A Comparison of Designs for Extensible and Extension-Oriented Compilers

by

Austin T. Clements

S.B. in Computer Science and Engineering
Massachusetts Institute of Technology (2006)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2008

Author .....
Department of Electrical Engineering and Computer Science
February 4, 2008

Certified by .....
M. Frans Kaashoek
Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Comparison of Designs for
Extensible and Extension-Oriented Compilers

by

Austin T. Clements

Submitted to the Department of Electrical Engineering and Computer Science
on February 4, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Today's system programmers go to great lengths to extend the languages in which they program. For instance, system-specific compilers find errors in Linux and other systems, and add support for specialized control flow to Qt and event-based programs. These compilers are difficult to build and cannot always understand each other's language changes. However, they can greatly improve code understandability and correctness, advantages that should be accessible to all programmers.

This thesis considers four extensible and extension-oriented compilers: CIL, Polyglot, xtc, and Xoc. These four compilers represent four distinctly different approaches to the problem of bridging the gap between language design and system implementation. Taking an extension author's point of view, this thesis compares the design of each compiler's extension interface in terms of extension structure, syntactic analysis, semantic analysis, and rewriting.

To perform the comparison, this thesis uses three extensions implemented variously in the four compilers: a bitwise rotation operator, function expressions, and lock checking. These extensions are designed to span a representative space of analysis and rewriting needs.

Based on this comparison, this thesis identifies the following implications of the design decisions of each extension interface: the expressiveness, understandability, and correctness of extension implementations can benefit from domain specific languages and language features tailored to the extension interface; compiler-managed scheduling trades loss of control for automatic extension composability; unifying internal and external program representation improves ease of use and extension composability, but gives up potentially useful control over the internal representation; concrete syntax patterns provide a natural interface to internal program representation, but must be more powerful than simple tree matching to be practical; grammars, types, and syntax interfaces have a natural correspondence; and accounting for semantic information in the final output enables hygienic rewriting, which can simplify extensions.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor

Acknowledgments

I owe a great deal of gratitude to my thesis advisor, Frans Kaashoek, for being a constant source of ideas, encouragement, focus, and inspiration.

Russ Cox, without whom the Xoc project simply would not exist, provided the insight and talent necessary to get us to where we are today. I would also like to thank Eddie Kohler and Tom Bergan for their unwavering skepticism, which pushed the boundaries of our thinking about extensible compilation. Todd Millstein provided much useful advice and feedback on the design of Xoc. Robert Grimm offered advice on using xtc and provided sanity checking for our xtc extensions.

Thank you to my Mom and my Dad for their constant encouragement and enthusiasm and for teaching me to always strive for my best. Thank you to my friends for being a unending source of support and head nodding when I needed it most. Finally, I would like to thank Emily Grosse for her love, her tireless attention to my happiness, and for ensuring that I slept and ate on a regular basis.

Portions of this thesis are adapted from text originally published in

> Russ Cox, Tom Bergan, Austin T. Clements, Frans Kaashoek, and Eddie Kohler. Xoc, an extension-oriented compiler for systems programming. In *Proceedings of the 13th Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

This project was partially supported by the National Science Foundation under Grant Nos. 0430425 and 0427202, and by Nokia Research Center Cambridge. This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Contents

List of Figures

Chapter 1

Introduction

Programmers have long found it useful to make domain-specific changes to general-purpose programming languages. Such linguistic extension allows programmers to create custom languages tailored to the direct expression of specific tasks, without sacrificing the generality and library support of a standard language. Unfortunately, few common languages natively support general linguistic extension, forcing programmers to match their needs to the language, instead of matching the language to their needs. The desire for domain-specific support has sparked numerous efforts to manually layer specific language extensions on top of existing languages. Recent examples of such extended general-purpose languages include Sparse [34], which augments C's type system with domain-specific features tailored to kernel checking; Tame [21], which extends C++ with constructs for event-driven programming; and Mace [18], which adds domain-specific features for building distributed systems to C++.

The most common approach to language extension is to construct a monolithic preprocessor: a brand new compiler front end that accepts the base language plus domain-specific changes, compiles the domain-specific constructs into base language constructs, writes out the equivalent base program, and invokes the original compiler or interpreter. This approach has two serious problems. First, extensions are heavyweight, requiring either a daunting implementation effort in order to understand the base language constructs or a great deal of cleverness to avoid having to understand the base language. In particular, this effort must be repeated for each different language extension. Second, extensions are isolated; they typically cannot be used together because they cannot understand each other's specialized constructs.

Recent extensible compiler research recognizes the need for general support for linguistic extension and makes an effort to focus the per-preprocessor cost of supporting the base language into a single one time effort. Extensible compilers are typically toolkit-oriented, providing a compiler for some base language whose code is, by design, open to additions and consists of a collection of compiler components that are designed to be reused, extended, and ultimately combined to build new compilers. Starting with such an extensible compiler, an extension author needs only write the code necessary to process the domain-specific changes. These compilers are typically structured as a collection of classes that provide the base language front end, which an extension author subclasses some or all of to produce a *new* compiler that accepts the base language with the domain-specific changes. This approach improves upon the implementation effort required for a preprocessor but still creates compilers that are not *composable*: changes implemented by two different extensions can't be used in the same program since the different extensions still produce different

compilers. Recent extensible compilers have some support for extension composition [29, 36], but still require that extension authors explicitly assemble each desired composition.

Current extensible compilers are targeted at *monolithic* extensibility, where an extension derives a new *language* from an existing one and is itself a brand new compiler. In this philosophy, an extension is expected to implement complicated and holistic changes in order to produce a language that differs significantly enough from the base language to represent a worthwhile new language. The design of these compilers naturally follows from their philosophy, which leads to high fixed costs for the implementation of new extensions (which is acceptable when extensions are expected to be large), a lack of automatic composability (which is unnecessary when extensions are expected to implement holistically different languages), and a great deal of exposure of the compiler's internals and operational model (which is natural when extensions are implemented by deriving one compiler from another). While this philosophy is well-suited for language design and experimentation, it is insufficiently agile for user-oriented, lightweight extensions.

In an *extension-oriented compiler*, new features are implemented by many small extensions that are loaded on a per-source-file basis, much as content-specific plugins are loaded by web browsers and other software. Whereas extensible compilers target extensions that define new languages, an extension-oriented compiler targets extensions that define new language *features* that can be mixed and matched on the fly within the compiler. As in current extensible compilers, this approach starts with a compiler for the base language, but extensions do not create whole new compilers. Instead, the base compiler loads the extensions dynamically for each source file.

This thesis explores the designs of three established extensible compilers: CIL, a compiler supporting analysis extensions [27]; Polyglot, an extensible Java compiler framework [28]; and xtc, an extensible C and Java compiler framework [12, 14]; plus a new extension-oriented compiler, Xoc, developed by our group to explore the extension interfaces necessary to achieve lightweight extensibility [6]. This thesis supports this exploration using three extensions, which span a representative space of analysis and rewriting problems. We have implemented these three extensions across the four compilers in order to identify the key design points of extensible and extension-oriented compilers and manifest the implications of each compiler's design decisions.

1.1 Compilers Overview

CIL, Polyglot, xtc, and Xoc all rely on syntactic substitution models, in which both an extension's input and output are syntactic representations of the program being compiled. Extensions that introduce new syntax are responsible for rewriting that syntax into the target language, reducing input programs written in the extended language to semantically equivalent output programs written in a fixed output language. The compilers themselves are all implemented as source-to-source translators, following this same basic principle.

Extensions for all four compilers are written in general purpose languages. However, one of the observations driving work on extensible compilers is that the ideal language for a given task is a language constructed specifically for that task. All of the compilers except CIL expose extension interfaces that include domain-specific language features tailored for extension authors, though the extent of these features varies greatly between the compilers.

CIL. Unlike the other three compilers, CIL is not intended for syntactic extensions. Instead, CIL specializes in the analysis and rewriting of a *fixed* base language: regular C. CIL tries to make this as approachable as possible by reducing C to a limited set of orthogonal constructs with clean semantics, eliminating the complexities and dark corners of the C language. The CIL core drives the compilation process, first parsing and simplifying the input program down into its simplified abstract syntax, then invoking extensions on this internal representation. This approach makes extensions relatively easy to write because they do not have to be concerned with the overall compilation process and the simplified representation vastly reduces the number of cases and easily overlooked details that an extension has to cope with. Because of CIL's fixed representation model, extensions compose naturally since extensions cannot introduce new constructs and never encounter unknown constructs. CIL exposes an OCaml-based extension interface, with no additional language support for writing extensions; however, OCaml's built-in pattern matching abilities are well suited for manipulating abstract syntax trees.

Polyglot. Of the four compilers, Polyglot most closely resembles a traditional compiler, adopting time tested approaches to lexical and syntactic analysis, as well as pass-based scheduling and object-oriented interfaces to the internal representation. Polyglot is essentially a traditional Java compiler designed with open interfaces that make extensive use of object-oriented programming principles and design patterns such as factories, delegates, and visitors to make them amenable to change. An extension takes Polyglot as a base and uses its flexible interfaces to replace or augment compiler components and yield a new compiler. Because each extension represents a new compiler, composing extensions requires an extension author to manually combine the extensions' drivers, syntax, and inheritance hierarchy, building a new extension that represents their combination.

Polyglot extensions are written in regular Java. Reflecting traditional compiler practices, Polyglot exposes the interface to its parser using a domain-specific language, PPG, which is similar to yacc [17], but tailored for extensible grammars with special support for adding and remove rules from a base grammar. The Polyglot parser generator compiles these parser specifications separately into Java code, which is then linked against the rest of the extension.

xtc. The xtc toolkit is a compiler framework, consisting of generic components for parsing and internal representation, as well as specific components for analyzing Java and C. Each extension is a complete program, fully responsible for driving the compilation process, which combines and extends this library of components to construct the desired compiler. This whole-program approach permits a great deal of flexibility, but incurs a high initial extension implementation cost and precludes any form of automatic extension composition.

Similar to Polyglot, xtc extensions are written in regular Java. xtc provides an expressive parser construction language, *Rats!*, which supports parameterized and conditional modular grammars. xtc's interface provides additional domain-specific language support for concrete syntax, which allows syntax fragments to be written in the syntax of the language being compiled. xtc compiles the parser specifications and the concrete syntax files to Java code, which is linked against the extension.

Xoc. While extensible compilers go to great lengths to expose compiler internals, Xoc goes to great lengths to hide them. The Xoc core drives the process of compilation. It loads extensions specified on the command line, parses the input file, performs analysis and

rewriting, and produces the output program. Extensions extend this process via on-demand (lazy) computation, which hides the details of scheduling and allows the Xoc core to manage dependencies between computations. Extensions use syntax patterns [4] to manipulate and rewrite Xoc's internal program representation using the concrete syntax of the programming language being compiled. This hides the details of both parsing and internal representation.

Xoc's interface is intended to be easy to use correctly and hard to use incorrectly. In particular, Xoc makes extensive use of static checking to identify errors in extensions as early as possible, and attempts to have reasonable default behavior whenever feasible in order to allow extensions to focus on the feature being added to the language instead of details like internal representation, type syntax, or variable naming.

While most compilers, including Polyglot and xtc, have some elements of domain-specific languages in their extension interfaces, Xoc focuses on exploring the application of language design to extension interfaces and its influence on the capabilities of extensions and extension-oriented compilers. Both Xoc and its extensions are written in Zeta, a metalanguage that was developed in parallel with Xoc. Zeta has served as a dynamic platform for experimenting with extensible compiler structure and interfaces. This approach enables many of Xoc's unique interface features.

1.2 Extensions Overview

This thesis uses three extensions to compare the compilers, each posing a unique set of challenges for each compiler's interface. Together, these three extensions span the space of rewriting and analysis. The bitwise rotation extension covered in chapter 4 is a simple, macro-like rewriting extension that explores the fixed costs of each compiler's interfaces and, despite its simplicity, requires good support for manipulations of symbols and types. The context analysis extension covered in chapter 5 implements a complex program analysis with minimal rewriting needs that pushes each compiler's performance and general computing abilities and poses unique challenges for extension composability. Finally, the function expression extension covered in chapter 6 is a complex rewriting extension with some simple analysis that challenges interfaces for manipulation and construction of syntax, symbols, and types.

1.3 Contributions

This thesis makes the following contributions to the study of extensible compilation:

- The designs and implementations of three extensions, intended to characterize the needs of different types of extensions and stress different dimensions of extensibility.
- A comparison of the implications of the design choices made by three established extensible compilers and one new extension-oriented compiler, supported experimentally by implementations of the three extensions.
- The identification of the following implications of the design decisions of each compiler: the expressiveness, understandability, and correctness of extension implementations can benefit from domain specific languages and language features tailored to the extension interface; compiler-managed scheduling trades loss of control for automatic extension composability; unifying internal and external program representation improves ease

of use and extension composability, but gives up potentially useful control over the internal representation; concrete syntax patterns provide a natural interface to internal program representation, but must be more powerful than simple tree matching to be practical; grammars, types, and syntax interfaces have a natural correspondence; and accounting for semantic information in the final output enables hygienic rewriting, which can simplify extensions.

1.4 Outline

The remainder of this thesis covers each compiler and extension in detail, examining their designs and implementations, the relations between their designs, and the implications of each compiler's approach to extensibility. Chapter 2 begins by exploring work related to the modern development of extensible and extension-oriented compilers, including past approaches to their underlying goals and work from other areas that has inspired their current designs. Chapter 3 introduces a set of key design points for extensible and extension-oriented compilers and briefly positions each compiler within the design space. Chapters 4–6 cover the extensions used to explore and exercise all four compilers, including the unique challenges each extension poses for an extension interface, the details of their implementations, and the implications of each compiler's design choices on the implementations. Chapter 7 examines the cross-cutting concerns of ease of use and extension composability. Finally, chapter 8 touches on some open problems and future directions for extensible and extension-oriented compilers and concludes.

Chapter 2

Related Work

There has been a desire for linguistic extension for nearly as long as there have been programming languages higher level than assembly. The study of extensible languages saw both its rise and fall in the 1960's and 1970's [26]. With the exception of syntax macros, few of the results of the field ever saw widespread use. Recently, there has been a resurgence of interest in customizing programming languages to match programmer- and project-specific needs, giving rise to the exploration of extensible compilers. This thesis focuses in depth on a few characteristic and new extensible and extension-oriented compilers united by a basis in general models of computation. However, there is also related work in alternate models of computation which have both influenced the designs of the compilers covered herein and given rise to entire compilers built solely on these models.

2.1 Extensible Languages

Early work on language extension focused primarily on *extensible languages*, where the compiler is fixed, but the language definition itself incorporates mechanisms to make new syntax or semantics available to the programmer. Extensible languages and extensible compilers are different approaches with a similar goal.

2.1.1 Macros

Macros were one of the earliest approaches to extensible languages [25], and are still the most popular method for extending a language, mainly due to the power demonstrated by Lisp's macros [11, 13, 32]. Lisp was also the first to introduce syntax patterns for constructing new fragments of syntax—the Lisp term is *quasiquotation*—to make code generation easier to write and understand. Programmers have ported Lisp's macros into other languages [e.g., 2, 3, 41].

While macros have a host of excellent properties—they are quick and easy to write, work at the level of source code without requiring knowledge of the underlying compiler, and can build upon and compose with each other—they are nevertheless relegated to purely syntactic transformations, making them excellent for adding syntactic short-hands but not useful for semantic changes or non-local transformations. Since macros can be invoked only explicitly and their access to program syntax is limited, they cannot perform arbitrary analyses or transformations. Furthermore, macros are essentially syntax sugar: they don't let the programmer access information or language capabilities beyond what they could

simply have written out by hand, meaning that they cannot access compiler internals such as type and symbol information.

However, the key idea behind macros—using syntactic substitution to reduce an extended language down to a fixed syntactic core—pervades modern extensible and extension-oriented compilers. A key difference is that, where macros inhabit the program text they are part of and thus exhibit the same limitations as any other program text, compiler extensions inhabit the compiler, and can be granted unfettered access to any information and capabilities available to the compiler itself.

2.2 Alternate Computation Models

Most compilers are written using general-purpose programming languages like C, Java, or ML, but some projects rely exclusively on domain-specific programming models. Domain-specific models can simplify common idioms or constructs and can provide stronger static guarantees but, if not coupled with a general purpose language, they can make some tasks considerably more difficult and pose a significant barrier to entry for most programmers. Xoc borrows from some of these domain-specific models, taking advantage of the flexibility afforded by its custom meta-language, but never forgoes an overall general purpose model. On the other hand, providing general-purpose constructs precludes static detection of errors like attribute circularity or coverage of new syntax.

2.2.1 Term Rewriting Systems

Term rewriting systems share the same key idea of syntactic substitution central to macros. Like macros, their extensibility is based solely on this idea. However, unlike macros, in a term rewriting system, syntactic substitution is not simply a feature that can be invoked when desired; instead, the term rewriting rules are given full reign on the entire syntax tree. Term rewriting is Turing-complete, so theoretically such systems could be used to build complete compilers; however, they tend to be better suited to more specialized tasks such as peephole optimization.

For example, Stratego [39], a term rewriting system built atop the ASF+SDF syntax framework [35], has been used to construct program optimizers [40], lambda calculus interpreters [7], and program transformers for adding concrete syntax support to existing languages [38].

Stratego's support for using concrete syntax embedded in the meta-language to construct syntax fragments [4] resembles that of xtc and Xoc. Both Stratego and Xoc take advantage of custom meta-languages built to permit the intermingling of concrete syntax fragments with other code. Xoc takes this embedding a step further, taking advantage of its meta-language's type system to statically ensure that syntax fragments are always well-formed and Xoc's knowledge of the semantics of the language being manipulated to enable hygienic rewriting and the natural manipulation of type syntax.

2.2.2 Attribute Grammars

Attribute grammars [19] are a formalism for decorating parse trees with semantic information computed from the tree. Based on high-level declarative definitions of how to compute semantic values for syntax nodes based on the semantic values attributed to related

nodes, attribute grammar systems are capable of automatically producing efficient plans for computing these values.

Attribute grammars have been used as a formal basis for turning many language specifications into implementations [30]. Silver [36, 37], a general attribute grammar specification system, is targeted specifically at building extensible compilers. Silver is implemented in itself as an extensible compiler in order to provide a means to overcome the expressive limitations of the core attribute grammar formalism. Unfortunately, while this approach permits the addition of general purpose language features, the overall computation model of attribute grammars limits the ability of extensions to perform operations such as non-local manipulations of the syntax tree.

Inspired in part by attribute grammars, Xoc eliminates passes and pass scheduling, representing analyses as lazy computations attributed to syntax tree nodes. However, Xoc's attributes do not replace, but instead augment, Xoc's computation model, leaving the expressiveness of a general computation model intact. On the other hand, because of this mixed paradigm approach, Xoc loses many of the static capabilities of the formalism, such as the ability to construct efficient traversal orders for computing attribute values.

2.2.3 Provable Extensions

Some recent work has focused on specialized computation models that are restricted so as to permit static checking of properties for specific classes of language extensions. For example, semantic type qualifiers [5] allow users to define typing rules for extended sets of qualifiers; the rules are automatically validated against desired runtime invariants. Other work has made progress in proving the correctness of dataflow analyses, compiler transformations, and optimizations [22, 23].

Chapter 3

Compiler Design Issues

This chapter introduces the key design points that distinguish the extension interfaces of CIL, Polyglot, xtc, and Xoc. In addition to providing a deeper view of the individual compilers, these design points provide a framework for reasoning about and comparing extension implementations. The implications of the design decisions of each compiler will be identified in the following chapters, as we go into depth on the three extensions and their corresponding implementations.

Overall extension structure is determined primarily by the interfaces for integrating extensions with core compiler components into a complete compiler and for combining their dependencies into a complete schedule. The interfaces for modifying the grammar and the internal representation influence the structure of syntactic analysis in the compilers and the extensions. Finally, the interfaces for examining and rewriting the internal representation determine the structure of semantic analysis.

3.1 Extension Integration

Polyglot and xtc are "compiler kits" in which each new extension represents a new compiler. In both, the extension itself is responsible for driving the compilation process. Because of this, mixing multiple extensions requires the manual intervention of an extension author to construct a new extension and combine the drivers of the desired extensions. Px, a version of Polyglot ported to the J& language [29] addresses composability problems that arise from the use of object-oriented inheritance, but still requires constructing a new compiler for each set of extensions.

In contrast, in CIL and Xoc, the compiler core drives the compilation process. CIL takes a set of extensions to enable on its command line, but the CIL core plus a given set of extensions still represent a new, standalone compiler because extensions are compiled directly into the CIL binary.

An extension-oriented compiler like Xoc, on the other hand, accepts plugins during compilation. It is not necessary to rebuild Xoc each time the user wants to try a new extension; it dynamically loads the requested extensions and integrates them into the compiler.

3.2 Scheduling

Many compilers are structured as a series of passes: first variable scoping, then type checking, then constant evaluation, and so on. Adding extensions in such a model requires defining how the extension's computation fits into the pass structure. Polyglot and xtc expand this pass-based approach to extensible compilation, requiring extensions to declare how they fit into the existing compiler pass structure. xtc requires extension drivers to explicitly invoke each necessary pass at the appropriate time, while Polyglot generates and executes a pass schedule based on extension-provided dependency information. Both approaches complicate extension design, since extensions must be aware of when various analyses take place, and extension composition must account for inter-extension dependencies.

CIL depends on the user to schedule passes, invoking extensions in the order specified on the command line. This is practical in CIL's model because extensions cannot change the input language, so CIL's standard passes remain the same and extension passes can simply come at the end.

Inspired in part by attribute grammars, Xoc eliminates passes and pass scheduling altogether using *lazy scheduling*, in which analyses are implemented as lazily-computed syntax node attributes whose values are computed on demand at first access and then cached. The dependencies implicit among Xoc's attributes determine the complete compiler structure: the required order naturally arises from the sequence of references to other attributes.

3.3 Extensible Grammars

Polyglot, xtc, and Xoc all provide support for changing the input grammar by allowing extensions to add grammar rules. Polyglot accepts context-free grammar rules but uses an LALR parser [1], making it possible for extensions to add valid, *unambiguous* rules to the base grammar that are nonetheless rejected by the parser. xtc solves this problem by switching formalisms, opting instead for parsing expressions grammars (PEGs) and a packrat parser [8], which permit unrestricted modifications to the grammar. PEGs replace the context-free alternation operator with an ordered-choice operator, using preference to eliminate the possibility of ambiguity.

Xoc uses context-free grammars, but, unlike Polyglot, relies on a GLR parser [33], which allows it to handle any context-free grammar, and thus arbitrary grammar additions. GLR parsing has the added benefit over packrat parsing that it detects ambiguities introduced by combinations of grammar extensions, instead of silently choosing one meaning or the other. However, since no grammar formalism can statically check context-free grammars for ambiguity [24, pp. 259–261], GLR settles for detecting ambiguity when it arises in the input. This makes GLR's ability to detect ambiguities a mixed blessing: while it provides no static guarantees on the lack of ambiguity in either composed grammars or even an individual extension's grammar, it also does not reject ambiguities unless they are actually stumbled upon by the input and, when it does so, it can precisely identify the source of the problem.

3.4 Syntax Typing

Syntax trees have implicit types: for example, a syntax tree representing a statement cannot be used where a syntax tree representing a variable name is expected. Compilers differ on whether they expose these types automatically in the implementation language, require

extension authors to explicitly define these types, or just use a single type for all syntax nodes.

CIL and Polyglot use explicitly typed syntax trees. This makes it possible for the implementation language's compiler to check that syntax trees are well-formed (where a statement node is expected, only a statement node can be used). This also gives the compiler writer more control over the internal syntax tree representation. For example, Polyglot uses Java interfaces to classify related syntax types, such as the Binary interface, which is the supertype of binary expressions, or Term, which is the supertype of AST nodes that can contribute to control flow. Such interfaces allow the implementation language to check that the required functionality of new node types has been implemented.

On the other hand, using a *single* type for every abstract syntax node provides a uniform representation that makes traversals of foreign syntax particularly easy, since the node object can expose a generic list of child syntax nodes. xtc takes this approach. In contrast, performing generic manipulations of abstract syntax that lacks a uniform representation requires a more heavyweight interface, such as the visitor pattern [9].

The type system and grammar support of Xoc's metalanguage allow it to combine many of the benefits of an explicitly typed syntax and a uniform representation. Grammar declarations implicitly define types, allowing the metalanguage to statically check that syntax manipulations are well-formed. However, a uniform representation underlies these myriad types, allowing also for generic access and traversals.

Because Xoc controls the representation of abstract syntax nodes, it trades flexibility for the loss of control over the internal representation of syntax nodes. However, the primary benefit afforded by Polyglot's requirement that extensions provide representations for abstract syntax nodes typically lies in the ability to specialize analysis methods for different node types, not in control of the representation itself. Xoc's extensible attributes largely make up for this, though in a less formal framework that precludes static coverage analysis.

3.5 Syntax Representation

Because CIL and Polyglot construct abstract syntax trees from user-written types, they require a similarly user-written translation from the concrete syntax trees produced by their parsers to the internal abstract syntax trees used by the rest of compilation. This approach clearly distinguishes the external and internal program representations and allows the internal representation to be tailored to the compiler's needs, but represents more work for an extension author, since an extension has to not only define parsing actions for newly introduced concrete syntax, but also implementations for newly introduced abstract syntax.

In contrast, both xtc and Xoc use *unified* representations, where the internal program representation can be constructed mechanically from the concrete syntax. In these, adding support for both parsing and representing new syntax requires nothing more than declaring the appropriate context-free grammar rules, from which the compiler core derives representations. In xtc, the parser simply generates instances of its generic syntax node type. In Xoc, the parser uses the syntax types generated by its metalanguage's grammar support. This approach depends on an expressive grammar formalism such as PEG or GLR, since it assumes that the concrete syntax will yield a reasonable abstract representation. In contrast, LALR's restrictiveness frequently requires grammars to be refactored in order to satisfy

the formalism, which obscures the relationship between concrete and abstract syntax (C is notoriously difficult to construct a correct LR grammar for [31]).

3.6 Syntax Patterns

Extensions for CIL, Polyglot, and xtc manipulate abstract syntax trees in terms of standard data types (variant types in OCaml and objects in Java). Extension authors must learn this abstract representation in order to manipulate programs and must translate transformations conceived in terms of concrete syntax into the appropriate operations on the abstract syntax. Even though xtc uses a unified representation, extension authors must still be aware of the representation generated from the grammar.

Rather than expose the abstract syntax tree using traditional data structures, Xoc extensions refer to the syntax tree using concrete syntax (the syntax of the programming language). For example, the *destructuring* pattern `~{(\a << \b) << \c}` matches a syntax tree consisting of two shift operators, binding the variables `a`, `b`, and `c` to their respective subtrees. Similarly, the *restructuring* expression `'{\a << (\b + \c)}` constructs a new abstract syntax tree, substituting the values of `a`, `b`, and `c` in their respective slots. Like Lisp, Xoc unifies internal and external program representation. Whereas Lisp makes programs look like an internal data structure, Xoc makes the internal data structure look like a program.

CIL, Polyglot, and xtc also provide various forms of syntax patterns, but only as secondary interfaces to their abstract syntax trees. Furthermore, because these other compilers are not based on a custom metalanguage like Xoc, their syntax patterns are less powerful and more prone to errors. CIL provides simple string-based primitives for restructuring and destructuring concrete syntax, like C's `printf` and `scanf`. Polyglot provides a similar `printf`-style restructuring syntax. xtc provides a more general mechanism in which patterns are stored in a separate file. A program called the `FactoryFactory` compiles them to Java methods that extensions can call. The CIL and Polyglot approach keeps the patterns near their use but cannot check that they are well-formed when compiling extensions. The xtc approach can check the syntax of patterns but requires that they be defined in a separate file, apart from their use.

Because support for syntax patterns is not integrated into the extension implementation languages of CIL, Polyglot, and xtc, they cannot provide the same level of integration between syntax patterns and the type system that Xoc does. Both CIL and Polyglot require explicit annotations of the syntax types of all slots and check at runtime that fragments are appropriately typed. Typed syntax is particularly important for syntax patterns: because xtc uses untyped syntax trees, even at runtime it cannot diagnose errors in which the wrong type of node is passed to a restructuring pattern (for example, using syntax for a statement where a variable name is expected), resulting in a malformed tree.

3.7 Hygienic Rewriting

Any program rewriting system must worry about unintended variable capture in expanded code: if a reference to a variable `x` is copied into a new syntax tree, it should by default continue to refer to the same variable, even if there is a different definition of `x` in scope at the new location. Avoiding unintended name capture in expansions is called *hygiene* [20].

CIL, Polyglot, and xtc require extension authors to manage hygiene. All three use only a symbol's name to identify it, so if two symbols have the same name, they are the same

symbol. CIL and xtc support creating new symbols by generating unique symbol names (akin to Lisp's gensym), so extensions can hygienically introduce new variables. Manipulating existing code fragments requires more care to prevent the meaning of names from changing or conflicting. For this, CIL and Polyglot provide "flattening," a general transformation in which all variable declarations are lifted to the beginning of their enclosing function and expressions are made side effect-free by introducing temporaries.

In accordance with Xoc's goal of being easy to use correctly, all rewriting is hygienic by default. Xoc's representation of a symbol captures that symbol's identity in addition to its name: each unique symbol in the program has a unique symbol object representing it in the compiler. As syntax fragments are rewritten, identifiers retain their associated symbol objects. Likewise, as new fragments are constructed, identifiers are assigned new symbol objects, so symbols introduced by extensions can never conflict with other symbols in the program, even if they have the same name.

Unlike CIL, Polyglot, and xtc, Xoc's final output is not simply a printed version of the syntax tree at the end of rewriting. Instead, the final step of compilation, which is typically hidden from extensions, reconstructs the syntax tree based on semantic information represented in attributes. While most of the syntax tree is left alone (statements and almost all expressions), this step gives Xoc the opportunity to produce an output program that accurately reflects *all* aspects of the internal representation by renaming variables when distinct symbols share a name and migrating and reordering declarations to ensure the validity of all name references.

Chapter 4

Bitwise Rotate Extension

The bitwise rotate extension adds support for left and right bitwise rotation operators. Although a trivial extension, this exercise explored the fixed costs of each compiler's extension interface and highlighted key differences between their approaches to extensibility. Even an extension as simple as a bitwise rotate operator presents a few critical challenges for an extensible compiler.

The extension introduces two operators: `<<<` and `>>>` (or `>>>>` in Java, to avoid conflicting with the standard unsigned right shift operator). Their behavior (and syntax) mimics the bitwise shift operators. Integer promotion is performed on the left operand (meaning that types smaller than 32 bits are padded to 32 bits) and the size of the resulting type is used to determine the rotation width. In C, like the C shift operators, the results are undefined if the value of the right operand is larger than the number of bits in the result type. In Java, like the Java shift operators, the value of the right operand is taken modulo the number of bits in the result type.

4.1 Challenges

As a baseline, consider the obvious substitution-style implementation of the rotate left operator, written here using the C preprocessor,

```
#define ROL(a, b)  (((a) << (b)) | ((a) >> (32 - (b))))
```

One immediate problem is that the macro is forced into using function call syntax, instead of defining the more natural infix `<<<` operator. Extensible grammars naturally fix this problem by allowing the definition of new expression syntax. The other immediate problem is the need to over-parenthesize everything, a style forced by the C preprocessor because it operates only on token streams. All extensible compilers operate on syntax trees and automatically parenthesize their output when appropriate, and thus avoid this problem entirely.

Three general, non-trivial issues remain, all of which can afflict any substitution-style definition of the rotate operator. Each of these issues represents not only a problem that had to be accounted for in each implementation of the bitwise rotate extension, but a challenge to each compiler's interface.

4.1.1 Double Evaluation

If the expressions passed for a or b contain side-effects, then the definition given above will cause these side-effects to occur twice. While this problem can be solved by calling on a library function instead of doing the substitution of the operator in place, this approach side-steps the issue and breaks down for more complicated extensions. An alternative solution is to hoist the side-effects out of a and b and replace a and b with side-effect-free expressions that can be safely duplicated. The downside of this approach is that it requires non-local manipulation of the syntax tree, since the hoisting operation must be done on the entire enclosing statement (not just the subexpression) because other effects in the same statement must also be hoisted. A more convenient but less portable solution is to take advantage of statement expressions. For example, with GNU C statement expressions, the above macro can be rewritten as,

```
#define ROL(a, b)    ({ int x = (a); int y = (b); \
                        (x << y) | (x >> (32 - y)); })
```

This guarantees that a and b will both be evaluated exactly once and is equivalent to performing effect hoisting in a lower level of the compiler, but without the inconvenience of having to handle hoisting in the definition of the operator itself.

4.1.2 Name Capture

This new definition exposes the issue of name capture. If, for example, the expression used for b were to contain a reference to a variable called x from an enclosing scope, then the expansion of the above definition would change the meaning of b. *Argument capture* [11] problems like this are typically solved either manually, by generating symbols that are guaranteed to be syntactically distinct from all other symbols (Lisp's gensym solution), or automatically using hygienic rewriting.

Free symbol capture, a different form of name capture, can befall a definition of the rotate operator that simply calls on a library function. If the name of that library function is itself shadowed in the scope where the expansion occurs, then the expansion will refer to the inner definition instead of the intended global definition. In this case, mere symbol generation is sufficient only if the name of the library function can also be generated. Failing this, some form of hygiene is necessary, and may require a global renaming pass in order to ensure the absence of name collisions.

4.1.3 Types

By performing a 32-bit rotation, both rotate definitions given above make over-arching assumptions about the types of their arguments. Specifically, this disagrees in spirit with the definition of the shift operators given by the C and Java standards, both of which require that the operand types be promoted and the result type be that of the promoted left operand [16, 10]. Thus, for example, 32-bit rotation should occur for any value that fits into a 32-bit int, but a 64-bit long operand should result in a 64-bit rotation. Achieving this requires not only access to the results of type analysis in order to determine the base of rotation, but also the ability to modify type analysis in order to give the appropriate result type. Furthermore, these two aspects must interact, for example, if the result type of one rotation operation influenced the operand type of another.

4.2 Implementations

The bitwise rotate extension is the most widely implemented of the extensions, with support for CIL, Polyglot, xtc, and Xoc. All of the implementations are based on in-place substitution, where the extension finds occurrences of the rotate operator in the input syntax tree and produces an output syntax tree where these are replaced with equivalent pure C code. However, the implementations differ significantly in how they approach this substitution while satisfying the challenges given above.

4.2.1 CIL

The core of the CIL extension is implemented as a visitor pass that follows the CIL simplification pass. Because CIL was not designed for syntactic extension, the simplification process itself was modified to accept rotate operators, treat them like shift operators, and pass them through to the extension. Alternatively, the simplification process could have been modified further to simplify rotate operators into shift and bit-wise or operators, but such an approach would have skirted CIL's entire extension mechanism. As part of the CIL simplification process, effects are hoisted from expressions, making them side-effect-free and thus amenable to the simple temporary-free expansion given earlier without concerns of double evaluation or name capture. The CIL abstract syntax tree includes type information, making it easy to determine the promoted type of a rotate expression and thus determine the appropriate rotation bit width.

4.2.2 Polyglot

Similar to the CIL extension, the core of the Polyglot extension is implemented as a visitor pass that searches for and replaces instances of the rotate operator in the input abstract syntax tree. Unlike CIL, because Polyglot is designed to accommodate syntax changes, the extension is self-contained, including its grammar modifications and the necessary extensions to type checking. Because Polyglot adopts a traditional approach that requires a hand-crafted internal representation, the Polyglot implementation of bitwise rotate declares a RotBinary class for representing rotate operators and provides the grammar declaration and parsing actions shown in Figure 4-1(a) to construct this abstract representation.

Because Polyglot uses a monolithic extension approach, the extension's entry point is the entry point to the extended compiler itself. The extension's driver runs the standard Polyglot passes, followed by a "flattening" pass, followed by the visitor that rewrites occurrences of the rotate operator. The flattening pass is part of the Polyglot toolkit and takes care of hoisting effects out of expressions. As for the CIL extension, this eliminates concerns about double evaluation and name capture and permits the simplest possible expansion of the rotate operator, though, unlike in CIL, this pass must be explicitly invoked by the extension. The Polyglot type-checker annotates all expression nodes with their type, making it easy to determine the rotation bit width.

4.2.3 xtc

Like the Polyglot extension, since xtc is designed for syntax extensions, so the xtc implementation of bitwise rotate is self-contained. Because xtc unifies external and internal

```
extend shift_expression ::=
    shift_expression:a LROT additive_expression:b
    {: RESULT =
         parser.nf.Binary(parser.pos(a, b), a, RotBinary.ROL, b); :}
|   shift_expression:a RROT additive_expression:b
    {: RESULT =
         parser.nf.Binary(parser.pos(a, b), a, RotBinary.ROR, b); :};
```

(a) The core of the grammar extension for Polyglot declares grammar rules in terms of lexemes (which are defined elsewhere by the extension) and provides parsing actions to instantiate abstract syntax classes provided by the extension. Precedence is encoded using a different expression non-terminal for each level; however, because this complexity is merely an artifact of parsing, the translation into abstract syntax represents expressions as a common Expr class with a few subclasses such as Binary.

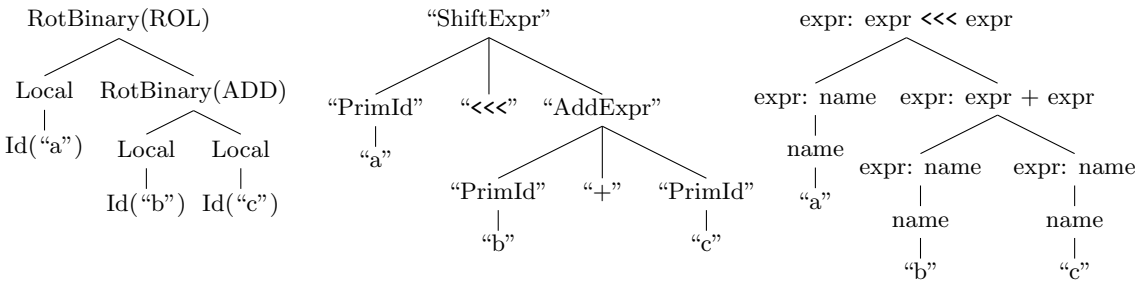
```
String ShiftOperator +=
    <Right> ...
  / <LeftRot>  "<<<":Symbol
  / <RightRot> ">>>":Symbol;
```

(b) The core of the grammar extension for xtc extends the set of symbols that can be used in a shift expression. The symbols themselves are declared separately by the extension. Though not visible here, like Polyglot, precedence is encoded using multiple expression non-terminals.

```
expr: expr "<<<" expr  [Shift]
    | expr ">>>" expr  [Shift];
```

(c) The grammar extension for Xoc, in its entirety, adds two new rules to the expression non-terminal. Xoc's parser generates the lexer and implicit parsing actions directly from this declaration. All expression types share the expr non-terminal and precedence is supported directly using named levels (i.e., Shift) and a partial order over these names (not shown).

Figure 4-1: Bitwise rotate extension grammar declarations for Polyglot, xtc, and Xoc.



(a) Polyglot's abstract syntax tree, generated by the parsing actions declared in the grammar, consists of Java objects where different classes represent different types of syntax. For example, "RotBinary" is the rotate extension's subclass of Polyglot's binary operator class.

(b) xtc's syntax tree directly reflects the parse tree of the expression, with simplifications for parentheses specified in the grammar. Each node has the same type and is distinguished by the string name of the corresponding grammar non-terminal ("ShiftExpr") and its children (i.e., "<<<").

(c) Like xtc's syntax tree, Xoc's reflects the expression parse tree, after the application of a canonicalization function. Each node has a static type ("expr"), generated from its grammar non-terminal, and is discriminated according to its grammar rule ("expr <<< expr").

Figure 4-2: Equivalent Polyglot, xtc, and Xoc representations of the expression a <<< (b+c).

representation and uses a generic abstract syntax tree, the xtc implementation of bitwise rotate only needs to provide the grammar declarations in Figure 4-1(b) to add support for both parsing and representing rotate operators.

The xtc extension follows a monolithic approach in which the entry point of the extension is the entry point of the produced compiler. Unlike CIL and Polyglot, xtc does not provide a convenient way to eliminate side-effects in expressions, so the xtc extension relies on GNU statement expressions to avoid double evaluation. While xtc's FactoryFactory provides a convenient means of constructing most of the replacement syntax, it determines the fragment's structure eagerly based on the PEG's ordered disambiguation and without the direction of type information for the slots. Thus, the extension cannot rely exclusively on syntax patterns because the syntax fragments to be substituted into the pattern often don't quite match the expectations of the pattern.

The rotate expression type-checker has convenient access to the symbol table at the point of the expression and can thus generate unique names for use in the expansion to avoid name capture. Type-checking annotates the rotate expression node with these unique names and the expression's computed type so that this information is available later during expansion.

4.2.4 Xoc

Unlike Polyglot and xtc, the Xoc extension adds rotate support to the Xoc compiler as a dynamically loaded library. Because the Xoc core is responsible for driving the compilation process, the extension does not need to provide a driver or even an entry point. The extension is simply loaded by the compiler and integrated into Xoc's scheduling at run-time when requested by the user. It declares new grammar rules, how to compute the type of a rotate expression, and how to compile a rotate expression in the input syntax tree into pure C.

Like the xtc extension, because Xoc utilizes a unified representation, as shown in Figure 4-1(c) the rotate extension needs only declare new grammar rules, which imply an internal representation. Despite the simplicity of a unified representation compared with a traditional abstract representation, Figure 4-2 demonstrates that the representations produced automatically by the rotate extensions for xtc and Xoc are remarkably similar to the hand-crafted representation produced by the Polyglot extension.

Also like the xtc extension, the Xoc extension relies on GNU statement expressions to eliminate double evaluation problems. Because Xoc automatically performs hygienic rewriting, the expansion is straight-forward without concern for name capture. Furthermore, because Xoc's type system, grammar support, and syntax patterns are integrated, Xoc can guarantee the correct construction of syntax fragments, making Xoc syntax patterns alone a sufficient interface to the abstract syntax tree. Finally, Xoc lazily annotates the syntax tree with the results of type checking, making it easy to determine the rotation bit width.

Chapter 5

Context Checking Extension

Sparse [34] is a source code checker for the Linux kernel. Sparse checks for violations of programming conventions for the Linux kernel and extends the C type system with annotations for a few important notions that are applicable beyond the Linux kernel. Here, we focus on Sparse's *context checking* features.

Context checking can be used to statically check lock/unlock pairs surrounding kernel critical sections. The *context* is an integer that follows every possible flow path through each function in a program. By default, the context entering a function must equal the context exiting the function, but a function can be annotated as increasing or decreasing the context by some amount. For example, an `acquire` function would be marked as increasing the context by one and a `release` function as decreasing the context by one. If an unannotated function were to call `acquire` without later releasing the lock, context checking would flag a context mismatch at the function's return point. Figure 5-1 shows two example functions: one that passes context checking and one that fails. While this form of lock checking is neither sound nor complete, it is effective for finding common locking mistakes.

Functions that are expected to change the context (such as `acquire`) declare their intention via the type system in the form of a GCC function type attribute `context(expr, down, up)`[1]. The context along a given control flow path can be changed explicitly using the new `__context__(expr, delta);` statement. For example, the `acquire` function in Figure 5-2 increases the context at any call point by one, and itself passes context checking by explicitly increasing the context within the function body by one.

5.1 Challenges

The context checking extension presents a different set of challenges for compiler interfaces than the other two extensions, both because it is an analysis extension instead of a rewriting extension, and because it is algorithmically intensive instead of consisting almost exclusively of type and syntax manipulations.

[1] The first argument is primarily meant for human consumption and is ignored by Sparse as of this writing. The context delta is computed as up − down.

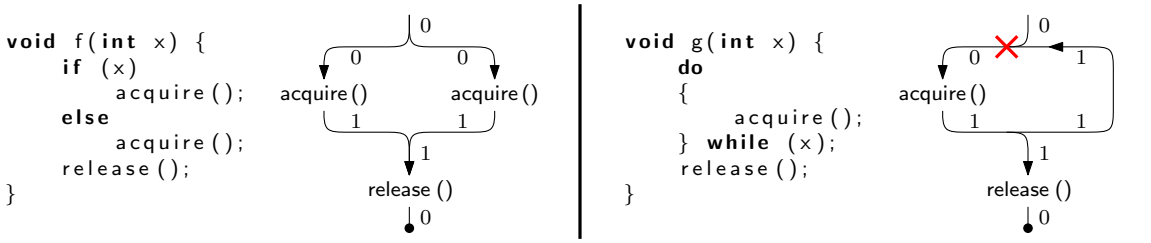


Figure 5-1: The left example function passes context checking because the context values are equal at every "join" point in the control flow of the function and the starting context equals the ending context. The right example fails context checking because there is a join point where one context is 0 and the other is 1.

```
void acquire(···) __attribute__((context(x, 0, 1))) {
    ⋮
    __context__(x, 1);
    ⋮
}
```

Figure 5-2: The acquire function's attribute annotation indicates that a call to acquire will increase the context by one at the call site. Within the function body, it explicitly changes the context by one in order to satisfy its own context declaration.

5.1.1 General Computation

Because context checking is largely algorithmic, the underlying interface—which includes the extension implementation language—must have good support for general computation in addition to good domain support for type and syntax manipulation.

5.1.2 Performance

This extension particularly stresses the performance of the underlying compiler, both because it intended for use on the Linux kernel (which not only consists of thousands of source files, but the results of pre-processing an individual source file can be tens of thousands of lines long), and because the analysis itself must traverse every syntax node.

5.1.3 Composability

Context checking in the presence of rewriting extensions presents a challenge for compiler interfaces because the extension must be aware of the *entire* language semantics in order to perform correct flow analysis, even if those semantics have been extended, for example, with new control-flow constructs. Furthermore, this should be possible without requiring awareness of other extensions in the context checking extension, or awareness of the context checking extension in other extensions.

5.2 Implementations

In addition to the Linux implementation, Sparse context checking is implemented as a CIL extension and a Xoc extension, both of which imitate the behavior of the native implementation.

5.2.1 Native

Regular Sparse was implemented from scratch in C as a full-fledged compiler front end by Linus Torvalds and others from the Linux kernel community. Not unlike Polyglot and xtc, Sparse has a modular, toolkit-like design; however, it was not explicitly designed to be extended without modifications to the core. The Sparse library parses and analyzes C programs to produce a "semantic parse tree," an internal representation based on basic blocks of linearized instructions. It recognizes, but does not analyze, extended features such as context attributes. Various back-ends operate on this tree, such as the Sparse checker itself, which performs context checking.

In order to roughly compare the performance of each context checking implementation, we compiled a typical Linux source file, do_mounts.c. Like most source files in the Linux kernel, a significant portion of this file is buried in headers; after preprocessing, it comes out to 600kB and 15,000 lines of code. The native Sparse implementation processed this file in a mere tenth of a second, a fraction of the time required by the other implementations.

5.2.2 CIL

As an analysis extension with essentially standard input language semantics, context checking fits neatly into CIL's extension model. Though CIL does not provide a general means of extending the input language, it does have support for defining new attributes. This and the function call-like syntax of the __context__ statement are enough to obviate the need for modifying the input language. The extension relies on CIL's standard library for general control flow graph computation. This, plus OCaml's excellent support for pattern matching and its standard libraries make the implementation relatively concise and straightforward.

The CIL implementation is somewhat less efficient than the native implementation, requiring three fourths of a second to check do_mounts.c. A negligible portion of this time is spent in the extension's code; half of it is spent parsing and a third of it converting to CIL's internal representation.

Unlike the native implementation of Sparse, the CIL extension can easily be composed with other extensions. Because CIL's internal representation is fixed, the context checking extension naturally composes with other CIL extensions because the fixed representation guarantees that the extension will never encounter an unknown control flow construct.

5.2.3 Xoc

The Xoc implementation of Sparse context checking builds upon Xoc's generic control flow graph library. Unlike the native implementation, where context analysis is modular but support for parsing and maintaining context information pervades the Sparse core, the Xoc implementation is entirely contained within the extension. The extension declares the syntax for context declarations and statements; extends the node structure to record context information; and hooks context checking into function compilation, analyzing function bodies only after all rewriting has been performed.

The Xoc implementation of context checking is significantly less efficient than the native or CIL implementations, coming in at fifteen seconds to check do_mounts.c. We believe most of the slowdown is due to Xoc's interpreter and not the extensibility mechanisms themselves. Preliminary tests suggest that replacing the interpreter with an on-the-fly compiler will produce a 20–40x speedup.

The Xoc extension operates on the *output* language of Xoc by performing analysis only after the application of other extensions. Thus, it automatically gains composability with other extensions, even those that add new control-flow constructs. For example, we experimentally verified that the extension can correctly analyze a program containing Alef iterators and GNU binary conditionals, both control-flow features implemented by extensions unknown to the context checking extension.

Chapter 6

Function Expressions Extension

The function expression extension adds heap-allocated lexical first-class closures to C. A far more in-depth rewriting extension than bitwise rotate, the function expression extension stresses compiler interfaces and explores the costs of implementing a complex and desirable feature as an extension to C.

This extension creates closures that behave like regular function pointers, without the usual C workaround of an extra void* parameter or the need for an alternate function pointer layout and calling convention. While simply changing the calling convention is a more natural way to implement closures from the ground up, binary compatibility with existing C libraries is worth the extra effort necessary to emulate regular function pointer semantics.

6.1 Semantics

The extension introduces a fn keyword, which, followed by a function definition, creates a heap-allocated closure that can be freed with free. For example, the snippet given in Figure 6-1 calls qsort with a newly constructed closure.

Function expressions provide safe lexical scoping by capturing by-value copies of all necessary variables from the enclosing scope directly in the closure structure at the time of creation. By-value semantics allow closures to have unlimited lifetime; unlike stack-based closures (for example, GNU C's nested functions), heap-allocated by-value closures

```
void alphabetize(int ignorecase, char **str, int nstr) {
    qsort(str, nstr, sizeof(char*),
          fn int cmp(const void *va, const void *vb) {
              const char **a = va, **b = vb;
              if (ignorecase)
                  return strcasecmp(*a, *b);
              return strcmp(*a, *b);
          });
    free(cmp);
}
```

Figure 6-1: This example shows how a function expression can be used to pass the standard library function qsort a dynamically constructed function that refers to the ignorecase variable from the enclosing scope.

```
struct env_cmp {
    int ignorecase;
};
int lambda_cmp(struct env_cmp *env, const void *va, const void *vb) {
    const char **a = va, **b = vb;
    if (env->ignorecase)
        return strcasecmp(*a, *b);
    return strcmp(*a, *b);
}
```

Figure 6-2: The function expression extension generates these top-level definitions when lifting the cmp function expression body from Figure 6-1.

are completely self-contained, and thus remain valid after the function that created them returns.

6.2 Implementation Approach

In all compilers, we implemented function expressions using a similar approach. The extension *lifts* the definition of each function expression to the top level of the program and adds an additional argument used to communicate the closure's *environment structure*. The environment structure is declared with fields corresponding to each of the variables the closure needs from the enclosing environment (ignorecase in the example in Figure 6-1). Each access to a variable from the enclosing environment within the function body is rewritten to access the appropriate field of the passed environment structure. For example, Figure 6-2 shows the lifted version of cmp, with the addition of an environment argument and a rewritten reference to ignorecase.

The value of the fn expression is a heap-allocated closure object that the extension structures as shown in Figure 6-3. It consists of executable *trampoline* code, a pointer to the lifted function's code, and the closure environment. The trampoline is a short wedge of dynamically generated assembly code that calls the lifted version of the function body with the appropriate environment structure pointer in addition to its regular arguments. This wedge of executable code is what allows the closure object to be treated like a regular function pointer. The trampoline definition takes advantage of the C calling conventions to prepend the environment structure pointer to the set of arguments it was called with before calling the lifted function.[1]

In addition to lifting the function body, the extension substitutes the fn expression with an expression that allocates, initializes and ultimately evaluates to a closure object.

[1]Note that the given trampoline code actually calls the lifted function with not one, but *two* additional arguments. As a side-effect, in addition to the environment structure pointer, this also passes the pointer to the instruction following the call to the trampoline. When the function body is lifted, this argument is also included in the argument list, but is never used in the body and, thus, is never visible to the user.

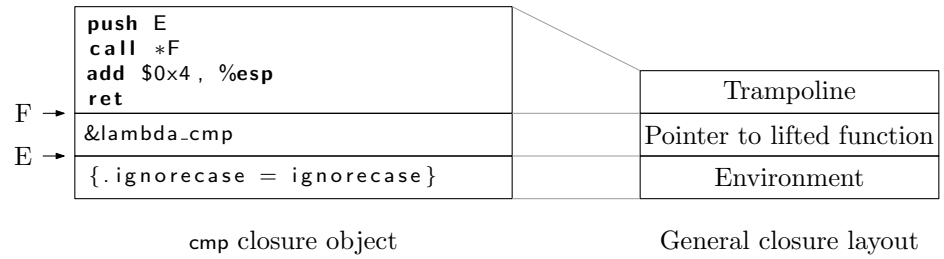


Figure 6-3: The closure object consists of an executable trampoline (shown here in x86 assembly), a pointer to the lifted function, and a filled-in environment structure. The trampoline is filled in with the necessary pointers so it can add the appropriate environment argument and call the lifted function.

6.3 Challenges

The function expression extension poses a number of challenges for each compiler's interface beyond those posed by the bitwise rotation extension. These challenges stem from the analysis necessary to lift function bodies, the complexity of the fn expression substitution, and the non-local syntax tree modifications that must be performed to lift function bodies to the global scope.

6.3.1 Generic Traversal

In order to determine the free variables of a function body and rewrite them into references to the environment structure, the function expression extension must be able to traverse the function body's syntax tree, identify variables, and cross-reference them with scope information. Short of exhausting the grammar rules that can exist in a function body, which is infeasible at best and impossible in the presence of unknown extension compositions, this requires some form of generic syntax tree traversal and rewriting, and the ability to dynamically discriminate different grammar rules.

6.3.2 Type Construction

Similar to the bitwise rotation extension, the function expression extension needs access to and control over type analysis in order to construct environment structures and to analyze the types of fn expressions. However, this extension places further strain on the interface to types because it must be able to construct and declare *new* types for the environment structures based on the results of type analysis. While the need to go from concrete type syntax to internal, abstract type representations is necessary for regular type checking, in order to construct the environment structures it must be possible to convert abstract type representations back into concrete syntax.

6.3.3 Syntax Construction

The generated code that constructs and initializes a closure object is relatively large and mostly, but by no means entirely, fixed. While any rewriting-based extension clearly needs to be able to construct syntax fragments, the need to construct a large fragment with

other fragments deeply embedded in the syntax tree places particular strain on the syntax manipulation interface.

6.3.4 Name Capture

Similar to the bitwise rotation extension, the function expression extension must deal with name capture problems. This extension suffers from both types of name capture discussed in section 4.1.2. Because it introduces new variables in the midst of user code—the environment argument and the top-level definitions of the lifted function and its environment structure—to protect against argument capture, it must guarantee both that these names do not shadow existing variables and that existing variables do not shadow generated references to these variables. Furthermore, because the generated code that constructs closure objects refers to library functions, in order to protect against free symbol capture, it must guarantee that it does not refer to a shadowed version of one of these names.

6.3.5 Type Hygiene

Beyond name capture, a subtler form of hygiene affects types and their relations to variables. Name capture problems arise when a code fragment is moved from one context to another that has potentially different meanings for symbols with the same names. Hygienic rewriting ensures that the code fragment's meaning does not change as it migrates contexts. In C, *type declarations* are scoped, just like variables. Thus, if the declaration of a variable migrates from one context to another that has different type definitions, its *type* may change or the declaration may simply become invalid if care is not taken. Similar to name capture, extensions can manage type hygiene using a purely syntactic preprocessing step that lifts all type definitions into the global scope, renaming the definitions and references as necessary.

6.4 Implementations

The function expression extension is implemented for xtc and Xoc. Unfortunately, it would not have been feasible to implement it for CIL because of the extent of the changes to the input language semantics. Likewise, because of its dependence on C specifics, a Java version implemented for Polyglot would have required a radically different approach, making a comparison meaningless.

6.4.1 xtc

The xtc implementation is structured as a sequence of explicit passes. As for the bitwise rotate extension, it must provide a driver that acts as the entry point to the extension and that parses the input, analyzes it, rewrites it, and prints the final syntax tree.

Because xtc does not automatically protect against name capture or guarantee type hygiene, the extension first rewrites the input syntax tree to lift all type declarations to the global scope. Unfortunately, this process both requires information from annotations produced by type checking and modifies the syntax tree enough to invalidate these annotations, so the extension must make *two* type checking passes: once on the original input and again after type lifting. Because the syntax tree and its type annotations are mutable, without the second pass, the type annotations would simply be incorrect.

Following this, the extension makes a pass to analyze the free variables of function bodies. This pass uses the xtc visitor framework and traverses the entire syntax tree to find primary identifiers and annotate function expression nodes with their free variables. xtc uses only a symbol's name to identify it, meaning that, as the free variable analyzer examines different parts of the syntax tree, other instances of the symbol may become difficult to correctly identify. Thus, the extension uses its own symbol representation that combines the symbol's name with its declaring scope to uniquely identify each distinct symbol.

Finally, the extension makes a pass to rewrite function expressions according to the scheme presented in section 6.2. This pass is also implemented as an xtc visitor; however, because xtc syntax nodes do not track parent pointers, the extension uses a zipper [15] structure to implement replacement of the current node and to insert declarations in the global scope. The extension manually constructs small syntax fragments such as references to variables via the environment structure, while larger syntax fragments where the overhead of writing and invoking a separate factory method is less significant are constructed via FactoryFactory templates. For the environment structure, the extension first constructs it using xtc's abstract type representation, then convert it to C syntax and finally inserts it into the syntax tree in an appropriate place.

6.4.2 Xoc

The Xoc implementation is structured as a set of new attributes and extensions to built-in attributes. Because scheduling in Xoc is entirely implicit, Xoc schedules analysis and rewriting as their results are needed, and potentially even interleaves their computations. This fine grained scheduling allows the extension to construct new syntax fragments at any time. Code generated during compile time does not need to be brought "up to speed" (e.g., if type information is necessary); analysis will be performed when needed, even if Xoc has finished the corresponding analysis of the rest of the program.

Xoc accounts for semantic information represented in attributes when producing its final output instead of simply printing the final abstract syntax tree. This has the effect of lifting type declarations in order to ensure the validity of type references, so the extension does not need to perform a type lifting pass, such as in the xtc implementation. Free variable analysis fits naturally into the attribute mechanism. The extension declares a new freevars attribute, valid over any node type, whose value is the set of free variables of that node. Xoc represents scope information using attributes, making it easily accessible to the free variable computation. Furthermore, because Xoc's standard representation of a symbol already captures uniquely identifying information, free variable analysis can work directly with standard symbol objects.

The extension extends the built-in compile attribute to declare how fn expressions should be rewritten into base C. The implementation makes extensive use of syntax patterns to construct the environment structure, the lifted function body, and the closure constructor. Xoc's syntax interface always performs hygienic rewriting, eliminating argument capture, and allows references to symbols from specific scopes (e.g. the global scope), eliminating free symbol capture. Because Xoc uses C type syntax as the interface to its (hidden) internal type representation, the environment structure can be constructed directly using C syntax.

Chapter 7

Ease of Use and Composability

Two important issues span all three extensions, and are best evaluated by considering the three extensions as a whole: ease of extension implementation and composability. Both of these issues capture the implications of large combinations of design choices made by the four compilers.

7.1 Ease of Implementation

There is no formal way to compare ease of implementation; however, there is a qualitative difference between the extension interfaces based on how natural they make the addition and manipulation of language features. While all four compilers would be difficult to use without a general understanding of compiler workings, compared to building a compiler from scratch or modifying an existing compiler, they significantly lower the barrier to entry for developers to extend languages. Between the compilers, major differences lie in how much of the compiler internals must be understood in order to implement an extension and in how many details must be managed by the extensions versus being handled by the compiler core.

As a rough measure of the ease of implementation of each version of each extension, Figure 7-1 shows the file and line counts for each implementation, all with comments and whitespace stripped.

		Rotate		**Function Exprs**		**Context**	
	Language	Files	Lines	Files	Lines	Analysis	Misc
CIL	OCaml	9	82			66	121
Polyglot	Java + PPG	13	294 + 28				
xtc	Java + *Rats!*	7	243 + 35	12	695 + 38		
Xoc	Zeta	1	34	1	170	75	183
libsparse	C					101	

Figure 7-1: Files modified and lines of code added to implement each extension in each compiler. For all three implementations of Sparse context checking, the code devoted to analyzing context has been counted separately from the code responsible for parsing and recording context. The "Misc" line counts for the Xoc and CIL context checking implementations include the emulation of code transformations automatically performed by the Sparse core.

7.1.1 CIL

The core of the rotate extension for CIL is only 36 lines of OCaml in a single new file. However, because CIL is targeted only at regular C input, adding support for the parsing and abstract syntax of the rotate operator required modifications to eight source files in the CIL core. OCaml's pattern matching facilities, CIL's `printf`-style restructuring library, and the simplicity of CIL's target language made the rotate implementation for CIL significantly shorter than the implementations for xtc and Polyglot.

The implementation of context checking for CIL is similarly succinct at 66 lines for context checking, plus 121 lines to emulate important features of the Sparse core (namely, inline function expansion). Unlike the rotate extension, because this extension fits into CIL's fixed representation model, it did not require any changes to the CIL core. OCaml's pattern matching and other language facilities, as well as CIL's simple abstract syntax tree made context analysis relatively straightforward. In fact, much of the succinctness of the native Sparse implementation derives from similar reasons: it uses C preprocessor macros to emulate higher-order functions, which OCaml supports naturally, and Sparse's "linearized" internal representation resembles CIL's abstract syntax. The extension did have to make up for one shortcoming in CIL's compilation process by manually expanding calls to inline functions, which added to its overall complexity.

7.1.2 Polyglot

The Polyglot-provided extension skeleton alone is 108 lines of Java and PPG (Polyglot Parser Generator) code and 98 lines of shell wrapper. The rotate operator implementation adds 214 lines of Java/PPG beyond this skeleton. Because the design patterns that make Polyglot flexible require many distinct interfaces and classes, even simple extensions consist of many classes and large amounts of boilerplate code. For example, in addition to specifying the translation of rotate syntax, the rotate extension has to provide implementations and factories for its abstract syntax, specify where it fits into the pass schedule, and provide a driver for compiling programs written in rotate-extended Java. Overall, these costs made a simple extension like bitwise rotate difficult to implement in Polyglot.

7.1.3 xtc

The rotate extension for xtc is 243 lines of Java code, plus 35 lines of *Rats!* parser specifications. The bulk of the implementation is concerned with navigating xtc's generic syntax trees and dealing with their dynamic types. For example, while xtc's query language, XForm, makes it easy to find all instances of rotate operator nodes in the tree and its mutable trees avoid the need to construct an output tree from scratch, xtc nodes do not maintain parent pointers, so extra bookkeeping is necessary to replace nodes in place. Similar to CIL, xtc's restructuring support reduced implementation effort, though, due to the limitations described in section 4.2.3, its use was fairly restricted.

The implementation of function expressions for xtc is 695 lines of Java, plus 38 lines of *Rats!*. Like the rotate extension, much of the code is concerned with navigating the generic syntax trees in ways that allowed the appropriate parts of the trees to be modified. Additionally, much of the implementation effort went into decomposing and constructing types in xtc's abstract type representation and avoiding potential variable capture and type hygiene problems that arise from Xoc's purely syntactic and non-hygienic rewriting.

7.1.4 Xoc

Ease of implementation is particularly important for an extension-oriented compiler like Xoc, where extensions must be easy to write and use so that the base effort required to create a new extension does not dwarf the incremental effort required to define the extension-specific details.

Bitwise rotate in Xoc is a mere 34 lines of Zeta code, including grammar definitions, type checking, and rewriting. This extension fits well in Xoc's lightweight extension model and thus was well-suited to Xoc's interface. The code consists principally of destructuring and restructuring expressions, which help make it concise. The relation between syntax patterns and ML pattern matching and the lack of extension drivers in both CIL and Xoc makes it unsurprising that the Xoc extension implementation and the core of the CIL extension implementation—at 34 and 36 lines, respectively—are similar in length.

The function expression extension for Xoc required 170 lines of Zeta code. Given the inherent complexity of implementing closures, we were happily surprised at the simplicity of the extension's implementation, and the ease with which closures could be added to the C language. In particular, Xoc's ability to manipulate types directly using C syntax and its automatic hygiene saved a great deal of effort compared to the xtc implementation. Xoc's lack of a large standard library made some algorithmically-oriented parts of the extension, such as free variable analysis, more difficult than they would have been otherwise, though the implementation was still simpler than the corresponding analysis in xtc (28 lines versus 114) because it was implemented as and took advantage of Xoc attributes.

The implementation of context checking for Xoc, at 75 lines for the actual analysis, is on par with both the CIL and native implementations. In this case, Xoc's domain-specific features were of relatively little help for simplifying the extension implementation, as it relied mostly on its general purpose features. Given that these features were modeled after C and ML, the fact that the length of the implementation fell between the lengths of the C and ML implementations lends validity to Zeta's suitability as a general purpose language.

Xoc's emphasis on reasonable default behavior contributes to ease of implementation. For example, the implementation of function expressions for xtc devotes nearly a fourth of its implementation to ensuring type hygiene. Xoc, on the other hand, provides type hygiene as default behavior because it is safe to do so, it eliminates a source of subtle bugs in extension implementations, and it can do so at virtually no performance cost. Similarly, Xoc's generated abstract representation makes grammars much easier to write (2 lines in Xoc, compared with 79 lines in Polyglot for bitwise rotate) and does not require extension authors to learn a new program representation in order to manipulate programs.

7.2 Composability

CIL and Xoc both support automatic extension composition, while Polyglot and xtc require an extension author to manually construct composed extensions. The enabling factor of automatic extension composition is that the CIL and Xoc cores produce a complete compiler by combining extension components, while Polyglot and xtc extensions produce a complete compiler by combining core components. The former plugin-like architectures focus the responsibility of composability into the compiler core, while the latter toolkit-like architectures make composability the responsibility of the extensions.

CIL has the easiest model of the four with respect to composition because the CIL core fully specifies the internal program representation. As described earlier, in section 5.2.2, the

implementation of context checking for CIL naturally composes with other CIL extensions because it can exhaustively cover CIL's entire abstract syntax. The other three compilers allow modifications to the internal representation, which makes composition much more challenging; in the presence of composition, the abstract syntax's domain is only partially known to any given extension. Extensions must be able to cope with unknown types of internal representation. Even the bitwise rotate extension illustrates the difficult issues inherent to representation extension. For example, the extension adds representations for `<<<` and `>>>` expressions, but in the context of a source file, other extensions may have added other expression types as well. An expression like "`(a <? (b <<< c)) <<< 4`" ("`<?`" is GCC's minimum operator) forces the rotate extension to analyze, compute with, and generate code for an operand that uses an extension unknown to the rotate extension's author; the extension that implements `<?` must do the same.

Handling of grammar ambiguity is particularly influential on composability because independently written extensions may define different meanings for the same syntax. For example, if the bitwise rotate extension and a "be like Java" extension were used together, a programmer might not realize that there are two definitions for `>>>` (rotate or Java's unsigned right shift). An LALR parser such as Polyglot's would statically reject this combined grammar. For extensions being composed manually by an extension author, this serves as a useful signal, though may still be difficult to identify as an actual ambiguity. For extensions composed automatically by the compiler, this error is unnecessarily strict, since it will arise even if the `>>>` operator is never used in the input program. A PEG parser like xtc's would silently choose one meaning or the other, depending on how the extensions were loaded or combined, failing to detect the problem with either manual or automatic composition. GLR can detect ambiguities only when they arise during parsing because detecting ambiguity in a context-free grammar is uncomputable. However, Xoc's GLR parser utilizes ambiguity as a signal that the programmer might not be getting the expected result and responds with an error. In the `>>>` example, Xoc would report the ambiguous operator when it appeared in the input, which is best in the case of automatic composition because it gives the benefit of the doubt to the composed extensions but still prevents ambiguity errors.

Xoc has the hardest composability challenge of the four compilers, since it supports syntax extensions and also targets automatic composition. In order to validate Xoc's claims of composability, we wrote a few programs to test various combinations of extensions. While Xoc's informal structure means there is no way to make sweeping statements about composability, and it is certainly possible to design extensions that are not usable together, these experiments at least represent a composability sanity check.

The most complex of these experiments, shown in Figure 7-2, combines the function expression extension with three others: Alef iterators, anaphoric `while`, and regular expressions. The `::` operator, introduced by the Alef iterators extension, executes its containing statement repeatedly, with each value from `0` to `nstr`. The `it` variable, introduced by the anaphoric `while` extension, is equal to the last condition evaluated by `while` (note that `it` is copied into the `fn` closure properly). Finally, the regular expressions extension contributes the `=~` match and the `$0.str` syntax. We compiled this program using all possible extension orderings and verified that they all compiled to the same, correct code.

```
void foreach(char **str, int nstr, void(*f)(char*))
{
    f(str[0::nstr]);
}

int main(int argc, char **argv)
{
    while(getline()) {
        foreach(argv+1, argc-1,
            fn void check(char *pat) {
                if(it =~ pat)
                    printf("%s\n", $0.str);
            });
        free(check);
    }
}
```

Figure 7-2: This convoluted test of Xoc's extension composition support combines four extensions to yield a program that matches each line of text returned by getline against a set of regular expressions given on the command line.

Chapter 8

Future Work and Conclusions

Despite the long history of research into making domain-specific language extension accessible to system implementers, many open problems remain in the area of compiler extensibility and there is always the opportunity for new computation models and interface paradigms.

Both Polyglot and Xoc have already begun to explore the possibility of recursively applying the capabilities afforded by extensible compilers back to the design of extension interfaces, doing for extensible compilers what extensible compilers aim to do for all systems. Xoc began as a self-bootstrapping compiler before moving to its current model to provide better experimental flexibility. However, now that Xoc is more mature, we are interested in exploring the possibility of multiversion bootstrapping, hosting the next version of Xoc atop the current version. Polyglot has already begun exploring self-hosting with Px, which is written in an extended dialect of Java, itself built in Polyglot.

The problems of composability are generally not well understood and could benefit from a formal treatment. Regarding current approaches, there are many unanswered questions regarding conflict detection and coverage analysis. While Xoc's detection of grammar ambiguities addresses one important class of conflicts, it currently has no means of detecting conflicting changes to the semantics of the base language. It also makes no guarantees that newly introduced syntax will have corresponding support throughout the rest of compilation.

Unified syntax representations are a promising approach to simplifying extension implementations and syntax patterns provide an excellent interface for hiding the details of representation from extension authors. However, these approaches, in their purest form, fail for the occasional but important differences between concrete and abstract program representation, ranging from simple differences such as the presence or absence of parenthesization to non-trivial differences such as C type representation. xtc solves simple differences directly via grammar support and abandons a unified representation for types. Xoc adheres to its unified representation, but currently depends on special support for parenthesis canonicalization and C type syntax. Better interfaces for reconciling concrete and abstract representation are an open problem.

In addition to identifying these open problems, this thesis also identifies a number of conclusions that can be drawn from the comparison of the four extensible and extension-oriented compilers. These conclusions are summarized as follows:

- Just as extensible compilers are founded on the principle that project- and domain-specific languages and language features can enhance the expressiveness, understandability, and correctness of system implementations, compiler extensions likewise benefit from languages and language features tailored to compiler construction. However, the

price of increasingly specialized extension languages is that extension authors face steeper learning curves and less library and tool support.

- Scheduling can be either the responsibility of the extensions or the compiler core, reflecting a trade-off between control over the compilation process and automatic extension composability.

- Fixed and unified internal representations both improve ease of extension implementation and enable extension composition. However, fixed representations sacrifice the ability to change the input language semantics and unified representations sacrifice the level of control that allows hand-crafted representations to easily capture language elements whose concrete and abstract representations disagree.

- Syntax patterns provide a natural interface to abstract syntax representations for extension authors. However, because syntax patterns, like unified internal representations, blur the boundaries between concrete and abstract syntax, they must provide escape mechanisms to accommodate disagreements between concrete and abstract syntax.

- Grammars, types, and syntax interfaces have a natural correspondence that, when exploited, allows a compiler to completely hide its abstract syntax representation and to statically guarantee the well-formedness of syntax transformations.

- Incorporating semantic information into the final output of source-to-source transformations sacrifices language neutrality, but enables hygienic rewriting, which can drastically reduce the complexity and subtlety of extension implementation.

Previously, tailoring a language to the needs of a project required building a preprocessor, modifying an existing compiler, or building a new compiler from scratch; all expensive undertakings. Extensible and extension-oriented compilers are now closing the gap between language designers and system implementers.

Bibliography

[1] A. V. Aho and S. C. Johnson. LR parsing. *ACM Computing Surveys*, 6(2):99–124, 1974.

[2] Jonathan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *Proceedings of the 16th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001.

[3] Jason Baker and Wilson C. Hsieh. Maya: Multiple dispatch syntax extension in Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[4] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2004.

[5] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[6] Russ Cox, Tom Bergan, Austin T. Clements, Frans Kaashoek, and Eddie Kohler. Xoc, an extension-oriented compiler for systems programming. In *Proceedings of the 13th Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[7] Eelco Dolstra and Eelco Visser. Building interpreters with rewriting strategies. In *Workshop on Language Descriptions, Tools and Applications (LDTA'02)*, volume 65/3 of *Electronic Notes in Theoretical Computer Science*, 2002.

[8] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, Massachusetts, 1994.

[10] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Prentice Hall PTR, June 2005.

[11] Paul Graham. *On LISP: Advanced Techniques for Common LISP*. Prentice-Hall, 1996.

[12] Robert Grimm. Better extensibility through modular syntax. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[13] Timothy P. Hart. MACRO definitions for LISP. AI Memo 57, MIT AI Project—RLE and MIT Computation Center, 1973. [reproduced in 32].

[14] Martin Hirzel and Robert Grimm. Jeannie: Granting Java native interface developers their wishes. In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2007.

[15] Gérard Huet. Functional pearl: The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.

[16] International Organization for Standardization. ISO: Programming Languages — C. ISO/IEC 9899:1999, 1999.

[17] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Computer Science Technical Report 32, Bell Laboratories, 1975.

[18] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: language support for building distributed systems. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[19] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[20] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, 1986.

[21] Max Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *Proceedings of the 2007 USENIX Annual Technical Conference*, 2007.

[22] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[23] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[24] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.

[25] M. Douglas McIlroy. Macro instruction extensions of compiler languages. *Communications of the ACM*, 3(4):214–220, April 1960.

[26] John Metzner. A graded bibliography on macro systems and extensible languages. *ACM SIGPLAN Notices*, 14(1):57–64, January 1979.

[27] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of the 11th International Conference on Compiler Construction*, 2002.

[28] Nathanial Nystrom, Michael Clarkson, and Andrew Myers. Polyglot: an extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, 2003.

[29] Nathanial Nystrom, Xin Qi, and Andrew Myers. J&: nested intersection for scalable software composiiton. In *Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006.

[30] Jukka Paakki. Attribute grammar paradigm: a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[31] Jim A. Roskind. A YACC-able C++ 2.1 grammar, and the resulting ambiguities. Independent Consultant, Indialantic FL, 1991.

[32] Guy L. Steele, Jr. and Richard P. Gabriel. The evolution of Lisp. In *Proceedings of the 2nd ACM SIGPLAN Conference on History of Programming Languages*, 1993.

[33] Masaru Tomita. An efficient augmented context-free parsing algorithm. *Computational Linguistics*, 13(1–2):31–46, January–June 1987.

[34] Linus Torvalds and Josh Triplett. Sparse – a semantic parser for C. http://www.kernel.org/pub/software/devel/sparse/ (retrieved December 2007), 2007.

[35] Mark G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, July 2002.

[36] E. Van Wyk, D. Bodin, L. Krishnan, and J. Gao. Silver: an extensible attribute grammar system. In *Proceedings of the 7th Workshop on Language Descriptions, Tools, and Analysis*, 2007.

[37] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute grammar-based language extensions for Java. In *Proceedings of the European Conference on Object-Oriented Programming*, 2007.

[38] Eelco Visser. Meta-programming with concrete object syntax. In *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, 2002.

[39] Eelco Visser. Program transformation with Stratego/XT. rules, strategies, tools, and systems in Stratego/XT 0.9. Technical Report UU-CS-2004-011, Institute of Information and Computing Sciences, Utrecht University, 2004.

[40] Eelco Visser, Zine el Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, 1998.

[41] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993.