A language-independent framework for syntactic extensibility

Felix Rieger

Fachbereich Mathematik und Informatik
Philipps-Universität Marburg

A thesis submitted for the degree of
Bachelor of Science

Supervisors:
Prof. Dr. Klaus Ostermann
Sebastian Erdweg, MSc.

June 2012

Abstract Many programming languages offer no facilities to extend their syntax. Programmers want to have syntactically extensible languages to make design patterns become language features. Syntactic extensions can also be defined to embed domain-specific languages in a safe way while keeping their syntax. SugarJ provides these features, but only to Java programmers. Syntax extensions can be used like libraries and can also be composed. We identify Java-dependent aspects in SugarJ, find general abstractions and build an abstract language library. Implementing this library as a plugin allows programmers to use SugarJ for their programming language. We demonstrate this with plugins for Java and Prolog.

Contents

1 Introduction **3**

2 Background **5**
2.1 Domain-specific languages and syntactic sugar 5
2.2 SugarJ 7

3 Language Plugin **11**
3.1 Design goals 11
3.2 Components of the Language implementation 12

4 Technical realization **20**
4.1 Changes to the SugarJ compiler to support arbitrary programming languages 20
4.2 Overview of the project structure 21
4.3 Development process 22

5 Case studies **24**
5.1 Java 24
5.2 Prolog 24
5.3 Haskell 29

6 Discussion and Future work **31**
6.1 A look at the design goals 31
6.2 Future work 33

7 Related Work **36**
7.1 Racket 36
7.2 SugarJ 37
7.3 Pure embedding 37

8 Conclusion **38**

1. Introduction

Today's software is written in an increasingly large number of programming languages. Many of these programming languages are general-purpose languages, such as C, Haskell and Java. General-purpose languages allow programmers to solve a wide array of problems. But there are also special-purpose programming languages tailored to a specific problem domain. These include languages for describing structured data, like XML, and languages for querying databases, such as SQL. These special-purpose languages are much better suited for their domain than any general-purpose language, but they lack their generality, so they are not used for writing arbitrary programs.

New approaches to programming are constantly invented. Problems can often be solved in a better, more structured way by thinking of suitable abstractions. For example, related data can be organized as objects. Programmers can then define methods operating on these objects and manipulating them. Making this relation explicit by using objects enables programmers to write better software. Without this, the relation would of course still be there, only implicitly; programmers are then forced to keep this relation in mind when writing methods to manipulate data. However, many languages offer no built-in provisions for using these concepts. Since general-purpose languages allow writing arbitrary programs, these concepts can of course be expressed somehow. However, this is often inelegant and leads to confusing code. This problem could be overcome if languages allowed defining new syntax, so programmers can replace complicated representations of concepts by easy-to-understand syntax. The mapping from the convenient syntax to the complicated expression would have to be defined by the programmer. However, many languages do not allow defining new syntax. We want programmers to be able to define their own syntax so they can write code tailored to their problems and abstractions. In Java, for example, a pair of values can not be expressed conveniently. If programmers want to write methods returning a boolean expressing whether the operation was successful and a message with details, they can not simply write **public (boolean, `String`) `method()`**, but have to define their own custom data type. If Java offered extensible syntax, we could implement `(type1, type2)` as a return type to mean "return a custom data type with a variable of type1 and a variable of type2."

SugarJ [2] allows programmers to define their own syntax extensions for

Java. Syntax extensions are used like Java libraries, so programmers can import a library providing syntax for pair types just like they would import a library providing functions for trigonometry.

Our goal is to allow programmers to use any programming language they want to develop their software. To this end, we want to enable every programming language to be extended, but we also want to keep compatibility with existing code, libraries and tools. We base our work on SugarJ, rewriting it to be used for programs written in any programming language. We achieve this by identifying and abstracting Java-dependent code in SugarJ and providing an abstract language library. New languages can then be created as implementations of this library and added as language plugins. SugarJ is then able to provide support for the programming language.

SugarJ, as a syntactic preprocessor, operates on abstract syntax trees, which can be built from source code in any language given a suitable parser. Thus, much of the existing SugarJ code can be reused to provide its features for programmers using other programming languages. As SugarJ is a preprocessor, it also generates source code. This generated source code does not contain any features foreign to the programming language, so existing tools operating on this source code will still work. The most important of these tools is the language's compiler, which does not need to be modified.

In the scope of this thesis, we have abstracted over SugarJ's Java-dependent features and developed a language plugin architecture. Programmers can implement their own language plugins with little effort. We developed a small language API, which allows rapid implementation of new language support. Using Eclipse plugins to support different programming languages allows seamless integration into SugarJ's new plugin architecture. We implemented language plugins for Java and Prolog and enabled the implementation of a Haskell plugin.

This thesis is organized as follows: In Section 2, we first provide some background on domain-specific languages, syntactic sugar, and the purpose, features and program flow of SugarJ. This is followed by a detailed look into our extensible language plugin infrastructure, its design goals and implementation in Section 3. We dedicate Section 4 to the development process and technical realization. The case studies we carried out by implementing plugins for different programming languages to demonstrate the usability of our language plugin architecture are described in Section 5. In Section 6, we reexamine our design goals and point out potential improvements and future work. We briefly survey related work in the field of language extensibility in Section 7 and conclude in Section 8.

2. Background

2.1 Domain-specific languages and syntactic sugar

General-purpose programming languages, such as Java, are designed to be used for writing arbitrary programs. However, their rich feature set is often quite detached from the actual problem to be solved in a certain application. Although the problems can be solved, the solutions often lack clarity and elegance, leading to convoluted code. Special-purpose programming languages, in contrast, lack the feature set that makes general-purpose languages suitable for writing any program. However, they are optimized for their specific problem domain, allowing the programmer to solve problems easily, producing easy-to-understand code. Examples of special-purpose languages include XML, which is used to provide structured data representations, and SQL, which is used to query databases. These are also called *domain-specific* languages, as they are used for specific domains.

Because a lot of different problems can be solved with general-purpose programming languages, programmers want to write code that is clear, easy to understand and concise. One way to achieve this is to provide a level of abstraction that hides the inner workings of the programming language. Introducing shorthand notations in the programming language to provide these abstractions will not change the expressivity. Programmers will just be able to understand the code better if sensible abstractions are provided. These abstractions are called *syntactic sugar*. To provide an example, in C, accessing elements in an array is syntactic sugar:

```
int a[11];
// ...
int third_element = a[2];
int third_element_ = *(a + 2);
```

Although programmers could write `*(a + i)` to access the $(i + 1)$th element of an array, writing `a[i]` provides a level of abstraction: The programmer need not concern herself with intricacies of C (in this case, that `a` is just a pointer to the first array element), but can make code clearer by leveraging the concept of an array as a data structure with a number of cells, each of which contains a value.

Syntactic sugar is usually provided by the programming language and can not be extended by the programmer. Often, new ways of providing better

abstractions, such as object orientation, are discovered. Some level of object orientation can be implemented by using structs in C. Listing 2.1 shows a very basic example: We define a struct to encapsulate two values and define a custom type. Replacing `struct` by a better keyword and automatically generating the `typedef` would provide a more convenient way to express this. Syntactic sugar could be used to implement it; however, since C does not allow for user-defined syntactic sugar, this is not possible. Therefore, to support object orientation in a convenient way, the language itself needs to be changed, which happened to C with the introduction of Objective-C and C++. As an example, compare Listing 2.1, which uses a struct to implement the equivalent of a class in C, and Listing 2.2, which implements a class in Objective-C.

```
struct Paper
{
	float height;
	float width;
};
typedef struct Paper Paper;
```

Listing 2.1: Using a struct to define an equivalent of a class in C

```
#import <Foundation/Foundation.h>
@interface Paper : NSObject
{
	@public
	float height;
	float width;
}
@property(nonatomic, assign) float height;
@property(nonatomic, assign) float width;
-(id) init;
@end
```

Listing 2.2: Defining a class in Objective-C

Changing programming languages leads to a number of problems: Most importantly, a lack of tool support and the need to learn the new programming language. Often, this might be too high a price to pay for just being able to use some new syntactic sugar.

Syntactic sugar and domain-specific languages share the idea of making programming easier by providing easier-to-understand (and thus, better) ways to solve problems. However, not all problems can be solved by a domain-specific language; programmers want to use their general-purpose programming language of choice to write programs, using DSLs only for specific domains. To use a domain-specific language in a program, it needs to be embedded somehow. The usual approaches are to embed it as a String or to rebuild the DSL's structure in terms of the general-purpose programming

language, which is called pure embedding [5]. While string embedding is easy to use, the program treats it just as a string, so no checking of the actual string content is available. This can lead to hard-to-diagnose runtime errors. Pure embedding will allow static checking. However, the abstractions introduced are not helpful to the programmer. We would like to abstract over the general-purpose programming language by using the DSL's syntax. Pure embedding does not provide a high enough level of abstraction by requiring the DSL to be encoded in terms of the general-purpose language. This makes it hard to understand for people without knowledge of both the domain-specific and the general-purpose programming languages used.

As we have seen, general-purpose programming languages can benefit from domain-specific languages for solving special problems. However, actually embedding domain-specific languages is hard because they need to be encoded in terms of the general-purpose language (we deem string embedding to be mostly inacceptable), which reduces code clarity. If we can define our own syntactic extensions, programmers can define embeddings for domain-specific languages as syntactic sugar. This syntactic sugar will enable the DSL to be embedded verbatim and desugar it to the general-purpose language. This way, we can have both the benefits of pure embedding and the DSL's syntax. We can use the static checks provided by the general-purpose programming language with the convenient syntax of the DSL.

2.2 SugarJ

2.2.1 Purpose, features

SugarJ allows programmers to define their own syntactic sugar for Java programs [2]. This also allows domain-specific languages to be embedded easily. If we allow user-defined syntactic sugar and embedded DSLs, programmers should be able to use them easily. To this end, SugarJ ties into Java's module system. Programmers can import modules containing syntactic sugar definitions just like they would import other modules (i.e. packages) in Java. Syntactic sugars can also be composed, allowing sugar definitions themselves to use syntactic sugar defined in other modules.

```
import Xml;
public void genDocs(ContentHandler ch) {
        ch.<book title="Sweetness and Power">
        <author name="Sidney W. Mintz" />
        </book>;
}
```

Listing 2.3: Using XML syntax in Java with SugarJ. From [2]

To make this possible, SugarJ builds upon the Syntax Definition Formalism (SDF) [4] and the Stratego [8] term-rewriting system. SDF provides a generalized LR parser, which we use to parse programs to abstract syntax

trees. SDF uses grammars for parsing and allows these grammars to be extended. This is important because it allows SugarJ to add syntactic sugar definitions to the parser on-the-fly, enabling it to recognize syntactic sugar in programs. Stratego allows us to define rules which are then pattern-matched on the abstract syntax tree and perform tree transformations. Stratego rules can also be composed. This is also important: While the extensibility of SDF allows us to recognize syntactic sugar in the program, the extensibility of Stratego allows us to modify the desugaring transformation to enable desugaring on the fly.

2.2.2 Program flow

SugarJ is a preprocessor for Java files. However, it will not work on the program source code directly: SugarJ's approach is to parse the program first, turning it into an abstract representation: The abstract syntax tree (AST). In this representation, statements in the source code are nodes of the tree, so SugarJ will only have to operate on trees, not on tokens.

A short overview

When a file is to be processed, SugarJ will first initialize the parser with the Java grammar. This is needed to be able to generate the abstract syntax tree from the source code. It will then enter a loop which processes the file (see Figure 2.1). This loop consists of parsing the source file to obtain the next top-level entity, applying transformations (i.e. desugarings) to the top-level entity, and then processing the top-level entity. Processing will first analyze the current entity in order to find out what kind of entity it is. Depending on the result, further processing steps will be carried out, such as processing a namespace declaration, processing module imports, processing sugar definitions or language-definable language-specific processing.

After the whole module has been processed, checks will be carried out to ensure the correctness of the grammar and transformation which have been built and amended by the processing. Then, the abstract syntax tree is converted to a source file, which is then handed to the language's compiler to be compiled into a program (cf. Figure 2.1).

Syntactic sugar processing: A closer look

The overview of SugarJ's program flow might have left some questions unanswered, so we take a closer look on the most interesting processing step: How SugarJ actually deals with a program using user-defined syntactic sugar.

When a declaration of syntactic sugar is recognized in the AST analysis step, SugarJ retrieves the body of this declaration. This body contains a piece of SDF grammar which describes what this specific syntactic sugar looks like in the source code. This is needed to generate an AST node (which will be a

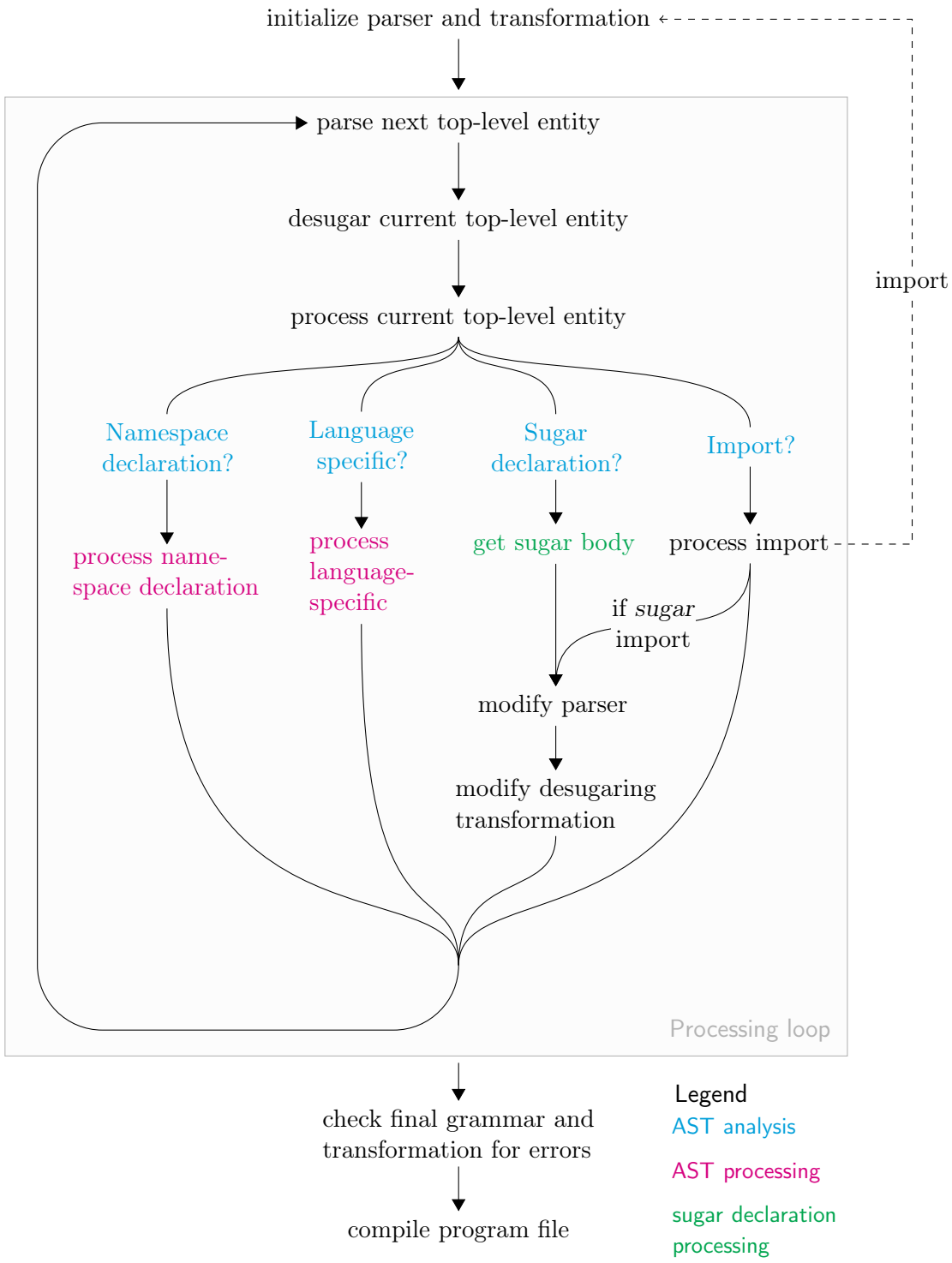

Figure 2.1: Steps to process a module

subtree for all but the most basic syntactic sugars) for the sugared expression; without having an AST node for everything in the source code, SugarJ will not be able to do anything. The sugar body will also contain Stratego rules defining how to transform the subtree expressed by the grammar into a valid subtree of the programming language.

Following that, SugarJ modifies the parser used to parse the program source code into an AST: Using the piece of SDF grammar defined in the sugar body, the parser can now produce a valid AST from code which was defined as syntactic sugar. This applies especially to code that is not valid code in the programming language. If the parser was not modified with this grammar, it would fail to recognize the user-defined code which can not be described by the programming language grammar, causing it to stop parsing[1].

However, merely being able to parse source code into a correct AST will not be of much use. The AST only describes the program's structure, not its meaning. Using the Stratego rules from the sugar body, the initial program transformation will be modified. It will then be able to transform the newly introduced AST nodes described by the SDF grammar into AST nodes defined by the programming language's grammar—also known as desugaring.

SugarJ's basis for extensibility

Operating on ASTs is the key to enabling us to adapt SugarJ to work with languages other than Java. In Section 4.1, we describe how we implemented this. SugarJ decides which processing steps to execute based on the content of the current AST node. Of course, the content of an AST node is different for each programming language. There are four meanings of the content that need to be recognized:

1. *Namespace declarations* provide SugarJ with information about the name of the current module and its place in the module hierarchy.
2. *Module imports* tell SugarJ to load the imported module. If it defines syntactic sugar, the parser and desugaring transformation will be modified.
3. *Sugar declarations* will be analysed to retrieve their content. They will then be used to modify the parser and desugaring transformation.
4. *Language-specific entities* are 'normal' nodes. They are also used for nodes that change the behaviour of the compiler or the module system in ways different from the other three categories. For example, when programmers define multiple classes in a single `.java` file, the Java compiler will generate a `.class` file for each class, which SugarJ needs to know.

All AST nodes need to fall in one of these four categories. This way, if SugarJ sees an unknown AST node after desugaring, we know that desugaring failed.

[1] Actually, error recovery mechanisms are used to still parse as much as possible

3. Language Plugin

In order to allow software developers to easily add support for their programming language of choice to SugarJ, we implemented a generic language plugin. This section describes the design considerations and its components.

3.1 Design goals

We reengineered the SugarJ system to achieve a number of design goals.

Generic Programmers should be able to implement support for any language. There is a large number of programming languages in use in industry and research. Many of these languages follow different philosophies and paradigms. We don't know what programming languages will be in use in the future, so concentrating on a small set of programming paradigms or even programming languages is not sustainable. Therefore, to enable programmers to use SugarJ for any programming language, past, present, or future, we want to keep the language plugin *generic*. By keeping the language library small, we also have the added benefit of reducing the interface to the absolutely essential. This, in turn, makes it possible to enable support for other programming paradigms, including non-imperative programming languages.

Small Implementing a language plugin should not require much effort, which is achieved by keeping the interface to SugarJ small. The utility of a system like SugarJ is largely dependent on its usability. We want to make sure that support for new programming languages can be implemented easily. This will allow rapid implementation of new programming language plugins, enabling more programmers to use SugarJ for their programming language of choice. We achieve this by focusing on *keeping the library small*. As a consequence, we consolidated SugarJ's code to allow reusing as much as possible.

Pluggable SugarJ is an Eclipse plugin and is intended to work in an Eclipse-based infrastructure. To allow extensibility, it makes sense to leverage the eclipse plugin infrastructure, allowing new languages to be implemented as plugins. Creating *pluggable programming language support* allows us

to distribute a base SugarJ system, allowing programmers to just write their own language plugins without touching SugarJ's code at all. This is very useful to ensure interoperability between language plugins implemented by different programmers, allowing them to be exchanged easily.

3.2 Components of the Language implementation

A language plugin consists of several components. They are shown in Figure 3.1.

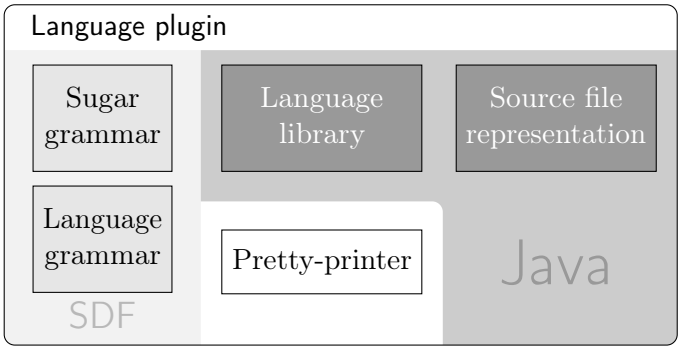

Figure 3.1: Components and implementation languages of the language plugin

3.2.1 SDF grammars describing the language and a way to define syntactic sugar

To implement a new language, SugarJ needs to know how to parse the program source code, how syntactic sugar is defined and how to pretty-print ASTs to an output which is accepted by the language's compiler.

To enable parsing the language to an AST, an SDF grammar describing the language is needed. We make no requirements concerning the structure and content of this grammar. However, as the implementation of the language API needs to discern namespace declarations, module declarations and module imports, we recommend to include distinct nonterminals for these. SugarJ currently does not have a mechanism to avoid the effects associated with name clashes; therefore, we recommend prefixing nonterminals with the language name to avoid future problems. To allow the desugaring transformation to use nonterminals defined in the grammar, a Stratego file needs to be created: This defines constructors for Stratego and can be generated from the SDF grammar by using Stratego's sdf2rtg and rtg2sig.

For defining syntactic sugar, the programmer also needs to implement the sugar grammar. This describes how sugar and its desugarings are defined. SugarJ expects syntactic sugar to be defined as an SDF syntax and Stratego

rules, which should be reflected in the grammar. The scope of this thesis was not to come up with a more intuitive or better integrated syntax for defining syntactic sugar and its desugarings, so for our case study, we only modified the grammar slightly to implement support for defining sugar for a new programming language. However, two things are important in this grammar: First, a nonterminal describing a sugar definition so that the implementation of the language API is able to recognize sugar definitions. Second, productions to define all the language's top-level entities and sugar declarations as *ToplevelDeclaration*. SugarJ works by incrementally processing the AST's top-level entities, loading desugarings and immediately applying them to the remaining AST. Without knowing the top-level entities, SugarJ will not work. As with the language grammar, a corresponding Stratego file defining the constructors is also needed for the sugar grammar.

3.2.2 A pretty-printer to generate source code from the AST

Compilers of programming languages do not work on the ASTs generated by SugarJ, so a means of transforming an AST back into source code is needed. This transformation is called pretty-printing and SugarJ requires a pretty-printer to be available. The implementation of this pretty-printer is left to the programmer. For the Java language library, we chose to use the Java pretty-printer included in Spoofax [6]; for the Prolog language library, we generated a pretty-printer from the Prolog grammar with Stratego.

3.2.3 Source file representation

SugarJ processes the program's AST. After the whole AST is processed, the AST needs to be converted back to code again, as the language's compiler will not understand SugarJ's ASTs. Whereas the language library is concerned with ASTs, the source file representation is concerned with code. The source file representation consists of two classes, `ISourceFileContent` and `SourceImport`. Each AST processing step will have to put a suitable code representation in the source file content. After processing is done, SugarJ will retrieve the source code from the source file content.

`ISourceFileContent` is an interface providing two methods:
isEmpty returns true if the source file is empty and false otherwise. A source file is empty if it does not contain any program code. In this sense, a source file containing only sugar definitions is empty. Empty source files are not written to disk and will not clutter the project. SugarJ will also not generate code to import empty source files, but will still import the sugar definitions. Therefore, source files which only define sugar should be empty.

getCode is called to obtain the source code of the processed file. It is up to the programmer how this is implemented. In our Java and Prolog implementations, each top-level entity is pretty-printed and then saved as

a String in the source file content. Some top-level entities, such as module declarations and imports, can not be pretty-printed directly and may require additional computations. To allow better handling of imports, we introduce SourceImport.

`SourceImport` is a class designed to help the programmer manage source file imports. It provides a way to store the actual path to an imported source file and its pretty-printed representation.

3.2.4 Language library

The language library is a collection of abstract methods that need to be implemented for each language. SugarJ uses these methods to initialize the parser, desugaring transformations and editor services, to recognize AST entities and to react on them, to process definitions of syntactic sugar, and to invoke the compiler. Listing 3.1 shows the abstract methods of the language library.

Initialization

SugarJ was redesigned as a modular, flexible and extensible system. As such, it operates on abstract data representations. As we focus on extensibility, we only make very limited assumptions about the structure of programs and no assumptions about their syntax. This allows developers to use SugarJ for any programming language. Because all programming languages are different, the first step is to initialize SugarJ with language specifications. This allows SugarJ to parse program source code, recognize elements in the thusly constructed abstract syntax tree (AST), process desugarings and provide editor services.

SugarJ needs to know the syntax of the language and the syntax of the sugar definitions. It also needs to know how to parse SDF and Stratego in order to extract these from sugar definitions. The language library features a method **getGrammars**, which returns grammars needed for parsing SDF, Stratego and editor services. The implementation of the language library should overwrite this method, getting the list and extending it by the language and sugar grammars. **getInitGrammar** provides a path to the SDF grammar containing the initial grammar used for parsing the programming language. **getInitTrans** returns the location of the initial desugaring transformation (which will be extended later by sugar definitions). **getInitEditor** returns the path to initial editor services.

File extensions

SugarJ will only operate on files having the correct file extension. All files that should be processed by SugarJ (including files neither using nor defining sugar, but being part of a program using sugar) are required to have the file extension returned by **getSugarFileExtension**. Currently, this needs to be `.sugj`
getGeneratedFileExtension defines the file extension of generated files, i.e.

```
public abstract class LanguageLib implements Serializable {
   // Initialization
   public List<File> getGrammars() { ... }
   public abstract File getInitGrammar();
   public abstract String getInitGrammarModule();
   public abstract File getInitTrans();
   public abstract String getInitTransModule();
   public abstract File getInitEditor();
   public abstract String getInitEditorModule();

   // File extensions
   public abstract String getGeneratedFileExtension();
   public abstract String getSugarFileExtension();

   // Source files and generated files
   public abstract void setupSourceFile(RelativePath sourceFile, Environment environment);
   public abstract ISourceFileContent getSource();
   public abstract Path getOutFile();
   public abstract Set<RelativePath> getGeneratedFiles();

   // AST analysis
   public abstract boolean isNamespaceDec(IStrategoTerm decl);
   public abstract boolean isLanguageSpecificDec(IStrategoTerm decl);
   public abstract boolean isSugarDec(IStrategoTerm decl);
   public abstract boolean isEditorServiceDec(IStrategoTerm decl);
   public abstract boolean isImportDec(IStrategoTerm decl);
   public abstract boolean isPlainDec(IStrategoTerm decl);

   // AST processing
   public abstract void processLanguageSpecific(IStrategoTerm toplevelDecl,
                                Environment environment) throws IOException;
   public abstract void processNamespaceDec(IStrategoTerm toplevelDecl,
                     Environment environment, IErrorLogger errorLog,
                     RelativeSourceLocationPath sourceFile,
                     RelativeSourceLocationPath sourceFileFromResult) throws IOException;
   // Namespace
   public abstract String getRelativeNamespace();

   // Module import handling
   public abstract boolean isModuleResolvable(String relModulePath);
   public abstract String getImportedModulePath(IStrategoTerm toplevelDecl) throws IOException;
   public abstract void addImportModule(IStrategoTerm toplevelDecl, boolean isCyclic)
                                                    throws IOException;
   // Sugar declaration processing
   public abstract String getSugarName(IStrategoTerm decl) throws IOException;
   public abstract IStrategoTerm getSugarBody(IStrategoTerm decl);

   // Editor services
   public abstract String getEditorName(IStrategoTerm decl) throws IOException;
   public abstract IStrategoTerm getEditorServices(IStrategoTerm decl);

   // Pretty-printing
   public abstract String prettyPrint(IStrategoTerm term) throws IOException;

   // Interface to the language compiler
   protected abstract void compile(List<Path> outFiles, Path bin, List<Path> path,
                                   boolean generateFiles) throws IOException;
   // Miscellaneous
   public abstract File getLibraryDirectory();
   public abstract LanguageLibFactory getFactoryForLanguage();
   public abstract String getLanguageName();
}
```

Listing 3.1: Abstract methods of the Language library

files generated by the language's compiler (e.g. .class for Java). This needs to be different from the sugar file extension. This way, SugarJ can recognize if program files have already been compiled and not compile them again.

Source files and generated files

These methods concern creating source files and generated files.
setupSourceFile is used to initialize a new source file and output file, i.e. create a file name for the output file and create a new `ISourceFileContent`. **getOutFile** returns a path to the output file, **getSource** should return this `ISourceFileContent`.

getGeneratedFiles returns the set of files generated by the SugarJ driver. It is modified by the driver and does not need to be used in the implementation of the language API. This is provided for languages which allow several files to be generated from a single source file. If this is the case, the language library should add the names of the other files that will be generated by the compiler to the set. Java, for example, allows the definition of several classes in one `.java` file, leading to the generation of multiple `.class` files.

AST analysis

There are several kinds of relevant AST nodes. When encountering them in the AST, special processing steps need to be carried out. Developing a language library includes implementing methods to check whether the current top-level declaration falls into one of these classes.

Namespace declarations are recognized by **isNamespaceDec**. Namespaces are required to assign names to modules, enabling building module structures and import systems. In Java, package declarations will be recognized by this method; the same applies for module declarations in Prolog.

Recognizing imports is done by **isImportDec**. When an import is recognized, the SugarJ compiler will resolve the import statement to a module on disk. If the imported module is a module containing definitions of syntactic sugar, this module is processed immediately, extending the parser and the desugaring transformation. This allows transforming AST nodes to desugared AST nodes. Software developers implementing a language library need not care about this, since it is all done automatically by the SugarJ compiler. If the import is not a module containing syntactic sugar definitions, it will be processed and added to the source file as an import module (cf. addImportModule).

isSugarDec recognizes declarations of syntactic sugar. Syntactic sugar will be processed, extending the parser and adapting the current desugaring transformation. If a sugar declaration is encountered, SugarJ extracts the SDF and Stratego-related parts from this declaration and modifies the parser on the fly. This allows the syntactic sugar to be used from the next top-level entity on.

isLanguageSpecificDec recognizes language-specific declarations. This should include all valid top-level entities of the programming language. It is required for two reasons. First, we want to ensure that SugarJ will stop processing the AST if unknown entities are encountered. Therefore, all known entities need to be defined somewhere. Some of them, as can be seen in this and the following section, trigger further processing steps. Others, like the language's top-level entities, just need to be known. Second, programming languages can be very different. We want to be as flexible and language-parametric as possible, so we allow language-defined processing steps to be carried out when certain entities are encountered. One example where this is needed is in our Java implementation: Having multiple class declarations in a single `.java` file requires several `.class` files to be created; a language-specific processing step defined in the Java language library modifies the list of generated files accordingly.

Editor services can be recognized by isEditorServiceDec. Extensible editors were not the focus of this thesis, but are processed like syntactic extensions.

AST processing

As we have seen, different kinds of AST nodes will lead to different processing steps to be carried out. We managed to keep most processing steps language-independent, so the programmer will just need to implement methods to identify the kind of AST node the SugarJ compiler is currently operating on. Processing of some AST nodes, however, can not be abstracted over all languages, so this has to be implemented separately for each language.

In SugarJ, this applies to language-specific top-level entities which are processed by processLanguageSpecific, which can be used if the language requires some extra processing to be carried out.
One example where this is sensible can be found in the Java library implementation: In Java, several classes can be implemented in a single `.java` file; upon compilation, separate `.class` files for each of these classes will be created. In the Java library, this step identifies the names of the classes and adds the filenames of their respective generated files to the list of generated files.

The other processing step which needs to be implemented by the library designer is processing namespace declarations, processNamespaceDec. SugarJ bases handling of modules and imports on the notion that namespaces are defined in each module, they are organized hierarchically and reflect the project's structure on disk. A language library designer needs to implement this processing step because we can not expect that namespaces are always defined in the same manner in a single top-level entity by all programming languages.

These are the only AST processing steps which need to be implemented. There is no processSugarDec: SugarJ will handle the processing of syntactic sugar automatically. Processing sugar declarations and applying desugarings

to the AST are done by the SugarJ driver. These processing steps will work for all programming languages, so programmers do not need to care about the implementation details. This reduces the effort spent in implementing a language library. However, methods to retrieve the name and definition of the sugar need to be implemented.

Namespace

SugarJ expects modules to have namespaces. Namespaces are also expected to be organized hierarchically. The project structure on disk should reflect the namespace structure, which allows SugarJ to infer the location of module files from their namespace. **getRelativeNamespace** should return a relative path in this sense.

Module import handling

By using the module system already known to the programmer, SugarJ makes using syntactic sugar equivalent to importing a module. Thus, module imports are processed separately. SugarJ will process imported modules, which requires their location on disk to be known. Checking for module availability also allows issuing warnings in the editor if the user tries to import a nonexistent or unavailable module. Import processing is almost completely done in the driver, so implementing support for a new language only needs a few simple methods.

isModuleResolvable checks whether the module can be resolved. Many languages feature library management mechanisms which allow modules to reside in a common library directory and be retrieved automatically by the compiler. Some languages also feature mechanisms to load modules from libraries which are different from the normal module import mechanisms. Modules in libraries do not need to be processed by SugarJ, so they are just added as imports.

getImportedModulePath returns the relative path of the imported module. This is important to find the module on disk, but even more so to distinguish SugarJ's standard modules from normal modules. The relative path of SugarJ's standard library modules begins with `org/sugarj`. Imports of standard library modules are processed differently. The language definition has to be a standard library module (e.g. `org/sugarj/languages/Java`).

addImportModule Imported modules are added to the source file and compiled. Some programming languages allow cyclic imports, which can be handled by SugarJ. Modules included in a cycle will be assumed to exist before actually being compiled. This also precludes having Sugar definitions in a cycle.

Sugar declaration processing

The methods **getSugarName** and **getSugarBody** are required to enable the processing of syntactic sugar declarations.

SugarJ uses a module system to manage different definitions of syntactic sugar. This is intended to integrate seamlessly into the programming language's module system; if an imported module happens to be a Sugar module containing definitions for syntactic sugar, SugarJ should extend the parser and desugaring transformation. For this to work, every sugar module needs to have a name which can be recognized by the module system. The purpose of the method **getSugarName** is to provide this name.

The other part of a sugar definition is returned by **getSugarBody**. It returns the SDF grammar describing the sugar's syntax, the program transformations used for desugaring written in Stratego and a list of Stratego rules to apply as the desugaring.

This is everything required for getting user-definable syntactic sugar to work. The driver will extract SDF and Stratego from the sugar definition and modify the parser as needed when the sugar is loaded.

Editor services

getEditorName returns the name of an editor extension. **getEditorServices** returns the definition of an editor extension. This works similar to syntactic sugar.

Pretty-printing

SugarJ produces an AST, but this is not what the programming language's compiler needs in order to produce a compiled program. A pretty-printer is needed to transform elements of the AST into program source code accepted by the compiler. The method **prettyPrint** should return the pretty-printed representation of the input IStrategoTerm.

Interface to the compiler

SugarJ is intended to act as an immediate step before the compiler. The language library provides the method **compile** to invoke the compiler. It is called after the complete AST has been processed. For Java, this should call javac. For interpreted languages like our choice of Prolog, this does not need to do anything. This method is protected because it is not supposed to be called directly; the language library contains a method to start compilation, which will call this method.

Miscellaneous

The language API also includes miscellaneous methods. The name of the programming language name should be defined to enable better logging messages. There also needs to be a method returning a factory to create a new language library. The library also needs to include a way to return its location on disk so filenames of grammars and transformations can be resolved correctly.

4. Technical realization

4.1 Changes to the SugarJ compiler to support arbitrary programming languages

After we built the abstract language library, adding support for arbitrary programming languages that can be implemented as language plugins to the SugarJ compiler was straightforward. SugarJ's processing steps can be seen in Figure 2.1. We will refer to this figure to explain what we modified.

We added the language library to SugarJ's initialization code as a parameter. SugarJ now uses the grammars and transformations defined in the language library. We did not need to modify parsing and desugaring.

To adapt SugarJ to work with languages other than Java, we modified all processing steps that analyze or extract information from AST nodes (refer to Section 2.2.2). Figuring out the meaning of an AST element—whether it is a namespace declaration, a language-specific entity, a module import, or a sugar definition—is now handled by the language library. We chose this approach because it allows developers to define their own names for these nonterminals in their language grammars. This makes development easier, because nonterminals in grammars can be given sensible names.

Processing namespace declarations, which will return the module's place in the module hierarchy, must be done by the language library. Module systems are vastly different across programming languages. Processing language-specific entities is, as the name implies, also completely done by the language library. To process imports, SugarJ calls the language library to extract the imported module's location from the AST node of the import. All other import processing is done automatically by SugarJ. To process sugar definitions, SugarJ requests the definition's body from the language library. Modification of the parser and transformation are handled by SugarJ.

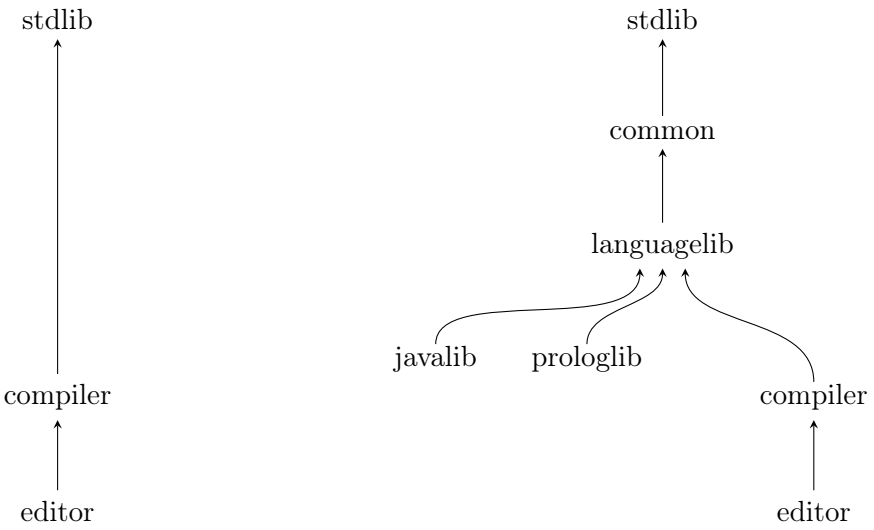

Figure 4.1: Project structure and dependencies before (left) and after (right) refactoring

4.2 Overview of the project structure

At the beginning of development, SugarJ consisted of three projects: compiler, editor and stdlib. The editor project is used for integrating SugarJ into eclipse, providing build services and editor support. The stdlib project contained the Java language and sugar grammars, initial transformations and editor services. The compiler project contained the whole program logic.

After reengineering SugarJ to provide an extensible plugin architecture, the SugarJ core consists of 5 projects: common, compiler, editor, languagelib, and stdlib. We also provide two language implementations, javalib and prologlib. The editor project was only modified slightly to support the language library. The common project contains commonly used classes and interfaces: Managing file paths, IO, logging, ATerm and java commands. The stdlib project contains grammars for SDF, Stratego, editor services, the common grammar defining files to consist of ToplevelDeclarations, as well as the standard transformation. The languagelib project defines SugarJ's interface for supporting new languages. Programmers can add support for a new language by creating a project which implements languagelib. This allows the SugarJ compiler to be used for providing SugarJ's features to this programming language without any further change. The languagelib defines the actual language API as well as an interface for representing source files. The compiler project contains the program logic of SugarJ which is responsible for transforming program source code to ASTs and processing desugarings. Care was taken to leave most classes in the compiler project and only extract the absolutely essential classes and methods to other projects.

4.3 Development process

SugarJ is implemented in Java as an Eclipse project. SugarJ was previously intended to provide a means to implement syntactic sugar for Java, whereas the goal of this thesis was to allow using SugarJ for arbitrary languages while keeping its main feature, library-based syntax extensibility. We chose an agile approach which was to first extract obviously Java-dependent code and then building a language API while reengineering SugarJ to turn it into an extensible plugin architecture. After this, the remaining Java-dependent aspects which might have been overlooked earlier were identified by implementing support for a different programming language in this architecture. This language implementation would also serve as a case study to demonstrate our new system.

The first step of development was to extract Java-dependent code. Some Java-dependent code was identified by stepping through a run of SugarJ with small Java source files and sugar files in the debugger and visualizing program flow. The identified code was then abstracted to create an initial version of the language library. We simultaneously built the Java library by moving the identified code there.

The original SugarJ implementation was characterized by tight coupling of language-specific and SugarJ-specific code as well as utility code which were organized as a single project (`compiler`). We extracted methods to the language API and its implementations, which were implemented as their own projects. This led to cyclic dependencies between projects, which can not be resolved by the Eclipse plugin system. For this reason, we spent some time on refactoring the program structure to resolve cyclic dependencies. There are three causes of cyclic dependencies, leading to three approaches for their resolution:
The first approach is to move whole classes to projects higher up in the dependency tree. This requires the class not to depend on other classes defined in the originating project. Some examples of this were utility classes for managing files and directories and ATerms. They can be found in the `common` project.

Moving whole classes as-is is not possible when classes depending on classes of their originating project are used in projects lower in the dependency tree. This problem is solved by the second approach: Identifying actual usage and extracting a suitable interface which is then implemented by the original class. One example of this can be seen in Result: This is a very compiler-specific class which stores the result of a compilation. In the original implementation, it also contained logging facilities required in the language library. Once these logging facilities were moved to a generic error-logger interface, Result could remain in the compiler, whereas the error logging is used in the language library.

Classes are used to group related methods. During refactoring, some

methods in classes became more related to methods in other classes, reflecting the change in program structure. Regrouping these methods helped in resolving cyclic dependencies. Increasing coherence inside classes also made the program more logical and easier to follow.

After refactoring into plugin projects was done, we added sugar support for another programming language. Adding support for a programming language consists of creating a language grammar and a sugar grammar, implementing a source file representation and the language API. The language we implemented was Prolog, which was chosen for several reasons. Prolog has a rather minimalistic syntax, making the grammar required for parsing small and easy to adapt to our needs. Prolog also features a module system, which is required for SugarJ. One additional point for choosing Prolog was the dissimilarity to Java, which would help to ensure that no Java-dependent features—or imperative-language-dependent features in general—would be left unidentified in the implementation.

The last step was then to refactor parts of the project, notably the language API: Consolidating similar methods, assigning meaningful names and cleaning up code.

5. Case studies

Using our language plugin architecture, we implemented support for several programming languages to show that the reengineered SugarJ system works. To give an overview about the effort it takes to implement a new language, we show lines of code of the language libraries.

5.1 Java

Support for the Java programming language was previously integrated into SugarJ. Obviously, making SugarJ language-parametric without providing a way to use legacy SugarJ code was not an option. Therefore, we built a Java language plugin while identifying and abstracting over Java-dependent code in the SugarJ implementation. Apart from minor changes in functionality due to API refinements, this plugin provides the same functionality as SugarJ originally did. Therefore, we expect all legacy SugarJ code to still work.

Language	**Lines of Code**
Java	398
SDF	1356
Stratego	446

Figure 5.1: Java language library

We want to focus on our Prolog language plugin, as the Java plugin contains mostly already existing code extracted from SugarJ. Therefore, no design choices had to be made.

5.2 Prolog

We implemented syntactic sugar support for Prolog. In the development process, this was useful to identify more Java-dependent aspects. However, as no support for Prolog existed previously, new code had to be implemented, thus enabling us to demonstrate developing a library for an arbitrary language.

Prolog is a declarative logic programming language. We chose SWI-Prolog[1] as our Prolog interpreter. Prolog was an attractive language to use in the development of SugarJ's language plugin system. It is attractive for several reasons: The language itself is quite small, offering only few top-level entities: Sentences (used for defining facts and rules) and Commands (used for querying). Therefore, the Prolog grammar is small, allowing us to concentrate on implementing and testing the language abstractions without spending too much time on writing grammars to support the parsing of programs. Another reason was the dissimilarity of Java and Prolog. This would help us to focus on keeping the language plugin small and generic, as well as open to declarative programming languages.

Apart from reasons rooted in the agile approach we chose to developing our language plugin architecture, Prolog is also an interesting language because it is actually used in research and industry. We wanted to avoid using toy languages to ensure that our abstractions can be used for programming languages in use.

5.2.1 Implementation

Figure 5.2 shows the amount of source code we required to implement the Prolog language plugin. Stratego code and the pretty-printer were generated automatically and fine-tuned by hand.

Language	Lines of Code
Java	407
SDF	321
Stratego	114
Pretty-printer	52

Figure 5.2: Prolog language library

Grammars

A large part of the effort of implementing support for a new language is implementing suitable grammars (see Figure 5.2). We used an existing SDF grammar for Prolog[2]. We added nonterminals for recognizing module declarations and module imports. We also require a much stricter Prolog program structure than specified in the grammar: Every Prolog file needs to be a module and start with a module declaration. Imports follow between the module declaration and the rest of the program. We want to enforce this so programmers will not be able to use predicates (and especially syntactic

[1] http://www.swi-prolog.org

[2] https://svn.strategoxt.org/repos/StrategoXT/prolog-tools/trunk/syn/Prolog.sdf

sugar) defined in a module before importing it. SugarJ only runs through the code once, so using an undefined syntactic sugar will lead to an error. We also modified the grammar slightly to forbid some reserved keywords.

Various modifications were carried out to allow parsing Prolog programs in a more suitable way for SugarJ's program flow. Overall, the modifications were minor. If an SDF grammar for the language of interest is available, this can therefore be modified to be usable for SugarJ with little effort. We mostly use the same syntax to define syntactic sugar as the original SugarJ implementation. We created the SugarProlog grammar from the SugarJava grammar with little modifications. If language library developers use a similar syntax to define syntactic sugar, Sugar grammars can be created easily.

AST analysis and processing

Prolog programs feature a simple structure with only few kinds of toplevel entities (see Listing 5.1). The AST analysis needs to be able to recognize every toplevel entity. In Prolog, recognizing module declarations `PrologModuleDec` is done by **isNamespaceDec**, as is recognizing sugar module declarations. `PrologModuleImport` is recognized by **isImportDec**; `SugarBody` is recognized by **isSugarDec**. All other toplevel entities—`PrologModuleReexport` and `PrologSentence`—are recognized by **isLanguageSpecific**.

If no special processing steps need to be carried out for language-specific entities, as is the case in our Prolog implementation, the AST processing is done by SugarJ. As could be seen in Section 3, only very little AST processing needs to be implemented by the programmer. To provide an example, Listing 5.2 shows the largest AST processing method which needed to be implemented in the Prolog library.

Implementing support for a module system

In Prolog, every module has a name and exports predicates. This is defined in the first line of a module with

```
:- module(modulename, [exported predicates]).
```

We require the module name to be equivalent to the file name.This is sensible because SWI-Prolog imports modules based on their file name, not on their module name. Other modules are imported by `:- use_module(modulename).` We require the module name in this command to be a path relative to the project's source directory. While this mimics Java's module system, we consider this essential for building programs structured into modules and especially for using sugar modules. Programmers want to use sugar modules by specifying their name without having to care about the location of this sugar module relative to the current source file.

```
%% Sugar module
SugarModuleDec PrologModuleImport* SugarBody
               -> SugarCompilationUnit {cons("CompilationUnit")}
%% module
PrologProgram -> SugarCompilationUnit {cons("CompilationUnit")}

%% toplevel entities
PrologModuleDec      -> ToplevelDeclaration
PrologModuleImport   -> ToplevelDeclaration
PrologModuleReexport -> ToplevelDeclaration
PrologSentence       -> ToplevelDeclaration
SugarModuleDec       -> ToplevelDeclaration
SugarBody            -> ToplevelDeclaration
```

Listing 5.1: Excerpt from the Prolog grammars showing toplevel entities

```
String moduleName = null;
if (isApplication(toplevelDecl, "ModuleDec")) { // prolog module
   moduleName = prettyPrint(getApplicationSubterm(toplevelDecl,
                                    "ModuleDec", 0));
   prologSource.setModuleDecl(prettyPrint(toplevelDecl));
} else if (isApplication(toplevelDecl, "SugarModuleDec")) { // sugar module
   moduleName = prettyPrint(getApplicationSubterm(toplevelDecl,
                                    "SugarModuleDec", 0));
   prologSource.setEmpty(true);
}
relNamespaceName = FileCommands.dropFilename(sourceFile.getRelativePath());
decName = getRelativeModulePath(moduleName);
log.log("The SDF / Stratego package name is '" + relNamespaceName + "'.");

if (prologOutFile == null) {
   prologOutFile = environment.createBinPath(getRelativeNamespaceSep() +
   FileCommands.fileName(sourceFileFromResult) + ".pro");
}
```

Listing 5.2: ProcessNamespaceDec from the Prolog library

Defining syntactic sugar

For defining sugar, our choice was to separate sugar definitions from normal Prolog code. Sugar is defined in Sugar Modules, for which we introduced a new command, `:- sugar_module(modulename).` The syntactic sugar declaration consisting of grammar, desugarings, and rules is then enclosed between `:-` and `.`, reminiscent of a Prolog command.

Source file representation

In the original SugarJ implementation, `.java` files were generated from source code only containing sugar declarations. These files were empty, but importing empty files is not a problem for the Java compiler, as they will be ignored. In our Prolog implementation, there are files which either contain no Sugar definitions or contain only Sugar definitions. An empty Prolog file is not a valid module. Thus, when empty files are imported as modules, the interpreter will not be able to consult the Prolog program and fail. SugarJ's source file representation allows source files to declare that they are empty. If they are, they will not be added to the list of imported modules. Our Prolog implementation makes use of this, elegantly avoiding the need to create essentially empty modules for each sugar definition. This is important for interpreted languages, where the intermediate code files—which would normally be passed to the compiler right away—are in fact the end result. Not creating unnecessary files helps to maintain a clean and uncluttered project structure.

5.2.2 Syntactic sugar in Prolog using SugarJ and the Prolog Library

We implemented a small example of syntactic Sugar in Prolog. Processing is entirely done by SugarJ with our Prolog language library, PrologLib.

Our implementation requires sugar to be defined in their own modules, which do not contain normal Prolog code. Listing 5.3 shows a sugar module containing a sugar definition. As an example, we implemented purely syntactic sugar:
Prolog clauses are written as `name(variables) :- body.` While this is a sensible notation, declarative programmers might also be interested in using a syntax inspired by S-expressions. We implemented syntactic sugar to allow clauses to be written as `(define-rule name (variables) (body))`

This is not a very interesting example on its own, but is sufficient for demonstrating SugarJ. Since the extension of the parser and desugaring transformation is independent of the implemented syntactic sugar and works for all correct sugar definitions, we only need to show that SugarJ will correctly carry out the processing steps, viz. parse the Prolog source file, import the sugar module, recognize the sugar declaration, invoke the mechanisms to

extend the parser and transformation, and pretty-print the processed AST. Therefore, showing that any sugar works is sufficient, irrespective of the sugar's sophistication.

Using this syntactic sugar, programmers can write Prolog programs like the one seen in Listing 5.4. The output generated by SugarJ can be seen in Listing 5.5. Note that the module import that imported the sugar definition does not show up in the generated source code.

5.3 Haskell

Outside the scope of this thesis, using our language-parametric SugarJ implementation, we implemented a language plugin to support Haskell [3]. This can be seen as further evidence that our language library interface is both useable and useful for a wide range of programming languages. As with the Java and Prolog libraries, lines of code are given in Figure 5.3.

Language	**Lines of Code**
Java	360
SDF	974
Stratego	329
Pretty-printer	267

Figure 5.3: Haskell language library, preliminary version

```
:- sugar_module(ruledef).
:- use_module(org/sugarj/languages/Prolog).

:-
context-free syntax
   "(" "define-rule" PrologWord "(" PrologVariable+ ")" "(" PrologBody ")" ")"
      -> PrologNonUnitClause {cons("RuleDef")}

desugarings
   desugar-ruledef

rules
   desugar-ruledef:
   RuleDef(rulename, ruleargs, rulebody) -> NonUnitClause(
                                               Func(
                                                  Functor(rulename), ruleargs
                                               ), rulebody
                                            )
.
```

Listing 5.3: Defining syntactic sugar in Prolog

```
:- module(sugartest, [myFact/2]).
:- use_module(prolog/ruledef).

myFact(0, 1).

(define-rule myFact ( A B ) (
   Y is A-1,
   myFact(Y, Z),
   B is A*Z
))

:- myFact(5, X), !, print(X).
```

Listing 5.4: Prolog code using sugar

```
:-module(sugartest,[myFact/2]).
myFact(0,1) .
myFact(A,B) :- Y is A - 1 , myFact(Y,Z) , B is A * Z .
:- myFact(5,X) , ! , print(X) .
```

Listing 5.5: Generated output from Listing 5.4

6. Discussion and Future work

As has been shown in this thesis, we have reengineered SugarJ to allow software developers to create their own language plugins. However, we also identified areas for improvement during development. We also discuss the design compromises we had to make.

6.1 A look at the design goals

Our reengineering effort concentrated on achieving three major design goals: Keeping the interface generic, small and pluggable (cf. section 3.1). We want language libraries to be pluggable, so SugarJ itself does not need to be changed to use a new language. This enables us to use Eclipse's plugin system to manage and distribute language plugins. Another goal is to keep the language libraries small to reduce implementation effort. Keeping the implementation effort low is crucial: We want SugarJ to be used, so interested programmers should be able to implement support for their language of choice easily. Our third goal is to make the language plugin API as generic as possible. We do not know which languages can benefit from SugarJ's features; designing the language API around a certain programming paradigm seems unnecessarily limiting. Even being able to provide just user-definable syntactic sugar would benefit all languages we can think of. While excluding programming languages due to the design choices we made is probably unavoidable, we still want to support as many as possible.

6.1.1 Design goal: Genericness

The main point of this thesis was to enable SugarJ to be used with programming languages other than Java. To this end, we identified Java-dependent code, recognized the underlying concepts and built a suitable abstraction. These abstractions constitute the language library. Our case studies show that we are able to implement support for languages following different programming paradigms, most notably imperative (Java) and declarative (Prolog, Haskell) languages.

However, there is a tradeoff between providing a generic language interface and enabling programmers to take advantage of the existing features. We

require languages to have a module system with certain properties:

1. They should have a module system which can infer the location of a module's source file from the name by which the module is imported. Reason: SugarJ has to process the source file.
2. Module imports may not affect preceding code. Reason: Programs are processed in linearly in one run.
3. Every source file of a program needs to be a module. Reason: SugarJ operates on modules.
4. Module imports may not interfere with each other: Modules will behave the same regardless of the order in which they are imported. Reason: SugarJ compiles and caches modules.

6.1.2 Design goal: Small language plugin

Keeping language plugins small will encourage software developers to implement their own language plugins. In order to keep language plugins small, we wanted to keep as much functionality as possible in the SugarJ compiler. Leveraging the already-existing codebase to provide most of the functionality will also keep the methods a programmer needs to implement for her new language library simple. We also extracted commonly used functions to their own plugin, so they can be used in the language library without having to reimplement them anew each time. While this has little influence on the number of methods that need to be implemented to fulfill the interface specifications, implementation effort for the actual language library is still reduced if programmers only have to implement few helper methods.

We acknowledge that keeping functionality in the compiler instead of requiring each language library to provide it might conflict with providing a generic library to implement support for all programming languages, our first design goal. However, our case studies provide hints to suggest that our language API is small and leads to small language implementations. Having working plugins for languages following different programming paradigms also suggests that we found a reasonable tradeoff between having small language plugins and being generic.

6.1.3 Design goal: Pluggability

We refactored SugarJ into multiple hierarchically organized plugin projects (see Figure 4.1). The SugarJ compiler was modified to only have dependencies to the LanguageLib plugin, which provides an abstract language API, not to concrete language implementations. The standard library plugin takes care of SDF- and Stratego-related aspects. This is important for a working language plugin architecture for a multitude of reasons: We could, in the future, make improvements to the standard library which language plugins

could then profit from. It also makes the individual language plugins more robust by not forcing the programmers to implement their own functionality to manage language-unrelated aspects. Providing a library for commonly used functions—such as file handling—as a separate `common` plugin enables us to update this without affecting the rest of SugarJ, in particular the language libraries implemented by our users. By implementing projects as Eclipse plugins, a good underlying plugin infrastructure is already in place to support pluggable language plugins.

However, even though all the provisions for providing pluggability are in place, our current implementation does not yet support languages as exchangeable plugins. Although the language libraries can define their own file extension, the current implementation of SugarJ requires this to be `.sugj`. Implementing the Eclipse plugin integration is left as future work. We explain this in detail in section 6.2.2. Our current architecture is already modeled to allow us to integrate it into Eclipse without further engineering effort.

6.2 Future work

SugarJ is an ongoing research project. By creating the possibility to use SugarJ's features for language extensibility for many programming languages, programmers and researchers alike are now able to spend less time on implementation effort and to solve their problems more easily.

6.2.1 Case studies

Currently, we have working implementations of three language plugins: Java, Prolog and Haskell. We would like to carry out more case studies by implementing more programming languages. This will help us to confirm the validity of our approach and the suitability of our language API for a broad range of programming languages.

Creating the Prolog plugin already showed us areas in which SugarJ can be improved. Even though Prolog's module system does not conform to SugarJ's expected criteria, we were able to implement Prolog support with little modifications to the language (see section 5.2.1). We will implement more language plugins to make SugarJ's features available to more developers.

6.2.2 Use the Eclipse plugin system

We reengineered SugarJ to enable language support as plugins. Languages can be implemented as Eclipse plugin projects and SugarJ is able to use them. Currently, programmers have to specify the language plugin they want to use by changing code in the SugarJ compiler. However, this only needs to be changed at a single place in a class specifically created for this purpose. To provide editor support, all code files also have to have the file extension

`.sugj`. We want to use Eclipse's plugin architecture to provide a menu to select the language SugarJ should use. SugarJ could also automatically use the correct language plugin depending on file extension.

6.2.3 Error messages

Our editor already provides programmers with error messages. They warn the programmer about unresolvable module imports and errors in desugaring. These features are enabled automatically by just implementing the language library. We also provide support for user-definable editor support and errors [1].

Language libraries specify the compiler used to compile program files. Currently, SugarJ will not take advantage of any compiler output. Compile errors will not be recognized as compile errors; instead, SugarJ will not be able to find the expected generated file and provide the user with a corresponding error. Many IDEs provide support to show compile errors directly, which is a valuable feature. We plan to introduce this feature to SugarJ by analyzing the compiler output. However, SugarJ is a preprocessor and will generate code. This generated code is stripped of non-functional layout and comments, and it also contains desugared syntactic sugar. Therefore, if a compile error occurs, we need to think of a way to map the error's location in the generated code to the error's location in the code written by the programmer. This will likely differ from language to language, so providing a generic, yet easy-to-implement way is an interesting question which could be explored further.

6.2.4 Module system

SugarJ has some requirements pertaining to the module system implemented by the programming language (see section 6.1.1). We can support a broad range of different programming languages, as we show by implementing plugins for Java, Haskell and Prolog.

SugarJ works well for languages with hierarchical module systems where the name of a file can be inferred through the module name. We want to carry out further research to find out how SugarJ's module system can be improved to support other kinds of module systems.

In Fortran 95, for example, each module has a name, which is defined using **module** `modulename`. Fortran source files containing module definitions are compiled to .mod module files which generally contain binary data. A compiled module's file name does not correspond to the module's source file, but to the module name. When modules are imported by **use** `modulename`, the Fortran compiler looks for `modulename.mod`. It is not possible, in general, to infer source code location from the module name.

A way to overcome this problem, as far as SugarJ is concerned, is to

modify the programming language to use SugarJ's module system. We did this with our implementation of the Prolog language plugin (see section 5.2.1). Yet, this leads to problems when using already-existing code: Module name definitions and module imports will need to be rewritten to conform to SugarJ's hierarchical module system.

We want to keep the language API simple, so extracting the complete module system to the language API and letting the programmer implement it will not lead to satisfactory results. We want to survey different programming languages and their module systems to identify recurring concepts in their module systems. With these concepts in mind, we will then find suitable abstractions and integrate them into SugarJ.

6.2.5 Language composability

For certain kinds of problems, even a general-purpose programming language might not be suitable. Examples of this include accessing low-level system features, for which many programming languages offer little to no support in the language itself. Usually, a way to call or encode code written in other general-purpose languages is provided. SWI-Prolog, for example, contains JPL[1], a library to use Java code in Prolog programs. JNI[2] can be used to use native code (written in C, C++ or other programming languages) in Java programs and to use Java code in C, C++ or other programming languages respectively.

These facilities, however, are often cumbersome to use. Thus, they are a prime candidate for defining user-definable syntactic sugar. SugarJ already enables sugar libraries to be composed. We would like to extend this to be able to compose languages libraries. This way, emebdded languages with their own language plugin could automatically benefit from the functions provided in this plugin: Module system support, editor services, and even user-definable syntactic sugar.

[1] http://www.swi-prolog.org/packages/jpl/
[2] http://docs.oracle.com/javase/7/docs/technotes/guides/jni/

7. Related Work

We compare our reengineered SugarJ version to Racket, which also supports user-defined languages. We also explain the differences between the previous version of SugarJ and our implementation.

7.1 Racket

Racket, a dialect of Scheme, supports user-defined languages [7]. Languages are defined as libraries and used by the programmer by writing `#lang languagename` at the beginning of the source file. Racket can just be reused and does not need to be modified, allowing programmers to use languages just as they would use libraries. However, all aspects of Racket can be changed by the language library, viz. notation, semantics, optimizations and the module system. In Racket, each module can be implemented in a different language. Racket only allows using one language per module.

In SugarJ, languages are also implemented as libraries. However, our approach is different: Instead of using a single base language which can then be extended by language libraries, SugarJ uses sugar libraries to extend languages, while language libraries are employed to support extending arbitrary languages. We do not allow modules to be written in different languages; we do allow using multiple sugar libraries in a single module. Racket transforms all code implemented in languages defined by language libraries to Racket code. SugarJ transforms code using syntactic sugar defined in a sugar library to code of the programming language implemented by the current language library.

Our focus is not on changing programming languages completely or being able to implement our own programming languages. We want programmers to still use the language, libraries and code they are already using, but with added extensibility through user-defined sugar. We therefore strive to make implementing language libraries as easy and effortless as possible. To make this possible, we give up the ability to change many aspects of the programming language, as is possible with Racket.

7.2 SugarJ

SugarJ provides syntactic extensions for Java as libraries [2]. These libraries can be defined by programmers using SDF and Stratego. Libraries can be composed to use sugar inside sugar. For example, if a sugar library for having pair types in Java and a sugar library for directly writing XML is imported, programmers can use pairs in their XML elements. SugarJ also provides extensible editors through editor libraries [1]. Like sugar libraries, editor libraries are imported like Java packages. They will then extend the editor with services such as code folding, syntax highlighting and code auto-completion.

We build upon the features of SugarJ. We provide a Java language plugin for SugarJ, which is fully compatible with all legacy SugarJ code. Thus, programmers can transition seamlessly from SugarJ to our implementation.

Our contribution is an abstract interface for adding support for any programming language to SugarJ. Other systems to provide syntactic extensibility through libraries by operating on the AST will require a similar set of features. Our interface contains methods for AST element analysis to decide how AST elements should be processed. It also contains methods for module definition processing and module import handling, which are used to establish a module hierarchy, find modules on disk and compile them separately. The interface also contains methods for extracting definitions of syntactic sugar and handing it to SugarJ's compiler.

This interface describes high-level features common to many programming languages. Programmers can implement it as an Eclipse plugin to add support for their programming language without changing SugarJ itself. We verify this approach through case studies and implement support for a declarative language. By being able to implement support for declarative languages, such as Prolog and Haskell [3], we show that our abstractions are suitable for a wide array of programming languages and paradigms.

7.3 Pure embedding

Hudak describes pure embedding, an approach for embedding DSLs in general-purpose programming languages [5]. Pure embedding works by encoding the DSL in terms of the host language's syntax.

In our implementation of SugarJ, as well as the original SugarJ, we allow DSL embedding by providing syntactic extensibility. Programmers can write their own sugar library that defines desugarings for the DSL syntax to programming language syntax. Hudak's approach does not provide a convenient syntax, but works with all programming languages. In order for SugarJ to work with a programming language, we have to implement a language plugin for this language. Also, our syntactic sugar libraries are not generally portable across languages.

8. Conclusion

Programmers can benefit from being able to extend the syntax of the programming language they are using. SugarJ provides syntax extensions for Java as libraries. Our reengineered version provides user-definable syntax extensions for arbitrary programming languages.

To this end, we identified aspects of programming languages related to extensibility, mainly their module system. We also identified aspects related to AST processing. We found abstractions for these aspects and extracted them into a language library. To enable SugarJ to support a new programming language, programmers can implement this language library for their language. We focus on keeping the library generic, so we can support many programming languages. We also keep it small to enable programmers to implement support for new programming languages easily. To keep the SugarJ compiler independent of the supported programming languages, programming language libraries are implemented as Eclipse plugins.

We implemented language plugins for Haskell, Java, and Prolog. The Java language plugin supports all legacy SugarJ code. The Prolog and Haskell language plugins show that our abstractions are also valid for declarative programming languages and different module systems.

We also identified some areas for further research on the topic of enabling syntactic extensions for arbitrary programming languages. We will look into different module systems used by various programming languages to find better abstractions for their module systems.

Bibliography

[1] S. Erdweg, L. C. L. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. Growing a language environment with editor libraries. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 167–176. ACM, 2011.

[2] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: Library-based syntactic language extensibility. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 391–406. ACM, 2011.

[3] S. Erdweg, F. Rieger, T. Rendel, and K. Ostermann. Everybody needs a syntax extension sometimes. 2012. Submitted to *Haskell Symposium*.

[4] 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.

[5] P. Hudak. Modular domain specific languages and tools. In *Proceedings of International Conference on Software Reuse (ICSR)*, pages 134–142. IEEE, 1998.

[6] L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 444–463. ACM, 2010.

[7] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011.

[8] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In *Proceedings of Conference on Rewriting Techniques and Applications (RTA)*, volume 2051 of *LNCS*, pages 357–362. Springer, 2001.

Our implementation can be found online at https://github.com/frieger/sugarj.