Harmonia: A Flexible Framework for Constructing Interactive Language-Based Programming Tools

Marat Boshernitsan*
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776 USA
maratb@cs.berkeley.edu

June 2001

Abstract

Despite many attempts in both research and industry to develop successful language-based software engineering tools, the resulting systems consistently fail to become adopted by working programmers. One of the main reasons for this failure is the closed-world view adopted by these systems: it is virtually impossible to integrate them with any outside technology. To address this problem, and to create a flexible research infrastructure, we created Harmonia, an open framework for constructing interactive language-based programming tools. This report presents the architecture of the Harmonia framework. We briefly review the design of the two earlier Berkeley projects, the Pan and Ensemble systems, discuss their influences on the design of Harmonia, and present the organization and interactions of the major components in the Harmonia framework.

*This work was supported in part by NSF grant CCR-9988531, by NSF Graduate Research Fellowship, and by Sun Microsystems Fellowship to Marat Boshernitsan.

Contents

1 Introduction **1**

2 Historical Notes **2**
2.1 The Pan System . . . 2
2.2 The Ensemble System . . . 3

3 Design Requirements **4**

4 The Architecture of the Harmonia Framework **6**

5 The Harmonia Language Kernel **7**
5.1 The Ensemble Language Kernel . . . 8
5.1.1 From Grammars to Trees . . . 8
5.1.2 Run-time Grammar Representation . . . 10
5.1.3 Syntax Trees as Persistent Self-Versioned Documents . . . 11
5.1.4 Editing and Analysis Model . . . 13
5.1.5 Incremental Lexer . . . 13
5.1.6 Incremental Parser . . . 15
5.1.7 Coping with Errors . . . 16
5.2 Extensions to the Syntax Tree Model . . . 17
5.2.1 Node Properties . . . 17
5.2.2 Node Attributes . . . 18
5.2.3 Multiple Representations for Non-deterministic Parsing . . . 18
5.3 Toward a More Abstract Syntax Tree Model . . . 20
5.3.1 High-level Grammar Abstractions . . . 21
5.3.2 Tree-views and Node Filtering . . . 23
5.4 Static Program Analysis Framework . . . 24
5.4.1 Specialized Support for Tree-based Analyses and Transformations . . . 25
5.4.2 Program Units . . . 27
5.4.3 Incremental Static Program Analyses . . . 28
5.5 Interfacing External Tools . . . 30
5.6 Designing an Exchange Format . . . 30
5.6.1 Trees vs. Graphs . . . 31
5.6.2 Designing an Exchange Schema . . . 31
5.6.3 Encoding Program Text . . . 32
5.6.4 Open Issues . . . 33

6 Encapsulating Language-Specific Information **33**
6.1 Flex . . . 35
6.2 Ladle II . . . 35
6.3 Bison II . . . 36
6.4 ASTDef . . . 38
6.4.1 High-level Structure . . . 38
6.4.2 Phyla and Operators . . . 38
6.4.3 Slots and Methods . . . 39
6.4.4 Attributes . . . 40
6.4.5 C++ Declarations . . . 40

7 Programming Language Bindings **41**

8 Conclusions **41**

9 Future Work **43**

1 Introduction

A recurring theme in the history of software engineering research has been the desire to build state-of-the-art tools that bear on the task of creating, manipulating, and analyzing program source code. It has long been recognized that to provide the most leverage, such tools must treat the source code structurally, following the syntax and semantics of the programming language. For example, Szwillus and Neal [44] aptly note that "programmers at all levels of expertise can benefit from the typing time saved, the formatting, the error detection and, most significantly, the ability to conceptualize at a higher level of abstraction rather than focusing on the language details." However, despite extensive research, numerous prototypes, and more than a few commercial attempts, the vast majority of today's programmers still use simple text-based tools – editors, compilers, debuggers – the technology fundamentally unchanged for twenty years.

The technology for constructing language-aware software engineering tools has been around since the early 1980s. Systems such as PSG [1], Centaur [8], Mentor [17], Synthesizer Generator [40], Gandalf [35], and many others have pioneered the use of language-based technology for programming tools. Yet, these systems failed to become widely adopted by computer programmers. Some of this resistance stems from the fact that the early systems failed to address the basic usability issues arising from the use of structure-based editing technology. Lang [26] notes that the advantages gained through use of the advanced editors do not sufficiently outweigh the inconveniences involved in learning and using a complex tool; Neal [32] concludes that as a result such systems are not likely to be accepted by the expert users.

However, a significant hindrance to acceptance of language-based programming tools is the closed-world view implied by the use of structure-based tools. Only those services provided within the programming tool are available to the user. The integration with external components is not possible. The development of third-party extensions is difficult, at best. Anecdotal evidence from USENET newsgroups suggests that these are precisely the reasons for the lukewarm acceptance of non-text-based commercial integrated development environments.

A related problem is faced by the researchers wanting to reap the benefits of language-based technology for their research prototypes, but not wishing to develop that technology *per se*. No open architecture currently exists that allows the research community to integrate with the existing program analysis technology or ensure interoperability of research tools. While this issue has been recently addressed in a series of workshops on a standard exchange format for program data (WOSEF 2000, WCRE 2000), no satisfactory solution has been proposed. The diversity of requirements and insufficient extensibility of existing analysis tools, drive researchers to constantly re-invent their own.

We created the Harmonia framework to address precisely these problems. The architecture of Harmonia is open and extensible and is aimed at constructing a large variety of tools. At one end of the spectrum are small tools for operating on individual source files, at the other are the larger systems for manipulating entire software projects. The Harmonia framework provides basic syntactic and semantic analysis services. Because our goal is to facilitate building *interactive* tools, these services are *incremental*, that is the internal program representation maintained by the framework is updated as a program is manipulated. The framework supports analyses of programs in many programming languages. In addition to built-in analysis services, the framework is easily extensible by third-party analyzers.

"Language" can be construed broadly – in addition to programming languages, developers use system architectural description languages, design languages, specification languages, command languages, scripting languages, document structuring or layout languages, etc. The languages we have in mind here are primarily those that have a syntactic and semantic structure akin to programming languages. Since there is no generally-accepted word for artifacts written in those languages, we use the term "programs." The context should indicate when the discussion pertains only to programming languages.

The Harmonia framework is an outgrowth of more than 16 years of research into language-based programming tools conducted at University of California, Berkeley under the guidance of Professor Susan L. Graham. The two earlier prototypes, the Pan and Ensemble systems described in Section 2, provided important insights into development of language-sensitive tools, leading to the design of the framework presented in this report. This design is discussed in Sections 3-7; Sections 8 and 9 presents conclusions and future research directions.

2 Historical Notes

This section discusses the Pan and Ensemble systems, including their research contributions, their shortcomings, and their influence on the Harmonia framework design.

2.1 The Pan System

The Pan system [5] was under development from 1984 to 1992, resulting in a fully-functioning Pan I prototype [6]. The focus of the Pan system was on editing and browsing of structured documents such as formal designs, specifications, and program components. The primary motivation of the Pan project was to provide rich user services based on document structure, with the understanding that such structure can, in most cases, be derived by an analysis of a textual document, rather than having to be explicitly manipulated by the user. The design of the Pan system is based upon the following assumptions:

- **Understanding rather than authoring is the primary activity.** Maintenance of a software system represents a significant portion of its life-cycle. The developers spend far more time trying to understand and modify programming artifacts than creating them in the first place [21]. Thus, a successful system must support a rich set of user-originated queries and provide a mechanism for presenting query results to the user.

- **There are many languages.** In addition to programming language, developers use a multitude of other formal and informal languages when working on a software system. Configuration languages, scripting languages, and document markup languages are just a few examples. Yet, a developer should not be required to learn a new editing system for every language. It is, therefore, imperative to provide support for this diversity of languages within the same system in a consistent and useful manner. Such support must accommodate the multitude of embedded languages found in many programming systems, for example, an SQL query in a C program. The system must also allow for evolving its language-specific capabilities as new languages are introduced and old languages change. The ways a language is used must also accommodate the changing requirements of new users and projects.

  Thus, the services of a language-based environment must be extensible and may grow with time. An effective language-based system must provide a language-description mechanism that can be used to define structural and presentation (fonts, colors, etc.) aspects for every language. Such a mechanism must also extend to describe language-specific services available to the users.

- **Users are fluent with their language.** An effective language-based editing system must provide for unrestricted text editing on the assumption that users are well-versed in their primary programming language. While programming artifacts may often appear ill-formed or incomplete, this is a by-product of how people work, rather than their incompetence with the medium. At the same time, even experienced users sometimes need extra support when confronted with an unfamiliar language, and such support should be readily available from the editing system.

As a direct consequence of the above requirements, the design of the Pan system is centered around the notions of syntax recognition, incrementality, coherent user interface, extensibility, and customization. The Pan system presents itself to the user as a syntax-recognizing editor.[1] The editor supports incremental document analysis, and provides a consistent and coherent interface tolerating ill-formedness, incompleteness, and inconsistency. The editor is extensible through a modular language description and is customizable through an extension language. The extension language for Pan is Lisp, one of its implementation languages. Figure 1 shows a typical Pan editing session.

The development of Pan resulted in significant research contributions. Ballance et al. [3] describe the use of grammatical abstractions and their relationship to incremental parsing algorithms. Butcher [11] presents Ladle, the grammar processing tool used in Pan to implement these abstractions. Ballance [2, 4] describes a

[1] *Syntax-recognizing* editors [19] are distinguished from *syntax-directed* (also known as *structure-oriented* [34]) editors by allowing the user to input documents as text and relying on subsequent analyses to recover document structure.

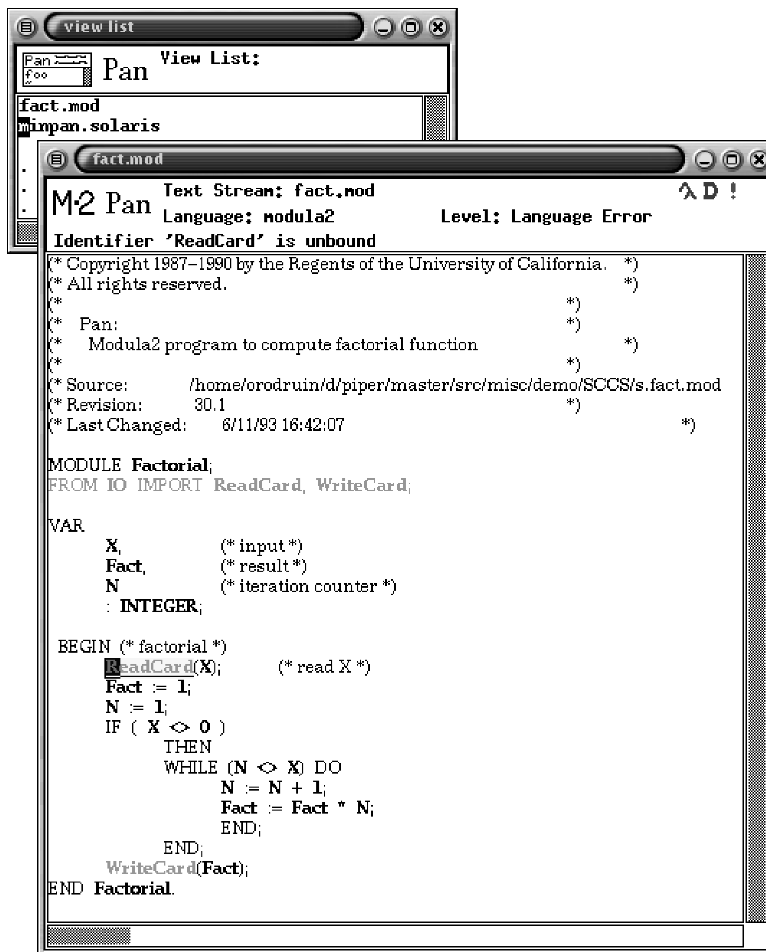


Figure 1: A Pan session. A small Modula-2 program has been analyzed and shows a semantic error.

mechanism for incremental semantic analysis based on logical constraint grammars. Van De Vanter [48, 49] presents new insights into building usable language-based systems based on his fine-grain analysis of the editing model presented to the user.

Yet, the Pan system fell somewhat short of its long-range goals. Language description techniques were aimed at a particular class of formal language ($LALR(1)$). Only one language per document was supported. No project management capabilities were available. An analysis could span multiple loaded documents, but only within one language. Moreover, all documents in the same language had to share the global program analysis database, making it impossible to work on more that a single program at a time. No configuration management was possible: information about language libraries was built into the system. A visual presentation mechanism was noticeably lacking. Issues such as support for novice programmers, graphical display and editing, and a persistent program database were deferred to Ensemble, the next generation system.

2.2 The Ensemble System

The Ensemble system [22], a successor to Pan, was under development from 1989 to 1997. The goal of Ensemble was to combine language-based technology of the Pan project with advanced document technology based on the VORTEX system [12]. The VORTEX system (for Visually ORiented TEX) was intended for the interactive development of documents using the TEX/LATEX description language and algorithm.

Based on the lessons from the VORTEX project, one of the major focuses of Ensemble was on developing

a structured document model that would be broad enough to accommodate all artifacts of program development, including, but not limited to, program source code and documentation. The Ensemble document model is object-based, with each document's structure being defined through a declarative specification [15]. (For a programming language, this would be the language grammar). A collection of rich device-independent presentation services is provided on top of the document model, facilitating device-dependent views of the presentation. Munson [31] describes a model of presentation based on attribute propagation, box layout, tree elaboration, and interface functions. Maverick [29] presents a different model based on tree transformations. Both presentation models are designed to be applied to a diverse array of documents drawn from many different media. Modeling structured data as persistent self-versioned documents [56] enables Ensemble's effective control-integration mechanisms. Compound documents are supported through structural nesting.

On the language analysis front, Ensemble achievements consist of new algorithms for incremental lexical and syntactic analyses. These algorithms, presented by Wagner [55, 59, 57], possess several novel aspects: they are more general than previous approaches, address issues of practical importance, including scalability and information preservation, and are optimal in both space and time. Since deterministic parsing is too restrictive a model to describe some common programming languages, Wagner [58, 55] also investigates support for multiple structural interpretations based on incremental Generalized LR (GLR) parsing. The incremental semantic analyzer is based on the Colander II system [28], which transforms an attribute grammar-based description of the analysis into a compiled incremental attribute evaluator that is applied to the syntax tree whenever semantic information is requested. In the spirit of Pan, Ensemble is a multilingual environment, simultaneously supporting multiple documents, each in a different language.

To the user, Ensemble appears as a multi-window, syntax-recognizing editor, much like Pan with an updated user interface. Multiple editable views and multiple different presentations of the same underlying document are possible (Figure 2). Ensemble provides a Scheme-based extension language, ExL, through which many aspects of the system, including the user interface, can be manipulated [14].

While Ensemble was a successful research project, the prototype was never as complete as the Pan system. To a large extent, Ensemble suffered the second-system effect, resulting in an over-arching design that was never entirely realized. This partly resulted from the two conflicting goals put forth by the Ensemble designers: using formal language analysis technology for natural language documents, *and* using the natural language presentation technology for programs. While resolving many of the research questions unanswered by Pan, the Ensemble architecture was not sufficiently extensible to facilitate broader investigations into constructing effective tools to assist software developers. The description mechanism for incremental semantic analysis was unnecessarily monolithic. Integration with external analysis tools was not possible. Even rudimentary project management was not supported. The overall complexity of the editing mechanism hindered any experimentation with advanced support for authoring and manipulation of programming artifacts.

3 Design Requirements

Development of Harmonia began in 1997. The primary motivation of the project was to build a flexible infrastructure capable of supporting research on language-based programming tools not otherwise possible within Pan or Ensemble systems. The Pan system was unsuitable due to its complicated three-tiered source code representation [6], somewhat dated program analysis technology, and archaic user interface. (Indeed, the first Pan prototype was developed before the X Window System!) The Ensemble system was unnecessarily bogged down by premature architectural commitments. Its reliance upon sophisticated presentation technology and tightly integrated analysis services made it difficult to extend the system and prototype new features and services. Instead, the vision of the Harmonia project was to develop an open *framework* rather than a monolithic system. Such a framework would allow researchers to rapidly prototype new applications benefitting from the source code representation and analysis services. Additionally, openness of the framework would facilitate integration with existing tools such as editors, code browsers, program analyzers, etc., without having to re-implement them within our system.

The scope of intended applications for the Harmonia framework is very broad. At the time of this writing, we are conducting research in tools that allow the programmer to manipulate source code in terms of high-level linguistic structure and semantics [9], and compose and edit programs by voice and gestures [7].

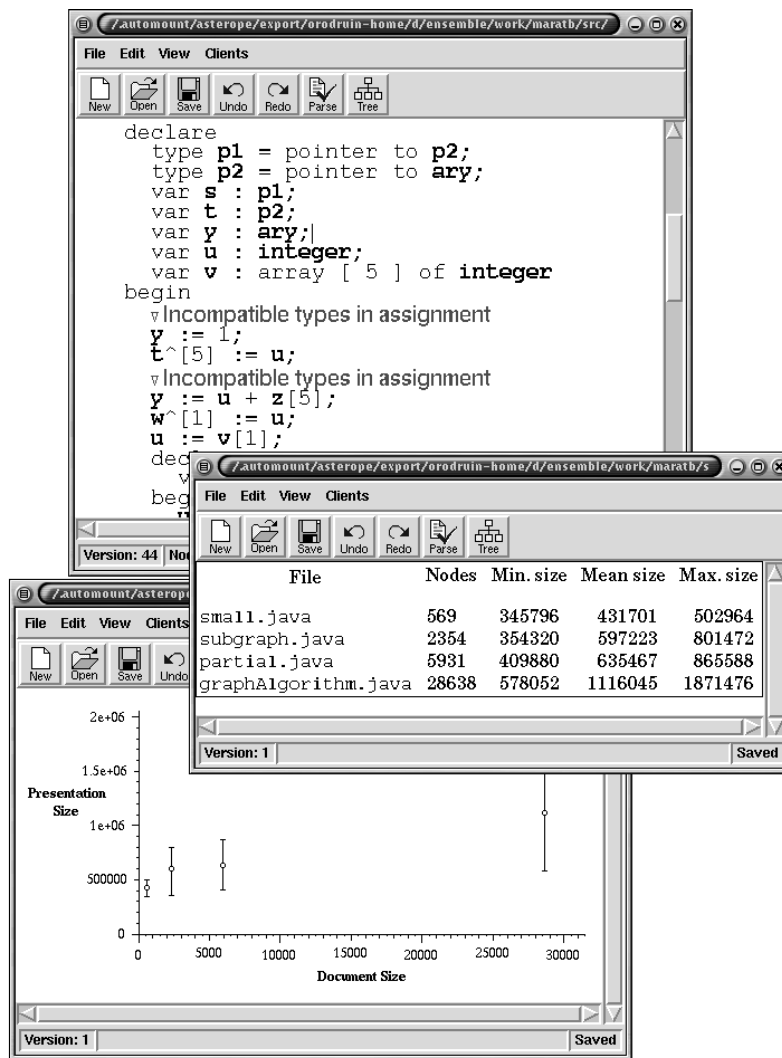


Figure 2: An Ensemble session. Two documents have been loaded: a program and a dataset. The program has been analyzed and shows some semantic errors. The dataset is displayed as a graph and as a table.

Many other language-sensitive applications such as editors, design assistants, and debuggers are also possible.

In summary, the tenets of the Harmonia framework are as follows:

- **The structure is essential.** The common theme for all Harmonia applications is that the language-induced structure of the source code is paramount. The syntactic and semantic information of the sort available in compilers can be used to provide a wide range of services to the user. At the same time, as argued in Section 2.1, this structure must permit unrestricted modification, not requiring that the document be in conformance with the language syntactic or semantic specification at any particular point in time.

- **The support is interactive and incremental.** The Harmonia framework is aimed at building *interactive* tools. As such, the structural source code model may require continuous update, potentially with every single modification. Additionally, due to the unrestricted nature of these modifications, the source code may be incomplete or ill-formed, resulting in failure to produce any sensible structural representation. The analysis services provided by the framework must, therefore, be incremental, both to achieve acceptable performance, and to avoid disturbing well-formed regions outside of any unsound

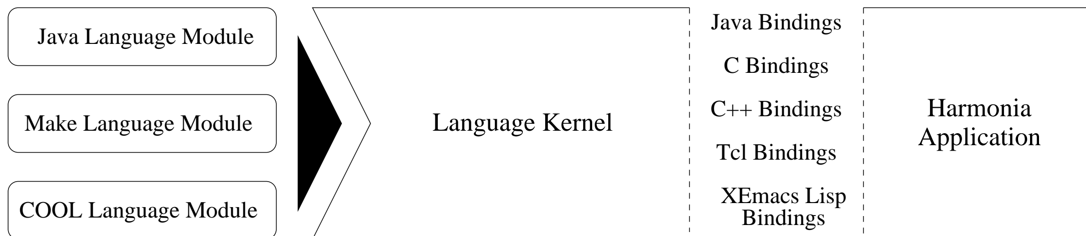


Figure 3: Component-level view of the Harmonia architecture. Language modules extend the analysis kernel with language-specific information. An application, implemented in one of the supported programming languages, utilizes the services of the analysis kernel through bindings appropriate for that programming language.

modification, by isolating the problematic change. An additional benefit of the incremental analysis services is the ability to preserve non-recoverable information associated with program structure in regions not affected by a modification.

- **There are many languages.** Our experience with Pan and Ensemble suggests that simultaneously supporting multiple languages is an important requirement for many tools. Moreover, the framework should be flexible enough to support *embedded languages*, that is, fragments of source code in a language distinct from the main language of a program. If appropriate (and if implemented), the analysis services should span language boundaries for artifacts comprised of source code in several languages.
- **The system supports rapid prototyping.** The ultimate goal of the Harmonia framework is to enable research on language-sensitive tools of various complexity. Some may be small tools, others may be larger, yet both must be easy to build and maintain. Because the implementation language of the framework (C++) may not provide the most appropriate infrastructure for building such tools, the framework should provide bindings for other programming languages which might be more appropriate for implementing a particular tool.
- **Do not reinvent the wheel.** A very conscious design objective in the Harmonia framework is to *not* re-implement every conceivable kind of program analysis or service within the framework itself. Consequently, in addition to the internal program analyses the framework must provide interfaces to incorporate services provided by other tools.

Many of these goals result from experiences with Pan and Ensemble. In fact, the current implementation of the Harmonia framework was originally derived by extracting language-analysis components of the Ensemble system and embedding them in a new and extensible architecture. The remainder of this report discusses this architecture in considerable detail and describes how the above requirements are realized in the Harmonia design.

4 The Architecture of the Harmonia Framework

Figure 3 presents a high-level view of the Harmonia architecture. The three major constituents of the Harmonia framework are the language kernel, the language modules, and the application interface layer for several programming languages.

The *language kernel* (LK) provides the abstractions for modeling programming artifacts and the language-independent infrastructure for incremental program analyses. Language-specific analysis details are encapsulated by *language modules* that can be demand-loaded into a running Harmonia application. The notion of separating the language-independent analysis kernel from the dynamically loadable language-specific components dates back to Pan and Ensemble. While not unique to these systems, it is an important step toward meeting the goal of simultaneously supporting multiple programming languages. Thus Harmonia

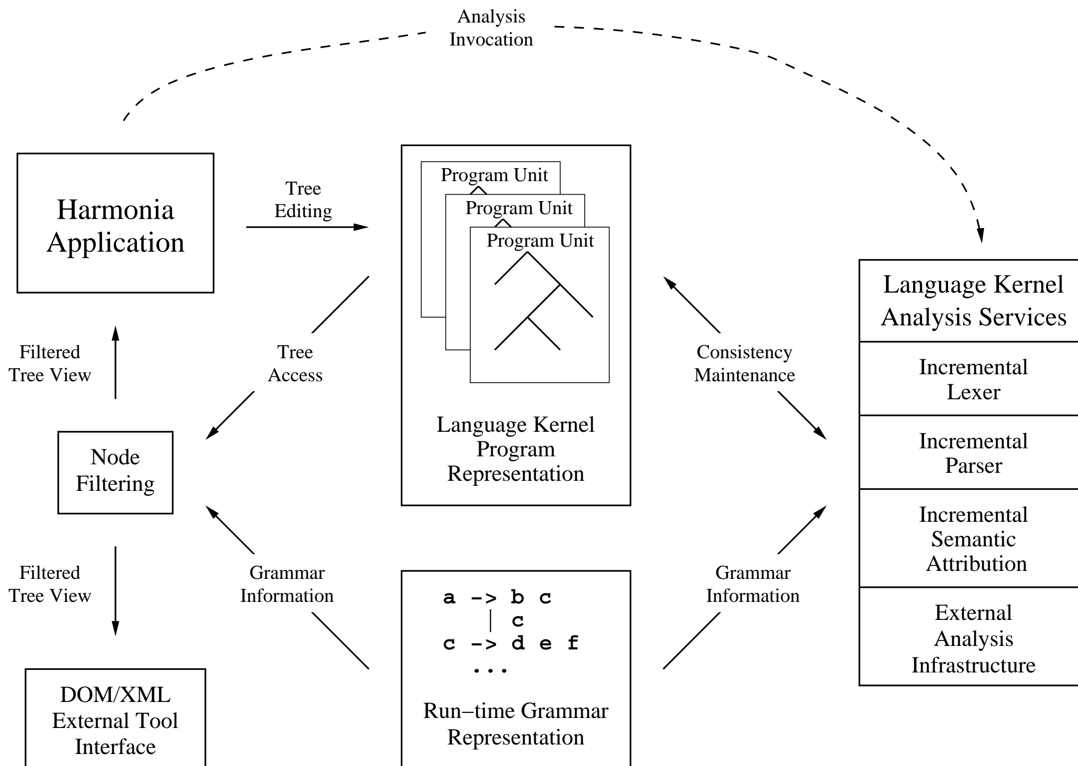


Figure 4: The architecture of the HARMONIA language kernel.

is not a "template-based" monolingual framework as are the environments produced by the Synthesizer Generator [41] and other similar systems.

Both PAN and ENSEMBLE designers recognized that extensibility is an important feature of any language-based system. PAN is extensible through Lisp, its implementation language, and ENSEMBLE through ExL, a Scheme-like language in which most of the ENSEMBLE user interface is built [14]. As a *framework*, HARMONIA takes extensibility one step further: an entire application utilizing the language kernel services can be implemented in the language of the programmer's choice. This is especially attractive in light of the recent popularity of relatively high-level scripting languages such as Tcl [37], Perl [60], Python [51], etc. Many such languages provide convenient graphical user interface libraries – an important factor for builders of interactive tools. Consequently, in addition to the API in its implementation language (C++), the HARMONIA framework includes bindings for other popular programming languages. At the time of this writing, bindings for C, Tcl, Java, and XEmacs Lisp are available.

The following sections describe each of the three major constituents in detail. The internal architecture of the language kernel is presented in Section 5. The mechanism for building languages modules is discussed in Section 6. Finally, Section 7 describes the design of the bindings interface to a number of programming languages.

5 The HARMONIA Language Kernel

Figure 4 presents the architecture of the HARMONIA language kernel. The central component of the language kernel is the program representation infrastructure, which consists of abstractions for modeling program source code and the programming language grammar. Program source code is represented as syntax trees that are organized into *program units* – language specific abstractions for project management and dependency tracking.

The run-time grammar representation encapsulates the programming language grammar used in building a language module. The purpose of this representation is two-fold. First, it supplies the incremental parser with the information needed to build syntax trees. Second, it facilitates *filtering* of syntax tree nodes based on their grammatical properties. For example, an application trafficking in the text representation of program source code may not need to "see" any non-terminal nodes, limiting its internal representation to tokens. This filtering is non-destructive and is implemented through a special navigational abstraction called a *tree-view*. Filtering through tree-views may also be performed on non-grammatical properties of syntax tree nodes.

Harmonia applications manipulate syntax trees by directly accessing the data structure. However, experience suggests that users are more comfortable manipulating programs as structure-less text. Thus, it is the job of the application to provide a reasonable user-model on top of the underlying representation. To facilitate this, the editing model is completely unrestricted. Free-form editing of syntax trees is supported, so that the application (and, by extension, the user) need not be concerned with making only syntactically or semantically sound modifications. Semantic and syntactic consistency can be restored following an edit by invoking the incremental analysis services supplied by the language kernel.

The two core analyses implemented by the language kernel are incremental lexing and parsing. Both analyses utilize language-specific information derived from a declarative specification and supplied as part of a language module. In addition to lexing and parsing, the language kernel includes an infrastructure for building incremental semantic analyzers. At present, the semantic analyzers rely on hand-coded tree-traversing computations (also specified as part of a language module); future versions of the Harmonia framework will provide a facility for generating semantic analyzer from a declarative specification as well.

To address the goal of integration with external tools and analyzers, the language kernel provides access to the syntax tree data structure using the Document Object Model (DOM) API [53]. DOM is an industry-standard interface for accessing structured documents (of which syntax trees are an instance) and as such is well-suited for integration with third-party components. An additional benefit of using DOM is that the tree data structure can be easily exported and stored in an XML encoding [62] (another widely accepted standard).

The basic constituents of the language kernel have been derived from the Ensemble system and repackaged under a new architecture. Section 5.1 summarizes the syntax tree model, the run-time grammar representation, the editing and analysis model, and the incremental lexing and parsing implementation borrowed from Ensemble, and also discusses the enhancements to these components in the Harmonia language kernel. The sections that follow describe the extensions to the Ensemble language kernel architecture to address Harmonia design goals presented in Section 3.

5.1 The Ensemble Language Kernel

This section summarizes the parts of the Harmonia language kernel derived directly from the Ensemble system. While, in most cases, the implementation of these components has been significantly updated, only those enhancements that are relevant to the overall Harmonia architecture goals are presented here as this discussion is largely aimed at providing the necessary background for the subsequent sections of this report. Further implementation and design details can be found in Tim Wagner's dissertation [55] and in the Harmonia Architecture Manual [10].

5.1.1 From Grammars to Trees

The program source code is represented in Harmonia as syntax trees, which are constructed by the parser from the textual representation of a program. As the source code is manipulated in its tree form by a Harmonia application, the parser is also responsible for incrementally updating the tree data structure in accordance with the programming language grammar. Figure 5 shows a grammar and two syntax trees described by that grammar. Every node in a syntax tree corresponds to a unique production in the language grammar (designated by the subscript on the left-hand-side symbol).

A language grammar can be said to define a *type system*: every non-terminal on the left-hand-side designates a type, called a *phylum*, and every production for that non-terminal designates a sub-type of that type, called an *operator*. In addition, every terminal in the grammar also defines an operator type. The

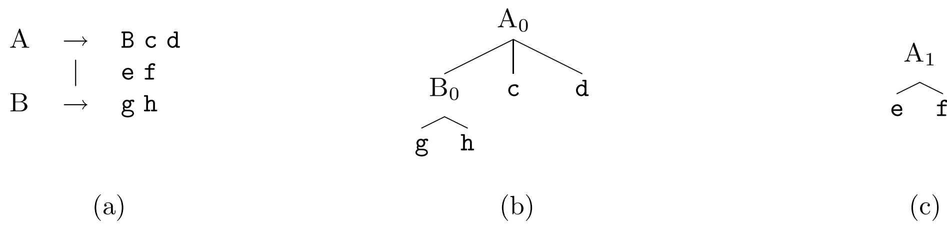


Figure 5: (a) A small grammar and (b), (c) two sample parse trees.

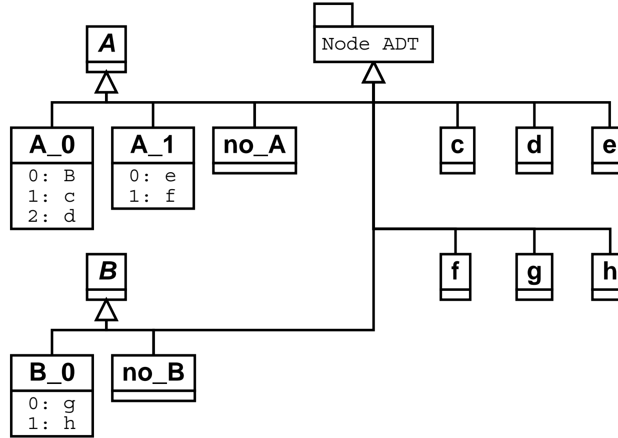


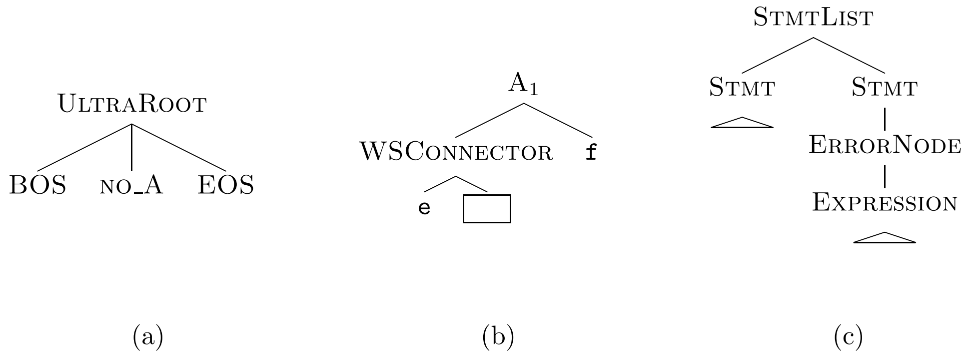
Figure 6: A UML diagram of the automatically-generated node classes for the grammar of Figure 5a.

phyla are abstract and cannot appear in a syntax tree; the nodes in a syntax tree are instances of operators. The rules by which the nodes are linked to form a syntax tree are also dictated by the grammar. The right-hand-side of every production gives the type of a node that may be linked as an appropriate child of the node (operator instance) for that production. A non-terminal symbol on the right-hand-side indicates that an instance of any operator for that non-terminal's phylum may appear in its place; the terminals represent their operator types.

In Harmonia (as in Ensemble), we find it convenient to map the type system defined by the language grammar to that of C++, the Harmonia implementation language. Among other advantages, this allows specification of analyses on syntax trees in an object-oriented fashion by providing methods for every distinct node type. (An example of such an approach is the semantic analysis framework described in Section 5.4.) This mapping is achieved by generating a hierarchy of C++ classes to represent the type relationships induced by the language grammar. Figure 6 presents a class diagram for the grammar of Figure 5 in the Unified Modeling Language (UML) notation [43]. Classes `A` and `B` represent the phyla. Classes `A_0`, `A_1`, and `B_0` are the operator classes for the corresponding non-terminal productions. Every such class has an appropriately typed slot to hold a subtree for every right-hand-side symbol. In addition, so-called *completing operators* (or completers: in our example classes `no_A` and `no_B`) are generated for every phylum to represent the absence of a subtree for that phylum. The completers have no right-hand-side slots and are used for constructing partial syntax trees that are still well-formed with respect to the type system induced by the grammar. Figure 7a demonstrates the use of a completer node to represent an empty program.

In addition to subclassing phyla, all operators extend one of the classes provided by the Node ADT package. The Node ADT package is a set of language independent classes supplied by the language kernel that implement the tree node abstract data type. These classes provide the basic tree operations as well as a number of extensions to the grammar-based syntax tree model. While the detailed discussion of these classes is beyond the scope of this report, they are summarized here for completeness. Further information can be found in the Harmonia Architecture Manual [10].

- **Basic Node Classes.** Basic node classes provide the necessary functionality for representing tree



(a) (b) (c)

Figure 7: (a) A rudimentary Harmonia syntax tree with the completing operator for the starting symbol. (b) The tree of Figure 5c with whitespace following the terminal `e`. (c) A syntax tree with an error node used to encapsulate a syntactic inconsistency.

nodes of arbitrary kind. These classes include low-level API for manipulating the syntax tree data structure.

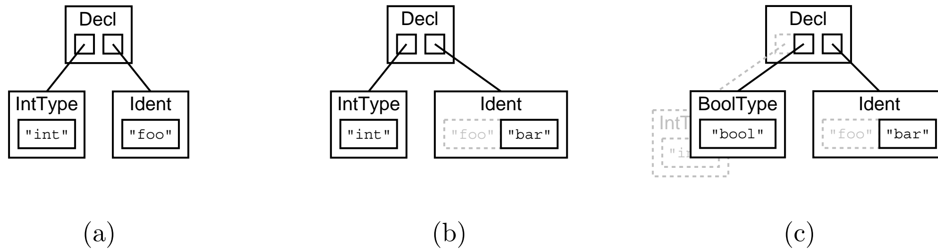
- **Sentinel Node Classes.** The three sentinel nodes *UltraRoot*, *BOS* (for Beginning-Of-Stream) and *EOS* (for End-Of-Stream) form the basic structure of every syntax tree. The `UltraRoot` is always at the root of the tree, with `BOS` as its first child, and `EOS` as its last child. The `UltraRoot`'s middle child is the real root of the syntax tree corresponding to the start production of the language grammar. A rudimentary Harmonia syntax tree with a completing start symbol production is depicted in Figure 7a.

- **Whitespace Node Classes.** Whitespace plays an important role in textual program source code by influencing visual formatting and providing lexical boundaries between tokens. Yet, whitespace is typically distinguished from the rest of the source code by being non-structural and therefore not representable in the language grammar. Because syntax trees are the sole representation of program source in Harmonia and because the user should be able to view the source code as an editable character stream, we must preserve whitespace in our syntax tree model. The whitespace (and other non-syntactic material such as comments) is incorporated into the syntax tree in structural manner through special *connector* nodes that can be attached to any place where a terminal is allowed to appear. Figure 7b demonstrates how the example in Figure 5 is augmented to add whitespace.

- **Error Node Classes.** While the whitespace and the comments represent entirely non-syntactic material in the syntax tree, frequently it is necessary to represent syntactically sound portions of the tree in a context that is syntactically illegal according to the language grammar. Such a need may arise following a structural modification to the syntax tree. For example, in Java moving an expression subtree into the context where a statement is expected may require creating a new internal "expression-statement" node. Since this consistency restoration may only happen upon running the parser, special provisions must be made to represent the intermittent state of the syntax tree. Error node classes serve precisely that purpose and allow to "escape" the typing constraints induced by the language grammar. Figure 7c illustrates this situation.

5.1.2 Run-time Grammar Representation

Traditional program translation tools make little use of a language grammar specification beyond driving the parser. This largely stems from the fact that such tools are typically oriented toward a single language, which affords "hard-wiring" many aspects of the language grammar into the tool. Such an approach does not work for a multilingual system such as Harmonia. Firstly, the grammar is needed for instantiating syntax tree nodes: since the type of each node is dictated by a programming language grammar, the language-independent kernel of the framework requires access to this grammar at run-time. The second use of the



(a) (b) (c)

Figure 8: Editing of a self-versioned syntax tree. (a) prior to the modifications, (b) after a textual edit (changing identifier name to “bar”), and (c) following a structural edit. Shaded portions represent the older versions of the tree.

grammar representation is to encapsulate information about the language grammar that may be useful to the clients of the framework.

In addition to the parse tables for driving the parser and the generated node classes for building syntax trees, Harmonia grammar processing tools generate a set of run-time data structures which model the language grammar. Every grammar phylum induces an information structure consisting of fields identifying the phylum as well as all operators that are instances of that phylum. Every grammar operator is modeled as a data structure that identifies the operator, the phyla for the right-hand-side symbols, as well as the phylum to which it belongs.

Because the Harmonia language kernel is language-independent, it cannot be aware of the names of the generated node classes. Since new classes in C++ cannot be introduced into an executing program, the instantiation of nodes into a syntax tree (for example, by the parser) is based on *prototype objects*. Creating a new node from a prototype is done by “cloning” that node, rather than instantiating a known class. A full set of prototype nodes is manifested at run-time through a collection of prototype node objects indexed by the rule number. Thus, instantiating a node involves merely “knowing” the rule to which it corresponds; this information is given by the parse table.

5.1.3 Syntax Trees as Persistent Self-Versioned Documents

Syntax trees in Harmonia serve as a *persistent* representation of the source code. This means that not only the current state of a syntax tree is maintained by the system, but also all older revisions of that syntax tree can be accessed. The primary use of this facility is to provide incrementality to the analysis algorithms by making results of older analyses readily available (see Section 5.1.4). In addition, persistent syntax trees simplify building intelligent undo services in Harmonia tools as well as provide the building blocks for sophisticated configuration management.

The traditional way to represent revision information is *temporal*: as each modification is being applied to a data structure such as an editor buffer, an entry is made in an “undo” log. The log establishes the order of operations on that data structure; any older version of the data structure may be re-created by “undoing” modifications in reverse order. By contrast, the Harmonia framework implements the means for maintaining change information *structurally*. That is, every object encapsulates its own “undo” log and a global temporal index is used for correlating any internal revision of that object with those of other objects. Harmonia’s implementation of this persistent data structure is a direct descendant of that in Ensemble and is also due to Tim Wagner [56, 55]; the theoretical underpinnings for that design are derived from the “fat node” model of Driscoll et al. [18].

Building Persistent Syntax Trees

The Harmonia versioning system is built from *primitive versioned data types*. Each versioned data type is manifested through a low-level versioned object, which is essentially a log of values stored in that object over the course of its lifetime. The primitive versioned data types presently provided in the Harmonia framework are booleans, integers, reals, and pointers. Composite versioned data types include pointer arrays (whose length is fixed at construction time), pointer sets (of variable length), and character strings.

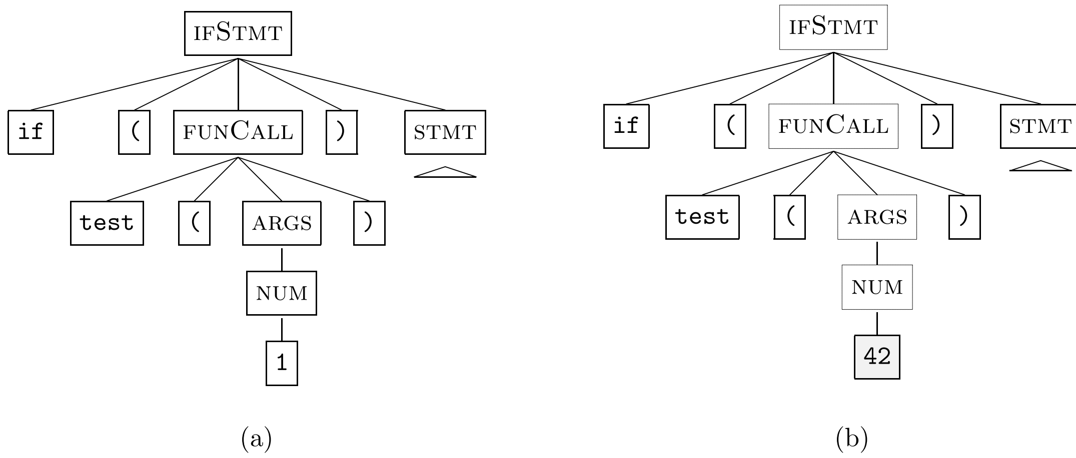


Figure 9: Change discovery in Harmonia syntax trees: (a) initial state of the syntax tree and (b) the syntax tree after a textual edit to the number node (from "1" to "42"). The shaded terminal node is the one having local changes and the dashed non-terminal nodes are those with nested changes.

A node in a syntax tree is an aggregate data structure each of whose components are primitive versioned objects. For example, a token node provides a versioned string slot for storing character data, a non-terminal node provides a versioned pointer array for storing children links. Figure 8 shows how a versioned syntax tree data structure can be constructed from such node objects: whenever a data field is updated, its new value is appended to the log of older values; node identity changes are reflected through updates to the corresponding child link in its parent.

A local revision of every primitive versioned object is marked with a global timestamp that allows correlation of the internal revisions of multiple data objects. These time stamps, called *global version identifiers* (GVIDs), are organized into a *global version tree* that represents the relationship between different versions. Much as in traditional version control systems, it is possible to "go back" to an earlier version of the tree and fork off a new version, hence the tree-like representation of versions. The global version tree is manipulated in a transaction-like fashion. A new version of the global version tree needs to be opened before any modifications to a versioned data structure are allowed. After all edits are performed, the opened version is "committed" and can no longer be modified.

Change Discovery Framework

Because every versioned object tracks changes to its value, the use of versioned objects enables a flexible framework for discovering changes to the entire versioned data structure. The change reporting mechanism is rooted in the concept of *local changes*. A versioned data structure (for example, a syntax tree node) is said to be *locally changed* in the current version if any of its versioned data fields have been changed since this version was opened. The local change information can be computed from the edit history logs and so a versioned data structure can also be locally changed between any two committed versions v_1 and v_2 (the version range excludes changes in v_1 and includes those in v_2). Depending on the semantics of the versioned data structure, it may be necessary to distinguish different kinds of local changes. For example, in Harmonia tree nodes we often differentiate between changes to the child links (child changes), changes to the parent link (parent changes), and changes to the textual content (text changes).

Harmonia analysis and transformation tools discover modifications to the syntax tree by traversing the tree and looking for local changes. To enable efficient change discovery, we restrict these traversals to the regions that are known to be modified. This is facilitated through use of *nested change* information. Nested changes in a node indicate that some node in its subtree has local changes. Nested changes are recorded by using a versioned boolean field at each node (changes to this field are excluded from the local changes to the node). The nested change information represents a *synthesized* tree attribute that can be computed through a bottom-up traversal of the node's subtree: the nested change bit is set if any of the immediate children of a node have either nested or local changes. Figure 9 demonstrates change propagation in a syntax tree

following a textual edit. In an effort to minimize node size, we do not categorize nested change information as we do with local changes. Thus, a client only concerned with a particular type of a local change (for example, a text change) may be misled into performing a larger traversal than necessary. We consider this to be an acceptable trade-off.

5.1.4 Editing and Analysis Model

One of the lessons of the Pan project is that the editing model of an effective language-based programming tool must be completely unrestricted [49]. In other words, a user must be able to freely edit program source code without concern for transient ill-formedness introduced during such an edit. To facilitate this, the editing model employed by Harmonia is as flexible as that of Ensemble: any number of textual or structural edits are allowed between any consistency-restoring analyses.

A textual edit is introduced by replacing the textual content in a terminal syntax tree node. The replacement text need not conform to the lexical description of the corresponding token type; when requested by the parser, the incremental lexical analyzer will restore the consistency. Similarly, free-form structural modification is permitted, with the understanding that structural soundness will be restored by the incremental parser. The change discovery framework described above is used to guide the incremental analyses following a change. Figure 10 illustrates a sample editing sequence followed by consistency-restoring analyses.

The frequency of the analyses is defined by Harmonia applications, rather than by the framework: potentially an analysis may be invoked upon every modification to program text. While it has been argued that such a fine-grain analysis is unnecessary because the (typically invalid) results may be distracting if presented to the user [52], for certain applications, the partial analysis results may be necessary for processing user input. For example, Begel [7] describes a speech-enabled program editor whose speech recognition engine requires the knowledge of the syntactic context surrounding the cursor location in order to property process and disambiguate speech input. Another example is the CodeProcessor editor [50] that uses lexical context to provide visual formatting as well as the appropriate user-level behavior in presence of embedded documents (comments and strings, in particular).

The analysis model employed by Harmonia is based on the history mechanism described in Section 5.1.3. A typical language analysis utilizes three versions of the syntax tree: *reference*, *previous*, and *current*. A reference version is the last version produced by the analysis in which consistency was successfully restored. The previous version is the one last edited by the user or another analysis. Finally, the current version is the one being constructed as the result of the analysis. If successfully constructed, the current version will serve as the reference version for the next invocation of the analysis. This organization dictates a general form of any analysis which is illustrated in Figure 11. The incrementality is achieved through the change discovery framework, which enables the analysis to locate all modifications in the syntax tree from the reference version to the current version.

The language kernel provides incremental lexical and syntactic analyses. In practice, only the syntactic analysis may be invoked directly; the incremental lexer is tightly coupled with the incremental parser and is invoked on an "as-needed" basis. The object-based language model permits specification of any additional analysis, such as static semantics, by the language module author. A reference version is automatically maintained by the language kernel per analysis: what constitutes a reference version for one analysis does not necessarily serve as such for another analysis. Additionally, the language kernel supplies a mechanism for analysis writers to register the dependency of their analysis on the result of another one. This imposes an ordering of the different analyses, if necessary.

5.1.5 Incremental Lexer

The incremental lexical analyzer in Harmonia is implemented as a wrapper around a traditional batch lexical analyzer. The analyzers currently used by Harmonia language modules are generated by Flex [38], but other analyzers (including hand-coded scanners) are possible. The only requirement of the batch lexers is that they conform to a simple interface for setting lexical states and accepting input characters. For efficiency reasons, the lexical analyzer cannot be invoked directly by the client application and is called on demand by the parser.

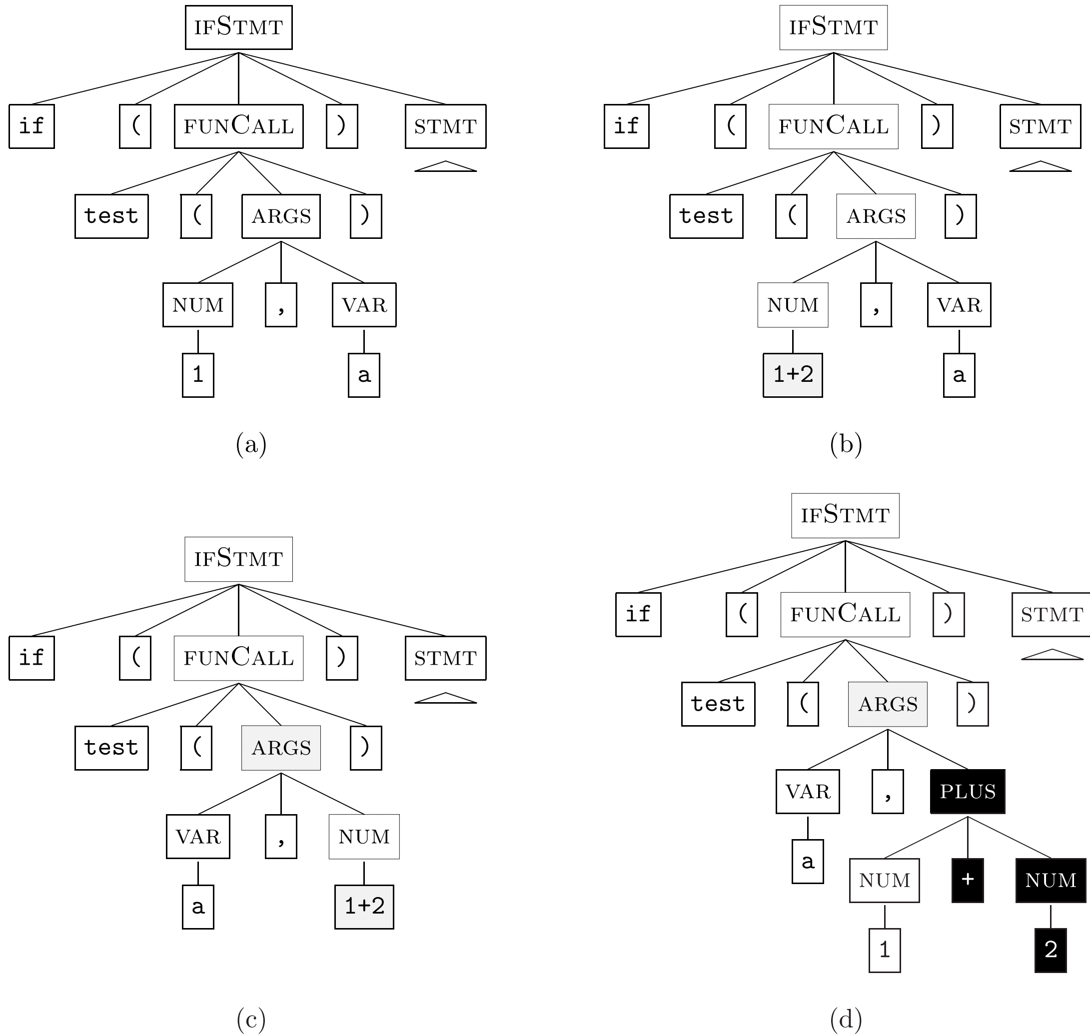


Figure 10: Sample editing sequence, starting from a consistent state (a), followed by (b), a text edit of the number node (from "1" to "1+2"), and then by (c), a structural edit of the argument node (swapping the two argument subtrees). (d) A new consistent state after incremental analysis. As before, shaded nodes are those with local changes, black – indicates new nodes, and dashed nodes represent the nested change path. (Adapted from Tim Wagner [55].)

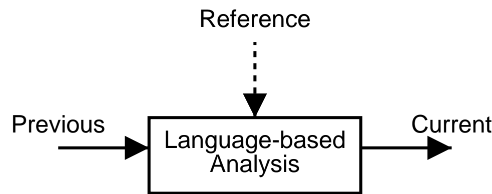


Figure 11: A general form of a Harmonia language analysis. (Adapted from Tim Wagner [55].)

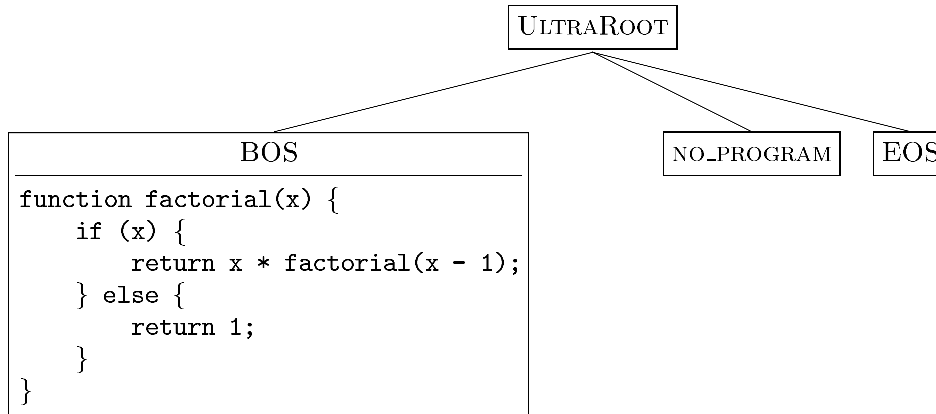


Figure 12: A "fake" syntax tree for simulating batch syntactic analysis in an incremental setting. The entire text of the program is inserted into the `BOS` node and a completer node (no_program) is used to represent the absence of the program structure. The document is subsequently re-parsed, resulting in a "real" syntax tree structure.

The lexical analysis model is very flexible and virtually any existing lexical description for Flex can be easily adapted for use with Harmonia's incremental lexer. Lexical state conditions are allowed, but must be stored as simple integers; stacks of state conditions are not supported. Similarly, no user-defined state can be preserved in the batch analyzer across token boundaries. This precludes use of global flags and counters, but does not appear to be a significant restriction. In practice, it is possible to extend the analysis model to allow arbitrary user-defined state (including stacks of lexical state conditions), but this extension would incur additional space cost as all such state would need to be stored in versioned fields on token nodes.

5.1.6 Incremental Parser

The Harmonia incremental parser is invoked by the client application whenever it needs to restore the structural consistency of the syntax tree. The parser attempts to repair the tree structure incrementally, based on the previously analyzed version (if one is available). An initial (batch) parse may be invoked by constructing a very simple "fake" tree structure as illustrated in Figure 12. The incremental lexer is invoked by the parser whenever it decides that a section of the syntax tree requires re-tokenization.

The Harmonia language kernel supplies two incremental parser implementations, both conforming to a common incremental parsing interface. One is a traditional deterministic bottom-up LR parser, the other is a much more powerful Generalized LR (GLR) parser. Both parsers implement generic table-driven parsing algorithms. The parse tables for each language are specified through Harmonia language modules. The choice of an incremental parser lies with the author of the language module.

The deterministic incremental LR parser implements a bottom-up shift-reduce parsing algorithm. Its tables are provided by a modified variant of Bison described in Section 6.3. This parser is simpler and somewhat more efficient than the GLR parser. However, it suffers the same drawbacks as any other parser of a restrictive grammatical class: the language grammar must be unambiguous and conform to the limitations of the particular table-generation algorithm. For Harmonia, this requires the grammar to be $LALR(1)$,

which, in many cases, is quite restrictive and requires significant "grammar-hacking." More importantly, the resulting "hacked" grammar becomes far removed from an abstract grammar representation making it less practical for describing syntax trees. For this reason, the LR parser in Harmonia is little utilized and is used mostly for testing purposes.

The second incremental parser implementation in Harmonia is based on the Generalized LR (GLR) parsing algorithm [45, 46, 47]. The descriptive power of GLR eliminates the need for most "grammar-hacking" and allows a syntax specification that naturally corresponds to abstract syntax, without the need for complex mappings as was done in Pan [3]. Additionally, GLR permits a syntactically ambiguous grammar specification, which is necessary because the syntax of many languages falls outside the $LR(k)$ formalism. In such a case, the validation of a program's phrase structure is better left to the later analysis stages such as static semantics.[2]

GLR parser works by processing the input *non-deterministically*: the parser functions virtually identically to a bottom-up LR parser until a shift-reduce conflict is encountered. At that point, the parser "forks" to explore both possibilities, thus freeing the grammar writer from having to choose one *a priori*. Indeed, in many cases such choice is not possible! The grammar may not be $LR(k)$ for a bounded value of k or just outright ambiguous.[3] Parse tables for the GLR parser come from a variant of Bison that had been modified to output a table of conflicts in addition to the usual $LALR(1)$ parse tables.[4]

Both the LR and the GLR parsers construct the parse tree by executing tree-building actions upon each reduction. (Thus, no actions are allowed in the Harmonia grammar specification). Since one of the distinguishing features of the GLR parsing algorithm is its ability to cope with ambiguous grammars, it includes provisions for dealing with inputs that have multiple legal parses. In case of such an input, the resulting tree is not a parse tree, but a *parse forest*, representing all possible syntactic interpretations of a particular input string. To avoid a potentially exponential increase in the size of the syntax tree, the GLR parser packs this parse forest into a *parse dag* by ensuring that all the parse trees resulting from such an input share the isomorphic portions of their structure. Packed representation is achieved through several extensions to the Harmonia syntax tree model that allow the GLR parser to encode alternative parses within a parse tree. This representation is an enhanced variant of that used by Ensemble and is described in more detail in Section 5.2.3. The actual mechanics of constructing the shared representation are beyond the scope of this report and are discussed by Earl Wagner et al. in a separate document [54].

5.1.7 Coping with Errors

Van De Vanter [48, 49] argues that any successful interactive tool must continue to function and provide useful services even in the presence of ill-formedness, incompleteness, and inconsistency in the program source code. These "three I's", as Van De Vanter calls them, are manifested in the Harmonia framework as lexical, syntactic, or semantic errors. In order to provide the maximum possible service to an application, the Harmonia framework must gracefully handle these errors during the analysis. For dealing with these errors, the Harmonia framework relies on the same mechanisms as the Ensemble system.

The handling of syntactic errors is achieved through a technique known as *error isolation*. Error isolation relies on a simple observation that a malformed region may be isolated from the rest of program source code by considering the sequence of edits from the previous valid version that led to that inconsistency.[5]

The process of isolating a parse error is illustrated in Figure 13. Informally, the error isolation works by looking for the smallest subtree in the last edited version of the document that covers the region in the input

[2] Batch programming tools cope with syntactic ambiguity by introducing *ad hoc* techniques, for instance performing partial type and scope analysis in tandem with parsing. Such an approach does not work in an incremental setting.

[3] $LR(k)$ grammars are those in which any parsing conflict may be resolved by at most k symbols of lookahead. However, for some grammars, the value of k cannot be statically determined at table construction time. Ambiguous context-free grammars are those for which more than one legal parse tree can be constructed for the same input string.

[4] Since the $LALR(1)$ class of grammars is smaller than the $LR(1)$ class, the parse table may, potentially, contain more conflicts than would otherwise be necessary. This is, of course, offset by having a more compact parse table and the ability of the GLR parser to resolve those conflicts dynamically.

[5] Such a view, of course, implies that no useful error recovery may be performed if historical information is not available, for example, on a first parse. This is a known limitation of the Harmonia error recovery techniques and is currently being remedied.

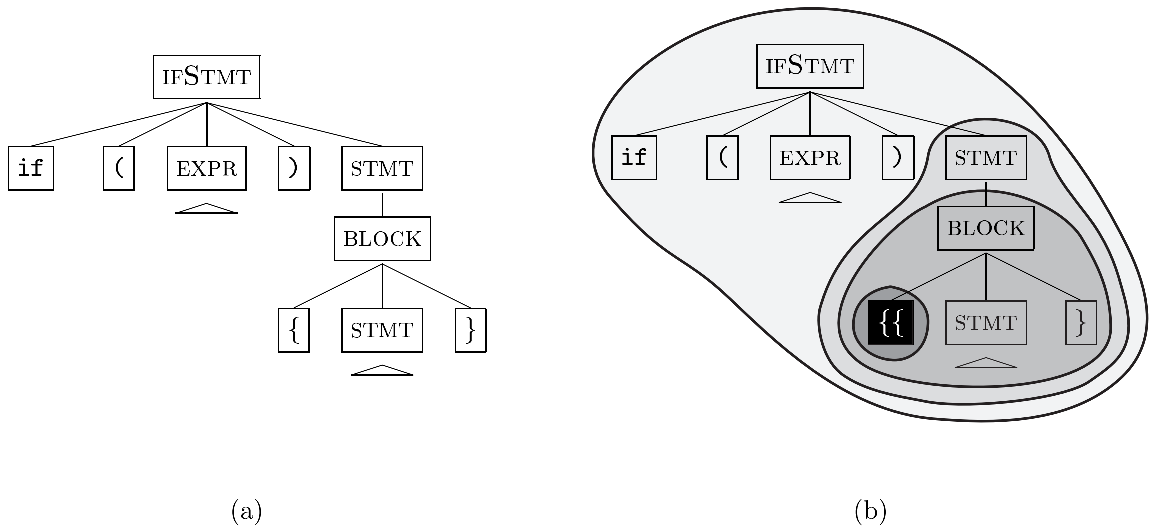


Figure 13: Isolating a parse error: (a) the program prior to modification and (b) trying a successive sequence of isolation regions following an erroneous edit.

containing the erroneous change. Once such a subtree is located, it is reverted to the pre-analysis state and skipped by the parser, thereby allowing parsing to continue in an incremental fashion beyond the malformed region.[6] The full algorithm for history based error isolation is given in Tim Wagner's dissertation [55]. More recently, Earl Wagner [54] extended his techniques to apply them in the presence of ambiguity in the parse tree.

The handling of lexical errors in Harmonia is performed in a traditional manner by creating a special token denoting unrecognized characters. When the parser is subsequently unable to incorporate this special token, the normal syntactic error recovery process results in the isolation of the lexical error.

5.2 Extensions to the Syntax Tree Model

This section discusses various extensions to the Ensemble syntax tree model that were introduced in Harmonia. The first two extensions, node properties and node attributes, were added to facilitate Harmonia functioning as an application framework. Node properties make it easy to attach application-specific data to the syntax tree nodes. Node attributes provide a uniform interface for getting and setting the values of various pre-defined attributes on syntax tree nodes. The third extension is provided for the benefit of the Generalized LR parser and is intended to allow multiple representations in ambiguous parse forests. This extension is a re-implementation of a similar facility present in Ensemble.

5.2.1 Node Properties

Many Harmonia applications, as well as the language kernel itself, frequently need to annotate the syntax tree with information in a sparse manner. Such annotations are manifested as values that need to be stored only on a handful of tree nodes, making allocating a slot for such values on all nodes of a given type somewhat wasteful. An example of sparse data value is the error display information set by the incremental parser: only the tree nodes containing an error should allocate storage for such data. However, whenever a new field is defined for a particular node class, all instances of that class have to bear the resulting increase in size. (This is particularly severe considering the number of nodes in even a small-sized program: on average 15

[6]This description is very high-level approximation of the error isolation algorithm. In actuality, the isolation subtree must not only span the malformed region, but also meet a number of special conditions that hold at its boundaries. Similarly, the isolation subtree is not merely reverted to the pre-analysis state, but an attempt is made to analyze regions within that subtree unaffected by the erroneous edit.

nodes per one line of source code.) Moreover, it is often not known that a node needs to reserve space for a piece of data when the class for that node is being defined; thus, it is not possible to declare a field for storing that data.

To remedy this situation, HARMONIA provides a special API for associating sparse node annotations with any syntax tree node. These annotations, called *node properties*, are represented as name-value pairs that can be stored within tree nodes. Two HARMONIA Language Kernel supports two kinds of node properties: *versioned* and *transient*. Versioned properties, as the name implies, use versioned objects for storing their values. Transient properties are not versioned and their values are available only in the current version of the tree. All transient properties are removed as soon as the current version is changed, either explicitly or by opening a new version for editing.

5.2.2 Node Attributes

There is a wealth of information available to clients at every syntax tree node. Some of it, such as a node's textual content, is stored directly in the node. Some of it, such as the next node in the DFS ordering of the tree, can be computed on demand. Not all of this information is available through a consistent interface: for some nodes the data may be stored in the node object itself, for others – using the node property interfaces. To provide uniform access to this information HARMONIA implements a system of *node attributes*.

Node attributes provide a form of reflection on syntax tree nodes. 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 at the language module compilation time. Accessing node attributes is functional: rather than reserve storage on a node for a piece of data, the attribute specification merely describes how to compute that data. In actuality, the data may be stored in a field or be computed on demand through a function call. Some attributes may be read-only (for example, the ones computed on demand); others – settable by the HARMONIA applications.

There are two independent aspects of node attributes that affect how their values can be accessed. Firstly, an attribute may refer to a versioned or an unversioned value. A versioned value corresponds to a particular version of the syntax tree. Such a value may be stored in a versioned object or computed from version-specific data. By contrast, an unversioned attribute's value is not stored using the version system. Such a value may only be valid within the current version of the syntax tree (for example, a transient node property), or be version-independent altogether (for example, node's arity).

Another aspect of the node attribute system reflects how an attribute's value is updated. This is important because some attributes may represent values computed by an analysis over the syntax tree and are thus valid only in the version of the syntax tree derived by that analysis. An attribute's value is said to be *synchronized* by an analysis when that value is computed or updated by that analysis. Such attributes are called *synchronizable*. Synchronizable attributes depend on a particular analysis and their values are only valid in the version of the syntax tree produced by that analysis (for example, syntax error information is computed by the parser). Non-synchronizable attribute values are valid in any version of the syntax tree (for example, the production name of a syntax tree node).

The combination of these two independent aspects yields four possible attribute types: *versioned-synchronizable*, *versioned non-synchronizable*, *unversioned-synchronizable*, and *unversioned non-synchronizable*.

The set of types that an attribute value may have is limited to integers, booleans, floating point numerals, characters, node references, and strings. An array of each of these types is also allowed, but the type system for attributes is not derived. (In other words, there's no single generic array type whose combination with other types would lead to array of integers, array of arrays of integers, and so on.) Three more special types are provided for convenience: "undefined" – denoting the absence of a value, "unavailable" – indicating that the value of an attribute is not synchronized, and "unknown" – advising that the system has been supplied an unknown attribute name.

5.2.3 Multiple Representations for Non-deterministic Parsing

Oftentimes, the parsing grammar for a programming language is more naturally expressed in a syntactically ambiguous form. In case of such a grammar, the output of the GLR parsing algorithm is a forest of parse trees representing multiple legal parses of the input. An important observation is that for a typical programming

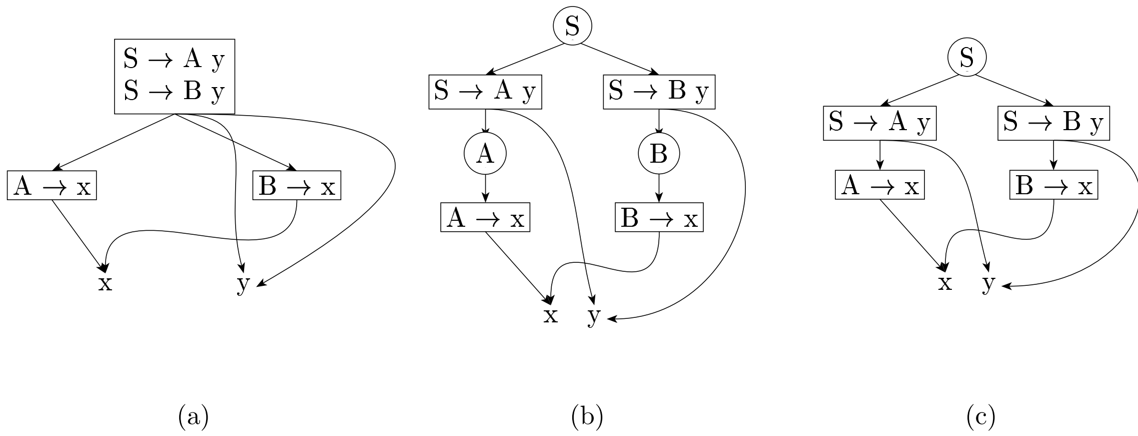


Figure 14: Syntax trees for the ambiguous grammar S → A y | B y; A → x; B → x for input xy. (a) Tomita's representation using a packed node to represent ambiguity, (b) Rekers' representation using symbol nodes, and (b) Harmonia representation with lazily instantiated symbol nodes.

language grammar the ambiguities are localized, the parsing algorithm is "almost" deterministic, and the resulting parse trees are "mostly" isomorphic.

Masaru Tomita, the author of the original GLR algorithm [45, 46, 47], suggests using a *parse dag* to compactly represent multiple parse trees by sharing the isomorphic portions of their structure. The roots of subtrees that are not isomorphic are merged into a so-called *packed node.* This representation is demonstrated in Figure 14a. Subsequent enhancements to the GLR parsing algorithm by Farshi [36] for handling *epsilon*-reductions and improvements to subtree sharing by Rekers [39] resulted in a more flexible, albeit less space efficient, encoding of the parse forest. This encoding, described by Rekers in his dissertation [39], introduces the concept of *symbol nodes*, representing the phylum of the production, *rule nodes*, denoting a particular rule for that phylum, and *term nodes*, representing tokens. In this encoding the parse dag is a bipartite graph with alternating rule and symbol nodes (Figure 14b). In an unambiguous parse, each symbol node has just one child – a rule node representing the unambiguous derivation. In the presence of ambiguity, multiple rule nodes may be children of the same symbol node. This representation allows Rekers to improve the sharing of nodes in the parse graph significantly. An unfortunate side-effect of this representation is that the presence of symbol nodes nearly doubles the size of the data structure. This is particularly severe since ambiguities are typically localized and thus most symbol nodes have only a single rule-node alternative.

The Ensemble implementation takes Rekers' approach one step further by introducing lazy instantiation of symbol nodes. In other words, a symbol node is only present in the syntax tree data structure when the parser requires the presence of multiple alternatives (Figure 14c). This results in significant space savings at the cost of minor implementation complexity. The disadvantage of the Ensemble implementation with respect to extensibility and flexibility is that symbol nodes have no distinct type identity in the implementation language. Thus the much-touted convenience of mapping the grammar-induced type system to that of the implementation language is lost.

Harmonia remedies this problem by treating symbol nodes as special operators in the language grammar, much like completer nodes. Consequently, a distinct class is generated for representing the symbol node for every phylum in the grammar. The only difference between symbol nodes and other operators is support for variable arity. This is required because the number of alternatives attached to a symbol node is not known *a priori* and depends on the particular input. Automatic generation of symbol node classes possesses the same benefits as generation of other node classes: a symbol node class can be easily extended to implement various features required by an analysis or an application. For example, special methods for semantic analysis may be used to deal with the ambiguous region represented by the symbol node. Another benefit of the generated symbol node classes is that the strong typing of nodes in the tree is preserved as the parent of a symbol node cannot distinguish it from any other valid non-terminal instance occurring in its place.

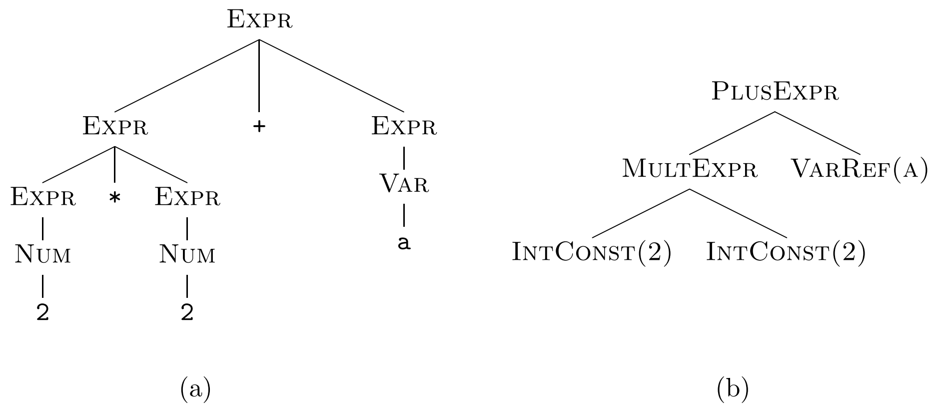


Figure 15: (a) A typical parse tree for the string `2 * 2 + a` in a simple expression language and (b) a corresponding abstract syntax tree. Values in parentheses denote attributes stored within nodes.

5.3 Toward a More Abstract Syntax Tree Model

So far our discussion of syntax trees has centered around the issues of representation and data structures. In this section, we focus on the ease of use of this data structure by syntax tree clients. The syntax tree clients we have in mind include HARMONIA applications, program analyses, and the language kernel itself.

Because syntax trees in the HARMONIA framework serve as the sole representation of the program source code, they represent a significant degree of source-level detail. However, when using syntax trees for a particular application, it is useful to consider the level of *abstraction* required for that application. Traditional compiler literature draws distinction between *concrete syntax trees* and *abstract syntax trees* (ASTs). Concrete syntax trees represent the steps taken by the parser when deriving the program structure. For this reason, concrete syntax trees are also called *parse trees*. An abstract syntax tree, on the other hand, is intended to capture the essence of program structure while *abstracting* the "uninteresting" details not used in further analysis of the source code. Such details might include keywords, punctuation, and other superficial deviations of shape typically present in concrete syntax trees. Figure 15 illustrates the differences between concrete and abstract syntax trees.

The shape of the abstract syntax tree is dictated by the abstract syntax, which is given by an abstract grammar for the language. Such a grammar would typically be very close to, if not the same as, the canonical grammar in a language specification document. The concrete syntax is given as a parse grammar employed by a particular language processor (for example, a compiler). There are two major ways in which a concrete syntax specification may differ from the abstract syntax. Firstly, the abstract syntax notation usually involves high-level grammatical abstractions such as symbol sequences, optional symbols, and nested alternations. Such abstractions are usually not directly supported by contemporary parsing tools, though there are notable exceptions such as JavaCC [16]. The second difference stems from the fact that most parsers restrict the class of accepted grammars to $LL(k)$ or $LR(k)$ (where k is typically 1), whereas abstract grammars for most programming languages are more naturally expressed using general context free grammars. A set of grammar transformations may be required to get the canonical language grammar into a form amenable to traditional parsing methods. These transformations are typically performed by the implementor of the tool and is what we call here "grammar-hacking".

Historically, language-based programming environments followed compilers in providing only the abstract representation of program source code. This representation was derived through a separate specification of language's abstract syntax as a mapping from the concrete grammar [33, 61], or vice versa [25, 42]. A textual representation would then be produced by *unparsing* the AST, possibly according to some pretty-printing rules [41]. Thus, the original textual representation of the program source code was lost (if it ever existed; the program could have been created with a syntax-directed editor). Some systems, such as PAN, avoided this problem by simultaneously maintaining several representations of program source code including text, token stream, and an abstract syntax tree. The higher-level representation would then be incrementally derived from the lower-level one upon each change. This approach has the disadvantage of having to simultaneously

maintain two representations and a potentially complicated mapping between character and token-based coordinate systems.

HARMONIA takes a different approach by representing program source code as a very stylized concrete syntax tree. This provides for explicit representation of keywords, punctuation, and even whitespace and comments, allowing such trees to serve as the sole representation of program source code. However, the HARMONIA framework employs several techniques that allow us to blur the distinction between concrete and abstract syntax. Firstly, the grammar specification notation is extended with high-level abstractions for symbol sequences, optional symbols, and nested alternations. This notation is presented in section 5.3.1. The actual parsing grammar is derived from this grammar through a series of simple grammar transformations during construction of the language module. These transformations are automated and reflected in the run-time grammar representation. Moreover, the information about what constitutes keywords, punctuation, and other non-abstract elements of the tree are also available. As a result, it is possible to provide an *abstract view* of the concrete syntax tree by selectively eliminating nodes. Tree-views, described in section 5.3.2, represent such a facility. A second source of non-abstractness resulting from transforming the grammar to an appropriate parsing class is eliminated by employing a powerful *Generalized LR* (GLR) parsing mechanism. For these reasons, we like to say that our syntax trees are "abstract-enough" and use the terms concrete syntax tree and AST interchangeably.

5.3.1 High-level Grammar Abstractions

The grammar specification language in HARMONIA is a variant of Extended Backus-Naur Form (EBNF) and provides special notations for symbol sequences (lists), optional symbols, and symbol nesting. This special specification language is processed by a grammar transformation tool called Ladle II (Section 6.2) to derive a canonical BNF specification to be processed by a specialized version of Bison (Section 6.3). An important side-effect of this transformation is that the correspondence of the high-level grammar abstractions and their expansions in the parsing grammar is captured and available at run-time for tree filtering (Section 5.3.2).

A *list* is a sequence of grammar symbols (terminals or nonterminals) of the same type, optionally separated by another grammar symbol. Our EBNF notation supports two list forms. The first form is a "star" list, which is a sequence of zero or more symbols. The second form is a "plus" list – a sequence of one or more symbols. For example:

```
BLOCK  →  '{' STMT* '}'
BLOCK  →  '{' STMT+ '}'
```

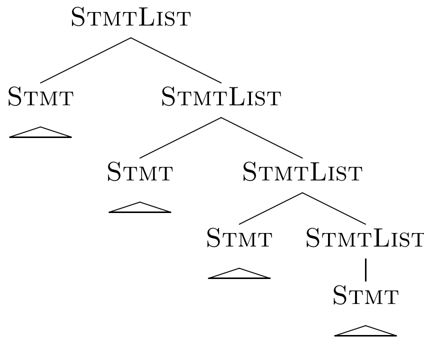
A useful extension to this traditional notation is the ability to specify a *separator* symbol, that is a symbol occurring between any two elements of a sequence.

```
BLOCK  →  '{' STMT* [';'] '}'
```

A number of techniques exist to incorporate the notion of sequences into parsing. One approach is to make the parser aware of the special semantics of symbol sequences. This technique is frequently used by the $LL(k)$ parser generators due to their inability to deal with left-recursion. An alternative is to rewrite the production containing a sequence into a canonical BNF form. This method does not require that a parser possess any special knowledge about the sequence notation. The most obvious transformation is the one traditionally employed by the grammar writers "manually": to use a left- or right-recursive expansion. For example, the "STMT+" above could be implemented with right-recursion as:

```
BLOCK     →  '{' STMTLIST '}'
STMTLIST  →  STMT
          |  STMT STMTLIST
```

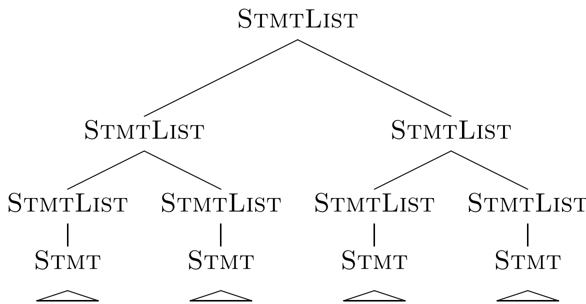
Parsing a sequence of "STMT" would then result in a "right-heavy" parse tree.



However, this right-recursive expansion has important implications for the Harmonia incremental analysis algorithms. In order for these algorithms to approach their asymptotic performance limits [59], the syntax tree data structure must support logarithmic searches. The right- or left- recursive implementation of repetitive structure leading to linked-list-like parse trees results in linear search performance. This problem can be alleviated by *non-deterministic* list expansion whereby the system guarantees the order of sequence items but says nothing about its internal structure. A non-deterministic expansion for for "Stmt" would correspond to the following production:

Block	→	`'{'` StmtList `'}'`
StmtList	→	Stmt
	|	StmtList StmtList

The ensuing parsing conflict can be statically resolved in favor of always elongating the sequence (shifting, for shift-reduce parsing), effectively resulting in a right-heavy tree similar to the one above. However, because the above grammar is ambiguous, the parse tree can be restructured to yield a more desirable shape, while still conforming to the grammar.



This mechanism for representing repeated structures originated with Pan. A similar method was utilized in the Ensemble system. However, neither of the two environments attempted to provide the full range of EBNF notational conveniences. This partly due to the fact that both Pan and Ensemble relied on an $LALR(1)$ incremental parser[7] which required a significant degree of hacking to get the grammar into the proper form. This grammar manipulation would remove the parsing grammar even farther from its "more abstract" equivalent.

The two other high-level grammatical abstractions provided by Harmonia, optional symbols and nesting, are aimed at reducing chain-rules in the grammar. Optional notation (denoted here by "?") allows right-hand-side symbols to be designated as optionally present. For example,

ForStmt → `'for'` `'('` InitStmt? `';'` TestExpr? `';'` UpdateExpr? `')'` Stmt

can be automatically translated to the BNF as follows

[7] Although the GLR parser found in Harmonia was prototyped in the Ensemble system, it was never utilized for any realistic languages.

ForStmt	→	'for' '(' InitStmtOpt ';' TestExprOpt ';' UpdateExprOpt ')' Stmt
InitStmtOpt	→	ϵ
	|	InitStmt
TestExprOpt	→	ϵ
	|	TestExpr
UpdateExprOpt	→	ϵ
	|	UpdateExpr

The EBNF notation for nesting permits "in-line" alternation. For example,

Block → '{' (IfStmt | WhileStmt | ForStmt) '}'

will be translated to the BNF as

Block	→	'{' StmtAlt '}
StmtAlt	→	IfStmt
	|	WhileStmt
	|	ForStmt

The transformations described here can be systematically applied by a special tool. Such a tool, called Ladle II, is provided as part of the Harmonia framework. In addition to translating the grammar from EBNF to BNF, Ladle II preserves the knowledge about each transformation and propagates it to the operator and phylum information structures. This knowledge is subsequently utilized to produce an abstracted view of the syntax tree by the tree-view facility described in the following section. Further discussion of the Ladle II tool is presented in Section 6.2.

5.3.2 Tree-views and Node Filtering

The grammar transformations in the previous section represent a mechanism for avoiding needless detail in the syntactic specification of the language structure. However, client applications still require a "reverse" mechanism to avoid exposing this detail when traversing a syntax tree. The *tree-view* facility of the Harmonia framework provides precisely this functionality.

A tree-view is a stateless adapter for traversing syntax trees. The underlying mechanism is based on omitting from the traversal nodes meeting a certain criteria. This "virtual removal" of nodes is achieved through a process known as *filtering*. A node filter is a simple function that can be applied to a tree node, with the result reflecting the decision of whether that node should be present in the filtered representation of the tree. Given a tree node, the filter function "responds" with one of the three results: keep this node in the filtered view, eliminate this node from the filtered view, or eliminate this node together with its subtree.

Many types of filters are possible. A useful kind of filter that we had previously discussed is the one providing an abstracted view of the syntax tree. Such a filter relies on the knowledge about the transformations performed on the high-level grammar representation by the Ladle II tool. For instance, this filter can "eliminate" the nodes corresponding to the sequence productions, whitespace connectors, keywords, etc. Figure 16 illustrates the use of a non-abstractness filter.

The use of filtering has some important consequences for syntax tree clients that use tree-views. Firstly, indexed access to child nodes is not permitted. This is because arbitrarily large sections of the subtree structure may be eliminated from the view, thus requiring $O(n^2)$ time to visit all child nodes, where n is the number of nodes in the subtree. (Basically, the child nodes need to be counted from the beginning until the right one is found.) By contrast, accessing child nodes in-order only requires $O(n)$. The second implication of filtered views is that the change discovery information described in the previous section can be distorted by filtering. This stems from the fact that what constitutes a node's immediate child in a filtered view may be a distant descendant of that node in the unfiltered view from which the change information has been derived. This leads to the invalidation of the nested changes invariant: the bit may be set while none of the child nodes in the filtered view has either local or nested changes. Moreover, the local change information that is, at least in part, derived from the changes in child node links may no longer be valid. For these reasons, the change discovery framework should be used in the presence of node filtering with a certain amount of caution.

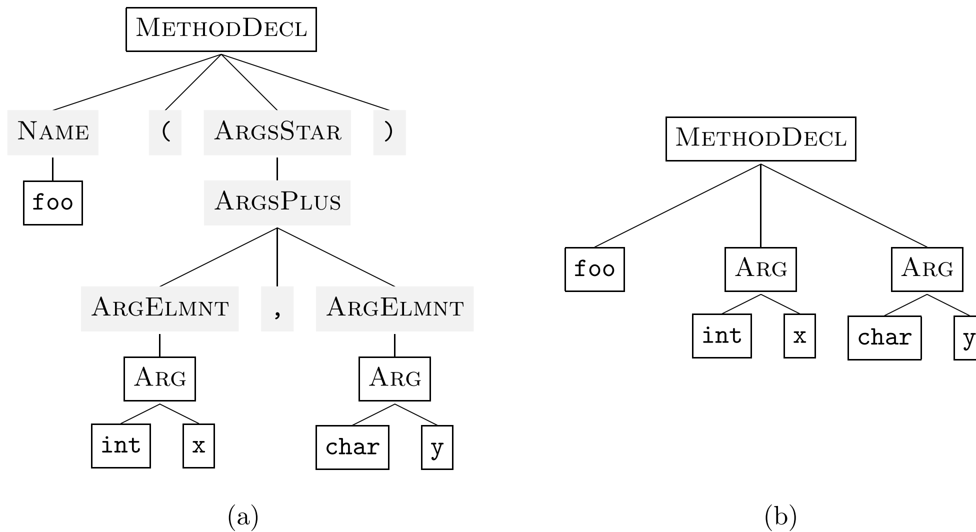


Figure 16: Use of tree-views to support abstraction: (a) a parse tree prior to filtering and (b) the same tree with non-abstract (shaded) nodes eliminated from the view.

5.4 Static Program Analysis Framework

A major benefit of building applications with the Harmonia framework is the wealth of information about the program source code that can be made available to the user. Internally, the Harmonia framework only requires that a lexical and a syntactic analyzer be provided for every supported language as these are needed for maintaining the syntax tree data structure. However, all but the simplest Harmonia applications can benefit from having more extensive knowledge about the program. Such information is typically computed by a variety of static program analyses.

There are two aspects to the static analysis of programs. The first is akin to the semantic phase of a compiler, namely, binding analysis, name resolution, type checking, and other analyses derived from the semantic definition of the language. We refer to that as "semantic analysis". The second determines properties of the particular instance, that is, the program. Examples of such analyses include data flow, alias analysis, complexity measures, slicing, etc. We refer to that as "program analysis".

Both types of static program analyses can be conveniently accommodated within the Harmonia framework. The only requirements are that an analysis be structured as a computation on the syntax tree data structure[8] and fit into the analysis model described in Section 5.1.4. The implementation of the analysis is supplied as part of the language module. The analysis may or may not be incremental. An incremental analysis is such that the amount of computation to be performed upon a modification is proportional to the amount of change. While the Harmonia framework does not insist on static program analyses being incremental, it is typically beneficial to the user that they be so. One particular way of making static program analyses incremental at the level of translation units is discussed in Section 5.4.2.

The Harmonia framework includes specialized support for implementing program analyzers using the *visitor* design pattern. This design pattern involves "visits" to nodes of the syntax tree through method invocations on node objects. Visits to child nodes are directed by the visit methods on the parent node. The visit methods for any static program analysis can be coded in C++ and are supplied by the language description writer as part of the language module definition. The implementation of the analysis can also be generated from a declarative specification. However, no tools for doing this are currently available in the Harmonia framework.

[8]While it is traditional to implement static semantic analysis as a computation on the syntax tree, other program analyses (for example, control/data flow, aliasing, etc.) typically use a lower-level intermediate representation. However, it has been shown (for example, in Morgenthaler [30]) that those analyses may be re-formulated in such a way as to work on syntax trees just as well.

5.4.1 Specialized Support for Tree-based Analyses and Transformations

Tree-based analyses are traditionally implemented on an abstract syntax tree data structure which eliminates the details of the concrete syntax. One way to achieve this in Harmonia is to use the previously described tree-view facility. However, because the tree-views are language independent and the analyses are usually language-specific, it is more convenient for the language module implementor to use specialized tree accessor and mutator methods.[9] These accessor and mutator methods, or simply accessors and mutators, can be generated automatically by the grammar processing tools and named in the manner specific to each programming language. Additionally, accessors and mutators provide a "view" of the tree in which internal nodes used for implementing the high-level grammar abstractions discussed in Section 5.3.1 are eliminated from the traversal. The notation for naming accessors and mutators is exemplified by the following grammar fragment:

$$\begin{array}{lll} \textsc{Expr} & \rightarrow & \textit{left}{:}\textsc{Expr}\ \texttt{'+'}\ \textit{right}{:}\textsc{Expr} \\ & | & \ldots \end{array}$$

The construct "*left*:Expr" denotes that the corresponding subtree maybe accessed through the name "left". The generated class for this production will include two accessor methods which take no arguments and whose return type is the phylum for that subtree, as well as two mutator methods.

```
class PlusExpr {
    ...
    Expr get_left();          retrieve the first child of this node
    Expr get_right();         retrieve the third child of this node
    void set_left(Expr e);    set up a link to the first child of this node
    void set_right(Expr e);   set up a link to the third child of this node
}
```

The generated methods may be used in the implementation of any program analysis using the visitor pattern. For example, to implement the type checking phase of the static semantic analysis on a `PlusExpr` node, one could use the following method

```
TypeInfo PlusExpr::type_check() {
    TypeInfo left_type = get_left()->type_check();    type-check the left child
    TypeInfo right_type = get_right()->type_check();  type-check the right child

    check that types are compatible and compute the type of this expression

    return my_type;
}
```

The accessors and mutators are also used for abstracting the implementation details of the high-level grammatical constructs such as sequences, optional symbols, and nested alternations.

Symbol sequences are abstracted using the *iterator* design pattern. A class implementing the iterator with all the appropriate accessors is generated for each symbol sequence on the right-hand-side.

$$\textsc{Decl} \rightarrow \textsc{ReturnType}\ \textsc{MethodName}\ \texttt{'('}\ \textit{args}{:}(\textit{type}{:}\textsc{Type}\ \textit{var}{:}\textsc{Name})*\ \texttt{')'}\ \textsc{MethodBody}$$

The generated class contains the accessors for the sequence as well as the definition for the iterator class.

[9] Mutator methods are necessary because an analysis might need to modify the shape of the syntax tree to reflect the new knowledge about the program discovered by the analysis.

```
class MethodDecl {
    ...
    class args_iterator {
        ...
        void advance();                    advance this iterator
        bool is_done();                    test whether this iterator is at the end
        int  get_arity();                  return the number of items in each element
        Type get_type();                   get item 'type' from the current element
        Name get_var();                    get item 'var' from the current element
        void set_type(Type t);             set item 'type' in the current element
        void set_var(Var v);               set item 'var' in the current element
        Node get_item(int i);              get the given item from the current element
        void set_item(int i, Node n);      set the given item in the current element
        void insert_new_element_before();  insert a new element before the current one
        void insert_new_element_after();   insert a new element after the current one
        void remove_current_element();     remove current sequence element
    }

    args_iterator get_args()               return a new args_iterator object
}
```

Such an iterator may be used in implementing the static semantic analysis on a `MethodDecl` node as follows:

```
void MethodDecl::semant() {
    args_iterator i = get_args();
    while (!i.is_done()) {
        i.get_type()->check_defined();              check that the type is defined
        i.get_var()->check_duplicates();            check for duplicate var names
        symbol_table.add(i.get_type(), i.get_var()); add type/var pair to the symbol table
        i.advance();                                advance the iterator
    }
}
```

Optional right-hand-side symbols yield a number of special methods for manipulating the subtree corresponding to optional productions. For instance, the following grammar segment

CLASS → 'class' *classname*:IDENT *super*:('extends' *name*:IDENT)? '{' CLASSBODY '}'

yields the corresponding class definition:

```
class ClassDecl {
    ...
    bool  has_super();                     test for existence of the optional subtree
    void  add_super();                     insert a new optional subtree
    void  delete_super();                  remove the optional subtree
    IDENT get_super_name();                get item 'name' from the optional subtree
    Node  get_super_item(int i);           get the given item from the optional subtree
    void  set_super_name(IDENT n);         set item 'name' from the optional subtree
    void  set_super_item(int i, Node n);   set the given item in the optional subtree
}
```

Our final example demonstrates the generated class definition for the nested alternation construct. Similarly to optional symbols, the class for nested alternation contains special methods for manipulating the corresponding subtree.

TableAccess → Table '[' *deref*:(*ident*:IDENT | *string*:STRING) ']'

```
class TableAccess {
    ...
    bool   is_deref_ident();            test if the alternative is an 'ident'
    bool   is_deref_string();           test if the alternative is a 'string'
    IDENT  get_deref_ident();           retrieve the alternative as 'ident'
    STRING get_deref_string();          retrieve the alternative as 'name'
    void   set_deref_ident(IDENT n);    set the alternative to 'ident'
    void   set_deref_string(STRING n);  set the alternative to 'string'
}
```

The code-generating transformations described in this section are performed by Bison II, a modified version of the Bison parser generator. The precise specification for these transformations are presented in Section 6.3 describing the Bison II tool.

5.4.2 Program Units

A distinguishing characteristic of many static program analyses is their need to consider the program in its entirety. For example, type resolution requires the knowledge of all types declared in the program. In our discussion so far, we assumed that the entire program is manifested through a single syntax tree. However, not only is this extremely impractical for any size-able program, but also precludes the use of any external applications components, like function or class libraries. In practice, a program may consist of multiple *translation units* such as files, modules, classes, etc. A compiler or interpreter processes these translation units together with *library units* to produce one or more *execution units*.

The translation and library units are organized to form a software system through a *system model*. The system model defines a collection of source entities and the context in which these entities are interpreted. The system model may be implicit, with a collection of translation units to use being determined by the compiler through searching some repository (for example, a directory path in the file system). The system model may be *ad hoc*, consisting merely of an informal description for how to derive the execution units; such is the system model based on UNIX makefiles. The system model may also be formalized, defining the precise collection of translation and library units to use for a given software system as well as the semantic dependencies between various translation units. The project management facilities of many modern integrated development environments provide such a capability.

The Harmonia framework takes a very rudimentary approach toward managing system models. A program is represented as a collection of *program units* (a generic term for translation and library units). This collection is managed by the Harmonia application. It is also the application's responsibility to provide a sensible user-visible system model. For example, such a model may be based on the notion of a *project* – a collection of program units that represent a single application. Program units in Harmonia need not correspond to the translation units of the compiler. Since Harmonia has no notion of "files", what constitutes a program unit is determined entirely by the language module implementor. For example, a program unit for C or C++ may consist of a single top-level declaration, for Java – a class, for Modula-2 – a module. Consequently, it is also the responsibility of the Harmonia application to "import" program units into the framework and "export" them to the outside world.

Program units representing source-level entities are called *source units*. The internal data structure underlying these source units is the syntax tree. A different type of a program unit is a *library unit*, which represents an external application component. The library units need to be included in the Harmonia system model, because many program analyses require knowledge of the library components that are used by the software system. The library unit's internal representation is also determined entirely by the language module's implementor.

5.4.3 Incremental Static Program Analyses

The goal of the static program analysis in Harmonia is to provide Harmonia applications with sufficient information to assist the user in manipulating and navigating program source code. For example, a user may wish to navigate from the use of a variable to its definition, which requires performing the name resolution phase of the static semantic analysis. Such a usage scenario dictates that the results of an analysis should be available to the user even when the program source code is in an intermediate state of semantic ill-formedness. Moreover, the results should be rapidly updatable to provide smooth uninterrupted user service. This suggests that a static program analysis should be incremental, so that the amount of work performed by the analysis scales with the size of a change.

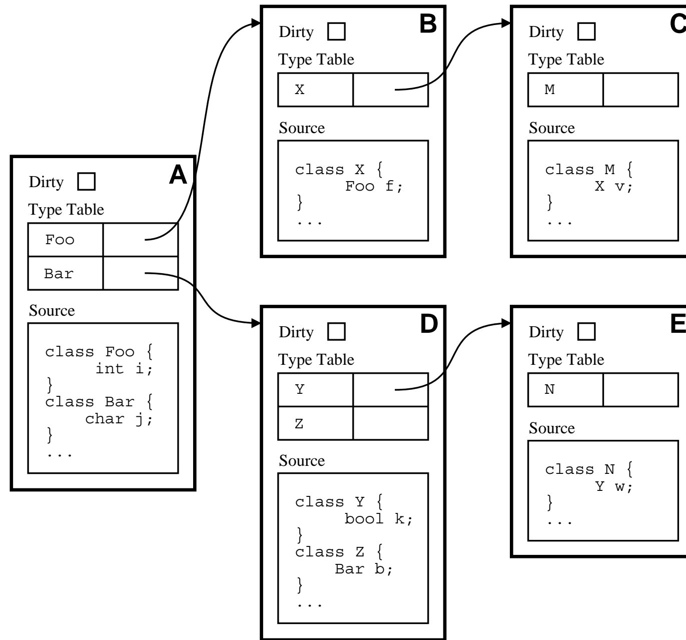
An important observation is that incremental static program analysis has a much greater scope and effect than the incremental syntactic analysis, which, in turn, has a greater scope and effect than the incremental lexical analysis. Static program analysis at each point in a program depends potentially upon the entire program, whereas parsing depends only on the enclosing phrase and lexical analysis depends only on adjacent tokens. Because the effects of lexing and parsing are so localized, the incrementality of the lexical and syntactic analysis in Harmonia is very fine-grain: the analysis updates only the nodes affected by a change. Maddox [28] shows that a static program analysis (in his case, static semantic analysis) can also be incrementalized at such a level. However, his approach suffers from several significant drawbacks:

- Because of the non-local nature of semantic analysis, extensive dependency information must be maintained for every semantic attribute. In particular, every data object involved in an attribute computation must be linked to every other object whose value depends on it. While some coarse-grain dependencies may be discovered through a static analysis of the declarative specification of attribute computation, the fine-grain dependencies need to be maintained dynamically, increasing space and time complexity of the analysis.
- Having to maintain detailed dependency information complicates the analyses whose scope constitutes the entire program, such as static semantics. Maintaining non-local dependency links into a syntax tree that represents a different module makes it non-trivial to maintain portions of that syntax tree outside the main memory if they are not being actively used.
- Libraries and other external components need to be treated as a special case by the analysis code which needs to be aware of a different internal representation underlying those modules.

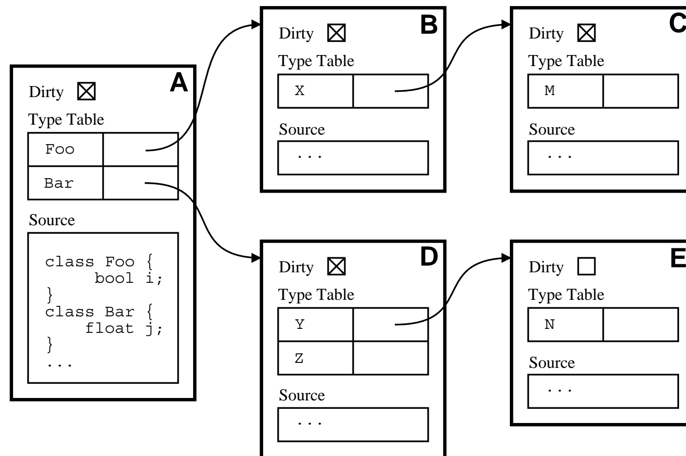
By contrast, Harmonia offers a framework for coarse-grain static program analysis based on program units. Each program unit is analyzed in a batch fashion: the results of the previous analysis for that program unit are discarded and recomputed as needed.[10] The frequency of re-analysis is determined by a Harmonia application. While nothing prevents the application from invoking the analysis upon every modification to the program, a more sensible strategy is to run the analysis code whenever a user-visible service requires the results of that analysis. Because the program source code may be in an intermediate state, the analysis must continue to function in presence of syntactic or semantic ill-formedness. In such a case, a query may return an ambiguous result (a conservative approximation of the actual value) with the expectation that the requester can deal with the ensuing ambiguity.

The incrementality of a static program analysis is achieved through the use of a stylized interface to the computed information. Any query for the analysis result may include a call-back routine to be invoked should the result of that query become invalidated by a subsequent modification to the program. The queries used internally by the analysis itself must include an invalidation call-back in order to provide the incremental behavior. Such queries originate while updating attributes of a program unit and the effect of the call-back is to mark the dependent program unit as *dirty*, that is, requiring re-analysis. If a semantic query requires the analysis results from a program unit marked "dirty", the execution of the query is suspended and that program unit is re-analyzed. Figure 17 demonstrates the use of this mechanism on a small example. A drawback of such an approach is that circular dependencies between program units may

[10] It is conceivable that some of the previous analysis results are kept to improve error reporting; however, no attempt is made to update them in an incremental fashion.



(a) The state of the program prior to the modification. Type table contains a list of types defined in that program unit and represents the queriable result of the semantic analysis. For each entry in the type table, a link to all program units that semantically depend on that entry is established using the call-back mechanism described in this section. (Transitive dependencies need not be recorded).



(b) The state of the program following a modification to program unit A. Program unit A is marked "dirty" as a direct result of the modification. Program units B and D are marked "dirty" because of their dependency on types `Foo` and `Bar`, respectively. Program unit C is marked "dirty" due to its transitive dependency on program unit A. (This is detected dynamically when the semantic information for `class X` is invalidated.) Program unit E is *not* marked "dirty" because it is unaffected by the modification.

Figure 17: Change propagation in incremental semantic analysis.

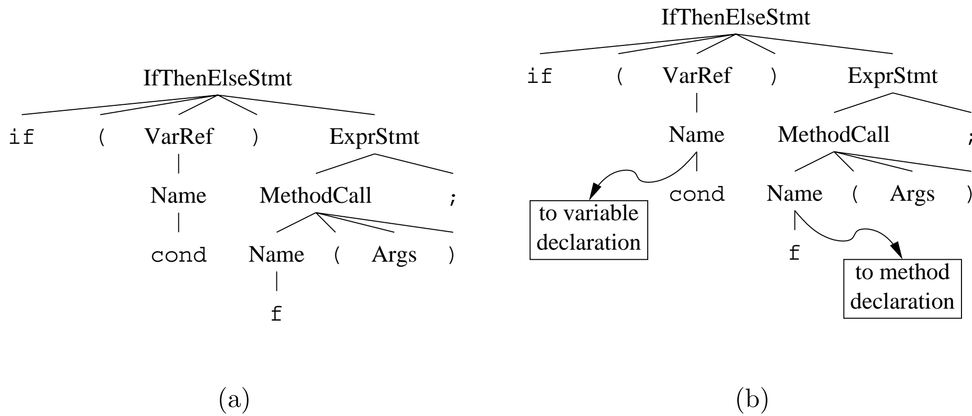


Figure 18: (a) Abstract syntax tree and (b) abstract syntax graph, which is an AST annotated with semantic links.

cause non-termination of an analysis. It is the job of the implementor of the analysis to ensure that such situations do not arise. A typical solution is to split the analysis into two distinct phases such as symbol table construction and type checking.

5.5 Interfacing External Tools

A conscious objective of the Harmonia framework is to coexist with software engineering tools constructed outside of the framework. For example, an application for reverse engineering may require a front-end for syntactic and semantic analyses, which can easily be built with Harmonia. Another example is an externally developed language analysis or transformation component that can complement language services built into the framework.

Integration with external tools may be achieved through several approaches. One is to provide an API for accessing the syntax tree data structure. An important requirement is that such an API constitute a well-recognized standard, so that applications and toolkits adhering to this API are widely available. The Document Object Model (DOM) [53], an API for accessing structured documents, represents one possibility. Since program source code can be considered a particular type of a structured document, DOM API enables Harmonia applications easily to connect not only other language-aware programming tools, but also the multitude of DOM-enabled applications such as viewers, parsers, or transformers. At the time of this writing the Harmonia framework is being augmented with an API implementing the DOM standard.

A different methodology for the integration of external tools is to devise an *exchange format* through which the data structures may be transferred in an application-neutral way. An attractive choice is to use XML, a language for encoding structured documents [62]. The choice of XML as an encoding format nicely complements the choice of DOM API: many libraries implementing various XML-based services for DOM are readily available. These services include XML input and output, tree transformations, etc. However, XML is merely a text-based encoding format; many choices still need to be made regarding the specifics of the XML-based representation. The following section considers some these choices in further detail.

5.6 Designing an Exchange Format

In this section, we will use the program of Figure 18 as our running example. The abstract syntax graph (ASG) of Figure 18 represents a syntax tree annotated with attributes pertaining to static program analysis. It is precisely this ASG data structure that needs to be encoded in XML in order to be useful to tools outside of Harmonia.

5.6.1 Trees vs. Graphs

Since an analyzed program may be viewed both as an attributed syntax tree and as a abstract syntax graph, it is possible to encode either the tree data structure (as nodes with attributes whose values may reference other nodes), or the graph data structure (as nodes and explicitly represented edges).

For the graph-like encoding, one option is to employ a general-purpose graph encoding similar to GraX [20] and GxL [24]. In such an encoding, the `MethodCall` fragment of the ASG in Figure 18 may be represented in XML as follows:

```
<node id=1 name=MethodCall>
    <edge target=2 type=child/>
    <edge target=4 type=child/>
    <edge target=5 type=child/>
    <edge target=6 type=child/>
</node>
<node id=2 name=Name>
    <edge target=3 type=child/>
    <edge target=42 type=decl/>
</node>
<node id=3 name=IDENT text="f"/>
<node id=4 name=LPAREN text="("/>
<node id=5 name=Args></node>
<node id=6 name=RPAREN text=")"/>
```

(where `decl` is a reference to a method declaration node somewhere in the syntax tree whose `id` is 42.)

Alternatively, we can encode the tree data structure directly, exploiting the fact that XML syntax is particularly well suited for representing hierarchical data. This approach is also attractive since in Harmonia the tree structure produced by the parser is the primary basis for all the manipulations, while the attributes resulting from further program analysis, which induce the graph structure, are secondary and may not be present at all if semantic analysis services are not invoked. The following demonstrates such a "tree-centered" encoding:

```
<node id=1 name=MethodCall>
    <node id=2 name=Name decl=42>
        <node id=3 name=IDENT text="f"/>
    </node>
    <node id=4 name=LPAREN text="("/>
    <node id=5 name=Args></node>
    <node id=6 name=RPAREN text=")"/>
</node>
```

5.6.2 Designing an Exchange Schema

Any general data exchange format requires a specification defining how a data structure is to be encoded in that format. This specification is called an (exchange) *schema*, and the encoded data is said to be an instance of that schema. The exchange schema may be implicit: as long as the communicating parties agree on the format, no separate specification of the encoding is necessary. Oftentimes, however, a separate exchange schema is provided. Not only can such a schema be used to validate the encoded data, but also it can define how that data is to be interpreted by the communicating parties. If the latter is the case, a schema must be exchanged along with the data.[11]

[11] Of course when exchanging schemas, the schema itself serves as data, requiring a *meta-schema* to define its format. If that meta-schema is to be exchange as well, a meta-meta-schema may be needed, ad infinitum!

The XML standard defines a special schema language called *document type definition* (DTD). A DTD defines the syntactic structure of an XML document. For example, the encoding in the preceding example may be described by a very simple XML DTD:

```
<!ELEMENT node (node)*>
<!ATTLIST node ...all possible node attributes...>
```

The disadvantage of this simplistic DTD is that it imposes almost no restrictions on the shape of the syntax tree, making tools using this exchange format amenable to malformed input. Additionally, an encoding in this DTD requires a separate data schema to be provided along with the data so that each tool can map tree nodes to entities and relations known to that tool. Since tree nodes are naturally typed according to the abstract grammar describing the shape of the AST (see Section 5.1.1), we can remedy both of these problems by providing a more rigorous DTD. On the one hand, this lets us employ a readily available validating XML parser. On the other hand, a tool can interpret this DTD as a data schema and map tree nodes to entities and relations (represented as XML element attributes) it can understand. The following is our running example in such a *typed* encoding:

```
<MethodCall id=1>
    <Name id=2 decl=42>
        <IDENT id=3 text="f"/>
    </Name>
    <LPAREN id=4 text="("/>
    <Args id=5></Args>
    <RPAREN id=6 text=")"/>
</MethodCall>
```

In Harmonia the DTD for such an encoding can be generated automatically from the very same language specification that drives the incremental analyzer. Consider the following simplified syntactic specification for the Java method call:

MethodCall	→	Name `LPAREN` Args `RPAREN`
Name	→	`IDENT`

Given the above grammar, the DTD below can be generated automatically by a simple transformation.

```
<!ELEMENT MethodCall (Name, LPAREN, Expr*, RPAREN)>
<!ELEMENT Name       (IDENT)>
<!ATTLIST MethodCall id ID #REQUIRED>
<!ATTLIST Name       id ID #REQUIRED
                     decl IDREF #REQUIRED>
```

This approach ties our exchange format to the language grammar, requiring tools to be able to interpret schemas as well as the data. An alternative approach is to incorporate all supported languages into a single schema. Such a schema is used in the DATRIX framework [27], which supports C, C++, and Java. While this approach works well for DATRIX because these languages share common concepts, it is not practical for Harmonia, which supports many languages of vastly different styles.

5.6.3 Encoding Program Text

The primary application of the Harmonia framework is the construction of language-sensitive *front-end* tools such as editors, source browsers, source-level transformers that may need to manipulate program source code, rather than (or as well as) its structural representation embodied in the AST. For such tools, it is important to provide the means to retrieve not only program structure, but also various non-structural source elements such as comments, whitespace, identifier spelling, or punctuation. While this information can be made

available as part of the tree structure, an attractive alternative is to simply "mark-up" the source code with the structural information represented in the syntax tree:

```
<MethodCall id=1>
    <Name id=2 decl=42><IDENT id=3>f<IDENT></Name>
    <LPAREN id=4>(</LPAREN>
    <Args id=5></Args>
    <RPAREN id=6>)</RPAREN>
</MethodCall>
```

An advantage of such an encoding (as opposed to, for example, encoding a node's textual content as one of its attributes) is that a tool interested in program text only needs to strip off XML tags, whereas a tool that only cares about program structure may safely ignore the text.

5.6.4 Open Issues

The exchange format presented in the previous section embodies a number of decisions, the most significant of which is the generation of exchange format schemas directly from language grammars, utilizing the same specification used to construct the language parser. This design, is currently being prototyped as part of the ongoing effort to implement the DOM API for the Harmonia syntax tree. While the presented exchange format is sufficient for simple data interchange, a number of important issues still need to be resolved:

- **Schema and grammar evolution.** Since the exchange format is based on the language grammar, there needs to be a way for tools that utilize different grammars for the same language, or different versions of the same grammar, to make sense of each other's data. We believe that this issue can be addressed by simply translating the XML documents from one format to another. An attractive tool for implementing such a translation is XSLT, a transformation language for XML [63].

- **Data granularity.** Our exchange format provides a considerable level of detail that may not be needed by some tools. Many high-level modeling tools may be interested only in the "declaration level" or even "architecture level" facts about the program. While such data may be computed and incorporated into the syntax tree by the Harmonia framework, transporting the entire AST (and indeed computing the entire AST) may prove to be unnecessarily expensive.

- **Incorporating revision information.** The exchange format described here discards the wealth of fine-grain revision information available as part of the Harmonia syntax tree data structure (see Section 5.1.3). However, some version-aware applications (including those built with the Harmonia framework) may require access to that data, while other tools may wish to ignore it. To facilitate this task, our future goal is to augment the presented exchange format to incorporate revision information.

6 Encapsulating Language-Specific Information

The multi-language support is provided in Harmonia through use of dynamically-loadable extensions to the language kernel called *language modules*. The language modules are loaded on demand; multiple modules can be loaded simultaneously. The only requirement of language modules is that they conform to a very simple interface for invoking program analyses. In theory, language modules may be entirely hand-coded; in practice, however, the generation of language modules is largely driven by a declarative description.

The typical process for building a language module in Harmonia is depicted in Figure 19. The input consists of a lexical specification, a grammar for the programming language, and a small hand-coded file describing the language module interface. Optionally, the input may include extra code to be included into the generated definitions of the AST classes (see Section 5.1.1).

The lexical specification is processed by the off-the-shelf Flex scanner generator [38] to produce a batch lexer. This lexer can be used by the Harmonia language kernel to provide incremental lexical analysis service (see Section 5.1.5). The syntactic specification is pre-processed by the Ladle II tool, whose main job

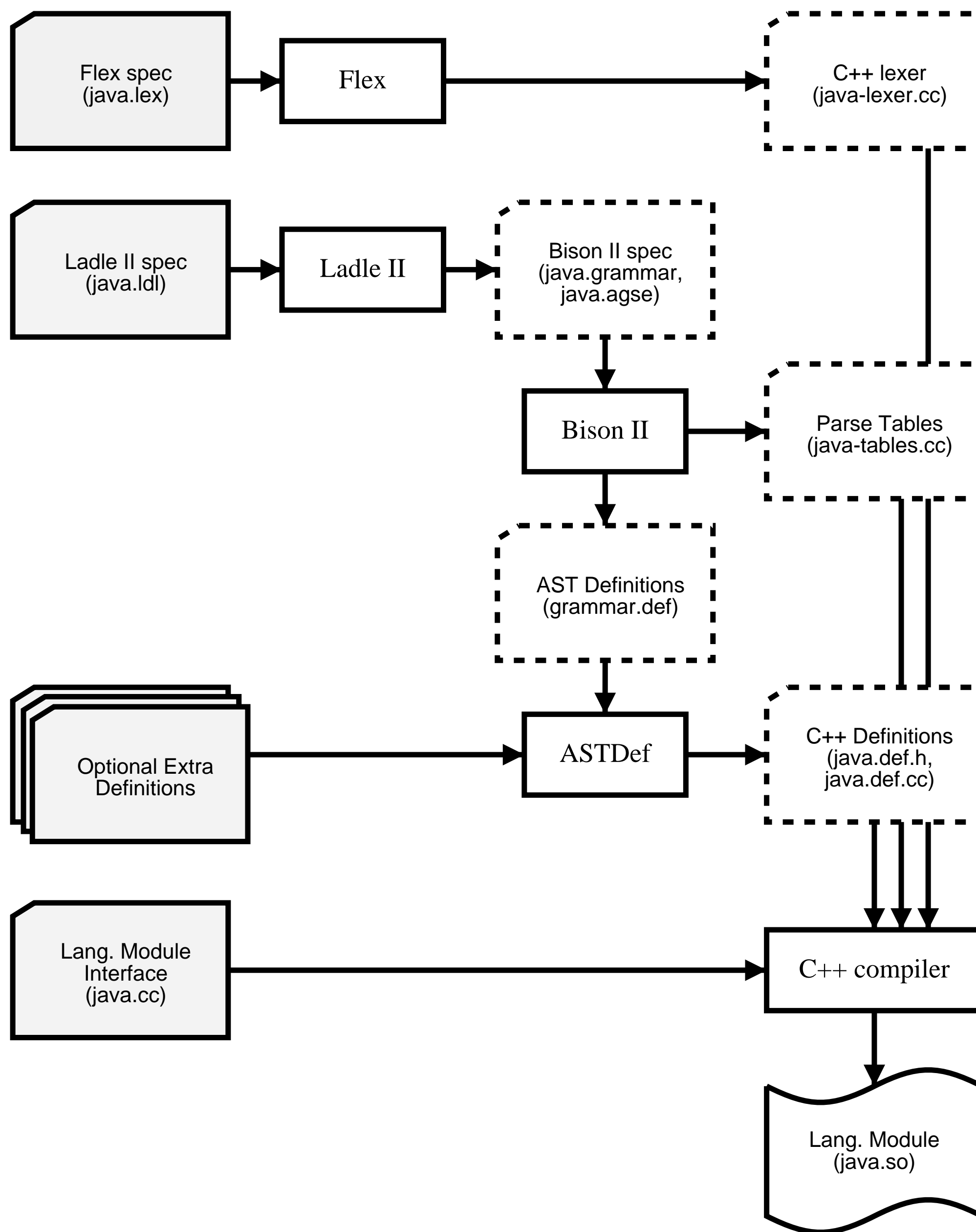


Figure 19: Building a language module in Harmonia.

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

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

is to perform the grammar transformations described in Section 5.3.1. The output of Ladle II is a syntactic specification compatible with Bison [13]. We use a modified variant of Bison, called Bison II, which outputs parse tables and AST class definitions rather than parser source code. The AST definitions are specified in a special language, called ASTDef, that provides minor "syntactic sugaring" on top of C++. These definitions are combined with any extra definitions provided by the language module implementor and translated into the C++ source code. Finally, a C++ compiler is used to combine the AST definitions, parse tables, the batch lexer, and the language module interface implementation into a dynamically-loadable library for the Harmonia language kernel.

6.1 Flex

The Flex scanner generator used when constructing a Harmonia language module is a standard Unix tool for building lexical analyzers. For this reason, we will not discuss it further in this report and refer the reader to the Flex manual [38]. The only restriction imposed on the lexical specification for a Harmonia language module is that it be written in such a way as to not preserve any state across the token boundaries. This invariant is an important prerequisite for constructing lexical specifications amenable to incremental lexing. While this precludes the common practice of maintaining lexical state in global variables, in practice it is not a significant impediment to writing efficient lexical analyzers. The incremental lexer driver does include support for Flex's "state conditions". However, only one state condition may be active at a time; stacks of state conditions (another Flex feature) are not supported.

6.2 Ladle II

The function of the Ladle II tool[12] is to transform the high-level EBNF syntactic specification to a BNF grammar amenable to passing to the Bison parser generator [13]. The input format of Ladle II is virtually identical to that of Bison with the following key differences:

- **EBNF grammar notation.** Ladle II provides a set of EBNF extensions to the Bison grammar specification language. These extensions, discussed in Section 5.3.1, include symbol sequences, optional symbols, and nested alternations. The complete set of grammar transformations involved in deriving

[12] Ladle II is a very distant relative of the original Ladle tool from Pan [11], hence the "II" in its name.

the BNF version of the grammar is given in Table 1. The EBNF constructs may be arbitrarily nested (as in "S → (X | Y)*") and so the transformations are applied to the input grammar in a top-down fashion until no more EBNF extensions remain.

- **Explicit class naming.** Ladle II includes means for the grammar writer to explicitly name the AST production node classes generated by Bison II (see Section 5.1.1) using the following notation:

  S → *A* ⇒ *"ProductionName"*

  For example, different productions for a statement may be named as follows:

STMT	→	FORSTMT	⇒ "ForStatement"
	|	WHILESTMT	⇒ "WhileStatement"
	|	...	

  If the explicit names are not specified, Ladle II will generate production class names automatically by appending the production number to the non-terminal which it derives, for example `Stmt_0`. The class name is communicated to Bison II which is responsible for generating class definitions.

- **Tree accessors and mutators.** Ladle II permits specification of the named tree accessors in the grammar (see Section 5.4.1). The accessor and mutator names are given using the "colon notation" as follows:

  S → $a{:}A$

  As a result, the production class for this rule will contain methods named `get_a()` and `set_a()`. The precise specification for generating these methods is presented in the following section on Bison II.

- **Special token declarations.** Ladle II also provides a notation for declaring which token types constitute whitespace and comments so that the parser knows to treat them specially (see Section 5.1.1). Whitespace and comment tokens are declared using `%whitespace` and `%comment` directives, similar to Bison's `%token` declaration. Multiple token types may be declared as belonging to the whitespace and comment token classes; these tokens should not appear in the grammar.

- **Absence of syntactic actions.** Since the actions for constructing a parse tree are built into the HARMONIA parser, no additional actions may be specified with the syntactic description.

6.3 Bison II

Bison II is a variant of the Bison parser generator modified from the original version to include features necessary for constructing HARMONIA language modules. Unlike its predecessor, Bison II does not generate any parser code. Instead, its output consists of parse tables that can be used by the HARMONIA parser driver and the definitions for the AST node classes (see Section 5.1.1). For the GLR parser, Bison II also outputs a table of conflicts which includes additional actions to be taken in parser states with shift/reduce and reduce/reduce conflicts.

To minimize the number of conflicts that the GLR parser has to deal with at runtime, Bison II incorporates mechanisms for static conflict filtering. Firstly, the Bison method of conflict elimination through precedence declarations is supported. Secondly, some special conflicts can be automatically resolved due to the special knowledge about the grammar. The conflicts in this category include those resulting from the non-deterministic expansion of symbol lists by Ladle II (see Section 5.3.1). These shift/reduce conflicts are simply resolved in favor of a "shift", indicating a right-recursive interpretation. This is possible because the order in which sequence nodes are constructed is unimportant due to the re-balancing taking place upon the completion of a parse.

The generated AST definitions include phyla classes representing terminal and non-terminal symbols in the grammar as well as the operator classes representing productions. If the grammar writer chooses to include tree accessors for writing tree-based analyses (see Section 5.4.1), the corresponding accessor methods

<table>
<tr><th>Grammar Construct</th><th>Generated AST Definition</th></tr>
<tr><td>S → a:A b:B</td><td><pre>operator S_0 extends S {
    ...
    A   get_a()
    B   get_b()
    void set_a(node)
    void set_b(node)
}</pre></td></tr>
<tr><td>S → seq:(a:A b:B)*</td><td><pre>operator S_0 extends S {
    ...
    class seq_iterator {
        ...
        A   get_a()
        B   get_b()
        void set_a(node)
        void set_b(node)
    }
    seq_iterator get_seq()
}</pre></td></tr>
<tr><td>S → opt:(a:A b:B)?</td><td><pre>operator S_0 extends S {
    ...
    bool has_opt()
    void add_opt()
    void delete_opt()
    A   get_opt_a()
    B   get_opt_b()
    Node get_opt_item()
    void set_opt_a(node)
    void set_opt_b(node)
    void set_opt_item(i, node)
}</pre></td></tr>
<tr><td>S → alt:(a:A | b:B)</td><td><pre>operator S_0 extends S {
    ...
    bool alt_is_a()
    bool alt_is_b()
    Node get_alt()
    A   get_alt_a()
    B   get_alt_b()
    void set_alt_a(node)
    void set_alt_b(node)
}</pre></td></tr>
</table>

Table 2: Deriving accessor and mutator method names from the syntactic specification. The generated definitions are given in the ASTDef language described in the subsequent section.

are generated for each operator class. Table 2 summarizes the rules for generating and naming accessor methods. The AST definitions are specified in the ASTDef language described in the following section.

The input to Bison II consists of the grammar specification post-processed by Ladle II. The language module implementor need not deal with Bison II input directly.

6.4 ASTDef

ASTDef is a language for specifying AST definitions. ASTDef specifications are processed by a tool of the same name to yield a C++ implementation of the AST node classes. On the surface, ASTDef provides some minor "syntactic sugaring" on top of C++. However, the reason for using such a domain specific language rather than C++ directly is twofold. Firstly, ASTDef permits combining definitions for the same AST node class from multiple sources. As a result, the various aspects of the AST node implementation may be conveniently factored. For instance, the implementation of static semantic analysis may be provided separately from the implementation of the control flow analysis. The second reason for using ASTDef is that the implementation details of the syntax node features such as slots, attributes, methods, etc. may be conveniently hidden from the grammar writer. For example, every named node attribute (Section 5.2.2) entails a special accessor method as well as an entry in several internal tables (see below); the use of ASTDef prevents the implementor of the language module from knowing about these details. The following sections describe the ASTDef language in more detail.

6.4.1 High-level Structure

An ASTDef file consists of two distinct sections: the preamble and the operator/phyla definitions. The preamble lists the external components required to process the AST definitions. The `import` directives specify where to find node class definitions referenced but not defined in this file. No code is generated for the imported definitions. The `include` directive lists C++ include files required by the C++ compiler to process the resulting output. For example, a preamble may look as follows:

```
import "Node.def";
import "ParentNode.def";
import "TokenNode.def";
import "SeqPairNode.def";

include "common/stl.h"
include "common/PooledString.h"
include "lk/isemant/Scope.h"
include "lk/isemant/NestedScope.h"
include "lk/isemant/SemantErrors.h"
include "lk/node/UltraRoot.h"
```

6.4.2 Phyla and Operators

Phylum and operator declarations get translated to C++ class definitions in the output. A phylum corresponds to a non-terminal in the grammar; an operator corresponds to a production. A phylum may not have a superclass, whereas an operator must be a subclass of its phylum class. This relationship is expressed by indicating that an operator `extends` its phylum. Additionally, the operator class must subclass one or more classes provided by the Harmonia language kernel that define the node abstract data type (ADT) (see Section 5.1.1). For example:

```
phylum program { ... }
operator CoolProgram extends program, ParentNode { ... }
```

The node ADT classes available for subclassing include `Node` – used for zero-arity productions, `ParentNode`

– for normal productions, `TokenNode` – for tokens, and `SeqPairNode` – for sequence pair productions. The logic for choosing the appropriate superclass is implemented in Bison II.

6.4.3 Slots and Methods

Both phyla and operator classes may contain methods and slots with given names and signatures:

```
phylum expr {
    public abstract method TypeInfo type_check(Context ctx);
    ...
}

operator Assignment extends expr, ParentNode {
    public  virtual method TypeInfo type_check(Context ctx) { ... }
    private slot TypeInfo type;
    ...
}
```

Methods get translated to C++ method definitions; slots are translated to fields. As a result, methods may be specified as `public`, `private`, or `protected`, as well as have other C++ method qualifiers such as `virtual`, `const`, and `static`. A method may also be declared as `abstract` (equivalent to C++ "pure virtual"), indicating that it has no implementation. The method body is always given in-line with the definition; however, ASTDef does not generate "in-line" method declarations in the C++ sense unless a method is given the `inline` qualifier. Slots may be qualified as `public`, `private`, or `protected`, as well as `static` and `const`. The meaning of these qualifiers is equivalent to their C++ counterparts.

Every operator and phylum may be defined multiple times in different ASTDef files. Such files must be processed together and the effect of having multiple definitions is to combine all the features of every definition into the same class. This arrangement permits factoring of different implementation aspects of the AST node definitions into separate files. This is important because part of a definition may be automatically generated, such as the grammar-specific information output by Bison II, whereas other parts may be hand-coded. For example, consider the following two node definitions:

```
operator MethodDecl extends feature, ParentNode {
    public inline const method OBJECTID get_name() { ... }
    ...
}

operator MethodDecl extends feature {
    public virtual method void check_decl(Context ctx) { ... }
    ...
}
```

The net effect of processing these by the ASTDef tool is equivalent to specifying all features and all superclasses in a single definition:

```
operator MethodDecl extends feature, ParentNode {
    public inline const method OBJECTID get_name() { ... }
    public virtual method void check_decl(Context ctx) { ... }
    ...
}
```

6.4.4 Attributes

Attributes represent one of the more advanced features of the ASTDef language and are used to implement the system of named node attributes described in Section 5.2.2. Both phyla and operators may contain attributes. Much like slots and methods, the attributes are inherited by subclasses. The attributes are always "public", that is, have no C++ access control restrictions.

Consider the following example of an attribute definition:

```
unversioned attribute arity {
    document as { "arity of this node" }
    value is    { info->arity }
}
```

The specification of an attribute consists of several parts. Firstly, an attribute must be qualified as `versioned` or `unversioned`. The meaning of these qualifiers is explained in Section 5.2.2. The body of an attribute definition consists of keyword-value pairs that define the implementation of that attribute. A keyword may be optionally followed by an appropriate preposition or a verb to improve the readability of the definition. The following is a list of the allowable keywords:

- `value` (or `value is`) defines a C++ expression that specifies how to compute the value for the attribute.
- `document` (or `document as`) specifies the documentation string for the attribute.
- `synchronized` (or `synchronized by`) gives the name of the analysis that synchronizes the attribute's value; such attributes are considered synchronizable (see Section 5.2.2).
- `set` (or `set with`) defines a C++ expression whose side-effect is to set the value of the attribute. Such an attribute is treated as mutable by the client application.
- `accessible` (or `accessible through`) indicates that the attribute may be accessed through an alternative document interface, such as DOM (see Section 5.6). This keyword may be used multiple times for different interfaces.

The following is a more complete example of an attribute definition:

```
versioned attribute error_message {
    document as { "error message at this operator" }
    value is    { get_error_message() }
    set with    { set_error_message(value) }
    synchronized by SYNTAX
    accessible through DOM
}
```

6.4.5 C++ Declarations

In addition to slots, methods, and attributes ASTDef permits other C++ member declarations such classes, structs, typedefs, etc. These get output verbatim into the resulting C++ implementation:

```
phylum program {
    public struct TypeInfo { ... }

    typedef Scope<PooledString,
                  const TypeInfo,
                  hash<PooledString>,
                  equal_to<PooledString> > TypeTable;
    ...
}
```

7 Programming Language Bindings

The implementation language of the HARMONIA framework is C++. However, C++ is frequently not the most appropriate language for building interactive applications. For this reason, the HARMONIA framework provides bindings for a number of popular programming languages. The languages for which bindings currently exist include Tcl, Java, C, and XEmacs Lisp. (The latter lets us combine the HARMONIA framework with the popular editor in a very natural fashion: HARMONIA structure-based services appear as extensions to the text-editing facilities that exist within XEmacs.)

The bindings for each language are designed in a very similar manner: a shallow *glue* layer implements a mapping from the data types in the target programming language to the data types in the HARMONIA framework. This mapping is not one-to-one. The complex class hierarchy of the HARMONIA framework is simplified to the few core types. For example, the hierarchy of node classes is collapsed to a single node data type in almost all cases (the Java interfaces do make some distinction, but not nearly as much as the language kernel). The data are not replicated across language boundaries: objects at the target language level provide only shallow handles (essentially pointers) to the HARMONIA data structure.

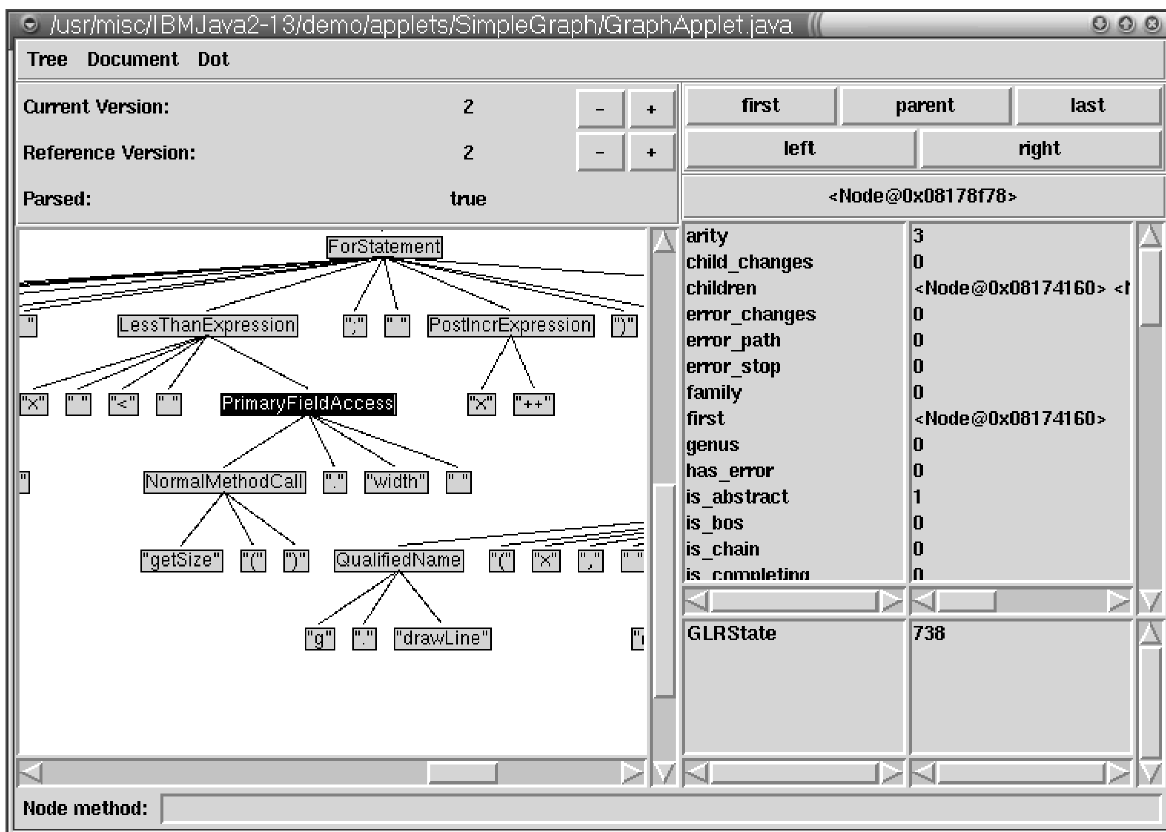
This design is mainly dictated by the needs of applications under development and may be extended in the future. At the time of this writing, three applications had been built using the integration mechanism described here. One is the batch front end to the HARMONIA analysis services that processes input files in batch mode and prints analysis errors to the console. This application is written in Tcl and consists of fewer than 300 lines of code. Another Tcl application (1400 lines) is an interactive viewer/editor for viewing HARMONIA syntax trees (Figure 20a). Finally, using XEmacs Lisp, an undergraduate student working with the HARMONIA project has built a language-sensitive source code editor as a "mode" in the XEmacs text editor (Figure 20b).

8 Conclusions

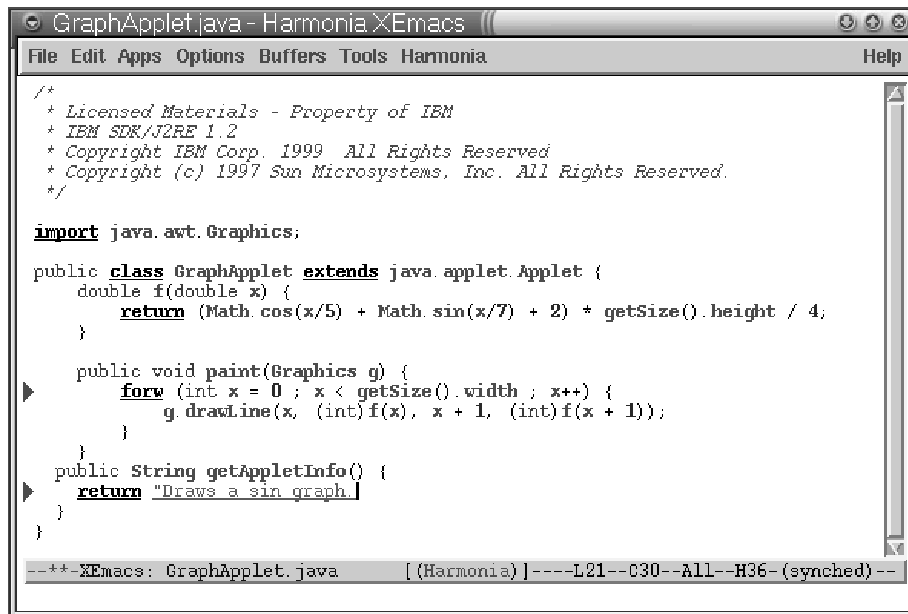
This report presents the architecture of HARMONIA– a framework for constructing interactive language-based programming tools. The primary purpose of the HARMONIA framework is to provide an open architecture for building interactive tools and services for a variety of languages.

Although parts of the HARMONIA infrastructure were derived from the ENSEMBLE system, the HARMONIA framework represents a significant step toward an extensible and open architecture. Our contributions include:

- A significantly re-engineered language analysis kernel that provides cleaner and more complete set of APIs for a variety of client applications.
- An enhanced language definition mechanism that permits the use of a GLR parser and provides higher-level grammatical abstractions than those available in ENSEMBLE.
- A framework for static program analysis, including specialized support for tree-based analysis and transformations.



(a)



(b)

Figure 20: Harmonia in action: (a) application for displaying syntax trees and (b) Harmonia-enhanced programming mode for the XEmacs editor.

- A design and implementation (ongoing) for the program data exchange format based on XML [62] and DOM [53].
- A set of significantly enhanced tools for compiling language definitions, including the ASTDef processor which is new to the Harmonia framework.
- A general extension mechanism through use of language bindings to programming languages other than C++.

9 Future Work

The Harmonia framework is an ongoing project. While the basic architecture is substantially complete, many issues remain unaddressed in the current implementation.

- **Absence of declarative mechanism for specification of static semantic analyses.** The semantic analysis infrastructure presented here facilitates building *ad hoc* analyzers without any formal specification. Our long-term goal is to provide a mechanism for generating such analyzers from a declarative specification.
- **A scripting language for creating new analyzers.** There are occasions on which developers need to create their own analyzers and transformations. One motive might be the need for incrementality, so nicely supported by the Harmonia framework. Another reason, is that it might be necessary to provide a fairly simple analysis or a transformation (for example, a style-checker) for which an external tool would be an overkill. It has been repeatedly noted that for the task of building program analyses and transformations it is helpful to have a small domain-oriented language [23]. Consequently, one of our objectives is to design and implement such a language within the Harmonia framework.
- **Aggregation of versioning.** One problem with using self-versioned documents is that fine-grained versioning captures too much information to preserve about the intermediate states of the document.[13] While this information is extremely useful during short term editing (for example, for providing a convenient undo/redo mechanism or for facilitating history sensitive parse error recovery), it is coarser grained versioning (for example, changes to a subroutine) that are important for development histories or configuration control. Consequently, mechanisms are needed to compose the fine-grained version sequences into fewer persistent document versions. The composition process must be as natural to the user as possible, while retaining a sufficient level of document version information to support history-based services.
- **Ambiguity resolution.** The syntax and semantics of most programming languages are designed to be unambiguous. For ease of analysis, formal syntactic specifications for most programming languages are given by unambiguous context-free grammars. Some languages, however, are *syntactically* ambiguous, but *semantically* unambiguous. The most notorious example is C++. A C++ parser cannot tell whether `a(x)` is a variable declaration (because `a` is a type name) or a function call (because `a` is a function name) without "knowing" the semantic context. However, Harmonia provides no mechanism for semantic ambiguity filtering. It is possible to hand-code an *ad hoc* filter using the semantic analysis infrastructure, but this is not a very satisfying solution.

These and other limitations will be addressed as the Harmonia framework is further developed.

[13] For normal text-based editing a change to each language token creates a new version of the document, but in some instances, keystroke-level change recording is enabled.

References

[1] Rolfe Bahlke and Gregor Snelting. The PSG system: From formal language definition to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, October 1986.

[2] Robert A. Ballance. Syntactic and semantic checking in language-based editing systems. Technical Report CSD-89-548, University of California, Berkeley, December 1989.

[3] Robert A. Ballance, Jacob Butcher, and Susan L. Graham. Grammatical abstraction and incremental syntax analysis in a language-based editor. *SIGPLAN Notices*, 23(7):185–198, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[4] Robert A. Ballance and Susan L. Graham. Incremental consistency maintenance for interactive applications. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 895–909, Paris, France, 1991. The MIT Press.

[5] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The Pan language-based editing system. *ACM Transactions on Software Engineering and Methodology*, 1(1):95–127, January 1992.

[6] Robert A. Ballance, Michael L. Van De Vanter, and Susan L. Graham. The architecture of Pan I. Technical Report CSD-88-409, University of California, Berkeley, September 1986.

[7] Andrew Begel. Spoken language support for software development. Qualifying examination proposal, December 2000.

[8] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, November 1988.

[9] Marat Boshernitsan. Interactive program transformations. Qualification examination proposal. In preparation.

[10] Marat Boshernitsan. Harmonia architecture manual. Technical report, University of California, Berkeley, 2001. In preparation.

[11] Jacob Butcher. Ladle. Technical Report CSD-89-519, University of California, Berkeley, November 1989.

[12] Pehong Chen, John Coker, Michael A. Harrison, Jeffrey McCarrell, and Steven Procter. The VorTeX document preparation environment. In J. Désarménien, editor, *Second European Conference on TEX for Scientific Documentation*, pages 45–54, Strasbourg, France, June 1986.

[13] Robert Corbett. Bison release 1.24, 1992.

[14] Brian M. Dennis. ExL: The Ensemble extension language. Master's thesis, Computer Science Division—EECS, University of California, Berkeley, Berkeley, CA 94720, May 1994.

[15] Brian M. Dennis, Roy Goldman, Susan L. Graham, Michael A. Harrison, William Maddox, Vance Maverick, Ethan V. Munson, and Tim A. Wagner. A document architecture for integrated software development, 1995. Unpublished.

[16] JavaCC Developers. JavaCC home page. `http://www.webgain.com/products/metamata/java_doc.html`.

[17] V. Donzeau-Gouge, G. Huet, and G. Kahn. Programming environments based on structured editors: the MENTOR experience. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*. McGraw-Hill, 1984.

[18] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[19] Frank J. Dudinsky, Richard C. Holt, and Safwat G. Zaky. SRE: A syntax recognizing editor. *Software Practice and Experience*, 15(5):489–497, May 1985.

[20] Juergen Ebert, Bernt Kullbach, and Andreas Winter. GraX—an interchange format for reengineering tools. In *Proc. of 6th Working Conference on Reverse Engineering*, pages 89–99. IEEE Computer Society Press, 1999.

[21] Adele Goldberg. Programmer as reader. *IEEE Software*, 4(5):62–70, September 1987.

[22] Susan L. Graham. Language and document support in software development environments. In *Proceedings of the DARPA '92 Software Technology Conference*, Los Angeles, April 1992.

[23] W. G. Griswold and D. C. Atkinson. Managing the design trade-offs for a program understanding and transformation tool. *Journal of Systems and Software*, 30(1–2):99–116, July–August 1995.

[24] Richard C. Holt, Andreas Winter, and Andy Schürr. GXL: Towards a Standard Exchange Format. Fachberichte Informatik 1–2000, Universität Koblenz-Landau, 2000.

[25] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pederson, and Kristen Nygaard. An algebra for program fragments. *ACM SIGPLAN Notices*, 20(7):161–170, July 1985.

[26] Bernard Lang. On the usefulness of syntax directed editors. In Tor M. Didriksen Reidar Conradi and Dag H. Wanvik, editors, *Proceedings of the International Workshop on Advanced Programming Environments*, volume 244 of *LNCS*, pages 47–51, Trondheim, Norway, June 1986. Springer.

[27] Sébastien Lapierre, Bruno Laguë, and Charles Leduc. Datrix source code model and its interchange format: Lessons learned and considerations for future work. Limerick, Ireland, 2000.

[28] William Maddox. *Incremental Static Semantic Analysis*. PhD thesis, University of California, Berkeley, January 14, 1998.

[29] Vance Maverick. *Presentation by Tree Transformation*. PhD thesis, Computer Science Division—EECS, University of California, Berkeley, Berkeley, CA 94720, January 14, 1998.

[30] J. D. Morgenthaler. *Static Analysis for a Software Transformation Tool*. PhD thesis, University of California, San Diego, Department of Computer Science & Engineering, August 1997. Technical Report CS97-552.

[31] Ethan V. Munson. *Proteus: An Adaptable Presentation System for a Software Development and Multimedia Document Environment*. PhD thesis, Computer Science Division—EECS, University of California, Berkeley, Berkeley, CA 94720, September 1994.

[32] Lisa Rubin Neal. Cognition-sensitive design and user modeling for syntax-directed editors. In *Proceedings of ACM CHI+GI'87 Conference on Human Factors in Computing Systems and Graphics Interface*, Adaptive Interfaces, pages 99–102, 1987.

[33] Robert E. Noonan. An algorithm for generating abstract syntax trees. *Computer Languages*, 10(3-4):225–236, 1985.

[34] D. Notkin. *Interactive Structure-Oriented Computing*. PhD thesis, Computer Science Department, Carnegie-Mellon University, February 1984. Available as technical report CMU-CS-84-103.

[35] D. Notkin, R. J. Ellison, B. J. Staudt, G. E. Kaiser, E. Kant, A. N. Habermann, V. Ambriola, and C. Montangero. Special issue on the GANDALF project. *Journal of Systems and Software*, 5(2), May 1985.

[36] Rahman Nozohoor-Farshi. Glr parsing for ϵ-grammars. In Masaru Tomita, editor, *Generalized LR parsing*, Norwell, MA, 1991. Kluwer.

[37] John K. Ousterhout. Tcl: An embeddable command language. Technical Report CSD-89-541, University of California, Berkeley, 1989.

[38] Vern Paxson. Flex 2.5.2 manual, November 1995. Free Software Foundation.

[39] Jan Rekers. *Parser Generation for Interactive Environments*. Ph.d. dissertation, University of Amsterdam, 1992.

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

[41] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, Heidelberg, Berlin, 1989.

[42] D. J. Rosenkrantz and H. B. Hunt. Efficient algorithms for automatic construction and compactification of parsing grammars. *ACM Transactions on Programming Languages and Systems*, 9(4):543–566, October 1987.

[43] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Refrence Manual*. Object Technology Series. Addison Wesley Longman, Reading, Mass., 1999.

[44] Gerd Szwillus and Lisa Neal. *Structure-Based Editors and Environments*. Academic Press, New York, NY, USA, 1996.

[45] Masaru Tomita. *An Efficient Context-free Parsing Algorithm for Natural Languages and Its Applications*. Ph.d. dissertation, Computer Science Department—Carnegie Mellon University, Pittsburgh , PA , USA, May 1985.

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

[47] Masaru Tomita, editor. *Generalized LR parsing*. Kluwer, Norwell, MA, 1991.

[48] Michael L. Van De Vanter. User interaction in language-based editing systems. Technical Report CSD-93-726, University of California, Berkeley, April 1993.

[49] Michael L. Van De Vanter. Practical language-based editing for software engineers. *Lecture Notes in Computer Science*, 896:251–267, 1995.

[50] Michael L. Van De Vanter and Marat Boshernitsan. Displaying and editing source code in software engineering environments. In *Proceedings of Second International Symposium on Constructing Software Engineering Tools (CoSET'2000)*, pages 39–48, Limerick, Ireland, 2000.

[51] Guido van Rossum. Python reference manual. Report CS-R9525, Centrum voor Wiskunde en Informatica, P. O. Box 4079, 1009 AB Amsterdam, The Netherlands, April 1995.

[52] Michael L. Van De Vanter, Susan L. Graham, and Robert A. Ballance. Coherent user interfaces for language-based editing systems. *International Journal of Man-Machine Studies*, 37(4):431–466, 1992.

[53] W3C. *Document Object Model (DOM) Level 2 Specification*, May 2000. `http://www.w3.org/TR/DOM-Level-2/`.

[54] Earl Wagner, Marat Boshernitsan, and Susan L. Graham. Generalized LR parsing for interactive applications. Technical report, University of California, Berkeley, 2001. In preparation.

[55] Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments*. PhD thesis, March 11, 1998.

[56] Tim A. Wagner and Susan L. Graham. Efficient self-versioning documents. In *Proceedings of COMPCON '97*, San Jose, CA, 1997.

[57] Tim A. Wagner and Susan L. Graham. General incremental lexical analysis, 1997. Unpublished.

[58] Tim A. Wagner and Susan 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.

[59] Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems*, 20(5):980–1013, September 1998.

[60] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly Associates, Inc., Sebastopol, CA, 1990.

[61] David S. Wile. Abstract syntax from concrete syntax. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 472–480, Berlin - Heidelberg - New York, May 1997. Springer.

[62] Extensible markup language (XML). http://www.w3.org/XML/.

[63] XSL tranformations. `http://www.w3.org/TR/xslt`.