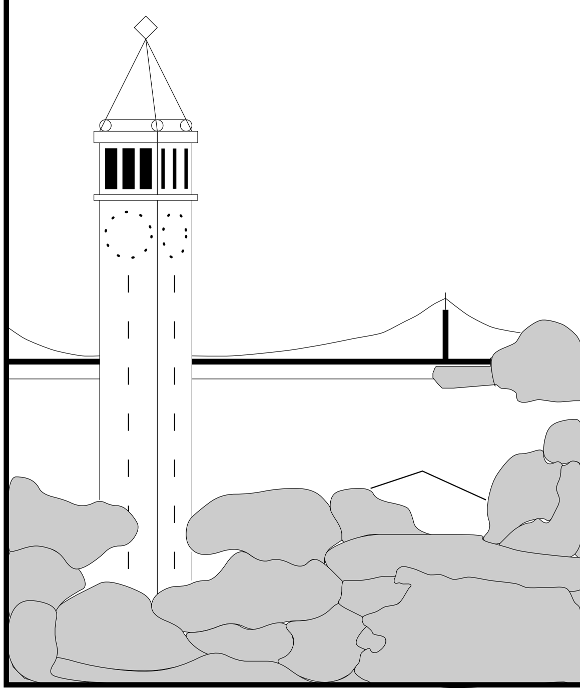
Transformational Generation of Language Plug-ins in the Harmonia Framework

Andrew Begel
Marat Boshernitsan
Susan L. Graham



Report No. UCB/CSD-05-1370

January 2005

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Transformational Generation of Language Plug-ins in the Harmonia Framework

Andrew Begel*
Marat Boshernitsan
Susan L. Graham

Report No. UCB/CSD-05-1370

Computer Science Division, EECS
University of California, Berkeley
Berkeley, CA 94720-1776, USA

{abegel,maratb,graham}@cs.berkeley.edu

January 2005

Abstract

The Harmonia framework provides an infrastructure for building language-aware interactive programming tools. Harmonia supports many languages through *language plug-ins*, which are dynamically-loadable system extensions generated from lexical, syntactic, and semantic descriptions. In this report, we describe our approach to generating Harmonia language plug-ins from a variety of domain-specific description languages. We present the process of configuring plug-in analysis components, the transformations for high-level syntactic and semantic descriptions, and the optimizations for generated code. This largely *ad-hoc* process makes our generation techniques expensive to create and difficult to maintain. We propose a new component-based architecture based on transformational generation, present its benefits, and outline several research directions that still need to be addressed by the generative programming community.

*This work was supported in part by NSF Grant CCR-0098314 and by an IBM Eclipse Innovation Grant.

Contents

1 Introduction

Generative programming is a technique long used in the programming language community. Lexers and parsers, which provide the front-end to a compiler, were originally coded by hand. Once it was realized that regular expressions and context-free grammars could be used for formal specification of syntax, tools were developed to generate lexers and parsers automatically from a specification. This automation not only alleviated the tedium of writing the compiler front-end, but also made it easier to ensure correctness and easy maintenance of increasingly complex programming language implementations. Techniques for generating other compiler phases were also studied, with varying degrees of success.

Today, almost all software is created and maintained using interactive tools, the most common being text editors. Although some text editors are quite simple, the ones used for programming often provide language-based services. For example, XEmacs [24], a popular open-source editor used by many programmers, supports language-aware syntax highlighting, indentation, and for some languages (notably Java), even provides basic integrated development environment (IDE) functionality. In most such tools, the language support is implemented in an *ad hoc* manner. In contrast, our Harmonia system is among those that provide infrastructure to support the creation of language-based tools that rely on formalized knowledge of a programming language to provide services. Harmonia's software architecture is designed to be language-independent in order to leverage its facilities to support many different programming languages.

Harmonia is an object-oriented framework providing an infrastructure for building language-based interactive tools [5, 12]. The framework includes incremental lexers and parsers, a static semantic analysis engine, and other language-based facilities. Program source code is represented by annotated syntax trees which are augmented with non-linguistic information such as whitespace and comments. The analysis engine can support any textual language that has a formal syntactic and semantic specification. Harmonia can be used to augment text editors to support language-aware editing and navigation of documents that are malformed, incomplete, or inconsistent (the document can remain in these states indefinitely).

The Harmonia framework has been successfully integrated with XEmacs and Eclipse [9] in order to facilitate our current research in high-level interactive transformations and voice-based programming. These integrated environments provide programmers interactive syntax highlighting, pretty printing, structural and semantic navigation and search-and-replace, structural undo, and hypertext annotations.

Experience has shown that existing tools for building compilers are inadequate for interactive language-aware applications [2, 15]. To support new programming languages, we have built several custom tools that translate descriptions of languages into executable *language plug-in* modules. These tools include not only traditional lexer and parser generators, but also generators for syntax tree node definitions (along with their attributes, fields, and methods).

To add a language to Harmonia, the system is given a lexical, syntactic and semantic description, which is compiled into a dynamically-loadable system extension. Descriptions exist for Java, C, C++, Titanium (a parallel superset of Java) [25], XML, Scheme, Cobol, and Cool (an object-oriented teaching language) [1]. Language-specific analysis details are encapsulated by the language plug-ins, which can be loaded on demand into a running Harmonia application. The notion of separating the language-independent analysis kernel from the dynamically-loadable language-specific components is not unique to the Harmonia system. Yet, it is a requirement for simultaneously supporting multiple programming languages, a feature expected from any language analysis framework.[1]

Harmonia language plug-ins consist of a number of components that parameterize the behavior of the framework for the particular language. These components are generated from domain-specific descriptions by several tools that constitute the generational component of Harmonia. The tools are largely *ad hoc*, that is they do not utilize systematic generational techniques.

In this report, we evaluate our tool design from a more formal perspective, in order to achieve more systematic generational tools. Section 2 sets up the framework for generating language plug-ins, explaining domain-specific terminology and specifics of the transformation process. Section 3 describes our current

[1] Thus Harmonia is not a "template-based" monolingual framework as are the tools produced by the Synthesizer Generator [17], ASF+SDF [21], and other similar systems.

approach to generation, and Section 4 puts forth a set of requirements for systematic transformational tools that which could replace the *ad hoc* approach currently used in Harmonia. Finally, we describe our ideal design for language plug-in generation tools, given the needs of the interactive program-development domain.

2 Language Plug-in Generation

The formal description created by a language implementer to describe a programming language must be transformed into a dynamically-loadable plug-in suitable for analyzing a user's program. In this section, we illustrate the process by which the plug-in module is generated from the description. The literature defines a more formalized version of our transformation process as *transformational generation* [10].

2.1 Domain-specific Concepts in Programming Language Implementation

The main domain-specific entities in compiler implementation typically stem from a lexical, syntactic and semantic description of a programming language. Lexemes and the regular expressions that describe them make up the lexical description of a language. Terminals, nonterminals, and productions comprise the grammar for a language. Types, symbol tables and scoping rules make up the entities used by the name resolution and type checking phases of semantic analysis. In this report, we concern ourselves with the lexical and grammatical domains and the structural form of semantic analysis, but not with the algorithms used for those analyses.

The user's program is the input to a lexer, which turns a sequence of characters into a sequence of lexemes. A lexeme is a character sequence described by a regular expression over the characters of the input stream. The possible lexemes emitted by the lexer are defined in a file that contains the lexical description of the programming language. A lexeme declaration may be annotated by one or more flags that control its properties and its interpretation by the runtime system.

Once a lexeme has been produced, it is passed to a parser, a program generated from a grammar description. The parser requests lexemes from the lexer and recasts them as terminals from the grammar. The parser reads sequences of terminals into groups called phrases. Each phrase is described by a production in the grammar, typically written in BNF (Backus-Naur Form), and illustrated in Figure 1b. Productions are logically associated with a nonterminal[2] – in fact, we often implement productions as instantiations of an abstract type given by the nonterminal. A right-hand side may consist of both terminals and nonterminals.

When a phrase is found by the parser, the runtime engine produces a tree whose root is a node representing the production and whose children are the elements of the right-hand side (the terminals and instantiations of nonterminals). As the input is read, the tree is recursively built up (in a bottom-up parser – top-down parsers build the tree from the root to the leaves) until the input is complete. The final tree is called the parse tree (or syntax tree) and represents the structural interpretation of the parser's input. Figure 1 presents a small sample syntax tree.

Semantic analysis occurs next. It normally begins with two phases, name resolution and type checking, although in some implementations these phases are combined into a single phase. Semantic analyses are not formally specified, nor automatically generated by Harmonia, but the template for the analysis code and the data structures are generated.

2.2 Configuration of Plug-In Features

Harmonia language plug-ins consist of four major components: a lexical analyzer, a parser specification, the definitions of the runtime data structures for syntax tree representation, and hand-written static semantic analyses. The Harmonia framework supports several kinds of lexical analyzers, two different parsing technologies, and a flexible runtime representation that can be configured by the language plug-in implementer. This section discusses how the various language plug-in parameters are configured.

[2]In grammar terminology, the nonterminal is called the left-hand side, and the remainder of the production is called the right-hand side.

Grammar Construct	Expansion	Comment
X $\rightarrow$ A B*[Sep] C	X $\rightarrow$ A B_STAR_SEQ C B_STAR_SEQ $\rightarrow$ ϵ $\mid$ B_PLUS_SEQ B_PLUS_SEQ $\rightarrow$ B $\mid$ B_PLUS_SEQ Sep B_PLUS_SEQ	Sequence of zero or more B's, optionally separated by Sep
X $\rightarrow$ A B+[Sep] C	X $\rightarrow$ A B_PLUS_SEQ C B_PLUS_SEQ $\rightarrow$ B $\mid$ B_PLUS_SEQ Sep B_PLUS_SEQ	Sequence of one or more B's, optionally separated by Sep
X $\rightarrow$ A B? C	X $\rightarrow$ A B_OPT C B_OPT $\rightarrow$ ϵ $\mid$ B	Optional occurrence of B
X $\rightarrow$ A (B | C | D) E	X $\rightarrow$ A BCD_CHAIN E BCD_CHAIN $\rightarrow$ B $\mid$ C $\mid$ D	Nested alternation of B, C, and D.

Table 1: EBNF to BNF grammar transformations. A, B, C, D, E, and Sep denote grammar variables that can stand for arbitrary sequences of grammar symbols.

2.2.1 Lexer and Parser Configuration

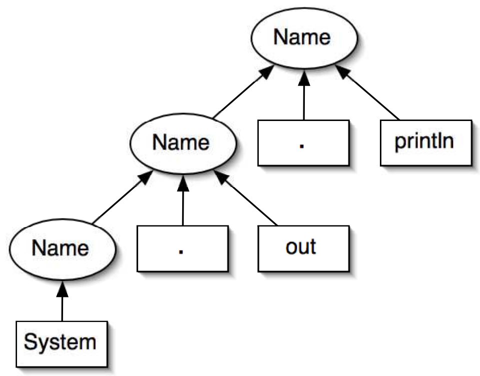
Harmonia's language analysis kernel supports the use of several different lexers with its parsing framework: two flavors of incremental lexers, and a voice-based lexer that supports program dictation (in this case, lexemes are produced by a speech recognizer). The decision of which lexer (or combination of lexers) is appropriate for a given language plug-in lies with the language implementer. To facilitate this choice, the description of the language plug-in is annotated with a list of acceptable lexers.

Harmonia also supports multiple parsing technologies. Currently, Harmonia contains LALR(1) and GLR-based parsers. An LALR(1) parser functions much like a traditional bottom-up parser, such as those generated by Bison [8] and YACC [13], but adapted for incremental use. The GLR parser is a Tomita-style parser [16, 20] engineered to support incremental parsing in an interactive setting [23]. As with lexers, Harmonia allows the language implementer to annotate the language plug-in description with a parser to use to process the incoming lexemes. Further discussion of the Harmonia analysis technologies is beyond the scope of this report, but is described elsewhere [5, 22, 3].

2.2.2 Configuration of Runtime Data Structures

The runtime data structure that Harmonia uses to represent program source code is a syntax tree. Much like the syntax tree of Figure 1, Harmonia syntax trees consist of nodes that represent terminals and nonterminals in the language grammar. The definition of syntax tree nodes is the major portion of the language plug-in. In addition to the generic features possessed by all nodes in all languages, each language plug-in supplies its own configuration, which determines which framework properties are included in the syntax tree nodes for that language.

The configuration is specified by annotating terminal and nonterminal declarations in the language grammar. Each annotation directly affects the memory usage of the resulting data structures, so using the minimum number of annotations is advisable. Annotations are not completely independent from one another. Harmonia language plug-in generation tools ensure that a consistent set of annotations is specified for each lexeme. As an example, each terminal in a language whose implementation permits input from dictation is annotated with the *PRONUNCIATION* property, which defines a set of strings that can be assigned as valid pronunciations of that lexeme in the speech recognizer. Many more annotations are permitted; they are not summarized here.



$$
\begin{aligned}
\textsc{Name} \;\rightarrow\;& \text{ID} \\
\mid\;& \textsc{Name}\ .\ \text{ID}
\end{aligned}
$$

(a) (b)

Figure 1: This figure (a) illustrates a small syntax tree corresponding to (b) a fragment of the Java grammar for qualified names. This syntax tree represents the input "System.out.println".

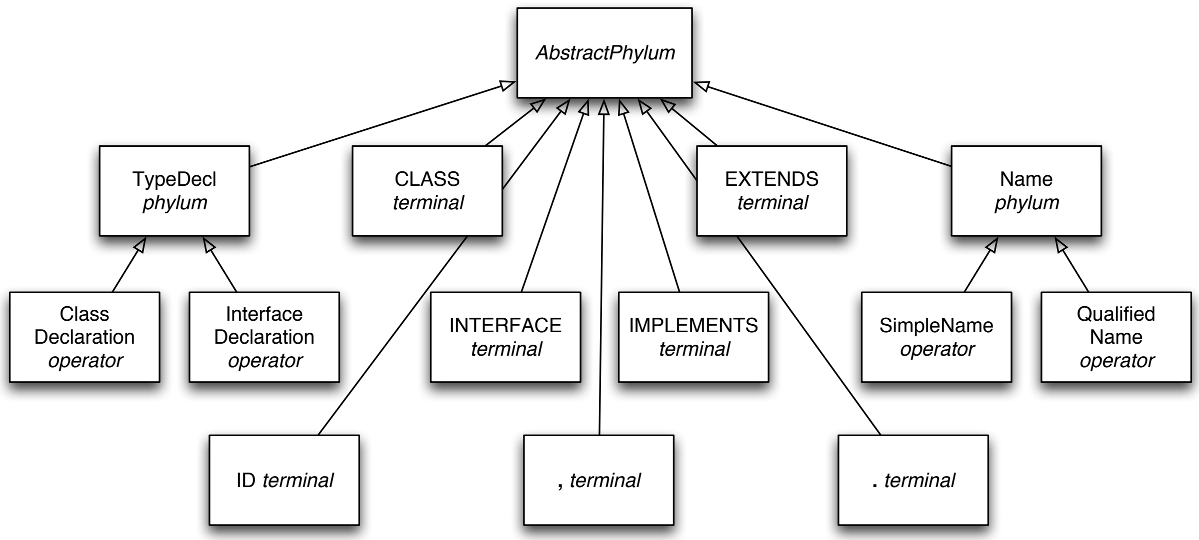
2.3 Grammar Transformation: EBNF to BNF

Productions in a Harmonia grammar are specified by a grammar description using EBNF notation. EBNF is an extension of BNF that allows succinct descriptions of sequences, optional elements, and alternatives. For example, "Name*" denotes a sequence of zero or more Names, "Name+" denotes one or more, "Name?" means that Name is optional, and "(Name1 | Name2)" denotes a choice between Name1 and Name2.

EBNF is a convenient notation for language implementers because it enables them to abstract some tedious details of their grammar specification and reduce the number of productions required to fill in sequences and optional elements in the grammar. However convenient this is for the programmer, most parser generators (including that used in Harmonia) can process only BNF grammars. We use a set of fairly straightforward grammar transformation to rewrite these EBNF grammars into BNF grammars. EBNF elements may be arbitrarily nested, so each nesting level is expanded out from outside-in via the transformation specification in Table 1. Several forms of EBNF notation are transformed:

- **Sequences.** There are two kinds of sequences, "plus" and "star". A "plus" sequence represents one or more occurrences of sequence elements. A "star" sequence represents zero or more occurrences. Sequence elements may be separated by a separator terminal or a nonterminal, which is optionally supplied following the sequence declaration. As indicated in Table 1, the "star" sequence translation is defined in terms of the "plus" sequence translation.
- **Optionals.** An optional element is found in the input exactly zero or one times.
- **Nested alternation.** Typically, to list a series of alternative right-hand sides, one needs to write a new nonterminal. EBNF nested alternation enables one to write an anonymous series of alternate expressions.

EBNF to BNF conversion may create duplicate productions. If a "star" sequence, for example, "Name*" were expanded, it would create two nonterminals: a Name_star_seq (and associated productions) and a Name_plus_seq. If "Name*" were expanded more than once, the new productions would be created again. Moreover, since the expansion of "Name+" creates Name_plus_seq, more duplication of nonterminals would occur. To prevent duplication, we memoize the generated nonterminals, and if they have already been created, we reference the existing one rather than create one from scratch.



(a)

TypeDecl	$\rightarrow$	CLASS ID (EXTENDS Name) (IMPLEMENTS Name+[,]) ClassBody	$\Rightarrow$ *ClassDeclaration*
	|	INTERFACE ID (EXTENDS Name+[,])? InterfaceBody	$\Rightarrow$ *InterfaceDeclaration*
Name	$\rightarrow$	ID	$\Rightarrow$ *SimpleName*
	|	Name . ID	$\Rightarrow$ *QualifiedName*

(b)

Figure 2: This figure (a) illustrates the phylum-operator class inheritance hierarchy for an excerpt of the C++ grammar specifying a method definition for (b) a fragment of the Java grammar (b). Each nonterminal is mapped to a phylum. Each production is mapped to an operator whose superclass is a phylum. Each terminal is mapped to an operator with no superclass. *Modifier*, *ClassBody*, and *InterfaceBody* phyla and operators are not shown in the graph.

Following EBNF to BNF transformation, we check that the grammar can be processed by the parser generator. A language implementer may accidentally introduce unbounded ambiguity in a production by nesting elements that may each derive an empty string. Validation routines are employed to catch this and several other obvious mistakes in the grammar.

2.4 Generation of Plug-in Components

Once the grammar has been placed in a suitable form, we perform two generation steps, emitting a parser specification and the definitions of the runtime data structures for the syntax tree representation. We also generate a lexical analyzer code from the lexical specification.

2.4.1 Lexer and Parser Generation

Following a pattern established in the early days of compiler writing, our lexer generator and parser generator (see Section 3.1) produce tables that are used by the runtime system to lex and parse the user's program. Since the generation performed by these tools is a conventional process, we do not describe these tools further. Section 3 gives a brief description of how these tools are used in Harmonia.

2.4.2 Data Structure Definitions

The Harmonia runtime representation for program source code is the syntax tree. While much of the generic tree manipulation code is provided by the framework, many per-language features are specified as in the language plug-in. The specification is achieved by generating language-specific definitions for classes of tree node objects.

The definitions of syntax tree node classes are generated in the intermediate textual notation called *ASTDef* (for AST definitions). After a number of operations on the ASTDef representation (Section 2.5), it is translated to C++ (Section 2.6). ASTDef is an extension to C++ that reflects important domain-oriented entities, and is easier to manipulate programmatically than standard C++. ASTDef classes are organized into two categories: phyla and operators. A phylum corresponds to a class representing a nonterminal, and an operator is a class that represents a production or terminal. Phyla and operators may contain *slots* (corresponding to C++ member fields), *attributes* (a field-like concept unique to Harmonia described in Section 2.5.4), and *methods* (the same as C++).

Phyla and operators are organized into a class hierarchy that directly reflects the relationships between nonterminals and productions in the grammar. For each language, there is an abstract phylum class that forms the root of the class hierarchy. Each nonterminal's phylum directly subclasses this abstract phylum. For each production associated with a nonterminal, the production's operator subclasses the nonterminal's phylum. Terminal operators directly subclass the abstract phylum. An example of this class hierarchy is given in Figure 2.

2.4.3 Static Syntax Tree Filtering

Because the Harmonia internal representation is essentially a parse tree, the grammar expansions that take place during the EBNF to BNF translation are exposed to all users of the tree data structure. Additionally, the Harmonia framework maintains non-linguistic material such as whitespace and comments as part of the syntax tree data structure. In order to provide a view of the syntax tree that is closer to the syntactic structure described by EBNF, the Harmonia framework employs a special mechanism called AST *accessors*. AST accessors are special methods generated on AST definition classes to enable navigating a filtered view of the syntax tree. The accessor method names are specified as part of the EBNF grammar for the language.

Consider the following example:

InterfaceDecl → INTERFACE *name*:ID *extends*:(EXTENDS *name*:Name+[,])? InterfaceBody

The example is a production from our Java grammar description that illustrates the specification of accessor names. Each symbol on the right-hand side has an optional AST accessor (written *accessor*:*symbol*) preceding it (e.g. the terminal `ID` has the accessor "name"). Some symbols on the right-hand side have nested accessors (e.g. "*extends*:(`EXTENDS` *name*:Name)?") – the outer accessor accesses the contents of the optional parenthesized expression, and the inner one refers to an instance of the Name nonterminal.

In addition to filtering non-linguistic material from the user's view of the syntax tree and providing access to the syntax tree nodes by their syntactic names, accessors implement abstractions over the BNF expansions of sequences, optionals, and alternations. Access to sequences is abstracted via an iterator interface that enables examining each element of the sequence, modifying the element, or inserting and deleting an element.[3] Access to an optional element consists of a boolean query "is there something there?" and an accessor to get the value, bypassing the nodes that represent BNF expansion. Access to nested alternation consists of a set of boolean queries "is it alternative A?", "is it alternative B?", etc., followed by accessors to retrieve the actual value.

[3]This abstraction from a traditionally written left-recursive or right-recursive sequence enables us to use a balanced binary tree implementation for sequences, accessing elements in $O(\log\ n)$ time.

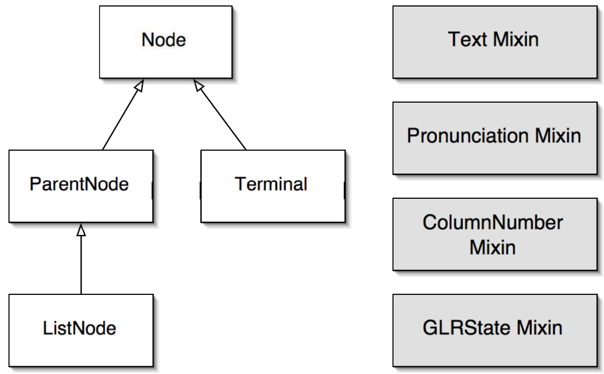


Figure 3: This figure illustrates a subset of the Harmonia framework class hierarchy used in the runtime system, including some of the mixin classes that implement runtime configuration (see Section 2.2.2.) Only four mixin classes are shown here: *TextMixin* – used for text storage in the terminals, *PronounciationMixin*, which permits definition of valid pronunciations for the lexeme, *ColumnNumberMixin* – used by languages that need to track the column in which the lexeme is used (e.g. Cobol), and *GLRStateMixin*, which is used in nonterminals constructed by the GLR parser.

2.5 ASTDef Transformations

Once the ASTDef representation has been generated by the lexer and parser generator, it is transformed in preparation for C++ code generation. This section briefly describes the set of transforming and generative operations that are performed on the ASTDef representation.

2.5.1 Merging of User-Specified Extensions

One of the hallmarks of object-oriented programming is the organization of code in terms of data types on which that code operates. Many researchers have noted that there are situations in which this organizing principle is detrimental to the quality of source code [19] and other, more functionality-centric organizations should be employed. Such is the case in the generated code for AST definitions. Each AST node encompasses several independent concerns cutting across the AST node class hierarchy. For instance, syntactic AST accessors, pretty printing, static program analyses (semantics, data flow, control flow, etc.) – all of these represent independent aspects. Furthermore, some of the static program analyses, such as name resolution and type checking, are separate concerns and could be further subdivided into independent phases.

Separation of concerns in AST definitions is achieved through textual merging of separate definition files. Each file consists of a set of phylum and operator class definitions. Phyla and operators defined in more than one file are merged together. (If a user defines the same slot, attribute or method in different files, the resulting code cannot be merged and an error is reported.)

In addition to merging phyla and operator definitions across files, ASTDef supports a simple code inlining feature. Code defined in inlineable templates may be merged into the definition of any phyla or operator. This feature facilitates semantic analyses that have identical implementations for methods in many different operators (such as type checking of arithmetic operators, or control flow computation).

2.5.2 Runtime System Behavioral Inheritance

The generated phyla and operator classes are connected with an extensive set of runtime classes that define most of the implementation of a syntax tree node. A small subset of the class hierarchy is shown in Figure 3. Rooted at the *Node* class, the hierarchy separates kinds of nodes based on function and memory storage. A *Node* is the basic superclass of all nodes in the syntax tree. Nodes contain much information, such as the type descriptor (used for reflection on the grammar properties of the particular node) and a pointer

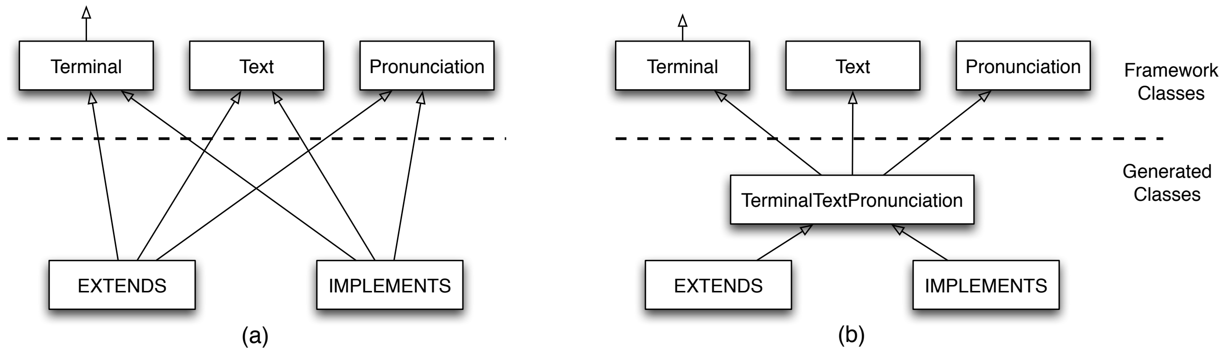


Figure 4: This figure (a) illustrates how two operator classes are supplemented with base classes from the framework's base hierarchy and (b) shows a superclass coalescing transformation, which reduces the amount of generated code.

to a node's parent node. In addition, much of the runtime class hierarchy's functionality is concentrated here (e.g. a set of reflection functions to retrieve node type information, tree traversal code, tree structural manipulation code, and attribute access code).

Non-leaf nodes (*ParentNode*) contain extra storage for a set of children pointers. Nodes that represent "plus" or "star" sequences (*ListNode*) contain methods that enable them to efficiently balance their subtree of sequence elements. All terminal operators inherit from the *Terminal* node class. Each operator may also inherit from one of several mixin classes. Each mixin class represents the runtime implementation of one of the terminal annotations described in Section 2.2.2.

During the ASTDef transformation process, each operator is made to inherit from the appropriate superclass. The selection of the superclass is influenced by the language plug-in feature configuration as well as by other configuration knowledge built into the ASTDef transformation tool. Figure 4a illustrates how the *EXTENDS* and *IMPLEMENTS* classes of Figure 2 fit into the framework hierarchy.

2.5.3 Class Hierarchy Optimizations

Optimization of the generated code is important to ensure readability of that code (should the plug-in implementer want to debug it) and to avoid overwhelming the C++ compiler. The latter is a much more important requirement than it seems. For instance, our Java grammar is translated into a class hierarchy consisting of 799 classes. That much source code puts significant strain not only on the compiler, but also on the debugger and the linker, each of which needs to deal with the enormous quantity of debug symbols in the object code.

ASTDef performs an important optimization in which the class inheritance graph is transformed by the *Extract Superclass* refactoring [11]. In this refactoring, we insert intermediate classes that represent common combinations of mixins. For instance, most terminals for a given language will contain a fixed set of mixins that represents all information that may be stored on a terminal for that language (e.g. each Java terminal is simply an aggregation of Terminal, and Text and Pronunciation). This combination of classes is refactored into an intermediate superclass, appropriately called *TerminalTextPronunciation*, which allows those terminals to inherit from a single superclass. This transformation is illustrated in Figure 4.

Superficially, such an arrangement may appear to increase the amount of generated code. However, this common superclass permits factoring of the generated runtime code which would otherwise need to be present in every leaf class. This optimization is slightly complicated by the observation that there will be some operators that end up being the only subclass of the newly generated intermediate class. In this case, the optimization is undone, since it does not help cut down on generated code at all.

To understand how useful this optimization is, in the Java language plug-in, it reduces the amount of code generated from 4.7 MB to 3.2 MB, a savings of 32%. In the C language plug-in, it reduces the generated code from 3.7 MB to 2.5 MB, a savings of 32%. In the C++ language plug-in, the size goes from 4.2 MB to 2.4 MB, a 43% savings.

2.5.4 Generation of Node Attribute Code

The Harmonia framework includes a mechanism for reflection on the syntax tree node objects called *node attributes*. The node attributes provide uniform named access to the wealth of information available to clients at every syntax tree node. Any node object may be queried for the set of available attributes and every attribute's value may be retrieved by its name. The set of attributes for every syntax tree node type is given by a declarative specification and is thus fixed during language plug-in generation. Accessing node attributes is functional: rather than reserving storage on a node for a piece of data, the attribute specification describes how to compute that data.

Node attributes are described in the ASTDef format using a simple declarative specification. Each attribute gives rise to generated methods for setting and retrieving an attribute's value as well as tables for efficient attribute retrieval.

Node attributes may be defined on operators and phyla as well as on the mixins and the base classes in the framework. The attributes are inherited by the operators, much like public fields and methods in C++. Because the set of attributes acquired through inheritance may be unique to every operator, efficient implementation dictates that the data structures for attribute access are defined in that operator's class. Consequently, we must traverse the class inheritance hierarchy, pushing attribute definitions from superclass to subclass. Once the traversal is finished, each operator contains a final listing of all attributes it will support.

2.5.5 Other Runtime Support Code

The final step before AST definitions are translated to C++ is the generation of special runtime support code for each operator and phylum. The Harmonia framework requires several features to be implemented in every AST node class; in this report, we discuss only generation of serialization code, which is sufficiently illustrative and not very complex.

Generation of serialization code is required if an operator or a phylum includes fields that need to be serialized when the syntax tree is stored in a file. Serialization code is also required when more than one superclass provides serialization methods. In this case, the serialization routines simply delegate serialization to each superclass in order. While generation of the boilerplate serialization code is not very interesting in itself, it presents another opportunity for optimizing of the amount of the generated code. Before generating code for a particular class, we compute the *needs-serialization-code* (NSC) predicate on every class in the hierarchy.

$$NSC(class) = \begin{cases} true & \text{if } class \text{ has fields that need to be serialized} \\ false & \text{if } class \text{ has no serializable fields and no superclasses} \\ \bigvee_{\substack{sc \in \\ super \\ classes}} NSC(sc) & \text{otherwise} \end{cases}$$

The NSC predicate governs whether the serialization code is emitted for each class, which allows factoring of the serialization code to the highest class in the inheritance hierarchy that requires it. For instance, for the hierarchy of Figure 4b, if the serialization code is not required for the *EXTENDS* and *IMPLEMENTS* classes, it will be generated in their common superclass *TerminalTextPronunciation*.

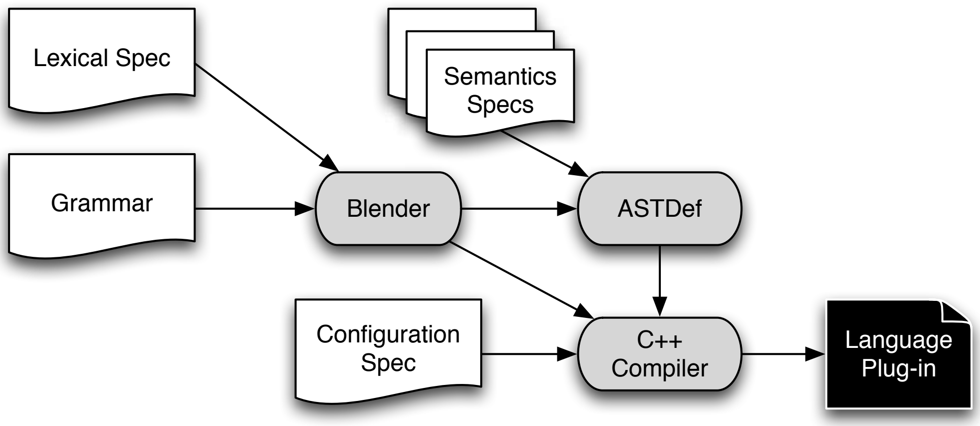


Figure 5: This figure illustrates how various input files are combined by the Blender, ASTDef and C++ compiler tools to produce a programming language plug-in.

2.6 ASTDef to C++ Translation

Following the ASTDef transformations described in the preceding section, AST definitions are translated to C++ code. Conveniently, ASTDef *is* for the most part C++. Phylum and operator classes can be implemented in C++ with a minimum of syntactic changes, mostly by deleting additional keywords added by ASTDef. C++ semantics, however, complicate our naive translation.

In order to satisfy the declaration before use semantics of C++, we must build a dependency graph of our class hierarchy. Operators with no generated superclasses come first (terminal operators), followed by phyla and then by the production operators. Sometimes the code for a method may access a symbol from another operator but this is not reflected by the class hierarchy. In these cases, the creator of the code (almost always the language implementer) must emit extra "depends" clauses in the declaration of the phylum or operator indicating the other operator. These depends clauses are used in constructing the dependency graph. When the code is finally emitted, we forward declare all classes first, and then the definitions for each class.

C++ requires that methods be declared in the class definition before their definition appears in code. When the C++ class definition for each operator and phylum is emitted, all ASTDef method definitions are transformed into method prototypes. A simplistic transformation would simply elide the body of the method and replace it by a semicolon. However, in the C++ standard, any default argument may only be declared in the method prototype, not in its definition. Thus, we have to strip out the arguments from the definition and place them solely in the declaration.

3 Harmonia Approach to Generation

The typical process for building a Harmonia language plug-in is shown in Figure 5. The input consists of a lexical specification, a grammar for the programming language, and a small hand-coded file describing the language module configuration. Optionally, the input may include extra code to be included in the generated definitions of the AST classes (see Section 2.4.2).

The lexical and syntactic specifications are processed by a combined lexer and parser generator named Blender. Blender produces a lexical analyzer, parse tables and ASTDef class definitions representing syntax tree nodes in the parse tree. AST definitions are subsequently checked, combined with any extra definitions provided by the language plug-in implementer, and translated into the C++ source code. Finally, a C++ compiler is used to combine the C++ class definitions, parse tables, the batch lexer, and the language module interface implementation into a dynamically-loadable library for the Harmonia language analysis kernel.

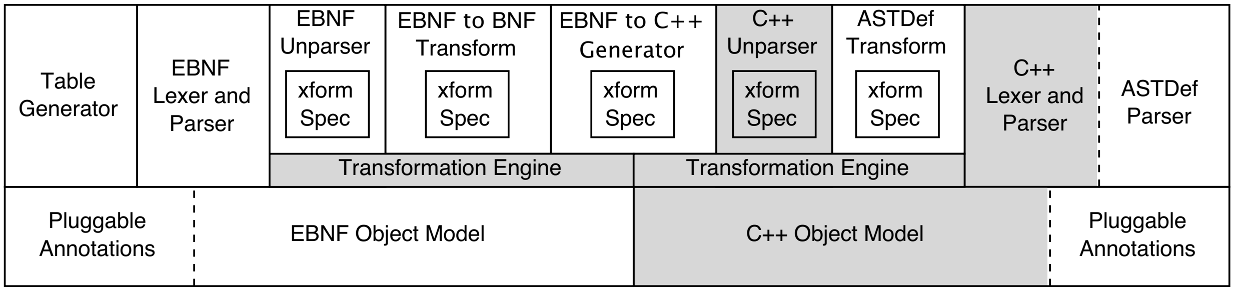


Figure 6: A new architecture for a plug-in generator. The shaded boxes are externally-provided components.

The following sections discuss the Blender and ASTDef tools in slightly more detail, concentrating on their transformational and generative facilities.

3.1 Blender: A Lexer and Parser Generator

The first function of Blender is to transform the high-level EBNF syntactic specification to a BNF notation amenable for processing by the parser generation algorithm. Blender processes EBNF grammar into an internal representation and carries out the sequence of three operations:

1. **Input Validation.** In this stage the grammar is checked for errors that would prevent its further interpretation by the tool. Such errors include duplicate or missing names, syntax errors, etc. The input validation step happens while translating the grammar into internal representation.
2. **EBNF to BNF Translation.** During this step, Blender transforms the internal representation of the grammar according to the rules in Table 1. Because BNF is a subset of EBNF, the transformations take place within the same data model. Since Blender does not allow language designers to design their own transformations, the transformations are hard-coded as sequences of low-level data structure manipulations.
3. **Grammar Verification.** The expanded BNF grammar is checked for semantic problems, such as certain ambiguities that cannot be handled by our parser generator. If such errors are found, the grammar is rejected.

Unlike typical parser generators, Blender does not generate any parser code. Instead, its output consists of parse tables that can be used by the Harmonia parser driver and AST node class definitions. When processing a GLR grammar, Blender outputs a conflict table which includes additional actions to be taken in parser states with shift/reduce and reduce/reduce conflicts.

The AST definitions are generated from the grammar as outlined in Section 2.4.2. Much of the code generation supporting our AST abstraction takes the form of C code that emits strings with ASTDef code inside. In some places, the code generated is pure boiler-plate code with no modifications (for example, code to integrate each generated class with the analysis runtime that uses that class). In most cases, however, we must pass in parameters to this boiler-plate code, turning it into more of a code template. All of the generated code is stored in string form within Blender, and is surrounded by the logic that decides what to emit when.

3.2 ASTDef: A Tree Definition Translator

ASTDef specifications are processed by a tool of the same name to yield a C++ implementation of the AST node classes. One source of ASTDef specifications is the Blender lexer and parser generator described in the preceding section. Additional AST definition code might be supplied by the language implementer, for hand-coded semantic analyses that traverse the syntax tree data structure. Automatically generated analyses might also produce AST definition code.

The first task of the ASTDef translator is to process all of the AST definition code. Since the ASTDef language is a derivative of C++, the specifications need to be parsed much like any other program. However, C++ is notoriously difficult to parse; thus, when we designed the syntax for ASTDef, we included some syntactic sugar that made parsing easier. Some modifications were to precede each method declaration with a *method* keyword, and each field declaration with a *slot* keyword. Additionally, method bodies are not parsed at all. Instead, lexical tricks are used to treat them as strings which are then stored within ASTDef's internal representation.

After processing AST definitions, ASTDef performs some simple validations such as checking that no method or field names clash (more rigorous error checking is left to the C++ compiler). It then carries out the transformations described in Section 2.5, and translates AST definitions to C++. ASTDef also generates all of the runtime support code described in Section 2.5.5. As in Blender, the generated code is handled as strings; very little of the underlying target language model is known to this simple tool. For instance, stripping out C++ default argument values (for generating method prototypes) is as simplistic as searching for equal signs in the string representing the method arguments.

4 A Systematic Approach for Generating Language Plug-ins

The objectives of the Harmonia project have not included building general-purpose transformational generators. As a result, maintenance and development of our specialized tools has largely been a necessary nuisance. Rather than relying on in-house technology, we would have liked to use transformation libraries and tools built by other researchers and/or practitioners.

Using transformation and generation components would afford us the following benefits.

- **Validating Generated Code** Code emitted by our tools consists of strings of text, rather than program structures. This prevents our tools from detecting any errors in the generated ASTDef and C++ code. The errors, ranging from simple syntax errors to gross semantic problems, can only be detected by the ASTDef and C++ compilers. A tool based on program structures would enable validation earlier in the generation process.

- **Simplifying Extensions** Currently, adding new features requires changing many places in many tools used in the generation process. For example, adding a new lexical annotation requires changing Blender and ASTDef. Moving to a component-based architecture would alleviate this problem.

- **Outsourcing Components** Leveraging standardized components built by outside providers would reduce our maintenance and development costs.

If we redesigned our system around pluggable components, the language plug-in generator would operate in two stages. The first stage would be responsible for the grammar transformations described in Section 2.3 and the translation of the grammar to the AST definition model (Section 2.4). The second stage would perform the transformations on the AST definition model and the final output to C++ (Sections 2.5 and 2.6).

Figure 6 outlines a hypothetical architecture for a language plug-in generator based on a more systematic generation methodology. At the heart of this architecture are the object models corresponding to EBNF and C++, the two languages used in the transformation process. An object model defines the data structures used for representing artifacts in these languages, as well as the API for manipulating these data structures. On top of the two object models are the numerous components that implement transformation and generation steps described in Section 2. The shaded boxes in the architecture diagram of Figure 6 represent the components we might expect to obtain from an outside provider; the unshaded boxes represent components we would build ourselves. Figure 7 illustrates the generation process within the new architecture.

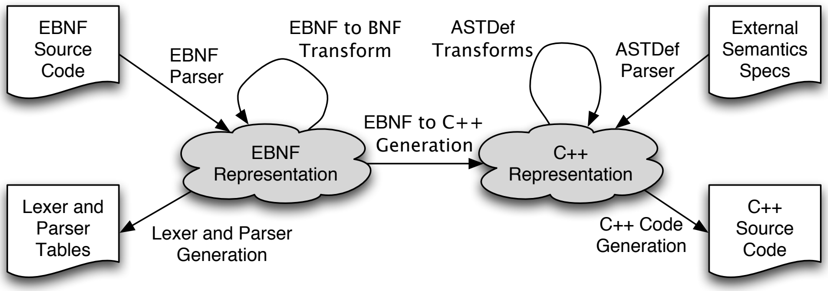


Figure 7: The new generation process supported by the architecture in Figure 6.

The following sections discuss the implementation of the various architectural components. We present the requirements for the externally-supplied components that would make them amenable to use in our toolchain. We also discuss how some of the in-house components could be developed through integration with providers' tools.

4.1 EBNF Object Model

Although most programming language grammars are written in BNF, grammar-processing tools differ in the internal representations they use. An EBNF object model is less standardized than BNF. Hence we would anticipate having to implement our own EBNF object model, rather than obtaining it elsewhere. Since the major user of the EBNF object model is the EBNF to BNF transformer, it might be possible to adapt the representation used by an externally supplied transformation engine (shown in Figure 6). This possibility is discussed in Section 4.3.

An important requirement for the EBNF object model is support for *pluggable annotations*. Although the overall structure of EBNF is fixed, the set of annotations used by the Harmonia language plug-in for configuration of runtime data structures (Section 2.2.2) changes as new features are added to the system. The EBNF object model must support the addition of new annotations (and the removal of deprecated annotations) without causing significant changes to the clients of the model. The EBNF object model must also support domain-specific operations on the grammar, implementing features like "list all nonterminals", and "for each production P of nonterminal NT, perform the following operation".

4.2 C++ Object Model

Unlike EBNF, C++ is a sufficiently widespread language that suitable implementations of its object model are readily available. One such provider is the Datrix project [4]; other implementations are also available. The provider's object model must cover the complete language syntax and semantics and not just its top-level structure. Treating C++ source code in a structured fashion has many advantages.

- It permits the process of generation to be described as a transformation on program structure, rather than as a sequence of text-emitting actions.
- It allows syntactic and semantic manipulation of the output that may be required by the language standard. Working with output source code structurally facilitates all three operations described in Section 2.6: generation of method prototypes and arranging classes in dependency order. Furthermore, structurally representing output source code would eliminate the need for our *depends* declaration, as the generator would be able to read the user's field initializers and method definitions to discover declaration order dependencies.

- It facilitates specification-driven pretty-printing of the output source code by "unparsing" the internal representation using one of the techniques published in the literature (de Jonge [14] is one of the more recent examples). Pretty-printing is an important enhancement to any generative tool since the generated source code is likely to be examined during debugging.

The object model for C++ must support high-level transformations including adding new classes and methods, and moving methods and fields between classes. It should also be possible to view and traverse the object model both as an abstract syntax tree and as a class hierarchy, as the latter view is required for some of the transformations performed during the generation process.

An important aspect of the tool is semantic verification. If a transformation invoked by the tool user would result in illegal code, the tool should issue a warning or disallow the transformation. (An example of a disallowed transformation would be adding a duplicate field or method to an object class.)

If this C++ object model approach were used, a custom ASTDef object model would no longer be necessary. As long as the C++ object model permits arbitrary annotations to be added to the entities representing classes and methods, we can augment the standard tool with the domain-specific features found in ASTDef (notably, phyla, operators, slots and attributes). To further the integration, the ASTDef parser, which is still needed to support manually written semantic analyses, could be modified to produce annotated objects in the C++ model directly. This scheme would enable ASTDef to remain domain-specific to the writer of a semantic analysis pass, while enabling us to use a standardized tool which we would not have to maintain.

4.3 Transformations in the New Generation Process

It would be very advantageous to use a general-purpose transformation system which can operate on the internal representation in a consistent and well-structured manner. Whereas most transformations described in this report can be implemented in an *ad hoc* manner using imperative code that directly manipulates internal data structures (as is the case for the Harmonia generative tools), it would be more appropriate to specify transforming actions using declarative rules. As illustrated in Figure 6, a flexible general purpose transformation engine could be used to implement the EBNF to BNF transformer, ASTDef transformer, EBNF to C++ generator, and the EBNF and C++ unparsers. We use the term "transformation" in a very general sense to refer to both *rewriting* and *generative* transformations. Rewriting transformations take place within a single data model. The EBNF to BNF transformations are of this flavor because a grammar specification constitutes both the domain and the target of the transformation (in that example, BNF is simply a subset of EBNF). In contrast, generative transformations produce a new data model (e.g. as in the BNF to ASTDef generation). The process of unparsing can also be viewed as a text-emitting transformation.

Many existing systems permit the declarative form of transformation specification. Notable examples include REFINE [6], and TXL [7]. The Intentional Programming environment [18] provides a complete programming environment that supports many operations based on program transformations. However, a system that could be successfully used for implementing generational transformations described in this report must allow for greater flexibility in the specification mechanism than that available in the existing tools.

4.3.1 Object Model For Transformations

Most existing program transformation tools operate on internal representations consisting of annotated syntax trees. These syntax trees correspond to the internal representations, which, in the case of Harmonia language plug-ins, consist of the domain-oriented input model, intermediate object model, and the target language model. However, the transformations for optimizing the class inheritance hierarchy described in Section 2.5.3 require operations on a different view of the intermediate representation. Operations on the semantic structure of the intermediate representation (e.g. the class inheritance graph) require that the transformation engine be sufficiently flexible to permit switching to this alternative view for some transformations, while providing access to the syntax tree representation for others.

Regardless of the object model, the transformation system must be able to validate that a transforming action does not violate the constraints of that data model. For instance, it should not be possible to inject a code statement in the context where statements are not permitted (e.g. outside of a method definition). Nor should it be allowable to introduce circularities in the class inheritance graph.

4.3.2 Domain-Oriented Transformation Specification

Traditionally, transformation rules are specified using *tree patterns*, a formalism for specifying patterns that match tree-like structures. The result of the transformation is usually represented by a parameterized *tree template*, which specifies how to generate a new subtree corresponding to that matched by the tree pattern. The values of tree template parameters can be computed programmatically or be based on the tree-pattern match. Unfortunately, tree patterns and tree templates, which are specified in terms of the shape of the syntax tree data structure, do not reflect the conceptual structure of the transformation. Not only does this create an inconvenience for the user of the transformation tool, but also makes templates and patterns very sensitive to the specification of the syntax tree model, which is likely to change over the course of development.

Some transformation tools (notably TXL and REFINE) take the necessary steps to permit specification of tree patterns and templates as source code fragments with appropriate meta-variables. Such specifications are subsequently parsed into the internal tree patterns and templates. In our opinion, this is the only permissible specification mechanism, as it allows the user of the transformation tool to specify the transforming action in domain-oriented terms. However, to our knowledge, these tools do not provide a mechanism for expressing more complex semantic relationships in source code patterns, such as a subtype relationship of two classes mentioned in a pattern, or the requirements that a method be defined in a particular class. Querying such relationships is an important requirement for implementing generation of the specialized code discussed in Section 2.5.5.

Last, many code templates instantiated by Harmonia tools constitute large chunks of boiler-plate code. For ease of maintenance, the transformation tool should allow for such templates to be stored outside of the transformation specification.

4.4 Scalability

Real programming language grammars tend to be of considerable size. The Harmonia grammar for Java consists of 201 productions which give rise to 799 ASTDef classes. The more sizable Cobol grammar has 317 productions that result in generation of 3,412 ASTDef classes. A completely translated Java definition consists of 1.5 million non-blank pre-processed lines (48 MB) of C++. Not only does this abundance of source code put significant stress on the GCC compiler used by Harmonia, but it also dictates that the ability to perform transformations on large internal representations within a reasonable time is an absolute must for any tool used for Harmonia language plug-in generation.

Several optimization mechanisms may be employed to reduce the size of the data model. One solution could be to avoid storing a complete syntax tree representation by unparsing non-essential regions to text and regenerating section of syntax trees on demand. For example, such an approach could be applied to ASTDef method bodies which are currently not transformed during language plug-in generation. Nevertheless, regardless of the solution, scalability is an important issue for any successful transformation tool and must be addressed by any system used for generative transformations described in this report.

4.5 Reaping the Benefits of the Component-Based Architecture

In this section we revisit the benefits we ought to achieve by a move to the component-based architecture described above. In addition to the original three benefits described in Section 4, we add two more reasons that would influence the future direction of the Harmonia project.

The transformation model, which is more structurally and semantically rigorous than our current means of emitting strings of text, would enable early detection of structural and semantic errors in the emitted code.

In addition, because the tool, rather than the C++ compiler, performs the validation step, error messages can be targeted toward the language plug-in implementer. Currently, since the language plug-in implementer never sees the generated code, C++ compiler error messages are incomprehensible.

By abstracting our tools to use both the EBNF and C++ object models for transformation and generation, we would be able to limit potential changes solely to that component. Additionally, because the component would be driven by a declarative specification, any extensions would not affect the component's code or structure, as they do now.

Acquiring the C++ object model and tools from an outside provider would reduce our developments costs significantly. C++ is a very complex language, for which a parser and semantic analyzer are difficult to construct. Our tools for processing C++ can only extract the top-level structure (class definitions and method declarations) which severely limits our ability to manipulate the generated code.

Soon we anticipate porting Harmonia to Java. This entails retargeting our language plug-in generation tools to emit Java instead of C++. With the current implementation of the Harmonia tools, changing the code generators to emit Java would present a significant challenge due to the liberal mixing of generator code and generated code templates. With the new architecture, we would merely replace the externally supplied C++ object model, lexer, parser, and unparser with a Java equivalent. Since the semantics of C++ and Java are so similar, the work required to adapt the EBNF to C++ generator, ASTDef parser, and ASTDef transformer would be minimal.

The main requirement of all of these tools is to provide a domain-oriented object model for a user to manipulate a program. This requires, for each target language, a lexer, parser and semantic analysis engine. The Harmonia framework, briefly described in this report, provides the necessary infrastructure upon which these components might be built.

Although Harmonia is designed for interactive use, it can be easily adapted for batch processing. In fact, we are using Harmonia to build our Blender lexer and parser generator tool. The tool is supported by two language plug-ins, one for lexical descriptions, and the other for syntactic specifications. In addition to using these plug-ins for batch processing, Harmonia enables us to leverage the *same* plug-ins for domain-oriented interactive use. For instance, integration with XEmacs enables us to provide the language plug-in implementer with services for these two new domain-specific languages.

The Harmonia framework and our Harmonia plug-in for the XEmacs editor are available from the project web site at *http://harmonia.cs.berkeley.edu*.

5 Conclusion

Generative techniques are often used to create programming tools. They are particularly powerful when combined with object-oriented programming. Generative programming techniques are instrumental in converting the domain-oriented notation to the object-oriented implementation. However, tools that use *ad hoc* generation techniques are expensive to build and difficult to maintain. A principled approach to implementation by generation is a promising direction, and would help make the implementation of Harmonia easier in the future.

References

[1] A. Aiken. Cool: A portable project for teaching compiler construction. *SIGPLAN Notices*, 31(7):19–24, 1996.

[2] R. A. Ballance, S. L. Graham, and M. L. Van-De-Vanter. The Pan Language-based Editing System for Integrated Development Environments. In *Proceedings of the Fourth ACM SIGSOFT '90 Symposium on Software Development Environments*, pages 77–93, Dec. 1990. Published as SIGSOFT Software Engineering Notes, volume 15, number 6.

[3] A. Begel and S. L. Graham. Language analysis and tools for ambiguous input streams. In *Fourth Workshop on Language Descriptions, Tools and Applications*, 2004.

[4] Bell Canada. DATRIX abstract semantic graph reference manual (version 1.4). Technical report, Bell Canada, May 2000.

[5] M. Boshernitsan. Harmonia: A flexible framework for constructing interactive language-based programming tools. Technical Report CSD-01-1149, University of California, Berkeley, 2001.

[6] S. Burson, G. B. Kotik, and L. Z. Markosian. A program transformation approach to automating software reengineering. In *Proceedings of the 14th Annual International Computer Software and Applications Conference*, pages 314–322. IEEE Computer Society Press, 1990.

[7] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[8] C. Donnelly and R. Stallman. *Bison: the Yacc-compatible parser generator*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, Bison Version 1.12 edition, Dec. 1990.

[9] Eclipse. `http://www.eclipse.org`.

[10] U. W. Eisenecker and K. Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[12] Harmonia project web site. `http://harmonia.cs.berkeley.edu`.

[13] S. C. Johnson. Yacc: Yet another compiler-compiler. 1978.

[14] M. d. Jonge. Pretty-printing for software reengineering. In *Proceedings; IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 550–559. IEEE Computer Society Press, Oct. 2002.

[15] W. Maddox. Incremental static semantic analysis. Technical Report CSD-97-948, University of California, Berkeley, Jan. 14, 1998.

[16] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[17] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A system for constructing language–based editors*. Springer–Verlag, 1988.

[18] C. Simonyi. The death of programming languages, the birth of intentional programming. Technical report, Microsoft, Inc., Sept. 1995. Available from `http://citeseer.nj.nec.com/simonyi95death.html`.

[19] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of ICSE'99*, pages 107–119, Los Angeles CA, USA, 1999.

[20] M. Tomita. *Efficient Parsing for Natural Language — A Fast Algorithm for Practical Systems.* Int. Series in Engineering and Computer Science. Kluwer, Hingham, MA, 1986.

[21] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF + SDF meta-environment: A component-based language development environment. *Lecture Notes in Computer Science*, 2027:365–370, 2001.

[22] T. A. Wagner. Practical algorithms for incremental software development environments. PhD thesis CSD-97-946, University of California, Berkeley, Mar. 11, 1998.

[23] T. A. Wagner and S. L. Graham. Incremental analysis of real programming languages. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–43, 1997.

[24] XEmacs: The next generation of Emacs. `http://www.xemacs.org`.

[25] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY, 1998. ACM Press.