Declarative Specification of Template-Based Textual Editors

Master's Thesis

Tobi Vollebregt

Declarative Specification of Template-Based Textual Editors

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Tobi Vollebregt
born in Zoetermeer, the Netherlands

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

Capgemini Nederland B.V.
Papendorpseweg 100
3528 BJ Utrecht, the Netherlands
www.nl.capgemini.com

Declarative Specification of Template-Based Textual Editors

Author: Tobi Vollebregt
Student id: 1184423
Email: T.J.Vollebregt@student.tudelft.nl

Abstract

Syntax discoverability has been a crucial advantage of structure editors for new users of a language. Despite this advantage, structure editors have not been widely adopted. Nevertheless, the Cheetah system, developed at Capgemini, leverages a structure editor to aid domain experts modeling tax-benefit rules in a domain specific language. The structure editor suffers from a lack of free form editing and conversions from/to plain text. The Spoofax language workbench, developed at Delft University of Technology, uses a textual editor, which is syntax-aware due to immediate parsing and analyses. In this thesis we describe a migration from Cheetah to Spoofax, which aims to bring the advantages of text editing to the tax-benefit rule modeling language.

During the migration, we experienced that current text-based language workbenches, such as Spoofax, require redundant specification of the ingredients for a template-based editor, which is detrimental to the quality of syntactic completion, as consistency and completeness of the definition cannot be guaranteed. We describe the design and implementation of a specification language for syntax definition based on templates. It unifies the specification of parser, pretty printer and template-based editor. We evaluate the template language by application to the tax-benefit rule modeling language and a language for mobile web applications.

Thesis Committee:

Chair:	Dr. Eelco Visser, Faculty EEMCS, TU Delft
University supervisor:	Dr. Lennart Kats, Faculty EEMCS, TU Delft
Company supervisor:	Dr. ir. Gert Veldhuijzen van Zanten, Capgemini Nederland B.V.
Committee Member:	Dr. M. Birna van Riemsdijk, Faculty EEMCS, TU Delft

Preface

About this thesis This work essentially consists of two parts. The first part (Chapter 2 up to Chapter 7) describes a migration of the Cheetah system at Capgemini to Spoofax, and provides the motivation for the second part. The second part (Chapter 8 up to Chapter 11) describes a language that unifies the specification of grammar, pretty printer, and templates for syntactic completion (content assist), and runtime support for the template-based editors generated from specifications in this language. The title of this thesis is derived from the second part.

Acknowledgements I am grateful to my supervisor Eelco Visser, company supervisor Gert Veldhuijzen van Zanten, and daily supervisor Lennart Kats, for the interesting discussions and the useful feedback on many drafts of various chapters of this thesis. I should also mention the work of Lennart and Eelco on the paper "Declarative Specification of Template-Based Textual Editors" for LDTA '12, which formed the basis for the current version of Chapter 9 and Chapter 10. Next, I would like to thank Betsy Pepels for her motivating talks at Capgemini, and Md. Adil Akhter for his support with SpoofaxLang. Last, but certainly not least, I am grateful to my parents, for their support during my time at the university.

Tobi Vollebregt
Delft, the Netherlands
March 25, 2012

Contents

List of Figures

Chapter 1

Introduction

1.1 Domain Specific Languages

By employing a domain specific language (DSL), a software engineer can raise the level of abstraction of their software. A well-designed DSL hides any accidental complexity in the code generator or interpreter of the language, so that the DSL user can focus on the complexity that is essential to the problem. Because a DSL can be used to hide irrelevant details, it is possible to enable domain experts to read (and sometimes even modify or write) programs in this language, thereby improving the communication between domain experts and software engineers. Additionally, a DSL can be used when the natural model of computation for a domain differs from what is available in general purpose languages. Specifically, many DSLs are declarative, instead of imperative, and often not Turing complete.

Introducing a DSL is an investment: tools need to be created that do not immediately add business value, but rather increase the rate at which business value can be generated in the future. While in the past a command line application to interpret, or generate code from domain specific programs may have been sufficient, nowadays programmers are used to Integrated Development Environments (IDEs) such as Eclipse and Visual Studio. As such, a new language that aims to increase productivity has a better chance to succeed if it comes with a development environment, preferably integrated with the development environment the team is already using.

1.2 Language Workbenches

Language workbenches [19, 34, 65] are tools that aim to drastically reduce the effort for the language developer to develop a new DSL and the accompanying development environment. Hence, a language workbench is an IDE for developing (domain specific) languages and IDEs for those languages. Language workbenches can be classified into those with language-aware structure editors (also known as projectional editors), and those with textual, parser-based editors.

Language-aware structure editors provide a template-based paradigm for editing programs, i.e. composing programs by selecting a template and filling in the placeholders, which can again be extended using templates. A crucial advantage of structure editors is syntax discoverability, helping new users to learn the language by presenting possible syntactic completions in a menu. Structure editors can be automatically generated from a syntax definition. Notable projects aiming at automatic generation of structure editors include MPS [65] and the Intentional Domain Workbench [51]. Structure editors can be used for general-purpose languages or for domain-specific languages (DSLs). A modern example of the former is the structure editor in MPS [65], for extensible languages based on Java. The Intentional Domain Workbench [51] is an example of the latter category.

Despite their good support for discoverability, structure editors have not been widely adopted. Pure structure editors tend to introduce an increased learning curve for basic editing operations. For example, they only support copy-pasting operations that maintain well-formedness of the tree and require small, yet non-trivial "refactoring" operations for editing existing code, e.g. when converting an `if` statement to an `if-else` statement. They also lack integration with other tools and expose the user to vendor lock-in. Transferring code across tools requires a shared representation that is generally not available. With software engineering tools such as issue trackers, forums, search, and version control being based on text, a textual representation is preferable, but requires the use of a parser and a parseable language syntax. This forces tools based on structure editors to find new solutions to problems long solved in the text domain.

Modern parser-based editors, such as those in Eclipse and Visual Studio, combine a plain text editor with language-aware editor services through parsers that run in the background while a program is edited. Over time, they have acquired features ranging from code folding to syntactic completions, allowing programmers to fill in textual templates, while seamlessly switching between template-based editing and text editing modes. The textual representation of the code makes interoperability with existing software engineering tools easy. Spoofax [34] and Xtext [17, 18] are modern examples of language workbenches that employ a textual, parser-based editor.

1.3 Cheetah

Cheetah is a language workbench with a language-aware structure editor, which has been developed by the *Functional Model Driven Development* group at the IT services company Capgemini. The group has developed several projects in Cheetah, among which a DSL for modeling tax-benefit rules, from which a set of .NET services can be generated. Cheetah's facilities for discoverability and the use of templates are particularly effective to aid a small audience of domain expert programmers manage the verbose syntax based on legal texts. The DSL hides the complexity of making decisions over a period of time, during which the inputs to the decision vary. Additionally, the calculation model allows decisions based on facts, as they were known to the system at a point in time in the past.

Despite successful application of Cheetah in multiple projects, Cheetah suffers from an underdeveloped structure editor, which does not have support for copy-pasting operations or "refactoring" operations at all. In fact, the programs can be converted to textual form only through code generators that output the program text in textual form, and programs can be modified only through a few primitive operations such as the introduction of a new node, removal of an existing node, or modification of the user defined text of a node. On top of the lack of features, the structure editor is rather slow and painful to work with, because it requires loading millions of Java objects into memory.

1.4 Spoofax

Spoofax [34] is a modern language workbench with a textual, parser-based editor. The Spoofax language workbench is an Eclipse plugin, and generates Eclipse plugins for the languages developed with it. Indeed, Spoofax reuses many of the capabilities of the Eclipse text editor, while shielding the language developer from the burden of implementing many Java interfaces for common editor services such as syntax coloring, code folding and outlining, which can easily be generalized to multiple languages. Spoofax employs scannerless generalized parsing [59], allowing languages to be freely composed and extended.

The first challenge we aim to address in this thesis is a migration of the tax-benefit rule modeling DSL of Capgemini from the Cheetah system to the parser-based Spoofax language workbench [34]. We use the migrated tax-benefit rule modeling DSL as a case study for our second research question. The first challenge leads us to the following research question:

Research Question 1: Can a DSL, such as the tax-benefit rule modeling DSL of Capgemini, be migrated from a structure editor based language workbench, such as the Cheetah system, to a parser-based language workbench, such as Spoofax?

1.5 Content Assist

We quickly realized that it would be impossible for a user to write new models in a language as verbose (see Figure 1.1) as the tax-benefit rule modeling DSL, without accurate and complete syntax discovery. In syntax-aware text editors this discovery is provided in the form of syntactic completion. Accurate and complete syntactic completion depends critically on two features of a language workbench:

- The syntactic completion proposals presented to the user must be relevant and complete.
- It must be feasible to create and maintain the specification necessary to make the editor aware of these completion proposals.

Current text-based language workbenches, including Spoofax, require redundant specification of the ingredients for a template-based editor, i.e. concrete syntax, abstract syntax, completion templates, and pretty-print rules, which is detrimental to the quality of syntactic completion in syntax-aware editors. Evolution of the language requires maintenance of all ingredients in order to maintain completeness and consistency. It is tedious and therefore easy to make mistakes while adding or adapting a completion template for each new or modified language construct.

```
Als gedurende minstens een dag in de periode van de eerste dag van het jaar van ... tot
    de eerste dag van het jaar volgend op ... geldt ( indicatie betrokkene aanwezig.Waar )
Dan
 Voer uit bepaal partner voor relevante gerelateerden bij burger
Anders
 "geen partnerbepaling (zie ontwerpbeslissing)"
Einde als
```

```
Het nieuwe beschikkingsnummer wordt samengesteld uit
(let op: bij numerieke deelwaarde posities van rechts naar links)
  beschikking.huishouden.aanvrager.burgerservicenummer van positie 1 t/m positie 9
  middeltype van positie 1 t/m positie 1
  beschikking.jaar van positie 1 t/m positie 2
  bepaal code type beschikking van positie 1 t/m positie 1
  beschikkingsvolgnummer van positie 1 t/m positie 3
  aanduiding regeling van positie 2 t/m positie 2
Einde declaratie
```

Figure 1.1: Tax-benefit rule modeling DSL examples

Thus, the second challenge we address in this thesis, is the design and development of a template-based syntax definition language[1] that unifies the specification of parsers, pretty printers, and template-based editors in order to support the efficient construction of template-based editing facilities in textual editors. This challenge leads to the second research question:

Research Question 2: Can a single, declarative language unify the specification of parser, pretty printer, and templates for syntactic completion?

A declarative specification of a template-based editor needs to be complemented with sufficient runtime support to be of any value. Hence, as the third challenge we address the runtime support for these template-based editing facilities in Spoofax by investigating an approach to compute the set of applicable templates at the location of the cursor. The third challenge leads to the third research question:

Research Question 3: Can we improve runtime support for template-based editors, so as to make the set of presented templates both relevant and complete?

1.6 Outline of this Thesis

To address Research Question 1, we first investigate the architecture of Cheetah in Chapter 2. Then, Spoofax is introduced, accompanied with a comparison to Cheetah, in Chapter 3. Since the persistent storage layer in Cheetah is based on XML, we start our exploration of a migration of the tax-benefits DSL to Spoofax with an extension to Spoofax to import a large body of XML in Chapter 4. Using this extension, we

[1] `http://strategoxt.org/Spoofax/TemplateLanguage`

transform the Cheetah meta model to an SDF grammar in Chapter 5, and we convert the Cheetah models to text files that adhere to this grammar in Chapter 6. In Chapter 7 we finish with a discussion on a migration of the code generators in Cheetah. An actual migration of these code generators is left as future work. To address Research Question 3, we describe changes to the Spoofax runtime that improve content assist in Chapter 8. As Spoofax is one of the text-based language workbenches that requires redundant specification of concrete syntax, completion templates, and pretty-print rules, we design and develop a template-based syntax definition language, which aims to unify the above artifacts, in Chapter 9. We validate this approach in Chapter 10 by applying it in two mature DSLs, thereby answering Research Question 2. Finally, we dedicate Chapter 11 to discussion of the pretty printing back-end of our template-based syntax definition language, as this aspect requires rather much space. We conclude this thesis with an overview of related work in Chapter 12, and a conclusion and a discussion of future work in Chapter 13.

Chapter 2

The Architecture of Cheetah

2.1 Overview

The Cheetah system is a meta modeling tool used at Capgemini. It enables the user to capture domain knowledge into a tree of *model elements*. From this knowledge base source code and other artifacts can be generated automatically using transformation rules written in Java. In practice, Capgemini leverages Cheetah, among other projects, to specify a DSL for modeling tax-benefit rules, and generating complete .NET solutions from programs in this DSL. Large parts of the tree with model elements are specified using primitive input elements like text boxes, check boxes, and lists. A designated part of the tree specifies a domain specific language, which is used in a structure editor to model domain concepts in a more expressive way.

2.2 Model Elements

The main constituent part of Cheetah is the model element. A model element is similar to a class in an object oriented language like Java, although prototypal inheritance is used instead of classical inheritance:

- It has a name.
- It may have any number of properties. Each property has a name, type and some boolean attributes. The type refers to another model element. The attributes include things such as *final* and *visible*. A final property cannot be modified in any instantiation or derivative of the model element. An invisible property is hidden from the user interface.
- It extends another model element (except for the inheritance hierarchy root).
- It may be abstract.

Additionally, a model element has meta data that is not typically present in a Java class:

- It specifies a table-based layout. This is used to organize the properties of the model element in a table in the user interface.
- It may specify a property definition. This is a reference to a model element that is to be used for property meta data for any property that has the first model element as its type. The property definition is present if and only if the model element is not used as a property definition.

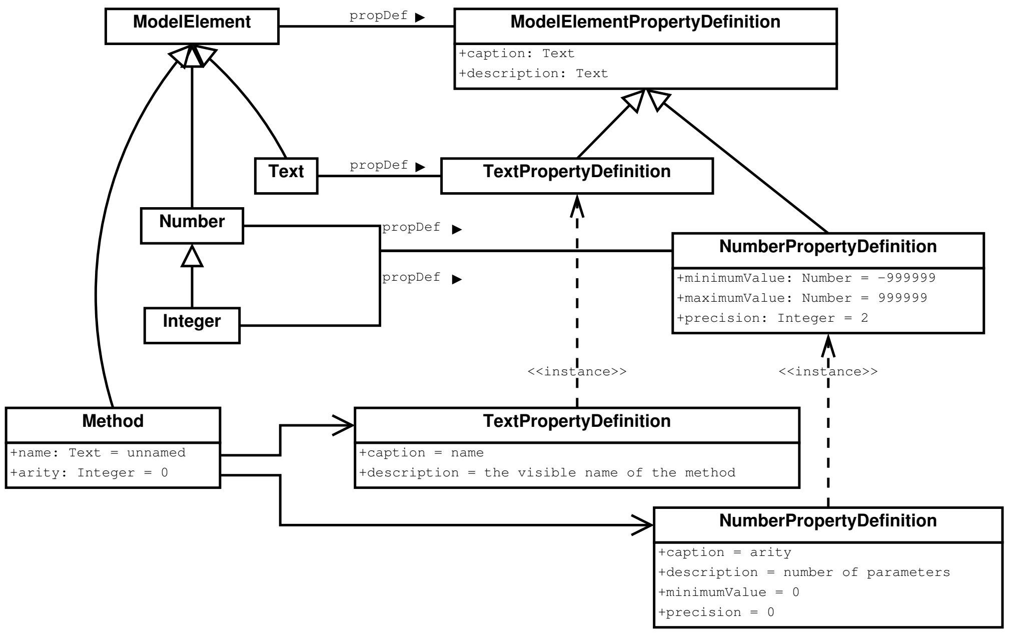
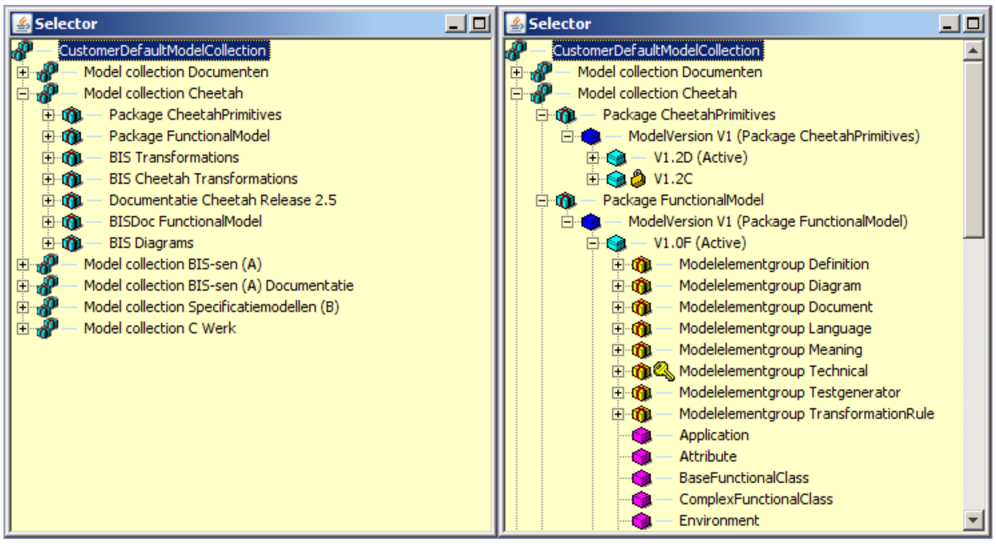


Figure 2.1: Property definitions and their use

For example, the model element at the root of the inheritance hierarchy, `Model-Element`, specifies the model element `ModelElementPropertyDefinition` as property definition. That model element contains the properties caption and description. Hence, all properties declared in the system have a caption and a description as meta data. In the same way the `NumberPropertyDefinition` model element extends all properties with the `Number` type with minimumValue, maximumValue and precision meta data. This example, including the `Method` model element that has two properties *name* and *arity*, is shown in Figure 2.1.

The link between model element and class is even stronger. In older versions of Cheetah each model element was stored as Java class. That is, saving a modification to a model element meant regenerating and compiling a Java source file. Loading a model element meant loading the Java class into the JVM. In a newer version each model element is stored in an XML file, optionally backed by a Java class for any custom functionality. The core package `cheetahPrimitives` leverages this custom functionality for features such as:

- Instantiating a GUI control of a type suitable to edit the model element. The `Boolean` model element instantiates a check box for example, while the `Text` model element instantiates a text box.
- Utility methods useful in custom Java source code in Cheetah, e.g. code generation.
- Querying the file system in, for example, the `ListOfFileSystem` model element.



Selector

- CustomerDefaultModelCollection
  - Model collection Documenten
  - Model collection Cheetah
    - Package CheetahPrimitives
    - Package FunctionalModel
    - BIS Transformations
    - BIS Cheetah Transformations
    - Documentatie Cheetah Release 2.5
    - BISDoc FunctionalModel
    - BIS Diagrams
  - Model collection BIS-sen (A)
  - Model collection BIS-sen (A) Documentatie
  - Model collection Specificatiemodellen (B)
  - Model collection C Werk

Selector

- CustomerDefaultModelCollection
  - Model collection Documenten
  - Model collection Cheetah
    - Package CheetahPrimitives
      - ModelVersion V1 (Package CheetahPrimitives)
        - V1.2D (Active)
        - V1.2C
    - Package FunctionalModel
      - ModelVersion V1 (Package FunctionalModel)
        - V1.0F (Active)
          - Modelelementgroup Definition
          - Modelelementgroup Diagram
          - Modelelementgroup Document
          - Modelelementgroup Language
          - Modelelementgroup Meaning
          - Modelelementgroup Technical
          - Modelelementgroup Testgenerator
          - Modelelementgroup TransformationRule
          - Application
          - Attribute
          - BaseFunctionalClass
          - ComplexFunctionalClass
          - Environment

Figure 2.2: Two instances of the selector

Start	:	ModelCollection
ModelCollection	:	List(ModelCollection) * List(Model)
Model	:	List(ModelMajorVersion)
ModelMajorVersion	:	List(ModelMinorVersion)
ModelMajorVersion	:	List(ModelElementGroup) * List(ModelElement)
ModelMinorVersion	:	List(ModelElementGroup) * List(ModelElement)
ModelElementGroup	:	List(ModelElementGroup) * List(ModelElement)

Figure 2.3: Organization of model elements

2.3 Organization of Model Elements

2.3.1 The Selector

The collection of model elements is large (about 20,000 different elements) so there must be a way to organize them. Cheetah allows the user to organize the model elements using a tree similar to the package explorer in Eclipse. In Cheetah it is called the *selector*; it is presented to the user after the application is started. Two screen captures of the selector are presented in Figure 2.2. The tree has the structure shown in Figure 2.3.

Model collections are used for a high level classification of models into different categories. In particular, language definition (meta model), language use (model) and code generation are separated into different subtrees using model collections. This separation is purely organizational, it is not enforced by the tool. *Models* are versioned containers of model elements. The versions are specified using *major and minor model versions*. For some models only major model versions are used. *Modelelement groups* are used similarly to model collections, except they help categorize model elements instead of models. Finally, model elements are the leaf nodes in the selector.

The tree is completely built from model elements. The model collection node is either a `ModelCollection` or a `ModelCollectionView` model element. The model node corresponds to a `Model` model element, the model version corresponds to a `ModelVersion` model element (major and minor are combined), and so on. Each of those model elements has (a) list(s) of references to the child model elements. Those lists contain a type-bound limiting it to include only model elements of a certain type. That is, the model collection only allows `ModelCollection`s in its first list and `Model`s in its second list.

2.3.2 On-disk organization of model elements

Besides the organization presented to the user through the *selector*, there is a natural organization of model elements in a directory tree on the file system. Although this organization is less visible to the user it is important because model elements are placed on a disk location relative to the Cheetah root directory, based on their fully qualified name. This naming scheme has the advantage that, given the fully qualified name of a model element, its file system location can be generated using a trivial procedure. For example, the model element with the name `cheetahPrimitives.v10.Boolean` has the file name `Boolean.cme`, and is found in the directory `cheetahPrimitives/v10`.

2.4 Prototypal Inheritance

We have seen many similarities between model elements and Java classes. At this point, however, the similarities end. When the user opens a model element, first the Java class for that model element is loaded. Then an instance is created of the model element. The user interface will be generated from that model element and any changes will be saved by the model element. Effectively the user edits a prototype instance of the model element.

When a model element is instantiated all its properties are instantiated too. More formally, for each property the model element that is the type of that property is instantiated. Since these model elements may have properties too, this procedure is recursive. The in-memory representation of the model element therefore contains the complete subtree of model elements and their properties below the model element. On persistent storage only the properties that are different from their origin model element are stored.

We recognize this as prototype-based programming, or prototypal inheritance. Specifically, the model elements as stored on persistent storage are the prototypes. Before a model element is instantiated all contained model elements are cloned from their prototype before the modifications declared in the edited model element are applied. Note that model elements are instantiated when they are edited, when a model element in which they are contained (either directly or indirectly) is edited or when code generation is run.

2.5 Primitive Data Types

As you may have noticed in the first part of this chapter, there are a number of basic model elements that correspond with simple user interface elements, such as:

- `Boolean`
- `Integer`
- `Number`
- `Enumeration`
- `Text`
- `LimitedText`
- `Color`
- `Date`

Then there are model elements that present the user with a list of model elements, references to model elements, or references to instantiated model elements in the subtree below the open model element. Among the model elements that present the user with a list are:

- `ListOfModelElements`
- `FilteredListOfModelElements`
- `NamedFilteredListOfModelElements`
- `OrderedNamedFilterdListOfModelElements`
- `ListOfFilteredReferenceToModelElement`
- `ListOfReferenceToNodeOfTree`

This works well for structuring simple data. As the data gets more complex it becomes tedious to map everything to these primitive data types.

2.6 An Example of a Business Rule Modeled Using Model Elements

As an example, we model the business rule “if price > 100 and customer.visit_count > 5 then discount = 10%” using the above data types. One way is to put the whole rule in a `Text` element and infer the structure at a later time (i.e., during code generation) using parser-based techniques. This is not the point of the whole mechanism of model elements however.

So can we do better by capturing the structure of the rule using model elements? We can, if we create model elements for the different parts of the rule, as shown in Figure 2.4. The figure should be read as “model element name { model element properties }”, where each property is a pair of its name and either an instance of a model element or a primitive value like a string (double quoted) or an integer.

2.7 Structure Editor

A representation of the tree of model elements using primitive user interface elements is certainly not ideal as it does not readily convey the meaning as expressed by the statement “if price > 100 and customer.visit_count > 5 then discount = 10%”.

```
IfThen {
  condition: LogicalAnd {
    lhs: GreaterThan {
      lhs: Variable { name: "price" }
      rhs: 100
    }
    rhs: GreaterThan {
      lhs: Variable { name: "customer.visit_count" }
      rhs: 5
    }
  }
  then: Assignment {
    lhs: Variable { name: "discount" }
    rhs: Percentage { value: 10 }
  }
}
```

Figure 2.4: Model elements for business rule example

```java
public interface TextObjectElementInterface {
  public String getText();
  public String setText(String aText);
  public boolean isTextEditable();
  public boolean isTabStopRequired();
  public Color getBackgroundColor();
  public SimpleAttributeSet getSimpleAttributeSet();
  public JFrame getFrame();
}
```

Figure 2.5: The `TextObjectElementInterface` in Cheetah

Cheetah contains a solution to this: there exists a `LanguageBlock` model element which employs a custom Java class `TextObjectEditor` as UI component. This class implements a rich edit control based on `JTextPane`[1] which may be used to edit a tree of model elements whose Java classes implement the `TextObjectElementInterface` interface displayed in Figure 2.5. Although the `TextObjectEditor` extends `JTextPane`, which is a free-form rich text editing component, `TextObjectEditor` restricts the text that can be entered. As can be seen in the interface definition, a model element can make itself read-only and can decide whether or not the user can cycle through the element using the tab key. Combined with content assist at insertion points, through which only certain model elements can be selected, the editor is transformed into a structure editor. Because it is based on a text editor, however, the projection of the tree of model elements can only contain text.

2.8 Language Definition

So, Cheetah provides a structure editor, which allows the user to edit model elements that implement a certain interface. That raises a question: how is the language defined?

[1]`http://download.oracle.com/javase/6/docs/api/javax/swing/JTextPane.html`

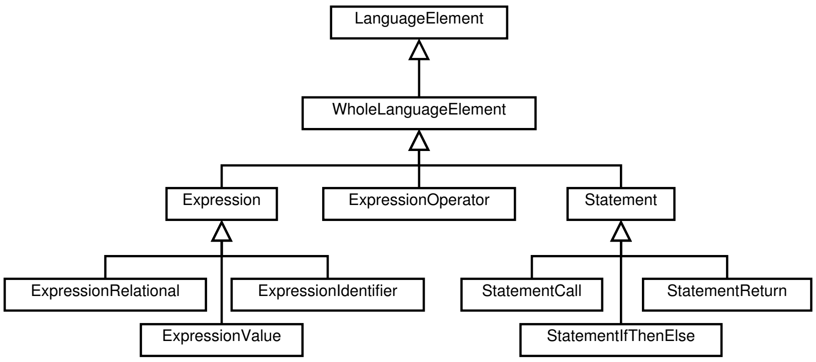


Figure 2.6: Some abstract language constructs

2.8.1 Abstract language constructs and language building blocks

The language in Cheetah is implemented using the package `functionalModel.v1.language`. This package contains model elements that aid in the definition of a DSL (the package itself is DSL independent). The `LanguageBlock` model element, which we have seen before, can be found here too. The majority of the model elements in this package inherit from the `LanguageElement` model element. The hierarchy of model elements splits in two parts right below this element. One subtree contains model elements, which can be extended by language constructs in the DSL meta model (Figure 2.6). We will refer to these as abstract language constructs. These language constructs are abstract because concrete syntax is not specified. Most of these model elements have an empty *elements* list property, that should be populated with concrete syntax elements in the DSL specification. The other subtree contains model elements which may be used to populate this list, i.e. to specify the syntax of the language constructs (Figure 2.7). `GLEText` is used for keywords; `GLETextUser` is free text (comments, the name of the variable in a declaration). SLE, ELE and GLE stand for *Statement Language Element*, *Expression Language Element* and *General Language Element*, respectively. We will refer to these as language elements.

The components of this package limit the DSLs that can be created in Cheetah: the structure of the DSL always needs to be mapped to statements, expressions and operators. An interesting design choice in Cheetah is the implementation of expressions. Contrary to an implementation using a tree, which is typical for parser-based tooling, Cheetah implements expressions as a list. Wherever an expression is expected there is an `ELEExpressions` language element. In the structure editor the user can enter a list of interleaved subexpressions and operators. Then only during code generation the list is parsed (taking into account the associativity and precedence of the operators) into a tree structure. This design alleviates some of the limitations of structure editors, and offers the user more freedom when editing expressions, by inferring the structure of the expressions only during code generation. For example, when changing an addition into a multiplication there is no need (for the user or the editor) to rearrange the tree to be consistent with the precedence and associativity of the new operator.

The `ExpressionOperator` that is involved here is only used for binary infix operators. Any other operator (including parentheses) is modeled as an `Expression` language construct, and therefore generates a tree-like structure. This is demonstrated with the examples in Figure 2.8.

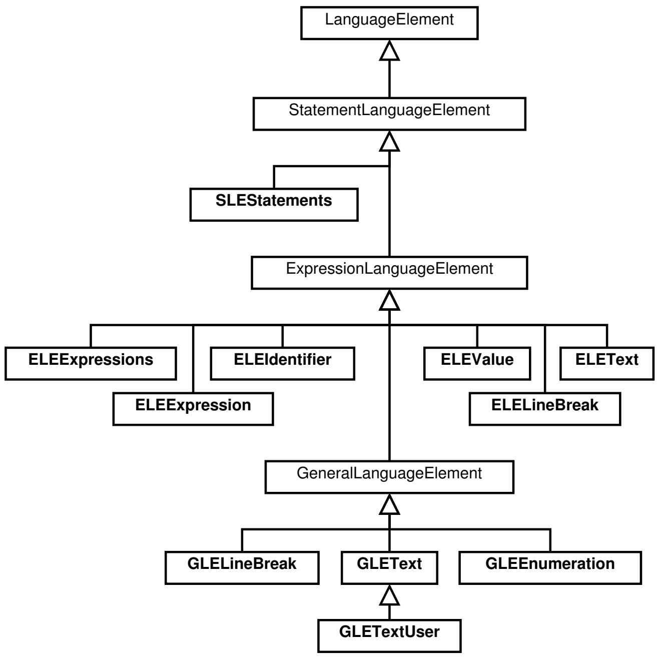


Figure 2.7: Some language elements

```
1 + 2 * 3 + 4:
  ELEExpressions([1, Plus, 2, Times, 3, Plus, 4])

(1 + 2) * (3 + 4):
  ELEExpressions([Parens([1, Plus, 2]), Times, Parens([3, Plus, 4])])

x present? and y present?:
  ELEExpressions([ValuePresent("x"), LogicalAnd, ValuePresent("y")])
```

Figure 2.8: Expressions and operators interleaved in a list

2.8.2 Concrete language definition

Using the `functionalModel.v1.language` package it is then possible to model a DSL. Concrete language constructs are created by extending one of the abstract language constructs, and populating it with language elements to define the projection. This is best illustrated by the example of an *if X then Y* statement in Figure 2.9. Because of the indirection employed by `StatementIfThenElse` through the *condition*, *whenTrue* and *whenFalse* properties *unless X then Y*, *X if Y* or *X unless Y* statements can also be created. Although it looks like the semantics of the language are specified by the abstract language construct that is extended, this is only partially true in Cheetah. Indeed, `StatementIfThenElse` and `StatementDoWhile` are used in execution path analysis for generating test cases. Most other semantic issues are not covered by the abstract language constructs, however. For those constructs the semantics are solely determined by the code generator and the semantics of the target language, while the language definition covers syntax only.

Given a language definition, the structure editor can be employed to edit any model

```
dsl.IfThenStatement extends fm.StatementIfThenElse {
  elements:
    1: GLELineBreak
    2: GLEText { text: "if " }
    3: ELEExpressionsFixedType { fixedType: "Boolean" }
    4: GLELineBreak
    5: GLEText { text: "then" }
    6: SLEStatements
    7: GLELineBreak
    8: GLEText { text: "end if" }
  condition: 3   // refers to item in list of elements
  whenTrue: 6
  whenFalse: n/a
}
```

Figure 2.9: Example of the definition of the projection of an *if X then Y* statement

element that extends `LanguageBlock`. In Cheetah, this applies to a property of the `BaseMethod` model element, which is extended by `Method` and some DSL-specific model elements. The fact that methods trigger loading of the structure editor implies that the method is the largest chunk of DSL code that can be edited in a single editor at any one time. Indeed, there is no concrete syntax present in Cheetah to describe methods (i.e., their signature) and higher level language elements. These elements are all handled using more primitive UI elements, such as lists and text boxes.

2.9 Artifact Generation

Cheetah employs Java code generation for the back-end storage of model elements, since each model element is a Java class / source file. The source file contains a number of *protected regions*, which can be used to customize the model element in ways that are not possible from within Cheetah.

The same mechanisms used to save model elements is used to perform generation of arbitrary artifacts. Cheetah also contains some model elements that capture knowledge about the transformations that are used to generate those artifacts. For a new transformation Cheetah generates a `*_CMETRN.java` file which contains a Java class and empty methods with the correct signature for the transformations. Each method contains a protected region so it is safe for a developer to implement those methods, even if the java is later regenerated. Nothing else is generated, so the body of the transformation rules is written in plain Java. The signatures of the methods are of the form:

```
SourceBlock transform(ModelelementAST input);
```

Which `transform` methods is called at a point in the tree is decided using a dynamic dispatch mechanism. The dynamic dispatcher decides, based on the type of the model element to transform, a `transform` method that fits. If there are multiple matches, it looks for the most specific match, and calls that method.

`SourceBlock` is an abstraction of a block of source code that may include protected regions. Cheetah substitutes `ModelelementAST` with the name of a model element plus an *AST* suffix. This name refers to a manually written interface, which contains methods to access the properties of the model element. Such an interface has been written for all abstract and concrete language constructs. When a method in the interface has to return a sub-model element, the method is declared to return the AST interface implemented by that model element. The goal of this is to decouple the transformations from the model element-based AST implementation. For the `IfThenStatement` example the interface might be like this:

```
public interface IfThenStatementAST {
  public ExpressionAST getCondition();
  public List<StatementsAST> getThenBlock();
}
```

2.10 Tax-Benefit Rule Modeling in Cheetah

The DSL we use as a case in the remainder of this thesis is a project developed with Cheetah for modeling tax-benefit rules. One requirement for this DSL was a *time traveling* feature, which was a main motivator for the use of model driven development techniques. Essentially, this feature requires the assignment of a *valid time* and *transaction time* to each fact recorded in the system. The valid time indicates the period during which the fact is true in the real world. The transaction time denotes the time at which the fact entered the system. Recording these data allows the system to answer questions based on facts as known to the system at any point in time in the past, and it allows the system to revisit decisions made in the past based on new facts. A database that stores facts in this way is called a *bitemporal database* [52]. At Capgemini, the database also stores the *reporting time*, making the database a tri-temporal database. The reporting time may be different from the transaction time: the reporting time is the time the fact was reported to the organization, whereas the transaction time is the time the fact was entered into the system. As the calculation model that backs the DSL for modeling tax-benefit rules makes decisions for a period of time, rather than a single point in time, even a simple if-then statement requires careful handling: in a single period, the branching condition may be both true and false, so that both the then-branch and the else-branch must be executed for different subperiods. The DSL is designed to hide these details as much as possible. From the DSL code, an application is generated, which consists of a collection of .NET services connected through a service bus, and which uses an SQL database for data storage. For our purposes the exact features of the application are irrelevant. The nature of the DSL is relevant, however. Unlike what a traditional synonym for DSLs — *little languages* [3, 58] — suggests, the DSL is a rather large language: it employs 317 unique keywords (some are reused in many constructs) in as much as 1036 language constructs[2], while Java 1.5, for example, uses only 53 unique keywords in 440 productions[3]. Furthermore, there are statements and expressions with up to 15 consecutive keywords, i.e. complete Dutch sentences. As such, many of the statements and expressions can be seen as natural language templates, with placeholders that should be filled by the domain expert with constants or other templates.

[2]Counted after the migration performed in Chapter 5

[3]Approximation based on the Java 1.5 syntax definition in java-front, found at: `https://svn.strategoxt.org/repos/StrategoXT/java-front/trunk/syntax/src/`

Chapter 3

Spoofax and a Comparison With Cheetah

3.1 Introduction

Spoofax is a parser-based language workbench implemented as a collection of Eclipse plugins. Spoofax is based on the original Spoofax editor: an Eclipse editor for the DSLs *Stratego* and *SDF* combined with a Java-based Stratego interpreter [28]. This original Spoofax editor has since been extended with an integrated Java implementation of *SGLR* [30], and Java has been added as a target language of the Stratego/XT compiler [33]. At this point, Spoofax came to depend on the *IDE Meta-tooling Platform* (IMP) [11], which provides an abstraction layer over the Eclipse APIs, and *Spoofax/IMP* was born. Spoofax/IMP has now superseded the original Spoofax editor completely. In practice, the name Spoofax is now used for the project, instead of Spoofax/IMP.

The goal of this chapter is to give an overview of the advantages and disadvantages of structure and parser-based editors, continuing with the way syntax is defined, code generation is performed, and editor services are declared in Spoofax, accompanied with a short comparison to Cheetah for each of those points.

3.2 Parser-Based vs. Structure Editors

While Cheetah uses a structure editor, Spoofax employs a text editor. Advantages of a structure editor are the capability to have multiple projections of the same underlying AST, some may even hide certain elements of the AST completely, and the fact that the concrete syntax can be changed at will without any model migration. Additionally, a structure editor may display certain (parts of) ASTs in a better suited format than plain text. Some examples are displaying a decision table or a mathematical formula, instead of a plain text approximation of a table or formula. For numerical data an alternate, read-only projection could graph the data.

The *Intentional Domain Workbench* [51] is a clear example of a language workbench that tries to exploit all of these features. Another state-of-the-art projectional language workbench is JetBrains *Meta Programming System* (MPS) [16], although it is still restricted to mostly textual projections.

A disadvantage of projectional editing is that much effort is required to support incorrect models. Incorrect models are often used by developers as an intermediate state between two correct models. Free-form cutting and pasting of parts of a model is also hard to accomplish in a structure editor: typically these features are restricted to complete subtrees of the AST, or copy-pasting is limited to the structure editor (i.e., no pasting from other tools). This last limitation could possibly be lifted by introducing parser technology, resulting in a hybrid editor, though we do not know of any tools employing this approach.

Parser-based editors naturally approach the last problem from the opposite direction: incorrect models and free-form cutting and pasting are principally possible, and (research) effort is focused on retrieving as much of the AST as possible from the potentially ill-formed textual input. Although parser-based editors cannot display any non-text content as projectional editors can, parser-based editing has a number of other advantages next to the intrinsic support for incorrect models. As the persisted representation of the AST is the concrete syntax, i.e. plain text files, many well-known development tools can be reused. Typical examples are version control systems, textual/regular-expression based search/replace, and the ability to easily exchange snippets of source code using e-mail and instant messaging protocols. Each of these examples needs special support in an editor which supports projectional editing only. The Intentional Domain Workbench, for example, includes a custom AST-node based version control system. The workaround for a developer is to exchange parts of the AST in the persistent representation used by the tool, e.g. XML.

Cheetah

Not many of the advantages of projectional editing are exploited in Cheetah. In Cheetah, there is at most one projection of a model. This projection is limited to styled text, as the structure editor has been built on top of a rich text editor. Cutting and pasting is not supported, nor is any other form of free-form editing: if a piece of code needs to be moved, it must be re-entered completely. Of the advantages mentioned above, the only other advantage — although it is barely used at Capgemini — is that the concrete syntax can be changed at will, without performing any model migration.

3.3 Language Definition

Spoofax uses SDF (Syntax Definition Formalism) [60, 54] for the specification of the syntax of a language. SDF is a fully declarative formalism: contrary to other BNF-based grammar specifications for both traditional (e.g., LLgen [21], YACC [26], ANTLR [46]), as well as generalized parser generators (e.g., Elkhound [38]), there is no target language embedding for semantic actions, or syntactic and semantic predicates [45]. The lack of specific-purpose target language embeddings in the grammar formalism eases reuse of a single grammar file in multiple applications.

Scannerless Generalized LR parsing [59] is the parser technique used in combination with SDF, and hence, Spoofax. *Generalized* parsing means the complete set of context-free languages can be parsed. This set is *closed under composition* [24], which means it is guaranteed that the combined language $L_1 + L_2$ is context-free, if the languages L_1 and L_2 are context-free. This property enables modularization, and thus

```
context-free syntax
  Exp "+" Exp -> Exp {cons("Plus"), left}
  Exp "*" Exp -> Exp {cons("Times"), left}
  "(" Exp ")" -> Exp {bracket}

context-free priorities
  Exp "*" Exp -> Exp >
  Exp "+" Exp -> Exp

context-free syntax
  ID       -> Type {cons("Type")}
  "String" -> Type {cons("StringType"), prefer}
```

Figure 3.1: Examples of SDF productions

helps with reuse of grammars. A side effect of generalized parsing is that ambiguous context-free languages are parseable: as such, ambiguities are allowed at parse table creation time, and the parser deals with them at run time, optionally returning multiple parse trees (a *parse forest*), instead of a single parse tree to the application. *Scannerless* parsing implies that the character stream is directly consumed by the parser, instead of first tokenizing it using a separate *lexical analyzer / scanner*. By not having a separate lexical analyzer, any problems with the composition of lexical analyzers can be avoided when composing or embedding languages. As such, because scannerless, generalized parsing is used, syntax definitions in SDF can be modularized as desired.

An example of SDF is shown in Figure 3.1. In this figure, we see three sections. The two **`context-free syntax`** sections contain productions (grammar rules) to parse arithmetic expressions containing only addition and multiplication, and to parse identifiers or the keyword `"String"` to the `Type` symbol. The **`context-free priorities`** section declaratively specifies that multiplication has a higher precedence than addition.

Curly brackets are used to assign attributes to each production. These attributes are used to specify operator associativity (e.g., **`left`** in Figure 3.1), and to prefer/avoid certain productions over other productions. For example, in Figure 3.1, it is preferred to parse "String" as a keyword, using the last production, and not as a generic identifier, using the before-last production. Attributes may also be used externally, i.e. by the application employing the parser: the most common attribute used in this way is the **`cons`** attribute, which specifies the name of the AST node that is created for an application of this rule by a so-called *imploder*, which converts the parse tree to an AST. Furthermore, Spoofax uses attributes to let the language engineer mark syntax as deprecated, and in a recent experiment, to automatically generate code to link usages of variables to their declarations.

Cheetah

In Cheetah, the only DSL is the language being defined: there is no separate DSL for syntax definition. Syntax is defined using generic (primitive) model elements, and their accompanying user interface elements. That is, to create a new statement, for example, the user needs to create a new model element that inherits from the `Statement` model element, and then populate its list of syntactic elements by repeatedly adding a single model element to this list (see also §2.8). This process is tedious, and must be repeated for each language construct. Cheetah does not offer a built-in projection that gives an overview of the syntax of the defined language, although transformations can be created to generate documentation that does provide this overview.

SDF is a step forward in terms of readability of the language definition because it is tailored towards syntax definition, as opposed to storage of arbitrary structured data. Because the tooling in Spoofax parses SDF to the terms that are used throughout Spoofax to represent ASTs, a syntax definition in SDF can be transformed as any AST can. A production in SDF can be annotated with meta data in custom attributes. In other words, even though Spoofax uses a separate DSL, it is possible to generate anything Cheetah can generate from the syntax definition.

3.4 Stratego

Spoofax employs Stratego for code generation and language specific editor service implementation. Stratego is a term rewriting DSL, which targets program transformation. It encourages *strategic programming* [37], which means that local rewrite rules are combined with generic traversal operators.

Stratego has originally been developed for the specification of optimizers for functional languages [64]. The theoretical basis for Stratego has been created in 1998 with the introduction of *System S* [63], which aimed to develop a calculus for rewriting languages, so as to solve the issues with ad-hoc implemented rewriting languages. Over the course of time Stratego has been used for many other purposes too, as is mentioned in the overview paper of *Stratego/XT 0.17* [8]:

> Stratego/XT has been used to build many types of transformation system, including compilers, interpreters, static analyzers, domain-specific optimizers, code generators, source code refactorers, documentation generators, and document transformers.

Stratego leverages *rewrite rules* to specify local transformations. A rewrite rule has the form

```
name : input -> output where condition
```

where `name` is the name of the rule, and `input` is a *pattern* that is matched against the current term. If there is a match, and the (optional) `condition` succeeds, then the current term is replaced by `output`. If not, then other rules with the same name are tried until one succeeds or all failed. A pattern such as `input` is a term optionally containing *wildcards* and *variables*. Wildcards match any subterm; variables do the same and bind this subterm to a name. By referring to this name the bound subterm can be used in the condition(s) for the rule, or placed at any position in the `output` term.

A collection of rewrite rules is an ideal way to express local transformations within an AST. Now the question is where in the AST, and in what order, those transformations should be applied. Stratego allows the developer to answer this question using generic traversal strategies. The standard library offers a rich set of traversal strategies. Some examples are: `innermost`, which repeatedly applies a rule from bottom to top until the rule fails on all nodes of the AST, `topdown` and `bottomup`, which apply a rule

```
rename-top = {|Renamed: alltd(rename) |}

rename:
  Scope(x) -> Scope(<rename-top> x)

rename:
  d@VarDef(x) -> VarDef(y)
  where
    y := x{<new>};
    rules(
      Renamed : x -> y
      VarDef  : y -> d
    )

rename:
  Var(x) -> Var(<Renamed> x)
```

Figure 3.2: Dynamic rules in action to bind variables to their declaration

once to each node of the AST in top-down resp. bottom-up manner, and `alltd`, which tries to apply a rule once to each node of the AST in a top-down manner, but does not traverse any subtree for which the rule succeeded.

Although generic traversals and rewrite rules offer a suitable paradigm for applying local transformations in a determined order to chosen parts of an AST, it is hard to express non-local transformations, as information may need to be passed from one rule application to arbitrary other rule applications. For this reason, Stratego provides *dynamic rules*, which are rewrite rules programmable at runtime. Dynamic rules are dynamically scoped using an explicit scoping construct, which implies that at any point of choice during an AST traversal a new scope may be entered. Therefore, the scope of dynamic rules can be matched exactly to the scope of the language constructs we are analyzing, which makes dynamic rules suitable for local variable renaming, as demonstrated by the `Renamed` dynamic rule in Figure 3.2. Furthermore, dynamic rules allow one to implement data flow analysis, totem propagation [29], and other analyses more easily [9], while they can also be used as a quick way to collect and transfer information from one part of a Spoofax editor to another (the `VarDef` dynamic rule in Figure 3.2).

3.5 Artifact Generation

Artifact generation in Spoofax is typically performed using Stratego. There are two main approaches to artifact generation in Spoofax. One approach is based on transforming the abstract syntax of the source language into the abstract syntax of the target language, and then pretty printing the resulting AST. This approach may be enhanced using *concrete syntax embedding* [62], which allows the developer of the transformation to write transformation rules using concrete syntax, as shown in Figure 3.3[1]. Concrete syntax embedding relies crucially on language composition, enabled by the scannerless generalized parsing technique used in Spoofax. An advantage of concrete

[1]Adapted from `http://strategoxt.org/Stratego/StrategoLanguage#Concrete_Syntax`

```
desugar : While(e, stm) -> If(e, DoWhile(stm, e))
```

(a) Transformation expressed using abstract syntax

```
desugar : |[ while (e) stm; ]| -> |[ if (e) do stm while(e); ]|
```

(b) The same transformation expressed using concrete syntax

Figure 3.3: A rewrite rule employing concrete syntax embedding

```
unparse : Procedure(name, args, body) -> $[
  void proc_[name] ([<separate-by(|",␣")> args]) {
    [body]
  }]
```

Figure 3.4: Example of indentation-safe string interpolation in Stratego

syntax embedding is that syntax errors are found earlier, and that target language syntax is highlighted in the editor. Concrete syntax embedding has a set-up cost, however: an SDF grammar for the target language must be available, and the quotation and anti-quotation operators must be defined. Additionally, syntax errors in the concrete syntax may prevent even the host language from being parsed, if recovery fails. The other approach is to concatenate strings together. Stratego recently acquired an *indentation-safe string interpolation syntax*[2], which helps to make transformations employing this approach more readable. An example is given in Figure 3.4. Although this way of specifying transformations has no associated set-up cost, there is no syntax highlighting for the target language, and syntax errors in the code that is being generated are found only after running the transformations.

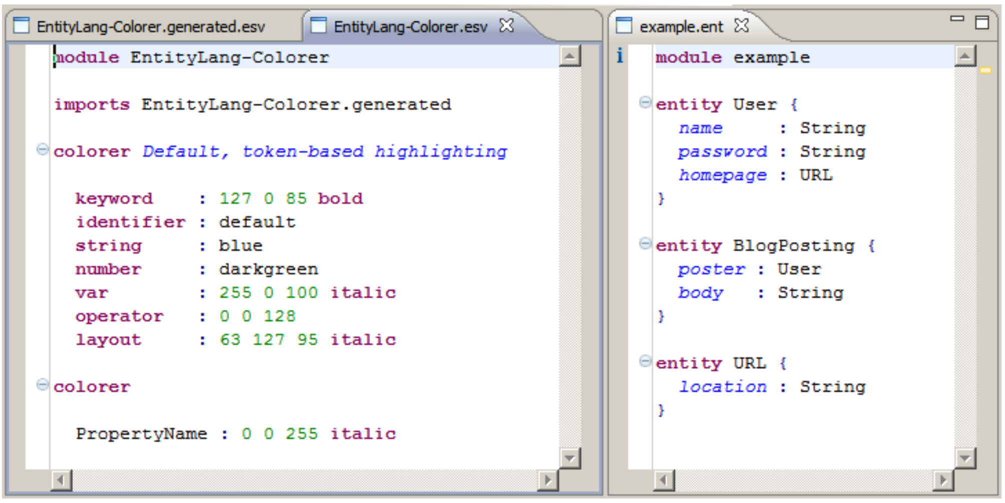
Cheetah

Cheetah uses transformations written in Java, supplemented with a library to perform code generation with protected regions, to generate artifacts (see also §2.9). The library offers abstractions for (pieces of) source files, and template based code generation. Because of the use of Java as host language, however, a multi line string template has to be represented by a concatenation of single line strings, a series of `StringBuilder.append()` calls, or a list of arguments to a method that takes a variable number of strings as argument.

3.6 Editor Services

Editor services in Spoofax can be classified in two categories: static, *syntactic* editor services, and dynamic, *semantic* editor services. Syntactic editor services, such as syntax highlighting (Figure 3.5), folding, the outline view (Figure 3.6), and completion templates, are described using the *ESV* editor service DSL [31]. Both constructors and symbols can be used to link concrete syntax to color, or mark the language element as eligible for outlining or folding.

[2] https://svn.strategoxt.org/repos/StrategoXT/strategoxt/trunk/news-archive/NEWS-0.18



```
module EntityLang-Colorer

imports EntityLang-Colorer.generated

colorer Default, token-based highlighting

  keyword    : 127 0 85 bold
  identifier : default
  string     : blue
  number     : darkgreen
  var        : 255 0 100 italic
  operator   : 0 0 128
  layout     : 63 127 95 italic

colorer

  PropertyName : 0 0 255 italic
```

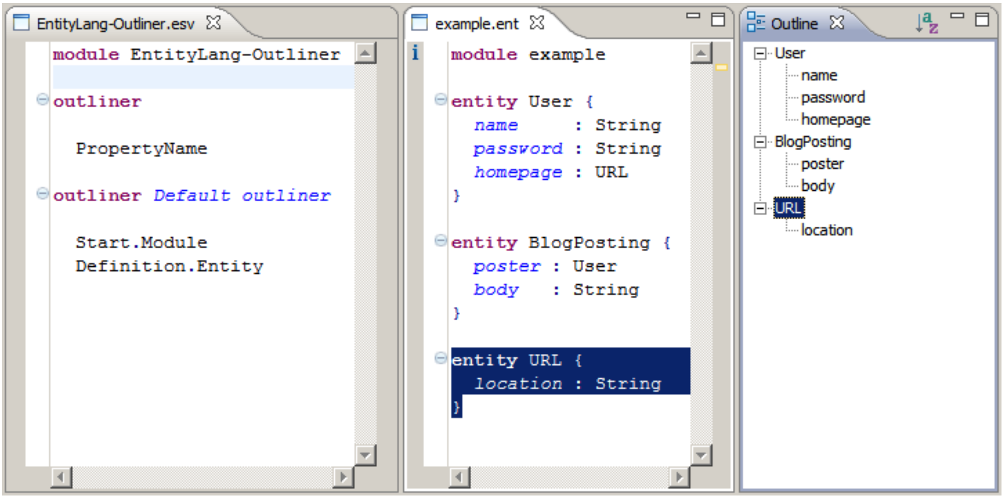
```
module example

entity User {
  name     : String
  password : String
  homepage : URL
}

entity BlogPosting {
  poster : User
  body   : String
}

entity URL {
  location : String
}
```

Figure 3.5: Syntax highlighting definition and use in Spoofax



```
module EntityLang-Outliner

outliner

  PropertyName

outliner Default outliner

  Start.Module
  Definition.Entity
```

```
module example

entity User {
  name     : String
  password : String
  homepage : URL
}

entity BlogPosting {
  poster : User
  body   : String
}

entity URL {
  location : String
}
```

```
User
  name
  password
  homepage
BlogPosting
  poster
  body
URL
  location
```

Figure 3.6: Outliner definition and outline view use in Spoofax

Within the ESV source of an editor in Spoofax there are also references to Stratego strategies that implement semantic editor services, such as error markers, semantic code completion and reference resolving. Generally, these strategies are invoked by the Spoofax runtime on certain user actions. They receive the AST or a subtree thereof, and should return a list of error/warning/info markers, completion proposals, or the AST node which declares an identifier. Additionally, any number of "free-form" editor services may be declared; these appear as items in a *Transform* menu, and can be used to test transformations on the current selection or file.

Cheetah

Many of the editor services present in Spoofax can be found in Cheetah too, and there are no editor services in Cheetah without an equivalent in Spoofax.

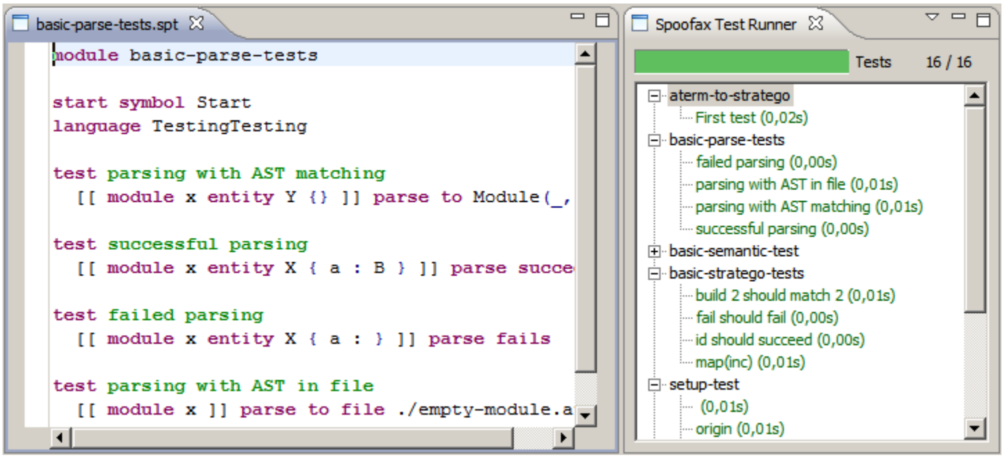
Syntax highlighting is present in the structure editor in Cheetah by means of a property of each language element that specifies font style and color. E.g., `GLEText` is set up to display in a blue color by default, so that keywords stand out. Because of the projectional nature of the editor in Cheetah, and the fact that a rich text editor has been used to implement it, Cheetah can use different fonts and font sizes for different language elements. Spoofax does not support this due to its use of the Eclipse code editor, which is limited to a single monospace font.

Code folding and an outline view are naturally absent in Cheetah as the largest piece of DSL code that can be edited at any one time is one method body. The list of method model elements serves as a good replacement of folding and an outline view.

Both syntactic and semantic content assist in Cheetah are well supported. For syntactic content assist, only statements and expressions allowed at the insertion position are displayed. This is guided by the syntax definition, which explicitly contains lists of statements and expressions allowed within other language elements (e.g., methods). Semantic content assist suggests only relevant local variables or object members, filtered on the expected type, if any.

There is limited error checking during editing. If errors are found, the involved language elements are highlighted using a red background. This is equivalent to the error marker and squiggle displayed in Eclipse/Spoofax. Error checking is implemented using custom Java code; the fact that only a limited amount of error conditions (certain type errors) are detected, suggests that error checking may benefit from an easier way to analyze a DSL program and detect errors.

Cheetah provides no reference resolving in the DSL editor. The only reference resolving that is present, is that, whenever a model element has to be selected in the basic model element GUI, a button is present to open the referred model element. In Spoofax reference resolving is implemented using a user supplied strategy that is invoked by the runtime whenever there is a need to resolve a reference, in combination with origin tracking of terms. That is, the user supplied strategy resolves the reference, and the Spoofax runtime figures out which editor to open, and where to put the cursor in that editor, using the origin information of the returned term.



```
module basic-parse-tests

start symbol Start
language TestingTesting

test parsing with AST matching
  [[ module x entity Y {} ]] parse to Module(_,

test successful parsing
  [[ module x entity X { a : B } ]] parse succe

test failed parsing
  [[ module x entity X { a : } ]] parse fails

test parsing with AST in file
  [[ module x ]] parse to file ./empty-module.a
```

```
Tests    16 / 16

aterm-to-stratego
    First test (0,02s)
basic-parse-tests
    failed parsing (0,00s)
    parsing with AST in file (0,01s)
    parsing with AST matching (0,01s)
    successful parsing (0,00s)
basic-semantic-test
basic-stratego-tests
    build 2 should match 2 (0,01s)
    fail should fail (0,00s)
    id should succeed (0,00s)
    map(inc) (0,01s)
setup-test
     (0,01s)
    origin (0,01s)
```

Figure 3.7: Testing language integrated in Spoofax

3.7 Testing

Spoofax provides a DSL for integration testing of editors developed with Spoofax [32] (Figure 3.7). This DSL allows both positive and negative parse tests, and tests of semantic editor services like content assist, reference resolving and any builders (i.e., custom transformations) defined for the editor. The fragments of tested DSL acquire certain editor services, like syntax coloring and content completion, themselves: the testing language redirects those editor services to the plugin under test. Tests in the current file are run whenever the user stops typing for a moment. Additionally, all tests in the project may be run in a graphical test viewer integrated with Eclipse, and Java code that interfaces directly with JUnit[3] and JSGLR can be generated.

Cheetah

Cheetah provides a testing component to aid in testing the models developed in it. This component analyzes a piece of model code, and generates a number of test cases based on the number of paths through the code. Regardless of the DSL implemented in Cheetah the number of code paths is known, because domain specific loops and conditions extend abstract model elements such as `StatementIfThenElse` or `StatementWhileLoop`. The component will also collect the input and output variables used in the code under test. Based on this analysis it will display a table of test cases with, for each test case, a list of the input and output variables. For each variable the user must then enter the input and/or expected output values, after which Cheetah can run the tests against the generated .NET services and add the actual output values. Rows will then be colored red or green depending on differences between expected output and actual output.

This way the testing component aids in performing integration tests of models and transformations to executable code. In Cheetah, there is no mechanism to test language definition, transformations of non-executable artifacts (e.g., documentation),

[3] http://www.junit.org/

or to test editor services, or anything else related to the user experience within Cheetah itself. Spoofax has the edge on this aspect because of its integrated testing language for testing of language syntax, editor services and builders (i.e., transformations).

3.8 Conclusion

Although it is hard to fit both Spoofax and Cheetah in a single feature comparison table, we summarized some of the differences and similarities described in this chapter in Figure 3.8. The most outstanding difference is the different editing paradigm the tools use: whereas Cheetah uses a structure editor, without support for cut & paste or free form editing, Spoofax uses a text editor, combined with SGLR parsing to convert the plain text to an AST. Besides the advantages of the text editor in Spoofax, the use of parser technology enables the persistent representation of any code to be plain text, which allows optimal use of existing text-based tooling, such as version control systems.

In Cheetah, a language is defined by creating a tree of model elements using primitive user interface elements such as list boxes and text boxes. Spoofax leverages SDF for the definition of a grammar of the language. We argue SDF is both easier to enter, and easier to understand, because the structure of the language can be seen at a glance.

The Cheetah system uses Java for code generation. Java has a number of disadvantages in this domain, such as the lack of multi line string literals and concise specification of tree rewrite rules. Spoofax includes Stratego, which is a DSL that focuses, among other things, on code generation. It allows multi line string literals for template-based code generation, as well as code generation based on AST rewriting and pretty printing.

Many editor services are available in both tools, although code folding and the outline view are not linked to the DSL editor in Cheetah. Instead, the DSL editor operates on a single method at a time, which is reached by navigating the tree of model elements. There is limited support for error markers in the Cheetah editor. There are no DSL-specific markers, which is probably due to a lack of support for writing static analyses and constraint checks. In Spoofax, error markers are implemented as rewrite rules in the Stratego DSL, which is arguably more suitable for such analyses than Java.

As the Cheetah system has been customized for the tax-benefit rule modeling DSL, it features a component for testing of DSL code. This component loads the generated .NET assemblies and invokes methods therein, based on test cases partially derived using path analysis of the DSL code, and partially entered by the user. Cheetah contains no tests of the integrated editor or any editor services. Spoofax, on the contrary, contains a testing DSL that can be used to test many editor services. With some work, this testing DSL can be linked to custom Stratego code to perform DSL code testing as Cheetah does.

Feature	**Cheetah**	**Spoofax**
License	In-house developed	LGPL
Editor	Structure editor	Text editor
Cut & paste	-	✓
Free form editing	-	✓
Parser technology	n/a	Scannerless Generalized
Syntax highlighting	✓ [∗]	✓ [∗]
Code folding	– [†]	✓
Outline view	– [†]	✓
Reference resolving	– [‡]	✓
Syntactic content assist	✓	✓
Semantic content assist	✓	✓
Error markers	✓ [§]	✓
Syntax definition	Tree of model elements	SDF
Runtime AST format	Tree of model elements	Terms
Persistent representation	XML & Java classes	Text files
Transformation language	Java	Stratego
Code generation	Template-based	Template-based or AST rewriting and unparsing
Multi line templates	-	✓
Use of multiple DSLs	-	✓
Language composition	-	✓
Testing of editor services	-	✓
Testing of DSL code	✓	✓ [‖]

∗ Only Cheetah allows font family and size to be customized per language element. Spoofax is limited to color, bold, and italics.

† Although the DSL editor in Cheetah does not support code folding or an outline view, the selector and other trees of model elements provide similar functionality.

‡ Not supported in the DSL editor; some of the primitive UI elements contain a button to jump to the referred model element.

§ Suboptimal: only a limited set of markers has been implemented.

‖ Not integrated, such tests are specific to the domain. Such tests can be integrated using the testing language, or custom Java strategies.

Figure 3.8: Feature comparison of Cheetah and Spoofax

Chapter 4

Parsing XML in Spoofax

4.1 Introduction

As Cheetah stores its model elements as XML files we need to know if, and how, Spoofax can read XML files before we can start on the actual migration. The collection of model elements in Cheetah consists of 1.3 gigabytes of XML spread over more than 20,000 XML files. To be able to rapidly develop a migration or an IDE which uses this data as input, it is important to be able to parse this XML within reasonable time and memory constraints. The XML files employed by Cheetah use a small subset of the features offered by the XML standard [10]. Namespaces, processing instructions, document type declarations, validation, CDATA and comments all remain unused. Cheetah uses only an XML declaration specifying the XML version (1.0) and the encoding (UTF-8), followed by elements, each containing only element content or only character data, i.e. mixed content elements are not present. Some of the character data elements contain only whitespace.

In this chapter we look at various existing representations of XML as ATerms in Stratego, and discuss their suitability for the conversion we want to do. We discuss the performance of JSGLR and implement a custom SAX-based XML parsing library for Stratego. We ensure both techniques for parsing XML will give us an identical, simplified AST, which is suitable for a transformation to an AST with a signature tailored towards Cheetah. This AST will be the input to our conversions of the Cheetah meta model (Chapter 5) and the Cheetah model (Chapter 6).

4.2 Parsing XML in Spoofax

There are three ways to load XML files into Stratego supplied with Stratego/XT [8]: *xml2aterm*, *parse-xml-doc*, and *parse-xml-info*. The three tools are part of the *xml-front* [7] package. They use SDF and SGLR to parse the XML using a shared XML grammar, and Stratego to transform the parsed AST into the desired output format.

xml2aterm (Figure 4.2) directly maps the XML structure to ATerms. That is, an element is considered an ATerm application. Its contents, both child elements and character data are converted to term and string arguments of the application, respectively. Strings of character data consisting of whitespace only are dropped by a call to *xml-doc-strip-whitespace* [7], which makes xml2aterm unsuitable for

```
<x> <y a="0" /> z </x>
```

Figure 4.1: XML fragment used as input for figures 4.2, 4.3, and 4.4

```
x(y(){("a", "0")}, " z ")
```

Figure 4.2: Sample output of xml2aterm for the XML fragment in Figure 4.1

```
Element(
  Name(None(), "x")
, []
, [ Element(
      Name(None(), "y")
    , [Attribute(Name(None(), "a"), "0")]
    , []
    )
  , Text(" z ")
  ]
)
```

Figure 4.3: Sample output of parse-xml-info for the XML fragment in Figure 4.1

our purpose. Even if we could fix this issue then it would still be suboptimal for the following three reasons:

Firstly, XML has a hedge structure [40], while ATerms have a tree structure. That is, ATerms are like function applications: they have a fixed arity (number of child nodes), although one name can be overloaded for different arities. On the other hand, for XML elements the exact number of child elements is rarely fixed: what is semantically a list child of an XML element, is in practice a list of children. Optional child elements in XML may not be present at all, while with ATerms the `Some`/`None` constructors are used to explicitly indicate presence/absence, similar to the `Maybe` datatype in Haskell [27]. In other words, an XML element has a variable size list of subtrees, while an ATerm has a fixed size list of subtrees. Because of this hedge structure, an expensive generic term deconstruction would be needed to match against the terms with a variable number of arguments.

Secondly, Stratego offers no pattern matching for items in unordered sets, such as the attributes of an XML element, so a linear search is needed every time an item must be retrieved from such as set.

Thirdly, XML attributes are converted to a list of (key, value)-tuple ATerm annotations. Annotations are treated specially in Stratego, which makes them less suitable for storing XML attributes. A transformation to store the attributes in some other way would be desired.

parse-xml-info (Figure 4.3) parses XML and then simplifies the AST using a number of transformations, one of which is — again — *xml-doc-strip-whitespace*. Hence parse-xml-info is not suitable for our purpose.

parse-xml-doc (Figure 4.4) is similar to parse-xml-info, but does not simplify the AST. This allows us to write a custom simplifying transformation which is better

```
Element(
  QName(None(), "x")
, []
, [ Text([Literal(" ")])
  , EmptyElement(
      QName(None(), "y")
    , [Attribute(QName(None(), "a"), DoubleQuoted([Literal("0")]))]
    )
  , Text([Literal(" z ")])
  ]
, QName(None(), "x")
)
```

Figure 4.4: Sample output of parse-xml-doc for the XML fragment in Figure 4.1

suited to the problem at hand. It includes all character data so it is suitable for our purpose.

We decided to start the conversion of the meta model (Chapter 5) and model (Chapter 6) defined in Cheetah, by importing the XML grammar into a Spoofax project for an AST with the parse-xml-doc structure. Then we copied some of the transformations used in parse-xml-info and added some of our own to replace namespace-qualified names with strings. This enabled us to open XML files in Spoofax and rapidly develop and test transformation strategies on single Cheetah XML files.

4.3 Performance of JSGLR for Parsing XML

Although JSGLR is sufficient to work with a single small XML file in the IDE, the performance on large XML files proved to be insufficient for the conversion as a whole. Parsing XML files larger than approximately 2.5 MB was not possible with JSGLR due to memory constraints (heap size 1024 MB) at the time the first conversion was implemented. Parsing files close to this size took more than a minute[1].

Halfway during this thesis project a new version of JSGLR has been deployed. This new version of JSGLR features a mechanism to build an AST directly, instead of needing to build a parse tree first. Contrary to the old JSGLR, the new build can parse the largest XML files (5 MB) found in Cheetah within one to two minutes (heap size 1280 MB and 1024 MB, respectively). It is impossible to parse those files with a heap size lower than 1024 MB or a stack size lower than 1 MB (eight times the default stack size). Certain small XML files (kilobytes) could still not be parsed by JSGLR, because these hit an ambiguity in the concatenation of character data and entities in the XML grammar. This ambiguity was never visible in the C tooling because the transformation from parse tree to AST suppressed it. We removed this ambiguity from the XML grammar so that JSGLR is able to parse all XML files. This change does not significantly influence the time JSGLR takes for the large XML files mentioned before, because entities are used scarcely in the project, except in a few small XML files.

[1]On an Intel(R) Core(TM)2 Duo P8600 running at 2.40 GHz

Using the new version of JSGLR and the patched XML grammar we measured how long it takes to parse the complete 1.3 GB set of XML files (21,094,379 lines with, on average, 66 characters). JSGLR does this in 155 minutes (140 KB/s). This result can be explained as follows. JSGLR is essentially a parse table *interpreter*, while a dedicated XML parser could use a parser generator (i.e., parse table *compiler*) or contain a hand-optimized parser. In either case there will be less overhead, ceteris paribus. Because JSGLR employs scannerless parsing, the size of the input to the parser will be larger than when a separate scanner reduces the input from a stream of characters to a stream of tokens beforehand. Such a scanner may employ a less expressive formalism (e.g., a regular language), which allows a faster reduction to lexical symbols than possible using SGLR parsing. Another cause may be local ambiguities that get resolved later on during a parse. For each such ambiguity there are effectively multiple parsers active as long as the ambiguity exists. An XML parser that employs a parsing algorithm that rejects ambiguous grammars at parse table construction time will never have to deal with such ambiguities at run time. Additionally, the XML grammar used with JSGLR is the grammar used by xml2aterm, parse-xml-info and parse-xml-doc. Because the last tool returns an AST that includes whitespace, the grammar is designed to parse whitespace, and include it in the AST. Possibly the JSGLR parser could be faster in our case using either a grammar tailored to the parsing of XML without significant layout, or a grammar tailored specifically to the XML schema used by Cheetah.

4.4 Parsing XML Using SAX

Before the new version of JSGLR was stable we desired to parse the largest XML files. A complete parse of the Cheetah XML files using JSGLR would take at least 155 minutes, which is too long for a rapid development/feedback cycle when developing a conversion from Cheetah to Spoofax. For these two reasons we decided to hook a *SAX-based* dedicated XML parser [39] into Stratego. Simple API for XML (SAX) is an event-driven API for parsing XML. The necessary glue code to use SAX from Stratego has been created as a Spoofax library, i.e. Java classes implementing *primitives* that can be invoked from Stratego using the `prim` construct.

After an initial Cheetah-specific prototype we have generalized the library to be useful to other projects as well. To accomplish this task, we ensured our SAX *ContentHandler*, which is responsible for building terms from the XML document events, can handle both namespace-aware and namespace-unaware parsers. The structure of the terms in both cases is shown in Figure 4.5. We also removed any Cheetah-specific shortcuts in the handling of character data.

SAX has a mechanism of *features* [39] to allow the application to choose whether a parser should be namespace aware, whether it should validate, etc. A feature is a tuple of an identifying URL and a boolean which specifies whether it is enabled or disabled. SAX contains methods to query whether a feature is enabled, and to enable/disable a feature. We expose this method to Stratego as another primitive, which may then be used by application code (i.e., our conversion) to toggle, for example, namespace-awareness of the parser. Our library adds two Spoofax-specific features:

```
Element   : Name * List(Attribute) * List(Element) -> Element
Attribute : Name * String -> Attribute
Text      : String -> Element
Name      : Option(String) * String
```

(a) Namespace-aware parser

```
Element   : String * List(Attribute) * List(Element) -> Element
Attribute : String * String -> Attribute
Text      : String -> Element
```

(b) Namespace-unaware parser

Figure 4.5: Structure of the terms created by StrategoTermBuilder

http://spoofax.org/sax/features/character-data[2] This feature is enabled by default. When enabled, XML character data (`Text` terms) is returned to the application.

http://spoofax.org/sax/features/mixed-content This feature is disabled by default. Only when enabled, character data in elements that also contain child elements is returned to the application. That is, it specifies whether character data in mixed content elements should be returned or not. Contrary to xml2aterm and parse-xml-info, our library never filters whitespace-only `Text` terms. Instead, the application can choose to disregard all character data in mixed content elements. For many XML documents this is sufficient, and it saves a lot of unnecessary processing of (whitespace) character data.

4.5 Performance of our SAX-Based XML Parser

We measured how long it takes our SAX-based XML parser library to parse the complete 1.3 GB set of XML files. On our system, the XML parser used by SAX was Apache Xerces-J 2.6.2[3], which was included with our Java Virtual Machine. It parses all files in 220 seconds (6 MB/s). Observation of the CPU usage revealed it was disk bound, which makes it at least forty times faster than JSGLR.

4.6 XML Canonicalization and Simplification

At this point we have two ways of parsing the XML files: we can use our SAX-based XML parser library for fast batch processing of many XML files, or we can use JSGLR with the XML grammar supplied with Stratego, for slow parsing, but better IDE support. Because we want to run transformations on either input (JSGLR for its support in the IDE and SAX for batch-parsing), we canonicalize the XML of either source into one common format, which is the input to all subsequent transformations. We perform a combined canonicalization and simplification step that employs the following transformations:

[2]It is common practice to name SAX extensions using a URI, *including* the `http://` protocol specifier. Requesting these URIs in the browser, however, will generally not return anything meaningful.

[3]`http://xerces.apache.org/`

```
xml-to-cme:
  Element("elem", attrs, childs) -> elem(name, cls, childs)
  with
    name := <get-opt-attr(|"name")> attrs;
    cls  := <get-opt-attr(|"cls")> attrs
```

Figure 4.6: Transformation from XML element to constructor application

- Removal of the outer `Document` node. We do not need this node.
- Removal of the separate closing tag in each `Element` node. In well-formed XML the closing tag is always identical to the opening tag, so it is redundant.
- Substitution of an empty `Element` for the special purpose `EmptyElement`. We do not need to distinguish between `<x></x>` and `<x/>`.
- Replacing qualified names (`QName`) with strings. Namespaces are not used in the Cheetah XML files.
- Entity resolution (e.g., $\&amp; \Rightarrow \&$) and concatenation of the constituent parts (literal text, entities) of text content into one string.
- Moving the inner text of *value*-elements to an attribute of their parent *text*-element. Both styles are used; there is no semantic difference.
- Removal of `Comment` nodes and any remaining `Text` nodes. Only `Text` nodes in *value*-elements are significant, and those have been converted to attributes in the previous step.
- Removal of any elements with a *deleted* or *deprecated* attribute with a true value. Elements with this attribute are not visible in Cheetah. Their name is kept alive (content is erased) through an element with one of these attributes, so that model elements opened after the act of deletion can find out what happened to the element.

4.7 Transforming XML to a Cheetah-Specific AST

Before any further transformation we transform the XML AST to the elements which are encoded by the XML. That is, each XML element *foo* is transformed to a constructor `foo`. Certain attributes are selected and stored as arguments of the constructor application. We determined whether an attribute should be included by manual inspection of the XML files and inferred XML schemas. This transformation is useful because, when pretty printed, the resulting ATerms are a lot less verbose, and hence, easier to inspect, than the original XML. The transformations also remove any dependency on the ordering and presence of XML attributes, which is desired because Stratego lacks pattern matching for unordered lists. The transformations for this step are performed using rewrite rules as displayed in Figure 4.6 for the transformation of the *elem* XML element to an ATerm. An example of the structure of a term that could be the output of this transformation is shown in Figure 4.7. This example has been simplified by removing repeated elements, and many elements that are not relevant to the example.

```
root(
  "mM_Basis.v1.basiselementen.taal.handelingen.AlsDanAnders"
, Some("functionalModel.v1.language.StatementIfThenElse")
, ...
, [ list(
      "elements"
    , [ elem("I1",  Some("GLELineBreak"),  [])
      , elem("I5",  Some("GLEText"),       [text("text", "Als ")])
      , elem("I11"
        , Some("ELEExpressionsFixedType")
        , [ elem("fixedType", None()
            , [ text("modelelementName", "Boolean")]
            )
          ]
        )
      , elem("I2",  Some("GLELineBreak"),  [])
      , elem("I8",  Some("GLEText"),       [text("text", "Dan ")])
      , elem("I9",  Some("SLEStatements"), [])
      , elem("I3",  Some("GLELineBreak"),  [])
      , elem("I6",  Some("GLEText"),       [text("text", "Anders ")])
      , elem("I10", Some("SLEStatements"), [])
      , elem("I4",  Some("GLELineBreak"),  [])
      , elem("I7",  Some("GLEText"),       [text("text", "Einde als")])
      ]
    )
  , ...
  ]
)
```

Figure 4.7: Output of transformation / input to the SDF generator

4.8 Conclusion

JSGLR is sufficient to enable a Spoofax editor for XML, which eases development of the conversion of the Cheetah XML files to Spoofax. JSGLR is, however, not efficient enough for a rapid development/feedback cycle on the complete conversion. Therefore an XML parsing library for Spoofax has been developed. This library supports parsing XML from strings and files and allows application code to set which XML features should be used. As shown in Figure 4.8, it is two orders of magnitude faster than JSGLR, because a dedicated SAX-based XML parser is used. Hence, this library enabled us efficiently develop the conversion of the Cheetah XML files to Spoofax. To test our transformations in the IDE and to run them in batches, we developed canonicalizing and simplifying transformations that transform either form of XML AST to a common form, which is further transformed to the Cheetah-specific AST of Figure 4.7, which is the input to the transformations in Chapter 6 and Chapter 5.

XML parser	**Performance & Notes**
Single file	
Old JSGLR implementation	Unable to parse files $\geq$ 2.5 MB*
New JSGLR implementation	1-2 minutes for one 5 MB* file†
Complete set of 1.3 GB / 21,094,379 lines of XML	
New JSGLR, disambiguated XML grammar	155 minutes
SAX-based XML parser	220 seconds‡

$*$ Here, 2.5 MB is equivalent to approx. 40,000 lines of XML

† Unable to parse XML files with entities due to ambiguous grammar

‡ Disk bound, instead of CPU bound

Figure 4.8: XML parsing performance

Chapter 5

Migrating Cheetah Meta Models to Syntax Definitions in Spoofax

5.1 Introduction

In this chapter we will migrate the Cheetah-specific AST obtained via the XML parsers developed in Chapter 4, to a grammar in SDF. With such a grammar available we can produce a minimal Spoofax-based Eclipse plugin for the tax-benefit rule modeling language.

There are two ways to perform such a migration. The first is a manual migration: we transform each language construct by manually inspecting its definition in Cheetah, and typing out the corresponding SDF. The second approach is an automatic migration: we develop a transformation from the Cheetah-specific AST to a grammar in SDF. We choose this second approach, because it makes it easy to experiment with different translations of various aspects of the language, and because the code for the automatic migration may be reused after a change to the syntax of the language, or even for a different language created in the Cheetah system. Additionally, an automatic migration reduces the chance of small errors and inconsistencies slipping in. The structure of our automatic migration is as follows: we transform the tree represented by the XML to an appropriate form, dropping any information that is not relevant to the grammar of the language, and convert the remaining tree to SDF concrete syntax.

5.2 Design Decisions

The language definition in Cheetah consists of a model element for each language construct. Each language construct contains an *elements* property with language elements, which encode the projection of model elements to text in the projectional editor in Cheetah. Before generating SDF some key design decisions have to be made.

Each method model element has a *statements* property, which specifies the statements that are allowed in the method. This includes statements nested at any level in the method, i.e. within a branch or a loop. If we enforce this in the grammar, then we need to generate multiple copies of all productions: one for each pair of a method type and a statement language construct that can directly or indirectly include other statements. For example, for two different method types `Declaration` and

`Procedure`, there may need to be a `DeclarationLoopStatement` and a `Procedure-LoopStatement`; the first allows only `DeclarationStatements` in the loop body, while the second only allows `ProcedureStatements`. Since each statement that contains a list of statements as child (e.g., control flow statements such as conditionals and loops) must be instanced for every method type, this design results in an explosion of the amount of productions, which, in turn, makes the generated grammar hard to maintain. Therefore we decided to allow all statements in all method types. A restriction can be implemented as part of the semantic analysis of the AST.

A similar reasoning applies to expressions. In Cheetah each concept (i.e., class, from the perspective of the DSL) specifies the expressions that can return this concept. It is theoretically possible to generate an expression production for each pair of a concept and an expression language construct. However, again this would result in an unmaintainable amount of productions, so we decided to stay with one expression symbol.

There is one exception: the abstract language construct `ExpressionRepeating` and the corresponding language element `ELEExpressionsRepeating`. The latter always refers to the first, i.e. where an `ELEExpressionsRepeating` is present in the meta model, the model contains a list of the referred concrete language construct that extends `ExpressionRepeating`. This matches exactly with the introduction of a special symbol for each matching pair of `ExpressionRepeating` and `ELEExpressions-Repeating`, so that is what we do.

A last consideration relates to the fact that in Cheetah, arithmetic expressions are lists of interleaved subexpressions and operators (see §2.8.1). In a parser-based environment it is advantageous to parse expressions as trees, because it results in an AST that correctly encodes the meaning of the expression, and it saves an extra parsing step during code generation. Besides, it is what is expected when a developer encounters parser-based tooling, thus maintainability is increased. Therefore we decided to generate a syntax definition that uses trees for arithmetic expressions.

5.3 Implementation

To generate a production these things need to be known:

Target symbol Since each language in Cheetah must be built on expressions and statements we can select a name for the target symbol by moving up the inheritance hierarchy until we visit such an abstract language construct. E.g., a language construct that (indirectly) extends `Expression` will reduce to the `Expression` symbol.

Symbols The symbols of the production are determined by the elements that encode the projection in Cheetah. Most of those language element have a clear mapping to SDF. E.g., `GLEText` maps to a keyword, `SLEStatements` maps to `Statement*`.

Constructor The constructor name should uniquely identify the production to be able to identify the language construct in the AST. Since the name of the language construct is unique this is a perfect fit.

With this knowledge it is conceptually straightforward to generate grammar productions from the language definition in Cheetah. The implementation is hairier be-

```
module mM_Basis/v1/basiselementen/taal/handelingen/AlsDanAnders

context-free syntax
  "\n" "Als" Expression
  "\n" "Dan" Statement*
  "\n" "Anders" Statement*
  "\n" "Einde" "als"       -> Statement {cons("AlsDanAnders")}

lexical restrictions
  "Als" "Dan" "Anders" "Einde" "als" -/- [A-Za-z0-9\_]
```

Figure 5.1: Output of the SDF generator, given Figure 4.7 as input

cause we need to tokenize `GLEText` elements to be able to generate a single literal for each keyword. To do this we developed a string tokenizer that splits strings *between* characters, instead of on characters, because this is not available in the Stratego standard library. Splitting between characters is necessary to correctly handle punctuation characters, as in “Aanmaken (alleen id) indien”: the generated SDF should allow layout as indicated: “Aanmaken␣(␣alleen␣id␣)␣indien”.

We ensure that, in the previous example, “alleen” and “id” are always separated by at least one space, by generating lexical restrictions that require each keyword to be followed by a non-alphanumeric character. We do, however, not *reserve* keywords by default: in other contexts “Aanmaken”, “alleen”, “id” and “indien” may still be used as identifiers. We cannot use reserved keywords because in the models in Cheetah variables often have a name that is also a keyword.

The language element `GLEEnumeration` provides an additional challenge. This element encodes a choice of one out of a list of literals, and is used often in Cheetah. It is used, for example, for the semantically irrelevant Dutch articles “de” and “het”, but also for semantically relevant literals like “<”, “>”, “<=” and “>=” in a comparison. (A comparison is not an operator in Cheetah.) For each `GLEEnumeration` we introduce a symbol `Enum-X`, where X is an integer that is not used for any other `GLEEnumeration`. We then use `Enum-X` at the place of the `GLEEnumeration`, and we generate an injection from each of the literals into `Enum-X`. Because the enumeration may have semantics we add a constructor annotation to each injection. The constructor name is generated from the literal itself, by removing punctuation, and converting the remaining words to *UpperCamelCase*. We included special cases such as “<” ⇒ “LowerThan”, or else the generated constructor name would be the empty string. An example of a matching input and output of the SDF generator are given in Figure 4.7 and Figure 5.1, respectively.

5.4 Issue: Optional and Mandatory Line Endings

In Cheetah, line breaks are integrated into the concrete syntax definition by means of special constructs to allow the user to insert line breaks at places where there is no line break in the language definition. For example, there is both a statement and an expression `NieuweRegel` (new line), and an expression `NieuweRegelEnInspringen`

```
context-free syntax
  "\n" "Voer" "uit" Expression -> Statement {cons("VoerUit")}

  "<-'" "\n"      -> Expression {cons("NieuweRegel")}
  "<-'" "\n" "->" -> Expression {cons("NieuweRegelEnInspringen")}
  "<-'" "\n"      -> Statement  {cons("NieuweRegel")}
```

Figure 5.2: Mandatory line breaks and line continuation constructs

(new line and indent). These constructs have no semantics. They are present only to change the layout of the program in the structure editor.

If we convert these constructs to SDF we end up with productions as shown in Figure 5.2. The expression productions work only when expressions are parsed as lists of interleaved subexpressions and operators (see §2.8.1), instead of trees. Since we decided to parse expressions as trees, the `NieuweRegel` and `NieuweRegelEn-Inspringen` productions are useless to us. The statement `NieuweRegel` production is compatible with the generated grammar, although it would be desired if separator lines between statements can be left blank, instead of forcing the user to enter the awkward three-character symbol "`<-'`" shown in Figure 5.2. We have examined three ways to deal with line endings.

No mandatory line endings

The first alternative we consider is to remove all `"\n"` literals from the grammar, add the newline character to the `LAYOUT` symbol, and remove the line continuation constructs from the models. This results in clean model code, without the syntactic noise of line continuation constructs. This approach is what is typically used in DSLs created with Spoofax.

Line continuation construct and mandatory line endings

The second alternative is to keep `"\n"` literals in the grammar, remove the newline character from the `LAYOUT` symbol, and insert a line continuation construct for those cases where a single-line statement needs to be split over multiple lines. This approach results in slightly more syntactic noise in the model code. Additionally, an extra line break, without a line continuation construct, will result in a parse error. We believe that both points are not ideal from a user experience point of view. Besides this, we had trouble in practice to get the model transformation output line continuation constructs at the proper places.

However, we observed much better parser performance than without mandatory line endings, in particular when recovering from parse errors. We believe this can be attributed to the fact that the language consists mainly of identifier-like tokens, partly because the identifier sublanguage is larger than in many DSLs, and partly because many language constructs consist of Dutch words with only minimal punctuation. This means it may be expensive for the parser to get back "in sync" with the stream of characters after a parse error, unless statements happen to be separated, terminated, or preceded by a *fiducial symbol* [43] such as the newline character.

Optional and mandatory line endings

The third and last alternative is to keep `"\n"` literals in the grammar, add the newline character to the `LAYOUT` symbol, and remove the line continuation constructs from the models. We believe this option is ideal from the user experience point of view: there is no need to remember to insert a line continuation symbol when a blank line is desired, and the parser is able to recover from parse errors a lot faster than when we remove mandatory line endings completely. Fast recovery is important for editor services like content assist.

There is a challenge, however: when a newline character may be part of `LAYOUT`, then we can choose, using a restriction, whether or not `LAYOUT` should greedily consume newlines. If we choose the greedy approach, then layout can not be allowed before a mandatory newline, for this layout *will* consume the mandatory newline before it can be matched against the newline at the start of a statement. This is not acceptable, as empty lines or space characters before a line break would generate parse errors in seemingly arbitrary contexts. If we choose the non-greedy approach, then we create an ambiguity every time an optional newline is present at a place where two or more consecutive optional layout symbols (`LAYOUT?`) are expected. Though SDF inserts only one optional layout between all symbols of each context-free production, consecutive optional layout still appears whenever any symbol is used that can produce the empty string. For example, in `A B? A` there are two consecutive optional layout symbols if `B?` produces the empty string. In such a case, the parser cannot know how to distribute the actual layout over the `LAYOUT?` symbols, which results in an ambiguity between the possible distributions over the two `LAYOUT?` symbols.

We have been able to reduce this ambiguity by using a production to greedily consume layout, including optional newlines, without preventing layout to appear before a mandatory newline (Figure 5.3). The remaining ambiguity does not occur frequently: there have to be two consecutive optional layout symbols in the grammar, as before, and at the position in the input text where these layout symbols are expected, there must be a line that ends with layout, followed by a line break. In this case the layout at the end of the line is consumed by the first `LAYOUT?` symbol, because of the context-free restriction in Figure 5.3. The line break (and any further white space) will be consumed by the `LineBreak` symbol. At this point it is ambiguous whether the `LineBreak` should be combined with the layout at the end of the line, and consumed by the first `LAYOUT?` symbol, or whether it should be consumed by the second `LAYOUT?` symbol. Because this ambiguity is lexical and involves layout only, which is not present in the AST in Spoofax, the ambiguity has the trivial structure “`amb([x, x])`”. The ambiguity has been disambiguated using a custom disambiguator written in Stratego (Figure 5.4).

Overview

We investigated three different ways to handle explicit line breaks present in the meta model in Cheetah. The first option we considered was to remove line breaks altogether, thereby making the migrated models insensitive to newline characters. This option resulted in an ambiguous grammar, thus making it impossible to parse model code reliably. The second option was to map line breaks in Cheetah to required newline char-

```
lexical syntax
  [\ \t\r] -> LayoutChar
  LayoutChar -> LAYOUT

  "\n" ("\n" | LayoutChar)* -> LineBreak
  LineBreak -> LAYOUT

lexical restrictions
  LineBreak -/- [\ \t\n\r]

context-free restrictions
  LAYOUT? -/- [\ \t\r]
```

Figure 5.3: Lexical syntax for optional and mandatory line endings

```
  editor-disambiguate:
    (ast, path, project-path) -> <alltd(disambiguate)> ast

  disambiguate:
    amb([x, x]) -> x
```

Figure 5.4: Disambiguator necessary for optional and mandatory line endings

acters in the grammar, while adding a line continuation construct for additional (optional) newlines. We dismissed this option because we believe such an approach would confuse the user, as it would be hard to learn at which places the grammar expects a newline character. The third option employs a carefully crafted grammar for layout (Figure 5.3), including newline characters, combined with a custom disambiguator (Figure 5.4) for the remaining occasional ambiguity. In practice, this approach results in a grammar that is both easy to use for the DSL user, is not ambiguous, apart from the aforementioned `LAYOUT` ambiguity, and has decent recovery characteristics due to the newline characters that act as fiducial symbols.

5.5 Issue: Priorities

After running the migration, one important part of the grammar was still missing: a specification of the priorities of certain constructs over other constructs. For the initial migration prototype we solved these ambiguities by defining ad-hoc disambiguation rules. These disambiguation rules are based on common sense (multiplication has a higher precedence than addition, etc.), and examination of actual occurrences of ambiguities in generated model code.

5.6 Issue: Conflicting Productions

Cheetah defines four different ways of reducing an `Identifier` to an `Expression` (Figure 5.5), which causes an ambiguity anytime an identifier is used. We disabled part of the generated grammar, so that only one of the responsible productions remains

```
context-free syntax
  Identifier -> Expression {cons("BevestigendeBetekenis")}
  Identifier -> Expression {cons("VragendeBetekenis")}
  Identifier -> Expression {cons("Identificatie")}
  Identifier -> Expression {cons("Subdeclaratie")}
```

Figure 5.5: Ambiguous injections

```
context-free syntax
  Expression "en" Expression -> Expression
  "bij" {Expression "en"}+ -> MethodParameters
```

Figure 5.6: Ambiguity between logical and operator and method parameters

active. During an analysis phase, the AST node created through this production will then need to be transformed into a context-specific node.

5.7 Issue: Method Invocation and the Logical And

Cheetah uses the keyword *en* for a logical and operator, and as parameter separator in method invocations (Figure 5.6). This introduces a number of ambiguities. In any expression of the form “A bij B en C” it is unclear whether this is a method invocation with two parameters B and C, a method invocation with a single boolean parameter “A bij (B en C)”, or a method invocation with a single parameter B whose result is the left hand side of the *en* operator: “(A bij B) en C”. This ambiguity is not just theoretical; constructs of exactly this form are omnipresent in the models. Worse yet, often the actual structure of “A bij B en C en D” is “(A bij B en C) en D” (method call with two parameters nested in a logical and). The occurrence of this construct makes it impossible to programmatically determine from the concrete syntax what is meant, *regardless* of the way priorities are specified.

Therefore, this ambiguity has to be solved by a simultaneous refactoring of the meta model and the model. We decided to title case the keywords *bij* and *en* to *Bij* and *En*, and to insert parentheses to delimit the parameter list. The first change solves the conflict between the parameter separator and the logical and operator, while the second change defers the issue of establishing a consistent precedence for the method call (i.e., how strong the method call binds to its last parameter). The parentheses can be dropped when a precedence for the method call has been established.

5.8 Conclusion

In this chapter we migrated the language definition in Cheetah to SDF, to be used in a Spoofax editor for the tax-benefit rule modeling language. Since Cheetah has not been designed with parser-based technology in mind, there was some mismatch between the concepts in Cheetah and the concepts in SDF. In particular, we had to make the language accepted by our grammar more permissive than what is allowed in Cheetah to prevent an explosion of grammar productions. Semantic analyses of the AST can be used to lock down the language later on in the parsing & analysis process.

The largest challenge we had to overcome to enable the Spoofax plugin for the tax-benefit rule modeling language to parse sized chunks of DSL code was the lack of statement separators. In a structure editor it is not a problem if the entire language consists of words, with barely any punctuation. However, we found that with generalized parsing, it is necessary to introduce a fiducial symbol, and require that identifiers consist of only a single word, for acceptable parsing performance. We designed a snippet of SDF that introduces line breaks as fiducial symbol without requiring a line continuation character whenever the user wants to insert a line break in a context where it is not required. Additional challenges consisted of disambiguating various part of the syntax, partly through adding a priority specification, partly through the removal of productions identical modulo constructor, and partly through changes to the syntax of the language.

Overall we can look back at a successful migration. In particular our choice to develop an automatic transformation was helpful when experimenting with different translations of certain aspects of the language, such as the syntax for method invocations. Although our migration resulted in a functional DSL editor in Spoofax, we recommend to refactor the language so as to make it more suitable for parser-based tooling. Such refactorings can be integrated into the automatic transformation, highlighting another advantage of an automatic migration. In the next chapter we migrate the DSL code to text that ought to be parseable by the grammar we generated in this chapter.

Chapter 6

Migrating Cheetah Models to Textual Models in Spoofax

6.1 Introduction

Since the tax-benefit rule modeling DSL is used for a system that runs in production, much code (models) has been written in this DSL. For similar reasons as mentioned in the introduction of Chapter 5, we choose to develop an automatic migration of these models, instead of performing a manual conversion. Hence, the goal of this chapter is to migrate the models in Cheetah to sentences in the language described by the grammar generated in Chapter 5. Using the results of this migration we are able to evaluate our work in Chapter 5.

This transformation is more involving than generating the grammar, because the model XML files contain only the variable elements (i.e., modified by the user), due to the prototypical inheritance in Cheetah (§2.4). Therefore, there is no one-to-one mapping between model element and generated artifact, but instead many auxiliary model elements need to be read to be able to generate the program text for a single model element. Since many of the language design issues with a migration from a structure editor to a parser-based language workbench are described in Chapter 5, we keep this chapter focused on the automatic transformation.

6.2 Model Transformation: Generating Text

To collect the information required to convert models we have to resolve two types of references to other model elements. First, we need to "flatten" the inheritance hierarchy, thereby merging all properties of inherited model elements into the current model element. Second, we need to bring in the values of properties that are unchanged in all model elements in the tree that results from the first step. We developed Stratego code that reads a single model element (using the parser developed in Chapter 4), and then traverses the inheritance hierarchy to bring in all inherited properties. Then, it walks the resulting tree, and brings in any properties missing from model elements referred to through the *cls* property, which indicates instantiation of a model elements.

Once all the information making up a model has been merged, generating the model text is as simple as traversing the tree and writing out the concrete syntax for each AST element. `GLEText`, for example, produces the text that is inside, while for `GLEEnumeration` we take the text of the enumeration item referred to by the *selectedItem* property of the AST element. When we encounter an `SLEStatements` we produce the text of the statements contained within, and then indent the whole block. We employ similar strategies for other language elements.

6.3 Issue: Free Form Identifiers

In Cheetah, references to identifiers are always inserted using content assist, so there is no need for any restrictions on the names of identifiers, apart from the pragmatic requirement that two distinct identifiers that may be used in the same scope must have a distinct representation in the user interface.

In practice, the lack of restrictions lead to identifiers consisting of multiple words separated by whitespace, identifiers with leading or trailing whitespace, identifiers containing characters which are often not allowed in mainstream GPLs, such as parentheses (used to indicate an optional plurality, e.g. "parent(s)") or percent signs, and identifiers that start with numbers (e.g., "50% calculation").

For each of these cases we had to choose to either map the undesired character to an allowed characters, or to change the ID lexical (and potentially other parts of the grammar) to allow the special character. We made the following decisions: Leading and trailing whitespace has been trimmed as it was not significant anywhere. Embedded whitespace has been converted to dashes to be able to recognize the end of an identifier. Parentheses, percent signs, hashes and question marks have been allowed in identifiers; these did not result in any ambiguities. Parentheses have to be balanced for a successful parse. Identifiers have also been allowed to start with a number, but must contain at least one non-numeric character to prevent ambiguity between identifiers and literal integers.

6.4 Conclusion

In this chapter we migrated the DSL code of the tax-benefit rule modeling language to plain text, parseable by the grammar we generated in Chapter 5. We had to overcome two challenges: collecting the information to convert a single model to text, i.e. resolving references to extended and instantiated model elements, and sanitizing identifiers. We addressed the first challenge by developing Stratego code to read in a model element, and all referred model elements, and merge those into a single tree containing all information; the second challenge was addressed by picking a sweet spot between complete sanitization of identifiers to alphanumeric characters only, and modifying the grammar to accept all characters occurring in identifiers in practice.

The combined work of Chapter 5 and this chapter resulted in an Eclipse plugin capable of opening real tax-benefit rule modeling DSL models, including basic editor services, such as syntax highlighting, which are automatically derived by Spoofax from the syntax definition. In the next chapter we discuss some options w.r.t. a migration of the code generators from Cheetah to Spoofax.

Chapter 7

Migrating Cheetah Transformations to Stratego

7.1 Introduction

There is more to a conversion of Cheetah to Spoofax than a conversion of the syntax. In particular, there is a set of code generators in Cheetah, which generate complete Visual Studio / .NET / C# projects from the models entered into Cheetah. Although there have been plans to add a transformation language to Cheetah, these never took off, so currently these code generators are written in Java. As Spoofax builds on Eclipse, which is also written in Java, there may be an opportunity to reuse the code generators. In this chapter we describe how this might be implemented, and which problems may appear. Alternatively, the code generators could be rewritten in a specialized transformation language, such as Stratego [8].

7.2 Reusing Existing Transformations

Code Generators in Cheetah

The code generators in Cheetah consist of custom Java source code, which operates on the `*AST` interfaces introduced in Chapter 2. As the code generators represent a mature code base, it would be ideal if we could reuse the transformations in case of a switch to Spoofax.

Stratego term abstraction

Stratego is not coupled to the particular term implementation it uses by default. Both the Stratego compiler and the interpreter operate on the `IStrategoTerm` interface. Any tree of objects that implement this interface can be rewritten by transformations written in Stratego. In the past this has been used to let Stratego operate on nodes of the ECJ AST [29], thereby allowing an Eclipse user to add custom analyses of Java code, written in Stratego. Additionally, Spoofax offers a simple foreign function interface (FFI), which allows the user to implement strategies in Java. By leveraging these two features of Stratego we can write Java classes that implement both `IStrategoTerm`, and one of the `*AST` interfaces.

Proxying model elements

We have examined whether the existing code generators can operate on a different implementation of the `*AST` interfaces by generating proxy objects that implement each respective interface. Each of the methods of the interface that returns another interface is then implemented so that it wraps the returned model element in the corresponding proxy object. Using this approach the code generators never directly communicate with the model elements, but operate on proxies instead, allowing us to find code in the code generators that would certainly not work with a different implementation of the interfaces.

Casts to the implementation type

Running the code generators using the proxy objects revealed some issues with the transformations. Foremost, some of the code generators cast the object they receive, of which the declared type is one of the `*AST` interfaces, to the type they expect to implement the interface. Typically this is done because they need more information then what is provided through the interface. This means that, to be able to run the code generators on a different AST implementation, the additional information must be added to the interface, and the casts must be removed from the code generators.

Reference equality vs. value equality

Another issue relates to the difference between the `equals` method and the == operator in Java. The `equals` method typically compares by value (*value equality*), while the == operator always compares identity (*reference equality*). For example, two strings `s1` and `s2` with equal contents will compare equal using `s1.equals(s2)`, but may or may not compare equal using `s1 == s2`. The strings are only considered equal in the last case if `s1` and `s2` point to the same `String` object, which may or may not be true depending on the source of both strings [4].

In the code generators this is an issue because the == operator is regularly used to compare two model elements, which implies our proxy objects cannot be instantiated on the fly every time they are requested, as in our initial approach. Instead, we need to keep track of all objects which already have a proxy object, and reuse that proxy object whenever the object is requested again. That is, there must be a global, one-to-one mapping from each model element instance to its proxy object. For an actual reuse of the code generators we would need to break the sharing of subterms before passing the AST to the code generators, to ensure the == operator never returns a true result where it would have returned a false result in Cheetah. The alternative is, of course, to remove all ==-based comparisons from the transformations, while ensuring the generated code remains the same. This is the recommended way to proceed.

Inheritance hierarchy mismatch

A last, but nevertheless important issue is that the inheritance hierarchy present between model elements is not accurately represented in the interfaces implemented by the model elements. This implies that proxy classes or `IStrategoTerm` implementations cannot be generated from the `*AST` interfaces alone: we need to look at the

Cheetah implementation of each `AST*` interface, to examine whether it also implements other `*AST` interfaces that should have been inherited by one of the interfaces it does implement. What makes matters worse is that Cheetah employs dynamic dispatch to find and invoke the best matching `transform` method. The dynamic dispatch mechanism determines the best matching method by checking whether the passed arguments can be assigned to the parameter types, and then taking the variant with the most specific parameter types. Hence, if a new AST implementation is used as input for the code generators, the dynamic dispatcher may invoke different transformation methods unless we carefully reproduce the assignability of the new implementation to the types expected by the code generators.

7.3 Converting Existing Transformations

A different approach to move the code generation into Spoofax is to convert the existing code generators to Stratego. The code generators consist of approximately 70,000 lines of Java code. When transforming Java to Stratego we move to a higher abstraction level and a different paradigm, so it is unlikely that a fully automatic conversion can be pulled off. That means large parts of the code generators need to be converted by hand.

7.4 Conclusion

Unlike Chapter 5 and Chapter 6, where we chose an automatic migration over a manual migration, we recommend a manual transformation of the code generators, because of the paradigm mismatch between Stratego and Java. With a manual transformation it will be possible to obtain idiomatic Stratego code, which is a requirement for maintainable code generators in Stratego. Another approach we investigated is reuse of the current code generators, by writing a set of adapter classes, to be able to use the code generators on Stratego term objects. Short term, this solution may be sufficient. For longer term maintenance, or new projects, we recommend to move the code generators to a specialized transformation language, such as Stratego.

Chapter 8

Content Assist

8.1 Introduction

High quality content assist is a prerequisite for a successful deployment of a parser-based, textual editor for the tax-benefit rule modeling language, because the language is rather verbose. In particular, it employs 317 unique keywords (some are reused in many productions) in 1036 productions[1], while Java 1.5 uses 53 unique keywords in 440 productions[2]. In a parser-based language workbench like Spoofax, the user needs to enter this content without making mistakes: a one character typo can make a complete statement unparseable. It is therefore important to have high quality content assist. This assertion is further reinforced by the fact that, even in the Eclipse Java editor, content assist is one of the most used features [41].

For template-based editing (i.e., using content assist to insert code templates) to be effective, only contextually relevant templates should be shown. Pure structure editors achieve this based on the currently selected placeholder. To achieve the same in textual editors, the editor must be aware of the syntactic category of the text at the cursor location at any time. The provided completions must be accurate: all applicable templates must be included, and no inapplicable templates may be included. Which completion templates are to be included can be determined from the syntactic category at the cursor location. Determining this syntactic category in a language-agnostic fashion is not trivial. If possible, changes to generated parsers to support this facility should be provided. In addition, syntax errors need to be taken into consideration; the editor must be able to determine what type of template should be inserted even when a program is edited and is in a syntactically incorrect state.

In this chapter we describe an approach for gathering an accurate list of templates in a parser-based editor, as well as a number of generic improvements to the content assist user experience in the Spoofax language workbench.

[1]Counted after the migration performed in Chapter 5

[2]Approximation based on the Java 1.5 syntax definition in java-front, found at: `https://svn.strategoxt.org/repos/StrategoXT/java-front/trunk/syntax/src/`

8.2 Determining Relevant Completion Templates

Original implementation

The solution originally implemented in Spoofax first creates a modified program text that includes a marker at the cursor location. The marker matches the syntax for identifiers, and is unlikely to be present anywhere else in the program text. The modified program text is then parsed, after which the AST is searched for the marker. Spoofax infers the symbols that should have been allowed at the position of the cursor from the token stream, and meta data attached to AST nodes.

The interaction between the involved components is a problem with this implementation. When the modified program text is parsed, and it has parse errors, error recovery gets involved. Unless completion is invoked at a position where an identifier is allowed, there will be parse errors. The error recovery algorithm in the SGLR parser used in Spoofax attempts to get the parser back on track by performing a minimum number of token insertions and/or removals. As such, it may remove the marker, or insert punctuation that pushes the marker into another language construct.

One solution is to make the parser aware of the cursor location, and report the allowable syntactic categories at that character offset during parsing. This solution is specific to SGLR, and needs to be re-implemented in every other parser. We look for a more generic and less complex solution.

Our solution

We add a production `CONTENTCOMPLETE -> X {cons("COMPLETION-X")}` for every symbol `X`, where the symbol `CONTENTCOMPLETE` recognizes the inserted marker text, including surrounding identifier characters. When the program text with marker is parsed, the marker can be parsed as every symbol allowed at its location. Because we encode the name of the symbol in the AST constructor, and the parser is able to return all the different, ambiguous parses of the marker, the editor runtime knows all symbols allowed at the position of the marker. The set of symbols is then used to select appropriate completion templates to display to the user.

8.3 Content Assist User Experience

Original implementation

Applying a completion proposal in Spoofax used to work as follows. The user triggers a content assist pop-up to appear, either by pressing a designated keyboard shortcut, or by entering some text in the editor that matches one of the *completion trigger* regular expressions declared in the editor services specification of the DSL. In this pop-up menu, the user selects a completion proposal, or the user continues typing, in which case the proposals will be filtered on the prefix entered by the user. The pop-up menu will disappear if no proposals match the prefix.

Once the user selects a proposal the text of the template will be inserted in the document. Both text elements and placeholders appear in the document as is. The text of the first placeholder will be selected, so the user can immediately start typing the contents of that placeholder. However, there are no visual cues apart from the text itself, as to the locations of other placeholders: the user will have to locate, select, and replace them manually.

Our solution

We modified content assist in the Spoofax runtime to use the Eclipse *linked mode*[3] feature. Linked mode is a state of the Eclipse text editor where multiple regions in the source are marked. These regions are called *linked positions* or *proposal positions*. While the editor is in linked mode, the tab key is used to jump to the next position, and each position acquires a visual cue in the editor to indicate its extents to the user (see Figure 8.1). Additionally, some or all of the positions may be part of a *linked position group*, which is a group of positions with identical text, so that when any one of these positions is modified, all positions in the group are updated. The rename refactoring in the Eclipse Java Development Tools is a well known use of these linked position groups. Linked mode is exited when the user starts typing outside the linked positions, or when the escape key is pressed.

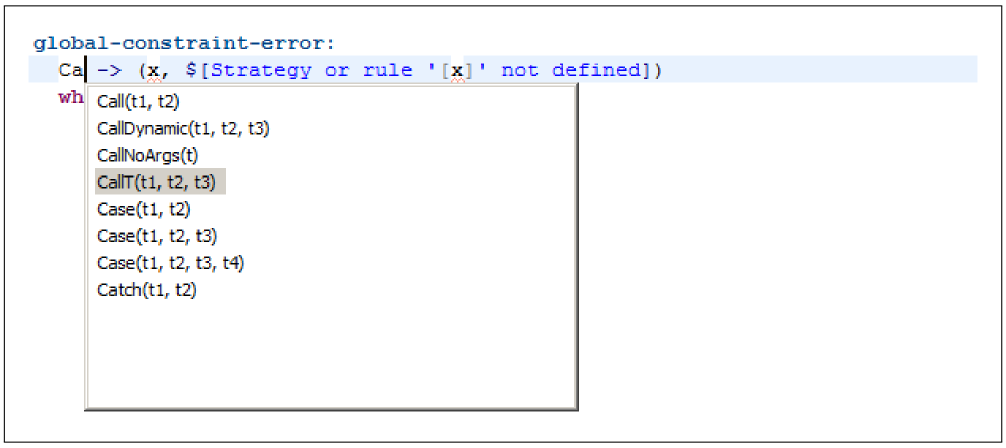
Our modification to content assist in the Spoofax runtime converts all placeholders in a completion template to linked positions or proposal positions, and benefits all DSLs developed with Spoofax.

8.4 Discussion

Although we have pushed content assist in Spoofax a small step forward, there are still many areas for improvement. It should not be necessary to clone the grammar modifications performed manually in §8.2 to all DSLs in Spoofax that want syntactic completion. Potentially these productions can be generated at the same time recovery productions are generated. Another approach to the same issue could be to have the content assist runtime parse the source code twice: once without the `CONTENTCOMPLETE`*x* literal, for syntactic completion, and once with the `CONTENTCOMPLETE`*x* literal, for semantic completion. A problem with this approach is that parsing is relatively expensive: parsing twice, instead of once, will double the response time of content assist, which can already be long in the case of large inputs or ambiguous grammars.

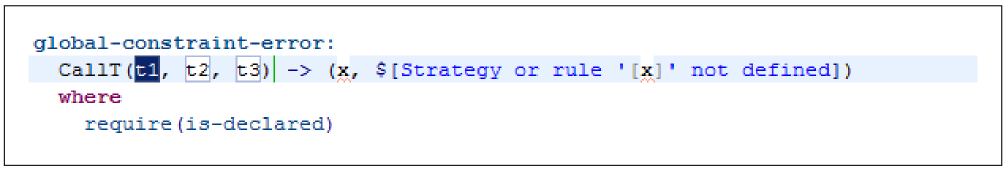
It may be interesting to research if syntactic content assist can be coupled tighter to the parser. During a parse the parser knows which symbols are acceptable at each position in the input text as it passes along that position. If a symbol is acceptable at a position, this implies the first symbol of each production with that target symbol would be acceptable too. If the first symbol may produce the empty string, then the second symbol should be added to this set, and so on. We could continue to add the whole *First-set* of the symbol to the completion proposals. At some point, however, we need to cut off the search to produce meaningful proposals to the user. Whether and how we can acquire meaningful proposals using this approach, remains to be seen.

[3]`http://help.eclipse.org/indigo/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/jface/text/link/package-summary.html`



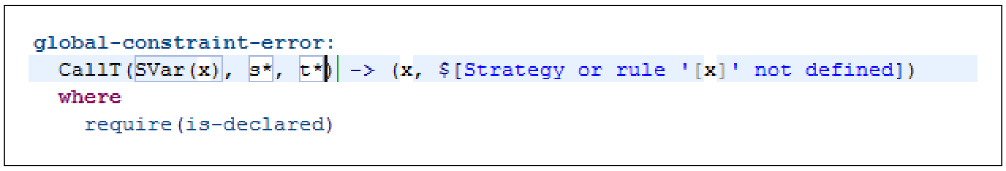
```
global-constraint-error:
  Ca| -> (x, $[Strategy or rule '[x]' not defined])
  wh
```

(a) Trigger content assist, and choose a completion proposal



```
global-constraint-error:
  CallT(t1, t2, t3) -> (x, $[Strategy or rule '[x]' not defined])
  where
    require(is-declared)
```

(b) Apply the completion proposal, and placeholders (*linked positions*) appear



```
global-constraint-error:
  CallT(SVar(x), s*, t*) -> (x, $[Strategy or rule '[x]' not defined])
  where
    require(is-declared)
```

(c) Jump between positions, and enter values

Figure 8.1: Spoofax content assist with *linked mode* support in the Stratego editor

Chapter 9

Eliminating Concrete Syntax Repetition using Syntax Templates

9.1 Template-Based Syntax Definition

In this chapter we introduce a new syntax definition language[1] based on templates as those found in template engines such as StringTemplate [47]. Additionally, we base the design of auxiliary features of the language, such as priority specification and lexical syntax, on that of SDF [22, 61]. The aim of the language is to eliminate the redundancy between different syntactic specifications, by combining concrete syntax, abstract syntax, formatting, whitespace, and placeholder names. We argue that through the use of templates, the language is rich in information yet elegant and simple.

Language Overview Basic template-based syntax definitions consist of *template productions* that correspond to production rules in a grammar. They have the following form:

```
s.label = <
  template
>
```

where *s* is the name of the symbol being defined, *label* is its constructor label used for the abstract representation, and *template* is a template that may include concrete syntax, references to other symbols (*placeholders*), and layout. Both the template and its placeholders are enclosed by `<...>` brackets.

As a first example, the following template productions define basic arithmetic expressions:

```
templates
  Exp.Num = <<INT>>
  Exp.Plus = <<Exp> + <Exp>>
  Exp.Times = <<Exp> * <Exp>>
```

The first production defines a template for number literals, defining a template for the `Exp` symbol based on a reference to the `INT` symbol. The other productions specify templates for the `+` and `*` operators. The last two templates consist of five elements: an `<Exp>` placeholder, whitespace, the `+` or `*` sign, more whitespace, and another placeholder. Of these elements, the whitespace elements are not considered for parser gener-

[1] http://strategoxt.org/Spoofax/TemplateLanguage

```
Statement.ArrayDeclInit = <
  array <ID> = { <INT; wrap, separator=","> };
>
```

(a) Syntax template with a placeholder with wrap option

```
array a = { 3,9,20,2,1,4,6,32,5,6,77,888,
2,1,6,32,5,6,77,4,9,20,2,1,4,63,9,20,2,1,
4,6,32,5,6,77,6,32,5,6,77,3,9,20,2,1,4,6,
32,5,6,77,888,1,6,32,5 };
```

(b) Output of Figure 9.1(a): wrapped lines start at the left margin

```
Statement.ArrayDeclInit = <
  array <ID> = { <INT; wrap, anchor, separator=","> };
>
```

(c) Syntax template with a placeholder with wrap and anchor option

```
array a = { 3,9,20,2,1,4,6,32,5,6,77,888,
            2,1,6,32,5,6,77,4,9,20,2,1,4,
            63,9,20,2,1,4,6,32,5,6,77,6,
            32,5,6,77,3,9,20,2,1,4,6,32,
            5,6,77,888,1,6,32,5 };
```

(d) Output of Figure 9.1(c): wrapped lines anchored to the left side of the placeholder

Figure 9.1: Wrap and anchor options demonstrated for a Java array initializer

ation. Instead they are used for formatting in a generated pretty printer and completion templates. Whitespace characters treated this way are spaces, tabs, and newlines.

Placeholders can use the common `*` and `+` operators for repetition, and `?` for optionals. For repeated symbols with a separator symbol *s*, <symbol*; **separator**=*s*> can be used. The following template productions illustrate these features, adding function calls and definitions to the expression language.

```
templates
  FunctionDef.Function = <
    function <ID>(<ID*; separator=",␣">) = <Exp>
  >
  Exp.Call = <<ID>(<Exp*; separator=",␣">)>
```

Placeholder options that affect only pretty printing, are available for line wrapping, and anchoring of wrapped lines to the left side of the placeholder (Figure 9.1). A **hide** placeholder option is available to hide the placeholder from completion templates (to be used for expert language features).

Disambiguation Grammars can be extended with disambiguation rules and annotations to express language characteristics such as associativity and operator precedence. In our running example, multiplication has a higher precedence than addition. Whereas in SDF [36] priorities are specified declaratively by *copying* the relevant productions and ordering them, separated by >-symbols, we add the option of specifying priorities through *references* to the relevant productions, so as to eliminate redundancy. The difference is shown in Figure 9.2.

```
templates
  Exp.Plus  = <<Exp> + <Exp>> {left}
  Exp.Times = <<Exp> * <Exp>> {left}
context-free priorities
  Exp.Times >
  Exp.Plus
```

(a) Associativity and priorities with templates and references

```
context-free syntax
  Exp "+" Exp -> Exp {left, cons("Plus")}
  Exp "*" Exp -> Exp {left, cons("Times")}
context-free priorities
  Exp "*" Exp -> Exp >
  Exp "+" Exp -> Exp
```

(b) Equivalent associativity and priorities in SDF

Figure 9.2: Expression grammar

```
lexical syntax
  ID      = [A-Z] [A-Za-z0-9]*
  INT     = [0-9]+
  LAYOUT = [\ \t\r\n]
```

(a) Lexical productions

```
lexical restrictions
  ID  -/- [A-Za-z0-9]
  INT -/- [0-9]
context-free restrictions
  LAYOUT? -/- [\ \t\r\n]
```

(b) Lexical and context-free restrictions

Figure 9.3: EBNF-ordered productions

Lexical Syntax A part of syntax definitions we have not discussed so far is lexical syntax. Lexical syntax elements such as `INT` and `ID` in our expression language, are, unlike the context-free productions we discussed so far, generally specified using a form of regular expressions. In the abstract representation they are usually represented as simple strings, making unparsing trivial. For consistency, lexical productions can be specified in symbol-first order, as shown in in Figure 9.3. The definition of the body of lexical productions is shared with SDF [22, 61]. Both lexical and context-free syntax can be disambiguated using restriction sections in SDF [36], a construct that we inherit in our syntax specification language (Figure 9.3).

The syntax of the template language is summarized in Figure 9.4. We proceed with a description of the mapping from syntax templates to SDF, completion templates, and pretty printing rules.

9.2 Generating Template-Based Editors

To SDF Syntax templates closely match context-free syntax in SDF. Specifically, to go from a syntax template to an SDF production, all layout is discarded: in SDF, there is implicit `LAYOUT?` between all symbols in a context-free production. The remaining elements (literals and placeholders) are converted in-order to their respective SDF equivalents. Literals are tokenized at every boundary between (sequences of) characters specified with the **`tokenize`** option, and (sequences of) other characters. The default value `"()"` ensures that language elements such as `if(cond)` can be written (and thus pretty printed) without whitespace between `if` and the parenthesis, while the grammar still allows `LAYOUT?` between those. Layout is trimmed from the separator option of list placeholders. An example of the transformation to SDF is shown in Figure 9.5. Optionally, if the **`newlines`** option is present with a value other than **`none`**, the generated grammar is modified to require newline characters between lines in the syntax templates. This situation is depicted in Figure 9.6.

Template language sections	
`templates` `t*`	Section with template productions $t*$
`template options` `o*`	Section with options $o*$
Productions	
`Symbol = <e*>`	Template with elements $e*$
`Symbol.Cons = <e*>`	Template with elements $e*$
`Symbol = [e*]`	Template with elements $e*$ (alt. brackets)
`Symbol.Cons = [e*]`	Template with elements $e*$ (alt. brackets)
Placeholders	
`<A>`	Placeholder (1)
`<A?>`	Optional placeholder (0..1)
`<A*>`	Repetition (0..n)
`<A+>`	Repetition (1..n)
`<A*; separator="\n">`	Repetition with separator
`<A; text="hi">`	Placeholder with replacement text
`<A; hide>`	Hidden from completion template
`<A*; wrap>`	Repetition with word-wrap
`<A*; wrap; anchor>`	Repetition with word-wrap, anchored to the left column of the placeholder
Escapes	
`<\ \t\r\n>`	Element containing escaped characters
`<\u0065>`	Unicode escape
`<\\>`	Escape next line break
`<>`	Empty escape sequence: no output; layout will be allowed here in the grammar
`\<, \>, \[, \], \\`	Brackets/backslash (prefer alt. brackets)
Priority specification	
`context-free priorities` `{left: Exp.Times Exp.Over} >` `{left: Exp.Plus Exp.Minus}`	References to template productions
Lexical syntax	
`lexical syntax` `ID = [A-Za-z] [A-Za-z0-9]*`	EBNF-order productions in SDF
Template options	
`keyword -/- [A-Za-z0-9]`	Follow restriction for keywords
`tokenize : "()"`	Layout is allowed around these characters
`newlines : none`	Grammar requires newline characters. Possible values: `none`, `separating`, `leading`, `trailing`

Figure 9.4: Summary of the template language syntax

```
Statement.IfThen = <
  if(<Exp>) <Statement*; separator="\n">
>
```

⇓ generate SDF ⇓

```
"if" "(" Exp ")" Statement* -> Statement {cons("IfThen")}
```

Figure 9.5: Generate an SDF production from a syntax template

```
Statement.IfThen = <
  if <Exp> then
    <Statement*; separator="\n">
  end
>
```

⇓ generate SDF ⇓

```
"if" Exp "then" ("\n" Statement)* "\n" "end" -> Statement {cons("IfThen")}
```

Figure 9.6: Generate an SDF production with `newlines: separating` option

To completion templates Completion templates in Spoofax consist of the following components:

- A symbol that indicates the context in which the template is applicable.
- A string, which is displayed in the completion pop-up, and is used to filter the list of proposals.
- A list of elements of the completion template. Each element is either a string, possibly including line breaks and indentation, a placeholder, or the special `(cursor)` directive. This last directive indicates the location of the cursor after the user has cycled through all placeholders. A placeholder consists of an initial replacement text, and an optional symbol, which is used to display a list of syntactic completions applicable at the position of that placeholder, as soon as the user switches to this placeholder.
- A set of annotations. The only relevant annotation in use is `(blank)`, which constrains the completion template to blank lines.

We do not perform a linear transformation from syntax templates to completion templates, as we did for the grammar. The reason is best illustrated with Figure 9.7. For certain language elements, we may want to factor out repeated constructs, such as the `<Statement*; separator="\n">` placeholder in the example. The user of the editor, however, should not be exposed to such implementation details. In particular, the user should not be forced to repeatedly apply completion to fill in required parts of a language construct: those required parts should be inserted into the program text as soon as the completion proposal for the language construct is applied.

```
FunctionDef.Function = <
  <MetaAnnos>
  function <QId>(<FArg*; separator=",␣">) : <Type> {
    <Statements>
  }
>
MetaAnnos  = <<MetaAnno*; separator="\n", hide>>
Statements = <<Statement*; separator="\n">>
```

⇓ expand template ⇓

```
FunctionDef.Function = <
  <MetaAnno*; separator="\n", hide>
  function <ID:QId>(<FArg*; separator=",␣">) : <ID:Type> {
    <Statement*; separator="\n">
  }
>
```

⇓ simplify template ⇓

```
FunctionDef.Function = <
  function <ID:QId>(<:FArg>) : <ID:Type> {
    (cursor)
  }
>
```

⇓ generate completion template ⇓

```
completion template FunctionDef: "function␣ID()␣:␣ID␣{␣}" =
  "function␣" <ID:QId> "(" <:FArg> ")␣:␣" <ID:Type>
    "␣{\n\t" (cursor) "\n}" (blank)
```

Figure 9.7: Generate completion templates from syntax templates

Therefore, we substitute the referred template for each placeholder with a multiplicity of one and higher (`<A>` and `<A+>`), unless the placeholder refers to the containing template. In the step *expand template* of the example in Figure 9.7, the `<MetaAnnos>` and `<Statements>` placeholders are expanded.

The *simplify template* step removes placeholders with the **`hide`** option, and processes placeholders that can generate the empty string (`<A?>` and `<A*>`). Call those placeholders ε-placeholders. These placeholders are treated differently depending on their location in the template. The **`(cursor)`** directive is substituted for the first line that consists of a single ε-placeholder. Further instances of ε-placeholders on a single line are ignored: we expect the user to retrigger completion when they desire to insert a template at these positions. In Figure 9.7, `<MetaAnno*; separator="\n", hide>` is removed, and the **`(cursor)`** directive is substituted for `<Statement*; separator="\n">`. Remaining ε-placeholders are replaced by an empty placeholder in the completion template that can be expanded to a single occurrence of one of the referred templates. In Figure 9.7 we demonstrate this by substituting `<:FArg>` for `<FArg*; separator=",␣">`.

```
Statement.IfThen = <
  if <Exp> then
    <Statement*; separator="\n">
  end
>
```

⇓ generate Stratego pretty printing rule ⇓

```
prettyprint-Statement:
  IfThen(a, b) -> zz
  with a′ := <prettyprint-Exp> a
      ; b′ := <map(prettyprint-Statement); separate-by(|"\n")> b
      ; zz := <concat-strings> ["if␣", a′, "␣then\n",
                               <pp-indent(|"␣␣")> b′, "\nend"]
```

Figure 9.8: Generate Stratego pretty printer from syntax templates

To pretty printing rules A simple set of recursive, bottom-up pretty printing rules can be generated from syntax templates. We generate these rules in the program transformation language Stratego. It can be seen in Figure 9.8 that a pretty printing rule consists of a number of components. The name of the rule, `prettyprint-Statement`, is composed from the name of the symbol. The rule matches the constructor `IfThen` with two arguments. The number of arguments is equal to the number of placeholders in the syntax template.

When the rule matches, child nodes are pretty printed by (recursively) invoking (other) pretty printing rules. Then, the text for all elements of the template is concatenated, while the text for child nodes is indented by the amount the respective placeholder is indented in the syntax template. In Chapter 11 we revisit the generation of pretty printers from syntax templates, covering topics such as the **`wrap`** and **`anchor`** options.

Chapter 10

Template Language — Evaluation

10.1 Introduction

In this chapter we evaluate the expressiveness of the template language developed in the previous chapter. We perform this evaluation by inspecting a syntax definition using both SDF and syntax templates in two languages: the tax-benefit rule modeling language that resulted from the the conversion in Chapter 5, and Mobl, a language for mobile web application development.

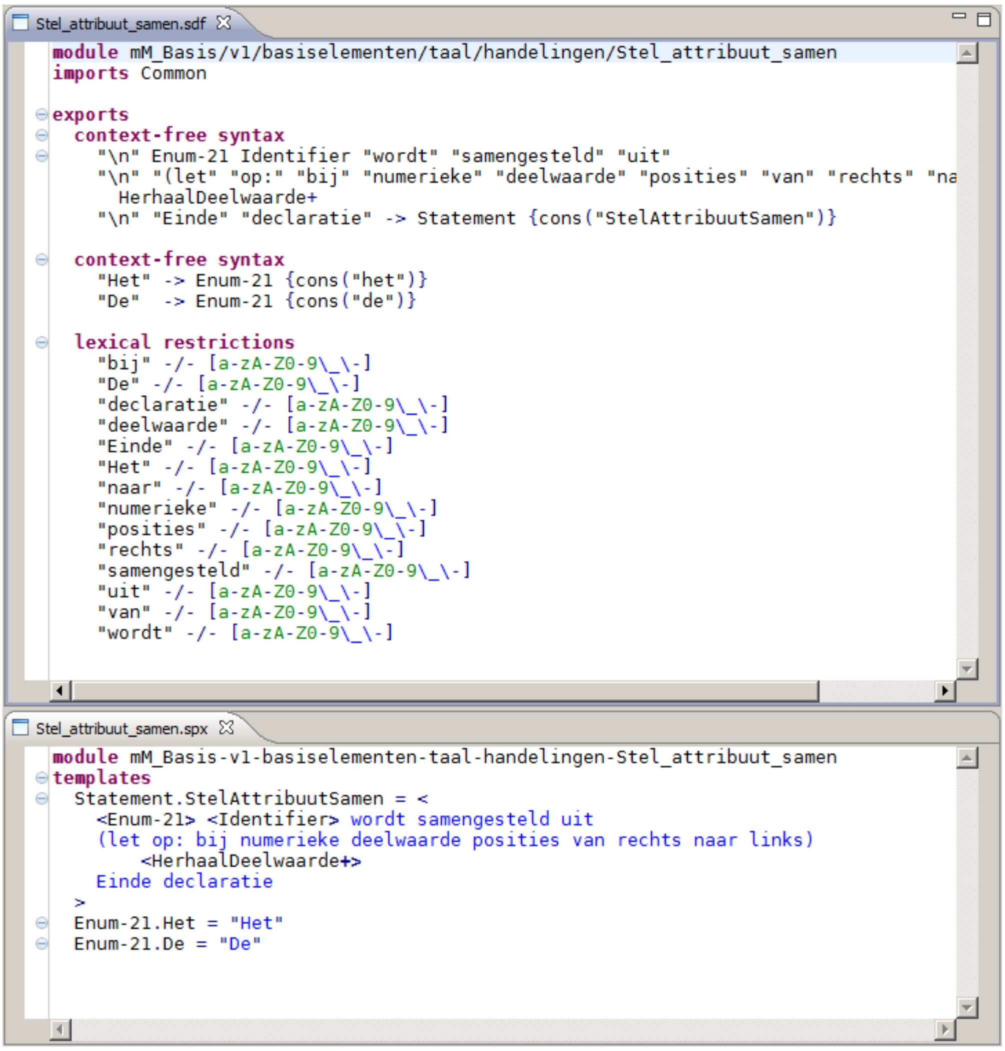
10.2 Use with the Tax-Benefit Modeling Language

10.2.1 Conversion

We changed the target language of the conversion we implemented in Chapter 5 from SDF to the template language we implemented in Chapter 9. As such, we acquired completion templates and a basic pretty printer for all language constructs, without writing any additional transformations specific to the conversion from Cheetah to Spoofax. Adapting the conversion was easy, because the structure of syntax templates and SDF productions is identical. Sanitizing steps, such as the tokenization of `GLEText` into separate keywords, could be dropped because they are performed by the template language. Indeed, the way language constructs are stored in Cheetah is closer to the syntax templates of the template language than to SDF productions, thereby reducing the effort required for the conversion. The models converted in Chapter 6 could be parsed using the grammar generated from the syntax templates without modification.

10.2.2 Results and discussion

An example of the brevity of a large language construct defined in the template language, versus the same language construct defined in SDF, is shown in Figure 10.1. A less extreme example, shown in Figure 10.2, still shows an increased legibility (less syntactic noise due to quotes, in particular), and that is in addition to the code reduction because pretty printer and completion templates can be generated from the syntax template. In both figures the SDF productions have been line wrapped by hand: the generator outputs them on a single line.



```
module mM_Basis/v1/basiselementen/taal/handelingen/Stel_attribuut_samen
imports Common

exports
  context-free syntax
    "\n" Enum-21 Identifier "wordt" "samengesteld" "uit"
    "\n" "(let" "op:" "bij" "numerieke" "deelwaarde" "posities" "van" "rechts" "na
      HerhaalDeelwaarde+
    "\n" "Einde" "declaratie" -> Statement {cons("StelAttribuutSamen")}

  context-free syntax
    "Het"  -> Enum-21 {cons("het")}
    "De"   -> Enum-21 {cons("de")}

  lexical restrictions
    "bij" -/- [a-zA-Z0-9\_\-]
    "De" -/- [a-zA-Z0-9\_\-]
    "declaratie" -/- [a-zA-Z0-9\_\-]
    "deelwaarde" -/- [a-zA-Z0-9\_\-]
    "Einde" -/- [a-zA-Z0-9\_\-]
    "Het" -/- [a-zA-Z0-9\_\-]
    "naar" -/- [a-zA-Z0-9\_\-]
    "numerieke" -/- [a-zA-Z0-9\_\-]
    "posities" -/- [a-zA-Z0-9\_\-]
    "rechts" -/- [a-zA-Z0-9\_\-]
    "samengesteld" -/- [a-zA-Z0-9\_\-]
    "uit" -/- [a-zA-Z0-9\_\-]
    "van" -/- [a-zA-Z0-9\_\-]
    "wordt" -/- [a-zA-Z0-9\_\-]
```

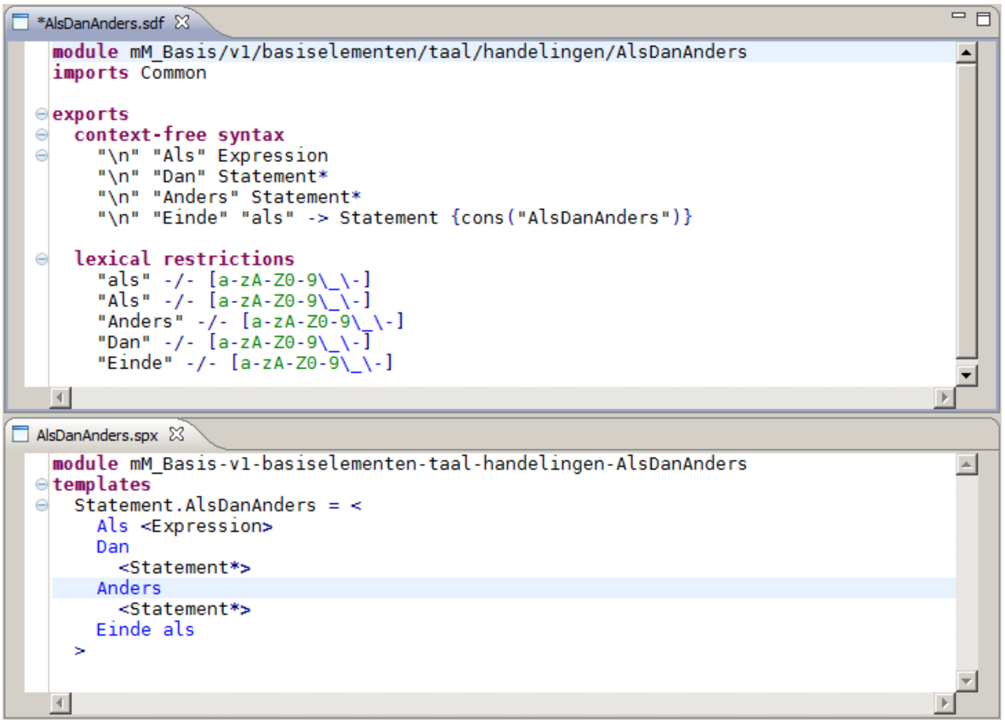
```
module mM_Basis-v1-basiselementen-taal-handelingen-Stel_attribuut_samen
templates
  Statement.StelAttribuutSamen = <
    <Enum-21> <Identifier> wordt samengesteld uit
    (let op: bij numerieke deelwaarde posities van rechts naar links)
        <HerhaalDeelwaarde+>
    Einde declaratie
  >
  Enum-21.Het = "Het"
  Enum-21.De = "De"
```

Figure 10.1: Language construct in SDF (top) and the template language (bottom)

If we look at the numbers we see that the generated SDF grammar consists of 2101 lines, of which 1805 are non-blank (in 129 files). The generated syntax templates for the same language consist of 997 lines, none of which are blank. Blank lines are not present because only one language construct is generated per file, and, contrary to SDF, the extra section `lexical restrictions` is not needed with syntax templates: it is assumed that no keyword may be immediately followed by an identifier, so the restrictions are generated automatically.

Even though the SDF grammar is twice as long as the template language syntax definition, all productions in the generated SDF are written on a single line, while the syntax templates are written on multiple lines if the language construct consists of multiple lines. Had the SDF productions been wrapped to a reasonable line length, then the magnitude of the difference in lines would have been larger.

The size of the SDF grammar can be attributed partially to the lexical restrictions,



```
module mM_Basis/v1/basiselementen/taal/handelingen/AlsDanAnders
imports Common

exports
  context-free syntax
    "\n" "Als" Expression
    "\n" "Dan" Statement*
    "\n" "Anders" Statement*
    "\n" "Einde" "als" -> Statement {cons("AlsDanAnders")}

  lexical restrictions
    "als" -/- [a-zA-Z0-9\_\-]
    "Als" -/- [a-zA-Z0-9\_\-]
    "Anders" -/- [a-zA-Z0-9\_\-]
    "Dan" -/- [a-zA-Z0-9\_\-]
    "Einde" -/- [a-zA-Z0-9\_\-]
```

```
module mM_Basis-v1-basiselementen-taal-handelingen-AlsDanAnders
templates
  Statement.AlsDanAnders = <
    Als <Expression>
    Dan
      <Statement*>
    Anders
      <Statement*>
    Einde als
  >
```

Figure 10.2: Language construct in SDF (top) and the template language (bottom)

which take up one line per keyword with our converter. Additionally, we emit a lexical restriction for each keyword in a language construct in the same file as the production, even though the same keyword may already have an identical lexical restriction in a different file. Technically it would have been easy to collect restrictions for the whole project and write the set to a single file. However, keeping the restriction for a keyword near the use of the keyword increases locality, which is important considering the syntax definition is intended to be maintained manually.

When limiting the line count to unique lines only, the magnitude of the difference in line count reduces significantly: 780 unique lines for SDF, versus 625 lines for the syntax templates. Common section headers present in every file, like **`templates`** and **`context-free syntax`**, explain a large part of the difference between the unique line count and the total line count. Although the template language syntax definition is only slightly smaller in terms of unique lines, only from the template language syntax definition a complete pretty printer and a set of completion templates can be derived. Additionally, verbose language constructs, such as those in Figure 10.1 and Figure 10.2, are arguably more readable in a syntax template.

The generated pretty printer has been tested on approximately 20,000 lines of DSL code in 124 files. For each of those files, parsing the pretty printed code resulted in the same AST as when parsing the original files. From this result we conclude that the pretty printer is correct. Using manual inspection we confirmed the prettiness of the pretty printed code.

```
context-free syntax
  "foreach" "(" LValue ":" Type "in" Exp ")" "{" Statement* "}"
      -> Statement {cons("For")}
```

(a) Original production in Mobl

```
templates
  Statement.For =
      <foreach ( <LValue> : <Type> in <Exp> ) { <Statement*> }>
```

(b) Intermediate (generated, unformatted) syntax template

```
templates
  Statement.For = <
    foreach(<LValue> : <Type> in <Exp>) {
      <Statement*; separator="\n">
    }
  >
```

(c) Syntax template after manual tweaking

Figure 10.3: From SDF production to syntax template

10.3 Use with Mobl

10.3.1 Introduction

Mobl [23] is a DSL for the construction of mobile web applications. It features an extensive standard library, declarative specification of user interface, static type checking, and embedding of Javascript, CSS styling rules and HTML.

In this section we reimplement the Mobl language syntax using our template language to investigate whether the template language is sufficiently expressive for a reimplementation of the complete Mobl language syntax. The implications are that the SDF generated from the syntax templates must be equivalent to the original syntax definition. The completion templates must behave as was intended with the design of the template language, and the pretty printer must be able to output correct, pretty code.

10.3.2 Conversion

To convert the Mobl syntax definition to syntax templates, we have added a converter to the SDF editor in Spoofax. It takes an SDF grammar as input, and outputs syntax templates, with each implicit `LAYOUT?` symbol replaced by a single space character. Within a few hours we formatted the 343 unformatted syntax templates, by inserting line breaks and indentation into 64 multi line language constructs, and adapting layout throughout the language using search-replace. An example of each of the three phases of the conversion (initial, intermediate, final) is displayed in Figure 10.3. Because the separator of a list element is specified whenever the list is used, we factored placeholders such as `<Statement*>` out into productions such as `Statements = <<Statement*; separator="\n">>`, so that we had to specify the separator option 2, instead of 55 times. Additionally, we refactored one production to three separate productions, because it employed the *alternative* operator, which is deprecated in SDF, and (by design) not present in the template language.

10.3.3 Observations

General

We found that the template language worked well on Mobl. After some minor fixes the syntax templates resulted in a syntax definition that could be used to parse the example code included with the Mobl project. Overall, the template language reduced the combined size of the syntax specification for Mobl from 1565 lines to 1163 lines, while delivering a complete pretty printer, and a complete set of completion templates. Although the pretty printer did not format certain constructs as well as we would have liked, the pretty printer did result in reasonably pretty, and syntactically correct code. The generated completion templates initially had an issue with the *meta annotations* present in Mobl, making them less suitable for their purpose. The various issues are covered in detail below.

SDF attributes

In the Mobl syntax, certain productions are marked as deprecated or rejected. The evaluated incarnation of the template language did not respect those attributes: deprecated constructs would sometimes be preferred by the generated pretty printer over their non-deprecated counterpart (depending on the order of the syntax templates), and completion templates and pretty printer rules were generated for reject productions, even though the language described by the reject production is, by definition, not part of the language described by the complete syntax definition. To solve this issue we changed the template language to not generate completion templates or pretty printer rules for reject productions, and to not generate completion templates for deprecated productions, while sorting pretty printing rules for deprecated productions to have a lower precedence than the pretty printing rules for non-deprecated productions.

10.3.4 Evaluation of the generated SDF

Keywords containing special characters

The grammar of the Mobl language as specified by our template language is slightly more permissive than the original Mobl grammar, due to keywords that contain special characters, such as `@<javascript>` and `@doc`, which get tokenized by the SDF generator to `"@<" "javascript" ">"` and `"@" "doc"`, so that layout is allowed between these tokens. Hence, we revisited this design decision: By allowing an empty sequence of escape characters as a *zero-length space* symbol, `LAYOUT?` can be inserted without inserting a space character in the pretty printer and completion templates. With such an escape present, we proceeded to remove most of the automatic tokenization, while making the remaining tokenization configurable (the **`tokenize`** option in Figure 9.4). Although this solution, using a global configuration option, hinders language composition, we believe that it is adequate for the time being. In the future this option (and other options) should be scoped, to allow different configuration of different sublanguages.

```
@service PUT /:name
function put(req : Request, res : Response) { /* ... */ }
```

(a) Function with a meta annotation in Mobl

```
templates
  FunctionDef.FunctionNoReturnType = <
    <MetaAnnos>
    function <QId>(<FArg*; separator=",␣">) {
      <Statements>
    }
  >
```

(b) Syntax template for this language construct

Figure 10.4: Meta annotations in Mobl

10.3.5 Evaluation of the generated completion templates

Mobl meta annotations

Initially, syntactic completion in our evaluation was suboptimal due to the *meta annotations* (Figure 10.4) present in Mobl at the start of many language constructs. Because the placeholder for these annotations is on a separate line, completion templates that produce a blank line before the language construct were generated. This behavior is likely not expected by the user, because these annotations are rarely used in Mobl. We corrected this by introducing the **`hide`** option to suppress the placeholders for annotations from the completion templates for many language constructs. The templates for annotations can be invoked separately, where desired.

10.3.6 Evaluation of the generated pretty printer

Correctness

To verify basic correctness of the pretty printer we parsed all Mobl library and sample code, pretty printed the resulting AST, parsed the pretty printed code, and compared the ASTs. For each of the 75 files (7,800 lines of code in total), the AST of the original file was equal to the AST of the pretty printed file, which suggests the pretty printer is correct. Again, prettiness of the pretty printed code has been confirmed by manual inspection of a sample of the pretty printed code.

Block/compound statements and if-else chains

A major limitation of the generated pretty printer is its lack of support for pretty printing block statements nested in control structures in anything but the GNU coding style [53]. The syntax of such a block statement in Mobl is shown in Figure 10.5, and various common ways of formatting such a statement are displayed in Figure 10.6. Only the leftmost coding style of Figure 10.6 — the GNU coding style — can be produced by our generated pretty printer for this block statement. This is due to the fact that the pretty printer operates in a bottom-up manner: first, the block statement is pretty printed, indenting the statements within, and then this opaque block of text is substituted for the `<Statement>` placeholder in the control structure, indenting it once

```
templates
  Statement.If = <
    if(<Exp>)
      <Statement>
    else
      <Statement>
  >
```

```
templates
  Statement.Block = <
    {
      <Statements>
    }
  >
```

Figure 10.5: Block/compound statements in Mobl

```
  if (exp)
    {
      // ...
    }
  else
    {
      // ...
    }
```

```
  if (exp)
  {
    // ...
  }
  else
  {
    // ...
  }
```

```
  if (exp) {
    // ...
  }
  else {
    // ...
  }
```

```
  if (exp) {
    // ...
  } else {
    // ...
  }
```

Figure 10.6: GNU coding style (left) and other common coding styles (right three)

```
  if (dx == -1 && dy == 0)
    (dx, dy) = (0, 1);
  else if (dx == 1 && dy == 0)
    (dx, dy) = (0, -1);
  else if (dx == 0 && dy == -1)
    (dx, dy) = (-1, 0);
  else
    (dx, dy) = (1, 0);
```

(a) Original

```
  if(dx == -1 && dy == 0)
    (dx, dy) = (0, 1);
  else
    if(dx == 1 && dy == 0)
      (dx, dy) = (0, -1);
    else
      if(dx == 0 && dy == -1)
        (dx, dy) = (-1, 0);
      else
        (dx, dy) = (1, 0);
```

(b) “Pretty” printed

Figure 10.7: If-else chain in Mobl

again because of the indentation of the `<Statement>` placeholder. Additionally, as shown in Figure 10.7(a), developers generally do not add an extra indentation level for every else-if block in a chain of if-else statements. Our pretty printer does, however, as Figure 10.7(b) illustrates. Ironically, the existing pretty printing infrastructure in Spoofax — GPP and BOX — do not handle these cases any better. So, although the limitation is severe, it does not put us behind the competition. In Chapter 11 we discuss these limitations and provide a solution.

10.4 Conclusion

As a first case study, we changed the target language of the Cheetah-Spoofax conversion from SDF to the template language. We could simplify the Cheetah-specific conversion because syntax templates have a structure similar to the syntax definition as stored in the Cheetah system. The syntax templates resulting from the adapted conversion were usable to parse the program text we converted in Chapter 6. Overall, the

Metric	SDF	PP	ESV	Total	Template language
Tax-benefit rule modeling language					
Lines	2101	n/a*	n/a*	2101	997
Non-blank/comment lines	1805	n/a*	n/a*	1805	997
Unique lines	780	n/a*	n/a*	780	625
Mobl					
Lines	1208	194†	163†	1565	1163
Non-blank/comment lines	854	193†	96†	1143	1072
Unique lines	753	171†	53†	977	820

∗ Not generated from the Cheetah system
† Not functional/complete

Figure 10.8: Template Language Evaluation

syntax definition in the template language was both smaller than the equivalent SDF definition, and it let us derive a complete, correct pretty printer and a set of completion templates.

Our second case study was an implementation of a conversion of the syntax of Mobl from SDF to the template language. The first stage of this conversion was an automatic rewriting of SDF to unformatted syntax templates. The second stage consisted of a number of manual and semi-automatic (i.e., search and replace) modifications of those syntax templates to beatify the pretty printed code and completion templates to a reasonable level. In this second case study, we found that SDF attributes such as `deprecated` and `reject` were not yet supported by the template language. We added support for those. Additionally, we found that the template language syntax definition is more permissive for keywords such as `@<javascript>`, which consist both of letters and punctuation, and suggested a solution. Many completion templates in Mobl started with an empty line: space for a Mobl meta annotation. Because meta annotations are rarely used in Mobl, it would be better if they were not present in the completion templates, so we added the `hide` option, which can be used to suppress a placeholder from the completion template, to the template language. The generated pretty printers have problems with block/compound statements and if-else-if chains, but existing pretty printing tools in Spoofax do not handle these cases either. In Chapter 11 we discuss these limitations of the generated pretty printers and provide a solution. Overall, the conversion of Mobl went well: besides the issues mentioned above, the generated SDF was sufficient to parse all example Mobl programs; the generated pretty printer could pretty print all of them, and generated completion templates worked as planned after the introduction of the `hide` option.

The reductions in code size for syntax definitions are summarized in Figure 10.8. The reductions are lower bounds, as for neither the Capgemini DSL nor Mobl the traditional pretty printer and the completion templates are complete, while syntax templates do result in a complete pretty printer and a complete set of completion templates.

Chapter 11

Generating Pretty Printers from Syntax Templates

11.1 Introduction

The recursive bottom up pretty printing rules we generated for syntax templates in Chapter 9 do not support the template language `wrap` and `anchor` options. The existing Spoofax pretty printing back-end BOX, however, does feature word wrapping, which is necessary for the `wrap` option. In Chapter 10 we observed that pretty printing of block/compound statements — both in our pretty printer and in BOX — is suboptimal: none of the popular styles for formatting block/compound statements can be produced. Therefore, in this chapter we present a new BOX operator, and a mapping from syntax templates to Stratego rules that generate a BOX AST. Together, these contributions add `wrap` and `anchor` support to the template language, and support for common styles of formatting block/compound statements to both the template language and other languages that use BOX as pretty printing back-end.

11.2 Overview of the BOX Formatting Language

BOX [56] is an intermediate language to describe the layout of text using boxes. Several formatting operators are present, among which `H` and `V` for horizontal and vertical formatting of sub-boxes, `HV` for horizontal formatting with line wrapping (inconsistent line breaking), and `HOV` for conditional formatting depending on the line width (consistent line breaking). Merijn de Jonge generalized the `HOV` operator to `ALT` [14], which takes two arbitrary sub-boxes, and prints the first one if there is enough horizontal space left, or the second one otherwise. Additionally, table based formatting is supported through the alignment operators `A` and `R`. These can be used to align, for example, the assignment operators in a list of assignments. Of these operators, the BOX dialect shipped with Spoofax supports `H`, `V`, `HV`, `ALT`, `A` and `R`, while the CWI/IMP dialect uses `HOV` instead of `ALT`, and additionally has a `G` grouping operator and an `I` indentation operator [1]. The behaviour of some formatting operators is displayed in Figure 11.1.

Many of the operators take parameters. For example, the `H` operator takes a *hs* parameter specifying the horizontal space between the sub-boxes. The `V` operator takes

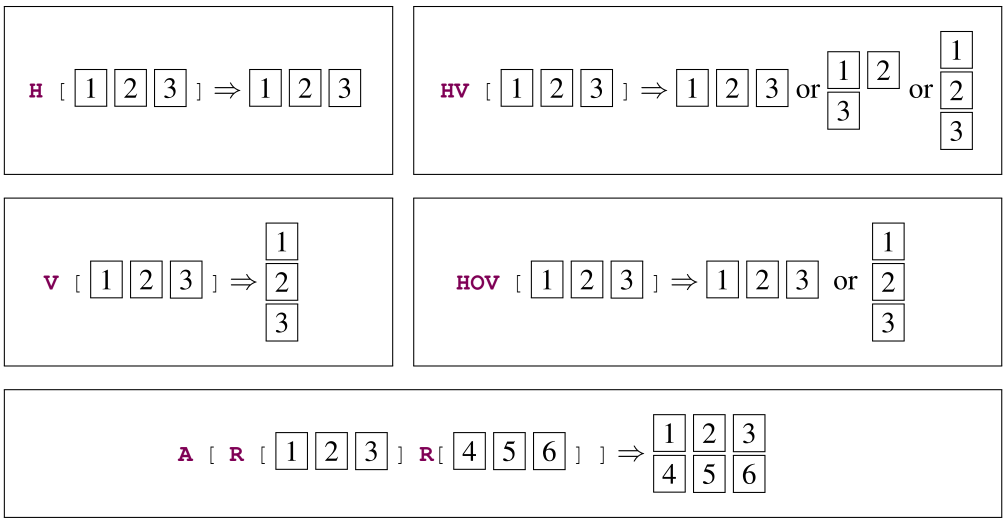

H [1 2 3] ⇒ 1 2 3
HV [1 2 3] ⇒ 1 2 3 or 1 2 3 or 1 2 3
V [1 2 3] ⇒ 1 2 3
HOV [1 2 3] ⇒ 1 2 3 or 1 2 3
A [R [1 2 3] R[4 5 6]] ⇒ 1 2 3 4 5 6

Figure 11.1: BOX formatting operators

an *is* parameter specifying the indentation of the sub-boxes (except the first), and a *vs* parameter, which specifies the vertical space between sub-boxes. An industrial case related to COBOL inspired Mark van den Brand et al. to add a *ts* (tab stop) parameter, which inserts spaces until the next box is at the specified column number [57]. The *ts* parameter is not available in Spoofax.

11.3 BOX Integration in Spoofax

The Spoofax language workbench integrates [31] the BOX formatting language and the generic pretty printer GPP [13]. By default, on each build, the tool *ppgen* generates a working set of GPP rules from the syntax definition, which do not include any formatting operators. The language developer can then override rules from this file with rules that include formatting operators. A rule is overridden by copying it from the generated file and inserting formatting operators into it. Every time the syntax of a language construct is changed, the change needs to be repeated in all overridden rules. To build a complete pretty printer many rules must be overridden, because the default of a single space character between all elements is sufficient only for small subsets of a grammar (e.g., expressions).

11.4 Block/compound Statements

It is not possible to pretty print code as in the rightmost two examples of Figure 11.2, because the first line of the compound statement ("{") is placed to the right of the remaining lines, which is not possible with a `V` box. As such, only the first coding style (GNU) and the second coding style can be produced, while only the first can be produced if we desire acceptable results when a non-compound statement is nested in the control flow statement.

```
if (exp)
  {
    // ...
  }
else
  {
    // ...
  }
```

```
if (exp)
{
  // ...
}
else
{
  // ...
}
```

```
if (exp) {
  // ...
}
else {
  // ...
}
```

```
if (exp) {
  // ...
} else {
  // ...
}
```

Figure 11.2: GNU coding style (left) and other common coding styles (right three)

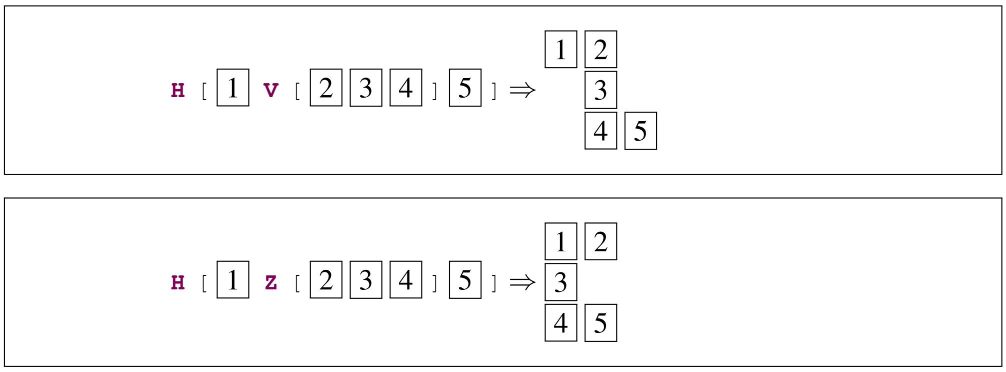


Figure 11.3: The difference between V and Z boxes

11.5 The Z Formatting Operator

There are two methods to produce the rightmost two coding styles of Figure 11.2 using BOX. The first method is a transformation that lifts the first line out of certain, designated boxes, and puts those in the preceding H box. This lifting transformation is complicated, as it needs to take the first sub-box of the designated box, and add it after the last sub-box of the preceding H box, which may live in the parent, grandparent, or further up the tree, if it exists at all. The second method adds a Z operator that flows back the text to the left margin after the first sub-box. An implementation of the Z operator takes only a few lines of Stratego code, and is nearly identical to the V operator: only the left margin of the sub-boxes differs. This difference is illustrated in Figure 11.3. To set the left margin that is employed by the Z operator, we introduce an I operator for indentation, similar to the I operator present in the CWI/IMP BOX dialect.

11.6 Mapping Syntax Templates to the Z Operator

In Figure 11.4 we see how syntax templates map to BOX expressions. A syntax template maps to an unqualified list of boxes. Within a syntax template, an unindented line maps to an H box, while the H box for an indented line is wrapped in an I box to set the new indentation level for nested Z boxes. For each placeholder the list of boxes for the child node is generated first. This list is put in a V box if the anchor option is

```
templates
  Statement.IfElse = <
    if (<Exp>) <Statement>
    else <Statement>
  >
  Statement.Compound = <
    {
      <Statements>
    }
  >
```

```
[
  H hs=0 [ "if␣(" Z <Exp> ")␣" Z <Statement> ]
  H hs=0 [ "else␣" Z <Statement> ]
]
[
  H hs=0 [ "{" ]
  I is=2 [ H hs=0 [ Z <Statements> ] ]
  H hs=0 [ "}" ]
]
```

Figure 11.4: Mapping from syntax template to BOX expression

```
IfElse(
  condition1,
  Compound(actions1),
  IfElse(
    condition2,
    Compound(actions2),
    Compound(actions3)
  )
)
```

```
if (condition1) {
  actions1
}
else if (condition2) {
  actions2
}
else {
  actions3
}
```

Figure 11.5: Pretty printing an if-else chain

present, and in a **Z** box otherwise. (Or in an **HV** box or **HZ** box respectively, if the **wrap** option is also present.)

Figure 11.5 demonstrates how two nested if-else statements are pretty printed according to Figure 11.4. Observe how $actions_2$ and $actions_3$ are aligned to $actions_1$, because the **Z** boxes generated for the compound statements let the text flow back to the left margin after each opening bracket. With traditional **V** boxes, the compound statements around $actions_2$ and $actions_3$ would have been pushed to the right of $condition_2$.

11.7 Conclusion

We successfully implemented an improved pretty printer generator for syntax templates by targeting a modified version of the BOX language. The existing dialects of BOX have no mechanism to let text flow back to the left margin, which forces the BOX user to perform non-linear transformations that move language elements across boxes, or to accept suboptimal formatting. We introduce a **Z** operator representing a vertical box that flows back the text to the left margin after the first sub-box. Using the **Z** operator we are able to describe a simple, one-to-one mapping from a syntax template to a partial BOX AST. As such, we were able to simplify our pretty printer generator, and at the same time add support for the **wrap** and **anchor** options.

Chapter 12

Related Work

12.1 Unified Syntax Specifications

There are a number of current syntax specification approaches that aim to unify the specification of parsing and unparsing.

SYN [6] aims to be one syntax definition language for the specification of ASTs, lexical analysis, parsing and pretty-printing. Its notation is similar to BNF, extended with a sublanguage for the specification of a lexical analyzer, and operators `h` (horizontal composition), `hv` (inconsistent line breaking), and `hov` (consistent line breaking) for the generation of a pretty printer. SYN has been implemented in Standard ML. Because the SYN compiler translates the syntax definition to input for the tools ML-Lex and ML-Yacc (ML implementations of the well known UNIX tools lex and yacc), a syntax definition in SYN faces the limitations of separate scanner and parser and LALR(1) parsing.

Extended SDF [48] is an extension of SDF that embeds other specification languages. An important application of the work is the embedding of PP pretty printing rules in SDF attributes. Although this improves the locality of the different syntax definitions, it does not solve redundancy, as elements of the syntax are present both in the SDF, and in the attached pretty printing rule.

More recently, Rendel and Ostermann [49] propose partial isomorphisms for invertible computation, and use these to implement a combined parser/pretty printer library in Haskell. Productions are specified using invertible combinators used for parsing, unparsing, and abstract syntax (de)construction.

These approaches differ from our approach in their use of explicit operators that specify layout and formatting. By using templates, we provide a concise syntax that forgoes the use of operators and uses plain whitespace instead, while still being sufficiently expressive for our case study with two complete DSLs. In addition, they also do not consider completion templates, which are necessary for template-based editing.

12.2 Pretty Printing Specifications

Oppen [42] describes an algorithm for pretty printing, which operates on an input consisting of non-blanks, blanks and the special brackets ⟦ and ⟧. The brackets denote blocks, and the algorithm tries to break as few blocks as possible while satisfying the line width constraint. Hughes [25] discusses a number of features of a Haskell pretty printing combinator library, such as horizontal and vertical composition, and indentation (or nesting). The focus of both Oppen and Hughes is on breaking lines at acceptable positions. Hughes extends this with separate combinators for horizontal or vertical composition. This configurability is taken further in the work of De Jonge [12], which combines the BOX formatting language [56] with pretty printing rules that map AST nodes to BOX expressions. We extend those pretty printing rules, and the BOX formatting back-end, with the `z` operator, which aids with formatting chains of if-else statements, and similar language constructs.

StringTemplate [44] is a template language for code generation and web development that enforces model-view separation. It only focuses on the generation of code, not parsing, but it has shown that templates are an effective tool to rapidly output well-formatted code, even though it is not considered a pretty printer. It has been a source of inspiration for the syntax and semantics of our syntax templates.

12.3 Template-Based Editing

Many early language workbenches used a template-based editing paradigm in structure editors. Examples include Centaur [5] and the Synthesizer Generator [50]. While these systems faced the same problem of having to specify both abstract and concrete syntax, they did not have the problem of specifying both a parser and an unparser.

Hybrid textual/structure editors make it possible to switch to a text editing mode for a part of a program. Systems used to specify these editors do have the added dimension of parsing and unparsing, where they need a specification of formatted concrete syntax and a specification that specifies how to parse concrete syntax independent of the layout. While there have been different ways to address the issue, there has not been a solution that unifies the specification of all syntactic aspects. Examples of hybrid systems include the Programming System Generator (PSG) [2], Pregmatic [55], and the ASF+SDF Meta-Environment [35].

Pregmatic [55] specifies syntax using as part of attribute grammars. The grammar formalism does not include a formatting specification: instead, unformatted templates are generated from the grammar. The user can then edit the layout in those templates to format them as desired.

The Meta-Environment [35] is based on SDF for syntax definition, and originally used the Generic Syntax-directed Editor (GSE) [15] as a hybrid editor. Contrary to many earlier structure editors it does not use pretty printing to convert abstract syntax into concrete syntax. Instead it maintains a two-way mapping between the text the user entered and the AST, so that a pretty printer is not needed during editing, and the user has full control over the layout of the program.

Template-based textual editors have text editing as their principal mode of operation, but can provide textual templates for editing. Examples of tools to create these editors include MontiCore [20], Xtext [17], and our own Spoofax [34]. Each of these systems has so far used a separate specification of syntax for parsing, pretty printing, and completion templates. As part of our work we implemented a template language for Spoofax, showing these aspects can be combined.

Chapter 13

Conclusions and Future Work

13.1 Summary

Cheetah & Spoofax

In Chapter 2 and Chapter 3 we looked at the architecture of the Cheetah system, and compared it with Spoofax. The most outstanding difference between the two language workbenches is the different editing paradigm. Whereas Cheetah uses a structure editor, without support for cut & paste or free form editing, Spoofax leverages the Eclipse text editor, combined with SGLR parsing to convert the plain text to an AST. The use of parser technology enables Spoofax to use plain text as the persistent representation of DSL code, which allows optimal use of text-based tooling, such as version control systems.

Although most editor services are present both in Cheetah and Spoofax, the implementation of many is lacking in Cheetah. Code folding and an outline view are not present for the DSL editor, as it operates on a single method at a time; instead, code folding and an outline view are implicitly provided by the tree the user has to navigate to reach the DSL editor. Error markers, although present in Cheetah, have limited value because they are only implemented for a few specific type errors. Clearly, the code required for the implementation of a DSL-specific error marker outweighs the benefits of such error markers. In Spoofax, many of the editor services that have to be coded in Java for the Cheetah system, can be specified declaratively in the editor service language, or concisely in Stratego, thus greatly reducing the effort required to develop them.

Code generation in Cheetah is also performed in Java, using libraries to represent source code files with protected regions, and to perform template-based code generation. As Java does not contain a multi line string literal, such templates are hard to modify as quotes and concatenation operators must be introduced at line endings. The Stratego DSL for program rewriting is more suitable for code generation than Java, because it has an indentation-safe string interpolation construct for template-based code generation, and it allows concise specification of AST rewrite rules for code generation using AST rewriting and unparsing.

Migration of the Tax-Benefit Rule Modeling DSL to Spoofax

In Chapter 4 we hooked Stratego up to native SAX-based XML parsing libraries to be able to reduce the development/feedback cycle when developing the conversions from Cheetah to Spoofax in Chapter 5 and Chapter 6. We were able to realize a speedup of two orders of magnitude, while at the same time making the parser more conformant.

Chapter 5 describes the migration from the DSL specification in the Cheetah system to SDF. This transformation was straightforward, although we had to overcome a number of challenges to disambiguate the grammar, and to make the parser sufficiently performant. In particular, we investigated how to deal with line endings as fiducial symbol (i.e., statement separator), without forcing the user to write line continuation constructs at seemingly arbitrary places.

For the migration of DSL code in Cheetah to plain text in Chapter 6 we had to resolve the prototypal inheritance present between model elements in Cheetah, using our XML library in combination with Stratego code to collect all child elements of a single model. With all information collected, the transformation reduced to a straightforward tree traversal, with the lack of restrictions on the characters in identifiers in Cheetah being the largest challenge.

The main remaining asset specific to the tax-benefit rule modeling DSL in Cheetah are the code generators. These code generators consist of approximately 70,000 lines of Java code. In Chapter 7, we describe an experiment to determine whether reuse of the existing code generators is an option. The results indicate that many assumptions in the transformation strategies must be fixed before such reuse is possible. The alternative, a migration of the code generators to Stratego, looks prohibitively hard to automate because of the different paradigms of the two languages. Hence, a manual conversion of the 70,000 lines would be the recommended course of action for a migration to idiomatic Stratego code.

The Template Language

The structure editor in Cheetah offers the user a complete set of language elements at each insertion point. This editing paradigm is also called template-based editing. Because the tax-benefit rule modeling DSL is rather verbose, it is crucial for Spoofax to properly support template-based editing. Template-based editing relies crucially on two things: first, runtime support for generated template-based editors, and second, concise specification of the templates for a particular DSL. Traditionally, Spoofax relies on the editor service language for the specification of completion templates, the pretty printing language for the specification of a pretty printer, and SDF for the specification of the grammar of a language. Each of these specifications includes the concrete syntax of the language, albeit in a slightly different form every time. Such redundant specifications increase the maintenance effort required to keep the system consistent.

To eliminate this redundancy, we designed and implemented the template language, which allows productions to be specified using syntax templates that include layout information, in Chapter 9. We derive completion templates and a complete pretty printer from these syntax templates. As such, we completely eliminate the redundancy between the three formalisms, thus reducing the maintenance effort and increasing the quality of template-based editors in Spoofax.

We evaluated the template language in Chapter 10 with two case studies. The first case study encompasses the tax-benefit rule modeling DSL, while the second case study looks at an implementation of Mobl, a DSL for the development of mobile web applications, using syntax templates. Although these case studies uncover some issues, the general principle appears to work: we achieve a reduction in specification size for both the tax-benefit rule modeling DSL, and Mobl, while at the same time adding a complete pretty printer and a complete set of completion templates to both.

Based on the evaluation of the generated string-based pretty printer in Chapter 10, and the desire to implement a feature such as word wrapping, we discuss a simple addition to the BOX formatting language that enables us to convert syntax templates one-to-one to pretty printing rules, in Chapter 11. We propose to add a `Z` operator to BOX, that stands for a box for which all sub-boxes, except the first, are aligned to the left margin, instead of the character position of the first sub-box. The left margin is defined by the `I` operator, which we added to the Spoofax BOX dialect for this purpose. Using the new `Z` operator it is possible to describe the layout of, for example, a chain of if-else statements, without the need for complicated transformations that move language constructs partially to different boxes. Additionally, the `Z` operator enabled us to convert syntax templates one-to-one to pretty printing rules, thus also adding support for the `wrap` and `anchor` word wrapping features of the template language.

Runtime Support for Template-Based Editors

In Chapter 8 we improved runtime support for template-based editing in Spoofax by introducing the Eclipse *linked mode* feature, known from, for example, the rename refactoring in the Eclipse Java Development Tools, to all Spoofax editors. Linked mode is a template-based mode of the Eclipse editor, where placeholders are marked visually with a bounding box, and the tab key is overridden to cycle through these placeholders. This mode allows the user to quickly substitute actual content for the placeholders in a completion template.

Because template-based editing relies crucially on the presentation of an accurate set of completion templates to the user whenever they invoke content assist, we investigated a way to encode the syntactic categories at the cursor location in the (ambiguous) AST. We apply a grammar generation technique to generate productions that parse the marker inserted by the Spoofax runtime to any symbol in the grammar. The ambiguous AST is then disambiguated by the runtime, while it records the syntactic categories encoded in the names of the ambiguous AST nodes. These syntactic categories are used to select only relevant completion templates to present to the user. Although this technique relies on a parser capable of parsing ambiguous fragments, it does not require any modifications to the implementation of the parser. Therefore, it is not limited to the current SGLR parser implementation in Spoofax; it should be equally applicable to other implementations of SGLR and SGLL parsers.

13.2 Contributions

- To effectively develop a conversion of the tax-benefit rule modeling DSL from the Cheetah language workbench to Spoofax, we developed and contributed an XML parsing library to Spoofax, which is two orders of magnitude faster than parsing XML using JSGLR.
- We showed it is possible to perform a conversion of the tax-benefit rule modeling DSL from the Cheetah system to an SDF grammar and plain text files that are parseable using this grammar.
- We designed and implemented a syntax definition language based on syntax templates, which eliminates the redundancy between the specifications of grammar, pretty printer / unparser, and completion templates. We evaluated this template language on the tax-benefit rule modeling DSL and the Mobl DSL for mobile web application development, showing a reduction in specification size, combined with the addition of a set of completion templates, and a working pretty printer.
- We extended the BOX formatting language with a `Z` operator for a box with sub-boxes aligned to the left margin. We adapted our template language to use the `Z` operator, thereby showing its effectiveness for specifying pretty printing rules for language constructs such as chained if-else statements, which could not be formatted properly using GPP and traditional BOX operators such as the `V` vertical box.
- We developed an experimental, parser-agnostic technique to determine the syntactic categories at the cursor in parser-based text editors.

13.3 Conclusions

Research Question 1: Can a DSL, such as the tax-benefit rule modeling DSL of Capgemini, be migrated from a structure editor based language workbench, such as the Cheetah system, to a parser-based language workbench, such as Spoofax?

The first research question we put forward in Chapter 1 covers the migration of a DSL from the Cheetah system to Spoofax. Such a migration consists of different aspects: syntax of the DSL, semantics, code generators. We showed the syntax of the DSL can be migrated automatically (Chapter 5, Chapter 6). We discussed potential reuse of the code generators, which, in the case of Cheetah, also define the semantics of the DSL. We argued an automatic migration of the code generators is not feasible, while reuse might be possible, though only after numerous fixes to the code generators to rid them of assumptions about the implementation of AST nodes (Chapter 7). Therefore, we can give a mixed positive and negative answer to Research Question 1: yes, the syntax of the tax-benefit rule modeling DSL can be migrated automatically from Cheetah to Spoofax, but no, the code generators (and thus semantics) need to be ported by hand. Based on this result, we speculate that it may be possible to migrate other DSLs from a structure editor based language workbench to a parser-based language workbench.

Research Question 2: Can a single, declarative language unify the specification of parser, pretty printer, and templates for syntactic completion?

The second research question is whether it is possible to create a declarative language that eliminates redundancy between parser, unparser /pretty printer, and completion templates. We can answer this question positively, as we successfully designed, implemented, and evaluated such a declarative language (Chapter 9, Chapter 10).

Research Question 3: Can we improve runtime support for template-based editors, so as to make the set of presented templates both relevant and complete?

The third and last research question covers the runtime support for template-based editors. Good runtime support is of crucial importance for editing programs in a verbose DSL such as the tax-benefit rule modeling language. With the template language, our answer to Research Question 2, we created a single specification language for a template-based editor, so as to reduce the maintenance effort to keep the editor up to date. The set of completion templates, however, is not useful, if irrelevant templates are shown, or relevant templates are not shown. We applied a grammar generation technique, combined with DSL- and parser-agnostic runtime support, to determine the syntactic categories at the cursor location, and thus the completion templates relevant to the user (Chapter 8). Hence, we can answer Research Question 3 positively.

13.4 Discussion/Reflection

Migrating from Cheetah to Spoofax

We have shown the syntax for a DSL in Cheetah can be migrated automatically to Spoofax. In retrospect, it would have been more useful to spend this time on the development of a complete prototype, including code generation and DSL testing, for a small subset of the language, since it is unlikely that the complete infrastructure for a working product such as the tax-benefit rule modeling DSL will be swapped out, whereas the use of new tools on a new project is more likely.

The Template Language

Initially, we implemented the template language as an editor in the Spoofax distribution, separate to all other projects under development. One of those projects was SpoofaxLang[1], an editor for a language integrating SDF, Stratego and the ESV editor service language. By not integrating TemplateLang into SpoofaxLang, we would force the user to choose either TemplateLang or SpoofaxLang: it would be impossible to use both at the same time. Clearly, this scenario was undesired, so we set out to integrate the TemplateLang editor into the SpoofaxLang editor.

Although the integration was functional for a while, keeping it up to date with rapid developments on SpoofaxLang, as well as improving TemplateLang while it was

[1]`http://strategoxt.org/Spoofax/SpoofaxLanguage`

tightly coupled to SpoofaxLang, proved to be a large time sink. Therefore, we decided to separate TemplateLang, and linked it as a library to SpoofaxLang. This separation of TemplateLang and SpoofaxLang made it easier to test and develop TemplateLang, without having to rely on correctness of SpoofaxLang and/or the integration of SpoofaxLang and TemplateLang. Thus, in retrospect we could have saved time by only coupling the projects loosely (through a library) in the first place.

Engineering Challenges in Spoofax/SDF/Stratego

During the work for this thesis we acquired much experience with Spoofax, Stratego, and accompanying tools. As such, we found a number of points that could be improved in these tools, so as to make the implementation of large systems easier.

Foremost, in Stratego currently all strategies live in the same global namespace: there are no separate namespaces for libraries or imported modules, and there is no mechanism to hide a strategy that is internal to a library or module, apart from inlining it where it is used. While it is reasonably easy to prevent conflicts between names in your own code, this quickly gets harder as more developers are added to a project. Currently, the only solution is to use a naming scheme that encodes the library/module name in the strategy name.

A related issue in Stratego/Spoofax, is the lack of a clearly defined way to create reusable source code libraries. We were able to create a library for the template language, and use it in the SpoofaxLang editor, but the steps required to create this library required much inside knowledge of Spoofax & Stratego. Ideally, Spoofax shall be extended with a wizard in Eclipse to create a library project, and a simple mechanism to set up another Spoofax project to depend on that library.

Initially, Spoofax projects could only be tested through carefully crafted custom transformations, which invoke your own strategies on various inputs and check their actual output against some expected output. During this thesis project, however, the Spoofax Testing Language [32] has been introduced, which alleviates many of the testing annoyances, by introducing a standard, low threshold way to test transformations and editor services.

13.5 Future work

Migrating from Cheetah to Spoofax

Before a new model-driven project with Spoofax is started at Capgemini, it is recommended to create a prototype on a small subset of a DSL, while covering all required aspects, including, but not limited to: DSL testing, editor service testing, code generation, debugging, error markers, and continuous integration of Spoofax projects. The development of this prototype should preferably be performed by the developers who build the real product later on. We recommend to keep track of the speed at which developers pick up Spoofax, SDF and Stratego, since in particular the last one is said to have a steep learning curve. Essentially, the project to create the prototype should not only focus on the engineering aspects, but also on the usability aspects of the tool.

Runtime Support for Template-Based Editors

Our improvements to the runtime support for template-based editing in Spoofax build on the assumption that users want to see a complete set of relevant templates. An interesting research subject may be whether users really want a complete set of templates: it might be better to infer common language constructs from existing code, for example, and only show those by default. Another approach would be to generate and display completion templates only for multi line language constructs. This approach builds on the assumption that basic (arithmetic) expressions are similar in most languages, so that users do not need discovery for those small language constructs. A user study would have to be set up to evaluate various configurations.

An open area in our runtime support for template-based editors work, is combining syntactic and semantic content assist. For example, for a method call, the best user experience may be achieved by combining semantically relevant method names with the syntax for the method call, as specified by the syntax template. Currently, the syntax for the method call must be copied to the semantic content assist editor service, while the syntactic completion template generated from the syntax template must be disabled.

The Template Language

The template language would benefit from case studies on DSLs with a different design than the tax-benefit rule modeling DSL and Mobl. Most of the work to be done however, is in making the interaction with SDF more seamless. For example, we could integrate the converter that creates unformatted syntax templates from SDF productions into the template language, so that SDF productions at least result in a pretty printing rule and a completion template, although unformatted.

Another interesting extension of the template language would be the generation of a code formatter for Eclipse. That is, a pretty printer that modifies the layout of existing code without erasing comments. Similarly, a minimizer, as is typically used to compress Javascript files that are published on the web, could be generated from syntax templates for any DSL. Other features currently lacking from the template language, are: the specification of *kernel syntax* (explicit layout instead of implicit layout), case insensitive literals, `ast` attributes, and parameterized modules & symbols.

The BOX Formatting Language

It would be good to perform a comprehensive evaluation of the BOX operators we introduced in Chapter 11. In particular, it would be interesting whether BOX, extended with the `Z` and `I` operators, is now sufficient to pretty print a variety of DSLs that are radically different from Mobl and the tax-benefit rule modeling DSL.

Bibliography

[1] IMP formatting: How to make a formatter. `http://www.eclipse.org/imp/documents/impFormattingHowto.pdf`.

[2] R. Bahlke and G. Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):576, 1986.

[3] J. Bentley. Little languages. *CACM*, 29(8):711–721, 1986.

[4] R. Biddle and E. Tempero. Java pitfalls for beginners. *ACM SIGCSE Bulletin*, 30(2):48–52, 1998.

[5] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. *ACM SIGPLAN Notices*, 24(2):14–24, 1989.

[6] R. Boulton. *Syn: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing*. University of Cambridge, Computer Laboratory, 1996.

[7] M. Bravenboer. Connecting XML processing and term rewriting with tree grammars. *Utrecht University*, nov 2003.

[8] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[9] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1-2):123–178, 2005.

[10] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (XML) 1.0 (fifth edition). Technical report, W3C, nov 2008.

[11] P. Charles, R. Fuhrer, and S. Sutton Jr. IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 485–488. ACM, 2007.

[12] M. de Jonge. A pretty-printer for every occasion. *CoSET2000*, 2000.

[13] M. de Jonge. Pretty-printing for software reengineering. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 550–559. IEEE, 2002.

[14] M. de Jonge. *To Reuse or To Be Reused: Techniques for Component Composition and Construction*. PhD thesis, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, Jan. 2003.

[15] M. H. H. Dijkvan Dijk and J. W. C. Koorn. GSE, a generic syntax-directed editor. Technical Report CS-R9045, Centrum voor Wiskunde en Informatica (CWI), 1990.

[16] S. Dmitriev. Language oriented programming: The next programming paradigm, 2004.

[17] S. Efftinge and M. Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, 2006.

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

[19] M. Fowler. Language workbenches: The killer-app for domain specific languages?, 2005.

[20] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Monticore: a framework for the development of textual domain specific languages. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Companion Volume*, pages 925–926. ACM, 2008.

[21] D. Grune and C. Jacobs. A programmer-friendly LL (1) parser generator. *Software: Practice and Experience*, 18(1):29–38, 1988.

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

[23] Z. Hemel and E. Visser. Declaratively programming the mobile web with mobl. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011*, Portland, Oregon, USA, 2011. ACM.

[24] J. Hopcroft, R. Motwani, and J. Ullman. Automata theory, languages, and computation. *International Edition*, 2006.

[25] J. Hughes. The design of a pretty-printing library. *Advanced Functional Programming*, pages 53–96, 1995.

[26] S. Johnson. *YACC-yet another compiler-compiler*. Citeseer, 1978.

[27] S. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge Univ Pr, 2003.

[28] K. Kalleberg and E. Visser. Spoofax: An extensible, interactive development environment for program transformation with Stratego/XT. 2007.

[29] K. Kalleberg and E. Visser. Fusing a transformation language with an open compiler. *Electronic Notes in Theoretical Computer Science*, 203(2):21–36, 2008.

[30] L. C. L. Kats, K. T. Kalleberg, and E. Visser. Generating editors for embedded languages. integrating SGLR into IMP. In A. Johnstone and J. Vinju, editors, *Language Descriptions, Tools, and Applications (LDTA'08)*, Electronic Notes in Computer Science, pages 91–107. Elsevier, April 2008.

[31] L. C. L. Kats, K. T. Kalleberg, and E. Visser. Domain-specific languages for composable editor plugins. *Electronic Notes in Theoretical Computer Science*, 253(7):149–163, 2010.

[32] L. C. L. Kats, R. Vermaas, and E. Visser. Integrated language definition testing. enabling test-driven language development. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011*, Portland, Oregon, USA, 2011. ACM.

[33] L. C. L. Kats and E. Visser. Encapsulating software platform logic by aspect-oriented programming: A case study in using aspects for language portability. In C. Marinescu and J. Vinju, editors, *Proceedings of the Tenth IEEE International Working Conference on Source Code Analysis and Manipulation 2010*, 2010.

[34] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM.

[35] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2):176–201, 1993.

[36] P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano, October 1994.

[37] R. Lämmel, E. Visser, and J. Visser. The essence of strategic programming. *Unpublished manuscript available online http://www. program-transformation. org/Transform/TheEssenceOfStrategicProgramming*, 2002.

[38] S. McPeak and G. Necula. Elkhound: A fast, practical GLR parser generator. In *Compiler Construction*, pages 2725–2725. Springer, 2004.

[39] D. Megginson and D. Brownell. Simple API for XML (SAX). `http://www.saxproject.org/`, apr 2004.

[40] M. Murata. Hedge automata: a formal model for XML schemata. 1999.

[41] G. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Elipse IDE? *Software, IEEE*, 23(4):76–83, 2006.

[42] D. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(4):465–483, 1980.

[43] A. B. Pai and R. B. Kieburtz. Global context recovery: A new strategy for syntactic error recovery by table-drive parsers. *ACM Trans. Program. Lang. Syst.*, 2:18–41, January 1980.

[44] T. Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th international conference on World Wide Web*, pages 224–233. ACM, 2004.

[45] T. Parr and R. Quong. Adding semantic and syntactic predicates to LL (k): pred-LL (k). In *Compiler Construction*, pages 263–277. Springer, 1994.

[46] T. Parr and R. Quong. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[47] T. J. Parr. Enforcing strict model-view separation in template engines. In S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 224–233. ACM, 2004.

[48] N. Pouillard. Extending SDF. Technical report, 2004.

[49] T. Rendel and K. Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing. In *Proceedings of the third ACM Haskell symposium on Haskell*, pages 1–12. ACM, 2010.

[50] T. Reps and T. Teitelbaum. The synthesizer generator. *ACM Sigplan Notices*, 19(5):42–48, 1984.

[51] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In P. L. Tarr and W. R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 451–464. ACM, 2006.

[52] R. Snodgrass. Temporal databases. *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, pages 22–64, 1992.

[53] R. Stallman et al. GNU coding standards. 1992.

[54] M. van den Brand, P. Klint, and J. Vinju. The Syntax Definition Formalism SDF, 2007.

[55] M. van den Brand and A. Pregmatic. *A Generator for Incremental Programming Environments*. PhD thesis, 1992.

[56] M. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):1–41, 1996.

[57] M. G. J. van den Brand, A. T. Kooiker, J. J. Vinju, and N. P. Veerman. An architecture for context-sensitive formatting. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 631–634. IEEE, 2005.

[58] A. Van Deursen and P. Klint. Little languages: Little maintenance? In *SIGPLAN Workshop on Domain-Specific Languages*. Citeseer, 1997.

[59] E. Visser. Scannerless generalized-LR parsing. Technical report, Citeseer, 1997.

[60] E. Visser. Syntax definition for language prototyping. 1997.

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

[62] E. Visser. Meta-programming with concrete object syntax. In *Generative programming and component engineering*, pages 299–315. Springer, 2002.

[63] E. Visser and Z. el Abidine Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15:422 – 441, 1998. International Workshop on Rewriting Logic and its Applications.

[64] E. Visser et al. Building program optimizers with rewriting strategies. In *Proceedings of the International Conference on Functional Programming (ICFP'98*. Citeseer, 1998.

[65] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. *Software Language Engineering, SLE*, 2010.