. In *LDTA 2003: Proceedings of the Third Workshop on Language Descriptions, Tools, and Applications*, volume 82 of *Electronic Notes in Theoretical Computer Science*, pages 575–591, April 2003.
- [26] M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation filters for Scannerless Generalized LR parsers. In *Proceedings 11th International Conference, CC*

BIBLIOGRAPHY

- 2002, held as part of *ETAPS 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer, April 2002.
- [27] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an extensible attribute grammar system. *Electronic Notes in Theoretical Computer Science*, 203(2):103 – 116, 2008. Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007).
- [28] J.J. Vinyu. UPTR: a simple parse tree representation format. *Software Transformation Systems Workshop*, October 2006.
- [29] E. Visser. Scannerless Generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.
- [30] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.
- [31] C. Weidenbach. Combining superposition, sorts and splitting. *Handbook of automated reasoning*, pages 1965–2013, 2001.
- [32] J. Whittle and J. Schumann. Automating the implementation of Kalman filter algorithms. *ACM Trans. Math. Softw.*, 30(4):434–453, 2004.
- [33] D.S. Wile. Abstract syntax from concrete syntax. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 472–480, New York, NY, USA, 1997. ACM.

Appendix A

Pico Unparser

```
[
template(
  <: match :>
    <: program( decls($decls), $stms ) =:>
      begin
        declare <: decls( $decls ) sort:ID-TYPE* :>;
        <: stms( $stms ) sort:STATEMENT* :>
      end
    <: end :>
)

decls(
  <: match :>
    <: [ $head, $tail ] =:>
      <: idtype($head) sort:ID-TYPE :>, <: $tail sort:ID-TYPE* :>
    <: [] =:>
  <: end :>
)

stms(
  <: match :>
    <: [ $head, $tail ] =:>
      <: stm($head) sort:STATEMENT :>;
      <: $tail sort:STATEMENT* :>
    <: [] =:>
  <: end :>
)

stm(
  <: match :>
    <: assignment( $id, $expr ) =:>
      <: $id :> := <: expr( $expr ) :>
    <: while( $expr, $stms ) =:>
      while <: expr($expr) :> do
        <: stms($stms) sort:STATEMENT* :>
      od
    <: if( $expr, $thenstms, $elsestms ) =:>
```

APPENDIX A

```

    if <: expr($expr) :>
      then <: stms($thenstms) sort:STATEMENT* :>
      else <: stms($elsestms) sort:STATEMENT* :>
    fi
  <: end :>
)

expr(
  <: match :>
    <: natcon( $natcon ) => <: $natcon sort:EXP :>
    <: id( $id ) => <: $id sort:EXP :>
    <: sub( $lhs, $rhs ) => <: $lhs sort:EXP :> - <: $rhs sort:EXP :>
    <: concat( $lhs, $rhs ) => <: $lhs sort:EXP :> || <: $rhs sort:EXP :>
    <: add( $lhs, $rhs ) => <: $lhs sort:EXP :> + <: $rhs sort:EXP :>
  <: end :>
)

idtype(
  <: match :>
    <: decl( $id, $type ) => <: $id :> : <: type($type) :>
  <: end :>
)

type(
  <: match :>
    <: natural => natural
    <: string => string
    <: nil-type => nil-type
  <: end :>
)
]

```

Appendix B

PicoJava

B.1 PicoJava.sdf

```
module PicoJava/syntax/PicoJava

imports basic/Whitespace

exports

context-free start-symbols Program

sorts Program Block BlockStmt ClassDecl VarDecl Stmt AssignStmt
      WhileStmt Identifier Use Exp BooleanLiteral IfStmt EmptyString
      Decl TypeDecl IdUse Dot Access String Id IntLiteral StringLiteral
      Operator BinOperator UnOperator BracketOp PlusOp MinOp TimesOp DivOp

context-free syntax

Block          -> Program      {cons("Program")}
"{" BlockStmt* "}" -> Block      {cons("Block")}
Stmt          -> BlockStmt    {cons("Skip_0")}
Decl         -> BlockStmt    {cons("Skip_1")}

TypeDecl     -> Decl         {cons("Skip_2")}
VarDecl     -> Decl         {cons("Skip_3")}
Type:Access Identifier ";" -> VarDecl    {cons("VarDecl")}

ClassDecl   -> TypeDecl     {cons("Skip_4")}
"class" Identifier ("extends" Superclass:IdUse)? Block
          -> ClassDecl {cons("ClassDecl")}

AssignStmt  -> Stmt         {cons("Skip_5")}
Variable:Access "=" Value:Exp ";" -> AssignStmt {cons("AssignStmt")}

WhileStmt   -> Stmt         {cons("Skip_6")}
"while" "(" Condition:Exp ")" Body:Stmt
          -> WhileStmt    {cons("WhileStmt")}
```

APPENDIX B

```

IfStmt          -> Stmt          {cons("Skip_7")}
"if" "(" Exp ")" "{" BlockStmt* "}" EmptyString*
                -> IfStmt          {cons("IfStmt")}
"if" "(" Exp ")" "{" BlockStmt* "}" "else" "{" BlockStmt* "}"
                -> IfStmt          {cons("IfStmt")}
""              -> EmptyString  {reject}

Access          -> Exp          {cons("Skip_9")}
IdUse           -> Access         {cons("Skip_10")}

Use             -> IdUse         {cons("Skip_11")}
Name:String     -> Use          {cons("Use")}

Dot             -> Access         {cons("Skip_12")}
ObjectReference:Access "." IdUse -> Dot          {cons("Dot")}

BooleanLiteral  -> Exp          {prefer, cons("BooleanLiteral")}
IntLiteral      -> Exp          {prefer, cons("IntLiteral")}
StringLiteral   -> Exp          {prefer, cons("StringLiteral")}

Id              -> Identifier   {cons("Skip_13")}
ActualName:String -> Id           {cons("Id")}

Operator        -> Exp          {cons("Skip_14")}
BinOperator     -> Operator    {cons("Skip_15")}
PlusOp          -> BinOperator  {cons("Skip_16")}
MinOp           -> BinOperator  {cons("Skip_17")}
TimesOp         -> BinOperator  {cons("Skip_18")}
DivOp           -> BinOperator  {cons("Skip_19")}
lhs:Exp "+" rhs:Exp -> PlusOp       {cons("PlusOp")}
lhs:Exp "-" rhs:Exp -> MinOp        {cons("MinOp")}
lhs:Exp "*" rhs:Exp -> TimesOp      {cons("TimesOp")}
lhs:Exp "/" rhs:Exp -> DivOp        {cons("DivOp")}

UnOperator      -> Operator    {cons("Skip_20")}
BracketOp       -> UnOperator   {cons("Skip_21")}
"(" Exp ")"     -> BracketOp   {cons("BracketOp")}

lexical syntax
[a-zA-Z] [a-zA-Z0-9]* -> String
"true"          -> BooleanLiteral
"false"         -> BooleanLiteral
[0-9]*          -> IntLiteral
["] [a-zA-Z0-9]* [\" -> StringLiteral

```

B.2 PicoJava.ast

```

Program ::= Block /PredefinedType:TypeDecl*/;
Block ::= BlockStmt*;

abstract BlockStmt;
abstract Stmt: BlockStmt;
abstract Decl: BlockStmt ::= Identifier;
abstract TypeDecl:Decl;

ClassDecl: TypeDecl ::= [Superclass:IdUse] Body:Block;
VarDecl: Decl ::= Type:Access Identifier;
AssignStmt: Stmt ::= Variable:Access Value:Exp;
WhileStmt: Stmt ::= Condition:Exp Body:Stmt;

abstract Exp;
abstract Access:Exp;
abstract IdUse: Access ::= <Name:String>;

Use: IdUse;
Dot:Access ::= ObjectReference:Access IdUse;
BooleanLiteral : Exp ::= <Value:String>;
IntLiteral : Exp ::= <Value:String>;
StringLiteral : Exp ::= <Value:String>;

// Added to make it easier to insert placeholders
abstract Identifier;
Id :Identifier ::= <ActualName:String>;

// Addition to the language: conditional
IfStmt : Stmt ::= Condition:Exp Then:BlockStmt* Else:BlockStmt*;

// Operations
abstract Operator : Exp;
abstract BinOperator : Operator ::= lhs:Exp rhs:Exp;
PlusOp : BinOperator;
MinOp : BinOperator;
TimesOp : BinOperator;
DivOp : BinOperator;
abstract UnOperator : Operator ::= Exp;
BracketOp : UnOperator;

// These nonterminals are used as classes in the
// semantic checker
PrimitiveDecl: TypeDecl;
UnknownDecl: TypeDecl;

// Rewrite rules are used in the semantic checker, Use will be
// rewritten to either of these depending on its location
TypeUse : IdUse;
VariableUse : IdUse;

```

B.3 Placeholders.ast

```

// Identifier
PhIdentifier      : Identifier ::= TreeQuery;
PhIfIdentifier    : Identifier ::= TreeQuery MatchPattern
                  Then:Identifier Else:Identifier;
PhMRIdentifier    : Identifier ::= TreeQuery Cases:PhMRIdentifierSort*;
PhMRIdentifierSort : Identifier ::= MatchPattern Identifier;

// Access
PhAccess         : Access      ::= TreeQuery;
PhIfAccess       : Access      ::= TreeQuery MatchPattern
                  Then:Access Else:Access;
PhMRAccess       : Access      ::= TreeQuery Cases:PhMRAccessSort*;
PhMRAccessSort  : Access      ::= MatchPattern Access;

// Exp
PhExp            : Exp         ::= TreeQuery;
PhIfExp         : Exp         ::= TreeQuery MatchPattern
                  Then:Exp Else:Exp;
PhMRExp         : Exp         ::= TreeQuery Cases:PhMRExpSort*;
PhMRExpSort     : Exp         ::= MatchPattern Exp;

// BlockStmt*
PhBlockStmt     : BlockStmt   ::= TreeQuery;
PhIfBlockStmt   : BlockStmt   ::= TreeQuery MatchPattern
                  Then:BlockStmt* Else:BlockStmt*;
PhForeachBlockStmt : BlockStmt ::= MatchVar TreeQuery BlockStmt*;
PhMRBlockStmt   : BlockStmt   ::= TreeQuery Cases:PhMRBlockStmtSort*;
PhMRBlockStmtSort : BlockStmt ::= MatchPattern BlockStmt*;

// Block
PhBlock         : Block       ::= TreeQuery;
PhIfBlock       : Block       ::= TreeQuery MatchPattern
                  Then:Block Else:Block;
PhMRBlock       : Block       ::= TreeQuery Cases:PhMRBlockSort*;
PhMRBlockSort  : Block       ::= MatchPattern Block;

// Stmt
PhStmt          : Stmt        ::= TreeQuery;
PhIfStmt        : Stmt        ::= TreeQuery MatchPattern
                  Then:Stmt Else:Stmt;
PhMRStmt        : Stmt        ::= TreeQuery Cases:PhMRStmtSort*;
PhMRStmtSort    : Stmt        ::= MatchPattern Stmt;

// IdUse
PhIdUse         : IdUse       ::= TreeQuery;
PhIfIdUse       : IdUse       ::= TreeQuery MatchPattern
                  Then:IdUse Else:IdUse;
PhMRIdUse       : IdUse       ::= TreeQuery Cases:PhMRIdUseSort*;
PhMRIdUseSort   : IdUse       ::= MatchPattern IdUse;

```

```
// for the rewriting on the Use
PhTypeUse      : IdUse      ::= TreeQuery;
PhIfTypeUse    : IdUse      ::= TreeQuery MatchPattern
                          Then:TypeUse Else:TypeUse;
PhMRTypeUse    : IdUse      ::= TreeQuery Cases:PhMRTypeUseSort*;
PhMRTypeUseSort : IdUse      ::= MatchPattern TypeUse;

PhVariableUse  : IdUse      ::= TreeQuery;
PhIfVariableUse : IdUse      ::= TreeQuery MatchPattern
                          Then:VariableUse Else:VariableUse;
PhMRVariableUse : IdUse      ::= TreeQuery Cases:PhMRVariableUseSort*;
PhMRVariableUseSort : IdUse      ::= MatchPattern VariableUse;
```

Appendix C

TML.ast

```
// TMLExpressions
abstract TMLExpression : MatchPattern;
plus : TMLExpression ::= lhs:TMLExpression rhs:TMLExpression;

abstract Elem : TMLExpression;

StrCon   : Elem ::= <con:String>;
IntCon   : Elem ::= <con:int>;
BoolCon  : Elem ::= <con:boolean>;
abstract TreeQuery : Elem;

MatchVar      : TreeQuery ::= <con:String> ;
Matchvarquery : TreeQuery ::= MatchVar Node*;
Root          : TreeQuery;
Absolute      : TreeQuery ::= Node*;
Relative      : TreeQuery ::= Node*;

Node ::= <id:String>;

abstract MatchPattern;

IntTerm      : MatchPattern ::= <con:String>;
RealTerm     : MatchPattern ::= <con:String>;
Function     : MatchPattern ::= fun:AFun;
Appl        : MatchPattern ::= fun:AFun args:MatchPattern*;
PlaceholderTerm : MatchPattern ::= type:MatchPattern;
ListTerm    : MatchPattern ::= elems:MatchPattern*;
Annotated   : MatchPattern ::= trm:MatchPattern Annotation;
blank       : MatchPattern;

abstract AFun;
Quoted     : AFun ::= <con:String>;
Unquoted   : AFun ::= <con:String>;

Annotation ::= annos:MatchPattern*;
```

Appendix D

Abstract Syntax Tree

```
1:0  AST.Program
2:0  AST.Block
3:2  AST.List
3:2  AST.PhIfBlockStmt
3:8  AST.Relative
3:8  AST.List
3:8  AST.Node
3:13 AST.MatchVar [$e]
4:4  AST.List
4:4  AST.VarDecl
4:4  AST.TypeUse [boolean] type: boolean
4:12 AST.Id [i]
5:4  AST.AssignStmt
5:4  AST.VariableUse [i] type: boolean
5:8  AST.PhMRExp type: $unknown
5:17 AST.Relative
5:17 AST.List
5:17 AST.Node
6:6  AST.List
6:6  AST.PhMRExpSort type: $unknown
6:9  AST.ListTerm
6:10 AST.List
6:10 AST.MatchVar [$e]
6:14 AST.MatchVar [$t]
6:23 AST.PlusOp type: $unknown
6:23 AST.PhExp type: $unknown
6:26 AST.MatchVar [$e]
6:34 AST.PhExp type: $unknown
6:37 AST.MatchVar [$t]
7:6  AST.PhMRExpSort type: int
7:9  AST.ListTerm
7:10 AST.List
7:17 AST.IntLiteral [0] type: int
0:0  AST.List
10:2 AST.AssignStmt
10:2 AST.VariableUse [i] type: boolean
10:6 AST.IntLiteral [0] type: int
```