Transformers: a C++ program transformation framework

Robert Anisko
Valentin David
Clément Vasseur

Technical Report n^o0310, revision 1, 6th May 2004

Many software engineering problems, such as refactoring or optimization, can be efficiently solved using source-to-source program transformation technologies.

Moreover, in the specific case of the C++ language, automatic program transformations can be used as an attempt to bridge the gap between a classic programming style, and the intensive meta-programming techniques involved in generative libraries.

In this report, we share our experience in the development of a C++ program transformation framework, ranging from our selection of meta-tools and the architecture of our system, to issues relevant to the bad syntactic and semantic properties of the language itself. Some new perspectives on active libraries are also discussed.

Keywords
C++ language, Parsing, Program transformation

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
lrde@epita.fr – http://www.lrde.epita.fr

Copying this document

Contents

List of Figures

List of Programs

List of Algorithms

Chapter 1

Introduction

This report is mainly a case study in program transformation. It discusses the development of a transformation system dedicated to the C++ language.

More specifically, we focus on the tools being used for the implementation of our framework, the global architecture of our system with respect to these implementation tools, and the issues encountered when dealing with C++.

1.1 Why program transformation?

Program transformation is indeed a very general term, which can cover both many different applications and many different technologies. However, it usually denotes transformations based on the structure of the programs being manipulated; traditionally performed with tree rewriting systems.

The very first consequence of using tree rewriting techniques is, obviously, the need for tools suitable to the implementation of the usual processing steps on trees:

- syntactic analysis,
- rewriting,
- unparsing or pretty-printing.

The second consequence of using tree rewriting systems instead of weaker techniques, such as mere text replacement, is simply the ability to cover a broad range of applications:

- program refactoring and renovation,
- program documentation and instrumentation,
- translation and compilation,
- high-level optimization, such as partial evaluation, for example.

This list is far from being complete. For a detailed taxonomy of program transformation, see Visser and Deursen (2000).

1.2 Applying program transformation to generative libraries

Above, we have seen an overview of the traditional motivations for using program transformation, but our interest in these techniques is more specific.

Actually, the applications of program transformation we are targeting at are closely related to the development of two generative libraries, *Olena* (dedicated to image processing) and *Vaucanson* (dedicated to finite state machines).

1.2.1 The libraries and the language

These libraries have very similar goals. They mainly aim at providing a high level of genericity (the ability to process inputs of very different kinds), while keeping at the same time a high level of performance (no abstraction penalty, no additional run-time cost).

In practice this can be achieved in C++ with the generic programming paradigm, using so-called meta-programming techniques, which rely on compile-time computations and the fact that C++ actually became a two-level language after the introduction of parametric polymorphism, using template constructions.

A discussion of C++ as a two-level language and the relationship with partial evaluation can be found in Veldhuizen (1999). The generic programming paradigm and its application in the Olena image processing library are discussed in Géraud (2002) and Darbon et al. (2002).

1.2.2 Drawbacks to generic programming

Unfortunately, although these programming techniques enable us to write both generic and efficient libraries, they suffer from some major issues:

- Since meta-programming involves compile-time computations and forbids separate compilation, the compilation process becomes extremely heavy.
- Worse, generic programming tends to make programs very complex and difficult to write, at least from the library implementor point of view. Of course, generative libraries are as well difficult to read, which makes the debugging and maintenance a tedious task.

1.2.3 A solution

Simplifying generic programming

Obviously, this is were program transformation techniques are needed.

In order to improve development in the generic programming paradigm, we intend to automate the process of deriving active libraries, that make use of intensive meta-programming techniques, from libraries written in a much simpler way, akin to the usual C++ programming style; in other words, classic C++, possibly equipped with some syntactic extensions designed to capture some concepts that exist only in generic programming.

The constraints

Of course, the need for applying any kind of structure-based transformation to C++ programs gave birth to a project of itself: the development of a transformation framework dedicated to the C++ language, whose first sketch is presented in this report.

Keep in mind, though, that this project was initiated with some strong constraints in mind, most noticeably concerning the grammar being used for the C++ language.

Our grammar was first extracted from the grammar given in the C++ language standard, and we tried to remain as close as possible to this original grammar, even though it is far from being perfect. There are several motivations for this guideline:

- At first, it makes our grammar very close to the reference grammar. This is important when considering our transformation system as a stand-alone project: it seems to be quite a reasonable claim for people working with C++ to be able to manipulate a grammar being nearly the standard grammar.
- While the standard grammar is not flawless[1], it is probably the simplest possible grammar for C++. This is a good property in the context of program transformation, for the grammar

[1] And cannot be. As it is explained later in this report, this grammar can only be ambiguous.

determines the shape of parse and abstract syntax trees; the more the grammar is complex, the more the trees are complex, and the transformations longer to specify.

1.3 The Transformers project

The work on the C++ transformation framework was initiated at LRDE[2] by Robert Anisko, in 1999. The work was supervised by Akim Demaille, languages and compilation teacher. In 2003, Robert worked at the University of Utrecht (the Netherlands), with Eelco Visser, one of the main authors of the tools we are using. Two new students took over the Transformers project: Valentin David and Clément Vasseur.

The first version of this document was written by Robert Anisko. When Valentin and Clément continued the project, the documentation was updated to explain the latest improvements.

[2] EPITA Research and Development Laboratory

Chapter 2

Tools and architecture of the framework

This chapter briefly describes the meta-tools that have been chosen for the implementation of our transformation system, and explains the motivations behind these choices. The global architecture of our system is then presented, and we show how it is integrated with this collection of meta-tools.

2.1 Meta-tools

Most of the tools and technologies we use are imported from the following projects or collections of tools:

- The *ASF+SDF Meta-Environment*, developed at the *Centrum voor Wiskunde en Informatica* under the *Generic Language Technology* project (Brand et al., 2001).
- The *Stratego* (Visser, 2001) language for specification of program transformations. The Stratego compiler is currently under development at the *Utrecht University*. The language and compiler were first prototyped in the *Pacific Software Research Center*, at the *Oregon Graduate Institute*.
- The generic pretty-printer *GPP* (Jonge, 2000).
- *StrategoXT* (Jonge et al., 2001; Jonge and Visser, 2001b), a collection of program transformation tools. This bundle includes among other things the *SGLR* parser, the Stratego compiler, and GPP, the pretty-printer.

2.1.1 Parsing with SGLR

One of the essential components used in our transformation system is the generic SGLR parser. This powerful tool implements scanner-less generalized LR parsing, and provides a large amount of advantages over traditional parsing techniques:

- There are no restrictions on the class of context-free grammars that can be handled. In practice, this implies that there is absolutely no need to massage and obfuscate a grammar before its use.
- Generalized parsing is not necessarily non-ambiguous. Thus, when dealing with an ambiguous grammar, the parser can build a parse forest rather than a single tree. This is especially useful when working with a language such as C++, that suffers from various ambiguities.

- Context-free grammars are closed under union, unlike subclasses such as *LALR*. This enables to define modular grammars when working with SGLR and the *SDF* grammar formalism. The readability and maintainability of the grammars being developed is significantly increased.
- This grammar modularity is also of interest in the specific case of our work, because we aim, among other things, at introducing various extensions to C++, possibly giving birth to different flavours of the language. In this context, we cannot afford the use of several one-chunk grammars that would make the maintenance of the shared core language unmanageable.

A more detailed description of SGLR can be found in Brand et al. (2002).

2.1.2 Rewriting with Stratego

Most software components of our system are written in Stratego, a language dedicated to program transformation, based on rewriting strategies. From these specifications, the Stratego compiler produces C code that is used to build the stand-alone programs that compose our processing chains.

In the specific case of our C++ transformation system, the following features are of interest in Stratego:

- Stratego primarily supports specifications of transformations on abstract syntax trees, unlike the ASF+SDF Meta-Environment, that focuses on concrete syntax. This is of prime importance when working with a language whose concrete syntax is ambiguous. Nevertheless, there exists a mecanism in Stratego to write transformations using the concrete syntax of the object language.
- Transformations can also be defined directly on parse trees in *AsFix* format. This is very helpful to address informations that cannot be represented in abstract syntax trees of the object language, such as parsing ambiguities.
- Rewriting strategies, used to define how and when rewrite rules should be applied, contribute to augment significantly the modularity and reusability of specifications. The strategic programming paradigm is discussed in Lämmel et al. (2002).
- Last, Stratego is provided with a very complete library of rules and strategies that implement generic traversals, standard data types, or various system interfaces (Visser, 2000).

2.2 Architecture

In many of its aspects, our transformation system is extremely classic, and similar to many other projects that use the meta-tools described in section 2.1.

2.2.1 Grammar and derived products

Due to the nature of the implementation tools, the grammar plays a central role in our architecture. Of course, it is used to generate the parse tables for the syntactic analysis stage, but also serves to the communication of parse and syntax trees between the various software components, by acting as a contract (Jonge and Visser, 2001a). Last, the grammar is a basis for the generation of a pretty-printing table.

Thus, several processing stages in our system are dedicated to transformations on the C++ grammar.

Grammar

The base grammar we use is based on the extended BNF specification of the C++ ISO/IEC international standard (iso, 1998). It is mostly a translation to SDF: except for some minor changes and corrections, the standard grammar required no massaging. Including both lexical and context-free syntax, our grammar is approximately 520 rules big.

Unfortunately, this grammar is far from being perfect:

- Since C++ is a context dependent language, the only possible way of making its grammar fit into a context-free specification is to produce an ambiguous grammar. This is the grammar we refer to as being our grammar for C++ language; actually, it does not define strictly the syntax of C++, but a super-set of the language.

  Of course, this is a major issue in our transformation system: a large effort must be devoted to a semantic analysis stage, whose purpose is the removal of parsing ambiguities.

- In this grammar, some rules do not constrain the input texts enough. This is the case in particular for definitions, which are all handled by a single rule, shown in figure 2.1.

  Some rules of this kind have been left unchanged in the grammar, to remain as close as possible to the reference grammar of the standard, but also because these very simple rules define each time a large amount of correct constructs; enumerating only all well-formed constructs would have certainly caused a blowup of the number of rules in the grammar.

Figure 2.1 Definitions in C++

$$\mathit{DeclSpecifierSeq}_? \ \mathit{InitDeclaratorList}_? \ \texttt{;} \quad \rightarrow \quad \mathit{SimpleDeclaration}$$

  Because of such rules, many ill-formed programs are accepted at parsing-time. As a consequence, there is a need in our framework to analyze the trees returned by the parser, not only to disambiguate them, but also to reject invalid inputs.

Grammar processing

As described above, this grammar is processed by several fully automated tools before it is actually used for program transformation (a summary of this processing chain is given in figure 2.2):

- Some minor corrections are applied to the original grammar. A tool named `sdf-option` introduces intermediate non-terminals to remove optional literals. Optional literals are a very convenient construct, as shown by figure 2.3, but they are troublesome in abstract syntax trees. The grammar excerpt from figure 2.3 is transformed in the rules shown in figure 2.4.

- The original grammar is annotated with constructors, using `sdf-cons`. This tool associates a constructor to each production rule; this constructor is a label used during implosions of parse trees into abstract syntax trees. The grammar chunk of figure 2.5 is transformed to the rules shown in figure 2.6.

- The annotated grammar is used to produce the parsing table, with the help of the generator

- The annotated grammar is used to produce a Stratego signature using `sdf2sig`. This signature is then used to define transformations on abstract syntax trees. Figures 2.7 and 2.8 give a small example of signature generation.

- A pretty-printing table is generated from the annotated grammar using `boxedsdf`.

Figure 2.2 C++ grammar processing

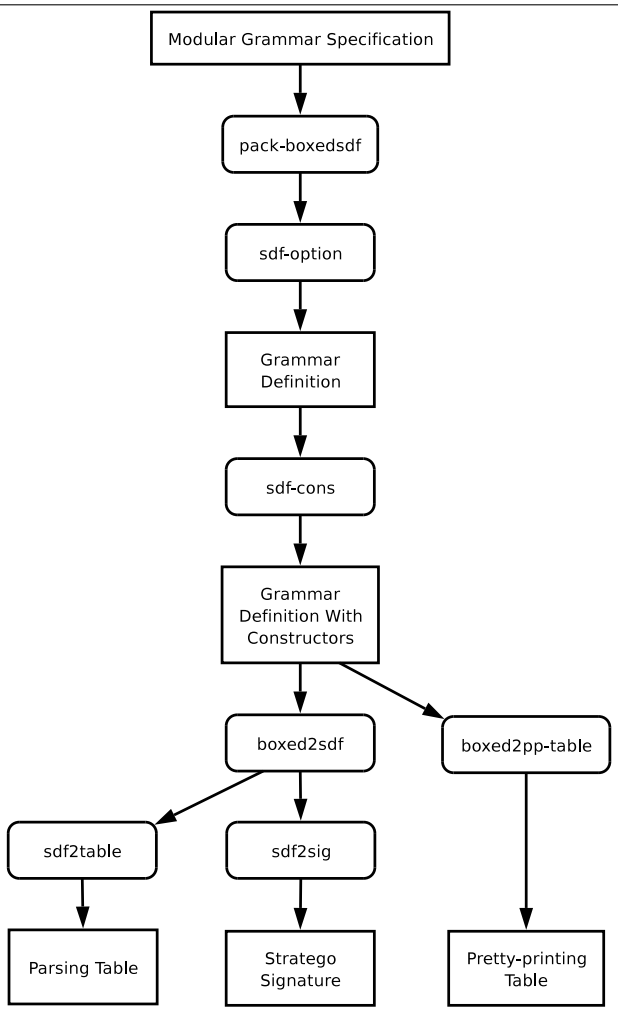


Figure 2.3 Removing optional literals - source

$$
\begin{array}{lcl}
\texttt{::}_? \; \textit{NestedNameSpec}_? \; \textit{ClassName} & \rightarrow & \textit{BaseSpec} \\
\texttt{virtual} \; \textit{AccessSpec}_? \; \texttt{::}_? \; \textit{NestedNameSpec}_? \; \textit{ClassName} & \rightarrow & \textit{BaseSpec} \\
\textit{AccessSpec} \; \texttt{virtual}_? \; \texttt{::}_? \; \textit{NestedNameSpec}_? \; \textit{ClassName} & \rightarrow & \textit{BaseSpec}
\end{array}
$$

Figure 2.4 Removing optional literals - result

$$
\begin{array}{lcl}
\texttt{virtual} & \rightarrow & \textit{Dummy7} \\
\texttt{::} & \rightarrow & \textit{Dummy0}
\end{array}
$$

$$
\begin{array}{lcl}
\textit{Dummy0}_? \; \textit{NestedNameSpec}_? \; \textit{ClassName} & \rightarrow & \textit{BaseSpec} \\
\texttt{virtual} \; \textit{AccessSpec}_? \; \textit{Dummy0}_? \; \textit{NestedNameSpec}_? \; \textit{ClassName} & \rightarrow & \textit{BaseSpec} \\
\textit{AccessSpec} \; \textit{Dummy7}_? \; \textit{Dummy0}_? \; \textit{NestedNameSpec}_? \; \textit{ClassName} & \rightarrow & \textit{BaseSpec}
\end{array}
$$

Figure 2.5 Annotating with constructors - source

```
while ( Condition ) Statement                                  →  IterationStatement
do Statement while ( Expression ) ;                            →  IterationStatement
for ( ForInitStatement Condition? ; Expression? ) Statement    →  IterationStatement
```

Figure 2.6 Annotating with constructors - result

```
while ( Cond ) Stm                          →  IterationStm  {cons ("while")}
do Stm while ( Expr ) ;                     →  IterationStm  {cons ("do-while")}
for ( ForInitStm Cond? ; Expr? ) Stm        →  IterationStm  {cons ("for")}
```

Figure 2.7 Signature generation - source

```
PostfixExpression ++                                  →  PostfixExpression
PostfixExpression --                                  →  PostfixExpression
dynamic_cast < TypeId > ( Expression )                →  PostfixExpression
static_cast < TypeId > ( Expression )                 →  PostfixExpression
reinterpret_cast < TypeId > ( Expression )            →  PostfixExpression
const_cast < TypeId > ( Expression )                  →  PostfixExpression
```

Figure 2.8 Signature generation - result

```
PostfixExpression  :  PostfixExpression          →  PostfixExpression
            decr1  :  PostfixExpression          →  PostfixExpression
 TypeId-Expression  :  TypeId * Expression       →  PostfixExpression
TypeId-Expression1  :  TypeId * Expression       →  PostfixExpression
TypeId-Expression2  :  TypeId * Expression       →  PostfixExpression
TypeId-Expression3  :  TypeId * Expression       →  PostfixExpression
```

2.2.2 Processing chain

When all the informations derived from the grammar have been successfully generated, the remaining software components of our transformation system are built from their Stratego specifications.

Unlike many transformation software based on SGLR and Stratego, the core of our system is not based solely on our grammar of the C++ language. Where a typical processing chain branches the various rewriting components after the syntactic analysis stage, we introduce a post-processing step to assist the parser.

The need for this post-processing stage is due to the nature of the grammar being used for the C++ language (section 2.2.1). We end up with a syntactic analyzer that is capable of reading an input text in several different fashions; upon success, it returns a parse forest rather than a single tree.

Of course, the purpose of these post-processing tools is to perform various analyses on the parse forest to filter it and reduce it to one correct tree that can be finally shipped to the transformation components.

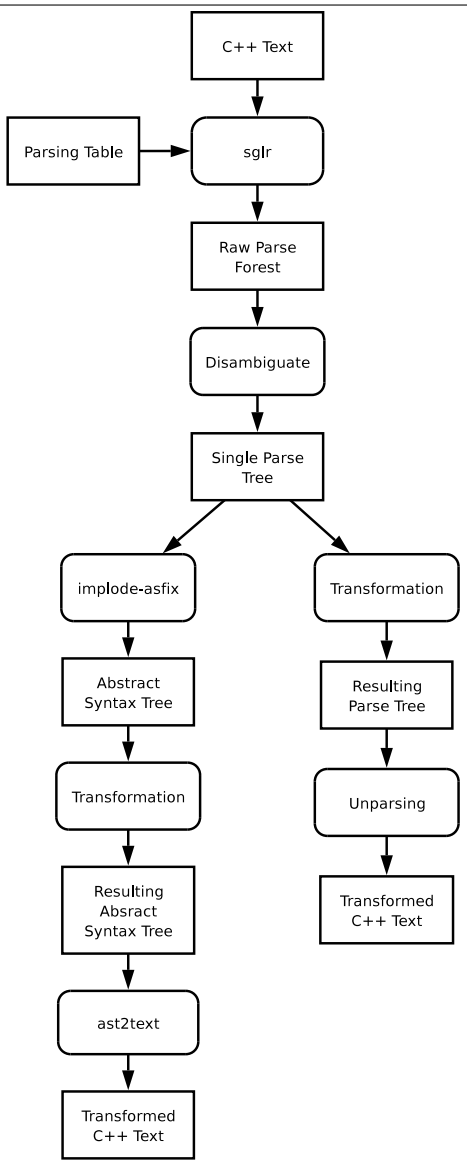
The processing of a C++ program includes the following steps:

- Parsing with `sglr` and the table generated from the grammar.
- Reduction of the parse forest by sequential application of several disambiguation tools. This step is discussed in depth in chapter 3.

- Implosion of the resulting parse tree to an abstract syntax tree. This is done by invoking `implode-asfix`.
- Transformation of the abstract syntax tree.
- Pretty-printing with `ast2abox` and a box back-end (such as `abox2text`, `abox2html`, or `abox2latex`).

For some applications, imploding the parse tree must be avoided[1]. In such cases, an alternative processing chain is used, where transformations work directly on parse trees, and the result is unparsed, rather than pretty-printed. The diagram shown in figure 2.9 is a summary of the possible processing chains.

Figure 2.9 C++ program processing



[1] Working on parse trees can be needed in some applications, for example to keep track of the layout and comments.

2.3 Non-ambiguous grammar

Some grammars, like the C++ one, have an ambiguous syntax. Tools for C++ disambiguation exist; they are described in the chapter 3. Sometimes, however, we cannot or do not want to use disambiguation tools. The developer of the C++ source has to disambiguate manually.

This manual disambiguation process uses markups that would be treated as comments in the classic grammar. From a grammar, the `det-gen` tool generates the extension to make the corresponding non-ambiguous grammar. In order to add the markups in the grammar, tags are added in the original grammar.

2.3.1 Grammar tags for non-ambiguous grammar generation

Since markups must be comments in the original grammar, the grammar definition must contain a proper comment production in the lexical syntax part of the grammar definition.

For each production that should accept markups, a "`dettag`" attribute has to be specified. The beginning markups will look like "*CommentBegin*`[`*DetTag*`]`*CommentEnd*" and the ending one "*CommentBegin*`[!`*DetTag*`]`*CommentEnd*".

To generate this new grammar, `det-gen` generates production rules to reject the markups as comments and new production rules to accept them as markups in the grammar. Figure 2.10 is an example of original grammar, the corresponding extended grammar is shown in the figure 2.11.

Figure 2.10 Grammar tagging for non-ambiguous grammar generation

```
lexical syntax
  "/*" (~ [\*]|ASTERISK)* "*/"   →   LAYOUT
context-free syntax
  Identifier                     →   UnqualifiedId   {dettag("uid"),
                                                      cons("Identifier3")}
```

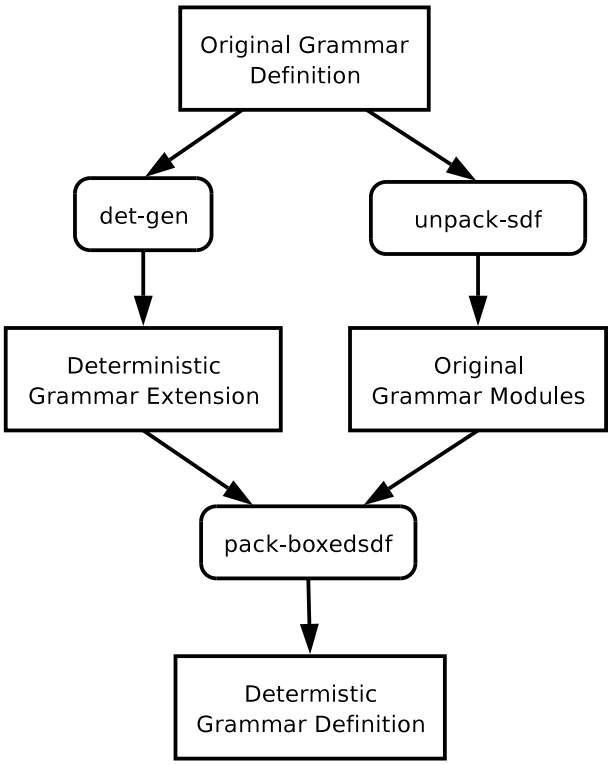
Figure 2.11 Generated non-ambiguous grammar

```
"/*[uid]*/" Identifier "/*[!uid]*/"   →   UnqualifiedId   {cons("Identifier3")}
"/*[uid]*/"                           →   LAYOUT         { reject }
"/*[!uid]*/"                          →   LAYOUT         { reject }
```

2.3.2 Generating non-ambiguous grammar

The generated production rules make a grammar extension. This grammar extension has to be added to the original grammar. This processing is shown in figure 2.12.

Figure 2.12 Non-ambiguous grammar generation



2.4 Transformations

2.4.1 Grammar extensions

Since the formalism used for the C++ syntax is modular, it would be nice to be able to extend the grammar so as to accept new operators, for example.

The SDF module in figure 2.13 shows an example of extension of the C++ grammar with a "<c" operator that tests inheritance between classes.

Figure 2.13 C++ extension

```
module SubclassCxx
imports Cxx
exports
 context-free syntax
  ClassName "<c" ClassName  →  ConditionalExpression  {cons("SubclassTest")}
```

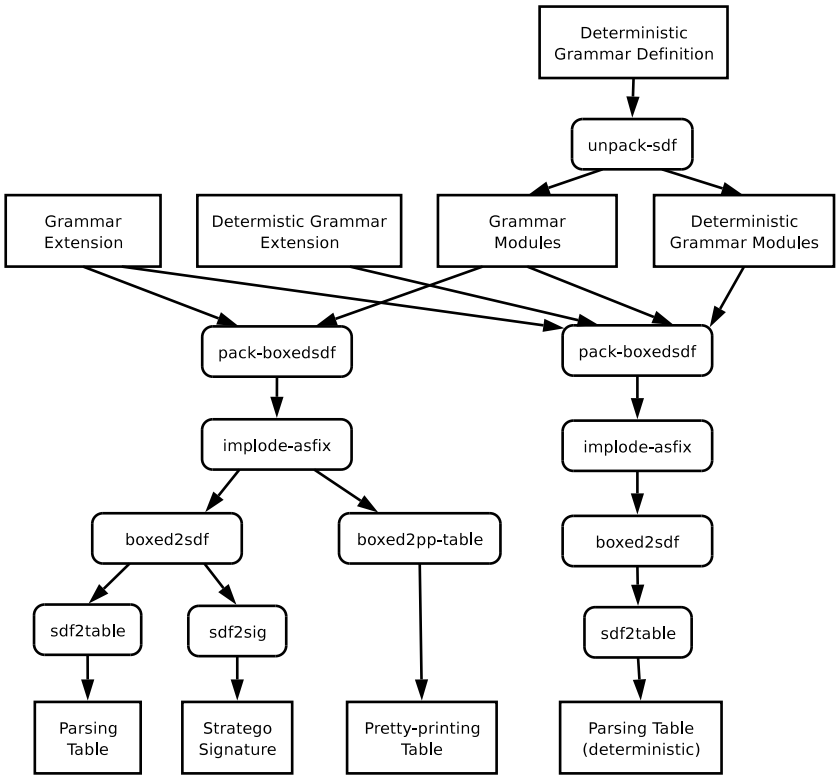
Since there is a new grammar, all tools have to be regenerated: the grammar parse table, the pretty-print table, the Stratego signature files. The same files have to be regenerated with the corresponding non-ambiguous grammar. To generate the non-ambiguous extension the original non-ambiguous grammar and the extended grammar have to be merged like in figure 2.14.

To generate all these grammar derived products, the processing is quite the same as the one described in section 2.2.1, but without some of the previous tools. The processing is shown on figure 2.15.

Figure 2.14 C++ extension

```
module SubclassDxx
imports SubclassCxx Dxx
```

Figure 2.15 Extended C++ grammar processing



2.4.2 Writing a transformation

There are two ways to write a transformation. One is to work on *asfix* trees. It allows to work on comments or deduce locations. The other way is to work on abstract syntax trees. The structure of abstract syntax tree is cleaner so transformations are easier to write.

The name of nodes in the abstract syntax tree is the constructor's name of the corresponding grammar production. This constructor's name is generated with the `sdf-cons` tool. This leads to many problems:

- The C++ grammar is big. There are nearly 400 context-free syntax production rules hence the same number of constructor's names.
- The constructor's names are generated from the production symbols. Some of them are very long and the default constructor's names become ugly.
- There are some constructor names that are very similar. For example, a *ClassName* identifier uses `Identifier9` as constructor's name, whereas a *UnqualifiedId* identifier uses `Identifier3`.
- The AST manipulation is easier, but not very natural. The transformation developer needs to have a deep knowledge of the grammar.

Concrete syntax

Stratego allows the specification of transformations using concrete syntax, as described in Visser (2002). The transformation rule seen in figure 2.16 can be written with concrete syntax as in figure 2.17.

Figure 2.16 Undo transformation using abstract syntax

```
Undo : do-while (s, e) -> StatementSeq-opt ([s, while (e, s)])
```

Figure 2.17 Undo transformation using concrete syntax

```
Undo : |[ do s while (e); ]| -> |[ { s while (e) s } ]|
```

The Stratego compiler will implode the concrete syntax into an AST. Hence, the writing of transformation, either with AST or with concrete syntax, stays exactly the same.

Extending the Stratego language with C++

The Stratego compiler can have its grammar easily extended, using a `meta` file. A new grammar must be written. This one has to include the C++ concrete syntax into the Stratego grammar. When a Stratego module is compiled, the compiler works on an AST where the concrete syntax part, when imploded, is contained in a `ToTerm` node. An example of StrategoCxx is shown in the figure 2.18.

Figure 2.18 StrategoCxx extension

```
module StrategoCxx
imports Dxx StrategoRenamed CxxVariables
exports
context-free syntax

  "|["  Declaration  "]|"  →  StrategoTerm  {cons("ToTerm"), prefer}
  "|["   Statement   "]|"  →  StrategoTerm  {cons("ToTerm"), prefer}
```

Sometimes, we have to match subtrees with variables, like when using the "?" operator in Stratego. We have to specify which variables have to be escaped from the concrete syntax to be Stratego variables. Figure 2.19 is an example where all variables beginning with "e", "f" or "g", followed by digits and quotes will be interpreted as meta variables that match expressions.

Figure 2.19 Cxx variables

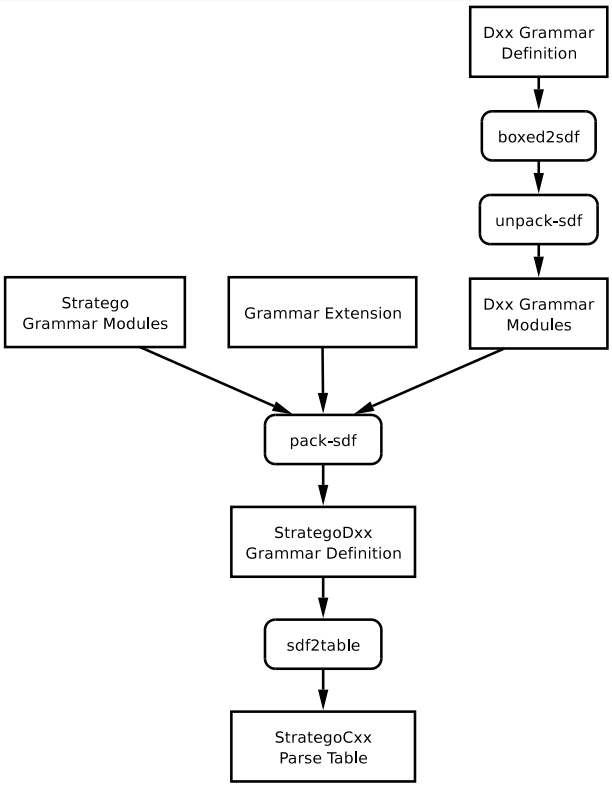
```
module CxxVariables
exports
variables

  [efg][0-9]*[\']*      →  Expression   { prefer }
  "s"[0-9]*[\']*"*"     →  Statement+   { prefer }
```

Since there is no C++ declaration in the StrategoCxx transformations, the disambiguation filters described in the chapter 3 cannot be used on them. That's why the non-ambiguous grammar is used in StrategoCxx instead of the classic C++ grammar.

The Stratego compiler, when finding a grammar in the meta file, tries to use the corresponding parse table. This parse table is made with the process described in figure 2.20.

Figure 2.20 StrategoCxx grammar processing



Chapter 3

Ambiguous parsing and parse forest filtering

This chapter focuses on selected parts of our approach to C++ program transformation. In particular, we show how our system relies on a non-deterministic parser assisted by a post-processing stage to deal properly with the C++ language, without sacrificing our grammar.

3.1 Overview

As stated in chapter 2, the architecture of our system is based on having a simple grammar for the language, assisted by a post-processing stage which aims at correcting its deficiencies.

This post-processing step is actually a collection of filters specified in Stratego, like any other transformation component. The major difference is that these filters work at a lower level, that is, directly on parse trees rather than on abstract syntax trees, to be able to see and handle parsing ambiguities.

In fact, the abstract syntax tree can keep ambiguities as well, but since it was not the case when the disambiguation filters were written, we are using the AsFix parse trees. Furthermore, working on the parse tree allows us to unparse the result, keeping the original source code layout and comments, which is valuable when the result of the transformation should remain human-readable.

An example of an input parse forest is given in figure 3.1, where ambiguities are depicted by diamond-shaped nodes. This sample tree shows how, from an ambiguous input text (program 3.2), the SGLR parser is able to produce a very concisely encoded parse forest[1].

Program 3.2 A simple declaration

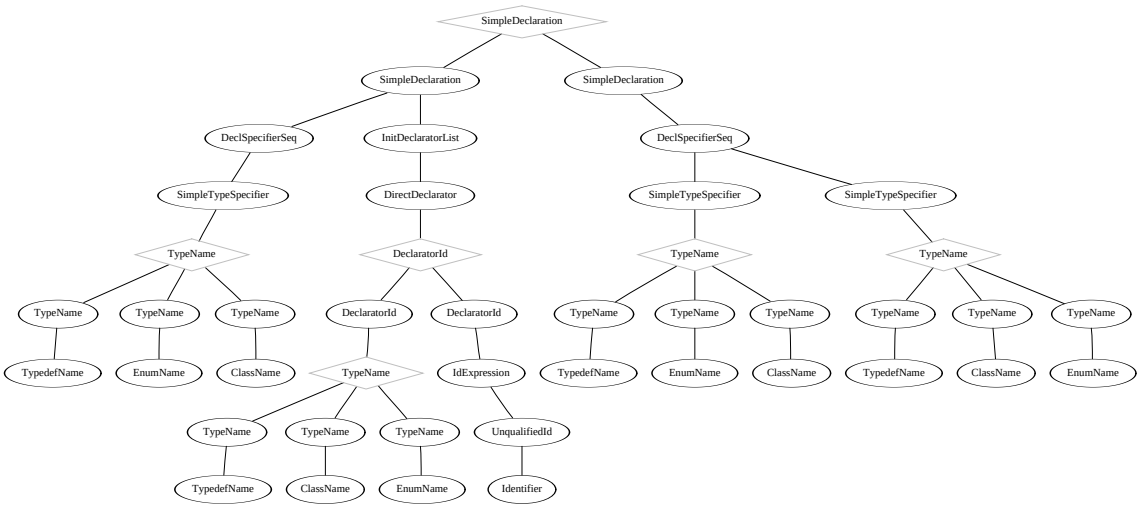
```
typedef int foo; // Removed from the parse tree for clarity.
foo bar;
```

The post-processing is performed on a forest of this kind, and is expected to completely remove all ambiguous nodes. The resulting tree is shown in figure 3.3. As expected, on every diamond-shaped node, a choice was made, and only one branch was kept; the forest is reduced to a single tree, which represents the only correct parsing of the input.

The filtering stage in itself is a multi-stage process, composed of many more-or-less complex transformation components, applied in the following order:

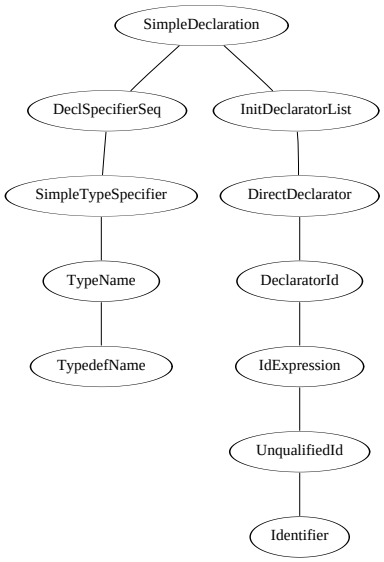
[1] A parse forest of this kind, while its size remains fairly reasonable, can store an exponential number of parse trees.

Figure 3.1 A parse forest



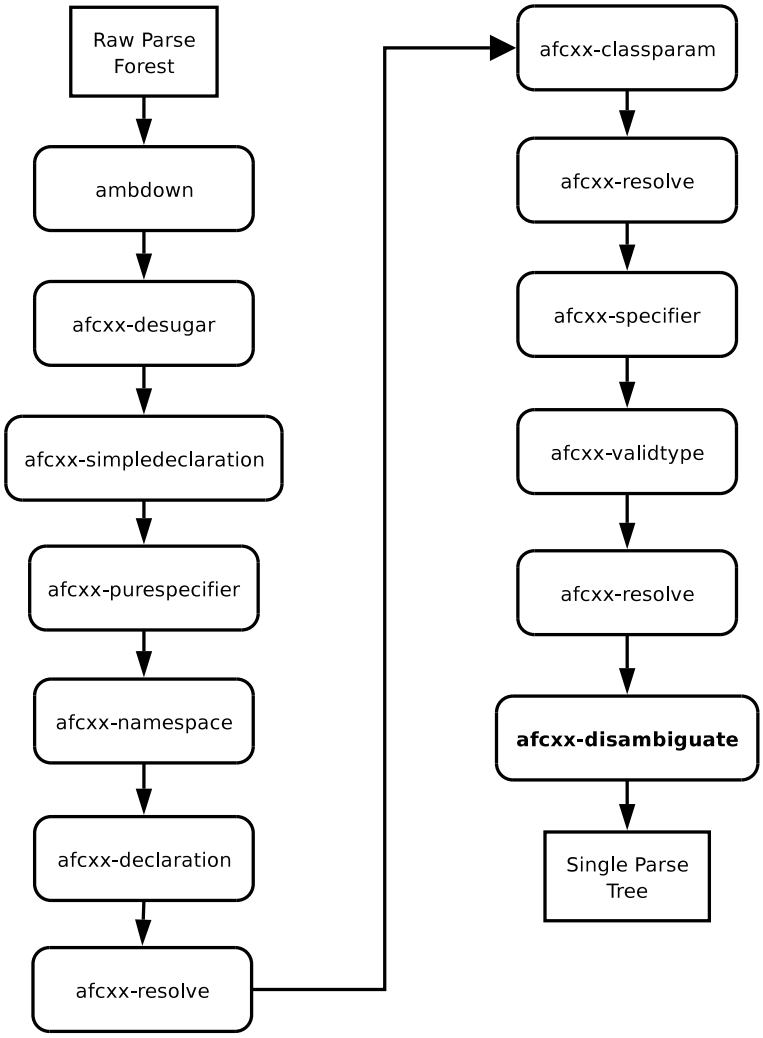
1. `ambdown` (3.2.1): This generic and language independent filter replaces the ambiguous nodes misplaced by the SGLR parser.
2. `afcxx-desugar` (3.3.1): `afcxx-desugar` desugar some basics constructions to simplify the AST.
3. `afcxx-simpledeclaration` (3.3.6): This filter delete illegal case of use empty type specifier in simple declaration when it is not a function declaration.
4. `afcxx-purespecifier` (3.3.3): Only virtual declarations can have a pure-specifier. The others have a constant-specifier.
5. `afcxx-namespace` (3.3.5): Namespace definitions need to be filtered, among other things because the standard C++ grammar makes a distinction between the first definition of a namespace, and the extension of an already defined namespace.
6. `afcxx-declaration` (3.3.6): As explained in section 2.2.1, declarations are a major source of ambiguities. This program is one of the several components that contribute to filter declarations.
7. `afcxx-resolve` (3.3.4): Some of our filters do not directly remove ambiguities, but cut branches they consider invalid. At some point, when an ambiguity node has only one child left, it can be removed. This generic ambiguity resolution is performed by `afcxx-resolve`.
8. `afcxx-classparam` (3.3.2): On ambiguities, only parameters viewed as typename are kept.
9. `afcxx-specifier` (3.3.6): This filter also processes declarations.
10. `afcxx-validtype` (3.3.6): This filter traverses the parse forest looking for wrong type qualifiers list.
11. `afcxx-disambiguate` (3.4): This is the last and most complex filter. When a parse forest has gone through the previous components, it has been reduced enough to be suitable for this large semantic analysis stage: `afcxx-disambiguate` walks the whole program to determine the kind of each symbol; with this knowledge, a second traversal finishes the reduction of the parse forest.

Figure 3.3 A post-processed parse forest



A summary of this processing chain in given in figure 3.4. More detailed descriptions of the components follow.

Figure 3.4 Filtering chain for parse forests



3.2 Global and generic filter

Some filters in our disambiguation process is generic and language independent. Such filters are describe into the next section.

3.2.1 SGLR misplaced ambiguities correction: ambdown

Sometimes, ambiguous nodes are misplaced by SGLR. Ambiguities appears too high in the tree. In practice, SGLR are right but theoretically we would except that the ambiguous node is more local. This problem is due to both optional layout and optional final term in an SDF rule.

The grammar describe by the SDF source code (figure 3.5 page 25) is a sample grammar which can generate a such misplaced ambiguous node in some cases. For instance, with the phrase “`a d`”, SGLR will produce the parse forest shown in the left side of the figure 3.6 (page 25). In this case, the problem comes from the `"A B? -> C"` rule and particularly from the last term `"B?"`. With the expression `"a d"`, SGLR inserts the space in the term `"C(A, B?)"` whereas it should be inserted in the term `"S(C, D)"`.

`Ambdown` looks for terms like `"C(A, B?)"` and moves the misplaced layout to the upper term. The result of `Ambdown` process is shown in the right side of the figure 3.6 (page 25).

Figure 3.5 SDF grammar with misplaced ambiguous node

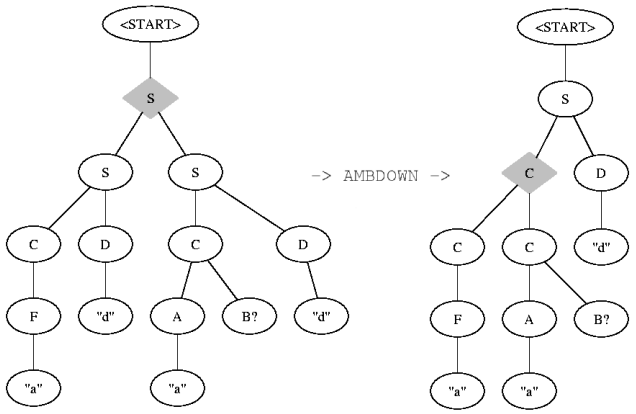
```
module Ambdown-sample
exports
  sorts S

    lexical syntax
      [\ \t\n]                  →   LAYOUT

    context-free restrictions
      LAYOUT?                  -/-  [\ \t\n]

    context-free syntax
      “a”                       →   A          { cons(“A”)   }
      “b”                       →   B          { cons(“B”)   }
      “d”                       →   D          { cons(“D”)   }
      “a”                       →   F          { cons(“F”)   }
      A B?                      →   C          { cons(“ABC”) }
      F                         →   C          { cons(“FC”)  }
      C D                       →   S          { cons(“S”)   }
```

Figure 3.6 `Ambdown` process



3.3 Local and specific filters

In this section, we discuss the simple filters that are applied to parse forests before the final analysis stage. In the processing chain described above, they range from `afcxx-desugar` to `afcxx-validtype`.

3.3.1 Desugar alternative tokens: afcxx-desugar

Alternative token representations are provided for some operators.

In all respects of the language, each alternative token behaves the same, respectively, as its primary token, except for its spelling.

The following table gives the alternative tokens, and the primary form of the token to which they are equivalent. Only the primary form is used in the AST, but the alternatives are recognized and desugared into primaries by `afcxx-desugar`.

Figure 3.7 Alternative tokens

Alternative	*Primary*	*Alternative*	*Primary*	*Alternative*	*Primary*
<%	{	and	&&	end_eq	&=
%>	}	bitor	|	or_eq	|=
<:	[	or	||	xor_eq	^=
:>	]	xor	^	not	!
%:	#	compl	~	not_eq	!=
%:%:	##	bitand	&		

3.3.2 Ambiguous template parameter: afcxx-classparam

Template parameters syntax is closed to function parameters syntax. But template parameters must be considered as typename whereas function parameters must be considered as value.

Program 3.8 Template parameters versus function parameters

```
template <class I>

struct A {};

struct B {};

template <B b>
struct C {};

template <class B b>
struct D {};

template <B>
struct E {};

template <class B>
struct F {};
```

The code of the program 3.8 is invalid, but has a good syntax. The template parameter `"class I"` of class template `A` is treated like a typename. But basic function parameters can be passed as template parameters. Then, the syntax can view `"class I"` as a typename or as a value where the declarator is omitted.

3.3.3 Pure virtual member functions disambiguation: afcxx-purespecifier

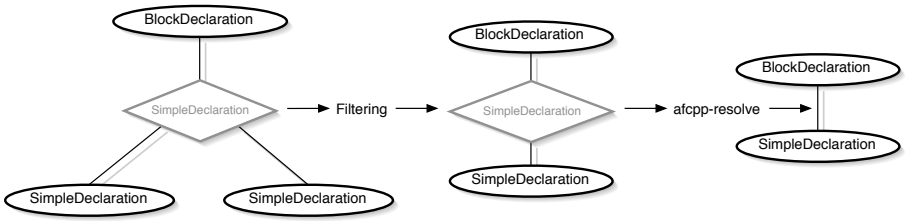
The C++ notation uses to specify a pure virtual member functions is ambiguous. Indeed, the `"= 0"` notation is similar to the notation uses to initialize a variable to zero.

This filter looks for such ambiguities in the parse forest and annotated only the `virtual` declaration with a `pure-specifier`. All the others case where `"= 0"` appears are considered as `constant-specifier`.

3.3.4 Generic ambiguity removal: afcxx-resolve

The simplest of our filters is `afcxx-resolve`. As described above, it is a generic filter which removes ambiguities that have already been reduced enough. An example of this process is given in figure 3.9.

Figure 3.9 `afcxx-resolve`



Since this filter is simple enough, we will comment here its Stratego specification as an example. It is given in program 3.10.

Program 3.10 `afcxx-resolve`

```
module afcxx-resolve

imports
  lib
  AsFix2-Syntax

rules

  Resolve : amb([a]) → a
            where <debug> ''Removed an ambiguous node''

strategies

  afcxx-resolve = iowrap(topdown(try(Resolve)))
```

As explained in section 2.1.2, the specification given above is composed of rules and strategies:

- Rules describe the actual transformations to be performed. In this case, we simply state that an ambiguous node to which only one subtree is attached, is transformed into this unique son.
- On the other hand, strategies are used to describe when and how the rewriting rules should be applied. In the case of `afcxx-namespace`, the processing we need is very simple: our

rule is applied during a top-down traversal of the input tree. Also, our rule is wrapped in a `try` strategy: each node for which our rule fails to match is left unchanged, which is precisely the expected behavior. Last, the `iowrap` strategy provides a very convenient interface with the system: it reads a tree on standard input, applies the strategy passed in parameter, and writes the resulting tree on standard output.

This example shows the benefits of using Stratego. With a very concise yet perfectly readable specification, we get a stand-alone software component that implements the wanted transformation: all ambiguities with a single branch left are removed.

3.3.5 Namespace definitions: afcxx-namespace

In the processing chain described earlier in this chapter, the very first filter applied to parse forests relates to namespace definitions.

The need for this post-processing is quite simple: in C++, namespace definitions do not have to be unique; actually, a given namespace can be defined at some place, but extended later with more members (a very basic example can be found in program 3.11).

Program 3.11 Namespace definition versus namespace extension

```
namespace foo { int a; } // First definition of namespace foo.
namespace bar { int a; } // First definition of namespace bar.
namespace foo { int b; } // Extension of namespace foo.
```

This distinction between the original definition of a namespace and its extension is reflected in our grammar, with two different production rules, shown in figure 3.12.

Figure 3.12 Production rules for namespace definitions

Identifier	→	*OriginalNamespaceName*
`namespace` *Identifier* `{` *NamespaceBody* `}`	→	*OriginalNamespaceDefinition*
`namespace` *OriginalNamespaceName* `{` *NamespaceBody* `}`	→	*ExtensionNamespaceDefinition*

Obviously, nothing in our context-free grammar enables us to express the right constraints: parsing ambiguities systematically arise from namespace definitions. As a consequence, a post-processing filter, `afcxx-namespace`, is applied to parse forests to disambiguate these definitions.

When stripped of its implementation details, the process is actually quite simple. A top-down traversal of the parse forest is performed. When an ambiguous namespace definition is found, its name is searched for in a symbol table. If the lookup is successful, the extension branch is kept. If the lookup fails, the original definition branch is kept, and the environment is updated. A more formal definition of this is given in algorithm 3.13.

While this filter is rather complex, compared to other components of our processing chain, it is the first one to be actually applied. The motivations for this are clear:

- Namespace definitions often are very large structures in C++ programs. Since they are initially duplicated, they significantly increase the size of parse forests. Since the ATerm format supports maximal sharing, a duplication does not require more memory. The real problem is the theoretical number of nodes, because tree trasversals will probably need to reach most of the nodes. Applying this filter at the beginning of the chain quickly reduces them to more reasonable sizes, and avoids the incoming filters to process duplicate data.

Algorithm 3.13 `afcxx-namespace`

$E \leftarrow \{\}$
top-down traversal
  for each ambiguous definition d of namespace n
  do
    if $n \in E$ **then**
      d is an extension of n
    else
      d is the original definition of n
      $E \leftarrow E \cup \{n\}$
    end if
  end for
end top-down traversal

- Namespace definitions do not depend on other structures of the language, which makes the early application of this filter possible. On the other hand, many filters need clean definitions of namespaces, and as soon as a filter plays with symbol names and lookups, it relies on the work done by `afcxx-namespace`: managing symbols while the parse forest is invaded with duplicate definitions does not seem very sane.

3.3.6 Post-processing declarations

When describing the various post-processing filters we have developed, we did mention several components dedicated to declarations. There are several reasons explaining this particular interest for declarations:

- At first, as explained in chapter 2, declarations in themselves are easily prone to produce parsing ambiguities; moreover, due to the great diversity of declarations in C++, these ambiguities are rather difficult to take care of.
- In addition to this, remember that the last part of our disambiguation chain is based on a semantic analysis step that solves ambiguities, in a general manner, by trying to determine the kind of every symbol in the program. To be able to perform this analysis, declarations must be processed enough to leave no ambiguity on the nature of the symbols being declared (aggregates, types, values, and so on...).

Therefore, to handle declarations properly, we apply several filters in sequence. Each filter of this collection manages a specific aspect of declarations, or implements a local heuristic.

Also, as it will become clear later on, splitting the post-processing of declarations into these many small pieces is not done only for simplicity or modularity purposes: some of the filters we will describe are dependent on tasks performed during previous passes. While we remain relatively free to modify the ordering of our processing chain, some of its components do not commute.

Declarations without declarators: afcxx-declaration

Recall the piece of grammar defining the syntax of declarations, given in figure 2.1. Actually, this rule states that a C++ declaration is only the concatenation of two lists:

- A list of specifiers which qualify the nature of the symbol(s) being declared. Some elements are always known to be specifiers (some keywords, such as `const`, `typedef`, base types, or some syntactic structures, for example class definitions), some are not.

- A list of declarators which name the object(s) being declared, and possibly assign a value to them. As in the case of specifiers, some constructions are clearly identified as being declarators, some are not, for example a single identifier.

Since at parsing time, some chunks of the input text cannot be classified as being specifiers or declarators, a declaration can often be read in several different ways, with respect to the grouping of tokens into the lists mentioned above. Program 3.14 gives some examples of ambiguous and non-ambiguous groupings.

Program 3.14 Grouping in declarations

```
int foo = 0;          // Not ambiguous.
int foo, bar;         // Not ambiguous.
foo bar;              // Ambiguous: ([foo bar], []) or ([foo], [bar]).
typedef foo bar;      // Ambiguous: ([typedef foo bar], []) or ([typedef foo], [bar]).
foo bar = 0;          // Not ambiguous.
foo bar, baz;         // Not ambiguous.
class A { };          // Not ambiguous.
class A { } a;        // Ambiguous: ([class A { } a], []) or ([class A { }], [a]).
```

Declarations are parsed most of the time with additional ambiguities, but the purpose of `afcxx-declaration` is only to handle this grouping issue. To this end, we only consider ambiguous declarations, and apply a very simple rule: their declarator list should not be empty; branches of declarations that do not satisfy this constraint are removed.

Of course, this is not the case for all declarations (typically, a class declaration does not have any declarator), but the declarations for which this rule does not apply do not suffer from any ambiguous grouping, and are therefore not seen during this stage.

The processing performed by `afcxx-declaration` is also described in algorithm 3.15.

Algorithm 3.15 `afcxx-declaration`

top-down traversal
  for each ambiguous declaration d **do**
    for each branch $b(b_s, b_d)$ of d **do**
      {b_s is the list of specifiers carried by b}
      {b_d is the list of declarators carried by b}
      if $b_d = []$ **then**
        remove b from d
      end if
    end for
  end for
end top-down traversal

Check for correct simpledeclaration: afcxx-simpledeclaration

This filter delete illegal case of use empty type specifier in simple declaration when it is not a function declaration.

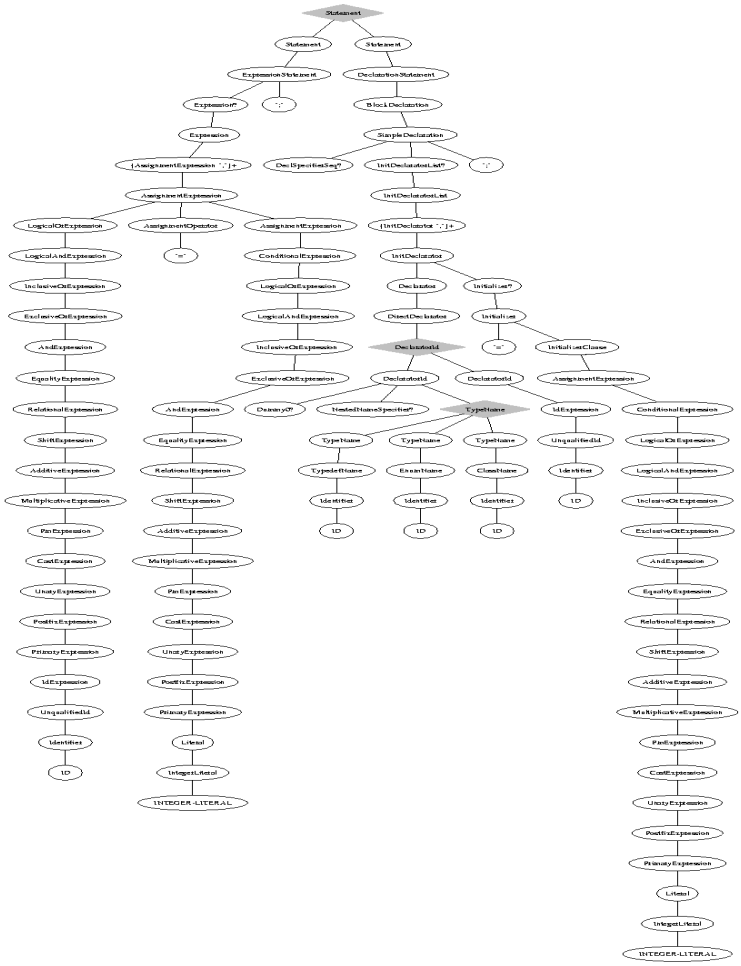
Consider the following C++ program:

Program 3.16 A simple declaration

```
int foo()
{
  int n;
  n = 1;
  return n;
}
```

Here, `"n = 1;"` can be treated as either a declaration or a expression because the `decl-specifier-seq` is optional in a `simple-declaration`.

Figure 3.17 A parse forest



The C++ standard authorize empty `decl-specifier-seq only` for functions, constructors, destructors and type conversion declarations. The other cases are deleted.

Ambiguous sequences of specifiers: afcxx-specifier

When both `afcxx-declaration` have been applied to the parse forest, declarations may still contain some ambiguities that can be processed at a local level.

A very curious problem arises in lists of specifiers, when qualified names are being used, and, once again, it is a matter of grouping.

Actually, many of the problems we encounter in declarations have the same root: the elements in a list of specifiers are not separated by any kind of token. This may sound quite harmless, but in practice, many forms of lists that involve qualified identifiers can be read in several ways. Some examples are given in program 3.18.

Program 3.18 Qualified names in lists of specifiers

```
typedef foo bar;                // Not ambiguous.
typedef foo::bar baz;           // Ambiguous: [foo::bar] or [foo, ::bar]?
typedef foo::bar::baz qux;      // Ambiguous: [foo::bar::baz] or [foo, ::bar::baz] or ...
```

As shown by this sample program, the size of the parse forest can quickly grow up as soon as some complex identifiers are being used. However, filtering such lists can be done with a very simple criterion: it seems pretty clear that, while a list of specifiers may be composed of many elements, primarily keywords (`static`, `const`, `virtual`...), it should not contain more than one type specifier. From there, we derive a new filter called `afcxx-specifier`, that applies the algorithm 3.19.

Algorithm 3.19 `afcxx-specifier`

top-down traversal
  for each ambiguous list of specifiers l **do**
    for each ambiguous branch l_i of l **do**
      compute the number of non-trivial specifiers k_i
    end for
    keep l_i such as $k_i = 1$
  end for
end top-down traversal

By applying this constraint, we are able to both disambiguate the lists of specifiers seen above, and reject some invalid inputs as well.

Detect wrong type qualifiers list: afcxx-validtype

In C++, you can type a variable with a list of type. The purpose here is to check the correctness of such a list. For instance :

Program 3.20 Well and ill formed qualifiers examples

```
unsigned long int       valid;      // OK.
unsigned long bool      unvalid;    // error.
```

This filter traverses the parse forest looking for wrong type qualifiers list.

3.4 Semantic analysis: afcxx-disambiguate

When a parse forest has been processed by all the components seen above, it is passed to the last filter, `afcxx-disambiguate`, that finishes the disambiguation stage by performing a global analysis of the input program.

This filter is rather complex when considered in all its details, but it is derived from a very basic and natural idea. At this stage of processing, all the remaining ambiguities are related to identifiers; the question being whether a given symbol is a type, or class, or value, etc. Therefore, to complete the reduction of the parse forest, the following two-stage strategy can be applied:

1. Traverse the parse forest, and gather informations from declarations. A collection of environments is built; each environment encodes a namespace of the input program, and associates in this namespace symbols to their kinds.
2. Traverse the parse forest, and for each ambiguous node, check its branches, and keep the correct candidate. In particular, on each symbol, perform a lookup in the relevant environments to check if its current interpretation is correct.

More details on this filter are given in this section. After the second pass, the filtering either failed[2], or completely reduced the parse forest. Pay attention, though, to the fact that we dot not yet guarantee any kind of correctness on a program that has passed the filtering process.

3.4.1 Classifying declarations

Above, we described the first pass of `afcxx-disambiguate` as a traversal that constructs the environments of a program out of the declarations. This description is not quite correct; in practice, all declarations are not systematically processed during this first pass.

Actually, this filter is implemented in two stages to enable us to properly manage constructs such as classes, where symbols may be used far before their definition (typically, public methods make use of private attributes defined later, see program 3.21).

Program 3.21 A class declaration

```
class Complex
{
public:

  Complex (float re,
           float im) : re_ (re),
                       im_ (im)
  {
  }

  inline float getRe () { return re_; }
  inline float getIm () { return im_; }

private:

  float re_;
  float im_;
};
```

[2]This means that the input program is ill-formed.

On the other hand, to be able to perform a correct analysis of the input program, there are many declarations that should not contribute to build environments during the first pass. This is the case for local declarations, since a local symbol cannot be used before its declaration.

Finally, we have two constraints on declarations that make it possible to determine which declarations should be considered during the first stage of this filter:

- In some specific constructs, symbols may appear before their declaration. This is the case for namespaces and classes, but not for local declarations.
- We need to build environments during the first processing pass, and reuse them during the second stage. Therefore, we have to find a way to name uniquely the symbols inserted into these environments. In C++, without any additional work (α-conversion, etc.), some symbols can be uniquely named, some cannot.

Luckily enough, these two sets nearly match: we need a preprocessing for constructs such as classes and namespaces, and these structures introduce named scopes that make the construction of our environments possible.

Some constructs that need a preprocessing stage cannot be associated easily with a unique qualified name, mainly local classes and anonymous classes. These have not yet been addressed in our framework.

3.4.2 Reading partially ambiguous declarations

The previous classification provides the very basic guideline that stands behind the last filter. We now know which declarations should be used to construct our "static" environments, and which declarations should only enrich these preexisting environments during the final traversal of the parse forest.

Of course, the filter is built on top of the previous disambiguation stages. At this point, while we still have a very ambiguous parse forest, we have reasonably reduced declarations; they are still ambiguous, but usable. In practice, these declarations give only partial information, but are verbose enough to determine the kind of every symbol being declared (whether it is a type, class, value, etc.).

Algorithm 3.22 is applied by `afcxx-disambiguate`. Notice how the second pass of the filter is a recursive checking process[3]; for each ambiguity, the different possible branches are disambiguated, but this is expected to fail on all branches but one. Failures are raised by incorrect subtrees, mainly identifiers seen as a wrong kind.

3.4.3 Handling template constructs

The C++ language allows us to use parametric polymorphism through the `template` keyword. At the parsing level, this means that any class name can be parameterized.

When we need to make a difference between several specializations of the same class, using only the class name as a 'kind' is not enough. Our current solution is to parameterize the class kinds with the kind of each parameter. This leads to a proper lookup mechanism that can actually find the symbols in the right template class, even if it is a partially specialized class.

[3] Actually, this filter associated with the previous passes is a refined version of the naive disambiguation algorithm. Disambiguation could be performed simply by checking every possible tree with an algorithm similar to algorithm 3.22, but the number of possible trees is exponential in the number of ambiguities.

Algorithm 3.22 `afcxx-disambiguate`

{build environments}
top-down traversal
  for each definition of namespace or class n **do**
    $E_n \leftarrow \{\}$
  end for
  for each declaration d of symbol s **do**
    s is declared in namespace n
    s is of kind k
    $E_n \leftarrow E_n \cup \{s : k\}$
  end for
end top-down traversal

{$disambiguate$}
top-down traversal
  for each ambiguous node $a(a_1, a_2, ..., a_n)$ **do**
    keep a_i such as $disambiguate(a_i)$ is successful
  end for
  for each symbol s seen with kind k **do**
    find namespace n in which s is declared
    if $E_n \vdash s : k' \neq k$ **then**
      fail
    end if
  end for
  for each local declaration d of symbol s **do**
    s is declared in namespace n
    s is of kind k
    $E_n \leftarrow E_n \cup \{s : k\}$
  end for
  for each scope in namespace n **do**
    save and restore E_n properly
  end for
end top-down traversal

Chapter 4

Conclusion and future work

In this report, we have presented the early stages of development of a framework for C++ program transformation.

We have described the additional software components that have been developed to work altogether with the tools SGLR and Stratego, as well as our original approach to the syntactic analysis of C++, a non-deterministic parser assisted by a bundle of disambiguation filters.

Yet, while the results are promising, there are still many limitations to our system, and much work to be done to achieve our primary goal of automatic derivation of active libraries.

4.1 Limitations

Most limitations of our system are related to the filtering process, where some constructs of the language are not yet properly handled. Among constructs of this kind, most problematic are:

- anonymous classes, which dot no fit yet in our simple name lookup mode,.
- class declarations local to functions, for the same reason,
- template-based constructions, which require an improved lookup mechanism, to be able to handle recursive template parameters.

4.2 Future work

Apart from the filtering stage, there are many more general issues that have not yet been addressed:

- The most critical point is the C pre-processor. Until now, we have not yet taken into account this stage, but simply applying our transformations after the pre-processing, as the compiler does, is not a satisfactory solution.

  When transforming programs, in particular when these are intended to remain human-readable, we cannot afford to let the pre-processor pollute the resulting source code, by, for example, copying into each file the definitions from the C++ standard library.

- Still in the context of transformations producing human-readable programs, we need a processing chain able to preserve comments in the code. This is actually not the case, since the implosion of parse trees into abstract syntax trees strips all layout information. Working solely on abstract syntax trees is, obviously, not the best suited method.

- Last, as explained in the previous chapter, our syntactic analyzer (parsing plus disambiguation) does not guarantee the syntactic correctness of the input programs.

While this deficiency was acceptable at the beginning of the project, with the C++ compiler acting ultimately as an oracle, this will have to be corrected sooner or later.

- The extending grammar method brings some problems for ambiguities resolution. It would be nice to have an `afcxx-disambiguate` filter that supports extensions.

Bibliography

(1998). ISO/IEC 14882:1998 (E). *Programming languages - C++*.

Brand, M. v. d., Deursen, A. v., Heering, J., Jonge, H. d., Jonge, M. d., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., and Visser, J. (2001). The ASF+SDF Meta-Environment: a component-based language development environment. In Wilhelm, R., editor, *Compiler Construction 2001 (CC'2001)*, volume 2027 of *LNCS*, pages 365 – 370. Springer-Verlag.

Brand, M. v. d. B., Scheerder, J., Vinju, J., and Visser, E. (2002). Disambiguation filters for scannerless generalized LR parsers. In *Computational Complexity*, pages 143 – 158.

Darbon, J., Géraud, T., and Duret-Lutz, A. (2002). Generic implementation of morphological image operators. In *Proceedings of the International Symposium on Mathematical Morphology VI (ISMM'2002)*, pages 175 – 184.

Géraud, T. (2002). Towards statically type-safe programming with the c++ language. Available on demand.

Jonge, M. d. (2000). A pretty-printer for every occasion. In Ferguson, I., Gray, J., and Scott, L., editors, *Proceedings of the Second Internal Symposium on Constructing Software Engineering Tools (CoSET'2000)*.

Jonge, M. d., Visser, E., and Visser, J. (2001). XT: a bundle of program transformation tools. In Brand, M. v. d. and Parigot, D., editors, *Proceedings of Language Descriptions, Tools and Applications (LDTA'2001)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers.

Jonge, M. d. and Visser, J. (2001a). Grammars as contracts. *Lecture Notes in Computer Science*, 2177:85 – ??

Jonge, M. d. and Visser, J. (2001b). XT capita selecta. In Visser, E., editor, *Proceedings of the Second Stratego Users Day*, number UU-CS-2001-54.

Lämmel, R., Visser, E., and Visser, J. (2002). The essence of strategic programming. (Draft).

Veldhuizen, T. (1999). C++ templates as partial evaluation.

Visser, E. (2000). The stratego library.

Visser, E. (2001). Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In Middeldorp, A., editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357 – 361. Springer-Verlag.

Visser, E. (2002). Meta-programming with concrete object syntax. In Batory, D., Consel, C., and Taha, W., editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315, Pittsburgh, PA, USA. Springer-Verlag.

Visser, E. and Deursen, A. v. (2000). Program transformation.org.

Index