A TOOLKIT FOR CONSTRUCTING REFACTORING ENGINES

BY

JEFFREY L. OVERBEY

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Research Associate Professor Ralph Johnson, Chair & Director of Research
Doctor Robert Bowdidge, Google, Inc.
Associate Professor Darko Marinov
Professor Vikram Adve
Professor David Padua

Abstract

Building an automated refactoring tool for a new programming language is an expensive and time-consuming process. Usually, only a small fraction of the tool's code is devoted to refactoring transformations; most of the code is devoted to supporting components. This dissertation shows that much of this support code can be placed in a language-independent library or code generator, significantly reducing the amount of code a tool developer must write to build a refactoring tool for a new language.

Part I focuses on the syntactic components: the lexer/parser, abstract syntax tree (AST), and source code manipulation infrastructure. Compiler writers have been generating lexers and parsers from grammars for years. However, this dissertation shows that it is possible to generate *all* of these components, including an AST that can be used to manipulate source code and is designed specifically for use in a refactoring tool. This is accomplished by annotating the grammar so that it describes both the abstract and concrete syntax of the programming language.

Part II focuses primarily on precondition checking, the procedure which determines whether or not a refactoring can be performed. It identifies preconditions as checking three properties: input validity, compilability, and preservation. Then, it shows how a language-independent component, called a *differential precondition checker*, can be used to eliminate explicit checks for compilability and preservation from many common refactorings. Since this component is language-independent, it can be implemented in a library and reused in refactoring tools for many different languages.

These techniques were used to implement automated refactoring support in Photran (a widely used, open source development environment for Fortran), as well as prototype, Eclipse-based refactoring tools for PHP and BC. In all three tools, about 90% of the code was either generated or contained in a language-independent library. The hand-written code comprised just a few thousand lines, most of which were devoted to the implementation of refactoring transformations.

Acknowledgments

Portions of this work were supported by the United States Department of Energy under Contract No. DE-FG02-06ER25752 and are part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign, its National Center for Supercomputing Applications, IBM, and the Great Lakes Consortium for Petascale Computation.

* * *

To my advisor, Ralph Johnson, for teaching me so much about research, software engineering, teaching, and writing, and for providing me with a challenging, exciting project and a stimulating working environment;

To my committee members: Vikram Adve, Robert Bowdidge, Darko Marinov, and David Padua, for their insight and for pushing my work in interesting new directions;

To Nick Chen, Chucky Ellison, Milos Gligoric, Munawar Hafiz, Fredrik Kjolstad, and Stas Negara, who, in an act of superhuman generosity, sacrificed several hours of their lives to workshop the first draft of my dissertation, providing absolutely invaluable feedback;

To Rob Bocchino, John Brant, Brett Daniel, Danny Dig, Vilas Jagannath, and other members of the Software Engineering Seminar who reviewed early drafts of the papers which later became Chapters 3 and 6;

To all of the students who worked on Photran and endured my attempts at mentoring, including Chin Fei Cheah, Matt Fotzler, Kurt Hendle, Esfar Huq, Ashley Kasza, Matthew Michelotti, Feanil Patel, Ken Schultz, Abhishek Sharma, Shawn Temming, Rui Wang, and Tim Yuvashev;

To Greg Watson and Beth Tibbitts at IBM, Jay Alameda at NCSA, Craig Rasmussen at Los Alamos National Lab, Mariano Mendez, Alejandra Garrido, Fernando Tinetti at UNLP, and all of my collaborators on Fortran-related work;

To Dennis Mickunas, who supervised my master's thesis, and Jerzy Wojdylo, who supervised my undergraduate thesis, both of whom taught me so much about mathematical writing;

To the many other colleagues I've had the privilege to work with, including Paul Adamczyk, Bariş Aktemur, Federico Balaguer, Brian Foote, Christine Halverson, Binh Le, Yun Young Lee, Kashif Manzoor, Maurice Rabb, Samira Tasharofi, Mohsen Vakilian, and Spiros Xanthos;

To Dad, Mom, Doug, Katie, and all of my friends and family:

Thank you.

Table of Contents

List of Symbols

$:$	Symbol typing relation (p. 41)
$:$	Tree typing relation (p. 59)
$<:$	Subtype relation (p. 59)
Ξ	Set of attributes in an attribute grammar (p. 52)
α, β, γ	Arbitrary strings of grammar symbols (p. 36)
ϵ	The empty string (p. 35)
Σ	An arbitrary alphabet (p. 35)
Σ^*	Set of all strings over alphabet Σ (p. 35)
τ	An arbitrary node type ($\tau \in \mathcal{T}$) (p. 41)
Ξ_X	Set of attributes associated with symbol X in an attribute grammar (p. 52)
ξ	An arbitrary attribute in an attribute grammar (p. 52)
$A ::= \alpha$	An arbitrary production in a grammar (p. 36)
A, B	Arbitrary nonterminal symbols (p. 36)
a, b, c	Arbitrary terminal symbols (p. 36)
$\mathcal{C}$, $\mathcal{C}_{gen1}$, $\mathcal{C}_{omit}$, etc.	Node classification sets (p. 40)
$\mathcal{C}_{fields}$	Set of classes with fields (p. 47)
$\mathrm{Dom}(\xi)$	Domain of attribute ξ in an attribute grammar (p. 52)
$\mathcal{F}_{\mathcal{K}}$	Set of fields in class $\mathcal{K}$ (p. 48)
$G = (N, T, P, S)$	Context-free grammar (p. 35)
$I_{\mathcal{C}}$	Set of class identifiers in an annotated context-free grammar (p. 37)
$I_{\mathcal{F}}$	Set of field identifiers in an annotated context-free grammar (p. 37)

induces	Symbol-field induction relationship (p. 48)
inherits-from	Inheritance relation for AST node classes (p. 45)
inlines	Inlining relation for AST node classes (p. 47)
is-associated-with	Field-symbol association relationship (p. 48)
Inh_X	Set of inherited attributes associated with symbol X in an attribute grammar (p. 52)
$\mathcal{K}$	An arbitrary node class name ($\mathcal{K} \in I_C$) (p. 41)
N	Set of all nonterminal symbols in a context-free grammar (p. 35)
$\mathrm{nann}(A)$	Annotation given to the nonterminal A in an annotated context-free grammar; one of ***generate***, ***omit***, ***list***, ***super***, or ***custom*** (p. 37)
$\mathrm{nlbl}(A)$	Label of nonterminal A (on the left-hand side of a production) in an annotated context-free grammar (p. 37)
P	Set of all productions in a context-free grammar (p. 35)
p	An arbitrary production in a context-free grammar (p. 36)
$\mathrm{plbl}(A ::= \alpha)$	Label of the production $A ::= \alpha$ (i.e., the label given following the symbol $\Leftarrow$) in an annotated context-free grammar (p. 37)
$\mathrm{Pos}(p)$	Set of generating positions in production p (p. 48)
R	Set of semantic rules in an attribute grammar (p. 52)
R_p	Set of semantic rules associated with production p in an attribute grammar (p. 52)
S	Start symbol in a context-free grammar (p. 35)
$\mathrm{sann}(A ::= \alpha, i)$	Annotation given to the i-th symbol on the right-hand side of the production $A ::= \alpha$; one of ***generate***, ***omit***, ***boolean***, or ***inline*** (p. 37)
$\mathrm{slbl}(A ::= \alpha, i)$	Label of the i-th symbol on the right-hand side of the production $A ::= \alpha$ in an annotated context-free grammar (p. 37)
Syn_X	Set of synthesized attributes associated with symbol X in an attribute grammar (p. 52)
$\mathcal{T}$	Set of node types (p. 41)
T	Set of all terminal symbols in a context-free grammar (p. 35)
$\mathrm{type}(\mathcal{K}, \ell)$	Type of field ℓ in class $\mathcal{K}$ (p. 50)
$\mathrm{vis}(\mathcal{K}, \ell)$	Visibility of field ℓ in class $\mathcal{K}$ (p. 50)

X, Y, Z	Arbitrary grammar symbols (p. 36)
x, y, z	Arbitrary strings of terminal symbols (p. 36)
‹*nonterminal-symbol*›	Nonterminal symbol in an example grammar (p. 36)
TERMINAL-SYMBOL	Terminal symbol in an example grammar (p. 36)

1 Introduction

1.1 A Brief History of Automated Refactoring

The idea of program transformation has been around for almost as long as there have been programs to transform. IBM's compiler for FORTRAN I (1959)—the first compiler for the first high-level language—performed common subexpression elimination, loop-independent code motion, and constant folding, among other transformations [9]. Behavior-preserving transformations at the source code level first gained interest during the 1970s, motivated initially by the desire to convert programs using `goto` statements into structured programs [12]. This application led to the term *restructuring,* which took on a more generalized meaning[1] that followed it into the 1980s and 1990s.

By that time, graphical user interfaces were becoming more widely available, and they proved to be a boon to restructuring tools. In contrast to the batch systems of the previous decades, the restructuring tools of this era were *interactive.*

This proved particularly beneficial to researchers working on parallelizing compilers, who were discovering that fully-automatic, coarse-grained parallelization could not rival the work of a competent human. In response, they built tools like PTOOL [10], $\mathbb{R}^n$ [22, 25, 26], ParaScope [13, 24, 44, 51, 53], Faust [43], and D [45], which integrated their compilers' dependence analyses and loop transformations into an interactive tool. While the tool could perform the transformations and (attempt to) verify their correctness, the choice of which transformations to apply could be left to the programmer.[2]

While those researchers were building interactive, behavior-preserving, source-level program transformation tools for performance tuning, two other researchers established the idea that these tools could be used for an entirely different purpose: They could be used to help programmers make design changes during software maintenance. In 1991, Bill Griswold's dissertation [42] identified several common transformations—including moving, renaming, inlining, and extracting program entities—and detailed their implementation, prototyping them in a restructuring tool for Scheme. Around the same time, Opdyke and Johnson introduced the term "refactoring" into the

[1] A commonly-cited definition of *restructuring* appears in Chikofsky & Cross [23]: "Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics). A restructuring transformation is often one of appearance, such as altering code to improve its structure in the traditional sense of structured design." In contrast, refactoring is a specific *technique* for performing restructuring which uses small-scale, behavior-preserving changes to achieve larger, behavior-preserving changes in software systems [38].

[2] Subsequent interactive parallelization tools include SUIF Explorer [58] and GPE [20].

literature [68][3]; Bill Opdyke's 1992 dissertation [67] catalogued transformations for building object-oriented frameworks and prototyped them in a refactoring tool for C++. While the big-picture ideas were similar, Griswold's dissertation focused largely on tool implementation and guaranteeing correctness, while Opdyke's focused more on the catalog of refactorings.

Shortly thereafter, Brant and Roberts began developing the Smalltalk Refactoring Browser [80], which lead to Roberts' dissertation [81]. The Refactoring Browser was destined to be more than a research prototype; it was intended to be a useful, production-grade tool. Unlike previous restructuring tools, the Refactoring Browser was *integrated* into the Smalltalk development environment, allowing refactoring to be seamlessly intermixed with coding. It quickly gained popularity, most notably among the developers at Tektronix who later invented eXtreme Programming (XP). XP became the first software process to advocate refactoring as a critical step.

The popularity of XP, coupled with the subsequent publication of Fowler's book, *Refactoring* [37], brought "refactoring" into the software development parlance. The 2000s saw a proliferation of refactoring tools. Automated refactoring became available to Java programmers on a large scale in 2001, when it was included in the (heavily Smalltalk-influenced) Eclipse JDT and IntelliJ IDEA. They were subsequently added to other IDEs including Microsoft Visual Studio, Sun NetBeans, and Apple Xcode, and other languages have been supported in the form of refactoring plug-ins for Eclipse, NetBeans, Visual Studio, and even *emacs* and *vi.*

1.2 The Problem

While automated refactoring tools have been commercially available for more than a decade, and the components comprising them are well-known, the task of building a new automated refactoring tool not much easier than it was ten years ago. In short, these tools suffer from an infrastructure problem.

The term *infrastructure* will be used to refer to supporting components: those that are necessary to perform refactoring but do not contribute directly to the functionality of any particular refactoring. Figure 1.1 on page 6 shows the components found in a typical refactoring tool; these will be described in more detail in Section 1.4.2. The analysis and transformation infrastructure generally consists of the bottom four tiers in Figure 1.1: the pseudo-preprocessor, lexer, parser, static analyses, program database, and source code rewriter.

The problem is that a refactoring tool's infrastructural requirements are large, often requiring tens or hundreds of thousands of lines of code, and most of this infrastructure must be in place before the refactorings themselves can be developed. Even the Rename refactoring—the most commonly-used refactoring and, anecdotally, the first refactoring implemented when new refactoring tools are built—requires every component shown in Figure 1.1. An existing parser and AST can sometimes be reused to partially mitigate the infrastructure problem, but as we will see in Section 1.4.1, refactoring tools have unique demands that often limit possibilities for reuse.

This infrastructure problem translates directly into an economic problem. While compilers and (to a lesser degree) debuggers are *essential* parts of the developer's toolkit, refactoring tools are not. Developers do not *have* to have automated refactoring tools. They can always perform the same

[3]However, Johnson admits that Kent Beck and others at Tektronix were using the term conversationally before then.

transformations by hand; it just takes longer. This can make it difficult to justify the expense of building such a tool.

As an example, consider Photran [75], a refactoring tool for Fortran. The entire Photran IDE is just under 250,000 lines of code (LOC). Refactorings comprise about 10,000 LOC. However, Photran's refactoring *infrastructure* is roughly 100,000 lines of code.

Consider the cost of this infrastructure, ignoring the refactorings themselves and the rest of the IDE. The COCOMO productivity average for a 100,000 LOC project is 2,600 LOC/year [60, Table 27-2]. By that metric, if Photran were constructed from scratch by hand, it would require 38.5 person-years, or, supposing a developer costs $250,000/year, $9.6 million. If a company were funding it, to achieve a positive return on investment, Photran would need to either (1) save more than 38.5 years of development time for the company or (2) provide them more than $9.6 million of income. And this is only to recoup the *infrastructure* cost; the costs to build the refactorings themselves (and the rest of the IDE) add considerably to this number. Given these estimates, most managers would probably conclude that Photran is too expensive to build.

1.3 The Results

In reality, Photran's developers did not write 100,000 lines of infrastructure code. They only wrote about 5,000. The remaining 95,000 LOC is due to two tools. Most of it was produced by a code generator called Ludwig. The rest is in a language-independent library called the Rephraser Engine.[4]

In other words, thanks to Ludwig and the Rephraser Engine, the amount of maintained infrastructure code in Photran is roughly 1/20 the size it would be if it were written entirely by hand. But while the number of lines of code is reduced by 95%, in cost estimation, the savings are even greater—closer to 96%. There is a *diseconomy* of scale in software development, so the COCOMO productivity average actually *improves* to 3,200 LOC/year for smaller projects. So the estimated time and cost to develop Photran's refactoring infrastructure using Ludwig and the Rephraser Engine is actually closer to 1.5 years, or $391,000—about 4% of the estimated cost to develop it entirely by hand.

Both Ludwig and the Rephraser Engine are the work of the present author. This dissertation describes some of the underlying concepts—the concepts that made it possible to reduce the amount of infrastructure code in Photran by about 95%. The next section will provide some background on the internal structure of refactoring tools; then, the following section will summarize the major results and provide an overview of the remainder of the dissertation.

1.4 The Architecture of Refactoring Engines

1.4.1 Refactoring Engines vs. Compilers

Many of the components needed to build a refactoring tool are also found in compilers. Both automated refactoring tools and compilers perform two tasks: they *analyze* source code, then *transform* it. Of course, the transformations are quite different, but the analyses are largely the

[4] Granted, these are highly nontrivial—Ludwig is about 40,000 LOC, and the Rephraser Engine is about 10,000 LOC—but they are stock tools that are reused without modification.

same. A parser (syntactic analysis) is absolutely essential, and usually some semantic analyses are required as well, such as the computation of name bindings (symbol tables) and type checking.

Since these analysis requirements are so similar, it should not be surprising that refactoring tools contain many of the same components as compiler front ends. In fact, some refactoring tools have been built by reusing components from compilers.

But, unfortunately, reusing components from compilers is not always possible. This is because refactoring tools and compilers have very different concerns, so a component built for a compiler will not always meet the requirements of a refactoring tool. Some of the salient differences are as follows.

- *The user owns the refactored code.* Ultimately, a refactoring tool must modify the user's source code, and *the user will maintain the refactored code.* This means that, as much as possible, the tool must change the source code in exactly the same way the user would if he were performing the transformation by hand. So a refactoring cannot just prettyprint an AST, ruining the user's formatting. It cannot even remove comments. And it certainly cannot lower the representation and output a semantically equivalent but visually dissimilar program.

- *Refactoring tools must handle preprocessing differently.* In languages like C, C++, C#, and Fortran, the user's code is generally run through a preprocessor before a compiler ever sees it. However, since a refactoring tool must modify the *user's* source code, it must be able to parse, analyze, and transform code with embedded preprocessor directives.

- *Refactoring tools must perform static analyses on source code, or at least be able to map the results back to source code.* Compilers do not perform all of their static analyses directly on the AST. Typically, more advanced analyses are performed on a simpler, lowered representation. Flow analysis is one common example. When a refactoring requires a control flow graph (CFG), it usually needs to know the flow between statements *in source code.* While it may be possible, in certain cases, to reverse engineer a source-level CFG from one for a lowered representation, in practice most refactoring tools simply compute their own CFG directly from the nodes in the AST.

- *Refactoring tools are interactive.* First, this means that analyses do not have to be completely conservative, and transformations do not have to be completely accurate; often it is acceptable for the tool to make a "guess" then ask the user to visually inspect the result. Second, the user decides what refactoring to invoke and can provide input, so there is generally no need for the tool to estimate the profitability of its transformations; that responsibility has been transferred to the user. Finally, there are speed considerations. While compiler optimizations must be *fast,* in order to ensure a short compile time, refactorings must be *responsive*—usable in "interactive time," and noticeably faster than if the user were to perform the transformation by hand.

The notion that a refactoring tool might sacrifice correctness to improve speed is unsettling, particularly to those with a background in compilers (indeed, a compiler that is fast but produces incorrect results would be useless). In fact, this is not completely unique to refactoring tools: In an IDE, syntax highlighting, automatic indentation, and indexing are often performed using incomplete

heuristics, simply because they are significantly faster, they are sufficient 99% of the time, and the inaccuracies are relatively inconsequential. Bowdidge [15] describes how a similar argument motivated the speed-accuracy trade-offs employed in the design of Apple Xcode's refactoring support for Objective-C.

The reason automated refactoring "caught on" in industry is because it provided a noticeable productivity gain; developers could intersperse refactoring with coding, maintaining high-quality design without sacrificing long periods of time to the arduous process of manual restructuring. So refactoring tools have to handle common cases well, and their balance between speed and accuracy must provide an overall benefit to the end user. The large number of bugs in production refactoring tools [31] is a good indication that these tools are still useful even when they do not always perform correct transformations. However, most of these correctness problems deal with obscure corner cases, and most of them result in refactored code that will not compile. So, users are unlikely to see them, and those that do can simply undo the refactoring and perform it correctly by hand.

Balancing speed and accuracy is tricky. Ideally, the goal is to find a compromise which maximizes the productivity of the developer using the refactoring. For example, it is probably not worth adding an expensive, interprocedural pointer analysis in order to catch a rare corner case in a refactoring; instead, simply notify the user that this case is not handled. On the other hand, a parallelization refactoring which is fast but introduces subtle bugs (like race conditions) would probably benefit from a more thorough analysis; the user will not perceive any benefit if he is forced to manually check for race conditions in the refactored code.

In sum, compilers and refactoring tools have many similarities, but they have important differences. In the next section, which discusses the components in a typical refactoring tool, keep in mind that the apparent similarities may be deceptive: There is the *potential* to reuse some compiler components to construct a refactoring tool, but this potential is often hard to achieve.

1.4.2 Components of a Refactoring Engine

Since refactoring tools have enjoyed more than a decade of commercial success and many more years of research, it is well known, at least at a high level, what components are required to build these tools. Figure 1.1 illustrates the architecture of a typical refactoring tool, biased somewhat toward the design of the Eclipse Java and C/C++ Development Tools (JDT and CDT, respectively). The architecture follows the relaxed layered model [19, p. 45], where each layer generally depends on several of the layers below it.

The bottommost tiers consists of an abstract syntax tree (AST) and the machinery needed to construct it: the lexer, parser, and (if applicable) preprocessor. The AST plays a critical role in a refactoring tool, as virtually every other component depends on it and makes assumptions about its structure. In fact, many refactoring tools also use the AST for source code transformation: they make changes to source code by adding, deleting, moving, or modifying nodes in the AST, and then use these AST changes to make corresponding changes to the concrete syntax.

When a language must be preprocessed before it can be parsed, some information about the effects of preprocessing must be included in the AST in order to refactor successfully. For example, it is often necessary to know which AST nodes originated from `#include` directives in the original code. This requires additional information to be passed from the preprocessor to the

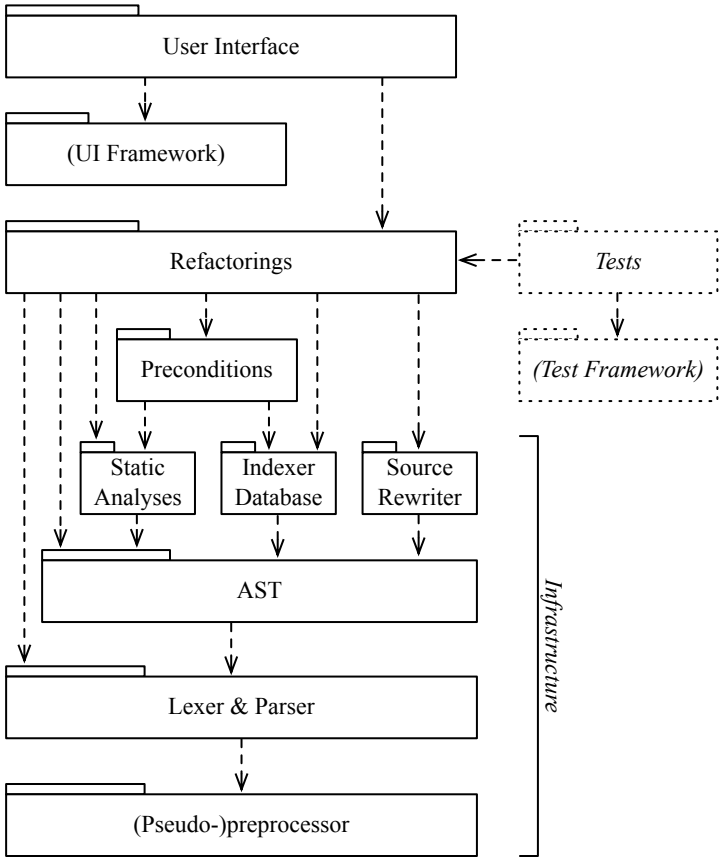


Figure 1.1: Architecture of a typical refactoring tool.

lexer and parser; otherwise the preprocessor behaves exactly as it would for a compiler. However, correctly handling conditional compilation (`#ifdef`) directives requires substantial changes to the preprocessor's behavior; Garrido [40] calls the resulting tool a *pseudo-preprocessor.*

Immediately above the AST is a suite of semantic analyses. Exactly what analyses are implemented depends on what refactorings are implemented. Rename and Move, for example, require only name binding analysis; Extract Method requires some level of flow analysis; and loop transformations require array dependence analysis. The choice of analyses can also depend on the language being refactored. For example, Extract Local Variable requires type checking in a statically-typed language but not in one that is dynamically typed; and Extract Method can be implemented without flow analysis if the language supports call-by-reference.

When the refactoring tool will operate on large projects, often some semantic analysis information must be saved to disk in order to achieve reasonable performance. For example, renaming a method requires knowing all of the call sites of that method and of any methods that override it; most tools choose to save this information (or some approximation thereof), based on the observation that (1) the brute-force approach of parsing and analyzing every file in the project while the user waits on the refactoring to complete would be prohibitively expensive, and (2) in a million-line project, most of the files will likely *not* call that method anyway. In an IDE, the cross-

reference database can be updated incrementally as the user edits individual files, so maintaining this database need not affect the tool's responsiveness.

The remaining components support refactoring more directly. There is a suite of precondition checks which are common among several refactorings. A source rewriter maps AST transformations to textual (offset/length) transformations, accounting for comments and reindentation. And the topmost layer is, of course, the user interface, which for convenience is generally integrated into an IDE or text editor.

1.5 The Solution

This dissertation shows that many of the infrastructural components in a refactoring tool can be placed in a language-independent library (the Rephraser Engine) or code generator (Ludwig), significantly reducing the amount of code a tool developer must write to build a new refactoring tool, regardless of what target language the tool will operate on.

1.5.1 Three Refactoring Tools

To prove this point, Ludwig and the Rephraser Engine were used to develop three different refactoring tools, which will be used throughout this dissertation as case studies. All were built as Eclipse plug-ins. Eclipse is a framework for building integrated development environments; it is particularly well known for its Java Development Tools [3].

- **Photran, an integrated development environment and refactoring tool for Fortran 95.** Photran is by far the largest, most visible, and most compelling case study. Although it originated from UIUC, Photran is now an open-source project hosted by the Eclipse Foundation and a component of the Eclipse Parallel Tools Platform. It has an active user community; new releases generally receive about 1,500 downloads per month. Photran's parser and AST (generated by Ludwig) were extended by colleagues at Fujitsu Japan to support XPFortran [59, 66]. Refactorings have been developed by several students at UIUC as well as by colleagues at Unijuí Universidade Regional (Brazil) and Universidad Nacional de la Plata (Argentina). In Fall 2010, approximately 120 students in Software Engineering I at UIUC created new automated refactorings for Photran for their class projects. Photran 7.0 (released in June 2011) contains 31 refactorings.

- **A prototype refactoring tool for PHP 5.** PHP is a popular scripting language used to develop Web applications. (Notably, Facebook is written in PHP.) PHP is dynamically typed and is usually interpreted.

- **A prototype refactoring tool for BC.** BC is a small, arbitrary-precision calculator language that is available on many Unix systems and is specified in the POSIX standard [47]. Its syntax resembles the C language. It is interpreted and is intended to be used interactively.

Some of the salient differences between Fortran, PHP, and BC are shown in Table 1.1. Fortran and PHP were chosen because they are production languages, but they are quite different from each other. Fortran 95 is a compiled, statically typed, procedural language. PHP 5 is an interpreted,

		BC	Fortran 90	PHP 5
General	Specification	POSIX/IEEE 1003.1-2008	ISO/IEC 1539-1:1991	(Implementation from www.php.net)
	Application	Calculation	Scientific	Web (server-side)
	Paradigm	Procedural	Procedural	OO/Procedural
	Compiled	No	Native Code	No
	Memory Management	None	Manual	Automatic
Typing	Type Safety	Unsafe	Unsafe	Safe
	Type Checking	None/Dynamic	Static	Dynamic
	Type Declarations	None	Optional	Limited
Bindings	Function Binding	Dynamic	Static	Dynamic
	Variable Scoping	Dynamic	Static	Static
	Implicit Variables	Globals	Locals	(None)
	Primitive Namespaces	3	1	4
	Resolution	Context	—	Sigils, Context
	Labeled Namespaces	No	No	Yes
	Single Assignment			
	Functions	No	Yes	Yes
	Variables	No	No	No
Flow	Structured Ctl. Flow	Yes	No	No
	Exceptions	No	No	Yes
	Parameter Passing			
	By Value	Yes	Yes	Yes
	By Reference	No	Yes	Yes

Table 1.1: Comparison of Fortran, PHP, and BC.

dynamically typed, hybrid procedural/object-oriented language. BC was chosen for illustrative purposes: It contains functions, scalar and array variables, and all of the usual control flow constructs, but it is a much smaller and simpler language than either Fortran or PHP. The refactoring tool for BC is intended to serve as the smallest realistic example; it will be described in some detail in Chapter 7.

1.5.2 Generating Rewritable Abstract Syntax Trees

Part I of this dissertation (which includes Chapters 2–4) focuses on the lexer/parser, abstract syntax tree (AST), and source code manipulation infrastructure—syntactic components in a refactoring tool. Compiler writers have been generating lexers and parsers from grammars for years. However, this dissertation shows that it is possible to generate *all* of these components, including an AST that can be used to manipulate source code and is designed specifically for use in a refactoring tool. This is accomplished by annotating the grammar so that it describes both the abstract and concrete syntax of the programming language.

Chapter 2 focuses on the abstract syntax tree (AST), which plays a crucial role in refactoring engines. Although some compilers also contain ASTs, they are often unsuitable for use in a refactoring tool. This chapter identifies three requirements that the AST in a refactoring tool must satisfy: (1) the AST must accurately model the original source code; (2) it must be able to map AST nodes to precise locations in the source code; and (3) if the language is preprocessed, it must capture enough information that the original, unpreprocessed code can be modified.

Chapter 3 describes a system that uses a grammar to generate AST node classes as well as a parser that constructs ASTs comprised of these nodes. This is implemented in Ludwig. The system starts with the grammar that would be supplied to a parser generator. Although it can generate a

"default" AST from this grammar, the grammar's author can also use seven *annotations* to customize the AST design, allowing the generated AST to have virtually the same structure as a hand-coded AST. The generated ASTs implement a very rich API. It is easy to traverse the AST or search for particular nodes. More importantly, the AST can be used to manipulate source code: There are a number of methods for restructuring the AST, and the revised source code can be emitted from the modified AST. The modified source code preserves all of the user's formatting, including spacing and comments; there is no need to develop a prettyprinter. Internally, this is accomplished through a process called *concretization,* where details about the concrete syntax are hidden in the AST nodes.

Chapter 4 describes, in detail, how the system in Chapter 3 is implemented. It provides a formal description of exactly what AST nodes are generated and how the generated ASTs are structured. Since it is possible to annotate a grammar in a way that is erroneous (e.g., it would cause a class to inherit from itself), the formalism also includes constraints that an annotated grammar must satisfy to be well-formed. This allows the AST generator to detect such problems in the grammar, pointing out the erroneous annotations, rather than generating code with errors in it.

In total, Part I makes four contributions:

1. It identifies three criteria that ASTs in refactoring tools generally must satisfy (Chapter 2).
2. It proposes seven annotations which can be applied to a grammar to describe the structure of abstract syntax trees for that language (Chapter 3).
3. It shows that such an annotated grammar can be used to generate AST nodes as well as a parser that constructs ASTs from these nodes, and the generated ASTs can be designed in a way that makes them ideal for manipulating source code (Chapter 3).
4. It provides a formal description of the AST generator's operation (Chapter 4).

1.5.3 Differential Precondition Checking

Part II of this dissertation (which includes Chapters 5 and 6, focuses primarily on precondition checking, the procedure which determines whether or not a refactoring can be performed. It identifies preconditions as checking three properties: input validity, compilability, and preservation. Then, it shows how a language-independent component, called a *differential precondition checker*, can be used to eliminate explicit checks for compilability and preservation from many common refactorings. Since this component is language-independent, it can be implemented in a library and reused in refactoring tools for many different languages.

Chapter 5 reviews the existing literature and the state of the practice, looking at what refactorings are available in current tools, what refactorings tend to be the most commonly used, and what semantic information (i.e., what static analyses) these refactorings require.

Chapter 6 describes how a differential precondition checker works. It builds a *semantic model* of the program prior to transformation, simulates the transformation, performs semantic checks on the modified program, computes a semantic model of the modified program, and then looks for differences between the two semantic models. The refactoring indicates what differences are expected; if the actual differences in the semantic models are all expected, then the transformation is considered to be behavior preserving. The changes are applied to the user's code only after the differential precondition checker has determined that the transformation is behavior preserving. This

technique is simple, practical, and minimalistic—just expressive enough to implement preconditions for the most common refactorings. Most importantly, the core algorithm can be implemented in a way that is completely language independent. This is done by representing semantic relationships (such as name bindings, control flow, etc.) using offset/length information in the source code. This means that a completely language-independent differential precondition checker can be implemented, optimized, placed in a library, and reused in refactoring tools for many different languages.

Part II makes makes five contributions, all in Chapter 6:

1. It characterizes preconditions as guaranteeing *input validity, compilability,* and *preservation*.
2. It introduces the concept of *differential precondition checking* and shows how it can simplify precondition checking by eliminating compilability and preservation preconditions.
3. It observes that semantic relationships between the modified and unmodified parts of the program tend to be the most important and, based on this observation, proposes a very concise method for refactorings to specify their preservation requirements.
4. It describes how the main component of a differential precondition checker (called a *preservation analysis*) can be implemented in a way that is both fast and language independent.
5. It provides an evaluation of the technique, considering its successful application to 18 refactorings and its implementation in refactoring tools for Fortran (Photran), PHP, and BC.

I

Syntactic Infrastructure

2

Reuse and the Role of the Abstract Syntax Tree

Since refactoring tools analyze and manipulate source code, the primary program representation in a refactoring tool is always one that is fairly close to source code—generally an abstract syntax tree. The AST plays an absolutely critical role in a refactoring tool. It is used for both analysis and transformation, and every refactoring makes use of it.

However, because refactoring tools have different concerns than compilers (as discussed in the previous chapter), refactoring tools impose somewhat different requirements on their ASTs. This chapter will explore how ASTs are used in refactoring tools and what unique requirements this may impose on them. Then, it will discuss the possibility of reusing ASTs from compilers or IDEs in a refactoring tool.

The observations made in this chapter are not completely original—virtually every developer of a serious refactoring tool has learned them "the hard way"—although this chapter is perhaps the first attempt to put them into writing. The main contribution is a list of three requirements that ASTs must satisfy to be useful in a refactoring tool. These will motivate and constrain the ASTs that will be discussed in the next chapter.

2.1 Abstract Syntax Trees: Theory and Practice

What makes an abstract syntax tree "abstract" is that it models only the *essential* elements of the syntactic structure—it omits some details of the concrete syntax. But what, exactly, is omitted?

In theory, the answer is straightforward: If an element of the concrete syntax serves a syntactic purpose but does not have semantics *per se,* then it is omitted from the AST. For example, in the expression `f(3,4)`, the comma and parentheses do not have any meaning by themselves; they exist only to identify the expression as a function call and to separate its arguments. This is a useful distinction for a language that has a formally-specified semantics. But most production languages do not, so the notion of an expression "having semantics" is not well-defined, and it is not always clear what should be included in, or omitted from, an AST. (Two examples from Fortran are given in the next section.)

Moreover, when an abstract syntax tree is implemented in a tool, it is implemented with a purpose: to facilitate compilation, or static analysis, or refactoring, other such tasks. So, in practice, the AST is not just a data structure; it is a software component. It may include additional information and functionality to accommodate the required tasks. In a compiler, the AST nodes might contain line number information (in order to produce error messages) and type information for expressions.

The AST for a prettyprinter would not need type information, but it might store comments so that they would not be lost during prettyprinting. And the requirements for an AST in a refactoring tool are even more unique...

2.2 Three Requirements for ASTs in Refactoring Tools

So, what makes an AST suitable for use in a refactoring tool? In the author's experience, there are three minimum requirements that need to be met: It needs to be an accurate model of the original source code, it needs to have precise source-location information, and, if the language is preprocessed, it needs to contain sufficient information about any preprocessing that was performed prior to parsing.

2.2.1 Requirement 1: Accurate Model of the Original Source Code

As noted above, ASTs omit details of the concrete syntax that are irrelevant, that do not serve the task at hand. But details of the concrete syntax that are irrelevant to translation or type checking may be very relevant to refactoring, so an AST used for refactoring may need to model the original, concrete syntax more accurately than an AST used for compilation or static analysis. The Fortran language provides two examples.

Example: Fortran Input/Output Statements

Fortran contains a number of statements that perform I/O. Two of these are PRINT and WRITE: the statements `print *, "X"` and `write (*,*) "X"` can be used interchangeably to print the string `X` to standard output. In general, the statement `print` *fmt*`,` *expr* is equivalent to the statement `write (*,` *fmt*`)` *expr*. Most compilers type check and translate them identically. So, a compiler may choose to represent PRINT statements as WRITE statements in its AST. GNU Fortran does this, for example.

Now, the ability to convert PRINT statements to WRITE statements actually makes for a useful refactoring. (Consider a program which uses PRINT statements to write to standard output, but the programmer later needs to use WRITE statements to write to a file instead.) To implement this refactoring, a refactoring tool needs to be able to distinguish PRINT statements from WRITE statements—a difficult task if they are represented identically in the AST.

Example: Fortran Specification Statements

In Fortran, there are three ways to specify that a variable is an array: (1) `real :: array(10)`, (2) `real, dimension(10) :: array`, and (3) `dimension array(10)`. In a Fortran compiler, these statements would likely be omitted from the AST, since their only effect is to add or refine information in the symbol table. (Again, GNU Fortran does this.)

However, a refactoring tool needs an accurate representation of these statements. An obvious refactoring is one which converts one form of the declaration into another; this is difficult to do without a representation of these statements in the AST. Less obviously, there are other refactorings which need information about these so-called *specification statements*. The refactoring Introduce IMPLICIT NONE is a good example. Fortran does not require variables to be declared before use unless

there is an IMPLICIT NONE statement. So, this refactoring adds this statement, determines the types of all variables that were declared implicitly, and adds type declaration statements for these variables. However, the Fortran grammar has very strict requirements about where the IMPLICIT NONE statement may or may not appear, relative to other specification statements. Again, if they omitted from the AST, it can be difficult to ensure that this statement is inserted at the correct location.

2.2.2 Requirement 2: Precise Source-Location Mapping

If an AST is not a completely accurate model of the original source code, as discussed previously, it might still be usable in a refactoring tool. However, it may limit what refactorings can be implemented.

In contrast, one thing that is virtually *required* of an AST in a refactoring tool is the ability to map AST nodes to locations in the source code—e.g., to determine that a particular function call node in the AST corresponds to the four-character string `f(3)` starting on line 9, column 10. Often, it is easiest to find this information by storing, in each AST node, a starting position (either line/column or character offset) and length.

This information is used for two purposes. Some refactoring tools use the source location information to perform source code manipulation; this will be discussed later. But in virtually all refactoring tools, the user is often required to select a region of source code in an editor. Rename requires the user to select an identifier. Extract Method requires the user to select a sequence of statements in a method. Interchange Loops requires the user to select a perfect loop nest. The AST nodes need to contain accurate source location information in order to map the user's selection in the editor to the corresponding node in the AST.

As an aside, the C preprocessor's `#line` directive may be used to change the line numbers that appear in a compiler's error messages. A compiler may choose to store this information in its AST, rather than the true source location. Obviously, a refactoring tool needs to know the *actual* source location of an AST node; the use of the `#line` directive should not change the behavior of the refactoring.

2.2.3 Requirement 3: Sufficient Preprocessing Information

Preprocessors (such as the C preprocessor and M4) do not pose much of a problem for compilers. In a compiler, the preprocessor can be run first, and then its output can serve as the input to the lexer/parser. The preprocessor can be completely unaware of what lexer/parser it is feeding into, and the lexer/parser can be unaware of the fact that the code was preprocessed (although it might handle directives like `#line` to give better error messages).

Unfortunately, the C preprocessor makes refactoring very difficult; it is the subject of Garrido's entire Ph.D. thesis [40]. The compiler's approach of preprocesing the code first and then parsing it does not work. Since refactoring tools must modify the *user's* source code, they must be able to parse, analyze, and transform code *with embedded preprocessor directives.*

This dissertation will not treat the topic of refactoring preprocessed code in detail, but suffice it to say that a compiler's AST will almost certainly be inadequate. At a minimum, a refactoring must be able to determine which AST nodes correspond to actual text in the user's source code and which

ones are the result of file inclusions and macro expansions. For more details, see Overbey et al. [72] and Garrido [40].

2.3 Reusing Existing Front Ends

2.3.1 When Reuse Succeeds

Since the parser and AST constitute a large part of the infrastructure problem for a refactoring tool, it makes sense to reuse an existing front end, if possible. This has been done successfully in a number of refactoring tools.

Integrated development environments often contain a parser and AST to support features like source code navigation and language-based searching. Usually, these features require fairly precise source-location information, making the AST a good candidate for use in a refactoring engine. Since a refactoring tool needs to be integrated into the programmer's development environment anyway, reusing the IDE's existing parser and AST is especially desirable. Peter Sommerlad's group at the Institute for Software [46] has used this approach quite successfully, added refactoring support onto the Eclipse IDEs for C/C++ [41], Ruby [27], Python, and Groovy [55].

Other tools have been built by reusing a stock parser and AST. Xrefactory adds C/C++ refactoring support to *emacs* based on a compiler front end from the Edison Design Group [93]. Refactoring support in NetBeans was built on the front end from Sun's javac compiler [14]. The JastAddJ compiler was similarly modified to support refactoring [84]. Refactoring support in Apple Xcode 3.0 was built using a stock Objective-C parser, although an AST had to be constructed specifically for the purpose of refactoring [16].

2.3.2 When Reuse Forces Compromise

Unfortunately, reuse of a front end is not a panacea. Often, it forces some compromises to be made.

One case where this compromise is visible to the end user involves the C preprocessor. The refactorings in the Eclipse C/C++ Development Tools (CDT) support only a single configuration of the C preprocessor, so if one block of code is guarded by `#ifdef WIN32` and another by `#ifdef LINUX`, it will only refactor one of them. This is directly attributable to the fact that the CDT's preprocessor and AST were designed to support only a single preprocessor configuration (indeed, that is acceptable for most IDE functionality); modifying it to support multiple configurations was estimated to require at least eight person-months of effort [76].

More often, the compromises are internal: the design and code quality suffer as refactoring-specific concerns are hacked onto components that were never designed to support them. (This is ironic, since the purpose of refactoring is to prevent this from happening.) For example, when Peter Sommerlad's group [46] added refactoring support to several existing Eclipse plug-ins, they frequently found that the original developers were unwilling to change the ASTs to accommodate refactoring support, forcing them to use various workarounds to avoid changing those components.

Another case where code quality is compromised is in how source code modification is performed. When an existing AST is used, usually source code is modified by using the offset/length information for the AST nodes to directly manipulate the source code. For example, to delete a

function, the refactoring would look up the offset/length for that function's AST node, expand that region to include any comments and whitespace surrounding the function, and then delete those characters from the source code. In contrast, the ASTs described in the next chapter allow the same functionality to be achieved by literally removing the function node from the AST—one line of code. This is because they are *designed* for the express purpose of supporting source code manipulation.

2.3.3 When Reuse Fails

Unfortunately, there are many cases when reusing an existing front end does not work at all. Of course, these cases are not documented in the literature, since the tools never materialized. But, based on the author's experience and discussions with others working in this area, there are several reasons why this happens.

Often, reuse fails for reasons that have nothing to do with refactoring. The front end might not be available under a suitable license. It might be written in the wrong language. It might be too difficult to understand and modify. It might have bugs that are exacerbated when it is used for refactoring. (These are the reasons why Photran did not use, respectively, the EDG parser, the GNU Fortran parser, the Open64 parser, and an M.S. student's parser.)

When a front end cannot be reused for a reason that is refactoring-specific, it is almost always because the parser outputs an intermediate representation (IR) that is too far removed from the original source code, and modifying the front end to correctly map the IR to source locations is impractical. This is particularly true when the IR is a lowered program representation like three-address code.

In these cases, one possibility for reuse is to decouple the parser from the existing program representation and use it to build an AST instead. Likewise, if a parser is part of an interpreter—this is the case with PHP's Zend Engine, for example—it may only be possible to reuse the parser if it can be completely decoupled from the interpreter and used to build an abstract syntax tree instead. Of course, the feasibility of this approach decreases dramatically if the parser changes its behavior based on the code it has already interpreted.

2.4 Grammar Reuse

Compiler front ends have one unique property that most other software components do not: Often, the lexer and parser are generated code. This opens up another possibility for reuse: The *grammar* supplied to the parser generator can be reused. It can serve as the input to a different tool which will generate a new parser and AST that are specifically designed to facilitate source code manipulation. This is the approach that will be used in the next chapter.

There are disadvantages to this approach, of course. It results in a completely new infrastructure that will probably not be compatible with, or easily interoperable with, the existing front end. The generator will only construct a parser, AST, and source rewriting infrastructure; symbol tables, a type checker, and other semantic analyses will still need to be coded by hand.

But this approach also has a number of advantages. It is easy to construct grammars for small languages; grammars exist for most larger languages; and, if an existing tool contains a grammar, it is usually easy to find and co-opt with little or no understanding of the original tool. And,

as mentioned above, the “new” front end will include an AST specifically designed for source rewriting.

So, when an existing front end can be reused as-is, or with slight modification (to support the requirements listed in Section 2.2), that is probably the best approach. But when reuse is not possible, or when it will be extremely time consuming, grammar reuse is a very appealing option.

3

Annotating Grammars to Generate Rewritable ASTs

3.1 Introduction[†]

This chapter describes a system that uses a grammar to generate abstract syntax tree (AST) node classes as well as a parser that constructs ASTs comprised of these nodes. The grammar can be *annotated* to customize the AST design. The generated ASTs are *rewritable* in the sense that refactorings and other source code transformations can be coded by restructuring the AST, and the revised source code can be emitted, preserving all of the user's formatting, including spacing and comments; there is no need to develop a prettyprinter. Moreover, while the generated AST nodes are usually comparable to hand-coded AST nodes, they can be customized, replaced, or intermixed with hand-coded AST nodes. This AST generation system has been implemented in a tool called Ludwig and has been used to generate rewritable ASTs for several projects, most notably Photran, an open source refactoring tool for Fortran.

The remainder of this chapter is organized as follows. §3.2 discusses existing systems for generating ASTs, and why a different approach is needed. §3.3 describes how to annotate a parsing grammar to produce a satisfactory AST. The system of annotations described in this section is new and is fundamentally different from (and more concise than) existing AST specification languages. §3.4 describes how to augment an AST to allow both rewriting and faithful reproduction of the original source code. Although it is simple, this process (called "concretization") also appears to be new. §3.5 describes the author's implementation, including the API implemented by the generated AST.

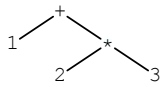
3.2 The State of the Art

There are several systems that generate abstract syntax trees. This section will look at three such tools: ANTLR, LPG, and Zephyr. None of them is ideal for use in an automated refactoring tool. However, a careful look at their advantages and disadvantages will help motivate the grammar annotation system proposed later in this chapter, which shares many of the desirable properties of these systems while avoiding the properties that make these systems difficult to use for implementing refactoring tools.

[†]Portions of this chapter are based on "Generating Rewritable Abstract Syntax Trees: A Foundation for the Rapid Development of Source Code Transformation Tools" [71], which describes an earlier version of this work.

3.2.1 ANTLR

ANTLR [1] is a widely-used parser generator. One of ANTLR's unique features is that it allows the user to easily build an abstract syntax tree by annotating the productions in a grammar. ANTLR's ASTs are, essentially, trees of tokens: Each subtree has a token as its root and a list of trees as its children. For example, in an expression grammar, the input `1 + 2 * 3` might produce the AST

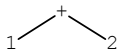


where the five input tokens are simply rearranged into a tree structure.

There are several ways to tell ANTLR what AST node should be built for each production in the grammar. One is particularly concise: A circumflex (^) is appended to a terminal symbol in the production to form a node with that token as the root. For example,

```
expr :  INT '+'^ INT ;
```

would result in ASTs like the following.



So, what can one do with such as AST? ANTLR has a second kind of input file, called a *tree grammar,* which is used to generate a *tree parser.* In a tree grammar, the user specifies patterns in the AST and a semantic action (i.e., Java code) to execute for each pattern. The tree parser, then, traverses the AST and, when it matches a pattern, it executes the corresponding semantic action.

Tree parsers are ideal for translation/code generation, prettyprinting, type checking—most of the usual things a compiler does. They can even be used for code instrumentation [74, Ch. 9]. In general, tree parsers work well for tasks that are *syntax-directed.*

Unfortunately, tree parsers are not quite ideal for a refactoring tool because refactorings are *not* syntax-directed: It is much more natural to describe them imperatively. Consider the refactoring Extract Local Variable. Even in its simplest form, the refactoring must (1) identify what expression the user selected, (2) compute its type, (3) insert a new local variable declaration, (4) insert an assignment to the new variable, and (5) replace the selected expression with a use of the variable. More than likely, this would require several tree parsers glued together with some hand-written code.

3.2.2 LPG

In Extract Local Variable, like most refactorings, tree traversals and pattern matching are relatively minor steps in a larger, imperative process. So it makes sense to implement a refactoring tool in a language like Java or C++, where the AST is a data structure that can be accessed, traversed, and manipulated in a way that is idiomatic for that language.

LPG [4] is a parser generator which also generates ASTs, but it is quite different from ANTLR. LPG generates one Java class for each production in the grammar. These classes form the nodes of the AST. It also generates an interface for each nonterminal symbol; the node classes for that symbol's productions all implement that interface. It also generates a Visitor, which can be used to traverse ASTs. Finally, it generates a parser which constructs ASTs from these nodes at runtime.

In a refactoring tool, LPG's ASTs are probably a better choice than ANTLR's. Since LPG's AST nodes are ordinary Java classes, they are a lot like the AST nodes one would code by hand. They can be traversed using the Visitor pattern. The type of a node is determined by its class, not by pattern matching. A separate domain-specific language is not needed to write traversals. In general, they are more natural to work with.

Unfortunately, the heuristic used to determine what AST node classes (and interfaces) to generate is imperfect. An abstract syntax tree should represent the *essential* syntactic structures, abstracting away unimportant aspects of the concrete syntax. For example, tokens like *if*, *endif*, and *for* can often be omitted from an AST. Most unit productions in a grammar do not warrant separate AST nodes. Writing an unambiguous expression grammar generally involves introducing superfluous nonterminal symbols that exist only as a means to enforce associativity and precedence rules; these need not translate into superfluous AST nodes.

Since such deviations from the concrete syntax are expected and desirable, the developer needs to be given more control over the AST structure. This is one point where Zephyr excels.

3.2.3 Zephyr

The Zephyr abstract syntax description language [91] is a domain specific language for abstract syntax tree nodes. A Zephyr input file resembles a declaration of an algebraic data types in ML.

$$\begin{aligned} \text{exp\_list} &= \textbf{ExpList}(\textit{exp, exp\_list}) \mid \textbf{Nil} \\ \text{exp} &= \textbf{Num}(\textit{int}) \end{aligned}$$

Zephyr reads an input file, such as the one above (excerpted from [91]), and uses it to generate AST nodes in one of several languages. In Java, the AST nodes are classes; in C, they are structs and unions (along with allocation/initialization functions); in ML, they are algebraic data types.

Zephyr gives the developer much more control over the structure of the AST than LPG does. But this comes at a cost. Zephyr is not integrated into a parser generator. Its input file is *completely separate* from the grammar. Zephyr has no knowledge of how its AST nodes relate to grammatical productions; the developer is responsible for making the parser build the desired AST from the nodes that Zephyr generates.

In total, the developer must supply Zephyr with a complete description of every AST node, and he must hand-code the parser actions to build ASTs from the nodes that Zephyr generates. So, while using Zephyr to generate AST nodes requires less work than hand-writing AST node classes in Java, it requires roughly the same amount of code it would require in ML. In other words, it requires a lot more work than ANTLR or LPG required.

3.2.4 A New Approach

ANTLR, LPG, and Zephyr all have different ways of determining what ASTs are generated. Each has advantages and disadvantages. ANTLR's approach of annotating the grammar is surprisingly concise and easy to learn, but the ASTs are nothing like what a human would code. LPG produces an AST instantly from a grammar; its ASTs look more like hand-written code, but their structure is too closely tied to the grammar. Zephyr's ASTs are highly customizable and are the most natural to work with, but the user has to specify the AST structure in its entirety.

This chapter advocates a new approach that combines the best parts of all three approaches. Like LPG, it can generate an AST instantly from a grammar; no additional work is required. It generates Java code that resembles what a developer would write by hand. Like Zephyr, the AST structure is highly customizable. When customization is needed, it can be done very concisely by annotating the grammar, like ANTLR. In fact, these annotations can be added iteratively to refine the AST design over time, as refactorings use more and more of the AST nodes. And, perhaps most importantly, the generated ASTs are specifically designed to be used for source code manipulation in a refactoring tool; they implement an API that makes source code modification surprisingly easy.

3.3 Abstract Syntax Annotations

The grammar supplied to a parser generator defines the *concrete* syntax of the language, which is almost always different from an ideal *abstract* syntax. For example, consider the following grammar for a language where a program is simply a list of statements, and a statement is either an if-statement, an unless-statement, or a print-statement.

‹*program*›	::=	‹*program*› ‹*stmt*› | ‹*stmt*›	(1)
‹*stmt*›	::=	‹*if-stmt*› | ‹*unless-stmt*› | ‹*print-stmt*›	(2)
‹*if-stmt*›	::=	IF ‹*expr*› THEN ‹*stmt*› ENDIF	(3)
	|	IF ‹*expr*› THEN ‹*stmt*› ELSE ‹*stmt*› ENDIF	(4)
‹*unless-stmt*›	::=	UNLESS ‹*expr*› ‹*stmt*›	(5)
‹*print-stmt*›	::=	PRINT ‹*expr*›	(6)
	|	PRINT ‹*expr*› TO STDOUT	(7)
	|	PRINT ‹*expr*› TO STDERR	(8)
‹*expr*›	::=	TRUE | FALSE | LITERAL-STRING | NOT ‹*expr*›	(9)

There are several ways to generate an AST directly from the grammar. Clearly, such an AST will not have an ideal design, but it is useful as a starting point; this section will describe seven annotations which can be used to refine the AST's design. These annotations were developed over time, based on the author's experience working with grammars for many different languages. The AST nodes we generate will be classes comprised of public fields, although adapting the technique to generate properly-encapsulated classes or even non-object-oriented ASTs (e.g., using structs and unions in C or algebraic data types in ML) is straightforward.

One obvious method for generating an AST directly from the grammar is to

- generate one AST node class for each nonterminal, where
- this class contains one field for each symbol that occurs on the right-hand side of one of that nonterminal's productions.

For example, the AST nodes corresponding to ‹*print-stmt*› and ‹*stmt*› would be the following, where *Token* is the name of a class representing tokens returned by a lexical analyzer.

```
class PrintStmtNode {
  public Token print;
  public ExprNode expr;
  public Token to;
  public Token stdout;
  public Token stderr;
}
```

```
class StmtNode {
  public IfStmtNode ifStmt;
  public UnlessStmtNode unlessStmt;
  public PrintStmtNode printStmt;
}
```

When these classes are instantiated in an AST, irrelevant fields will be set to `null`.

3.3.1 Annotation 1: Omission

The aforementioned method for generating ASTs directly from a grammar has several things wrong with it. One of the most obvious is that keywords like IF and PRINT are almost never included in an AST. We will indicate that these tokens can be *omitted* or *elided* by ~~striking out~~ their symbols in the grammar.[1] For example,

⟨*print-stmt*⟩ ::= ~~PRINT~~ ⟨*expr*⟩
| ~~PRINT~~ ⟨*expr*⟩ ~~TO~~ STDOUT
| ~~PRINT~~ ⟨*expr*⟩ ~~TO~~ STDERR

would generate a *PrintStmtNode* with only three fields: `expr`, `stdout`, and `stderr`.

Generalizing this slightly, we will establish the following rules.

- If a symbol on the right-hand side of a production is annotated for omission, no field is generated for that symbol.
- If a nonterminal on the left-hand side of a production is annotated for omission, no AST node class is generated for that nonterminal.[2]

3.3.2 Annotation 2: Labeling

Determining the names of AST classes and their fields from the names of nonterminal and terminal symbols in the grammar is often sufficient, but sometimes these names need to be customized. This can be done by *explicitly labeling* symbols in the grammar and interpreting these labels as follows.

- Labeling the nonterminal on the left-hand side of a production determines the name of the AST node class to generate.
- Labeling a nonterminal or terminal symbol on the right-hand side of a production determines the name of the field to which that symbol corresponds.

The idea of labeling symbols is simple, yet it is extremely powerful, having several uses and implications.

Labeling to Distinguish Fields

Labeling is the only annotation that is strictly required, at least in certain cases. *IfStmtNode* is one example. The problem is production (4):

⟨*if-stmt*⟩ ::= IF ⟨*expr*⟩ THEN ⟨*stmt*⟩ ELSE ⟨*stmt*⟩ ENDIF

Since the symbol ⟨*stmt*⟩ appears twice in the same production, we need *two* fields so that the node can have separate fields for the then-statement and the else-statement. We will accomplish this by adding distinctive labels to these symbols in the grammar, rewriting productions (3) and (4) to generate distinct fields for the two occurrences of ⟨*stmt*⟩.

[1] Ludwig uses an ASCII notation, prefixing the symbol with a hyphen and a colon, as in `-:print`.

[2] The ability to omit entire AST node classes is useful when only a partial AST is desired. For example, the Eclipse JDT [3] and CDT [2] both contain a "lightweight" AST which describes high-level organizational structures (classes, methods, etc.) but not statements or expressions.

$$\begin{array}{lcl} \langle\, \textit{if-stmt} \,\rangle & ::= & \texttt{IF}\ \langle\, \textit{expr} \,\rangle\ \texttt{THEN}\ \overset{\text{thenStmt}}{\langle\, \textit{stmt} \,\rangle}\ \texttt{ENDIF} \\ & | & \texttt{IF}\ \langle\, \textit{expr} \,\rangle\ \texttt{THEN}\ \overset{\text{thenStmt}}{\langle\, \textit{stmt} \,\rangle}\ \texttt{ELSE}\ \overset{\text{elseStmt}}{\langle\, \textit{stmt} \,\rangle}\ \texttt{ENDIF} \end{array}$$

This will generate the following node instead.

```
class IfStmtNode {
  public ExprNode expr;
  public StmtNode thenStmt;
  public StmtNode elseStmt;
}
```

Labeling to Merge Fields

Just as we can give symbols distinct labels to create distinct fields, we can also give several several symbols the *same* label to assign them to the *same* field . . . as long as they occur in different productions. One example of this appears above: Since both occurrences of ‹*stmt*› were given the label *thenStmt*, the `thenStmt` field will be populated regardless of whether production (3) or (4) was matched.

Labeling to Rename Fields

Labeling can also be used to avoid illegal or undesirable field names. For example, were the IF token not omitted, it would generate a field named `if`, which is illegal in Java, so it could be labeled *ifToken* instead. Similarly, ‹*expr*› might be labeled *guardingExpression* to make its corresponding field name more descriptive.

Labeling to Distinguish, Rename, and Merge Node Classes

Just as labeling the symbols on the right-hand sides of productions allows us to distinguish, rename, and merge fields, labeling the nonterminals on the left-hand sides of productions allows us to distinguish, rename, and merge AST node classes.

The text of the label assigned to a left-hand nonterminal determines the name of the AST node class generated for that nonterminal. By assigning a distinct label to each left-hand nonterminal, we ensure that a different AST node class will be generated for each nonterminal. By assigning the same label to several left-hand nonterminals, they can all correspond to the same AST node class.

But when is it useful to have just one node class correspond to several nonterminals?

Sometimes a grammar uses several nonterminals to refer to the same logical entity. This is probably most common in expression grammars, as we will see later, but we can illustrate it with the sample grammar. Note that our sample programming language contains two conditional constructs: an if-statement and an unless-statement. Suppose we want to represent both of these with a single node, *ConditionalStmtNode*. We will label the *left-hand* nonterminals with this name

$$\begin{array}{lcl} \overset{\text{ConditionalStmtNode}}{\langle\, \textit{if-stmt} \,\rangle} & ::= & \texttt{IF}\ \langle\, \textit{expr} \,\rangle\ \texttt{THEN}\ \overset{\text{thenStmt}}{\langle\, \textit{stmt} \,\rangle}\ \texttt{ENDIF} \\ & | & \texttt{IF}\ \langle\, \textit{expr} \,\rangle\ \texttt{THEN}\ \overset{\text{thenStmt}}{\langle\, \textit{stmt} \,\rangle}\ \texttt{ELSE}\ \overset{\text{elseStmt}}{\langle\, \textit{stmt} \,\rangle}\ \texttt{ENDIF} \\ \overset{\text{ConditionalStmtNode}}{\langle\, \textit{unless-stmt} \,\rangle} & ::= & \texttt{UNLESS}\ \langle\, \textit{expr} \,\rangle\ \overset{\text{elseStmt}}{\langle\, \textit{stmt} \,\rangle} \end{array}$$

in order to generate a *ConditionalStmtNode* class with the same three fields as before (cf. page 23). When an if-statement is matched, the `expr` and `thenStmt` fields will be set, and the `elseStmt` field

may or may not be `null`. When an unless-statement is matched, `expr` and `elseStmt` will be set, but `thenStmt` will always be `null`.[3]

3.3.3 Annotation 3: Type/Value

Consider the print-statement, defined in productions (6), (7), and (8). Suppose that a print-stmt can write either to standard output or standard error. Although we can omit the PRINT and TO tokens from the *PrintStmtNode* class, we cannot omit the STDERR token without losing the semantic distinction between the two variants of the print-statement. The token STDERR is not important *per se;* what matters is whether or not it was present at all. Rather than storing the STDERR token, we would like the AST node class to contain a Boolean field, *stderr.*

We will annotate a symbol with *(type=X)* to indicate that its field should be of type `X` rather than of type `Token`. For example, if a token were annotated with *(type=IASTNode)*, the token would be stored in a field of type *IASTNode* rather than a field of type *Token*.

When an AST node cannot be assigned to a field of the given type (e.g., if the type is `boolean`), we must supply a *value* in the annotation, as in *(type=boolean,value=true)* or *(type=int,value=5)*. In fact, the `boolean` type occurs frequently enough that we have a shorthand notation for it: either *(bool)* or *(bool,value=true)* can be used as a shorthand for *(type=boolean,value=true)*.

In the example grammar, we can use the Boolean annotation on the STDERR token.[4] When the corresponding token is present, the field's value will be set to true; otherwise, it will be set to false.

$$\begin{array}{lcl} \langle \textit{print-stmt} \rangle & ::= & \sout{\textsc{print}}\ \langle \textit{expr} \rangle \\ & | & \sout{\textsc{print}}\ \langle \textit{expr} \rangle\ \sout{\textsc{to}}\ \overset{\substack{\text{stderr}\\ \text{(bool,value=false)}}}{\textsc{stdout}} \\ & | & \sout{\textsc{print}}\ \langle \textit{expr} \rangle\ \sout{\textsc{to}}\ \overset{\substack{\text{stderr}\\ \text{(bool,value=true)}}}{\textsc{stderr}} \end{array}$$

will generate the node class

```
class PrintStmtNode {
  public ExprNode expr;
  public boolean stderr;
}
```

whose `stderr` field can be tested to determine whether the print-statement is intended to write to standard output or standard error. Note that, if the print-statement contains neither a TO STDOUT or TO STDERR clause, the Boolean field will default to `false`, which is equivalent to printing TO STDOUT.

3.3.4 Annotation 4: List Formation

Recursive productions are idiomatically used to specify lists. In the example grammar, the productions on line (1) indicate that a program is a list of one or more statements. In an AST, it is usually preferably to replace these recursive structures with an array, list, or whatever iterable construct is most common in the implementation language.

We will annotate left-hand nonterminals with *(list)* to indicate that the productions for that nonterminal describe a list.

$$\overset{\text{(list)}}{\langle \textit{program} \rangle} \quad ::= \quad \langle \textit{program} \rangle\ \langle \textit{stmt} \rangle \mid \langle \textit{stmt} \rangle$$

[3] This is based on the intuition that **unless** *E S* is equivalent to **if** *E* **then no-op else** *S*.

[4] Again, Ludwig uses a straightforward ASCII equivalent: `(bool):stderr`.

Now, there is no need to generate a *ProgramNode* class: In its place, we can simply use a `List<StmtNode>`.

3.3.5 Annotation 5: Superclass Formation

Idiomatic Form

The productions in line (2) illustrate another common idiom in BNF grammars:

$$\langle stmt \rangle \quad ::= \quad \langle if\text{-}stmt \rangle \mid \langle unless\text{-}stmt \rangle \mid \langle print\text{-}stmt \rangle$$

states that an if-statement is a statement, an unless-statement is a statement, and a print-statement is a statement. In object-oriented languages, this *is-a* relationship is generally modeled using inheritance. Instead of generating a *Stmt* node with fields for the various types of statements, we can instead make *Stmt* an abstract class (or interface, in Java or C#) which is *subclassed* by *IfStmt, UnlessStmt,* and *PrintStmt.* We will indicate this preference by a *(superclass)* annotation on the left-hand nonterminal.

$$\overset{\text{(superclass)}}{\langle stmt \rangle} \quad ::= \quad \langle if\text{-}stmt \rangle \mid \langle unless\text{-}stmt \rangle \mid \langle print\text{-}stmt \rangle$$

This generates the following.[5]

```
interface StmtNode { /*empty*/ }
class IfStmtNode implements StmtNode { ... }
class UnlessStmtNode implements StmtNode { ... }
class PrintStmtNode implements StmtNode { ... }
```

Non-idiomatic Form

As in the preceding example, the (superclass) annotation is generally applied to a nonterminal whose productions are all of the form $A ::= B$, for nonterminals A and B. But it is also possible to apply this annotation when the productions do not have this form. To do this, we must explicitly label each *production* with a node class name. For example,

$$\overset{\text{(superclass)}}{\langle if\text{-}stmt \rangle} ::=$$

$$\begin{array}{lll} & \sout{\texttt{IF}}\ \langle expr \rangle\ \sout{\texttt{THEN}}\ \overset{\text{thenStmt}}{\langle stmt \rangle}\ \sout{\texttt{ENDIF}} & \Leftarrow \text{IfThenNode} \\ \mid & \sout{\texttt{IF}}\ \langle expr \rangle\ \sout{\texttt{THEN}}\ \overset{\text{thenStmt}}{\langle stmt \rangle}\ \sout{\texttt{ELSE}}\ \overset{\text{elseStmt}}{\langle stmt \rangle}\ \sout{\texttt{ENDIF}} & \Leftarrow \text{IfThenElseNode} \end{array}$$

allows us to have two *different* nodes for an if-statement, one for the if-then form and another for the if-then-else form.

```
interface IfStmtNode { /*empty*/ }

class IfThenNode                    class IfThenElseNode
  implements IfStmtNode {             implements IfStmtNode {
  public ExprNode expr;               public ExprNode expr;
  public StmtNode thenStmt;           public StmtNode thenStmt;
                                      public StmtNode elseStmt;
}                                   }
```

[5] In Ludwig's implementation, the interface/abstract superclass contains no fields or methods, as shown here. Later in this chapter, we describe several ways to customize generated nodes. This empty interface is often an excellent candidate for customization, since certain behaviors may be common among the various subclasses.

It should be noted that the labeling principles discussed in §3.3.2 apply to production labels as well. For example, two productions (or a production and a nonterminal) can be given the same label to assign them to the same node class. For example, the ‹*expr*›-productions for TRUE and FALSE could both be assigned to the AST node class *BoolExprNode* as follows.

(superclass)
‹*expr*› ::=

	value (bool,value=true) TRUE	⇐ BoolExprNode
|	value (bool,value=true) FALSE	⇐ BoolExprNode
|	value LITERAL-STRING	⇐ LiteralExprNode
|	~~NOT~~ ‹*expr*›	⇐ NotExprNode

3.3.6 Annotation 6: Inlining

To illustrate the next annotation, suppose the productions defining an if-statement had been written as follows. Note that the syntax of the if-statement has not changed: The grammar is just slightly different.

‹*if-stmt*›	::=	‹*if-then-part*› ‹*endif-part*›
	|	‹*if-then-part*› ‹*else-part*› ‹*endif-part*›
‹*if-then-part*›	::=	~~IF~~ ‹*expr*› ~~THEN~~ thenStmt ‹*stmt*›
‹*else-part*›	::=	~~ELSE~~ elseStmt ‹*stmt*›
~~‹*endif-part*›~~	::=	ENDIF

Compare these to productions (3) and (4) in the original sample grammar. In this version, IF ‹*expr*› THEN ‹*stmt*› has been "factored out" into its own nonterminal, ‹*if-then-part*›. This is commonly done to minimize duplication in the grammar. However, left alone, it adds unnecessary nodes to an AST. In this case, there will be *three* AST nodes, all devoted to defining the structure of an if-statement: *IfStmtNode, IfThenPartNode,* and *ElsePartNode.*[6]

To create the same nodes as before, we would *like* to do away with *IfThenPartNode* and *ElsePartNode* and instead have their fields—`expr`, `thenStmt`, and `elseStmt`—placed directly into the *IfStmtNode* class. In other words, we would like to *inline* these nodes: In the *IfStmtNode* class, rather than declaring an *IfThenPartNode* field, we will simply insert all of the fields that would be in an *IfThenPartNode* instead. We will denote this with an *(inline)* annotation

‹*if-stmt*›	::=	(inline) ‹*if-then-part*› ‹*endif-part*›
	|	(inline) ‹*if-then-part*› (inline) ‹*else-part*› ‹*endif-part*›
~~‹*if-then-part*›~~	::=	~~IF~~ ‹*expr*› ~~THEN~~ thenStmt ‹*stmt*›
~~‹*else-part*›~~	::=	~~ELSE~~ elseStmt ‹*stmt*›
~~‹*endif-part*›~~	::=	ENDIF

which gives us the desired AST node. Notice that we have now omitted ‹*if-then-part*› and ‹*else-part*›: Since their contents are always inlined, there is no reason to generate these node classes.

```
class IfStmtNode {
  public ExprNode expr; // Inlined from IfThenPartNode
  public StmtNode thenStmt; // Inlined from IfThenPartNode
  public StmtNode elseStmt; // Inlined from ElsePartNode
}
```

[6]Notice that we have omitted ‹*endif-part*›, since its AST node is unnecessary.

3.3.7 Annotation 7: Extraction

Extraction is exactly the opposite of inlining: It "pushes down" part of a production into a separate AST Node. In §3.3.2, we provided one AST structure for an unless-statement. Alternatively, realizing that **unless** *E S* is equivalent to **if not** *E* **then** *S*, we could use extraction to wrap the ‹*expr*› in a *NotExpr* node, effectively representing the unless-statement as an if-statement with a negated expression.

ConditionalStmtNode: ‹*unless-stmt*› ::= ~~UNLESS~~ NotExprNode(‹*expr*›) thenStmt(‹*stmt*›)

Extraction is most often used when a token (or sequence of tokens) is contained in one production but really represents a nested syntactic construct. For example, suppose ~~PRINT~~ ‹*expr*› ~~TO~~ INTEGER (where INTEGER represents a file descriptor) were a valid statement. The AST node for this would have two children: an expression node (for ‹*expr*›) and a token (INTEGER). However, suppose we wanted to be able to treat the file descriptor (the INTEGER) as an expression in the AST. This would allow AST visitors to treat it uniformly with other expressions in the AST. To do this, we could use extraction to push the INTEGER token into an *IntConstantNode*.

‹*print-stmt*› ::= ~~PRINT~~ ‹*expr*› ~~TO~~ fileDescriptor IntConstantNode(value INTEGER)

Extraction can be combined with labeling, as shown in this example. Notice that we labeled the extraction *(fileDescriptor)*, but we also labeled the symbol(s) being extracted. This example generates the following two AST node classes.

```
class PrintStmtNode {
  public ExprNode expr;
  public IntConstantNode fileDescriptor;
}
```

```
class IntConstantNode {
  public Token value;

}
```

3.3.8 Customization

In the author's experience, these annotations—omission, labeling, type/value, list formation, inlining, extraction, and superclass formation—allow satisfactory AST nodes to be generated in the vast majority of cases. However, it is sometimes desirable to "tweak" some of the generated AST node classes or to mix them with non-generated nodes. Two of the most important AST customizations are the following.

- *The user must be able to add methods to the generated node classes.* For example, it may be desirable to add a `getType()` method to expression nodes or a `resolveBinding()` method to identifier nodes. This can be achieved using a pattern described by Vlissides [90, p. 85]: The system generates an AST node class, and the user places additional methods in a custom subclass (or superclass).

- *The user must be able to write custom AST nodes when necessary.* Sometimes, the "obvious" grammatical representation of a language construct does not satisfy the constraints of the parser generator; this can result in productions which deviate wildly from the conceptual

structures of the constructs they are intended to represent. In these cases, the user must be able to hand-code an AST node for that construct. Being able to intermix hand-coded and generated nodes is critical, because it provides the user with the flexibility of hand-coding when it is needed while alleviating the tedium and cost of developing and maintaining an entirely hand-coded infrastructure.

3.3.9 An Example

To conclude this section, we will look at how to use annotations to construct an AST for an expression grammar. In the author's experience, the expression grammar for a programming language almost always requires the most complex annotations, particularly when annotating an unambiguous grammar. This example supposes that we have an enum

enum Op { PLUS, TIMES, EXP; }

providing an enumeration of binary operators.

IExpression (superclass) ‹*expr*›	::=	‹*expr*› lhs — operator (type=Op,value=Op.PLUS) PLUS — ‹*term*› rhs	⇐ BinaryExpr	(1)
	|	‹*term*›		(2)
IExpression (superclass) ‹*term*›	::=	‹*term*› lhs — operator (type=Op,value=Op.TIMES) TIMES — ‹*factor*› rhs	⇐ BinaryExpr	(3)
	|	‹*factor*›		(4)
IExpression (superclass) ‹*factor*›	::=	‹*primary*› lhs — operator (type=Op,value=Op.EXP) EXP — ‹*factor*› rhs	⇐ BinaryExpr	(5)
	|	‹*primary*›		(6)
IExpression (superclass) ‹*primary*›	::=	‹*constant*›		(7)
	|	~~LPAREN~~ ‹*expr*› ~~RPAREN~~		(8)
‹*constant*›	::=	INTEGER-CONSTANT value		(9)
	|	REAL-CONSTANT value		(10)

The preceding grammar is a stereotypical example of an unambiguous grammar for arithmetic expressions. From lowest to highest precedence, it incorporates addition (left-associative), multiplication (left-associative), exponentiation (right-associative), and nested expressions. Annotations are used as follows.

- *Superclass Formation.* Production (1) indicates that a superclass, *IExpression,* will be generated, and that expressions of the form ‹*expr*› PLUS ‹*term*› will be parsed into a node called *BinaryExpression* which implements *IExpression.* Production (2) indicates that the AST node class for ‹*term*› will also implement *IExpression;* however, since the AST node class for ‹*term*› *is IExpression,* this has no effect. Likewise, production (7) indicates that *Constant,* the AST node class for ‹*constant*›, will implement *IExpression.* Production (8) indicates that the AST node for ‹*expr*› should implement *IExpression,* but, again, this is trivially true.

- *Labeling.* Labels are used to name the fields in *BinaryExpr;* they are used to assign the same AST node class, *IExpression,* to ‹*expr*›, ‹*term*›, ‹*factor*›, and ‹*primary*›; and they are used to assign productions (1), (3), and (5) to the same node class, *BinaryExpr.* The parentheses delineating a nested expression in (8) are omitted, so they will not be included in the AST.

- *Type/Value.* Instead of storing the PLUS, TIMES, and EXP tokens, the *BinaryExpr* node class contains a field of type *Op* whose value depends on which token was present.

Ultimately, this results in the following node classes.

```
interface IExpression { /*empty*/ }

class BinaryExpr implements IExpression {
  public IExpression lhs;
  public Op operator;
  public IExpression rhs;
}

class Constant implements IExpression {
  public Token value;
}
```

3.4 Augmenting ASTs to Support Rewriting

The previous section described a system for annotating grammars to describe AST structure. Ultimately, the goal of this system is to generate ASTs that are *rewritable*—i.e., it is possible to modify source code simply by adding, modifying, moving, and deleting nodes in the AST.

Traditionally, refactoring tools have modified source code either by prettyprinting a modified AST or by computing textual edits from an AST. Prettyprinting is a popular choice for source-to-source compilers and academic/research source code transformation tools (including Stratego/XT [50, 89], TXL [28], ASF+SDF [17], and many academic refactoring tools): Prettyprinting is straightforward, at least conceptually, and preserving comments and source formatting is usually a non-goal. On the other hand, commercial refactoring tools (including the Eclipse JDT and CDT, NetBeans, and Apple Xcode) generally use textual edits: AST nodes are mapped to offset/length regions in the source file, and the source text is changed by manipulating a text buffer directly (by specifying text to be added, removed, or replaced at particular offsets) or indirectly (by manipulating AST nodes and computing text buffer changes from the AST modifications). Prettyprinting is easier to implement in a prototype but sacrifices output quality; textual edits can produce "better" results at the expense of a more tedious implementation.

We will take a somewhat different approach, which allows the AST to be manipulated directly while maintaining the code quality of textual edits: We will add the "missing pieces"—dropped tokens, spaces, comments, etc.—back into the AST, but they will not be visible in its public interface. This will allow us, internally, to reproduce the original source text exactly using a simple traversal. Moreover, they will be tied to individual nodes in such a way that they move *with* the nodes when the AST is modified.

3.4.1 Whitetext Augmentation

If an AST contains every token in the original text, in the original order, the original source code can be reproduced almost exactly. The only pieces missing are what we refer to as *whitetext:* spaces, comments, line continuation characters, and similar lexical constructs. In order to reproduce the original text *exactly—including* whitetext—we propose the following.

- Rather than discarding whitetext, the lexical analyzer must "attach" all whitetext to exactly one token: either the token preceding it or the one following it.

- Most whitetext is attached to the *following* token.
- However, whitetext appearing at the end of a line is considered to be part of the *preceding* token, along with the carriage return/linefeed following it. Any additional whitetext beyond the carriage return/linefeed is attached to the *following* token.

The latter part of this heuristic ensures that any trailing comments on a line, as well as the carriage return/linefeed itself, are associated with the preceding token, while any subsequent indentation is associated with the token on the next line.

3.4.2 AST Concretization

Note that the concretization heuristic uses tokens to *partition* the original source text: Every character in the original text is also present in a token, every character in a token is also present in the original text, and there is no overlap between tokens. Moreover, the characters retain their original source text order.[7]

Since a whitetext-augmented token stream partitions the original source text, it can be used to reproduce exact source text. Thus, it should also be possible to reproduce source text from an AST ... as long as every token is present, in the original order. Reviewing the list of AST annotations, we can see that this is not necessarily the case:

1. Tokens may be omitted.
2. Node classes may be omitted.
3. Nodes may be replaced with constant values.
4. A node's children are assigned to named fields; however, since several productions may be assigned to a single type of node, there is not necessary a single order in which these children can be traversed to preserve the original order. (For example, consider the node generated for the productions $A ::= ab \mid ba$.)

We can overcome these problems with the following, respectively.

1. Rather than omitting tokens from node classes, store them in a *private* field, making them accessible for source text reproduction while remaining absent from the node's interface.
2. When an omitted node class is used (and not inlined), simply store a string containing the node's original text or, equivalently, a list/array of tokens.
3. Rather than storing a literal value, store the original node/token, and provide an accessor method for that field that returns the literal value.
4. If there *is* a single order in which children can be traversed to preserve the original token order, there is no problem; if no such order exists, then the node must include a field indicating the appropriate traversal order.

[7] This assumes that there is at least one token in the original file. If the original source consists solely of whitetext—say, a C program that contains only a comment—this must be treated as a special case.

One method to determine an order for printing a node's children is the following. Each production generating a particular node defines a partial order on some of that node's children. The union of these partial orders is a relation describing the ordering of all of the node's children. We may treat this relation as a directed graph and topologically sort it using an algorithm that also detects cycles [29, p. 546]. If no cycles are found, the sorted graph gives a correct (total) order for printing the node's fields; if cycles are found, no such order exists.

3.4.3 AST Rewriting

When the above changes are made, it is possible to reproduce the exact source code from which an AST was created simply by traversing the tree and outputting each token and its associated whitetext. However, such a tree is also ideal for *rewriting.* When a node is moved or removed within the tree, every token under that node—omitted or not—is moved with it, as are any comments and line endings associated with that node. Suppose, for example, that an *IfStmtNode* is moved to a new location in the AST: When the modified AST is traversed to reproduce source code, the entire if-statement—including the IF token, the line ending, and any comments—will appear at the new location within the source code.[8]

3.5 Ludwig: Experience and Implementation

This technique has been used to implement a rewritable AST generator in Ludwig, a lexer and parser generator written by the author [5]. Ludwig's AST generator has been used successfully to generate parsers and ASTs for several projects, including

- the tools discussed in Chapter 1: Photran, the prototype refactoring tools for PHP and BC, and Fujitsu's prototype parser for XPFortran [59, 66];
- a prototype refactoring tool for Lua;
- a (subset) Smalltalk interpreter; and
- the EBNF parser in Ludwig itself.

The current set of annotations was refined over six years of experience using Ludwig in these projects. The Fortran and PHP grammars were borrowed from existing tools and annotated to produce an AST. The Lua, BC, and Smalltalk grammars were based directly on published syntax specifications. Ludwig's EBNF grammar was designed from scratch.

Photran provides the most challenging test case in terms of scalability. Fortran 95 has an exorbitant amount of syntax—Photran's grammar is nearly 2,500 lines long (a similar annotated grammar for Java 1.0 is just over 600 lines)—which has made it an ideal candidate for AST generation. At the time of writing, Photran contains 329 AST node classes comprising 33,081 lines of code; the AST base classes, parser, and semantic actions comprise another 25,909 lines. In total, then, its 2,500-line annotated grammar generates nearly 59,000 lines of code.

[8] One should note that the if-statement will retain its indentation from the previous location. If the level of indentation needs to be adjusted, this must be done manually by modifying the whitetext of the tokens under the affected node.

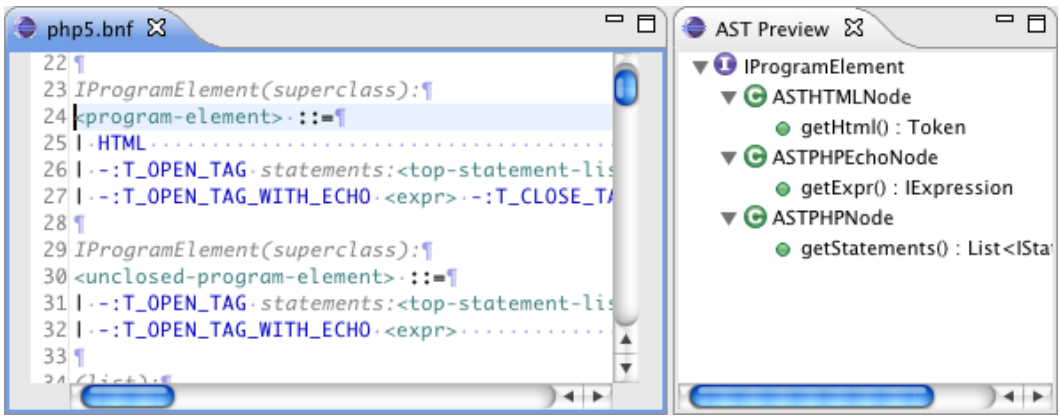


Figure 3.1: Ludwig grammar editor and AST Preview view.

3.5.1 User Interface

Ludwig can be used as a command line tool as well as an Eclipse plug-in. The command line version is useful for headless builds (e.g., when the parser/AST generator needs to be run from an Ant script). The Eclipse plug-in is most useful for grammar development.

Among other features, the Ludwig Eclipse plug-in provides a syntax highlighting editor and an AST Preview view, shown in Figure 3.1. Syntax highlighting makes it easier to distinguish annotations (shown in gray) from grammar symbols (shown in blue and cyan). When the cursor is moved over a particular production group in the editor, the AST Preview view shows the AST node class(es) that will be generated for those productions. As the grammar is annotated or modified, the AST Preview view is updated in real time. In the author's experience, this is indispensable for annotating large grammars, where the effects of a particular annotation (such as inlining) may not be immediately obvious.

3.5.2 AST Node API

Ludwig-generated ASTs have been used in Photran since about 2006. Initially, it was not obvious what application programming interface (API) the AST nodes would need to implement (besides getter/setter methods and a traversal mechanism). However, as more refactorings were developed in Photran, we constantly reviewed their code, aggressively pushing (language-independent) behavior into the Ludwig-generated AST to keep the refactorings' implementation as simple and straightforward as possible. Photran 8.0 (to be released in June 2012) will contain more than 30 refactorings. The current API implemented by Ludwig-generated ASTs is the result of five years of experience developing these refactorings, as well as many others, including refactorings for PHP and BC.

A complete description of the APIs implemented by Ludwig's generated ASTs is given in Appendix A. The following aspects of the API are perhaps the most important.

There is a common API *(IASTNode)* implemented by every AST node, including tokens and lists. It includes three categories of methods.

- *Search and traversal.* This includes methods to traverse a subtree using a Visitor, to get the

parent or children of a node, to find the nearest ancestor or the first descendent node of a particular type, and to iterate through all descendent nodes of a particular type.

- *Source manipulation.* This includes methods to delete a node, replace it with another node, replace it with a literal string, and return a deep copy of a subtree.
- *Source code reproduction and location mapping.* This includes methods to reproduce the source code from an AST node (including whitetext). It also includes methods to return the offset and length of the region of source code from which that AST node was constructed.

Every generated AST node implements this interface. Generated nodes also add getter and setter methods specific to that node.

List nodes (produced by the *(list)* annotation) implement either *IASTListNode* or *IASTSeparatedListNode* (the latter is often used for comma-separated lists). These extend both the *IASTNode* interface and the standard Java *List* interface, allowing children to be located, traversed, and modified by index.

Finally, there is also a *Token* class. This class implements *IASTNode*, but it also contains methods to get and set the token's text, the corresponding terminal symbol in the grammar, and the whitetext affixed to that token.

3.6 Conclusions

This chapter described seven *grammar annotations*—omission, labeling, type/value, list formation, inlining, extraction, and superclass formation—that allow an abstract syntax tree for a language to be defined based on the concrete syntax provided in a parsing grammar. A tool can use such an annotated grammar to generate both a parser and a rewritable AST. *Concretizing* the AST allows it to preserve the formatting of the original code even after rewriting. An AST generator has been implemented in a tool called Ludwig [5] and has been used to generate the rewritable AST in refactoring tools for Fortran, PHP, and BC; in all cases, the annotated grammar was more than an order of magnitude smaller than the generated code.

The grammar annotations presented in this chapter correspond to AST node classes in a fairly natural way, so implementing a naïve AST generator is straightforward. Unfortunately, it is easy to develop an annotated grammar that does not "make sense." For example, a node might indirectly inline itself, a field might have an ambiguous type, or a node might be simultaneously declared as both a superclass and a list. Worse, two grammar symbols might be assigned to the same field in an AST node (as in the if-statement example earlier). An AST generator must be able to detect such errors and supply the user with an informative message rather than failing or generating invalid code (indeed, tracing errors in generated code back to the offending grammar annotation(s) can be extremely difficult). In the next chapter, we will provide a formal description of how the AST generator operates and enumerate the safety checks that it must provide.

4

An Algorithm for Generating Rewritable ASTs

The previous chapter described how to annotate the grammar supplied to a parser generator so that it can generate AST node classes as well as a parser that constructs ASTs. This chapter provides a formal description of the AST generator's operation.

It is possible to annotate a grammar in a way that doesn't "make sense." As a simple example, consider the production

$$\langle \textit{if-stmt} \rangle \quad ::= \quad \text{~~IF~~} \langle \overset{\text{expr}}{\textit{expr}} \rangle \ \text{~~THEN~~} \langle \overset{\text{stmt}}{\textit{stmt}} \rangle \ \text{~~ELSE~~} \langle \overset{\text{stmt}}{\textit{stmt}} \rangle \ \text{~~ENDIF~~}$$

which would generate a node class with two fields: `expr` and `stmt`. Both the then-statement and the else-statement are assigned to the same field, `stmt`. Clearly this is a problem. The same problem exists, but is less apparent, when the grammar is written as follows.

$$\begin{aligned}
\langle \textit{if-stmt} \rangle \quad &::= \quad \langle \overset{\text{(inline)}}{\textit{if-then-part}} \rangle \ \langle \overset{\text{(inline)}}{\textit{else-part}} \rangle \ \text{~~ENDIF~~} \\
\langle \textit{if-then-part} \rangle \quad &::= \quad \text{~~IF~~} \langle \textit{expr} \rangle \ \text{~~THEN~~} \langle \overset{\text{stmt}}{\textit{stmt}} \rangle \\
\langle \textit{else-part} \rangle \quad &::= \quad \text{~~ELSE~~} \langle \overset{\text{stmt}}{\textit{stmt}} \rangle
\end{aligned}$$

In either case, if the AST generator did not detect the error, then either the then-statement or the else-statement would probably be omitted from the generated AST, since only one could be assigned to the `stmt` field. But this would likely go undetected until someone used the generated code to parse an if-then-else statement and realized that part of it was missing from the AST. By that point, it would be far from obvious that the underlying problem was a duplicated label in the annotated grammar. Problems like this are exacerbated when an annotated grammar describes a language like PHP or Fortran: The generator produces tens or hundreds of AST node classes and many thousands of lines of code, making it quite difficult to trace errors in the generated code back to problems in the annotated grammar.

Thus, it is critical that such errors be detected by the AST generator; blindly generating invalid code is not an option. By formalizing the AST construction, we can establish what it means for an annotated grammar to be *well formed,* and we can *prove* that the ASTs generated from well formed grammars will satisfy certain safety properties—for example, an AST node class will not inherit from itself, every field in the class will have a unique type, and at most one symbol in a production will be assigned to each field. To a large extent, this allows the AST generator to guarantee that, if it does not detect an error, it will generate code that compiles and builds a "correct" AST at runtime.

Overview and Organization

This chapter describes, in detail, how an AST generator operates. It provides a formal description of how AST node classes are constructed from an annotated grammar, as well as how ASTs are constructed from these nodes. (The AST generator in Ludwig is a straightforward implementation of the definitions given in this chapter.) This chapter discusses two main topics:

1. *Generating AST node classes.* This is discussed in §§4.1–4.7. Section 4.2 introduces a running example which will be used throughout the chapter. Section 4.3 formalizes the definition of an annotated grammar. Section 4.4 describes how to determine what AST node classes will be generated and what type should be associated with each grammar symbol. Section 4.5 describes how to form these classes into an inheritance hierarchy based on *(superclass)* annotations. Section 4.6 defines the fields and methods comprising each AST node class, and section 4.7 summarizes the AST node class construction algorithm.

2. *Generating a parser which builds ASTs from these nodes.* This is discussed in §§4.8–4.11.

4.1 Background: Formal Language Theory

Most of the mathematical terminology and notation used is well-known (cf. [61]). The symbol $\mathbb{Z}$ denotes the set $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ of integers, $\mathbb{N}$ the set $\{0, 1, 2, 3, \ldots\}$ of natural numbers, and $\mathbb{Z}^+$ the set $\{1, 2, 3, \ldots\}$ of positive integers. The cardinality of a set S will be denoted by $|S|$. The empty set will be denoted by $\varnothing$. Logical implication will be denoted by $\Longrightarrow$, logical negation by $\neg$, and unique existential quantification by $\exists! \cdots$. The biconditional logical connective will be denoted interchangeably by $\Leftrightarrow$, the phrase *if and only if,* and its abbreviated form *iff.* The symbol $:=$ will indicate a definition.

Strings and Alphabets

An **alphabet** is a finite, nonempty set. Given an alphabet Σ, a **string over** Σ (or simply "string" when Σ is clear from context) is a finite sequence of symbols in Σ. Given a string $w = x_1 x_2 \ldots x_n$, the **length** of w, denoted $|w|$, is equal to n. The **empty string**, denoted ϵ, is the string of length 0. Σ^* denotes the set of all strings over Σ.

Context-free Grammars

Definition. *A **context-free grammar** is a quadruple $G = (N, T, P, S)$ where*

- *N denotes a finite, non-empty set of **nonterminal symbols,***
- *T denotes a finite set of **terminal symbols,***
- *$N \cap T = \varnothing$,*
- *$P \subseteq N \times (N \cup T)^*$ denotes a finite set of **productions,** and*
- *$S \in N$ is the **start symbol** of the grammar.*

*Symbols in $N \cup T$ (i.e., terminals and nonterminals) are collectively called **grammar symbols.** For each production $(A, \alpha) \in P$, A is termed the **left-hand side (LHS)** and α the **right-hand side (RHS)**.*

When giving a concrete example of a grammar, nonterminals will be denoted by names in ‹*angle-brackets*›, while terminals will be denoted by names in SMALL-CAPS, as shown in the example below.

On the other hand, when discussing grammars in the abstract, the following conventions will be used. Capital letters toward the beginning of the alphabet (A and B) denote nonterminals, while the same letters in lowercase (a, b, c) denote terminals. Capital letters toward the end of the alphabet (X, Y, Z) denote symbols in $N \cup T$, while lowercase x, y, and z denote strings in T^*. Lower-case greek letters toward the beginning of the alphabet (α, β, γ) denote strings in $(N \cup T)^*$.

A production $(A, \alpha) \in P$ will be denoted $A ::= \alpha$. If $|\alpha| = 0$, the production will be denoted $A ::= \epsilon$. Sometimes the symbol p will be used to denote an arbitrary production in a grammar. A set of productions $\{(A, \alpha_1), (A, \alpha_2), \ldots, (A, \alpha_n)\}$ will sometimes be denoted by $A ::= \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$.

We will often use a list of productions (such as the following) as a shorthand notation for a grammar whose start symbol is the nonterminal on the left-hand side of the first production and whose terminal and nonterminal symbols can be inferred from the list of productions. For example, the following denotes a context-free grammar consisting of three[1] productions, where $N = \{$‹*expr*›, ‹*if-expr*›$\}$, $T = \{$INTEGER, IF, THEN, ELSE$\}$, and $S =$ ‹*expr*›.

‹*expr*› ::= INTEGER | ‹*if-expr*›
‹*if-expr*› ::= IF ‹*expr*› THEN ‹*expr*› ELSE ‹*expr*›

4.2 Running Example

We will use the following example grammar throughout this chapter. This annotated grammar is essentially an amalgamation of several examples from the previous chapter. We will assume code will be generated in Java; this example supposes that we have an enum

enum Op { PLUS, TIMES; }

providing an enumeration of binary operators.

(list) ‹*program*›	::=	‹*program*› ‹*stmt*› | ‹*stmt*›	
(superclass) ‹*stmt*›	::=	‹*if-stmt*› | ‹*print-stmt*›	
‹*if-stmt*›	::=	(inline) ‹*if-then-part*› (inline) ‹*endif-part*›	
~~‹*if-then-part*›~~	::=	~~IF~~ ‹*expr*› ~~THEN~~ thenStmt ‹*stmt*›	
~~‹*endif-part*›~~	::=	ENDIF	
‹*print-stmt*›	::=	~~PRINT~~ ‹*expr*›	
IExpression (superclass) ‹*expr*›	::=	lhs ‹*expr*› operator (type=Op,value=Op.PLUS) PLUS rhs ‹*term*›	⇐ BinaryExpr
	|	‹*term*›	
IExpression (superclass) ‹*term*›	::=	lhs ‹*term*› operator (type=Op,value=Op.TIMES) TIMES rhs ‹*primary*›	⇐ BinaryExpr
	|	‹*primary*›	
IExpression (superclass) ‹*primary*›	::=	value INTEGER-CONSTANT	⇐ ConstantExpr
	|	~~LPAREN~~ ‹*expr*› ~~RPAREN~~	

[1] Recall that ‹*expr*› ::= INTEGER | ‹*if-expr*› defines *two* productions, not one.

4.3 Annotated Grammars

We will begin by formalizing our notion of an annotated context-free grammar.

Definitions

Definition 1. *An **annotated grammar** is an hendecuple*

$$(N, T, P, S, I_C, I_{\mathcal{F}}, \text{nlbl}, \text{plbl}, \text{slbl}, \text{nann}, \text{sann})$$

such that

1. (N, T, P, S) *is a context-free grammar,*
2. I_C *is a nonempty set of* ***class identifiers,***
3. $I_{\mathcal{F}}$ *is a nonempty set of* ***field identifiers,***
4. $\text{nlbl} : N \to I_C$,
5. $\text{plbl} : P \to I_C$,
6. $\text{slbl} : P \times \mathbb{Z}^+ \rightharpoonup I_{\mathcal{F}}$,
7. $\text{nann} : N \to \{\textbf{\textit{generate}}, \textbf{\textit{omit}}, \textbf{\textit{list}}, \textbf{\textit{super}}, \textbf{\textit{custom}}\}$,
8. $\text{sann} : P \times \mathbb{Z}^+ \rightharpoonup \{\textbf{\textit{generate}}, \textbf{\textit{omit}}, \textbf{\textit{boolean}}, \textbf{\textit{inline}}\}$, *and*
9. *the partial function* sann *is defined on the pairs* $\{(p, i) \mid 1 \leq i \leq |p|\}$ *for every* $p \in P$.
10. *the partial function* slbl *is defined on the pairs* $\{(p, i) \mid 1 \leq i \leq |p| \wedge \text{sann}(p, i) \neq \textbf{\textit{inline}}\}$ *for every* $p \in P$.

Intuitively, the functions nlbl, plbl, and slbl assign labels to (left-hand) nonterminals, individual productions, and right-hand symbols in a production, respectively. We will assume that plbl assigns individual productions the same label as their left-hand nonterminal unless an explicit production label was given. Similarly, nann and sann assign an annotation (besides a label) to the left- and right-hand symbols in a production, respectively. Left-hand nonterminals for which "normal" AST nodes are generated and right-hand symbols for which "normal" fields will be generated are given the ***generate*** annotation. The other annotations' interpretations should be clear from the discussion in the previous chapter.

There are two differences from the previous chapter. First, this definition of an annotated grammar does not permit *(extract)* annotations. This will be discussed in Section 4.11. Also, to simplify the discussion, we are using a ***boolean*** annotation rather than the generic type/value annotation discussed in the previous chapter. Extending the construction to accommodate a more generic type/value annotation is fairly trivial.

Now, some important assumptions are implicit in this definition. Note that the nlbl is a total function: This means that *every* nonterminal is assigned *exactly one* label. Similarly, sann assigns exactly one annotation to every symbol on the right-hand side of each production. Furthermore, the sets I_C and $I_{\mathcal{F}}$ are intended to represent sets of *valid* identifiers in some programming language, so every nonterminal or production label should be a valid class name in the generated code, and every symbol label should be a valid field name.

Each of the definitions, lemmas, and theorems in the remainder of this chapter is implicitly parameterized by an annotated grammar

$$G = (N, T, P, S, I_C, I_{\mathcal{F}}, \text{nlbl}, \text{plbl}, \text{slbl}, \text{nann}, \text{sann}),$$

although we will omit this statement in the interest of brevity.

Example

Consider our running example, the grammar given in §4.2. Since the definition of an annotated grammar requires the label and annotation functions to be total, this means that every symbol in the grammar needs to be given an explicit label and an explicit annotation. For our example grammar, this will be done as follows. (For consistency, we have represented ~~omitted~~ symbols using an explicit *(omit)* annotation.)

ProgramNode (list) ‹*program*›	::=	program (generate) ‹*program*› stmt (generate) ‹*stmt*›	⇐ ProgramNode
	|	stmt (generate) ‹*stmt*›	⇐ ProgramNode
IStmt (superclass) ‹*stmt*›	::=	ifStmt (generate) ‹*if-stmt*›	⇐ IStmt
	|	printStmt (generate) ‹*print-stmt*›	⇐ IStmt
IfStmtNode (generate) ‹*if-stmt*›	::=	ifThenPart (inline) ‹*if-then-part*› endifPart (inline) ‹*endif-part*›	⇐ IfStmtNode
IfThenPartNode (omit) ‹*if-then-part*›	::=	hidden1 (omit) IF expr (generate) ‹*expr*› hidden2 (omit) THEN thenStmt (generate) ‹*stmt*›	⇐ IfThenPartNode
EndIfPartNode (omit) ‹*endif-part*›	::=	endif (generate) ENDIF	⇐ EndIfPartNode
PrintStmtNode (generate) ‹*print-stmt*›	::=	hidden3 (omit) PRINT expr (generate) ‹*expr*›	⇐ PrintStmtNode
IExpression (superclass) ‹*expr*›	::=	lhs (generate) ‹*expr*› operator (type=Op,value=Op.PLUS) PLUS rhs (generate) ‹*term*›	⇐ BinaryExpr
	|	term (generate) ‹*term*›	⇐ IExpression
IExpression (superclass) ‹*term*›	::=	lhs (generate) ‹*term*› operator (type=Op,value=Op.TIMES) TIMES rhs (generate) ‹*primary*›	⇐ BinaryExpr
	|	primary (generate) ‹*primary*›	⇐ IExpression
IExpression (superclass) ‹*primary*›	::=	value (generate) INTEGER-CONSTANT	⇐ ConstantExpr
	|	hidden4 (omit) LPAREN expr (generate) ‹*expr*› hidden5 (omit) RPAREN	⇐ IExpression

(This fully-labeled, fully-annotated version of the grammar should be referenced for the remainder of the chapter.) Unlabeled nonterminals with the ***super*** annotation have been given labels starting with *I* (e.g., ‹*stmt*› was labeled *IStmt)* since those labels will eventually become the names of Java interfaces in the generated code. Other unlabeled nonterminals have been given names ending in *-Node*, e.g., ‹*program*› has been labeled *ProgramNode*. Omitted symbols on the right-hand side have been given arbitrary, unique labels *(hidden1, hidden2,* etc.). Other right-hand symbols have been given labels matching their names—e.g., occurrences of ‹*expr*› have been labeled *expr*—

although some labels have been adjusted to ensure that they are valid (and idiomatic) identifiers (e.g., ‹*if-then-part*› has been labeled *ifThenPart*).

Note that this fully-labeled, fully-annotated grammar can be constructed automatically from the grammar given earlier. In Ludwig's implementation, the user-supplied annotated grammar is parsed, and then the "missing" labels and annotations are added to Ludwig's internal representation of the grammar.

Now, the meaning of the various functions comprising the annotated grammar is straightforward. The set of class identifiers I_C and the set $I_{\mathcal{F}}$ of field identifiers are both the set of valid Java identifiers (excluding reserved words). The function nlbl gives the (left-hand) label of a nonterminal, so nlbl(‹*program*›) = ProgramNode and nlbl(‹*primary*›) = IExpression. The function plbl gives the production label, so

$$\text{plbl}(\text{‹}\textit{program}\text{›} \ ::= \ \text{‹}\textit{program}\text{›}\ \text{‹}\textit{stmt}\text{›}) = \text{ProgramNode}$$

and

$$\text{plbl}(\text{‹}\textit{primary}\text{›} \ ::= \ \textsc{integer-constant}) = \text{ConstantExpr}.$$

The function slbl(p, i) returns the label of the i-th symbol on the right-hand side of production p, so

$$\text{slbl}(\text{‹}\textit{program}\text{›} \ ::= \ \text{‹}\textit{program}\text{›}\ \text{‹}\textit{stmt}\text{›}, 2) = \text{stmt}.$$

The annotation functions are interpreted similarly: nann(‹*program*›) = ***list*** and

$$\text{sann}(\text{‹}\textit{program}\text{›} \ ::= \ \text{‹}\textit{program}\text{›}\ \text{‹}\textit{stmt}\text{›}, 2) = \textbf{\textit{generate}}.$$

4.4 Node Classes

We will now turn to the topic of determining what AST node classes are generated from an annotated grammar. Each of the following subsections begins with definitions. These are followed by an example, and the subsection concludes with proofs of safety properties.

4.4.1 Node Classifications

Definitions

The labels given to nonterminals and productions in an annotated grammar—i.e., nlbl(A) and plbl(p)—determine the names of the AST node classes that will be generated. We will begin by classifying each node as a "normal" AST node, an omitted AST node class, a list, an abstract class/interface, or a custom (user-defined) node class.

Definition 2. *We define several* ***node classification*** *sets as follows. (Note that each of these is a set*

of labels, i.e., a subset of I_C.)

$$
\begin{aligned}
C &= C_{generate} \cup C_{omit} \cup C_{list} \cup C_{super} \cup C_{custom} \\
C_{generate} &= C_{gen1} \cup C_{gen2} \\
C_{gen1} &= \{\text{nlbl}(A) \mid A \in N \wedge \text{nann}(A) = \textbf{\textit{generate}}\} \\
C_{gen2} &= \{\text{plbl}(A ::= \alpha) \mid A \in N \wedge A ::= \alpha \in P \wedge \text{nann}(A) = \textbf{\textit{super}} \wedge \text{plbl}(A ::= \alpha) \neq \text{nlbl}(A)\}. \\
C_{omit} &= \{\text{nlbl}(A) \mid A \in N \wedge \text{nann}(A) = \textbf{\textit{omit}}\}; \\
C_{list} &= \{\text{nlbl}(A) \mid A \in N \wedge \text{nann}(A) = \textbf{\textit{list}}\}; \\
C_{super} &= \{\text{nlbl}(A) \mid A \in N \wedge \text{nann}(A) = \textbf{\textit{super}}\}; \\
C_{custom} &= \{\text{nlbl}(A) \mid A \in N \wedge \text{nann}(A) = \textbf{\textit{custom}}\}.
\end{aligned}
$$

Intuitively, each of these sets contains the names of all of the AST nodes of a particular type: regular node, abstract class/interface, etc.

Example

For our running example,

$$
\begin{aligned}
C_{gen1} &= \{\text{IfStmtNode}, \text{PrintStmtNode}\} \\
C_{gen2} &= \{\text{BinaryExpr}, \text{ConstantExpr}\} \\
C_{omit} &= \{\text{IfThenPartNode}, \text{EndIfPartNode}\} \\
C_{list} &= \{\text{ProgramNode}\} \\
C_{super} &= \{\text{IStmt}, \text{IExpression}\} \\
C_{custom} &= \emptyset
\end{aligned}
$$

4.4.2 Idiomatic Lists

Definitions

Next, we will formally define the notion of an *idiomatic list.* In our example grammar, we should not generate an AST node for the nonterminal ‹*program*› because we will use a `List<StmtNode>` object in its place. The following definition enumerates the conditions under which a nonterminal will be represented in an AST by a `List<Token>` or `List<T>` where `T` is another AST node's type.

Definition 3. *Let A be an arbitrary nonterminal. We will say that A* ***defines an idiomatic list*** *iff all of the following hold.*

1. *The grammar contains exactly two productions with A as their left-hand nonterminal (we will refer to these as "A-productions").*
2. *These two productions match one of the following eight patterns, for some $B \neq A \in N$ and $b, c \in T$.*

$$A ::= AB \mid B \quad (4.1) \qquad A ::= BA \mid B \quad (4.5)$$

$$A ::= AB \mid \epsilon \quad (4.2) \qquad A ::= BA \mid \epsilon \quad (4.6)$$

$$A ::= Ab \mid c \quad (4.3) \qquad A ::= bA \mid c \quad (4.7)$$

$$A ::= Ab \mid \epsilon \quad (4.4) \qquad A ::= bA \mid \epsilon \quad (4.8)$$

3. $\mathrm{nann}(A) =$ ***list***.
4. $\mathrm{nann}(B) \neq$ ***omit***.
5. *For each A-production p, for each* $1 \leq i \leq |p|$, $\mathrm{sann}(p, i) =$ ***generate***.

If A defines an idiomatic list, we say that A ***defines a left-recursive idiomatic list*** *when its productions match one of patterns* (4.1)–(4.4)*; A* ***defines a right-recursive idiomatic list*** *when its productions match one of patterns* (4.5)–(4.8)*; A* ***defines an idiomatic list of*** *B when its productions match pattern* (4.1)*,* (4.2)*,* (4.5)*, or* (4.6)*; and A* ***defines an idiomatic list of tokens*** *when its productions match pattern* (4.3)*,* (4.4)*,* (4.7)*, or* (4.8)*.*

Definition 4. *An annotated grammar is said to be* ***properly list-forming*** *iff, for every* $A \in N$*, if* $\mathrm{nann}(A) =$ ***list****, then A defines an idiomatic list.*

Example

In the example grammar, the nonterminal ‹*program*› defines a left-recursive idiomatic list of ‹*stmt*›; its productions match pattern (4.1).

4.4.3 Symbol Typing

Definitions

Now, we will use the node classification sets defined above to assign a *type* to each terminal and nonterminal in the grammar.

Definition 5. *The set* $\mathcal{T}$ *of* ***node types*** *is the least (infinite) set given by the following, for every* $k \in I_C$*.*

$$\frac{}{\bot \in \mathcal{T}} \qquad \frac{}{\textbf{Token} \in \mathcal{T}} \qquad \frac{}{\textbf{Boolean} \in \mathcal{T}} \qquad \frac{}{\textbf{Concrete } k \in \mathcal{T}} \qquad \frac{}{\textbf{Abstract } k \in \mathcal{T}} \qquad \frac{}{\textbf{Custom } k \in \mathcal{T}}$$

$$\frac{\tau \in \mathcal{T}}{\textbf{List } \tau \in \mathcal{T}}$$

Definition 6. *The* ***symbol typing relation*** $: \subseteq (N \cup T) \times \mathcal{T}$ *is the least relation given by the following, for every* $a \in T$ *and* $A, B \in N$*.*

$$\frac{\text{nlbl}(A) \in \mathcal{C}_{\textit{generate}}}{A : \textbf{Concrete}\ \text{nlbl}(A)} \quad (4.9) \qquad \frac{}{a : \textbf{Token}} \quad (4.12) \qquad \frac{\text{nlbl}(A) \in \mathcal{C}_{\textit{omit}}}{A : \bot} \quad (4.13)$$

$$\frac{\text{nlbl}(A) \in \mathcal{C}_{\textit{super}}}{A : \textbf{Abstract}\ \text{nlbl}(A)} \quad (4.10) \qquad \frac{\text{nlbl}(A) \in \mathcal{C}_{\textit{list}} \quad A \textit{ defines an idiomatic list of } B \quad B : \tau}{A : \textbf{List}\ \tau} \quad (4.14)$$

$$\frac{\text{nlbl}(A) \in \mathcal{C}_{\textit{custom}}}{A : \textbf{Custom}\ \text{nlbl}(A)} \quad (4.11) \qquad \frac{\text{nlbl}(A) \in \mathcal{C}_{\textit{list}} \quad A \textit{ defines an idiomatic list of tokens}}{A : \textbf{List Token}} \quad (4.15)$$

Finally, we will establish the properties that an annotated grammar should have to guarantee that every node has only one type.

Definition 7. *An annotated grammar is **class consistent** iff all of the following hold.*

1. *The grammar is properly list-forming.*
2. *For each $A, B \in N$,*
$$\text{nlbl}(A) = \text{nlbl}(B) \implies \text{nann}(A) = \text{nann}(B). \quad (4.16)$$
3. *For each $A ::= \alpha \in P$ and $B \in N$,*
$$\begin{aligned} \text{plbl}(A ::= \alpha) = \text{nlbl}(B) \implies \\ (A = B \vee \text{nann}(B) = \textit{generate}). \end{aligned} \quad (4.17)$$
4. *For each $A \in N$,*
$$\text{nann}(A) = \textit{list} \implies \forall B \neq A\ \text{nlbl}(A) \neq \text{nlbl}(B). \quad (4.18)$$

Example

The reader may verify that the example grammar is class consistent. Its symbols are typed as follows.

‹*program*› : **List Abstract** IStmt
‹*stmt*› : **Abstract** IStmt
‹*if-stmt*› : **Concrete** IfStmtNode
‹*if-then-part*› : $\bot$
‹*endif-part*› : $\bot$
‹*print-stmt*› : **Concrete** PrintStmtNode
‹*expr*› : **Abstract** IExpression
‹*term*› : **Abstract** IExpression
‹*primary*› : **Abstract** IExpression

4.4.4 Safety Properties

The following lemma asserts that every nonterminal corresponds to at least one AST node class.

Lemma 1. *For every $A \in N$,* $\text{nlbl}(A) \in \mathcal{C}$.

Proof. Observe that $\text{nlbl}(A) \in \mathcal{C}_{\text{nann}(A)}$ for every $A \in N$. □

Lemma 2. *If an annotated grammar is properly list-forming, then for every nonterminal A, exactly one of the following is true:*

- *A defines an idiomatic list of B, for some* $B \in N$.
- *A defines an idiomatic list of tokens.*
- *A does not define an idiomatic list.* □

Proof. This follows directly from Definition 3. □

Lemma 3. *If an annotated grammar is properly list-forming, then for every* $X \in N \cup T$*, there exists* $\tau \in \mathcal{T}$ *such that* $X : \tau$.

Proof. The proof that τ exists proceeds by case exhaustion. If $X \in T$, then $\tau =$ **Token** by (4.12). Otherwise, $X \in N$. By Lemma 1, $\text{nlbl}(X) \in \mathcal{C}_{\textbf{\textit{generate}}}, \mathcal{C}_{\textbf{\textit{omit}}}, \mathcal{C}_{\textbf{\textit{list}}}, \mathcal{C}_{\textbf{\textit{super}}}$, or $\mathcal{C}_{\textbf{\textit{custom}}}$. If $\text{nlbl}(X)$ is in $\mathcal{C}_{\textbf{\textit{generate}}}, \mathcal{C}_{\textbf{\textit{omit}}}, \mathcal{C}_{\textbf{\textit{super}}}$, or $\mathcal{C}_{\textbf{\textit{custom}}}$, then clearly τ exists due to (4.9), (4.13), (4.10), and (4.11), respectively. If $\text{nlbl}(X) \in \mathcal{C}_{\textbf{\textit{list}}}$, then by Lemma 2, X defines an idiomatic list of B ($\exists B \in N$) or X defines an idiomatic list of tokens. In these cases, τ must exist by (4.14) or (4.15), respectively. □

Lemma 4. *In a class consistent annotated grammar, for* $a \in \{\textbf{\textit{generate}}, \textbf{\textit{omit}}, \textbf{\textit{list}}, \textbf{\textit{super}}, \textbf{\textit{custom}}\}$*, the sets* $\mathcal{C}_a$ *are disjoint.*

Proof. By contradiction. Suppose G is a class consistent annotated grammar and there exists $\kappa \in I_C$ common to at least two of the above sets. There are $\binom{5}{2} = 10$ cases to consider.

- Case 1. Suppose $\kappa \in \mathcal{C}_{\textbf{\textit{generate}}}$ and $\kappa \in \mathcal{C}_{\textbf{\textit{omit}}}$. There are two sub-cases to consider.
  - Case 1a. Suppose $\kappa \in \mathcal{C}_{\textbf{\textit{gen1}}}$ and $\kappa \in \mathcal{C}_{\textbf{\textit{omit}}}$. Then

    $$\exists A \in N \text{ s.t. } \text{nann}(A) = \textbf{\textit{generate}} \wedge \text{nlbl}(A) = \kappa, \text{ and}$$

    and

    $$\exists B \in N \text{ s.t. } \text{nann}(B) = \textbf{\textit{omit}} \wedge \text{nlbl}(B) = \kappa.$$

    Now $\text{nlbl}(A) = \text{nlbl}(B) = \kappa$, but $\text{nann}(A) \neq \text{nann}(B)$, which violates (4.16), contradicting our assumption that G is class consistent.
  - Case 1b. Suppose $\kappa \in \mathcal{C}_{\textbf{\textit{gen2}}}$ and $\kappa \in \mathcal{C}_{\textbf{\textit{omit}}}$. Then

    $$\exists A \in N, A ::= \alpha \in P \text{ s.t. } \text{nann}(A) = \textbf{\textit{super}}$$
    $$\wedge\ \text{plbl}(A ::= \alpha) \neq \text{nlbl}(A) \wedge\ \text{plbl}(A ::= \alpha) = \kappa,$$

    and

    $$\exists B \in N \text{ s.t. } \text{nann}(B) = \textbf{\textit{omit}} \wedge \text{nlbl}(B) = \kappa.$$

    Now $\text{plbl}(A ::= \alpha) = \text{nlbl}(B) = \kappa$. If $A = B$, then implication (4.16) is violated because the nonterminal must have both ***omit*** and ***super*** annotations; if $A \neq B$, then implication (4.17) is violated because $\text{nann}(B) \neq \textbf{\textit{generate}}$. Thus, our assumption of class consistence is contradicted in both cases.
- Case 2. Suppose $\kappa \in \mathcal{C}_{\textbf{\textit{generate}}}$ and $\kappa \in \mathcal{C}_{\textbf{\textit{super}}}$. There are two sub-cases to consider.

– Case 2a. The case where $\not{k} \in \mathcal{C}_{\textbf{\textit{gen1}}}$ and $\not{k} \in \mathcal{C}_{\textbf{\textit{super}}}$. is proved similarly to Case 1a.

– Case 2b. Suppose $\not{k} \in \mathcal{C}_{\textbf{\textit{gen2}}}$ and $\not{k} \in \mathcal{C}_{\textbf{\textit{super}}}$. Then

$$\exists A \in N, A ::= \alpha \in P \text{ s.t. } \text{nann}(A) = \textbf{\textit{super}}$$
$$\wedge\ \text{plbl}(A ::= \alpha) \neq \text{nlbl}(A) \wedge\ \text{plbl}(A ::= \alpha) = \not{k},$$

and

$$\exists B \in N \text{ s.t. } \text{nann}(B) = \textbf{\textit{super}} \wedge \text{nlbl}(B) = \not{k}.$$

Now $\text{plbl}(A ::= \alpha) = \text{nlbl}(B) = \not{k}$. If $A = B$, then $\text{plbl}(A ::= \alpha) = \text{nlbl}(B) = \text{nlbl}(A)$, but, from above, $\text{plbl}(A ::= \alpha) \neq \text{nlbl}(A)$ and so we have a contradiction. If $A \neq B$, then implication (4.17) is violated because $\text{nann}(B) \neq \textbf{\textit{generate}}$. Again, our assumption of class consistence is contradicted in both cases.

- Cases 3–10. Proved similarly. □

Lemma 5. *In a class consistent annotated grammar, if $X : \tau$, then τ is unique.*

Proof. By induction on the height of the derivation tree for $X : \tau$.

- Base Case. Suppose the derivation tree has height 1. By the definition of a context free grammar, the sets of terminals and nonterminals (T and N) are disjoint; it follows, then, that rule (4.12) ensures a unique type for terminal symbols. Lemma 4 guarantees that the sets $\mathcal{C}_a$ are disjoint, so rules (4.9), (4.13), (4.10), and (4.11) also guarantee a unique type. By Lemma 2, (4.14) and (4.15) have distinct premises, similarly ensuring a unique type. Rule (4.14) cannot be the initial step in a derivation.
- Inductive Case. Suppose $\forall Y \in (N \cup T)\ \exists! \tau \in \mathcal{T}$ such that $Y : \tau$. If the derivation tree has height greater than 1, clearly the only applicable rule is (4.14). By the inductive assumption, $B : \tau$ for a unique type τ, and thus X has the unique type **List** τ. □

Theorem 1. *If an annotated grammar is class consistent, then the relation* : *is a function; that is, it associates exactly one type with each terminal and nonterminal.*

Proof. This is a direct consequence of Lemmas 3 and 5. □

4.5 Inheritance Hierarchy

Definitions

In the previous section, we established the names and types of the various AST node classes that would be generated. Now, we will build an **inherits-from** relation based on *(superclass)* annotations and production labels.

We will begin by defining what it means for a production to *have a sole RHS nonterminal,* which captures the idea of the idiomatic form of the *(superclass)* annotation presented in the previous chapter.

Definition 8. *Let $A ::= X_1 X_2 \cdots X_N$ be a production (in P), which we will denote by p. We say that p* ***has a sole RHS nonterminal*** *X_i iff all of the following hold:*

1. $\text{nann}(A) =$ ***super***,
2. $\text{plbl}(p) = \text{nlbl}(A)$,
3. $\exists i \; s.t. \; X_i \in N \wedge \text{sann}(X_i) =$ ***generate***,
4. $\forall j \neq i, X_j \in T \wedge \text{sann}(p, j) =$ ***omit***.

Definition 9. *Given an annotated grammar, the* ***inheritance relation*** **inherits-from** $\subseteq \mathcal{C} \times \mathcal{C}$ *is the least relation satisfying the following. For each $A \in N$ and for each production $A ::= \alpha \in P$,*

1. *if* $\text{plbl}(A ::= \alpha) \neq \text{nlbl}(A)$*, then*

$$\text{plbl}(A ::= \alpha) \textbf{ inherits-from } \text{nlbl}(A); \textit{ and} \tag{4.19}$$

2. *if* $\text{plbl}(A ::= \alpha) = \text{nlbl}(A)$*, $A ::= \alpha$ has a sole RHS nonterminal B, and* $\text{nlbl}(B) \neq \text{nlbl}(A)$*, then*

$$\text{nlbl}(B) \textbf{ inherits-from } \text{nlbl}(A). \tag{4.20}$$

Next, we will define the notion of a *properly inheriting* annotated grammar. This ensures that every production for a *(superclass)* nonterminal either has a production label or a sole RHS nonterminal. If it has a production label, we will generate a concrete class for that production and have it inherit from the superclass; if it has a sole RHS nonterminal, then that nonterminal's class will inherit from the superclass. Note that these are the only two cases in which a *(superclass)* annotation is legal. This definition also ensures that no class inherits from itself.

Definition 10. *An annotated grammar is* ***properly inheriting*** *iff all of the following hold for every production $A ::= \alpha \in P$.*

1. $\text{plbl}(A ::= \alpha) \neq \text{nlbl}(A) \implies \text{nann}(A) =$ ***super***. (4.21)
2. *If* $\text{nlbl}(A) =$ ***super*** *and $A ::= \alpha$ does not have a sole RHS nonterminal, then* $\text{plbl}(A ::= \alpha) \neq \text{nlbl}(A)$.
3. *For every $k \in \mathcal{C}$ it is not true that k* **inherits-from**$^+$ k.

Example

The **inherits-from** relation for the example grammar is constructed as follows. By (4.19),

BinaryExpr **inherits-from** IExpression
ConstantExpr **inherits-from** IExpression

and by (4.20),

IfStmtNode **inherits-from** IStmt
PrintStmtNode **inherits-from** IStmt

Note, in particular, that the production

$$\overset{\text{IExpression}}{\underset{\text{(superclass)}}{\langle primary \rangle}} ::= \overset{\text{hidden4}}{\underset{\text{(omit)}}{\text{LPAREN}}} \overset{\text{expr}}{\underset{\text{(generate)}}{\langle expr \rangle}} \overset{\text{hidden5}}{\underset{\text{(omit)}}{\text{RPAREN}}} \quad \Leftarrow \text{IExpression}$$

does have a sole RHS nonterminal, ‹*expr*›, but it does not satisfy (4.20) since nlbl(‹*expr*›) = nlbl(‹*primary*›) = IExpression.

4.5.1 Safety Properties

Finally, we prove that a class consistent, properly inheriting, annotated grammar will generate AST node classes with two important properties. First, no class inherits from itself, either directly or indirectly. (This would prevent the generated code from compiling.) Secondly, if a class inherits (directly or indirectly) from $\mathcal{K}'$, then $\mathcal{K}'$ has the type **Abstract** $\mathcal{K}'$. When generating Java code, this means that $\mathcal{K}'$ is an interface. In other words, a concrete node class will never inherit from another concrete node class—so there is no possibility of accidental overriding—and multiple inheritance is legal. These two properties are formalized in the following theorem.

Theorem 2. *Given a class consistent, properly inheriting annotated grammar, if* $\mathcal{K}$ **inherits-from**$^+$ $\mathcal{K}'$*, then*

1. $\mathcal{K}' \neq \mathcal{K}$*, and*
2. $\mathcal{K}'$: **Abstract** $\mathcal{K}'$.

Proof. By induction.

- Base Case. Suppose $\mathcal{K}$ **inherits-from** $\mathcal{K}'$. There are two cases to consider.
  - If (4.19) holds, then for some $A ::= \alpha \in P$,

    $$\mathcal{K} = \text{plbl}(A ::= \alpha) \textbf{ inherits-from } \text{nlbl}(A) = \mathcal{K}'.$$

    The supposition in (4.19) guarantees that $\mathcal{K}' \neq \mathcal{K}$. Since the grammar is properly inheriting, (4.21) guarantees that nann(A) = ***super***, and thus $\mathcal{K}'$: **Abstract** $\mathcal{K}'$ by (4.10).
  - If (4.20) holds, then some $A ::= \alpha \in P$ has sole RHS nonterminal B, nlbl(B) $\neq$ nlbl(A), and

    $$\mathcal{K} = \text{nlbl}(B) \textbf{ inherits-from } \text{nlbl}(A) = \mathcal{K}'.$$

    Since nlbl(B) $\neq$ nlbl(A), $\mathcal{K}' \neq \mathcal{K}$. Definition 8 guarantees that nann(A) = ***super***, and thus $\mathcal{K}'$: **Abstract** $\mathcal{K}'$ by (4.10).
- Inductive Case. Suppose the theorem holds for $\mathcal{K}$ and $\mathcal{K}'$ such that $\mathcal{K}$ **inherits-from**$^+$ $\mathcal{K}'$, and suppose furthermore that $\mathcal{K}'$ **inherits-from** $\mathcal{K}''$. By the inductive assumption, $\mathcal{K}'$ = nlbl(B) for some $B \in N$, so we need only consider the case where (4.20) holds. In this case, some $A ::= \alpha \in P$ has sole RHS nonterminal B, nlbl(B) $\neq$ nlbl(A), and

  $$\mathcal{K}' = \text{nlbl}(B) \textbf{ inherits-from } \text{nlbl}(A) = \mathcal{K}''.$$

Since nlbl(B) $\neq$ nlbl(A), $\mathcal{K}' \neq \mathcal{K}''$; by the inductive assumption, $\mathcal{K} \neq \mathcal{K}'$, and therefore $\mathcal{K} \neq \mathcal{K}''$. As before, definition 8 guarantees that nann(A) = ***super***, and thus $\mathcal{K}''$: **Abstract** $\mathcal{K}''$ by (4.10). □

4.6 Node Members

Finally, we turn our attention to constructing the fields comprising each node class. First, we note that only some classes will have fields. In particular, superclasses do not contain any fields or methods, and no class is generated for idiomatic lists, so they cannot contain fields either. Omitted classes may be inlined, so, for the time being, they *do* have fields.

Definition 11. *The set $\mathcal{C}_{\textit{fields}}$ of **classes with fields** is defined such that*

$$\mathcal{C}_{\textit{fields}} \quad = \quad \mathcal{C}_{\textit{generate}} \cup \mathcal{C}_{\textit{omit}}.$$

4.6.1 Inlining

Definitions

The inlining relation defines which node classes inline the fields of which other node classes.

Definition 12. *The **inlining relation** **inlines** $\subseteq I_C \times I_C$ is defined, for each p and i, such that*

$$\text{plbl}(p) \ \textbf{inlines} \ \text{nlbl}(p, i) \textit{ iff } \text{sann}(p, i) = \textbf{\textit{inline}} \textit{ and } p_i \in N.$$

The following property guarantees that inlining "makes sense:" Only nonterminals for generated or omitted classes may be inlined, and no class may inline itself, directly or indirectly.

Definition 13. *An annotated grammar is **properly inlining** iff for every production $p = A ::= \alpha \in P$,*

1. sann(p, i) = ***inline*** $\Longrightarrow$ $p_i \in N$, *and*
2. sann(p, i) = ***inline*** $\Longrightarrow$ (nann(A) = ***generate*** $\vee$ nann(A) = ***omit***), *and*
3. *it is not the case that c* **inlines**$^+$ *c for any c.*

Example

The reader may verify that the example grammar is properly inlining. Its inlining relation is as follows.

IfStmtNode **inlines** IfThenPartNode

IfStmtNode **inlines** EndIfPartNode

4.6.2 Member Association

Definitions

To make some subsequent definitions easier, we define the set of *generating positions* for a production to be the indices of the right-hand symbols which are *not* inlined and are not recursive symbols used to form lists. These are all the symbols which directly generate a field in the node class, whereas inlined symbols generate fields indirectly.

Definition 14. *Given a production $p = A ::= \alpha$ in an annotated grammar, the set* $\mathrm{Pos}(p)$ *of* ***generating positions*** *in p is defined such that*

$$\mathrm{Pos}(p) = \left\{ \begin{array}{ll} \{2, 3, \ldots, |\alpha|\} & \textit{if } A \textit{ defines a left-recursive} \\ & \textit{idiomatic list and } \alpha_1 = A \\ \{1, 2, \ldots, |\alpha| - 1\} & \textit{if } A \textit{ defines a right-recursive} \\ & \textit{idiomatic list and } \alpha_{|\alpha|} = A \\ \{1, 2, \ldots, |\alpha|\} & \textit{otherwise} \end{array} \right\} - \{\, i \mid 1 \leq i \leq |\alpha| \wedge \mathrm{sann}(p, i) = \textbf{\textit{inline}} \}.$$

Next, we define several more notions for inlining.

Definition 15. *Given an annotated grammar, for each $k \in C_{\textbf{\textit{fields}}}$, the set $\mathcal{F}_k$ of* ***fields in class*** *k, the* **is-associated-with** *relation, and the* **induces** *relation are the least such constructs defined according to the following. For each $p \in P$ such that* $\mathrm{plbl}(p) = k$*, and for each $i \in \mathrm{Pos}(p)$,*

1. *if* $\mathrm{sann}(p, i) \in \{\textbf{\textit{generate}}, \textbf{\textit{omit}}, \textbf{\textit{boolean}}\}$*, then*
   - $\mathrm{slbl}(p, i) \in \mathcal{F}_k$.
   - $(k, \mathrm{slbl}(p, i))$ **is-associated-with** (p, i).
   - (p, i) **induces** $(k, \mathrm{slbl}(p, i))$.
2. *if* $\mathrm{sann}(p, i) =$ ***inline*** *and $p_i \in N$, then*
   - $\mathcal{F}_{\mathrm{nlbl}(p_i)} \subseteq \mathcal{F}_k$.
   - $\forall m \in \mathcal{F}_{\mathrm{nlbl}(p_i)}.\ (k, m)$ **is-associated-with** (p, i).
   - $\forall q, j, m$ *s.t.* (q, j) **induces** $(\mathrm{nlbl}(p_i), m).\ (q, j)$ **induces** (k, m).

The **is-associated-with** and **induces** relations are illustrated in Figure 4.1. When an ASTANode is constructed for the production $A ::= B\ C$, the **is-associated-with** relation indicates that the field B is due to the nonterminal B in the production, while the field c1 is due to the (inlined) nonterminal C in the production. In contrast, the **induces** relation indicates where the value of those fields was actually assigned. The field B was assigned the AST node for B in the A-production. Since the nonterminal C was inlined, the field c1 was assigned its value earlier in the parse—it was assigned the token corresponding to the terminal c1 in the production $C ::= c1\ c2$.

These relations provide the machinery necessary to ensure that the fields in each AST node will be assigned at most once during a parse. First, we will establish what it means for a grammar to be *properly field-associated.* In essence, this guarantees that every field in an AST node is associated with at most one symbol in a given production. For example, in the production $A ::= B\ C$, the field B is associated with the RHS nonterminal B, but not C; so the field B may be assigned when the parser recognizes B, but it will definitely not be assigned during the recognition of C.

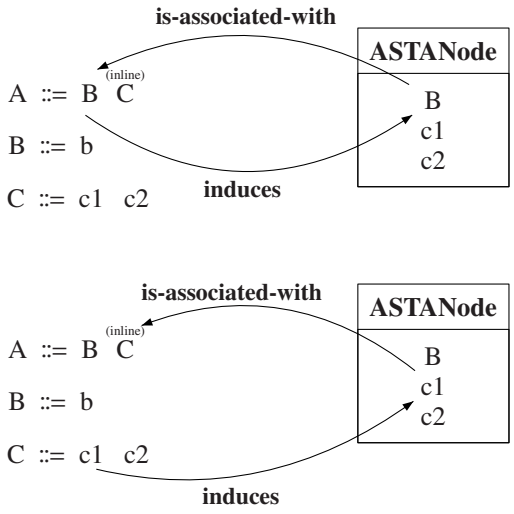


Figure 4.1: Illustration of the **is-associated-with** and **induces** relations.

Definition 16. *An annotated grammar is **properly field-associated** iff the following condition holds: If m* **is-associated-with** *(p, i) and m* **is-associated-with** *(q, j), then either*

1. *$p = q$ and $i = j$, or*
2. *$p \neq q$.*

Next, we will establish what it means for a grammar to be properly field-inducing: namely, that any field will be assigned the AST node corresponding to at most one symbol in the RHS of any given production.

Definition 17. *An annotated grammar is **properly field-inducing** iff the following condition holds: If (p, i)* **induces** *(k, m) and (p, j)* **induces** *(k, m), then $i = j$.*

Definition 18. *An annotated grammar is **uniquely assigning** iff it is both properly field-associated and properly field-inducing.*

Example

The example grammar is uniquely assigning. The sets $\mathcal{F}_k$ are as follows.

$$
\begin{aligned}
\mathcal{F}_{\text{IfStmtNode}} &= \{\text{hidden1, expr, hidden2, thenStmt, endif}\} \\
\mathcal{F}_{\text{IfThenPartNode}} &= \{\text{hidden1, expr, hidden2, thenStmt}\} \\
\mathcal{F}_{\text{EndIfPartNode}} &= \{\text{endif}\} \\
\mathcal{F}_{\text{PrintStmtNode}} &= \{\text{hidden3, expr}\} \\
\mathcal{F}_{\text{BinaryExpr}} &= \{\text{lhs, operator, rhs}\} \\
\mathcal{F}_{\text{ConstantExpr}} &= \{\text{value}\}
\end{aligned}
$$

We will not give the **is-associated-with** and **induces** relations in their entirety, but some examples are as follows.

$$\begin{aligned}
(\text{PrintStmtNode}, \text{expr}) \quad &\textbf{is-associated-with} \quad (\langle \textit{print-stmt} \rangle ::= \textsc{print}\ \langle \textit{expr} \rangle, 2) \\
(\text{IfStmtNode}, \text{hidden1}) \quad &\textbf{is-associated-with} \quad (\langle \textit{if-stmt} \rangle ::= \langle \textit{if-then-part} \rangle\ \langle \textit{endif-part} \rangle, 1) \\
(\text{IfStmtNode}, \text{expr}) \quad &\textbf{is-associated-with} \quad (\langle \textit{if-stmt} \rangle ::= \langle \textit{if-then-part} \rangle\ \langle \textit{endif-part} \rangle, 1)
\end{aligned}$$

$$\begin{aligned}
(\langle \textit{print-stmt} \rangle ::= \textsc{print}\ \langle \textit{expr} \rangle, 2) \quad &\textbf{induces} \quad (\text{PrintStmtNode}, \text{expr}) \\
(\langle \textit{if-then-part} \rangle ::= \textsc{if}\ \langle \textit{expr} \rangle\ \textsc{then}\ \langle \textit{stmt} \rangle, 1) \quad &\textbf{induces} \quad (\text{IfThenPartNode}, \text{hidden1}) \\
(\langle \textit{if-then-part} \rangle ::= \textsc{if}\ \langle \textit{expr} \rangle\ \textsc{then}\ \langle \textit{stmt} \rangle, 1) \quad &\textbf{induces} \quad (\text{IfStmtNode}, \text{hidden1}) \\
(\langle \textit{if-then-part} \rangle ::= \textsc{if}\ \langle \textit{expr} \rangle\ \textsc{then}\ \langle \textit{stmt} \rangle, 2) \quad &\textbf{induces} \quad (\text{IfThenPartNode}, \text{expr}) \\
(\langle \textit{if-then-part} \rangle ::= \textsc{if}\ \langle \textit{expr} \rangle\ \textsc{then}\ \langle \textit{stmt} \rangle, 2) \quad &\textbf{induces} \quad (\text{IfStmtNode}, \text{expr})
\end{aligned}$$

4.6.3 Member Typing and Visibility

Definitions

Now, we will determine the static type and visibility for each field in each AST node class, and we will establish a criterion which guarantees that each field has a unique type and visibility.

Definition 19. *Given a class name $k \in I_C$ and a field name $\ell \in I_{\mathcal{F}}$, the **type of field** ℓ **in class** k is given by the relation*

$$\text{type} \subseteq (\mathcal{C}_{\textbf{\textit{fields}}} \times I_{\mathcal{F}}) \times \mathcal{T}$$

which is the least relation defined as follows. For each production $p = A ::= X_1 X_2 \ldots X_n \in P$, and for each $i \in \text{Pos}(p)$,

1. *if* $\text{sann}(p, i) = \textbf{\textit{boolean}}$, *then* $\text{type}(\text{plbl}(p), \text{slbl}(p, i)) \ni \textbf{Boolean}$;
2. *if* $\text{sann}(p, i) = \textbf{\textit{omit}}$, *then* $\text{type}(\text{plbl}(p), \text{slbl}(p, i)) \ni \bot$;
3. *if* $\text{sann}(p, i) = \textbf{\textit{inline}}$, *then* $\forall \ell.\ \forall \tau.\ \ \text{type}(\text{nlbl}(X_i), \ell) \ni \tau$, $\text{type}(\text{plbl}(p), \ell) \ni \tau$;
4. *otherwise, if* $X_i : \tau$, *then* $\text{type}(\text{plbl}(p), \text{slbl}(p, i)) \ni \tau$.

Definition 20. *Given a class name $k \in I_C$ and a field name $\ell \in I_{\mathcal{F}}$, the **visibility of field** ℓ **in class** k is given by the relation*

$$\text{vis} \subseteq (I_C \times I_{\mathcal{F}}) \times \{\textbf{\textit{public}}, \textbf{\textit{private}}\}$$

which is the least relation defined as follows. For each production $p \in P$, and for each $i \in \text{Pos}(p)$,

1. *if* $\text{sann}(p, i) = \textbf{\textit{generate}}$ *or* $\text{sann}(p, i) = \textbf{\textit{boolean}}$, *then* $\text{vis}(p, i) = \textbf{\textit{public}}$, *and*
2. *if* $\text{sann}(p, i) = \textbf{\textit{omit}}$, *then* $\text{vis}(p, i) = \textbf{\textit{private}}$.

Definition 21. *Given an AST node class k and a field ℓ in that class, (k, ℓ) is said to **have a unique type and visibility** iff the following statement holds: if (p, i)* **induces** *(k, ℓ) and (q, j)* **induces** *(k, ℓ), then $\text{vis}(p, i) = \text{vis}(q, j)$ and $\text{type}(p, i) = \text{type}(q, j)$.*

Example

Again, the type and vis relations will not be given in their entirety, but some representative examples are as follows.

$$
\begin{aligned}
\text{vis}(\langle \textit{print-stmt} \rangle ::= \textsc{print}\ \langle \textit{expr} \rangle, 1) &= \textbf{\textit{private}} \\
\text{vis}(\langle \textit{print-stmt} \rangle ::= \textsc{print}\ \langle \textit{expr} \rangle, 2) &= \textbf{\textit{public}} \\
\text{type}(\langle \textit{print-stmt} \rangle ::= \textsc{print}\ \langle \textit{expr} \rangle, 1) &= \text{Token} \\
\text{type}(\langle \textit{print-stmt} \rangle ::= \textsc{print}\ \langle \textit{expr} \rangle, 2) &= \text{IExpression} \\
\text{type}(\langle \textit{if-stmt} \rangle ::= \langle \textit{if-then-part} \rangle\ \langle \textit{endif-part} \rangle, 1) &\quad \text{is not defined} \\
\text{type}(\langle \textit{if-stmt} \rangle ::= \langle \textit{if-then-part} \rangle\ \langle \textit{endif-part} \rangle, 2) &\quad \text{is not defined}
\end{aligned}
$$

4.6.4 Safety Properties

Theorem 3. *If, for every AST node class $k \in \mathcal{C}_{\textbf{\textit{fields}}}$ and every field $\ell \in \mathcal{F}_k$, it is true that (k, ℓ) has a unique type and visibility (according to Definition 21), the the relations* type *and* vis *are partial functions.* □

4.7 Summary: Generating Node Classes

The previous sections provided all of the definitions needed to be able to construct AST node classes from an annotated grammar. We can summarize this by defining the conditions under which we can safely generate an AST from an annotated grammar.

Definition 22. *An annotated grammar is* ***well-formed*** *if it is*

1. *class consistent (Definition 7),*
2. *properly inheriting (Definition 10),*
3. *properly inlining (Definition 13),*
4. *uniquely assigning (Definition 18),*
5. *every field has a unique type and visibility (Definition 21), and*
6. *the start symbol does not correspond to an omitted AST node (i.e.,* $\text{nlbl}(S) \neq$ ***omit****).*

In an AST generator, the process of generating AST node classes and checking well-formedness proceeds roughly as follows.

1. Ensure that the grammar meets the criteria in Definition 1: Every nonterminal is given a label and an annotation; every production is given a label; and right-hand symbols are given labels and annotations.
2. Check that the grammar is class consistent.
3. Build the set $\mathcal{C}$ of node classes. Mark each node with its classification: $\mathcal{C}_{\textbf{\textit{generate}}}$, $\mathcal{C}_{\textbf{\textit{omit}}}$, $\mathcal{C}_{\textbf{\textit{list}}}$, $\mathcal{C}_{\textbf{\textit{super}}}$, or $\mathcal{C}_{\textbf{\textit{custom}}}$.
4. Build the **inherits-from** relation, simultaneously checking that the grammar is properly inheriting.
5. Build the **inlines** relation, and check that the grammar is properly inlining.
6. Construct the **is-associated-with** and **induces** relations, and populate the node classes, checking that each field has a unique type and visibility.

4.8 Background: Attribute Grammars & Parser Generators

Now, we will turn our attention to constructing ASTs from the generated AST node classes. In a parser generator like Yacc [49], the user provides a grammar and supplies a snippet of code—a *semantic action*—that is executed as each production is recognized. Since Yacc "can be regarded as an implementation of simple S-attributed grammars" [73, p. 203], it follows that S-attributed grammars are a natural way to formalize the input to such a parser generator. In the next section, we will do exactly that: We will use an attribute grammar to formalize the semantic actions associated with productions in a parser generator like Yacc.

Definition 23. *(from Paakki [73]) An **attribute grammar** is an ordered triple (G, Ξ, R), where all of the following hold.*

- *$G = (N, T, P, S)$ is a context-free grammar.*
- *Ξ is a finite set of **attributes.** For each grammar symbol $X \in N \cup T$, there exists a set $\Xi_X \subseteq \Xi$ of **attributes associated with** X. $\Xi = \bigcup_{X \in N \cup T} \Xi_X$.*
- *Each set Ξ_X can be partitioned into two (disjoint) subsets: the **inherited attributes** Inh_X and **synthesized attributes** Syn_X of X. It is required that $\mathrm{Inh}_S = \varnothing$, and for all $a \in T$, $\mathrm{Inh}_a = \varnothing$.*
- *Each attribute $\xi \in \Xi$ has a **domain** denoted by $\mathrm{Dom}(\xi)$.*
- *R is a finite set of **semantic rules**. For each production $p \in P$, there exists a set $R_p \subseteq R$ of **semantic rules associated with** p; R is defined such that $R = \bigcup_{p \in P} R_p$.*
- *The synthesized attributes of terminal symbols are assumed to be defined elsewhere.*

Definition 24. *An attribute grammar G is **S-attributed** iff it contains only synthesized attributes, i.e., $\forall X \in N \cup T$. $\mathrm{Inh}_X = \varnothing$.*

Returning to the example grammar from §4.1, we could define an interpreter in Yacc using the following (excerpt).

```
/* #define YYSTYPE int */
expr    : INTEGER                       { $$ = $1; }
        | if_expr                       { $$ = $1; }
        ;
if_expr : IF expr THEN expr ELSE expr   { $$ = $2 ? $4 : $6; }
        ;
```

Assuming an obvious lexical grammar and the C language's Boolean interpretation of integers, this produces evaluations such as the following.

Input	Evaluates To
`-8`	-8
`if 0 then 111 else 222`	222
`if 2 then 111 else 222`	111
`if -999 then if 0 then 111 else 333 else 222`	333

A mathematical equivalent of such an interpreter is the S-attributed grammar defined as follows.

- G is the aforementioned grammar.

- The only attribute is a synthesized attribute VALUE with $\mathrm{Dom}(\textsc{value}) = \mathbb{Z}$.
  $\Xi := \Xi_{\langle expr \rangle} := \Xi_{\langle if\text{-}expr \rangle} := \Xi_{\textsc{integer}} := \{\textsc{value}\}$.
- We will define INTEGER.VALUE to be the integer value represented by the INTEGER terminal.
- The semantic rules are as follows. Conventionally, we have added a subscript to each symbol to avoid ambiguity.
  - For the production $\langle expr \rangle_0 ::= \textsc{integer}_1$, we define
    $$\langle expr \rangle_0.\textsc{value} := \textsc{integer}_1.\textsc{value}.$$
  - For the production $\langle expr \rangle_0 ::= \langle if\text{-}expr \rangle_1$, we define
    $$\langle expr \rangle_0.\textsc{value} := \langle if\text{-}expr \rangle_1.\textsc{value}.$$
  - For the production $\langle if\text{-}expr \rangle_0 ::= \textsc{if}_1\ \langle expr \rangle_2\ \textsc{then}_3\ \langle expr \rangle_4\ \textsc{else}_5\ \langle expr \rangle_6$, we define
    $$\langle if\text{-}expr \rangle_0.\textsc{value} := \begin{cases} \langle expr \rangle_4.\textsc{value} & \text{if } \langle expr \rangle_2.\textsc{value} \neq 0 \\ \langle expr \rangle_6.\textsc{value} & \text{otherwise.} \end{cases}$$

In this example, the first semantic rule for the attribute grammar was

$$\langle expr \rangle_0.\textsc{value} := \textsc{integer}_1.\textsc{value}.$$

In the Yacc specification, the corresponding semantic action was

```
$$ = $1.
```

The similarity of notation is intentional. Effectively, both mean, "The value of the expression is defined to be the value of the INTEGER at position 1."[2]

4.9 Preliminary Definitions

In Section 4.10, we will describe how to use an annotated grammar to construct an attribute grammar which builds a tree from AST node classes. Before we can do that, however, we will require a few definitions.

4.9.1 Lists

Definition. *Given a set T, a **list over** T is a finite sequence of elements from T. The set of all lists over T will be denoted $\mathcal{L}(T)$. The **length** of a list L is denoted by $|L|$, and the **i-th entry** in the list ($1 \leq i \leq |L|$) is denoted by L_i.*

[2]The attribute grammar defined in the next section will use the above notation for semantic rules and will not require a more formal definition. For the sake of completeness, a formal definition would be the following.

Definition. *Each **semantic rule** $r \in R_p$ is an ordered triple (i, ξ, f), subject to the following.*

- *Either $i = 0$ and $\xi \in \mathrm{Syn}_{X_0}$, or $1 \leq i \leq n$ and $\xi \in \mathrm{Inh}_{X_i}$.*
- *Suppose p has the form $X_0 ::= X_1 \cdots X_n \in P$. An attribute ξ_j is said to **occur in** p **at** i if $\xi_j \in \Xi_{X_i}$. Now suppose p has attributes $\xi_1, \xi_2, \ldots, \xi_k$ occurring at $i_1, i_2, \ldots, i_k$, respectively. Then $f : \mathrm{Dom}(\xi_1) \times \mathrm{Dom}(\xi_2) \times \cdots \times \mathrm{Dom}(\xi_k) \longrightarrow \mathrm{Dom}(\xi)$.*

Thus, the semantic rule denoted in the previous example by $\langle expr \rangle_0.\textsc{value} := \textsc{integer}_1.\textsc{value}$ is formally the ordered triple $(0, \textsc{value}, f)$ where f denotes the identity mapping $n \mapsto n$. Similarly, the semantic rule for the $\langle if\text{-}expr \rangle$-production is $(0, \textsc{value}, g)$, where $g : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ is defined by

$$g(n_1, n_2, n_3) := \begin{cases} n_2 & \text{if } n_1 \neq 0 \\ n_3 & \text{if } n_1 = 0. \end{cases}$$

By convention, lists will be written as a sequence surrounded with square brackets, e.g., $[a, b, c]$. The empty list will be denoted [], and a singleton list containing a will be denoted by $[a]$.

Definition. *Given two lists* $L_1 = [x_1, x_2, \ldots, x_m]$ *and* $L_2 = [y_1, y_2, \ldots, y_n]$*, their* ***concatenation*** $L_1 \sqcup L_2$ *is the list* $[x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n]$.

The notation $\bigsqcup_{1 \le i \le n} L_i$ will denote the concatenation $L_1 \sqcup L_2 \sqcup \cdots \sqcup L_n$. It will, at times, be helpful to regard a list L as a set, in which case L refers to the set $\{L_i \mid 1 \le i \le |L|\}$. Notations such as $x \in L$ and $L \subseteq L'$ should be interpreted in this sense.

4.9.2 Trees and Fields

In the next section, we will describe what AST nodes are constructed for each production, and what subtrees are assigned to their fields. We will use a fairly intuitive notation; for example,

AST-Node PrintStmtNode **With-Fields** [
Field hidden3 **With-Value** (**Token** PRINT)]

indicates that a PrintStmtNode should be constructed with the given PRINT token assigned to its hidden3 field and no value (null) assigned to its expr field.

Formally, we will describe what AST nodes are constructed by the parser using the following two algebraic data types.

$$
\begin{array}{rcl}
\mathit{Field} & ::= & \textbf{Field } I_{\mathcal{F}} \textbf{ With-Value } \mathit{Tree} \\
\mathit{Tree} & ::= & \textbf{AST-Node } I_{\mathcal{F}} \textbf{ With-Fields } \mathcal{L}(\mathit{Field}) \\
 & | & \textbf{Token } T \\
 & | & \textbf{List-Node } \mathcal{L}(\mathit{Tree}) \\
 & | & \textbf{Omitted-Node } \mathcal{L}(\mathit{Field}) \\
 & | & \mathcal{L}(\mathit{Tree}) \textbf{ Affixed-To } \mathit{Tree} \textbf{ Affixed-To } \mathcal{L}(\mathit{Tree})
\end{array}
$$

The meaning of each constructor should be fairly self explanatory, except for the last one. Consider the production

$$
\overset{\substack{\text{IExpression}\\ \text{(superclass)}}}{\langle \mathit{primary} \rangle} \quad ::= \quad \sout{\text{LPAREN}} \ \langle \mathit{expr} \rangle \ \sout{\text{RPAREN}}.
$$

It has a sole RHS nonterminal, ‹*expr*›, so the action of the parser should be to simply pass the AST node for ‹*expr*› upward in the tree (in Yacc notation, the action would be something like `{ $$ = $2; }`). However, since we want to be able to reproduce source code from the AST, this is insufficient; we cannot "lose" the surrounding parentheses. Our solution, then, is to *affix* these to the AST node for ‹*expr*›. In Yacc notation, this might be

```
{ $2.prependToken($1); $2.appendToken($3); $$ = $2; }.
```

In the attribute grammar, this will be denoted by

$$[\text{LPAREN}_1] \textbf{ Affixed-To } \langle \mathit{expr} \rangle_2 \textbf{ Affixed-To } [\text{LPAREN}_3].$$

The set of all node classes which may have tokens affixed is the least set computed as follows.

- If $A ::= \alpha$ is a production in the grammar with a sole RHS nonterminal B, and $|\alpha| > 1$, then nlbl(B) may have tokens affixed.

Note that some of these AST nodes may be abstract (i.e., Java interfaces). In such cases, all of their (concrete) subclasses may need to implement methods to support token affixes. The set of all subclasses (and subinterfaces) can be computed: If $\mathcal{K}$ **inherits-from*** $\mathcal{K}'$, and $\mathcal{K}'$ may have tokens affixed, then $\mathcal{K}$ may have tokens affixed.

4.10 Attributed Translation

Now, we can define how a parser constructs ASTs from generated AST nodes. We will do this by constructing an S-attributed grammar from the annotated grammar.

Conceptually, we will associate an action with each production in the grammar. These actions are (more or less) the same as what a user would write if he were manually writing actions to construct an AST. Each production will take one of three actions. (Suppose the production is $A ::= \alpha$.)

1. If the AST node for A is an abstract class/interface, or if it is a concrete node class that may be inlined, then we can simply construct (or propagate) the appropriate AST node.
2. If the AST node for A is omitted, or if it is a concrete node class that may be inlined, then we will construct a list of fields.[3] This list will be propagated upward in the tree and later used to construct a concrete AST node.
3. If the AST node for A is a list, then we will construct a list of AST nodes. This list will be propagated upward in the tree and later used to construct a **List-Node**.

To make it easier to disguish between cases 1 and 2 above, we will define a set N_{map} which contains all of the nonterminals whose productions will use action 2 above.

Definition 25. $N_{map} \coloneqq \{A \in N \mid A : \bot\} \cup \{p_i \mid p_i \in N \wedge \text{sann}(p, i) = \textbf{\textit{inline}}\}$.

Definition 26. *Given a well-formed, annotated grammar*

$$G = (N, T, P, S, I_C, I_{\mathcal{F}}, \text{nlbl}, \text{plbl}, \text{slbl}, \text{nann}, \text{sann}),$$

*the **AST construction attribute grammar** for G is the S-attributed grammar formed as follows.*

- *The underlying context-free grammar is* (N, T, P, S).
- *There are three synthesized attributes:* TREE, LIST, *and* FIELDS, *with domains* $\mathcal{T}ree$, $\mathcal{L}(\mathcal{T}ree)$, *and* $\mathcal{L}(\mathcal{F}ield)$, *respectively.*
- *For every* $a \in T$, a.TREE *is defined such that* a.TREE = **Token** a.
- *For each production* $p = A ::= p_1 p_2 \ldots p_n \in P$, *semantic rules are defined as follows. The*

[3] In Ludwig's implementation, this is actually a `HashMap<String,IASTNode>` which maps field names to the AST nodes that should be assigned to those fields.

shorthand $fields(p, i)$ *denotes the list*

$$\begin{cases} [\textbf{Field } \text{slbl}(p,i) \textbf{ With-Value } p_i.\text{TREE}] & \text{if } p_i \notin N_{map} \wedge (p_i \in T \vee \text{nann}(p_i) \neq \textbf{\textit{list}}) \\ [\textbf{Field } \text{slbl}(p,i) \textbf{ With-Value } (\textbf{List-Node } p_i.\text{LIST})] & \text{if } p_i \notin N_{map} \wedge p_i \in N \wedge \text{nann}(p_i) = \textbf{\textit{list}} \\ p_i.\text{FIELDS} & \text{if } p_i \in N_{map} \wedge \text{sann}(p,i) = \textbf{\textit{inline}} \\ [\textbf{Field } \text{slbl}(p,i) \textbf{ With-Value } (\textbf{Omitted-Node } p_i.\text{FIELDS})] & \text{if } p_i \in N_{map} \wedge \text{sann}(p,i) \neq \textbf{\textit{inline}} \wedge p_i : \bot \\ [\textbf{Field } \text{slbl}(p,i) \textbf{ With-Value} \\ \quad (\textbf{AST-Node } \text{nlbl}(p,i) \textbf{ With-Fields } p_i.\text{FIELDS})] & \text{if } p_i \in N_{map} \wedge \text{sann}(p,i) \neq \textbf{\textit{inline}} \wedge p_i \not: \bot. \end{cases}$$

1. *If* A : **Concrete** τ *and* $A \notin N_{map}$*, or*
2. *if* A : **Abstract** τ *and* A *does not have a sole RHS nonterminal,*

$$A.\text{TREE} = \textbf{AST-Node } \text{plbl}(p) \textbf{ With-Fields } \bigsqcup_{1 \leq i \leq |p|} fields(p, i)$$

3. *If* A : **Concrete** τ *and* $A \in N_{map}$*, or*
4. *if* $A : \bot$*,*

$$A.\text{FIELDS} = \bigsqcup_{1 \leq i \leq |p|} fields(p, i)$$

5. *If* A : **Abstract** τ *and* p *has a sole RHS nonterminal at position* j*,*

$$A.\text{TREE} = \begin{cases} p_j.\text{TREE} & \text{if } |p| = 1 \\ \left(\bigsqcup_{1 \leq i \leq j-1} p_i.\text{TREE}\right) \textbf{ Affixed-To } p_j.\text{TREE} \textbf{ Affixed-To } \left(\bigsqcup_{j+1 \leq i \leq |p|} p_i.\text{TREE}\right) & \text{otherwise.} \end{cases}$$

6. *If* A : **List** τ *and* p *has the form* $A ::= \epsilon$*:*

$$A.\text{LIST} = []$$

7. *If* A : **List** τ*,* A *defines an idiomatic list, and* p *has the form* $A ::= X$ *for some* $X \in (N \cup T)$*:*

$$A.\text{LIST} = [X.\text{TREE}]$$

8. *If* A : **List** τ*,* A *defines a left-recursive idiomatic list, and* p *has the form* $A ::= AX$ *for some* $X \in (N \cup T)$*:*

$$A.\text{LIST} = A.\text{LIST} \sqcup [X.\text{TREE}]$$

9. *If* A : **List** τ*,* A *defines a right-recursive idiomatic list, and* p *has the form* $A ::= XA$ *for some* $X \in (N \cup T)$*:*

$$A.\text{LIST} = [X.\text{TREE}] \sqcup A.\text{LIST}$$

10. *If* A *is the start symbol for the grammar (i.e.,* $A = S$*)...*
    (a) *If* A : **List** τ*, then define* $A_0.\text{TREE} :=$ **List-Node** L*, where* L *is the value assigned to* $A.\text{LIST}$ *(above).*
    (b) *If* A : **Concrete** τ *and* $A \in N_{map}$*, then define* $A_0.\text{TREE} :=$ **AST-Node** $\text{plbl}(p)$ **With-Fields** L*,*

where L is the value assigned to A.FIELDS *(above).*

4.10.1 Example

‹*program*›$_0$ ::= ‹*program*›$_1$ ‹*stmt*›$_2$ is attributed according to rules 8 and 10 above:

‹*program*›$_0$.LIST := ‹*program*›$_1$.LIST ⊔ [‹*stmt*›$_2$.TREE]
‹*program*›$_0$.TREE := **List-Node** (‹*program*›$_1$.LIST ⊔ [‹*stmt*›$_2$.TREE])

‹*program*›$_0$ ::= ‹*stmt*›$_1$ matches rule 7:

‹*program*›$_0$.LIST := [‹*stmt*›$_1$.TREE]

‹*stmt*›$_0$ ::= ‹*if-stmt*›$_1$ matches rule 5:

‹*stmt*›$_0$.TREE := ‹*if-stmt*›$_1$.TREE

‹*stmt*›$_0$ ::= ‹*print-stmt*›$_1$ matches rule 5:

‹*stmt*›$_0$.TREE := ‹*print-stmt*›$_1$.TREE

‹*if-stmt*›$_0$ ::= ‹*if-then-part*›$_1$ ‹*endif-part*›$_2$ matches rule 1:

‹*if-stmt*›$_0$.TREE := **AST-Node** IfStmtNode **With-Fields** (IF-THEN-PART$_1$.FIELDS ⊔ ENDIF-PART$_2$.FIELDS)

‹*if-then-part*›$_0$ ::= IF$_1$ ‹*expr*›$_2$ THEN$_3$ ‹*stmt*›$_4$ matches rule 4:

‹*if-then-part*›$_0$.FIELDS := [**Field** hidden1 **With-Value** IF$_1$.TREE,
Field expr **With-Value** EXPR$_2$.TREE,
Field hidden2 **With-Value** THEN$_3$.TREE,
Field thenStmt **With-Value** STMT$_4$.TREE]

‹*endif-part*›$_0$ ::= ENDIF$_1$ matches rule 4:

‹*endif-part*›$_0$.FIELDS := [**Field** endif **With-Value** ENDIF$_1$.TREE]

‹*print-stmt*›$_0$ ::= PRINT$_1$ ‹*expr*›$_2$ matches rule 1:

‹*print-stmt*›$_0$.TREE := **AST-Node** PrintStmtNode **With-Fields** [
Field hidden3 **With-Value** PRINT$_1$.TREE,
Field expr **With-Value** EXPR$_2$.TREE]

‹*expr*›$_0$ ::= ‹*expr*›$_1$ PLUS$_2$ ‹*term*›$_3$ matches rule 2:

‹*expr*›$_0$.TREE := **AST-Node** BinaryExpr **With-Fields** [
Field lhs **With-Value** EXPR$_1$.TREE,
Field operator **With-Value** PLUS$_2$.TREE,
Field rhs **With-Value** TERM$_3$.TREE]

‹*expr*›$_0$ ::= ‹*term*›$_1$ matches rule 5:

‹*expr*›$_0$.TREE := ‹*term*›$_1$.TREE

‹*term*›$_0$ ::= ‹*term*›$_1$ TIMES$_2$ ‹*primary*›$_3$ matches rule 2:

‹*term*›$_0$.TREE := **AST-Node** BinaryExpr **With-Fields** [
Field lhs **With-Value** TERM$_1$.TREE,
Field operator **With-Value** TIMES$_2$.TREE,
Field rhs **With-Value** PRIMARY$_3$.TREE]

$\langle term \rangle_0 ::= \langle primary \rangle_1$ matches rule 5:

$$\langle term \rangle_0.\text{TREE} := \langle primary \rangle_1.\text{TREE}$$

$\langle primary \rangle_0 ::= \text{INTEGER-CONSTANT}_1$ matches rule 2:

$$\begin{aligned} \langle primary \rangle_0.\text{TREE} \quad := \quad & \textbf{AST-Node} \text{ ConstantExpr } \textbf{With-Fields} \\ & [\textbf{Field} \text{ value } \textbf{With-Value} \text{ INTEGER-CONSTANT}_1.\text{TREE}] \end{aligned}$$

$\langle primary \rangle_0 ::= \text{LPAREN}_1 \langle expr \rangle_2 \text{ RPAREN}_3$ matches rule 5:

$$\langle primary \rangle_0.\text{TREE} := [\text{LPAREN}_1.\text{TREE}] \textbf{ Affixed-To } \langle expr \rangle_2.\text{TREE} \textbf{ Affixed-To } [\text{RPAREN}_3.\text{TREE}]$$

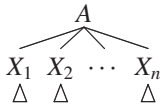
4.10.2 Safety Properties

Lemma 6. *Let p be a production in a properly field-inducing annotated grammar. If* $\text{sann}(p, i) \neq$ ***inline*** *and* $\text{sann}(p, j) \neq$ ***inline*** *and $i \neq j$, then* $\text{slbl}(p, i) \neq \text{slbl}(p, j)$*. In other words, the label given to each non-inlined symbol on the RHS is unique within that production.*

Proof. By contradiction. Suppose the antecedent holds and $\text{slbl}(p, i) = \text{slbl}(p, j) = \ell$. Then (p, i) **induces** $(\text{plbl}(p), \ell)$ and (p, j) **induces** $(\text{plbl}(p), \ell)$ but $i \neq j$, contradicting the assumption that the grammar is properly field-inducing. □

Notation. *The notation name*$(\cdot)$ *will denote a* **Field***'s name, i.e., name*(**Field** ℓ **With-Value** v) $:= \ell$.

Lemma 7. *Let $p = A ::= X_1 X_2 \cdots X_n$ be a production in a well-formed, annotated grammar and*



be a subtree of an attributed parse tree resulting from Definition 26. Let L denote the list $\bigsqcup_{1 \leq k \leq n} \mathit{fields}(p, k)$*. The following claims hold.*

(A) For every i, if $\text{name}(L_i) = \ell$, then for every $j \neq i$, $\text{name}(L_j) \neq \ell$. In other words, the name of each field in the list is unique; no two fields' names conflict.

(B) $\ell \in \mathcal{F}_{\text{nlbl}(A)}$, i.e., the field exists in the AST node class for the nonterminal on the left-hand side of the production.

Proof. (Sketch) By simultaneous induction on the height of the attributed parse tree.

- Base Cases.
  - If the attributed tree has height 0, then $p = A ::= \epsilon$, so $|p| = 0$, $L = []$, and the proposition trivially holds.
  - If the attributed tree has height 1, then $p = A ::= x$ for some $x \in T^*$. By Lemma 6, $\text{slbl}(p, i)$ is unique for each i, and L has the form $\bigsqcup_{1 \leq i \leq |p|}$ **Field** $\text{slbl}(p, i) \cdots$, so proposition (A) holds. Proposition (B) holds by Definition 15.
- Inductive Case. Suppose the attributed tree has height greater than 1 and the claims hold for all subtrees. The proof of the inductive case proceeds by a subinduction on the number of inlined symbols.

– Subbasis: No Symbols Inlined. If no symbols are inlined in p, then the claims follow from Lemma 6 and Definition 15.

– Subinductive Step: Symbols Inlined. Suppose $\text{sann}(p, i) = \textbf{\textit{inline}}$ for some i. By the induction hypothesis, the field names in $p_i.\text{FIELDS}$ do not conflict with each other. By the subinductive hypothesis, the field names in $\bigsqcup_{1 \leq k \neq i \leq n} \mathit{fields}(p, k)$ do not conflict with each other. So suppose, for the purpose of contradiction, that an inlined field name conflicts with another field name, i.e., $p_i.\text{FIELDS}$ contains a field with name ℓ and either $\text{slbl}(p, j) = \ell$ (for some $j \neq i$, $\text{sann}(p, j) \neq \textbf{\textit{inline}}$) or $p_j.\text{FIELDS}$ contains a field with name ℓ (for some $j \neq i$, $\text{sann}(p, j) = \textbf{\textit{inline}}$). In either case, by Definition 15, $(\text{nlbl}(A), \ell)$ **is-associated-with** (p, i) and $(\text{nlbl}(A), \ell)$ **is-associated-with** (p, j), contradicting the assumption that the grammar is well-formed (and, thus, properly member-associated); so claim (A) holds. Claim (B) follows from the fact that $\text{nlbl}(A)$ **inlines** $\text{nlbl}(p_i)$ and thus, according to Definition 15, $\mathcal{F}_{\text{nlbl}(p_i)} \subseteq \mathcal{F}_{\text{nlbl}(A)}$. □

In Theorem 4, we want to show that value assigned to each field is well typed. To do this, we need to define a subtype relation which captures what types of values we expect to be assigned to each type of field.

Definition 27. *The* ***subtype relation*** $<: \subseteq \mathcal{T} \times \mathcal{T}$ *is the least relation defined by the following.*

- *For every* $\tau \in \mathcal{T}$, $\tau <: \tau$.
- *For every* $\tau \in \mathcal{T}$, $\tau <:$ **Boolean**.
- *For every* $\mathcal{K} \in \mathcal{C}_{\textbf{\textit{generate}}}$, *if* $\mathcal{K}$ **inherits-from**$^+$ $\mathcal{K}'$ *for some* $\mathcal{K}'$, *then* **Concrete** $\mathcal{K} <:$ **Abstract** $\mathcal{K}'$.
- *For every* $\mathcal{K} \in \mathcal{C}_{\textbf{\textit{super}}}$, *if* $\mathcal{K}$ **inherits-from**$^+$ $\mathcal{K}'$ *for some* $\mathcal{K}'$, *then* **Abstract** $\mathcal{K} <:$ **Abstract** $\mathcal{K}'$.
- *For every* $\tau, \tau' \in \mathcal{T}$, *if* $\tau <: \tau'$, *then* **List** $\tau <:$ **List** τ'.

Two of these rules are a bit unusual. Rule 27 captures the idea that any node can be assigned to a `boolean` field (or, rather, it can be given a `boolean` accessor method). In an implementation, this may require some conversion (e.g., returning whether or not the tree is `null`), although it is not necessary to make this distinction in the type system. Rule 27 is permissible as long as lists are immutable. In the attribute grammar (and the following proofs), this is true; lists are never modified, but rather used to construct new lists. In an implementation, is it always possible to construct a mutable list of the "correct" declared type (since, for each A-production, it is known that A defines an idiomatic list of B).

Definition 28. *The* ***tree typing relation*** $: \subseteq \mathcal{T}\mathit{ree} \times \mathcal{T}$ *is the least relation defined by the following.*

$$\begin{aligned}
\textbf{AST-Node } \mathcal{K} \textbf{ With-Fields } \ell &: \textbf{Concrete } \mathcal{K} \\
\textbf{Token } a &: \textbf{Token} \\
\textbf{List-Node } L &: \textbf{List } \tau \qquad \textit{if } \forall t \in L.\ t <: \tau \\
\textbf{Omitted-Node } L &: \bot \\
L_1 \textbf{ Affixed-To } t \textbf{ Affixed-To } L_2 &: \tau \qquad \textit{if } t : \tau
\end{aligned}$$

Lemma 8. *If the attribute grammar in Definition 26 assigns a value to* $A.\text{LIST}$, *and* $A :$ **List** τ, *then*

List-Node $A.\text{LIST} <:$ **List** τ. □

Lemma 9. *If the attribute grammar in Definition 26 assigns a value to A.*TREE*, and $A : \tau$, then*

$$A.\text{TREE} <: \tau.$$

Proof. There are two cases where A.TREE is assigned.

1. A : **Concrete** $\tau \wedge A \notin N_{\text{map}}$.

   By (4.9), $\text{nlbl}(A) \in \mathcal{C}_{\textbf{\textit{generate}}}$. Since the grammar is properly inheriting, $\text{nlbl}(A) = \text{plbl}(p)$, and, thus, **AST-Node** $\text{plbl}(p)$ $\cdots$: **Concrete** τ.

2. A : **Abstract** $\tau \wedge A$ does not have a sole RHS nonterminal.

   By (4.10), $\text{nlbl}(A) \in \mathcal{C}_{\textbf{\textit{super}}}$, so **Abstract** τ = **Abstract** $\text{nlbl}(A)$. Since the grammar is properly inheriting, $\text{plbl}(p) \neq \text{nlbl}(A)$ and $\text{plbl}(p)$ **inherits-from** $\text{nlbl}(A)$. Now $\text{plbl}(p) \in \mathcal{C}_{\textbf{\textit{gen2}}} \subseteq \mathcal{C}_{\textbf{\textit{generate}}}$ and, by Definition 27, **Concrete** $\text{plbl}(p)$ <: **Abstract** $\text{nlbl}(A)$ and, hence, **AST-Node** $\text{plbl}(p)$ $\cdots$: **Concrete** $\text{plbl}(p)$ <: **Abstract** $\text{nlbl}(A)$ = **Abstract** τ. □

Lemma 10. *If the attribute grammar in Definition 26 constructs* **Field** ℓ **With-Value** v *in a parse for a production p, then $\ell \in \mathcal{F}_{\text{plbl}(p)}$.*

Proof. By the definition of $\mathit{fields}(p, i)$, $\ell = \text{slbl}(p, i)$ and $\text{sann}(p, i) \neq \textbf{\textit{inline}}$, so, by Definition 15, $\ell = \text{slbl}(p, i) \in \mathcal{F}_{\text{plbl}(p)}$. □

Lemma 11. *If the attribute grammar in Definition 26 constructs* **Field** ℓ **With-Value** v *in a parse for a production p, then*

$$\exists \tau.\ v : \tau \wedge \tau <: \text{type}(\text{plbl}(p), \ell).$$

Proof. There are four cases under which **Field** ℓ **With-Value** v may be constructed (under the definition of $\mathit{fields}(p, i)$).

1. **Field** $\text{slbl}(p, i)$ **With-Value** p_i.TREE.

   By Lemma 9, p_i.TREE <: τ where $p_i : \tau$. By Definition 19, $\text{type}(\text{plbl}(p), \ell = \text{slbl}(p, i)) = \tau$ or **Boolean**, so p_i.TREE <: $\text{type}(\text{plbl}(p), \ell)$.

2. **Field** $\text{slbl}(p, i)$ **With-Value** (**List-Node** p_i.LIST).

   By Lemma 8, **List-Node** p_i.LIST <: **List** τ where $p_i : \tau$, and $\text{type}(\text{plbl}(p), \ell = \text{slbl}(p, i)) = \tau$ by Definition 19.

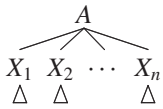
3. **Field** $\text{slbl}(p, i)$ **With-Value** (**Omitted-Node** p_i.FIELDS).

   Omitted-Node $L : \bot$ and $p_i : \bot$, so $\text{type}(\text{plbl}(p), \ell) = \bot$ by Definition 19.

4. **Field** $\text{slbl}(p, i)$ **With-Value** (**AST-Node** $\text{nlbl}(p_i)$ $\cdots$).

   AST-Node $\text{nlbl}(p_i)$ $\cdots$: **Concrete** $\text{nlbl}(p_i)$. By Definition 19, **Concrete** $\text{nlbl}(p_i)$ = $\text{type}(\text{plbl}(p), \ell = \text{slbl}(p, i))$. □

Lemma 12. *Let $p = A ::= X_1 X_2 \cdots X_n$ be a production in a well-formed, annotated grammar and*



*be a subtree of an attributed parse tree resulting from Definition 26. If X_i.*FIELDS *contains* **Field** ℓ **With-Value** v *and* $\text{sann}(p, i) =$ ***inline****, then*

$$v <: \text{type}(\text{plbl}(p), \ell).$$

Proof. By Lemma 11 and Definition 19, $v <: \text{type}(\text{nlbl}(X_i), \ell) = \text{type}(\text{plbl}(p), \ell)$. □

Notation. *An* **AST-Node***'s name will be denoted by name*$(\cdot)$*: name*(**AST-Node** k **With-Fields** L) $:= k$.

Theorem 4. *If an attribute grammar constructed using Definition 26 constructs*

AST-Node k **With-Fields** [

Field ℓ_1 **With-Value** v_1, **Field** ℓ_2 **With-Value** v_2, ..., **Field** ℓ_n **With-Value** v_n],

then, for every i ($1 \le i \le n$),

(A) $k \in \mathcal{C}_{\textbf{\textit{generate}}}$ *(i.e., k is a concrete AST node class),*

(B) $\ell_i \in \mathcal{F}_k$ *(i.e., every field is a member of that node class),*

(C) $\ell_i = \ell_j$ *iff* $i = j$ *(i.e., every field is is assigned at most once), and*

(D) $\exists \tau.\ v_i : \tau$ *and* $\tau <: \text{type}(k, \ell)$ *(i.e., the value assigned to each field is correctly typed).*

Proof. Parts (B) and (C) follow from Lemma 7. Part (D) follows from Lemmas 8–12. We will prove part (A). There are three cases where an **AST-Node** is instantiated.

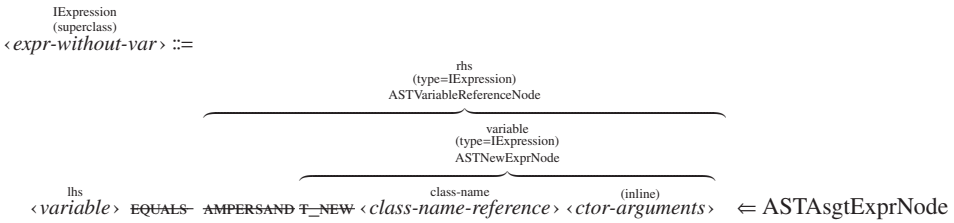
1. *As the value of a field in fields*(p, i). According to the definition of *fields*, $p_i \in N_{\text{map}}$ and $p_i \not= \bot$. By the definition of N_{map}, we must have $p_i \in N$. Because the grammar is properly inheriting and $p_i \not= \bot$, we must have $\text{nann}(p_i) =$ ***generate***, and thus $\text{nlbl}(p, i) \in \mathcal{C}_{\textbf{\textit{gen1}}}$ and $\text{nlbl}(p, i) \in \mathcal{C}_{\textbf{\textit{generate}}}$.
2. *As the value of A.*TREE *when A :* **Concrete** τ *and $A \notin N_{map}$.* By the symbol typing relation, $\text{nann}(A)$ must be ***generate***, and since the grammar is properly inheriting, $\text{plbl}(p) = \text{nlbl}(A)$. Hence, $\text{plbl}(p) = \text{nlbl}(A) \in \mathcal{C}_{\textbf{\textit{gen1}}} \subseteq \mathcal{C}_{\textbf{\textit{generate}}}$.
3. *As the value of A.*TREE *when A :* **Abstract** τ *and A does not have a sole RHS nonterminal.* Here, $\text{nann}(A) =$ ***super***, and, because the grammar is properly inheriting, $\text{plbl}(p) \not= \text{nlbl}(A)$. Thus, $\text{plbl}(p) \in \mathcal{C}_{\textbf{\textit{gen2}}} \subseteq \mathcal{C}_{\textbf{\textit{generate}}}$.

□

4.11 Extraction

One annotation was not included in the previous construction: extraction. This is because extraction does not need to be handled in the construction *per se.* It is more easily handled in implementation. This is done by rewriting the grammar to remove *(extract)* annotations, generating the AST, and then reconstructing the original grammar before the parser is generated.

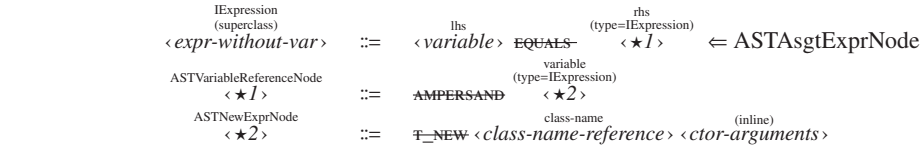
This process is most easily illustrated by example. The following production from the PHP grammar shows a fairly complex use of extraction annotations.



Here, "T_NEW ‹*class-name-reference*› ‹*ctor-arguments*›" is extracted into an ASTNewExprNode; that node and the preceding ampersand are extracted into an ASTVariableReferenceNode; and that node becomes the right-hand side of an assignment statement.

Now, the following steps would be followed to generate a parser and AST from a grammar containing this production.

1. **Eliminate *(extract)* annotations.** Create a new production (for a fresh nonterminal) from each extracted region, and replace the extracted region with a use of this nonterminal. Repeat this process until no extracted regions remain. In the preceding example, this process would produce a grammar like the following.



   Formally, this means rewriting each production

$$\overset{\ell_A}{\underset{a_A}{A}} \quad ::= \quad \overset{\ell_1}{\underset{a_1}{X_1}} \cdots \overset{\ell_{i-1}}{\underset{a_{i-1}}{X_{i-1}}} \overbrace{\overset{\ell_i}{\underset{a_i}{X_i}} \cdots \overset{\ell_j}{\underset{a_j}{X_j}}}^{\substack{\ell_k \\ a_k \\ k_B}} \overset{\ell_{j+1}}{\underset{a_{j+1}}{X_{j+1}}} \cdots \overset{\ell_n}{\underset{a_n}{X_n}}$$

   in the annotated grammar into a new production

$$\begin{array}{lcl} \overset{\ell_A}{\underset{a_A}{A}} & ::= & \overset{\ell_1}{\underset{a_1}{X_1}} \cdots \overset{\ell_{i-1}}{\underset{a_{i-1}}{X_{i-1}}} \overset{\ell_k}{\underset{a_k}{B}} \overset{\ell_{j+1}}{\underset{a_{j+1}}{X_{j+1}}} \cdots \overset{\ell_n}{\underset{a_n}{X_n}} \\ \overset{k_B}{\underset{\text{(generate)}}{B}} & ::= & \overset{\ell_i}{\underset{a_i}{X_i}} \cdots \overset{\ell_j}{\underset{a_j}{X_j}} \end{array}$$

   where B is a nonterminal not occurring elsewhere in the (rewritten) grammar.

2. **Construct AST node classes and semantic actions for the modified grammar** using the process described in the previous sections. Code for the AST classes may be generated at this point, but the parser (and its semantic actions) cannot be generated until step 4, below.

3. **Reconstruct the original grammar by inlining extracted productions and their semantic actions.** For each nonterminal B constructed in step 1, there is exactly one production and one use. Inline this production, and inline its semantic action. Do this for every production created in step 1 in order to reconstruct the original context-free grammar. (At this point, AST nodes have already been constructed, so the labels and annotations on each symbol do not matter.)

   Inlining productions is straightforward. Inlining semantic actions may be slightly more tricky, depending on implementation details. For example, Yacc uses `$$`, `$1`, `$2`, etc. to refer

to symbols positionally. When a production is inlined, the number of symbols on the RHS changes, so these references may need to be changed. For example,

```
‹expr-without-var›   ::=   ‹variable› EQUALS ‹⋆1›                          { $$ = f($1, $2, $3); }
      ‹⋆1›           ::=   AMPERSAND ‹⋆2›                                   { $$ = g($1, $2); }
      ‹⋆2›           ::=   T_NEW ‹class-name-reference› ‹ctor-arguments›    { $$ = h($1, $2, $3); }
```

would, after inlining, become something like the following.

```
‹expr-without-var›   ::=   ‹variable› EQUALS AMPERSAND T_NEW ‹class-name-reference› ‹ctor-arguments›
                           { tmp1 = h($4, $5, $6);
                             tmp2 = g($3, tmp1);
                               $$ = f($1, $2, tmp2); }
```

4. **Generate the parser** from the reconstructed grammar and its semantic actions.

Why Reconstruction Is Necessary

Since the original annotated grammar and the grammar without *(extract)* annotations generate the same language and produce the same AST, why is it necessary to reconstruct the original grammar? In short, it may not be possible to generate a parser directly from the modified grammar. Consider the grammar

$$A \quad ::= \quad \overbrace{a}^{\cdots}\ b\ c \ \mid\ \overbrace{a}^{\cdots}\ b\ d$$

which, after removing *(extract)* annotations, becomes

$$\begin{array}{lll} A & ::= & \star_1\ b\ c \mid \star_2\ b\ d \\ \star_1 & ::= & a \\ \star_2 & ::= & a \end{array}$$

Ludwig currently generates LALR(1) parsers. While the original grammar is LALR(1), the modified grammar is not: The parser generator will fail, giving a reduce/reduce conflict since it cannot tell whether to reduce a to $\star_1$ or $\star_2$ on lookahead b. (Needless to say, this error message would be tremendously confusing to a user, since the nonterminals $\star_1$ and $\star_2$ are not in his grammar!) Inlining the productions for $\star_1$ and $\star_2$ eliminates the conflict.

II

Semantic Infrastructure

5

Analysis Requirements in Refactoring Engines

In Part I, we were able to discuss the syntactic infrastructure of an automated refactoring tool without being very specific about what refactorings the tool would support. Unfortunately, this is less true for Part II. The question of what semantic information does—or does not—need to be available in a refactoring tool depends heavily on what refactorings the tool will perform (and the level of accuracy desired). This chapter will begin by looking at empirical data on what refactorings are most commonly implemented and what static analyses these refactorings require. This will be used to motivate the *program graph* representation that will be used in the next two chapters.

5.1 Refactorings: What Is Available, What Is Used

5.1.1 Refactorings Available in Current Tools

Table 5.1 shows nine popular tools that support automated refactoring for various languages. Eclipse JDT and IntelliJ IDEA support Java[1], ReSharper and Visual Studio support C#, Eclipse CDT and Visual Assist X support C and C++, Apple Xcode supports C and Objective-C, Zend Studio supports PHP, and the Refactoring Browser is for Smalltalk.

The filled and open circles indicate which refactorings are, or are not, supported by each tool. As indicated in the last row, some of the tools support other refactorings besides those listed. (The table only lists refactorings that were supported by at least two of the tools.) Many of the unlisted refactorings are language-specific. For example, ReSharper's "Convert Method to Property" refactoring only makes sense in C# since properties are a feature unique to C#.

So, which refactorings are the "most important," regardless of language? One way to determine this is to look at which refactorings are implemented by a majority of the tools listed (i.e., at least 5 of the 9 tools). There are eight such refactorings.

1. **Rename.** Changes the name of a class, method, variable, etc. The user selects an identifier and enters a new name. The declaration(s) and all references to that identifier are changed to refer to the new name.
2. **Extract Local Variable.** Removes a subexpression, assigning it to a new local variable and replacing the subexpression with a reference to that variable. The user enters a name for the

[1] IDEA also supports some other languages, including Groovy.

Refactoring	Eclipse JDT 3.4–3.5 (Java)	IntelliJ IDEA 10[1] (Java)	ReSharper 5[2] (C#/VB)	Visual Studio 2010[3] (C#)	Eclipse CDT 7.0 (C/C++)	Visual Assist X[4] (C/C++)	Apple Xcode 3[5] (C/Obi-C)	Zend Studio 8[6] (PHP)	Refac. Browser[7] (Smalltalk)	Uses in JDT
Rename	●	●	●	●	●	●	●	●	●	179,871 (74.8%)
Extract Variable	●	●	●	○	●	○	○	●	●	13,523 (5.6%)
Move	●	●	●	○	○	○	○	●	●	13,208 (5.5%)
Extract Method	●	●	●	●	●	●	●	●	●	10,581 (4.4%)
Change Signature	●	●	●	●	○	●	○	○	●	4,764 (2.0%)
Inline	●	●	●	○	○	○	○	○	●	4,102 (1.7%)
Extract Constant	●	●	○	○	●	○	○	○	N/A	3,363 (1.4%)
Encapsulate Field	●	●	●	●	○	●	●	○	●	1,739 (0.7%)
Extract Interface	●	●	●	●	N/A	N/A	N/A	○	N/A	1,612 (0.7%)
Convert Local to Field	●	●	○	○	○	○	○	○	●	1,603 (0.7%)
Pull Up	●	●	●	○	○	●	●	○	●	1,134 (0.5%)
Extract Class	●	●	○	○	○	○	○	○	○	983 (0.4%)
Move Member Type to New File	●	●	●	○	○	○	N/A	○	N/A	838 (0.3%)
Infer Generic Type Arguments	●	●	○	○	N/A	N/A	N/A	N/A	N/A	703 (0.3%)
Extract Superclass	●	●	●	○	○	○	○	○	○	558 (0.2%)
Introduce Parameter	●	●	●	○	○	○	○	○	○	416 (0.2%)
Push Down	●	●	●	○	○	●	●	○	●	279 (0.1%)
Convert Anonymous Class to Nested	●	●	●	○	N/A	N/A	N/A	○	N/A	269 (0.1%)
Introduce Parameter Object	●	●	●	○	○	○	○	○	○	208 (0.1%)
Generalize Declared Type	●	●	○	○	○	○	N/A	○	N/A	173 (0.1%)
Introduce Indirection	●	○	○	○	○	○	○	○	○	145 (0.1%)
Use Supertype	●	●	●	○	○	○	N/A	○	N/A	143 (0.1%)
Introduce Factory	●	●	●	○	○	○	○	○	N/A	121 (0.1%)
Safe Delete	○	●	●	○	○	○	○	○	●	N/A
(Other Refactorings)	○	●	●	○	○	●	●	○	●	N/A

Legend: ● *Included* ○ *Not Included* N/A *Not Applicable*

[1] http://www.jetbrains.com/idea/features/refactoring.html
[2] http://www.jetbrains.com/resharper/features/code_refactoring.html
[3] http://msdn.microsoft.com/en-us/library/719exd8s.aspx
[4] http://www.wholetomato.com/products/featureRefactoring.asp
[5] http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/XcodeWorkspace/150-Refactoring/refactoring.html
[6] http://www.zend.com/en/products/studio/features#refactor
[7] http://www.refactory.com/RefactoringBrowser/Refactorings.html

Table 5.1: Automated refactorings in popular tools. The last two columns give usage in Eclipse by approximately 13,000 Java developers, according to Murphy-Hill et al. [65, Table 1]. Refactoring commands were used a total of 240,336 times; the last column gives the percentage of uses relative to that number.

new local variable. Some implementations provide the option to replace all (syntactically-identical) occurrences of the expression.

3. **Move.** Moves a field, method, etc. from one class to another, or moves a class from one package to another.
4. **Extract Method.** Removes a sequence of statements or a subexpression, placing it in a new method and replacing the statements or subexpression with a call to that method. Local

variables are passed to and returned from the method as needed. The user enters a name for the new method. Some implementations provide the option to replace all (syntactically-identical) occurrences of the statements or subexpression.

5. **Change Method Signature.** Allows the user to permute a method's parameters, add or remove parameters, change parameter types, and change the method's return type. Call sites are updated accordingly.
6. **Encapsulate Field.** Reduces the visibility of a field to *private,* adds getter and setter methods for that field, and replaces all uses of the field with calls to the appropriate getter or setter. The user provides the getter and setter method names. Field accesses within the declaring class may not be replaced with getter and setter invocations, at the user's discretion.
7. **Pull Up.** Moves methods or fields into a superclass.
8. **Push Down.** Moves methods into a subclass.

5.1.2 Empirical Data on Refactoring Usage

While it is important to know what refactorings tools commonly implement, it is perhaps more important to know which of these refactorings users *actually use.* Arguably, the most useful data are provided by Murphy-Hill et al. [65] and Murphy et al. [64]. Murphy et al. used the Mylar Monitor to collect data from 41 Java developers on the features used in the Eclipse IDE. Murphy-Hill et al. compare this data set with two others, including a publicly-available data set available from the Eclipse Foundation [7]. Eclipse 3.4 included the Usage Data Collector (UDC), a monitor that logged all of the commands executed by a user; this data set consists of these logs from more than 13,000 developers who consented to sending this information to the Eclipse Foundation. Although it is a convenience sample, the UDC data set is by far the largest and probably gives the best indication of what automated refactorings are most used by Java developers.

The last two columns in Table 5.1 show the numbers collected from the UDC data set, according to Murphy-Hill et al. [65, Table 1, "Everyone"]. Refactoring commands were used a total of 240,336 times; the second-to-last column indicates the number of times each refactoring was used, and the last column gives the percentage of uses relative to the total.

5.2 Priorities in a New Refactoring Tool

There are three important facts we can learn from Table 5.1.

- **Rename is used almost three times more than all other refactorings combined.** This was also true in Murphy's data set [65]. The importance of Rename is bolstered by Murphy's observation that almost 100% of the developers in her sample used this refactoring, while the next-most-frequently used refactorings (Move and Extract) were each used by fewer than 60% of developers. The importance of Rename has also been noted in Dig and Johnson [32] and Counsell et al. [30]
- **90% of the refactorings tool usage was due to just four refactorings:** Rename, Extract Local Variable, Move, and Extract Method. This fact is also bolstered by Murphy's data set, in which about 86% of uses were devoted to these four refactorings [65, Table 1, "Users"].

- **Many popular IDEs provide fewer than 10 refactorings,** including Apple Xcode (8 refactorings), Microsoft Visual Studio (6), and Zend Studio (4).

The studies from Murphy-Hill et al. [65] and Murphy et al. [64] measured the usage of Java refactorings in Eclipse. It is debatable whether their results can be extrapolated to other tools and other languages. However, the refactorings they identify as the most commonly used are not Java-specific, and anecdotal evidence suggests that they are equally important for other tools and other languages.

So, what does this mean for someone building a new refactoring tool? The first priority should be to implement Rename, and to implement it well. Clearly, a robust Rename refactoring should be the highest priority in a refactoring tool. Extract Local Variable, Move, and Extract Method should be the next priorities.

Extract Method is particularly important to refactoring toolsmiths. Martin Fowler [36] identified this refactoring as a tipping point for refactoring tools, claiming that refactoring tools with a robust Extract Method refactoring had "crossed refactoring's rubicon"—the implication being, if a refactoring tool could implement Extract Method well, it could implement many other refactorings. Some of the first refactoring tools attempted to use regular expression matching or other ill-advised techniques to perform transformations, but inevitably these approaches failed. Since Extract Method is virtually impossible to implement well without a fairly sophisticated program representation (an AST, often with flow analysis), the existence of a good Extract Method refactoring provided an easy way to distinguish serious refactoring tools from those that were doomed to failure.

5.3 What Semantic Information is Required

5.3.1 Name Bindings, Class Hierarchy, and Expression Typing

Now that we identified the most crucial refactorings, we can begin to look at what is required to implement them in a tool. One thing that is not obvious from Table 5.1 is that *the most popular refactorings do not require very complicated static analyses.* Rename, Move, Change Method Signature, Encapsulate Field, Pull Up, Push Down, and many other refactorings can usually be implemented with just three types of semantic information.

1. *Information about name bindings and scoping.* The tool must be able to map a use of an identifier to its declaration, determine all of the identifiers in scope at a particular location in the program, and find all of the references to a particular declaration. For example, Rename must be able to determine whether the new name provided by the user is already in scope at a particular point in the program, as well as whether it will shadow an existing name.
2. *Information about the class hierarchy.* In an object-oriented language, the tool must be able to determine the superclass(es) of a particular class, as well as all of its subclasses. It must be able to determine which methods override which other methods. For example, if a method is renamed (or if its signature is changed), then all overriding and overridden methods must be modified similarly.
3. *Type information.* In a statically-typed language, the tool must be able to determine the static type of an expression. This is often required to compute name bindings—for example,

in `(point1+point2).x`, the type of `point1+point2` must be known in order to determine what declaration `x` refers to.

Type information is also necessary to implement Extract Local Variable, Extract Method, and Extract Constant in a statically-typed language: If the user is allowed to extract an expression, then the type of that expression must be computed to create an appropriate declaration for the new variable/method/constant.

5.3.2 Control Flow and Def-use Chains

Extract Method, as discussed earlier, is one of the more complex refactorings in a typical tool. To implement this refactoring successfully, the refactoring tool must be able to do the following (among other things).

1. *Ensure that the extracted statements do not contain a* `return` *statement.* If a `return` statement is moved into a new method, it will return from the extracted method rather than the original method; this could change the behavior of the original method.[2]
2. *Ensure that, if the extracted statements contain a* `break` *or* `continue` *statement, the entire loop is being extracted.* Similarly, in a language with `goto` statements, statement labels must be contained in the extracted block iff all (`goto`) statements referencing those statements are also contained in the extracted block.
3. *Determine which local variables are used within the extracted statements.* These will be passed as arguments to the extracted method or declared as local variables in the extracted method.
4. *Determine which local variables are modified by the extracted method.* These assignments may need to be propagated back to the original method (e.g., by passing the affected variables to the extracted method by reference, or by having the extracted method return the value of these variables).

Implementing items 3 and 4 correctly can be difficult. It is easy enough to determine, using only the AST and name binding information, which local variables are used in the extracted statements and which of those are used in assignment statements. One option is to pass as arguments those local variables that are used within the extracted block, and return the values of those locals that are assigned; locals that are used in the extracted block and nowhere else can be declared as locals in the new procedure rather than being parameters of the procedure. This approximation is good enough to work on many programs, but there are cases where its results are less than ideal. For example, if a variable is unconditionally assigned following the extracted statements, it is not necessary to return that variable's value from the new procedure. Similarly, it is not necessary to pass the variable to the procedure if it is always assigned before it is read inside the procedure. So, a better analysis is desirable.

Extract Procedure is the refactoring analog of a compiler optimization usually called *outlining* [94] (although some papers call it *partial cloning* [92], *folding* [57], or simply *procedure extraction* [56]). In the context of this optimization, the sets of local variables are determined using

[2]Technically, it is possible to extract code containing a `return` statement in certain cases, but most tools do not.

a reaching definitions dataflow analysis; one good description is given by Lakhotia and Deprez [57]. A similar description, in the context of a refactoring tool, is given by Verbaere et al. [87].

Like any dataflow analysis, computing reaching definitions requires a control flow graph. (Incidentally, the first two requirements of Extract Method listed above can also be formulated in terms of control flow requirements.) Thus, we can classify Extract Procedure and similar refactorings (Extract Local Variable and Inline) as having two additional infrastructural requirements.

1. *Control flow information.*
2. *Def-use chains for local variables.*

5.4 History

While it is fairly clear what semantic information is required of a refactoring tool today, the situation was not so clear in the early days of refactoring (partly because it was not even clear what transformations would be needed). The first refactoring tool was Griswold's prototype restructurer for Scheme [42]. It used an AST as its syntactic program representation and program dependence graphs [34, 35] (PDGs) as its semantic representation. However, since PDGs do not capture the scoping of names, Griswold augmented the PDG with *contours;* this allowed him to detect when a transformation would change name bindings. Griswold found that keeping two distinct program representations (the AST and PDG) synchronized was difficult. [42] However, a somewhat bigger problem with his prototype was its speed: It was very slow, sometimes taking several minutes to perform a transformation.[3]

When Brant and Roberts built the Smalltalk Refactoring Browser [80] a few years later, they took a very different approach: They built all of the refactorings' preconditions using only syntactic pattern matching and name lookups, with no additional program analyses [18]. These checks were very inexpensive, which made the Refactoring Browser usable in interactive time. While the authors were careful not to sacrifice safety, avoiding expensive analyses *did* occasionally compromise precision. For example, renaming the selector (i.e., method) *open* would result in both *File#open* and *Socket#open* being renamed, even if the two were theoretically distinguishable (e.g., through type inference). In the few cases where a safe transformation could not be guaranteed (e.g., inlining a dynamically-dispatched method), rather than attempt an expensive analysis, the tool would simply ask the user for input; then the correctness of the transformation depended on the accuracy of the user's input [18].

Using only syntactic checks worked well for Smalltalk because it has a surprisingly small syntax [11]. There are no syntactic control flow constructs syntax except for a *return* statement (which is always the last statement in a method or block). Even conditional execution and iteration are achieved by sending messages to objects.

Unfortunately, the syntactic pattern matching approach does not work so well for larger languages. Consider the problem of determining whether a sequence of statements constitutes a single-entry, single-exit block. (This is a prerequisite for the Extract Method refactoring.) In Smalltalk, this amounts to checking for a *return* statement. In Java, one must also test for *break*

[3]Note that Griswold's dissertation was published in 1991. His tool's performance would probably be quite good on today's machines.

and *continue* statements. In C, there are labels and *goto* statements... and obtaining this information from a control flow graph begins to look quite appealing.

Of course, if the control flow graph is separate from the AST, the two representations must be kept in sync—exactly one of the problems Griswold encountered. One obvious way to avoid mapping between separate program representations is to combine them into a single representation. For a source-level restructuring tool, this would usually mean adding some semantic information to the AST. Morgenthaler [63] overcame this problem for control flow by allowing AST nodes to dynamically compute which other AST nodes constituted its control flow successors and predecessors. Verbaere [86, 88] added control flow and def-use chains to the AST by superimposing a graph structure directly onto the AST (we will return to this idea in the next chapter.)

While the syntactic pattern matching approach may be too weak for some languages, building an entire program dependence graph (as Griswold did) is probably overkill. None of the common refactorings in Table 5.1 require information about control or data dependences. Information about name bindings, inheritance/overriding, control flow, and def-use chains will suffice.

The next chapter will look at *program graphs,* which have enjoyed a good deal of success in the research literature. Program graphs are appealing because they combine all of this information—name bindings, control flow, etc.—into a single program representation by "layering" it on top of the abstract syntax tree. Moreover, they are very generic, making them ideal for implementation in a language-agnostic framework for refactoring engines.

6 Differential Precondition Checking

6.1 Introduction[†]

Automated refactorings have two parts: the transformation—the change made to the user's source code—and a set of *preconditions* which ensure that the transformation will produce a program that compiles and executes with the same behavior as the original program. Authors of refactoring tools agree that precondition checking is much harder than writing the program transformations.

This chapter shows how to construct a reusable, generic precondition checker which can be placed in a library and reused in refactoring tools for many different languages. This makes it easier to implement a refactoring tool for a new language.

This chapter introduces a new technique for checking preconditions, which will be called *differential precondition checking*. A differential precondition checker builds a *semantic model* of the program prior to transformation, simulates the transformation, performs semantic checks on the modified program, computes a semantic model of the modified program, and then looks for differences between the two semantic models. The refactoring indicates what differences are expected; if the actual differences in the semantic models are all expected, then the transformation is considered to be behavior preserving. The changes are applied to the user's code only after the differential precondition checker has determined that the transformation is behavior preserving.

This technique is simple, practical, and minimalistic. It does not guarantee soundness, and it is *not* a general method for testing program equivalence. Rather, it is designed to be straightforward, fast, scalable, and just expressive enough to implement preconditions for the most common refactorings. Most importantly, the core algorithm can be implemented in a way that is completely language independent, so it can be optimized, placed in a library, and reused in refactoring tools for many different languages.

This chapter makes five contributions. (Relevant section numbers are noted parenthetically.)

1. It characterizes preconditions as guaranteeing *input validity, compilability,* and *preservation* (§6.3).
2. It introduces the concept of *differential precondition checking* (§6.3) and shows how it can simplify precondition checking by eliminating compilability and preservation preconditions (§6.5).

[†]Portions of this chapter are based on J. Overbey and R. Johnson, "Differential Precondition Checking: A Lightweight, Reusable Analysis for Refactoring Tools," to appear in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011).*

3. It observes that semantic relationships between the modified and unmodified parts of the program tend to be the most important and, based on this observation, proposes a very concise method for refactorings to specify their preservation requirements (§6.5).
4. It describes how the main component of a differential precondition checker (called a *preservation analysis)* can be implemented in a way that is both fast and language independent (§6.7).
5. It provides an evaluation of the technique (§6.8), considering its successful application to 18 refactorings and its implementation in refactoring tools for Fortran (Photran), PHP, and BC.

6.2 Precondition Checking

In most tools, each refactoring has its own set of preconditions. These are tested first, and the transformation proceeds only if they pass. Unfortunately, designing a sufficient set of preconditions for a new refactoring is extremely difficult. The author of the refactoring must exhaustively consider every feature in the target language and somehow guarantee that the transformation is incapable of producing an error. Consider Java: Even a "simple" refactoring like Rename must consider naming conflicts, namespaces, qualifiers, shadowing, reserved words, inheritance, overriding, overloading, constructors, visibility, inner classes, reflection, externally-visible names, and "special" names such as `main`.

One promising alternative to traditional precondition checking is to analyze the program *after* it has been transformed, comparing it to the original program to determine whether or not the transformation preserved behavior. This has been used for some dependence-based compiler transformations (e.g., a *fusion preventing dependence* [52, p. 258] is most easily detected after transformation), but researchers have applied it to refactoring tools only recently. Although this technique is not yet used in any commercial tools, research indicates that it tends to make automated refactorings simpler and more robust [82].

So, how can a refactoring tool analyze a program after transformation? Refactorings preserve certain relationships in the source program. The Rename refactoring preserves a name binding relationship: It ensures that every identifier refers to the "same" declaration before and after transformation. Extract Method and Extract Local Variable preserve control flow and def-use chains at the extraction site. As we will see later in this chapter, Pull Up Method preserves a name binding relationship, as well as a relationship between classes and methods they override. In our experience, the most common refactorings preserve invariant relationships related to name bindings, inheritance, overriding, control flow, and def-use chains. Analyzing a program after transformation means ensuring that these invariant relationships are preserved across the transformation.

Schäfer et al. have suggested one way to refactor using invariants like these. To implement a Rename refactoring for Java, they stored the original name bindings, changed names, then checked the resulting bindings, adding qualifiers as necessary to guarantee that the name bindings would resolve identically after the transformation was complete [84]. They used a similar approach to implement Extract Method: They stored the original control flow, performed the transformation, then added control flow constructs as necessary to restore the original flow [85]. They have applied this approach to many other refactorings as well [82, 83]. In short, their approach maintains invariants *by construction*—i.e., while performing the transformation, the refactoring checks the

invariant and, if possible, adjusts its behavior to preserve it.

The approach taken in this chapter is based on some of the same ideas as that of Schäfer et al., but there is a substantial difference in *how* we perform the preservation check. The main difference is that our technique, when implemented appropriately, is language independent; the mechanism for specifying preservation requirements and the algorithm for performing the preservation analysis are the same, regardless of what refactoring is being checked and regardless of what language is being refactored. This means that, unlike the approach of Schäfer et al., our preservation analysis can be implemented in a library and reused verbatim in refactoring tools for many different languages.

6.3 Differential Precondition Checking

Preconditions determine the conditions under which the program transformation will preserve behavior. Logically, this means that they guarantee three properties:

1. *Input validity.* All input from the user is legal; it is possible to apply the transformation to the given program with the given inputs.
2. *Compilability.* If the transformation is performed, the resulting program will compile; it will meet all the syntactic and semantic requirements of the target language.
3. *Preservation.* If the transformation is performed and the resulting program is compiled and executed, it will exhibit the same runtime behavior as the untransformed program.

Clearly, input validation needs to be performed before the program is transformed, since it may not even be possible to perform a transformation if the user provides invalid input. But compilability is actually easier to determine *after* transformation; essentially, it means running the program through a compiler front end. It turns out that preservation can often be checked *a posteriori* as well.

When *differential precondition checking* is employed, refactorings proceed as follows:

1. Analyze source code and produce a program representation.
2. Construct a *semantic model,* called the *initial model.*
3. Validate user input.
4. Simulate modifying source code, and construct a new program representation. Detect compilability errors, and if appropriate, abandon the refactoring.
5. Construct a semantic model from this new program representation. This is the *derivative model.*
6. Perform a *preservation analysis* by comparing the derivative model with the initial model.
7. If the preservation analysis succeeds, modify the user's source code. Otherwise, abandon the refactoring.

What distinguishes differential precondition checking is how it ensures compilability and preservation. These topics will be discussed in detail in Sections 6.4 and 6.5, respectively. It ensures compilability by performing essentially the same checks that a compiler front end would perform. It ensures behavior preservation by building *semantic models* of the program before and after it is transformed. The refactoring informs the differential precondition checker of what kinds of semantic

differences are expected; the checker ensures that the actual differences in the semantic models are all expected differences—hence the name *differential* precondition checking.[1]

Note that a differential precondition checker contrasts the program's semantic model *after* transformation with its semantic model *before* transformation. This is different from program metamorphosis systems [79], which provide an "expected" semantic model and then determine whether the transformed program's semantic model is equivalent to the expected model. As we will see in §§6.5.4–6.5.6, the mechanism for specifying expected differences in a differential precondition checker is fairly coarse-grained; it does not uniquely characterize the semantics of a particular transformed program but rather identifies, in general, how a refactoring is expected to affect programs' semantics.

6.4 Checking Compilability

Checking for compilability means ensuring that the refactored program does not contain any syntactic or semantic errors, i.e., that it is a legal program in the target language. These errors would usually be detected by the compiler's front end. Typically, these check constraints like "no two local variables in the same scope shall have the same name" and "a class shall not inherit from itself."

When differential precondition checking is employed, these checks are performed in Step 4 (above), and they are used *in lieu of traditional precondition checks.* For example, a refactoring renaming a local variable A to B would not explicitly test for a conflicting local variable named B; instead, it would simply change the declaration of A to B, and, if this resulted in a conflict, it would be detected by the compilability check.

In fact, most refactoring tools already contain most of the infrastructure needed to check for compilability. It is virtually impossible to perform any complicated refactorings without a parser, abstract syntax tree (AST), and name binding information (symbol tables). A type checker is usually needed to resolve name bindings for members of record types, as well as for refactorings like Extract Local Variable. So, refactoring tools generally contain (most of) a compiler front end. Steps 1 and 4 (above) involve running source code through this front end. So checking for compilability in Step 4 is natural.

The literature contains fairly compelling evidence for including a compilability check in a refactoring tool. Compilability checking subsumes some highly nontrivial preconditions—preconditions that developers have "missed" in traditional refactoring implementations. Verbaere et al. [88] identify a bug in several tools' Extract Method refactorings in which the extracted method may return the value of a variable which has not been assigned—a problem which will be identified by a compilability check. Schäfer et al. [84] describe a bug in Eclipse JDT's Rename refactoring which amounts to a failure to preserve name bindings. Daniel et al. [31] reported 21 bugs on Eclipse JDT and 24 on NetBeans. Of the 21 Eclipse bugs, 19 would have been caught by a compilability check. Seven of these identified missing preconditions;[2] the others were actually errors in the transformation that manifested as compilation errors.

[1] Why differential "precondition" checking? A refactoring takes user input I and uses it to determine a program transformation $T(I)$. However, a precondition for the *application* of $T(I)$ to the user's source code is that it satisfies the properties of compilability and preservation.

[2] Bugs 177636, 194996, 194997, 195002, 195004, 194005, and 195006

Compilability checking also serves as a sanity check. In the presence of a buggy or incomplete transformation, it analyzes what the transformation *actually did,* not what it was *supposed to do.* If the code will not compile after refactoring, the transformation almost certainly did something wrong, and the user should be notified.

6.5 Checking Preservation

Compilability checking is important but simple. Checking for preservation is more challenging. It involves choosing an appropriate semantic model and finding a preservation analysis algorithm that balances speed, correctness, and generality. In this section, we will use a *program graph* as the semantic model. In Section 6.7, we will use a slightly different semantic model based on the same ideas.

In the remainder of this section, we will discuss what program graphs are (§6.5.1) and how they can be used as an analysis representation for a refactoring tool (§6.5.2). Then, we will discuss what *preservation* means in the context of a program graph (§6.5.3) and how it can be used instead of traditional precondition checks, using Safe Delete and Pull Up Method as examples (§§6.5.4–6.5.6).

6.5.1 Program Graphs

One program representation which has enjoyed success in the refactoring literature [62, 87] is called a program graph. A *program graph* "may be viewed, in broad lines, as an abstract syntax tree augmented by extra edges" [62, p. 253]. These "extra edges"—which we will call *semantic edges*—represent semantic information, such as name bindings, control flow, inheritance relationships, and so forth. Alternatively, one might think of a program graph as an AST with the graph structures of a control flow graph, du-chains, etc. superimposed; the nodes of the AST serve as nodes of the various graph structures.

An example of a Java program and a plausible program graph representation are shown in Figure 6.1. The underlying abstract syntax tree is shown in outline form; the dotted lines are the extra edges that make the AST a program graph. Three types of edges are shown. *Name binding* edges link the use of an identifier to its corresponding declaration. Within the method body, *control flow* edges form the (intraprocedural) control flow graph; the method declaration node is used as the entry block and null as the exit block. Similarly, there are two du-chains, given by *def-use* edges.

Program graphs are appealing because they summarize the "interesting" aspects of both the syntax and semantics of a program in a single representation, obviating the need to maintain a mapping between several distinct representations. Moreover, they are defined abstractly: *the definition of a program graph does not state what types of semantic edges are included.* A person designing a program graph is free to include (or exclude) virtually any type of edge imaginable, depending on the language being refactored and needs of the refactorings that will be implemented. For the 18 refactorings considered (see §6.8), five types of edges were found to be useful: name binding, control flow, def-use, *override* edges (which link an overriding method to the overridden implementation in a superclass), and *inheritance* edges (which link a class to the concrete methods it inherits from a superclass).

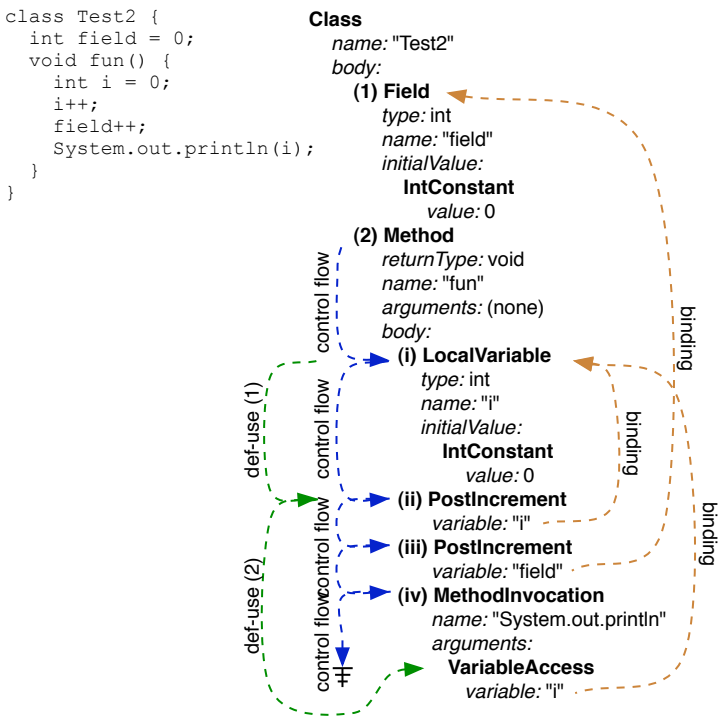


Figure 6.1: Example Java program and corresponding program graph

6.5.2 Program Graphs and AST Manipulation

In the end, refactoring tools manipulate source code. However, when building a refactoring, it is helpful to think of manipulating the AST instead. Adding a node means inserting source code. Replacing a node means replacing part of the source code. And so on.

This does not change when a program graph is used in a refactoring tool. A program graph is always *derived from* an AST. The content of the AST determines what semantic edges will be superimposed. Semantic edges cannot be manipulated directly; they can only change as a side effect of modifying the AST.

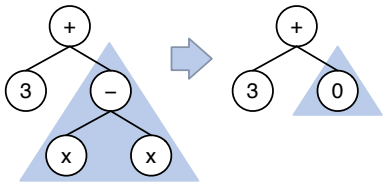
6.5.3 Preservation in Program Graphs

As stated above, semantic edges cannot be manipulated directly; they can only change as a side effect of modifying the AST. In fact, that observation will serve as the basis of our preservation analysis. When we modify an AST, we will indicate which semantic edges we expect to be preserved and which ones we expect to change. Then, after the source code has been modified, we will determine what semantic edges were actually preserved and compare this with our expectations.

This raises a question: What does it mean for a semantic edge to be "preserved" when an AST is modified?

We would like to say: If both the modified and unmodified ASTs contain an edge with the same type and the same endpoints, that edge has been preserved. Unfortunately, it is not clear what the "same" endpoints are, since the AST has been modified, and the endpoints are AST nodes.

Consider a refactoring which replaces the expression $x - x$ with the constant 0. When applied to the expression $3 + (x - x)$, this corresponds to the following tree transformation.

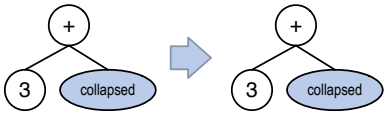


When a subtree is changed (i.e., added, moved, removed, or replaced) in an AST, we will call that an *affected subtree.* A gray triangle surrounds the affected subtrees in the figure above. Using that figure as an example, consider how AST nodes in the unmodified AST correspond with nodes in the modified AST:

- There is an obvious correspondence between AST nodes outside the affected subtrees, since those parts of the AST were unaffected by the transformation.
- As a whole, the affected subtree before the transformation corresponds to the affected subtree after the transformation.
- In general, there is no correspondence between nodes inside the affected subtrees.

Recall that our goal is to determine if a semantic edge has the "same" endpoints before and after an AST transformation. This is easy if an endpoint is outside the affected subtree, or if that endpoint is the affected subtree itself. But if the endpoint is *inside* the affected subtree, we cannot determine exactly which node it should correspond to…except that, if it corresponds to anything, that node would be in the other affected subtree.

Since we cannot determine a correspondence between AST nodes inside the affected subtree, we will *collapse* the affected subtrees into single nodes. This makes the AST before transformation isomorphic to the AST after transformation.



Now, suppose we have superimposed semantic edges to form a program graph. When we collapse the affected subtree to a single node, we will also need to adjust the endpoints of the semantic edges accordingly:

- When an affected subtree is collapsed to a single node, if any semantic edges have an endpoint inside the affected subtree, that endpoint will instead point to the collapsed node.

Note, in particular, that if an edge has *both* endpoints inside the affected subtree, it will become a self-loop on the collapsed node. Also, note that a program graph is not a multigraph: If several edges have the same types and endpoints in the collapsed graph, they will be merged into a single edge.

Collapsing the affected subtree in a program graph actually has a fairly intuitive interpretation: If we replace one subtree with a different subtree that supposedly does the same thing, then the new

subtree should interface with its surroundings in (mostly) the same way that the old subtree did. That is, all of the edges that extended into the old subtree should also extend into the new subtree, and all of the edges that emanated from the old subtree should also emanate from the new subtree. There may be some differences within the affected subtree, but the "interface" with the rest of the AST stays the same.

In some cases, we will find it helpful to replace one subtree with *several* subtrees (or, conversely, to replace several subtrees with one). For example, Encapsulate Variable removes a public variable, replacing it with a private variable, an accessor method, and a mutator method. In other words, we are modifying several subtrees at the same time. In these cases, we have an *affected forest* rather than a single affected subtree. However, the preservation rule is essentially the same: All of subtrees in the affected forest are collapsed into a single unit. So if an edge extended into some part of the affected forest before transformation, it should also extend into some part of the affected forest after transformation. When one subtree is replaced with several, this captures the idea that, if an edge extended into the original subtree, then it should extend into one of the subtrees in the affected forest. In the case of Encapsulate Variable, this correctly models the idea that every name binding that pointed to the original (public) variable should, instead, point to either the new (private) variable, the accessor method, or the mutator method. (We will see an example of an affected forest when we discuss Pull Up Method in §6.5.6.)

6.5.4 Specifying Preservation Requirements

Now that we have established how to determine whether a semantic edge has been preserved across a transformation, we turn to a different question: How can we express which semantic edges we expect to be preserved and which ones we expect to change?

Edge Classifications

From the above description, we can see that whether we want to preserve an edge depends on its type as well as its relationship to the affected subtree. Therefore, it is helpful to classify every semantic edge as either **internal** (both endpoints of the semantic edge occur within the affected subtree), **external** (neither endpoint occurs within the affected subtree), **incoming** (the head of the semantic edge is outside the affected subtree but the tail is inside it), or **outgoing** (the head is inside the affected subtree and the tail is outside it).

Notation

Now, we can establish some notation. To indicate what edges we (do not) expect to preserve, we must indicate three things:

1. *The type(s) of edges to preserve.* We will use the letters N, C, D, O, and I to denote name binding, control flow, def-use, override, and inheritance edges, respectively. (Note, however, that program graphs may contain other types of edges as well, depending on the language being refactored and the requirements of the refactorings being implemented.)
2. *The classification(s) of edges to preserve.* We will use $\leftarrow$, $\rightarrow$, $\cup$, and $\times$ to indicate incoming, outgoing, internal, and external edges, respectively. We will use $\leftrightarrow$ as a shorthand for

describing both incoming and outgoing edges.

3. *Whether we expect the transformation to introduce additional edges or remove existing edges.* If additional edges may be introduced, we denote this using the symbol $\supseteq$ (i.e., the transformed program will contain a superset of the original edges). If existing edges may be eliminated, we denote this by $\subseteq$. If edges may be both added and removed, then we cannot effectively test for preservation, so those edges will be ignored; we indicate this using the symbol $\not=$. Otherwise, we expect a 1–1 correspondence between edges, i.e., edges should be preserved exactly. We indicate this by =.

6.5.5 Example: Safe Delete (Fortran 95)

To make these ideas more concrete, let us first consider a Safe Delete refactoring for Fortran which deletes an unreferenced internal subprogram.[3]

The traditional version of this refactoring has only one precondition: There must be no references to the subprogram except for recursive references in its definition.

What would the differential version look like? To determine its preservation requirements, it is often useful to fill out a table like the following (note that Fortran 95 is not object oriented and thus cannot have *O*- or *I*-edges):

	N	C	D
$\leftarrow$	=	=	=
$\rightarrow$	$\subseteq$	=	=
$\circlearrowleft$	$\subseteq$	$\subseteq$	$\subseteq$
$\times$	=	=	=

When a subprogram is deleted, all of the semantic edges inside the deleted subroutine will, of course, disappear, and if the subprogram references any names defined elsewhere (e.g., other subprograms), those edges will disappear. Otherwise, no semantic edges should change.

Notating preservation requirements in tabular form is somewhat space-consuming, since in practice most cells contain =. Therefore, we will use a more compact notation. For each edge type, we will use subscripts to indicate which cells are *not* =, i.e., what edges should *not* be preserved exactly. If all cells are =, we will omit the subscript. Using this notation, the preservation requirements in the above table would be notated $N^{\rightarrow\circlearrowleft}_{\subseteq\subseteq}C^{\circlearrowleft}_{\subseteq}D^{\circlearrowleft}_{\subseteq}$.

Thus, we can describe the differential version of this refactoring in a single step: *Delete the subprogram definition, ensuring preservation according to the rule* $N^{\rightarrow\circlearrowleft}_{\subseteq\subseteq}C^{\circlearrowleft}_{\subseteq}D^{\circlearrowleft}_{\subseteq}$.

6.5.6 Example: Pull Up Method (PHP 5)

For a more interesting example, let us consider a Pull Up Method refactoring for PHP 5, which moves a concrete method definition from a class C into its immediate superclass C'.[4] First, consider the traditional version.

Preconditions.

[3] A slightly more complete and much more detailed specification for this refactoring is given in Appendix B.

[4] Again, a more complete and detailed specification is available [70].

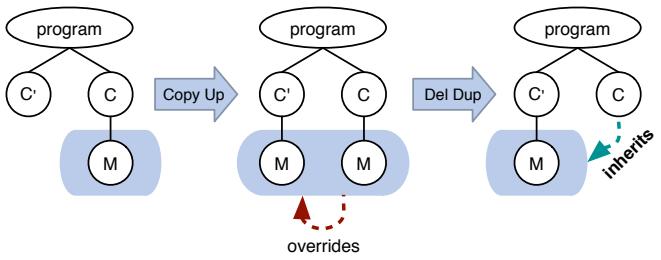
1. *A method with the same name as M must not already exist in C′.* If *M* were pulled up, there would be two methods with the same name, or *M* would need to replace the existing method.
2. *If there are any references to M (excluding recursive references inside M itself), then M must not have private visibility.* If it were moved up, its visibility would need to be increased in order for these references to be preserved.
3. *M must not contain any references to the built-in constants* `self` *or* `__CLASS__`. If it were moved up, these would refer to *C′* instead of *C*. (Note that PHP contains both `self` and `$this`: The former refers to the enclosing class, while the latter refers to the *this* object.)
4. *M must not contain any references to private members of C (except for M itself, if it is private).* Private members of *C* would no longer be accessible to *M* if it were pulled up.
5. *M must not contain any references to members of C for which there is a similarly-named private member of C′.* These references would refer to the private members of *C′* if the method were pulled up.
6. *If M overrides another concrete method, no subclasses of C′ may inherit the overridden method.* Pulling up *M* would cause these classes to inherit the pulled up method instead.
7. *The user should be warned if M overrides another concrete method.* If *M* were pulled up into *C′*, then *M* would replace the method that *C′* inherited, changing the behavior of that method in objects of type *C′*, although the user might intend this since he explicitly chose to pull up *M* into *C′*.

Transformation. Move *M* from *C* to *C′*, replacing all occurrences of `parent` in *M* with `self`.

Now, consider the differential version. The transformation can be expressed as the composition of two smaller refactorings:

1. *Copy Up Method.* Using preservation rule $NO^{\hookleftarrow\cup}_{\supseteq\supseteq}I^{\times}_{\subseteq}$, copy the method definition from *C* to *C′*, replacing all occurrences of `parent` in *M* with `self`.
2. *Delete Overriding Duplicate.* Remove the original method definition from *C*, with rule $NO^{\cup}_{\subseteq}I^{\sqsubseteq}_{\supseteq}$.

Pictorially, the process is as follows. The affected forests are highlighted in gray.



When the method is copied from *C* to *C′*, an internal override edge will be introduced, as may incoming override edges (if another class will override the pulled up method), hence the rule $O^{\hookleftarrow\cup}_{\supseteq\supseteq}$. If the method being pulled up overrides a method inherited from the immediate superclass, then an inheritance edge will be lost, hence $I^{\times}_{\subseteq}$. However, the new method in *C′* should not be inherited by any subclasses, and all identifiers should bind to the same names they did when the method was

contained in C, so no other inheritance or name binding edges are expected to change. Once we have established that no subclasses will accidentally inherit the pulled up method, we can delete the original method from C. This will remove the override edge introduced in the previous step, and C will inherit the pulled up method, so the preservation rule is $NO^{\circlearrowleft}_{\subseteq}I^{\supseteq}$.

Now, consider how the differential version of this refactoring satisfies all of the traditional version's preconditions. Precondition 1 would be caught by a compilability check. Preconditions 2–5 are simply preserving name bindings. A program that failed Precondition 6 would introduce an incoming inheritance edge. If a program failed Precondition 7, an outgoing inheritance edge from C' would vanish.

For the differential version, we redefined Pull Up Method as the *composition* of two smaller refactorings. Whenever this is possible, it is generally a good idea: It allows preservation rules to be specified at a finer granularity; the smaller refactorings are often useful in their own right; and, perhaps most importantly, simpler refactorings are easier to implement, easier to test, and therefore more likely to be correct.

6.6 The Preservation Analysis Algorithm

If one understands what a program graph is, and what the preservation rules mean, the preservation analysis algorithm is straightforward. A program graph becomes an abstract data type with

Sorts: ProgramGraph, Edge, Type
Operations:
getAllEdges : ProgramGraph → finite set of Edge
classify : Edge → {←, →, ↻, ×}
type : Edge → Type
equiv : Edge × Edge → {TRUE, FALSE}.

The *equiv* operation determines whether two edges—one in the original program graph and one in the transformed program graph—are equivalent, i.e., if the edge was preserved. For simplicity, we have left this underspecified, although its intent should be clear from the previous section. Now, preservation is determined by the following algorithm.

Input: P : ProgramGraph *(Original program)*
P' : ProgramGraph *(Transformed program)*
rule : Type $\times \{\leftarrow, \rightarrow, \circlearrowleft, \times\} \rightarrow \{=, \subseteq, \supseteq, \neq\}$

Output: PASS or FAIL

let E := **getAllEdges**(P)
let E' := **getAllEdges**(P')
for each Edge $e \in E$
  if *rule*(**type**(e), **classify**(e)) is $\supseteq$ or $=$
    but $\nexists\, e' \in E'$ s.t. **equiv**(e, e') = TRUE, then
      FAIL
for each Edge $e' \in E'$
  if *rule*(**type**(e'), **classify**(e')) is $\subseteq$ or $=$
    but $\nexists\, e \in E$ s.t. **equiv**(e, e') = TRUE, then
      FAIL
otherwise, PASS

6.7 Analysis with Textual Intervals

6.7.1 Overview

The key to an efficient implementation is being able to determine, for a particular edge, whether an equivalent edge exists in the transformed program. If this can be done in $O(1)$ time, then the above algorithm's execution time is linear in the number of edges in the two program graphs. This section will illustrate one way to do this (which also makes the implementation language independent).

The ASTs in refactoring tools tend to model source code very closely. This means that they tend to exhibit a very useful property: Every node in an AST corresponds to a particular textual region of the source code, *and this textual region can be mapped back to a unique AST node.* Consider the program graph from Figure 6.1. The source code is 115 characters long. The *Class* AST node corresponds to the entire source code—the characters at offsets 0 through 114, inclusive, or the interval $[0, 114]$. The field declaration `int field = 0;` corresponds to the interval $[14, 30]$. The post-increment `field++;` becomes $[70, 82]$.

Since AST nodes can be represented as intervals, we can use these intervals to describe the semantic edges of a program graph. For example, the name binding edge from the post-increment to the field declaration becomes $[70, 82] \triangleright_B [14, 30]$. (The interval representation of the program graph in Figure 6.1 is shown in Figure 6.2(a).)

During a refactoring transformation, it is possible to track what regions of the original source code are deleted or replaced, as well as where new source code is inserted. These textual regions are contained in the affected forests. Since we know exactly how many characters were added or deleted at what positions, then for any character *outside* these regions, it is possible to determine exactly where that character should occur in the transformed program. Suppose we have a (partial) function *newOffset*(n) that can determine this value, for a given character offset n in the original program.

Initial Model (a)	**Norm. Initial** (b)	**Norm. Deriv.** (c)	**Deriv. Model** (d)
$[\ 74,\ 78]\triangleright_B[\ 14,\ 30]$	$*\quad\triangleright_B\quad *$		
$[\ 65,\ 65]\triangleright_B[\ 46,\ 60]$	$[61,\ 61]\triangleright_B[42,\ 56]$	$[61,\ 61]\triangleright_B[42,\ 56]$	$[61,\ 61]\triangleright_B[42,\ 56]$
		$*\quad\triangleright_B[42,\ 56]$	$[70,\ 70]\triangleright_B[42,\ 56]$
$[106,106]\triangleright_B[\ 46,\ 60]$	$[98,\ 98]\triangleright_B[42,\ 56]$	$[98,\ 98]\triangleright_B[42,\ 56]$	$[98,\ 98]\triangleright_B[42,\ 56]$
$[\ 31,113]\triangleright_C[\ 46,\ 60]$	$[27,105]\triangleright_C[42,\ 56]$	$[27,105]\triangleright_C[42,\ 56]$	$[27,105]\triangleright_C[42,\ 56]$
$[\ 46,\ 60]\triangleright_C[\ 61,\ 69]$	$[42,\ 56]\triangleright_C[57,\ 65]$	$[42,\ 56]\triangleright_C[57,\ 65]$	$[42,\ 56]\triangleright_C[57,\ 65]$
$[\ 61,\ 69]\triangleright_C[\ 70,\ 82]$	$[57,\ 65]\triangleright_C\quad *$	$[57,\ 65]\triangleright_C\quad *$	$[57,\ 65]\triangleright_C[66,\ 74]$
$[\ 70,\ 82]\triangleright_C[\ 83,109]$	$*\quad\triangleright_C[75,101]$	$*\quad\triangleright_C[75,101]$	$[66,\ 74]\triangleright_C[75,\ 101]$
$[\ 83,109]\triangleright_C[\ -1,\ -1]$	$[75,101]\triangleright_C[-1,\ -1]$	$[75,101]\triangleright_C[-1,\ -1]$	$[75,101]\triangleright_C[-1,\ -1]$
$[\ 46,\ 60]\triangleright_D[\ 61,\ 69]$	$[42,\ 56]\triangleright_D[57,\ 65]$	$[42,\ 56]\triangleright_D[57,\ 65]$	$[42,\ 56]\triangleright_D[57,\ 65]$
$[\ 61,\ 69]\triangleright_D[106,106]$	$[57,\ 65]\triangleright_D[98,\ 98]$	$[57,\ 65]\triangleright_D[66,\ 74]$	$[57,\ 65]\triangleright_D[66,\ 74]$
		$*\quad\triangleright_D[98,\ 98]$	$[66,\ 74]\triangleright_D[98,\ 98]$

Figure 6.2: Textual interval models. These correspond to the program graph from Figure 6.1 when `field` is renamed to `i`

Now, suppose we take each edge of the derivative model, and if an endpoint is contained in the affected forest, we replace that interval with $*$. We will call the result the *normalized derivative model.* Then, we can take each edge of the initial program graph and use the *newOffset* function to determine the equivalent edge in the normalized derivative model, likewise replacing endpoints in the affected forest with $*$. We will call this the *normalized initial model.*

If the normalized models are stored as sets (eliminating duplicate edges), then each edge in the initial model corresponds to exactly one edge in the normalized initial model, and each edge in the derivative model corresponds to exactly one edge in the normalized derivative model. Now, an edge in the initial model is equivalent to an edge in the derivative model (in the notation of the previous section, $equiv(e, e')$ if, and only if, their corresponding edges in the initialized models are *equal.* By storing the edges of the normalized models in appropriate data structures (e.g., hash sets), we can determine in $O(1)$ time if a particular edge occurs in either model.

An example is shown in Figure 6.2. Suppose, in the Java program in Figure 6.1, we attempt to rename the field declaration from `field` to `i`. The transformation is simple: replace the five characters `field` at offsets 20–24 (the declaration) and 74–78 (the reference) with the one-character string `i`. Since four characters are deleted in each case,

$$newOffset(n) = \begin{cases} n & \text{if } n \leq 19 \\ n-4 & \text{if } 25 \leq n \leq 73 \\ n-8 & \text{if } 79 \leq n. \end{cases}$$

The affected forest consists of the field declaration and the second post-increment (initial intervals $[14, 30]$ and $[70, 82]$, derivative intervals $[14, 26]$ and $[66, 74]$). Since `field++` changes to `i++`, the name binding edge for the field reference disappears and becomes a reference to the local variable `i` in the derivative model. Also, a new def-use chain is introduced. Since the renaming transformation would not preserve name bindings (or du-chains, for that matter), it should not be allowed to proceed.

Implementing the preservation analysis using textual intervals, rather than directly on the program graph, has a number of advantages. It allows the preservation analysis to be highly decoupled from the refactoring tool's program representation, which makes it more easily reusable.

It is fairly space-efficient, since semantic edges are represented as tuples of integers. Also, there is a fairly natural way to display errors: highlight the affected region(s) of the source code.

6.7.2 Detailed Construction

We will now turn to the details of implementing a textual interval-based preservation analysis. (Readers uninterested in these details may skip ahead to Section 6.8.)

We begin with some preliminary definitions. We will denote textual regions using *right half-open integer intervals*. Using half-open intervals allows for many different empty intervals; e.g., $[3, 3)$ denotes an empty interval at position 3, while $[5, 5)$ denotes an empty interval at position 5. This will become important momentarily when we introduce *replacements*.

Definition 29. *A **right half-open interval over** $\mathbb{Z}$ (or simply "interval") is an ordered pair denoted*

$$I = [\underline{I}, \overline{I}),$$

*where $\underline{I}, \overline{I} \in \mathbb{Z}$ and $\underline{I} \leq \overline{I}$. $\underline{I}$ is called the **lower bound** of I, and $\overline{I}$ is called the **upper bound** of I. The set of all such intervals will be denoted $\mathbb{IZ}$. An interval $[\underline{I}, \overline{I})$ intuitively corresponds to the set $[\![I]\!] \coloneqq \{\underline{I}, \underline{I} + 1, \ldots, \overline{I} - 1\}$, so we will adopt the following notations from set theory. Let $n \in \mathbb{Z}$.*

- *$n \in I$ denotes $\underline{I} \leq n < \overline{I}$.*
- *$I \subseteq J$ denotes $\underline{J} \leq \underline{I} \leq \overline{I} \leq \overline{J}$.*
- *$I \subset J$ denotes $I \subseteq J \wedge I \neq J$.*
- *$|I|$ denotes $\max(\overline{I} - \underline{I}, 0)$.*
- *$I \cap J$ denotes the set $\{\max(\underline{I}, \underline{J}), \max(\underline{I}, \underline{J}) + 1, \ldots, \min(\overline{I} - 1, \overline{J} - 1)\}$.*

As stated earlier, a textual interval model requires that every node in an AST be mapped to a unique textual region of the source code, *and that this textual region be mapped back to a unique AST node.* Formally, this means that there must be a *textual mapping function* defined on the tree, as follows.[5]

Definition 30. *For a directed tree T with vertex set V, a **textual mapping function***

$$rgn : V \xrightarrow{1\text{–}1} \mathbb{IZ}$$

is an injective (1–1) function with the following properties, for $v, u \in V$.

1. *If u is a descendent of v in T, then $rgn(u) \subset rgn(v)$.*
2. *If u is a sibling of v in T, then $rgn(u) \cap rgn(v) = \varnothing$.*

Predicting Offsets

When the AST is modified during a refactoring transformation, the modified subtrees of the AST comprise the affected forest. The textual mapping function allows these modified subtrees to be mapped to textual regions. So, it is possible to track what regions of the original source code are

[5] Actually, in practice, the requirement is not so strict: Two tree nodes can correspond to the same textual region as long as only one of them can ever occur as the endpoint of a semantic edge in a program graph.

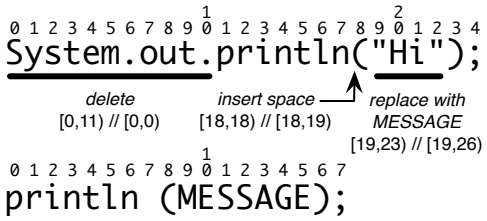
deleted or replaced, as well as where new source code is inserted, based on the changes made to the AST.

In the author's implementation, this is accomplished using the Observer pattern [39]: The preservation analysis registers an observer on the relevant AST(s), so it can be informed when AST nodes are added, modified, or deleted. The observer uses the textual mapping function to map the changed part(s) of the AST to textual regions.

In a textual interval-based analysis, the changes made to the source code are represented as a set of *nonoverlapping replacements*. Each replacement describes an AST node (equivalently, a region of source code) that was added, deleted, or modified.

Definition 31. *A **replacement** is an ordered pair denoted by $J /\!\!/ K$, where $J, K \in \mathbb{IZ}$ and $\underline{J} = \underline{K}$. We will let R denote the set of all replacements.*

For example, the string "`System.out.println("Hi");`" can be transformed into the string "`println (MESSAGE);`" using three replacements, as shown below.



Definition 32. *A set $S \subseteq R$ of replacements is **nonoverlapping** iff*

$$\bigcap_{J /\!\!/ K \in S} J = \varnothing.$$

Determining whether two intervals overlap is quite simple, due to the following theorem.

Theorem 5. *(Efficient Computation of Interval Overlap)*
Given $I, J \in \mathbb{IZ}$,

$$[\![I]\!] \cap [\![J]\!] = \varnothing \text{ iff } (\overline{I} \leq \underline{J} \vee \underline{I} \geq \overline{J}).$$

Proof.

	$[\![I]\!] \cap [\![J]\!] = \varnothing$	
iff	$\nexists i.\ (i \in [\![I]\!] \wedge i \in [\![J]\!])$	
iff	$\forall i.\ \neg(i \in [\![I]\!] \wedge i \in [\![J]\!])$	
iff	$\forall i.\ (i \notin [\![I]\!] \vee i \notin [\![J]\!])$	(By de Morgan's Law)
iff	$\forall i.\ (i \in [\![I]\!] \Rightarrow i \notin [\![J]\!])$	(By Law: $\neg P \vee \neg Q = P \Rightarrow \neg Q$)
iff	$\forall i \in [\![I]\!].\ i \notin [\![J]\!]$	
iff	$\forall i \in [\![I]\!].\ (i < \underline{J} \wedge i \geq \overline{J})$	
iff	$\overline{I} \leq \underline{J} \vee \underline{I} \geq \overline{J}.$	□

□

Since we know exactly how many characters were added or deleted at what positions, then for any character *outside* these regions, it is possible to determine exactly where that character should occur in the transformed program.

Definition 33. *Given $n \in \mathbb{Z}$ and a set S of nonoverlapping replacements, the **new offset of** n (according to S) is given by the function*

$$newOffset_S(n) \coloneqq n + \sum_{J/\!/K \in S} adjust_{J/\!/K}(n)$$

where

$$adjust_{J/\!/K}(n) \coloneqq \begin{cases} 0 & \text{if } n < \overline{J} \\ |K| - |J| & \text{if } n \geq \overline{J}. \end{cases}$$

Intuitively, any single replacement $J/\!/K$ inserts $|K| - |J|$ characters, so all of the characters after the affected interval will be shifted by that many characters. The summation simply computes the total amount by which n will be adjusted after every replacement has been applied. In the example above, the final semicolon was at offset 24 in the original string and offset 17 in the modified string. This is predicted by the newOffset function as follows. The set S consists of three nonoverlapping replacements:

$$S \coloneqq \{ \quad [0,11)/\!/[0,0), \quad [18,18)/\!/[18,19), \quad [19,23)/\!/[19,26) \quad \}.$$

Then, we have

$$\begin{aligned} \text{adjust}_{[0,11)/\!/[0,0)} &= -11 \\ \text{adjust}_{[18,18)/\!/[18,19)} &= 1 \\ \text{adjust}_{[19,23)/\!/[19,26)} &= 3 \\ \sum_{J/\!/K \in S} \text{adjust}_{J/\!/K}(24) &= -7 \\ \text{newOffset}_S(24) &= 24 + -7 = 17. \end{aligned}$$

Interval Models

Thus far, we have seen that a textual mapping function is used to map between AST nodes and textual intervals; the affected forest can be represented as a set of nonoverlapping replacements; and the newOffset function can determine, for any character occurring outside the affected forest, where that character will be located in the transformed program. Now, we will show how to construct semantic models based on these results.

Since an interval uniquely determines an AST node, a semantic edge in the AST can be represented as an ordered triple consisting of (1) the interval corresponding to the head AST node, (2) the edge type, and (3) the interval corresponding to the tail AST node. An *interval model*, then, is simply the set of all semantic edges in a program graph.

Definition 34. *Given a set Σ_E of edge types, an **interval model** is a finite subset of $\mathbb{IZ} \times \Sigma_E \times \mathbb{IZ}$. The members of this set are called **(semantic) edges.** An edge (I, ℓ, J) will be denoted by $I \triangleright_\ell J$.*

The **initial model** is the interval model constructed from the original source code, and the **derivative model** is the interval model constructed from the modified source code. To check for preservation, we must construct *normalized* initial and derivative models. This means that we must

be able to determine whether an endpoint of an edge lies within the affected forest so that we can replace that endpoint with $*$.

Recall that the affected forest is denoted by a set of nonoverlapping replacements. For a replacement $J \parallel K$, the interval J describes the offsets within the original source code that are affected.

Definition 35. *The **affected initial interval** of a replacement $J \parallel K$ is given by*

$$aii(J \parallel K) \coloneqq J.$$

Now, we can determine what part(s) of the *modified* source code lie within the affected forest using the newOffset function.

Definition 36. *Let S be a set of nonoverlapping replacements. The **affected derivative interval** of a replacement $J \parallel K \in S$ is given by*

$$adi_S(J \parallel K) \coloneqq \left[newOffset_{S-\{J\parallel K\}}(\underline{K}), newOffset_{S-\{J\parallel K\}}(\underline{K}) + |K|\right).$$

We can collapse the edges in the derivative model to construct the normalized derivative model.

Definition 37. *Given a set S of nonoverlapping replacements and an interval model D (the* derivative model*), the **normalized derivative model** is the interval model*

$$dnorm_S(D) \coloneqq \{collapse_S(I) \triangleright_t collapse_S(J) \mid I \triangleright_t J \in D\}$$

where

$$collapse_S(I) \coloneqq \begin{cases} * & \text{if } \exists r \in S.\ I \subseteq adi_S(r) \\ I & \text{otherwise.} \end{cases}$$

Constructing the normalized initial model is slightly more difficult. We know what regions of the original source code (and, thus, what AST nodes) lie within the affected forest, and for any character *outside* these regions, we have a function to determine exactly where that character should occur in the transformed program. So, for any interval in the original program, we can predict what the equivalent interval will be in the normalized derivative model: If it lies within the affected forest, it will be $*$; otherwise, we can determine the exact bounds using the newOffset function.

Definition 38. *Given a set S of nonoverlapping replacements and an interval model I (the* initial model*), the **normalized initial model** is the interval model*

$$inorm_S(I) \coloneqq \{predict_S(I) \triangleright_t predict_S(J) \mid I \triangleright_t J \in D\}.$$

where

$$predict_S(I) \coloneqq \begin{cases} * & \text{if } \exists J \parallel K \in S.\ I \subseteq aii(J \parallel K) \\ \left[newOffset_S(\underline{I}), newOffset_S(\overline{I} - 1) + 1\right) & \text{otherwise.} \end{cases}$$

Perhaps the most surprising part of the above definition is the presence of $\cdots - 1) + 1$. This ensures that, if $\overline{I}$ is the start of the affected derivative interval, it is kept alone and not extended to the

right side of the interval. For example, suppose a statement S covers offsets 10–20, inclusive; this would be represented by the interval $[10, 21)$. If a new statement is inserted after S, this should not change S's textual interval; it should still be $[10, 21)$. However, suppose S (and the new statement) are contained in a compound statement covering the interval $[9, 22)$. Then the addition of the new statement *should* change the textual interval for the compound statement, since the new statement was added to it.

Performing the Preservation Analysis

Once the normalized initial and derivative models have been constructed according to the above definitions, the preservation analysis is a straightforward implementation of the algorithm from Section 6.6. Let E denote the normalized initial model (i.e., a set of ordered triples as defined in Definition 38) and E' the normalized derivative model (Definition 37). Define

$$\begin{aligned} \mathbf{type}(v \triangleright_t u) &\coloneqq t \\ \mathbf{classify}(v \triangleright_\ell u) &\coloneqq \begin{cases} \leftarrow & \text{if } v \neq * \wedge u = * \\ \rightarrow & \text{if } v = * \wedge u \neq * \\ \circlearrowleft & \text{if } v = * \wedge u = * \\ \times & \text{if } v \neq * \wedge u \neq *. \end{cases} \end{aligned}$$

Then the preservation analysis proceeds as follows.

for each $e \in E$
  if *rule*(**type**(e), **classify**(e)) is $\supseteq$ or $=$
    but $e \notin E'$, then
      FAIL
for each $e' \in E'$
  if *rule*(**type**(e'), **classify**(e')) is $\subseteq$ or $=$
    but $e' \notin E$, then
      FAIL
otherwise, PASS

Summary

In sum, a differential precondition checker based on interval models operates as follows.

1. Analyze source code and produce an AST.
2. Construct the initial model I from the AST, performing any requisite static analyses.
3. Validate user input.
4. Perform the transformation, recording the AST changes as a set of nonoverlapping replacements S.
5. Detect compilability errors, and if appropriate, abandon the refactoring.

6. Construct the derivative model D from an AST for the modified source code, again performing any requisite static analyses.
7. Construct the normalized initial model $\text{inorm}_S(I)$ and the normalized derivative model $\text{dnorm}_S(D)$.
8. Apply the preservation analysis algorithm as described above.
9. If the preservation analysis succeeds, modify the user's source code. Otherwise, abandon the refactoring.

6.8 Evaluation

In the preceding sections, we used Safe Delete, Pull Up Method, and Rename as examples to illustrate differential precondition checking, and we also sketched a linear-time algorithm for performing the preservation analysis and argued for its language independence. But is this technique effective in practice? This section will focus on three questions:

Q1. *Expressivity.* Are the preservation specifications in §6.3 sufficient to implement the most common automated refactorings?

Q2. *Productivity.* When refactorings are implemented as such, does this actually reduce the number of preconditions that must be explicitly checked?

Q3. *Performance.* When preconditions are checked differentially, what are the performance bottlenecks? How does the performance compare to a traditional implementation?

For our evaluation, we implemented a differential precondition checker which we reused in three refactoring tools: (1) Photran, a popular Eclipse-based IDE and refactoring tool for Fortran; (2) a prototype refactoring tool for PHP 5; and (3) a similar prototype for BC.

6.8.1 Q1: Expressivity

To effectively answer question Q1, we must first identify what the most common automated refactorings are. The best empirical data so far are reported by Murphy-Hill et al. [65]. Table 6.1 shows several of the top refactorings; the *Eclipse JDT* column shows the popularity of each refactoring in the Eclipse JDT according to [65, Table 1, "Everyone"]. For comparison, we have also listed the availability of these refactorings in other popular refactoring tools for various languages.

We selected 18 refactorings (see Table 6.2): 7 for Fortran, 9 for BC, and 4 for PHP. Five of these refactorings are Fortran or BC analogs of the five most frequently-used in Eclipse JDT. Nine others are support refactorings, necessitated by decomposition. The remaining refactorings were chosen for other reasons. Add Empty Subprogram and Safe Delete were the first to be implemented; they helped shape and test our implementation. Introduce Implicit None preserves name bindings in an "interesting" way. Pull Up Method required us to model method overriding and other class hierarchy issues in program graphs.

It is worth noting that many popular IDEs provide fewer than 10 refactorings, including Apple Xcode (8 refactorings), Microsoft Visual Studio (6), and Zend Studio (4). So while generality is important and desirable (certainly, a technique that works for 18 refactorings will apply to many

Refactoring	Eclipse JDT (Rank)	IntelliJ IDEA[1]	IntelliJ ReSharper[2]	MS Visual Studio[3]	Eclipse CDT	Visual Assist X[4]	Apple Xcode[5]	Zend Studio[6]
Rename	1	●	●	●	●	●	●	●
Extract Variable	2	●	●	○	●	○	○	●
Move	3	●	●	○	○	○	○	●
Extract Method	4	●	●	●	●	●	●	●
Change Signature	5	●	●	●	○	●	○	○
Pull Up Method	11	●	●	○	○	●	●	○

Legend: ● *Included* ○ *Not Included*

[1] http://www.jetbrains.com/idea/features/refactoring.html
[2] http://www.jetbrains.com/resharper/features/code_refactoring.html
[3] http://msdn.microsoft.com/en-us/library/719exd8s.aspx
[4] http://www.wholetomato.com/products/featureRefactoring.asp
[5] http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/XcodeWorkspace/150-Refactoring/refactoring.html
[6] http://www.zend.com/en/products/studio/features#refactor

Table 6.1: Automated refactorings in popular tools.

Fortran	1.	Rename
	2.	Move
	3.	*Introduce* U
	4.	Change Function Signature
	5.	Introduce I N
	6.	Add Empty Subprogram
	7.	Safe Delete
BC	8.	Extract Local Variable
	9.	*Add Local Variable*
	10.	*Introduce Block*
	11.	*Insert Assignment*
	12.	*Move Expression Into Assignment*
	13.	Extract Function
	14.	*Add Empty Function*
	15.	*Populate Unreferenced Function*
	16.	*Replace Expression*
PHP	17.	Pull Up Method
	18.	*Copy Up Method*

Table 6.2: Refactorings evaluated.

others), expediting and improving the implementation of a few common refactorings is equally important, perhaps even more so.

We wrote detailed specifications of all 18 refactorings in a technical report [70].[6] These specifications have also been included as Appendix B of this dissertation. Each specification describes both the traditional and the differential version of the refactoring, both at a level of detail sufficient to serve as a basis for implementation. (Several undergraduate interns working on Photran implemented refactorings based on our specifications.) The style of the specifications is similar to the Pull Up Method example from §6.3 but more precise. For example, the Fortran refactoring specifications use the same terminology as the Fortran 95 ISO standard.

[6] We also published the traditional versions of the Fortran refactoring specifications in ACM Fortran Forum [69].

We divided refactorings among the three languages as follows. For all of the refactorings that rely primarily on name binding preservation, we targeted Fortran, since it has the most complicated name binding rules. We targeted flow-based refactorings for BC: It contains functions, scalar and array variables, and all of the usual control flow constructs, but it is a much smaller and simpler language than either Fortran or PHP. This kept the specifications of these (usually complex) refactorings to a manageable size without sacrificing any of the essential preconditions. The one object-oriented refactoring targeted PHP 5.

We implemented a differential precondition checker (following §6.7) and used it to implement differential refactorings in the three refactoring tools, following our detailed specifications. For BC and PHP, we implemented refactorings as listed in Table 6.2. Since there are no comparable refactoring tools for these languages, we could not perform differential testing. However, we ported several relevant unit tests from the Eclipse CDT and JDT, as well as two informal refactoring benchmarks [77, 78]. For Fortran, we implemented differential versions of Rename, Introduce Implicit None, Add Empty Subprogram, and Safe Delete. Photran included traditional versions of these refactorings, with fairly extensive unit tests, so we were able to reuse the existing test cases to test the differential implementations.

6.8.2 Q2: Productivity

For each of the 18 refactorings, we chose a target language that would provide a challenging yet representative set of preconditions. This brings us to our second research question: Does using a differential engine reduce the number of preconditions that must be explicitly checked? To answer this question, we needed to be able to compare traditional and differential forms of the same refactorings. We also needed to be able to quantify the "amount of precondition checking" required for each refactoring.

To do this, we looked used the detailed specifications in Appendix B (taken from our a technical report [70]). Each specification describes both the traditional and the differential version of the refactoring. We wrote these specifications with the specific intent to provide a "fair" comparison between the traditional and differential versions of the refactorings.[7] We broke down the preconditions for each refactoring into steps, mimicking an imperative implementation, and factored out duplication among refactorings. Assumptions about analysis capabilities were modest—roughly equivalent to a compiler front end coupled with a cross-reference database.

A summary of the refactoring specifications is shown in Table 6.3. Following the name of the refactoring, the next several columns enumerate all of the preconditions in our specifications and indicate which ones were eliminated by the use of the differential engine. The next two columns attempt to quantify the "amount of precondition checking" involved in each refactoring. A precondition such as "introducing *X* will preserve name bindings" is far more complicated than a precondition like "*X* is a valid identifier," so we chose to look at the number of *steps* devoted to precondition checking in the specification of each refactoring. We attempted to make the granularity of each step consistent, so the total number of steps should be a relatively fair measure of the complexity of precondition checking. The *Trad.* column gives the total number of steps in all of

[7] We originally considered comparing the actual implementations (e.g., by measuring lines of code), but it is well known that such numbers could easily be skewed by details of the implementation not directly attributable to the use of the differential engine, making conclusive results more difficult for the reader to verify.

		Refactoring	Preconditions IN	SI	PP	RN	II	Other	Warn	Steps Trad.	Diff.	Xform
Fortran	1.	Rename	●	N/A	N/A	N/A	N/A	N/A	○	**29**	**1**	4
	2.	Move	N/A	N/A	●	●	N/A	○	N/A	**23**	**3**	58
	3.	*Introduce* U	●	N/A	N/A	N/A	N/A	●	N/A	**29**	**0**	5
	4.	Change Function Signature	N/A	N/A	N/A	N/A	N/A	○	N/A	**4**	**4**	13
	5.	Introduce I N	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**0**	**0**	6
	6.	Add Empty Subprogram	●	N/A	N/A	N/A	N/A	N/A	N/A	**27**	**0**	2
	7.	Safe Delete	N/A	●	N/A	N/A	N/A	N/A	N/A	**4**	**0**	5
BC	8.	Extract Local Variable	N/A	N/A	N/A	N/A	N/A	●	N/A	**3**	**0**	4
	9.	*Add Local Variable*	●	N/A	N/A	N/A	N/A	N/A	N/A	**20**	**0**	4
	10.	*Introduce Block*	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**0**	**0**	1
	11.	*Insert Assignment*	N/A	N/A	N/A	N/A	N/A	●	N/A	**1**	**0**	1
	12.	*Move Expression Into Assignment*	N/A	N/A	N/A	N/A	N/A	●	N/A	**2**	**2**	2
	13.	Extract Function	N/A	N/A	N/A	N/A	N/A	○	N/A	**1**	**1**	4
	14.	*Add Empty Function*	N/A	N/A	N/A	N/A	N/A	●	N/A	**1**	**0**	1
	15.	*Populate Unreferenced Function*	N/A	N/A	N/A	N/A	N/A	●	N/A	**2**	**0**	17
	16.	*Replace Expression*	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**0**	**0**	13
PHP	17.	Pull Up Method	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**0**	**0**	2
	18.	*Copy Up Method*	N/A	N/A	N/A	N/A	●	●	○	**12**	**1**	3

Legend: ● *Eliminated* ○ *Not eliminated* N/A *Not applicable*

Table 6.3: Summary of traditional and differential specifications of 18 refactorings. The precondition acronyms and step counts are described in the written specifications [70].

the refactoring's precondition checks in the traditional version; the *Diff.* column gives the number of steps in the differential version. These two numbers are also shown as bar graph in Figure 6.3. For comparison, the last column in the table (*Xform*) gives the number of steps in the transformation; this is not shown in the bar graph.

The data in Table 6.3 and Figure 6.3 support our hypothesis that using a differential refactoring engine reduces the amount of explicit precondition checking that must be performed. Notably:

1. *The number of precondition checking steps decreased for most refactorings, often substantially.* When no precondition checks were eliminated, it was generally because the preconditions were not related to compilability or preservation.

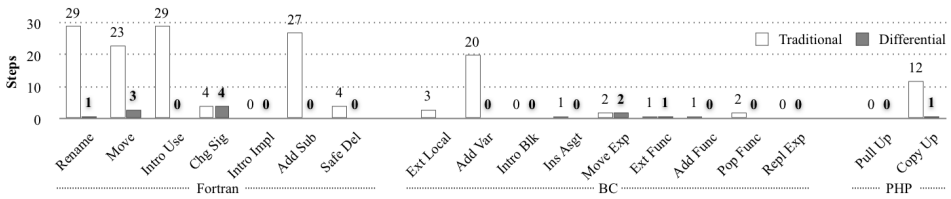


Figure 6.3: Precondition step counts from Table 6.2.

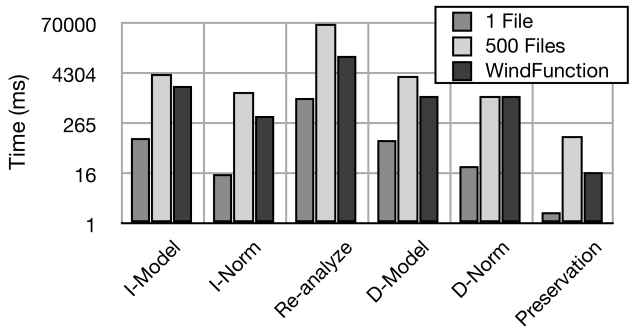


Figure 6.4: Rename performance measurements

2. *The precondition checks that were eliminated tended to be complex,* including a 25-step name binding preservation analysis for Fortran.
3. *The number of precondition checking steps never increased.* In fact, using a differential refactoring engine *cannot* increase the number of preconditions that must be checked. A differential engine provides a "free" check for compilability and preservation preconditions. In the worst case, a refactoring has none of these—in which case, it requires as many precondition checks as it would in a traditional refactoring engine. So, the number of precondition checking steps can only decrease (or stay the same).

6.8.3 Q3: Performance

Since a differential precondition checker's performance depends on the speed of the language-specific front end, as well what refactoring is being performed and what program is being refactored, it is difficult to make any broad claims about performance. In our experience, when a refactoring affects only one or two files in a typical application, the amount of time devoted to precondition checking is negligible. Most of the refactorings we implemented fall into this category. Performance becomes a concern only at scale, e.g., when a refactoring potentially affects every file in a project. We will use Photran's Rename refactoring as an illustrative example. Rename is the most expensive of the refactorings we implemented, since it can potentially change name bindings in every file in the program, it often makes many changes to a single file, and computing name bindings involves accessing a index/cross-reference database.

Figure 6.4 shows performance measurements[8] for the Rename refactoring on three Fortran programs. Two are examples intended to test scalability: "1 File" is a project with 500 subroutine definitions in a single file, while "500 Files" contains 1 subroutine in each of 500 files. "WindFunction" shows the results of renaming of the wind function in an atmospheric dispersion simulation (a production Fortran program consisting of about 53,000 LOC in 29 files, four of which were ultimately affected by the refactoring). From left to right, the performance measurements represent creation of the initial interval model, normalization of this model, running the front end to re-analyze the modified code, construction of the derivative interval model, normalization of this model, and, finally, the preservation analysis.

[8]The tests were performed on a 2 GHz Intel Core 2 Duo (MacBook), Java 1.6.0_24, with the JVM heap limited to 512 MB.

Note the logarithmic scale on the y-axis: In all three cases, the performance bottleneck was, by far, the *Re-analyze* measurement—i.e., the amount of time taken for the front end to analyze the modified program and recompute name bindings. This was generally true for other refactorings as well. It is not particularly surprising: When an identifier in one file can refer to an entity in another file, computing name bindings involves populating and accessing a cross-reference database.

In our experience, differential precondition checking is not as fast as traditional precondition checking, but its speed is acceptable. After all, the amount of time it requires is essentially the amount of time the front end takes to analyze the affected files. In the WindFunction example, differential precondition checking took about 9 seconds, while traditional checks took just over 1 second. Photran's name binding analysis is not particularly fast, and its traditional Rename refactoring has been heavily optimized over the course of six years to compensate. In contrast, for the refactorings which made localized changes to only one or two files, the time devoted to precondition checking was unnoticeable.

6.9 Limitations

Our preservation analysis has two notable limitations.

First, it *assumes* that, if a replacement subtree interfaces with the rest of the AST in an expected way, it is a valid substitute for the original subtree. It is the refactoring developer's responsibility to ensure that this assumption is appropriate. For example, replacing every instance of the constant 0 with the constant 1 would almost certainly break a program, but our analysis would not detect any problem, since this change would not affect any edges in a typical program graph. However, the refactoring developer should recognize that name bindings, control flow, and du-chains do not model the conditions under which 1 and 0 are interchangeable values.

Second, for our preservation analysis to be effective, the "behavior" to preserve must be modeled by the program graph. There are several cases where this is unlikely to be true, including the following.

Interprocedural data flow. One particularly insidious example is illustrated by an Eclipse bug (186253) reported by Daniel et al. [31]. In this bug, Encapsulate Field reorders the fields in a class declaration, causing one field to be initialized incorrectly by accessing the value of an uninitialized field via an accessor method. In theory, this could be detected by a preservation analysis, as it is essentially a failure to preserve du-chains for fields among their initializers. Unfortunately, these would probably not be modeled in a program graph, since doing so would require an interprocedural analysis.

Constraint-based refactorings, such as Infer Generics [54]. These refactorings preserve invariants modeled by a system of constraints; a program graph is an unsuitable model.

Library replacements, such as replacing primitive `int` values with `AtomicInteger` objects in Java [33], or converting programs to use `ArrayList` instead of `Vector`. Program graphs generally model *language* semantics, not *library* semantics, and therefore are incapable of expressing the invariants that these refactorings maintain.

6.10 Conclusions & Future Work

In this chapter, we classified refactoring preconditions as ensuring input validity, compilability, and behavior preservation, and we proposed a technique for many compilability and preservation preconditions to be checked after transformation in a generic way. We showed that, if essential semantic relationships are treated as edges in a program graph, these edges can be classified based on their relationship to the modified subtree(s). The preservation requirements for common refactorings can be expressed by indicating, for each kind of edge, whether a subset or superset of those edges should be preserved. By exploiting an isomorphism between graph nodes and textual intervals, the preservation checking algorithm can be implemented in a way that is both efficient and language independent. We implemented this technique in a library and applied it to 18 refactorings for Fortran 95, PHP 5, and BC.

III

Conclusions

7

Conclusions and Future Work

Part I of this dissertation described how grammars can be annotated to generate a parser, abstract syntax tree, and syntactic rewriting infrastructure suitable for use in a refactoring tool. Part II discussed the semantic requirements of refactoring tools and introduced the concept of differential precondition checking. This chapter will illustrate how these techniques were applied to build a small refactoring tool for BC. It will also discuss directions for future research.

7.1 A Refactoring Tool for BC

7.1.1 Syntactic Infrastructure

The BC refactoring tool's parser and AST are generated from a Ludwig grammar which is just over 100 lines long. The grammar is based on the POSIX standard [47], along with some extensions to support GNU BC.

bc.ebnf

```
# Ludwig EBNF Grammar for BC
# IEEE Standard 1003.1-2008 with GNU Extensions
# Jeffrey L. Overbey (18 December 2009)

                <program> ::= input-items:<input-item>*

(superclass):<input-item> ::= <semicolon-list-newline>
                            |  <function>

 <semicolon-list-newline> ::= statements:<semicolon-list> -:NEWLINE

ASTStatementListNode(list):
         <semicolon-list> ::= (empty)
                            |  <statement>
                            |  <semicolon-list> ";" <statement>
                            |  <semicolon-list> ";"

ASTStatementListNode(list):
         <stmt-list> ::= (empty)
                            |  <statement>
                            |  <stmt-list> NEWLINE
                            |  <stmt-list> NEWLINE <statement>
                            |  <stmt-list>   ";"
```

```
                          |  <stmt-list>   ";"   <statement>

 (superclass):<statement> ::= <expr>                                        <= ASTExprStmtNode
                          |  STRING                                         <= ASTStringStmtNode
                          |  "break"                                        <= ASTBreakStmtNode
                          |  "quit"                                         <= ASTQuitStmtNode
                          |  "return" <return-expression>                   <= ASTReturnStmtNode
                          |  "for" "("
                                    init-expression:<expr> ";"
                                    test-expression:<rel-expr> ";"
                                    loop-expression:<expr>
                                    ")" -:NEWLINE?  <statement>             <= ASTForStmtNode
                          |  "if" "(" test-expression:<rel-expr> ")"
                                   -:NEWLINE?
                                   then-statement:<statement>
                                   (inline):<else-clause>?                  <= ASTIfStmtNode
                          |  "while" "("
                                   test-expression:<rel-expr>
                                   ")" -:NEWLINE?  <statement>              <= ASTWhileStmtNode
                          |  lbrace:"{" statements:<stmt-list> rbrace:"}"   <= ASTBlockNode
                          |  <print-statement>

            <else-clause> ::= "else" else-statement:<statement>

ASTPrintStmtNode:
        <print-statement> ::= "print" arguments:<print-stmt-argument>+ (separated-by) ","

(superclass):
    <print-stmt-argument> ::= <argument>
                          |  STRING               <= ASTPrintStringNode

               <function> ::= "define" name:LETTER
                                        "(" <define-list>? ")"
                                        -:NEWLINE?
                                        "{" -:NEWLINE
                                        <auto-define-list>?
                                        statements:<stmt-list> "}"

       <auto-define-list> ::= "auto" <define-list> -:NEWLINE
                          |  "auto" <define-list> ";"

            <define-list> ::= variables:<variable>+ (separated-by) ","

ASTVariableNode:
               <variable> ::= name:LETTER
                          |  name:LETTER is-array(bool):"[" "]"

  (superclass):<argument> ::= <variable>
                          |  <expr>

IExpression(superclass):
               <rel-expr> ::= <expr>
                          |  LHS:<expr>       op:REL_OP RHS:<expr>       <= ASTBinaryExprNode
                          |                   op:"!"    RHS:<rel-expr>   <= ASTUnaryExprNode
```

```
                          |  LHS:<rel-expr>   op:"&&"   RHS:<rel-expr>   <= ASTBinaryExprNode
                          |  LHS:<rel-expr>   op:"||"   RHS:<rel-expr>   <= ASTBinaryExprNode
                          |  "(" <rel-expr> ")"

IExpression(superclass):
      <return-expression> ::= (empty)                                    <= ASTEmptyReturnExprNode
                          |  <expr>

IExpression(superclass):
                    <expr> ::= <named-expr>
                          |  NUMBER                                      <= ASTNumberExprNode
                          |  "(" <expr> ")"
                          |  name:LETTER "("
                                 arguments:<argument>* (separated-by) ","
                                 ")"                                     <= ASTFunctionCallExprNode
                          |             op:"-"    RHS:<expr>       <= ASTUnaryExprNode
                          |  LHS:<expr> op:"+"    RHS:<expr>       <= ASTBinaryExprNode
                          |  LHS:<expr> op:"-"    RHS:<expr>       <= ASTBinaryExprNode
                          |  LHS:<expr> op:MUL_OP RHS:<expr>       <= ASTBinaryExprNode
                          |  LHS:<expr> op:"^"    RHS:<expr>       <= ASTBinaryExprNode
                          |  prefix-op:INCR_DECR <named-expr>      <= ASTIncrDecrExprNode
                          |  <named-expr> postfix-op:INCR_DECR     <= ASTIncrDecrExprNode
                          |  <named-expr> op:ASSIGN_OP <expr>      <= ASTAssignmentExprNode
                          |  "length" "(" argument:<expr> ")"      <= ASTLengthExprNode
                          |  "sqrt" "(" argument:<expr> ")"        <= ASTSqrtExprNode
                          |  "scale" "(" argument:<expr> ")"       <= ASTScaleExprNode
                          |  "read" "(" ")"                        <= ASTReadExprNode

IExpression(superclass):
                <named-expr> ::= name:LETTER                           <= ASTVariableNode
                          |  name:LETTER "[" argument:<expr> "]"   <= ASTArrayReferenceNode
                          |  name:"scale" <= ASTVariableNode
                          |  name:"ibase" <= ASTVariableNode
                          |  name:"obase" <= ASTVariableNode

   (skip) ::= "/*" ~ "*/" | [ \t]+ | "\\" "\r"? "\n" | "#"~("\r"? "\n")
  NEWLINE ::= "\r"? "\n"
   STRING ::= "\"" ~ "\""
   NUMBER ::= [0123456789ABCDEF]+
            |                     "." [0123456789ABCDEF]+
            |  [0123456789ABCDEF]+ "."
            |  [0123456789ABCDEF]+ "." [0123456789ABCDEF]+
   LETTER ::= [A-Za-z][A-Za-z0-9_]*
ASSIGN_OP ::= "=" | "+=" | "-=" | "*=" | "/=" | "%=" | "^="
   MUL_OP ::= "*" | "/" | "%"
   REL_OP ::= "==" | "<=" | ">=" | "!=" | "<" | ">"
INCR_DECR ::= "++" | "--"
```

Although the grammar is fairly short, BC contains user-defined functions, global and local variables (both scalars and arrays), control flow constructs borrowed from the C language (including an *if* statement, *for* loop, and *while* loop, as well as *break* and *return* statements), and a full complement of mathematical expressions (also borrowed from C).

Ludwig generates 51 files from this grammar, listed below. These include the lexer and parser

(*Lexer* and *BCParser*), BC-specific AST node classes (*ASTArrayReferenceNode*, *ASTAssignmentExprNode*, *IArgument*, *IExpression*, etc.), and several AST support classes (*ASTNode*, *IASTListNode*, *ASTVisitor*, etc.).

ASTArrayReferenceNode.java	ASTNode.java	ASTUnaryExprNode.java
ASTAssignmentExprNode.java	ASTNodePair.java	ASTVariableNode.java
ASTAutoDefineListNode.java	ASTNodeUtil.java	ASTVisitor.java
ASTBinaryExprNode.java	ASTNodeWithErrorRecoverySymbols.java	ASTWhileStmtNode.java
ASTBlockNode.java	ASTNumberExprNode.java	BCParser.java
ASTBreakStmtNode.java	ASTPrintStmtNode.java	GenericASTVisitor.java
ASTDefineListNode.java	ASTPrintStringNode.java	IASTListNode.java
ASTElseClauseNode.java	ASTProgramNode.java	IASTNode.java
ASTEmptyReturnExprNode.java	ASTQuitStmtNode.java	IASTObserver.java
ASTExprStmtNode.java	ASTReadExprNode.java	IASTVisitor.java
ASTForStmtNode.java	ASTReturnStmtNode.java	IArgument.java
ASTFunctionCallExprNode.java	ASTScaleExprNode.java	IExpression.java
ASTFunctionNode.java	ASTSemicolonListNewlineNode.java	IInputItem.java
ASTIfStmtNode.java	ASTSeparatedListNode.java	IPrintStmtArgument.java
ASTIncrDecrExprNode.java	ASTSqrtExprNode.java	IStatement.java
ASTLengthExprNode.java	ASTStatementListNode.java	Lexer.java
ASTListNode.java	ASTStringStmtNode.java	Token.java

The entrypoint for the generated code is generally the *BCParser#parse* method, which parses a given file and returns an abstract syntax tree (specifically, an *ASTProgramNode*, which is the root of every BC AST). This AST can be traversed using individual nodes' getter methods or by using an *ASTVisitor*. The original source code can be reproduced verbatim from the AST by calling the *toString* method on the root node. This is used in the test suite, for example: A file's original source code is compared character-by-character against the source code reproduced from the AST to ensure that the AST does not "lose" any information.

ParserTests.java (excerpt)

```
1    protected void test(File file) throws Exception {
2        String expectedSourceCode = StringUtil.read(file); // Get contents of file as a string
3        ASTProgramNode ast = new BCParser().parse(new Lexer(file)); // Parse file into AST
4        String actualSourceCode = ast.toString(); // Reproduce source code from AST
5        assertEquals(expectedSourceCode, actualSourceCode);
6    }
```

7.1.2 Semantic Infrastructure

After the syntactic infrastructure is stable, we can change our focus to the semantic infrastructure. The BC refactoring tool has three analyses: a name binding analysis, a control flow analysis, and a reaching definitions (def-use) analysis. All three analyses have a similar structure; we will use the name binding analysis as an example.

The entrypoint for the name binding analysis is the method *BCNameBindingAnalysis#analyze*. It takes three arguments: a filename, an AST, and a semantic model. A semantic model is simply a Strategy object [39] with one method, *addEdge*, that is invoked for every name binding in the AST.

BCNameBindingAnalysis.java (excerpts)

```
1  public final class BCNameBindingAnalysis extends ASTVisitor {
2
3      public static void analyze(String filename, ASTProgramNode ast, ISemanticModel semanticModel) {
```

```
4           ast.accept(new BCNameBindingAnalysis(filename, semanticModel));
5       }
```

The *BCNameBindingAnalysis* is a Visitor [39] which traverses the AST in a syntax-directed fashion, building a symbol table in the same way as a compiler. However, the symbol table is only used internally. Name binding information is available to the outside world through two mechanisms:

1. The AST is attributed with name bindings.
2. The *ISemanticModel* is informed of every name binding discovered in the file.

These are redundant, of course, but they serve different purposes. Refactoring transformations access the AST directly, so they can acquire name binding information by querying the AST nodes themselves. On the other hand, the preservation analysis is provided by the Rephraser Engine and has no knowledge of the AST structure. So, the preservation analysis implements the *ISemanticModel* interface

```
public interface ISemanticModel {
    void addEdge(int headOffset, int headLength, int tailOffset, int tailLength, int type);
}
```

which allows it to be informed of name bindings in terms of offsets/lengths.

```
6  public final class BCNameBindingAnalysis extends ASTVisitor {
7      public static enum Namespace { FUNCTION, VARIABLE; }
8      private SymbolTable<Namespace, Token> symtab;
9      ...
10     private void bind(Token reference, Namespace namespace, boolean isDeclaration, EdgeType edgeType) {
11         Token declaration = symtab.lookup(namespace, reference.getText());
12         reference.setAttribute(AttributeType.BINDING, declaration);
13         ...
14         if (reference != declaration) {
15             semanticModel.addEdge(
16                 reference.getOffset(), reference.getLength(),
17                 declaration.getOffset(), declaration.getLength(),
18                 edgeType.ordinal());
19         }
20     }
21 }
```

7.1.3 Refactorings

Now, we can turn our attention to constructing refactorings. The Rephraser Engine provides an interface which all refactorings are expected to implement.

```
public interface IRefactoring {
    ...
    String validateSelection();
    String validateUserInput();
    RefactoringResult transform();
}
```

The protocol is straightforward. First, the *validateSelection* method is invoked to determine whether this refactoring can be applied (e.g., the Rename refactoring can be applied only if the user has selected an identifier). This method returns a human-readable error message, or `null` if the selection is valid. If the selection is valid, a user input dialog is displayed (if necessary). Then, the *validateUserInput* method is invoked to validate the input supplied by the user. After this input is validated, the *transform* method is invoked to perform the refactoring.

In the BC refactoring tool, all refactorings are similar: They are invoked from a text editor, they operate on a single file (i.e., a single AST), and they use differential precondition checking. So, we will construct an abstract class called *BCRefactoring* which overrides the *transform* method to accommodate these similarities. Its source code is shown below.

The *BCRefactoring#transform* method begins by constructing a *PreservationAnalysis*, constructing the initial model, and adding an Observer [39] to the AST (the observer is notified every time the AST is modified, so it can determine what regions of source code changed during the refactoring). The *transform* method uses the Template Method pattern [39] to delegate the details of the transformation to its subclasses. Specifically, it invokes the abstract *modifyAST* method, which every concrete subclass is required to override. After the transformation is complete, it asks the observer (an *ASTChangeTracker*) to determine a set of replacements and provide this to the *PreservationAnalysis*. It constructs the derivative model from the modified AST. Finally, it asks the *PreservationAnalysis* to check for preservation, again using the Template Method pattern so that each individual refactoring can supply its own preservation rule (*getPreservationRule*).

BCRefactoring.java (excerpt)

```
1  public abstract class BCRefactoring extends EditorRefactoring {
2      protected ASTProgramNode ast;
3
4      @Override public final RefactoringResult transform() {
5          PreservationAnalysis preservationAnalysis = new PreservationAnalysis();
6
7          String initialSourceCode = getContentsOfFileInEditor();
8
9          preservationAnalysis.beginConstructingInitialModel();
10         constructProgramGraph(preservationAnalysis);
11
12         ASTChangeTracker changeTracker = new ASTChangeTracker();
13         ast.addObserver(changeTracker);
14
15         RefactoringResult result = new RefactoringResult();
16         String error;
17         try {
18             error = modifyAST(preservationAnalysis, result);
19         } catch (Exception e) {
20             error = "The refactoring could not be completed due to an internal error.\n\n"
21                     + e.getClass().getName() + ": " + e.getMessage();
22         }
23         if (error != null)
24             return new RefactoringResult(error);
25         result.setFileContents(selection.getFileInEditor(), ast.toString());
26
27         changeTracker.addReplacementsTo(preservationAnalysis, ast);
28
```

```
29              recomputeTokenOffsets(ast);
30              preservationAnalysis.beginConstructingDerivativeModel();
31              constructProgramGraph(preservationAnalysis);
32
33              String derivativeSourceCode = result.getChanges().get(selection.getFileInEditor());
34
35              preservationAnalysis.checkPreservation(getPreservationRule()).logErrorsTo(
36                      result.getErrorWarningLog(),
37                      selection.getFileInEditor(),
38                      initialSourceCode,
39                      derivativeSourceCode);
40
41              ASTChangeTracker.resetModifications(ast);
42
43              return result;
44          }
45
46          private void constructProgramGraph(PreservationAnalysis preservationAnalysis) {
47              String filename = selection.getFileInEditor().getPath();
48              BCNameBindingAnalysis.analyze(filename, ast, preservationAnalysis);
49              BCControlFlowAnalysis.analyze(filename, ast, preservationAnalysis);
50              BCDefUseAnalysis.analyze(filename, ast, preservationAnalysis);
51          }
52
53          protected abstract String modifyAST(PreservationAnalysis preservationAnalysis,
54                                              RefactoringResult result) throws Exception;
55
56          protected abstract PreservationRule getPreservationRule();
57
58          ...
59      }
```

Now, implementing individual refactorings is straightforward. Each refactoring subclasses *BCRefactoring* and overrides the following methods.

```
public abstract String validateSelection();
public abstract String validateUserInput();
protected abstract String modifyAST(PreservationAnalysis preservationAnalysis,
                                    RefactoringResult result) throws Exception;

protected abstract PreservationRule getPreservationRule();
```

Add Local Variable is a simple example. Its complete implementation requires only about 50 lines of Java. The Ludwig-generated code provides all of the functionality necessary to construct, traverse, and modify ASTs. The Rephraser Engine provides the user interface. And the differential precondition checker (the preservation analysis) ensures that the addition of the new local variable is legal.

AddLocalVariableRefactoring.java (imports excluded)

```
1   public final class AddLocalVariableRefactoring extends BCRefactoring {
2       private ASTFunctionNode function = null;
3       private String name = null;
4
5       @Override public String validateSelection() {
```

```
6           return findSelectedFunction();
7       }
8
9       private String findSelectedFunction() {
10          this.function = findNode(ast, ASTFunctionNode.class, selection.getOffset(), selection.getLength());
11          if (function == null)
12              return "Please select a function.";
13          return null;
14      }
15
16      @UserInputString(label = "Enter Variable Name:")
17      public void setName(String name) {
18          this.name = name;
19      }
20
21      @Override public String validateUserInput() {
22          if (!name.matches("[A-Za-z][A-Za-z0-9_]*"))
23              return "The new name must begin with a letter and must consist of only letters, digits, and underscores.";
24          else
25              return null;
26      }
27
28      @Override protected String modifyAST(PreservationAnalysis preservation, RefactoringResult result) {
29          if (function.getAutoDefineList() != null) {
30              ASTSeparatedListNode<ASTVariableNode> variables =
31                  function.getAutoDefineList().getDefineList().getVariables();
32              ASTVariableNode var = new ASTVariableNode();
33              var.setName(new Token(Terminal.LETTER, name));
34              variables.add(new Token(Terminal.LITERAL_STRING_COMMA, ", "), var);
35          } else {
36              ASTDefineListNode defineList = new BCParser().parseAutoDefineList("auto " + name + "\n");
37              function.setAutoDefineList(autoDefineList);
38          }
39          return null;
40      }
41
42      @Override protected PreservationRule getPreservationRule() {
43          PreservationRule rules = new PreservationRule();
44          rules.preserveExact(EdgeType.FUNCTION_REFERENCE.ordinal());
45          rules.preserveExact(EdgeType.VARIABLE_REFERENCE.ordinal());
46          rules.preserveExact(EdgeType.CONTROL_FLOW.ordinal());
47          return rules;
48      }
49  }
```

7.2 Conclusions

In total, the BC refactoring prototype consists of about 10,000 lines of Java, not including the Rephraser Engine or unit tests. Of that, about 1,000 lines of code are dedicated to static analyses (name bindings and flow analysis). Another 1,000 lines are dedicated to refactorings (the tool contains the nine refactorings listed in the previous chapter, as well as a Rename refactoring). The

refactorings do not contain any code to explicitly check compilability or preservation preconditions; all such preconditions are handled by the differential precondition checker. The other 8,000 lines are generated from the Ludwig grammar presented earlier.

The Rephraser Engine library consists of about 10,000 lines of Java code (although not all of this is actually used in the BC refactoring tool). If this is included, then, the total amount of code in the tool is about 20,000 lines. Of this amount, 50% is contained in the Rephraser library, 40% is generated by Ludwig, and only 10% is actually hand-written.

While the BC tool is just a small, illustrative prototype, the author achieved similar results with Photran, a production refactoring tool for Fortran. As mentioned in Chapter 1, Photran's refactorings comprise about 10,000 lines of Java code. A 5,000-line Ludwig grammar generates about 90,000 lines of Java, and, again, the Rephraser Engine adds about 10,000 LOC. So, of the 110,000 total lines of Java code,[1] about 9% are contained in the Rephraser library, 82% are generated by Ludwig, and only about 9% are hand-written.

In all three refactoring tools—Photran, as well as the BC and PHP prototypes—the hand-written, language-specific code was kept to a very manageable size: just a few thousand lines of code. Even in Photran, most refactorings consist of only a few hundred lines of code. Overall, these results are very promising. By aggressively pushing language-independent behavior into the the Rephraser Engine library, and by continuously refining the search and rewriting API provided by the Ludwig-generated rewritable AST, this work has demonstrated that it is possible to significantly reduce the amount of code needed to implement a refactoring tool for a new language.

7.3 Future Work

There are several directions for future extensions of this work.

7.3.1 User Interface/Error Reporting

This dissertation has focused on the infrastructure underlying a refactoring tool; all three prototype tools use Eclipse (and the Rephraser Engine) to provide a "good enough" user interface. Improvements to the user interface are certainly possible (and desirable). Two such improvements have been implemented in commercial tools [15, 21] and presented at the annual Workshop on Refactoring Tools.

One area where even commercial tools tend to disappoint is in the presentation of error messages, i.e., what happens when a refactoring *cannot* be completed. The author's first differential precondition checking prototype gave notoriously cryptic error messages ("Unexpected derivative edge [35,38) ==(2)==> [38,53)") since these were the easiest to produce. The current version is somewhat more palatable; an example is shown in Figure 7.1. Nevertheless, more user-friendly presentations of errors are possible, and a user study to measure their effectiveness would be useful.

[1] The reader may recall from Chapter 1 that Photran is just under a quarter million total lines of code. The other 140,000 lines do not support refactoring; they support other IDE facilities, including the Fortran editor, new project wizards, search, etc.

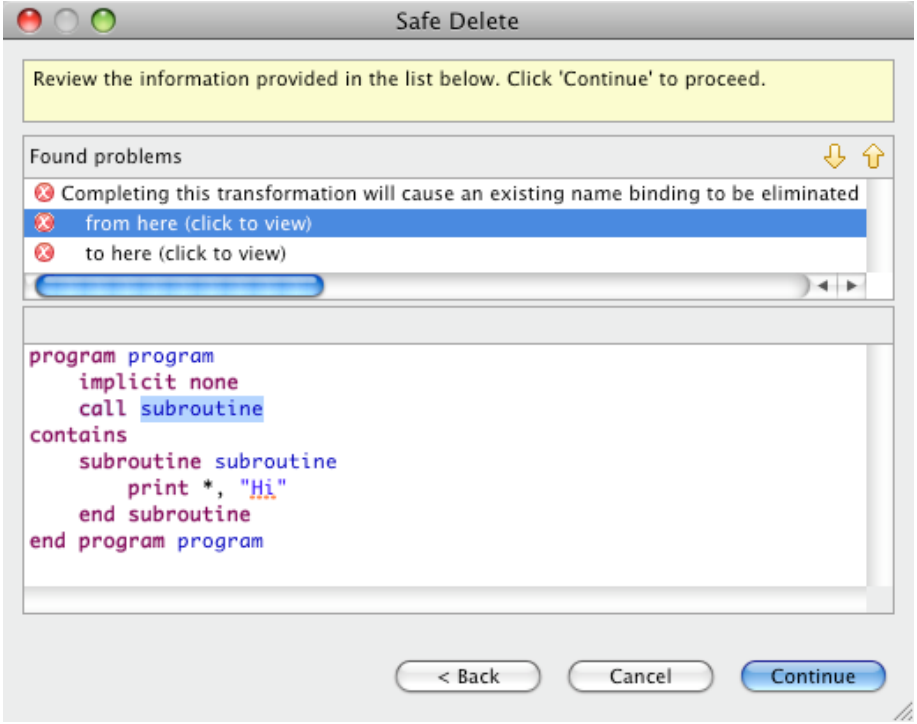


Figure 7.1: Error message produced by the differential refactoring prototype.

7.3.2 Pattern Matching and Declarative Rewriting

Ludwig's generated ASTs are designed to be traversed and manipulated imperatively. They could be extended with the ability to perform syntactic pattern matching and declarative rewriting. Many systems provide declarative rewriting facilities, including ASF+SDF [17] and Stratego/XT [50, 89]. A declarative pattern matching and rewriting facility has drawbacks (e.g., it can be difficult to learn and debug), and it was not necessary for the refactorings implemented thus far (including all 31 refactorings in Photran). Nevertheless, it could provide a very concise and powerful means to manipulate source code. Moreover, it would be provided by Ludwig as part of the generated code, allowing it to be written once and reused in many tools.

One way to integrate this into Ludwig would be to allow the user to indicate that certain productions (of the form $A ::= a$) in the grammar correspond to *pattern variables*. For example:

```
(superclass):<expr> ::= lhs:<expr> op:"+" rhs:<expr>      <= ASTBinaryExprNode
                      | lhs:<expr> op:"*" rhs:<expr>      <= ASTBinaryExprNode
                      | "(" <expr> ")"
                      | INTEGER                           <= ASTLiteralExprNode
                      | (pattern):PATTERN-VARIABLE

PATTERN-VARIABLE ::= "@" [A-Za-z]
```

Then, these pattern variables could be used to search and/or rewrite portions of the AST, perhaps as follows.

```
ast.rewriteExpr("2 * @A", "(@A + @A)");
```

7.3.3 Preprocessing

Another direction for future work is handling preprocessed code. As discussed in Chapter 1, the preprocessor poses challenges for refactoring tools, both syntactically and semantically.

The author has already proposed a mechanism for including preprocessor directives in Ludwig-generated ASTs [71,72]; this is partially implemented in Photran. Ultimately, it would be beneficial to have a standard interface for preprocessors, so almost arbitrary combinations of (pseudo-)preprocessors can be chained together and integrated into the Ludwig-generated infrastructure. (This is common practice in Fortran, for example.) This would be particularly beneficial since the same preprocessors could be reused with parsers for many different languages. What refactorings could be performed remains an open question, however.

One possibility for handling the semantic aspects of the preprocessor could be to add additional semantic edges to a program graph. For example, the relationship between a `#define` directive and subsequent use of a macro is analogous to a def-use relationship for a variable. It is an error for a refactoring to move a macro invocation above it the definition of that macro. If this def-use relationship for macros were modeled in a program graph, this type of error could be detected using differential precondition checking.

7.3.4 Other Topics

Beyond the topic of preprocessing, much more future work is possible. When differential precondition checking is used, how does it affect the amount of time taken to implement a refactoring? Do refactorings implemented with differential precondition checking tend to have more or fewer bugs than those implemented with traditional precondition checks? Both of these questions will require empirical data from many developers to answer conclusively. What other refactorings can be implemented using the preservation specifications described in the previous chapter? Are they sufficient to describe dependence-based loop transformations? Is it useful to extend a differential precondition checker with expensive interprocedural analyses for the purposes of testing but to replace these analyses with cheaper, traditional precondition checks in production? Can Ludwig-generated ASTs be constructed (and modified) using an incremental parser? What if preprocessor directives are included in the AST; can preprocessors also operate incrementally? These and other questions provide a number of directions for future research on differential precondition checking and generating rewritable ASTs.

Appendix A

Ludwig AST Node API

This appendix describes the application programming interface (API) implemented by Ludwig-generated abstract syntax trees. An overview of the API was given in Section 3.5.2.

The API described here does not include methods to create new AST nodes. Most often, AST nodes are constructed by the Ludwig-generated parser. This can be done for both entire programs as well as fragments of programs (e.g., expressions). Although it is possible to invoke a node class's constructor and setter methods, often it is easier (and more concise) to simply parse a string (e.g., `parser.parseExpression("3+4")`) and use the returned AST node. In other cases, it is preferable to copy an existing node (by invoking its `clone()` method) and modify the copy.

Recall that each generated AST node class contains getter and setter methods specific to that node class, in addition to the methods described here. The following sections describe the API that is common among all Ludwig-generated ASTs.

A.1 Common API (`IASTNode`)

Every node in the generated AST implements an interface called `IASTNode`, which contains the following methods. This means that all AST nodes—including generated node classes, the `Token` class, and list nodes (formed using the *(list)* annotation)—implement this common API. Generated AST node classes also add their own getter and setter methods.

Source Manipulation

- **`IASTNode clone().`** Returns a deep copy of this subtree. The returned tree is a copy of this node and all of its descendents. The parent of the root node is `null`, and the parents of all descendents are set to the appropriate nodes in the copied tree.
- **`void replaceChild(IASTNode node, IASTNode withNode).`** Replaces the given child node (subtree) with the given replacement. Throws an exception if `node` is not an (immediate) child of this node.
- **`void removeFromTree().`** Deletes this subtree from the AST. Throws an exception if this is the root node of the AST.
- **`void replaceWith(IASTNode newNode).`** Replaces this node (subtree) with the given replacement.
- **`void replaceWith(String literalString).`** Replaces this subtree with the given text.

Search and Traversal

- **`void accept(IASTVisitor visitor).`** Traverses this subtree using the given Visitor [39].
- **`IASTNode getParent().`** Returns the immediate parent of this node, or `null` if this is the root node of the AST.
- **`<T extends IASTNode> Iterable<T> findAll(Class<T> targetClass).`** Returns an `Iterable` which performs a preorder traversal of this tree, returning all nodes which are an instance of the given `targetClass`. The tree is traversed incrementally as the `Iterator#next` method is invoked; therefore, the behavior is undefined if the tree is modified during the traversal.
- **`<T extends IASTNode> List<T> collectAll(Class<T> targetClass).`** Performs a preorder traversal of this tree, constructing a list of all nodes which are an instance of the given `targetClass`, and returns this list. The nodes are listed in preorder. The tree is traversed completely before this method returns.
- **`<T extends IASTNode> T findNearestAncestor(Class<T> targetClass).`** Returns the nearest ancestor of this node which is an instance of the given class, or `null` if no such node exists.
- **`boolean isFirstChildInList().`** Returns true iff the immediate parent of this node is a list (i.e., an instance of `IASTListNode`) and this node is the first entry in the list.
- **`Iterable<? extends IASTNode> getChildren().`** Returns an `Iterable` which traverses the (immediate) children of this node from left to right. Every node returned by the `Iterable` is guaranteed to be non-`null`.
- **`<T extends IASTNode> T findFirst(Class<T> targetClass).`** Performs a preorder traversal of this subtree, returning the first descendent of this node which is an instance of the given class. Returns `null` if no such node exists.
- **`<T extends IASTNode> T findLast(Class<T> targetClass).`** Performs a preorder traversal of this subtree, returning the last descendent of this node which is an instance of the given class. Returns `null` if no such node exists.
- **`Iterable<Token> findAllTokens().`** Returns an `Iterable` which returns all of the tokens in this tree, in order. The tree is traversed incrementally as the `Iterator#next`
- **`Iterable<Token> findAllTokens(Terminal terminal).`** Returns an `Iterable` which returns, in order, all of the tokens in this tree which correspond to the given terminal symbol (i.e., tokens for which `Token.getTerminal() == terminal`. The tree is traversed incrementally as the `Iterator#next` method is invoked; therefore, the behavior is undefined if the tree is modified during the traversal.
- **`Token findFirstToken().`** Equivalent to `firstFirst(Token.class)`.
- **`Token findLastToken().`** Equivalent to `firstLast(Token.class)`.

Attribution

- **`<T> T getAttribute(String key).`** Returns the value associated with the given `key` for this node, or `null` if no value is associated. The `key` must not be `null`. Throws `ClassCastException` if the value is not an object of type `T`.

- **`<T> T getAttribute(String key, T defaultValue).`** Returns the value associated with the given `key` for this node, or `defaultValue` if no value is associated. The `key` must not be `null`. Throws `ClassCastException` if the value is not an object of type `T`.
- **`void setAttribute(String key, Object value).`** Associates the given value with the given `key` for this node. If a value is already associated with the given key (i.e., `this.getAttribute(key) != null`), the given value is stored instead of the previous value. The `key` must not be `null`.
- **`Map<String, Object> getAllAttributes().`** Returns a map of all of the key-value pairs associated with this node. The returned map may be empty but is never `null`.

Source Location Mapping

- **`int getOffset().`** Returns the source offset on which the first character of this node's source text occurs, excluding leading whitetext, or -1 if this node does not contain any tokens. In the former case, this is equivalent to `findFirstToken().getOffset()`.
- **`int getLength().`** Returns the length of this node's source text, excluding leading and trailing whitetext, or -1 if this node does not contain any tokens. In the former case, this is equivalent to `findLastToken().getOffset() + findLastToken().getLength() - findFirstToken().getOffset()`.
- **`int getOffsetIncludingWhitetext().`** Returns the source offset on which the first character of this node's source text occurs, including leading whitetext, or -1 if this node does not contain any tokens. In the former case, calling this method is equivalent to invoking `findFirstToken().getOffsetIncludingWhitetext()`.
- **`int getLengthIncludingWhitetext().`** Returns the length of this node's source text, including leading and trailing whitetext, or -1 if this node does not contain any tokens. In the former case, this is equivalent to `findLastToken().getOffsetIncludingWhitetext() + findLastToken().getLengthIncludingWhitetext() - findFirstToken().getOffsetIncludingWhitetext()`.

Source Code Reproduction

- **`void printOn(PrintStream out).`** Prints the source code corresponding to this subtree on the given `PrintStream`.
- **`String toString().`** Returns the source code corresponding to this subtree as a string.

A.2 List Node API (`IASTListNode`)

List nodes (i.e., AST nodes constructed for nonterminals with a *(list)* annotation) implement the `IASTListNode` interface. This interface extends both `IASTNode` and `java.util.List`. The latter provides methods to add, remove, and modify elements of the list, as well as to iterate through the list, test for membership, and so forth. Ludwig's `IASTListNode` provides two methods in addition to those provided by `IASTNode` and `java.util.List`:

- **`void insertBefore(T insertBefore, T newElement).`** Finds the first occurrence of the given element (`insertBefore`) in the list and inserts the given `newElement` immediately

prior to that element. Throws `IllegalArgumentException` if the element `insertBefore` is not found in the list.

- **`void insertAfter(T insertAfter, T newElement).`** Finds the first occurrence of the given element (`insertAfter`) in the list and inserts the given `newElement` immediately after that element. Throws `IllegalArgumentException` if the element `insertAfter` is not found in the list.

A.3 Token API (Token Class)

In addition to the above methods, the `Token` class implements the following methods. The four asterisked methods have a corresponding setter method, used to facilitate source manipulation.

- **`String getLeadingWhitetext()`*`.`** Returns this token's leading whitetext.
- **`String getTrailingWhitetext()`*`.`** Returns this token's trailing whitetext.
- **`String getText()`*`.`** Returns the text of this token, *excluding* leading and trailing whitetext.
- **`String toString().`** Returns the text of this token, *including* leading and trailing whitetext.
- **`Terminal getTerminal()`*`.`** Returns the terminal symbol in the grammar to which this token corresponds.
- **`int getLine().`** Returns the source line on which the first character of this token's text (as returned by `getText()`) occurs. The first line of the file is line 1.
- **`int getColumn().`** Returns the source column on which the first character of this token's text (as returned by `getText()`) occurs. The leftmost column is column 1.
- **`int getOffset().`** Returns the source offset on which the first character of this token's text (as returned by `getText()`) occurs. The first character is at offset 0.
- **`int getLength().`** Returns the length of this token's text (as returned by `getText()`). Equivalent to `getText().length()`.

Appendix B

Refactoring Specifications

B.1 Introduction[†]

This appendix contains detailed specifications of several automated refactorings for Fortran, BC, and PHP. The specifications are written somewhat like an ANSI or ISO programming language specification, mathematically informal but precise, in English prose but with sufficient detail to serve as a basis for implementation.

To the extent possible, the constructs in each language are described syntactically. For example, an External Subprogram in Fortran is defined to be a ‹*function-subprogram*› or a ‹*subroutine-subprogram*› nested under an ‹*external-subprogram*›. Such ‹*bracketed-names*› correspond to nonterminal symbols in a normative grammar for each programming language: the grammar in the ISO standard for Fortran 95 [48], the grammar in the POSIX specification for BC [47], and the Yacc grammar in the source code for the official distribution of PHP 5 [6]. The BC and PHP grammars use recursive productions to form lists of elements; in these cases, we will often ignore the recursive structure and, instead, refer to the list as a whole (e.g., “remove *X* and an appropriate adjacent comma from the list”), since implementations are likely to represent them as a list structure rather than a tree in abstract syntax anyway.

All algorithms are described imperatively, as a sequence of steps that may be executed to test the precondition or perform the transformation. It is not essential that an implementation execute these steps in the order listed; in many cases, the steps can be reordered and still produce the same results. For example, many precondition checks require a number of conditions to be checked, but these conditions are mutually disjoint, and therefore the order in which they are checked is inconsequential.

B.1.1 Terminology

This document contains four types of descriptions. **Predicates** return either TRUE or FALSE and are used in the specification of preconditions. **Preconditions** either PASS or FAIL and are used in the specification of refactorings. **Refactorings** consist of a list of preconditions and a program transformation. All of the preconditions must PASS if the program transformation is to be applied.

[†]This chapter is based on “A Collection of Refactoring Specification for Fortran 95, BC, and PHP 5” [70], co-authored with Matthew J. Fotzler, Ashley J. Kasza, and Ralph E. Johnson. The traditional versions of the Fortran refactoring specifications were published in ACM Fortran Forum [69].

Procedures describe algorithms used in the definition of a predicate, precondition, refactoring, or another procedure. Generally they will return a value.

The following conventions are used throughout.

in the immediate context of. A syntactic construct occurs *in the immediate context of* another if the former is an (immediate) child of the latter in a parse tree. For example, the Fortran 95 grammar contains the production

‹*program-stmt*› ::= PROGRAM ‹*program-name*›.

If a program contained the statement `program hello`, then `hello` would be a ‹*program-name*› which occurred in the immediate context of a ‹*program-stmt*›.

in the context of. A syntactic construct occurs *in the context of* another if the former is a descendent of the latter in a parse tree. It may be a child, grandchild, great-grandchild, etc.

contains. A syntactic construct *contains* another syntactic construct if the former is an ancestor of the latter in a parse tree. (Note that *A* contains *B* if, and only if, *B* occurs in the context of *A*—i.e., these terms are opposites.)

existing vs. new. When it is not clear from context, syntactic constructs will be qualified as either an *existing* (i.e., the construct exists in the program being analyzed/transformed) or *new* (i.e., the construct is constructed from scratch or supplied by the user). For example, the Rename refactoring takes two names as input: an existing name—this is the entity in the program that will be renamed—as well as a new name for that entity.

← When a refactoring must construct new syntax to be inserted into a program, the new construct is given in the concrete syntax of the language. Consider the following example.

Given a ‹subroutine-name› N, append to the ‹program›

‹*subroutine-subprogram*› ← `subroutine` *N* ↵
`end subroutine` ↵

(The symbol ↵ indicates an end-of-line.) The above statement means, "Construct a new ‹*subroutine-subprogram*› corresponding to the given concrete syntax (with the new subroutine name substituted for *N*), and append it to the ‹*program*›." (The meaning of "the ‹*program*›" would presumably be clear from context.) The left arrow is intended to denote that the new construct may be parsed from the given concrete syntax (although implementations may choose to construct the equivalent abstract syntax programmatically).

♦ In refactoring specifications, steps in which source code may be modified have been labeled with a black diamond.

◊ In some refactorings specifications, the precondition checks and the transformation traverse the program in similar ways. In these cases, it was simpler to intermix precondition checking steps with transformation steps. Precondition steps have been labeled with a white diamond.

B.1.2 Organization

The remainder of this appendix is organized as follows. One section is devoted to each language: Fortran, BC, and PHP. Each section begins with a list of definitions specific to that language. Defined terms are subsequently capitalized in order to make their usage more apparent. Following the list of definitions is a list of *requirements*—expectations that are made about the semantic analysis capabilities of the refactoring tool. For the most part, these are roughly equivalent to the capabilities of a (partial) compiler front end coupled with a cross-reference database.

The list of requirements is followed by a set of common predicates, preconditions, and procedures. These have been "factored out" of the refactoring specifications in order to keep the latter more concise and to avoid redundancy. These have each been given a two-letter abbreviation, enclosed in square brackets. Predicates and preconditions are abbreviated using two capital letters, e.g., [SI] or [LC]. Procedures are appreviated with a capital and lowercase letter, e.g., [Pr]. These abbreviations are used subsequently to indicate explicitly that a particular predicate, precondition, or procedure is being referenced.

Each part concludes with specifications of refactorings. Again, capitalization and abbreviations are used to indicate references to defined terms, predicates, preconditions, and procedures.

B.2 Fortran

B.2.1 Definitions

Body. The statements between the header statement and the end-statement of a construct. E.g., for a ‹*module*›, the Body consists of the statements between the ‹*module-stmt*› and ‹*end-module-stmt*›.

Declaration. An occurrence of a name that first introduces it into a Lexical Scope. Syntactically, one of the following:

1. ‹*type-name*› in the immediate context of a ‹*derived-type-stmt*›
2. ‹*component-name*› in the immediate context of a ‹*component-decl*›
3. ‹*object-name*› in the immediate context of an ‹*entity-decl*›
4. ‹*namelist-group-name*› in the immediate context of a ‹*namelist-stmt*›
5. ‹*common-block-name*› in the immediate context of a ‹*common-stmt*›
6. ‹*where-construct-name*› in the immediate context of a ‹*where-construct-stmt*›
7. ‹*forall-construct-name*› in the immediate context of a ‹*forall-construct-stmt*›
8. ‹*if-construct-name*› in the immediate context of a ‹*if-then-stmt*›
9. ‹*case-construct-name*› in the immediate context of a ‹*select-case-stmt*›
10. ‹*do-construct-name*› in the immediate context of a ‹*label-do-stmt*› or ‹*nonlabel-do-stmt*›
11. ‹*program-name*› in the immediate context of a ‹*program-stmt*›
12. ‹*module-name*› in the immediate context of a ‹*module-stmt*›
13. ‹*local-name*› in the immediate context of a ‹*rename*› or ‹*only-rename*›
14. ‹*block-data-name*› in the immediate context of a ‹*block-data-stmt*›
15. ‹*generic-name*›, ‹*defined-operator*›, or = in the immediate context of an ‹*interface-stmt*›

16. ‹*external-name*› in the immediate context of an ‹*external-stmt*›
17. ‹*intrinsic-procedure-name*› in the immediate context of an ‹*intrinsic-stmt*›
18. ‹*function-name*› in the immediate context of a ‹*function-stmt*›
19. ‹*subroutine-name*› in the immediate context of a ‹*subroutine-stmt*›
20. ‹*entry-name*› in the immediate context of an ‹*entry-stmt*›
21. ‹*function-name*› in the immediate context of a ‹*stmt-function-stmt*›
22. The first occurrence of a variable name which causes that variable to become implicitly declared.

Definition. A Declaration that is *not* any of the following:

1. ‹*external-name*› in the immediate context of an ‹*external-stmt*›
2. ‹*intrinsic-procedure-name*› in the immediate context of an ‹*intrinsic-stmt*›
3. ‹*function-name*› or ‹*subroutine-name*› in the immediate context of a ‹*function-stmt*› or ‹*subroutine-stmt*› in the immediate context of an ‹*interface-body*›

Except for COMMON blocks, every entity is assumed to have at most one Definition (assuming the Fortran program is valid).†

External Subprogram. A subprogram defined in File Scope; i.e., a ‹*function-subprogram*› or ‹*subroutine-subprogram*› in the immediate context of an ‹*external-subprogram*›. (See File Scope.)

File Scope. A ‹*program*›. (A File Scope is one kind of Lexical Scope; see Lexical Scope.‡)

Global Entity. A Program Unit or a ‹*common-block*›. (§14.1.1) (Note that a Global Entity may have multiple Declarations: An External Subprogram may also be declared in INTERFACE blocks and/or EXTERNAL statements, and a common block will usually be declared in several different COMMON statements in other scopes.)

Host. A program unit that may contain a CONTAINS statement and internal subprograms or module subprograms. Syntactically, one of ‹*main-program*›, ‹*module*›, ‹*function-subprogram*›, ‹*subroutine-subprogram*›, with the exception that neither a ‹*function-subprogram*› nor a ‹*subroutine-subprogram*› in the immediate context of an ‹*internal-subprogram*› can be a Host. [8, pp. 448, 544]

Import. If a Named Entity in a Scoping Unit S is use associated (§11.3.2) with a Named Entity N from a module M, we will say S *Imports* N from M.

Internal Subprogram. A subprogram following a CONTAINS statement in a Host. Equivalently, a ‹*function-subprogram*› or ‹*subroutine-subprogram*› in the immediate context of an ‹*internal-subprogram*›. [8, pp. 534–537]

Lexical Scope. A ‹*program*› or a Scoping Unit.‡

Local Entity (Class 1, 2, 3). Cf. §14.1.2. Refactorings herein deal exclusively with Class 1 Local Entities, which are "named variables that are not statement or construct entities (14.1.3), named constants, named constructs, statement functions, internal procedures,

module procedures, dummy procedures, intrinsic procedures, generic identifiers, derived types, and namelist group names."

Local Scope. A Scoping Unit. (§14.1.2) [8, p. 534]

Name. A ‹*name*›, or any syntactic construct named ‹*xyz-name*› (e.g., ‹*module-name*›).

Named Entity. A Name, the assignment symbol "=", or a ‹*defined-operator*›. (§14) [8, p. 532]

Outer Scope. A Lexical Scope that properly contains a given Lexical Scope in a parse tree; i.e., a Lexical Scope which is an ancestor of a given Lexical Scope).

Program Unit. One of: ‹*main-program*›, External Subprogram, ‹*module*›, or ‹*block-data*›. (§11; R202)

Reference. Any occurrence of a Name that is not a Definition.

Scoping Unit. One of the following: ‹*derived-type-def*›, ‹*main-program*›, ‹*module*›, ‹*block-data*›, ‹*function-subprogram*›, ‹*subroutine-subprogram*›. [8, p. 532]

Subprogram. One of: ‹*function-subprogram*› or ‹*subroutine-subprogram*›.

Subprogram Part. (The part of a Host that contains Internal Subprograms.) Either a ‹*module-subprogram-part*› or an ‹*internal-subprogram-part*›.

Subroutine. A ‹*subroutine-subprogram*›.

† Some entities may be declared in several locations. For example, an external subroutine may be defined in one file, while an INTERFACE block makes it available in another scope. In such cases, the declaration in the INTERFACE block is both a Declaration and a Reference, but it is *not* a Definition.

‡ Our concept of a Lexical Scope is different from the concept of "scope" in the Fortran standard [8, pp. 534–537]. Specifically, implied-DO variables, FORALL index variables, and statement-function parameters exist in a new scope according to the ISO specification, but for our (refactoring) purposes, we will treat them as references to a local variable in the enclosing scope. Also, the concept of File Scope is new.

B.2.2 Requirements

We will assume that the refactoring tool's capabilities are roughly those of a parser coupled with a syntax tree rewriter, name binding analysis (symbol tables), and cross-reference database. This means that the tool is able to construct and traverse a syntax tree, manipulate source code based on that syntax tree, find all Declarations of a Global Entity, find all Declarations in a given Lexical Scope (including implicit variables), find all References to a given Declaration, determine what type of entity a given name refers to (common block, local variable, function, etc.), determine an entity's attributes (PARAMETER, PUBLIC, etc.), find all Lexical Scopes which USE a particular module, and determine what entities are imported from that module.

B.2.3 Predicates, Preconditions, & Procedures

Predicate [LC]: Introducing N into S introduces a local conflict with N'

```
subroutine s
    integer :: n
    common /c/ n
contains
    !! subroutine n cannot be introduced here
    !! subroutine c can be introduced here
end subroutine
```

This determines whether two declarations cannot simultaneously exist in the same Lexical Scope.

Input. A new Named Entity N and an existing Named Entity N' with a Declaration in a Lexical Scope S. N and N' have the same name.

Procedure.

1. If N and N' both name Global Entities, return TRUE. (§14.1.1)
2. If N is the name of a common block and N' names a Local Entity, or vice versa, return FALSE. (§14.1.2)
3. If N is the name of an external procedure and N' is a generic name given to that procedure, return FALSE. (§14.1.2)
4. Otherwise, return TRUE. (§14.1.2)

Notes. This is a compilability check. A compiler uses these same rules when determining if a symbol can be added to the symbol table for a particular scope. If this predicate returns TRUE but N is introduced into S anyway, the program will not compile.

Predicate [SH]: Named Entity N in S cannot be shadowed in S'

```
subroutine s
    integer :: n
contains
    !! subroutine :: s cannot be introduced here
    subroutine t
        !! integer :: n can be introduced here
        !! integer :: s can be introduced here
        !! integer :: t cannot be introduced here
    end subroutine
end subroutine
```

If there is a Named Entity N defined in S, this check determines if another entity in a contained Lexical Scope S' cannot also be named N.

Input. A Named Entity N defined in a Lexical Scope S, and a Lexical Scope S' contained in S.

Procedure.

1. If S is the File Scope, return FALSE. (Entities defined at File Scope are Global Entities. They are accessible to, but not inherited by, contained scopes.)

2. If S' is a Scoping Unit and N is the name of S', return TRUE. (The name of a main program, module, or subprogram has limited use within its definition. – §11.1, 11.3, 14.1.2)
3. If S' is an Internal Subprogram, return FALSE. (Declarations in Internal Subprograms may shadow Declarations in their Hosts. – §14.6.1.3)
4. Otherwise, return TRUE.

Notes. This is a compilability check. A compiler uses these same rules when determining if a symbol can be added to the symbol table. If this predicate returns TRUE but N is introduced into S' anyway, the program will not compile.

Predicate [IC]: Introducing N into S introduces conflicts into an importing scope S'

```
module m1              module m2                                  subroutine s
   integer :: a           !! integer :: a cannot be introduced here    use m1; use m2
   integer :: b           !! integer :: b can be introduced here       print *, a
end module             end module                                 end subroutine
```

Suppose a new Named Entity N is to be introduced into a module, and another scope S' imports that module and will import N if it is introduced. This check determines whether S' already contains an entity with the same name as N.

Input. A new Named Entity N, a module S, and a Lexical Scope S' that (directly or indirectly) imports entities from the module S.

Procedure.
1. If there is an entity N' in scope in S' with the same name as N...
   (a) If N' is imported from a module but is unreferenced[†] in S', return FALSE. (§11.3.2) (This includes both the case where N' is imported without renaming and the case where N' is a ‹*local-name*› for a renamed module entity.)
   (b) If N' is inherited in S' from an Outer Scope, return TRUE iff N' cannot be shadowed by N in S [SH].
   (c) Otherwise, return TRUE iff introducing N introduces a local conflict [LC] with N'.
2. Otherwise, return FALSE.[‡]

Notes. This is a compilability check. If this predicate returns TRUE but N is introduced into S anyway, the program will not compile.

[†] There is some ambiguity as to what "unreferenced" means. The relevant clause of the ISO standard (§11.3.2) states: "Two or more accessible entities, other than generic interfaces, may have the same name only if the name is not used to refer to an entity in the scoping unit." The question is what "refer to" means. Specifically, (1) is `USE M, X => A, X => B` legal if the name X is never actually used, and (2) if M contains a module entity named X, should `USE M, X => A` be permitted (in which case the local name X would presumably shadow the module entity X)? IBM XL Fortran 12.1, GNU Fortran 4.4.2, PGI Fortran 10.0, and Intel Fortran 10.1 all exhibit different behaviors.

‡ There may be an entity with the same name in a contained scope, but it will be allowed to shadow the imported entity N; cf. [SH].

Predicate [SK]: Introducing N into S skews references in S'

```
module m
    integer n
contains
    subroutine s
        !! integer :: n cannot be introduced here
        call t
    contains
        subroutine t
            n = 1
        end subroutine
    end subroutine
end module
```

In the above code, a local variable named `n` *cannot be introduced into* `s` *because it would change the meaning of the reference to* `n` *in* `t`*, which could change the behavior of a program. This predicate detects situations such as this.*

Suppose a new Named Entity N is to be introduced into a scope but shadows an existing entity N'. This check determines whether any references to N' will instead become references to N if it is introduced.

Input. A new Named Entity N, a Lexical Scope S into which N is intended to be introduced, and a Lexical Scope S' which is either S itself or a Lexical Scope contained in S.

Procedure.

1. For each reference in S' to a Named Entity N' with the same name as N...
   (a) If N' is inherited from a scope S'' (where S' is contained in S''), return TRUE. (§14.6.1.3) (If N is introduced into S', N will shadow N', changing the reference.)
   (b) If N' is a reference to a procedure whose name has not been established (§14.1.2.4.3), return TRUE. (If N is introduced into S', the name will be established, changing the reference.)
2. For each Lexical Scope S'' contained in S', return TRUE if introducing N into S skews references [SK] in S''.
3. Otherwise, return FALSE.

Notes. This is both a compilability and a semantic preservation check. If bindings are skewed, say, from a variable to a subroutine, the program will not compile; if they are skewed, e.g., from one variable to another, behavior might not be preserved. In any case, if this predicate returns TRUE, name bindings will not be preserved if the transformation proceeds.

Precondition [IN]: Introducing N into S must be legal and name binding-preserving

This precondition makes two guarantees: (1) if a particular declaration is added to a program, the resulting program will compile (i.e., the addition of the declaration is legal); and (2) if the declaration will shadow another declaration, it will not inadvertently change references to the shadowed declaration.

Input. A new Named Entity N and a Lexical Scope S.

Procedure.

1. If there is a Named Entity N' in scope in S which has the same name as N...
   (a) If N' is local to S or is imported into S, FAIL if introducing N introduces a local conflict [LC] with N' in S.
   (b) If N' is declared in an Outer Scope, FAIL if N' cannot be shadowed [SH] by N in S.
   (c) FAIL if the introduction of N in S skews references [SK] in S.
2. For each Lexical Scope S' contained in S, if there is a Named Entity N' with the same name as N that is local to S' or is imported into S', FAIL if N cannot shadow [SH] N' in S'.
3. For each Lexical Scope S' that imports S, if S' will import N due to the absence of an ONLY clause...
   (a) FAIL if the introduction of N in S introduces conflicts [IC] into the importing scope S'.
   (b) FAIL if the introduction of N in S' skews references [SK] in S'.
4. PASS.

Notes. This precondition combines the previous four predicates into a single check which guarantees that, if N is introduced into S, then (1) the program will compile, and (2) name bindings will be preserved. The previous four predicates enumerate all of the conditions required for this guarantee to be made *a priori.* In a differential refactoring engine, this precondition can be eliminated entirely, since introducing N into S and testing for compilability and name binding preservation satisfies this precondition's checks: If name bindings will be skewed, predicate [SK] will fail. If the resulting program will not compile, one of predicates [LC], [SH], or [IC] will fail.

Precondition [SI]: Non-generic Internal Subprogram S must have only internal references

This precondition guarantees that there are no calls to a given Internal Subprogram except for directly recursive calls.

Input. An Internal Subprogram S in a Host H. S must not be a generic subprogram.

Procedure.

1. For each Reference R to S, FAIL if *neither* of the following hold:

(a) R occurs in the context of an ‹*access-stmt*› in the ‹*specification-part*› of H.

(b) R occurs in the Definition of S.

2. PASS.

Predicate [PR]: Private Entities in D are referenced outside D

Given a set D of module entities, this predicate determines whether any entities in D with PRIVATE *visibilities are referenced by definitions that are not in D.*

Input. A set D of Named Entity Definitions in a Module M.

Procedure.

1. For each Named Entity N in D...
   (a) If N has PRIVATE visibility, then...
      i. For each Reference R to N...
         A. If R does not occur in the Definition of an entity in D, return TRUE.
2. Return FALSE.

Notes. See Precondition [PP] and the refactoring Move Module Entities.

Procedure [Ou]: Determine Named Entities in $M - D$ referenced by D

Given a set D of module entities, this predicate determines whether any definitions in D reference entities in the module that are not included in D.

Input. A set D of Named Entity Definitions in a Module M.

Output. A set E of Named Entities in a Module M.

Procedure.

1. Initially, let $E \coloneqq \emptyset$.
2. For each Named Entity N in D...
   (a) For each Reference R in the Definition of N...
      i. If R names a public module entity from M that is not in the set D, define $E \coloneqq E \cup \{N\}$.
3. Return E.

Predicate [OU]: D references Named Entities in M outside D

Given a set D of module entities, this predicate determines whether any definitions in D reference entities in the module that are not included in D.

Input. A set D of Named Entity Definitions in a Module M.

Procedure.

1. Determine the set E of Named Entities in $M - D$ referenced by D [Ou].
2. Return TRUE iff $E \neq \emptyset$.

Notes. See Precondition [PP] and the refactoring Move Module Entities.

Precondition [PP]: D must partition private references in M

Given a set D of module entities, this precondition ensures that references to PRIVATE *entities occur such that either (1) both the entity and the reference are in D, or (2) neither the entity nor the reference is in D.*

Input. A set D of Named Entity Definitions in a Module M.

Procedure. Let $\overline{D}$ denote the set of all module entities declared in M that are not members of the set D.

1. FAIL if private entities in D are referenced outside D [PR].
2. FAIL if private entities in $\overline{D}$ are referenced outside $\overline{D}$ [PR].
3. PASS.

Procedure [Pr]: Construct a Set of Pairs from U Statement U

Given a USE *statement, this procedure returns a set of ordered pairs which model the module entities imported by that* USE *statement. The first component of each pair is the name of the module entity; the second component is its name in the local scope, which may be the same or different from the original name. For example, suppose a module* `MOD` *contains entities named* `a`, `b`, *and* `c`. *For the statement* `USE MOD`, *this procedure would return* $\{(a,a),(b,b),(c,c)\}$; *for the statement* `USE MOD, x => c`, *it would return* $\{(a,a),(b,b),(c,x)\}$; *and for the statement* `USE MOD, ONLY: a, x => b`, *it would return* $\{(a,a),(b,x)\}$.

Input. A ‹*use-stmt*› U.

Output. A set of ordered pairs of Names.

Procedure. Let N_M denote the set of names of all public entities in the module referenced by U.

1. If U contains neither a ‹*rename-list*› nor an ‹*only-list*›, return

$$\bigcup_{N \in N_M} (N, N).$$

2. If U contains a ‹*rename-list*›, return

$$\bigcup_{N \in N_M} \begin{cases} \{(N, N')\} & \text{if } N' \texttt{ => } N \text{ appears in the ‹}\textit{rename-list}\text{›, for some } N' \\ \{(N, N)\} & \text{if } N \text{ does not appear as a ‹}\textit{use-name}\text{› in the ‹}\textit{rename-list}\text{›} \end{cases}$$

3. If U contains an ‹*only-list*›, return

$$\bigcup_{N \in N_M} \begin{cases} \{(N, N')\} & \text{if } N' \texttt{ => } N \text{ appears in the ‹}\textit{only-list}\text{›, for some } N' \\ \{(N, N)\} & \text{if } N \text{ appears in the ‹}\textit{only-list}\text{›} \\ \varnothing & \text{if } N \text{ does not appear in the ‹}\textit{only-list}\text{›} \end{cases}$$

Procedure [Us]: Construct a U Statement for Module M from Sets of Pairs X and Y

This procedure is essentially the opposite of Procedure [Pr]: It takes as input a set of ordered pairs and uses them to construct a USE *statement. For example, for the module name* `mod` *and ordered pairs* $\{(a, a), (b, x)\}$*, it would return the statement* `USE MOD, ONLY: a, x => b`.

Input.

1. A Name M of a module.
2. A set X of ordered pairs of Names. (This set denotes the entities that the USE statement should import.)
3. A set Y of ordered pairs of Names of entities with public visibility in M. (This set denotes *all* of the public entities available to import from M. This set is provided as input to accommodate the Move Module Entities refactoring: it will need to construct a USE statement assuming that some entities have been moved out of one module and into another.)

Output. A new ‹*use-stmt*› U.

Procedure.

1. If $\{N \mid \exists N'.\ (N, N') \in X\} = \{L \mid \exists L'.\ (L, L') \in Y\}$, then every entity in M is imported.
   (a) If $X = Y$, then every entity in M is imported, and no entites are renamed, so return

   ‹*use-stmt*› ← `use` M ↵

   (b) If $\{N \mid \exists N' \neq N.\ (N, N') \in X\} \neq \emptyset$, then every entity in M is imported, but at least one entity is renamed. Let $(N_1, N_1'), (N_2, N_2'), \ldots, (N_k, N_k')$ denote the members of the set $\{(N, N') \in X \mid N \neq N'\}$, and return

   ‹*use-stmt*› ← `use` M`,` N_1' `=>` N_1`,` N_2' `=>` N_2`, ...,` N_k' `=>` N_k ↵
2. Otherwise, not all members of M are imported.
   (a) Initally, let U denote the ‹*use-stmt*›

   ‹*use-stmt*› ← `use` M`, only:` ↵

   which has an empty ‹*only-list*›.
   (b) For each pair (N, N') in X...
      i. If $N = N'$, append

      ‹*only-use-name*› ← N

      to the ‹*only-list*› of U (with a separating comma, if necessary).
      ii. If $N \neq N'$, append

      ‹*only-rename*› ← N' `=>` N

      to the ‹*only-list*› of U (with a separating comma, if necessary).
   (c) Return U.

Precondition [RN]: Module M' must not rename entities D from Module M

Given a set D of entities defined in a module M, this precondition ensures that, if any entities in D are directly imported into M′, they are not renamed.

Input. A set D of Named Entity Definitions in a Module M.

Procedure.
1. If M' contains a ‹*use-stmt*› U' with a ‹*module-name*› naming M...
   (a) Construct a set of pairs X from U' [Pr].
   (b) If X contains an element (N, N') where $N \in D$ and $N \neq N'$, FAIL.
2. PASS.

Procedure [Rn]: Replace References in C according to X

This procedure replaces occurrences of one name with a different name.

Input.
1. A set X of ordered pairs (N, N') where N is an existing Name and N' is a new Name.
2. Any syntactic construct C.

Output. C is modified such that References to N have their name changed to N'.

Procedure.
1. For each pair $(N, N') \in X$...
   (a) For each Reference R to N in C...
      i. Replace the occurrence of N in R with N'.

B.2.4 Refactorings

Add Empty Internal Subroutine

Requires: [LC],[SH],[IC],[SK],[IN]

This refactoring adds a new Subroutine as an Internal Subprogram of a given Host. The Subroutine initially has an empty body. The refactoring fails if the Subroutine will conflict with an existing declaration. Although this refactoring may be used by itself, but it is perhaps more useful as a building block for other refactorings (like Extract Subroutine).

Input.
1. A Host H into which the empty subroutine will be added as an internal subprogram.
2. A new Name N for the subroutine.

Preconditions. Introducing an Internal Subprogram into H with name N must be legal and name binding-preserving [IN].

Transformation.
1. ♦ If H does not contain a Subprogram Part, append to H

$$\text{Subprogram Part} \leftarrow \texttt{contains}\ \texttt{subroutine}\ N\ \texttt{end subroutine}$$

2. ♦ If H contains a Subprogram Part P, append to P

$$\langle \textit{internal-subprogram} \rangle \leftarrow \texttt{subroutine}\ N\ \texttt{end subroutine}$$

Notes. In a differential refactoring engine, precondition [IN] can be eliminated as described in its description. The new subroutine must not shadow an existing entity (i.e., skew references), which would be manifested as an incoming binding. It must not conflict with an existing entity, which would result in a compilation error. The addition of a new subroutine cannot introduce an outgoing name binding (although introducing a new function could, depending on its return type). Control flow and du-chains are intraprocedural and, therefore, are unaffected. The only new name binding edge will be an internal edge from the ‹*end-name*› to the ‹*subroutine-name*›. Therefore, the differential version of this refactoring consists of a single step: introducing the subroutine with rule $N^{\circlearrowleft}_{\subseteq}$.

Safe-Delete Non-Generic Internal Subprogram

Requires: [SI]

This refactoring removes an Internal Subprogram from a given Host. The refactoring fails if there are any references to the subprogram.

Input. An Internal Subprogram S in a Host H.

Preconditions. S must have only internal references [SI].

Transformation.

1. For each Reference to S in the context of an ‹*access-stmt*› A in the ‹*specification-part*› of H...
   (a) ♦ If the ‹*access-id-list*› of A contains only one ‹*access-id*› (i.e., a ‹*use-name*› with the name of S), remove A.
   (b) ♦ If there is more than one ‹*access-id*› in the ‹*access-id-list*› of A, remove the ‹*use-name*› with the name of S and an appropriate adjacent comma.
2. ♦ If H contains only one Internal Subprogram (S), remove the Subprogram Part of H.
3. ♦ If H contains more than one Internal Subprogram, remove S.

Notes.

This specification requires that the subprogram not be used in an ‹*interface-block*›. Extending the refactoring to remove this restriction is straightforward.

In a differential refactoring engine, the precondition [SI] can be eliminated. There must be no incoming bindings to the entity to delete; deleting a referenced subroutine would be manifested as a missing incoming binding. A subroutine may contain variable references and subroutine calls, so outgoing bindings may be deleted. Internal name binding edges, representing recursive calls and the link from the ‹*end-name*› to the ‹*subroutine-name*›, will also be deleted. Control flow and du-chains are intraprocedural and, therefore, are unaffected. Therefore, this refactoring consists of a single step: deleting the subroutine according to rule $N^{\supset\cup}_{\subseteq\subseteq}$.

Rename

Requires: [IN],[LC],[SH],[SK],[IC]

This refactoring changes the name of an entity, both in declarations and references. It fails if the new name will conflict with an existing name, or if it will shadow an existing name in such a way that existing name bindings will change.

Input.

1. A Declaration of a Name N in a Lexical Scope S. N must designate a Global Entity or Class 1 Local Entity.
2. A new Name N' for N.

Preconditions.

1. Introducing N' into S must be legal and name binding-preserving [IN].
2. WARN if a reference to N appears in the context of a ‹*namelist-group-object*›: To preserve behavior, the user may need to manually update input files to reflect the new variable name.
3. If N names a subprogram, matching declarations in INTERFACE blocks should uniquely bind to N.

Transformation.

1. For each Declaration D of the Named Entity N...

(a) ♦ Replace D with N'.

(b) For each Reference R to D...

i. ♦ Replace R with N'.

Notes. In a differential refactoring engine, precondition [IN] can be eliminated as described in its description. This refactoring should preserve the program graph in its entirety.

Introduce Implicit None

Requires: none

This refactoring adds an IMPLICIT NONE *into a Lexical Scope and all nested scopes and adds type declaration statements for all implicit variables. Its specification is greatly simplified by the infrastructural assumptions stated in Section B.2.2.*

Input. A Lexical Scope S.

Preconditions. (none)

Transformation. 1. If IMPLICIT NONE does *not* appear in the ‹*specification-part*› of S...

Let I' denote

‹*implicit-stmt*› ← `implicit none`↵

(a) ♦ If an ‹*implicit-stmt*› I appears in the ‹*specification-part*› of S, replace I with I'.

(b) ♦ If such an ‹*implicit-stmt*› does *not* appear, insert I' into the ‹*specification-part*› of S. (Note that the Fortran grammar requires that I' appear after all occurrences of ‹*use-stmt*› but before all occurrences of ‹*declaration-construct*›.)

(c) For each implicitly-typed variable N which is local to S...

Let T be a new ‹*type-spec*› corresponding to the type of N. (If the ‹*implicit-stmt*› I existed in Step 1a above, it is preferable to copy the concrete syntax of the ‹*type-spec*› from the existing ‹*implicit-stmt*›, when possible, in order to ensure that formatting and symbolic representations of kinds are reproduced verbatim.)

i. ♦ Insert the following into the ‹*specification-part*› of S:

‹*declaration-construct*› ← T :: N↵

2. Repeat Step 1 for each Lexical Scope S' contained in S.

Notes. This refactoring has no preconditions, since it is always legal to add explicit type declaration statements. If a scope is already IMPLICIT NONE, the transformation has no effect.

In a differential refactoring engine, this transformation will change name bindings such that they point to the variable declaration rather

than the first occurrence of the variable name (which implicitly declared the variable). Therefore, the affected forest must consist of both the first occurrence of the variable and the explicit declaration; then, the refactoring will introduce a new internal name binding edge (from the first use to the explicit declaration) but will otherwise preserve the program graph in its entirety. Therefore, this refactoring consists of a single step: introducing the explicit declaration according to rule $N^{\circlearrowleft}_{\supseteq}$.

Permute Subroutine Parameters Requires: none

This refactoring permutes the arguments to a subroutine, adjusting any call sites accordingly. Note that, if the actual arguments at a call site include function invocations with side effects, reordering these function calls may not preserve behavior.

Input.
1. A ‹*subroutine-subprogram*› S with n dummy arguments, $n \geq 2$, and
2. A permutation $\sigma = \begin{pmatrix} 1 & 2 & \dots & n \\ j_1 & j_2 & \dots & j_n \end{pmatrix}$ providing a new order for the arguments of S.

Preconditions.
1. Alternate return specifiers must retain the same relative order. That is, if the ‹*dummy-arg-list*› in S's ‹*subroutine-stmt*› has * for the ‹*dummy-arg*›s at indices $i_1, i_2, \dots, i_k$ where $i_1 < i_2 < \cdots < i_k$, then $\sigma(i_1) < \sigma(i_2) < \cdots < \sigma(i_k)$.
2. The permutation must not place an optional argument before an alternate return.
3. Matching declarations in INTERFACE blocks should uniquely bind to S.
4. (Checked during transformation)

Transformation.
1. ♦ Permute the ‹*dummy-arg*›s in the ‹*dummy-arg-list*› of S's ‹*subroutine-stmt*› according to σ.
2. For each ‹*call-stmt*› C which references S...

   The ‹*actual-arg-spec-list*› of C contains m ‹*actual-arg-spec*›s, for some $m \leq n$.

   (a) Initially, let $K \coloneqq$ FALSE.

   (b) Initially, let L' be an empty ‹*actual-arg-spec-list*›.

   (c) For $i \coloneqq \sigma(1), \sigma(2), \dots, \sigma(n)$:

   Let D denote the i-th dummy argument of S *before* its dummy arguments were permuted. If C contains an ‹*actual-arg-spec*› corresponding to D, denote it by A_i.

   i. If A_i is not defined, define $K \coloneqq$ TRUE. (An OPTIONAL argument was omitted, so all subsequent arguments must have keywords.)

ii. If A_i is defined...

Let A_i denote the ‹*actual-arg-spec*›.

A. If A_i contains ‹*keyword*› =, define K := TRUE.

B. If K = FALSE or A_i contains ‹*keyword*› =, append A_i (with a separating comma, if necessary) to L'.

C. ◇ FAIL if K = TRUE and A_i is an alternate return argument. (Permuting call sites must not place an alternate return argument after an argument with ‹*keyword*› =, since every subsequent actual argument must contain ‹*keyword*› =, but alternate return arguments cannot be given keywords.)

D. If K = TRUE and A_i does not contain ‹*keyword*› =, let N denote the ‹*dummy-arg-name*› of the i-th ‹*dummy-arg*› in S's ‹*subroutine-stmt*› before it was permuted, and append

‹*actual-arg-spec*› ← $N = A_i$.

to L'.

(d) ♦ Replace C's ‹*actual-arg-spec-list*› with L'.

3. For each ‹*subroutine-stmt*› S' in the context of an ‹*interface-block*› such that S' matches S...

(a) ♦ Permute the ‹*dummy-arg-list*› of S' according to σ.

Notes. None of this refactoring's preconditions are eliminated by using differential precondition checking. Precondition 1 ensures a kind of preservation that is not modeled by a program graph: Specifically, it ensures that the semantics of `return` statements are preserved in the context of alternate returns. Preconditions 2 and 4 perform input validation; the transformation cannot be performed unless they pass. Precondition 4 (placement of a keyword argument after an alternate return) could potentially be replaced with a compilability check, although this would require explicitly omitting ‹*keyword*› = from the alternate return argument even though it is known to be required—thus, it is reasonable to raise an explicit error rather than to generate code that is guaranteed to be erroneous.

Add Use of Named Entities E in Module M to Module M' [Prerequisite]

Requires: [IN],[LC],[IC],[SH],[SK]

This refactoring adds the statement `use M, only: E` *to the module M', if a similar statement does not already exist. It fails if this will result in a naming conflict, the introduction of circular dependencies between modules, or if a statement* USE *M already exists but renames entities in E.*

Input. 1. A Module M.

2. A set E of public Named Entities in M.
3. A distinct Module M'. The statement USE M will be inserted into M', if necessary.

Preconditions.

1. FAIL if M uses M'. (It would be necessary to introduce the statement USE M into M', but this would introduce a circular dependency.)
2. (Checked during transformation)

Transformation.

1. If M' contains a ‹*use-stmt*› U' with a ‹*module-name*› naming M, and U' contains an ‹*only-list*›...
   (a) For each Named Entity N in E that does not occur as a ‹*use-name*› in the context of U''s ‹*only-list*›...
      i. ◇ Ensure that introducing N into M' is legal and name binding-preserving [IN].
      ii. ♦ Append a separating comma and
      $$\text{‹}only\text{›} \quad \leftarrow \quad N$$
      to the ‹*only-list*› of U'.
2. If M' does not contain a ‹*use-stmt*› with a ‹*module-name*› naming M...
   Let $E_1, E_2, \ldots, E_k$ denote the elements of E.
   (a) For each Named Entity E_i, $1 \leq i \leq k$...
      i. ◇ Ensure that introducing E_i into M' is legal and name binding-preserving [IN].
   (b) ♦ Insert the statement
   $$\text{‹}use\text{-}stmt\text{›} \quad \leftarrow \quad \texttt{use}\ M, \texttt{only:}\ \ E_1,\ E_2,\ \ldots,\ E_k$$
   into the ‹*specification-part*› of M'.

Notes.

This refactoring fails precondition checking if a USE statement already exists but renames an entity in E: This is to simplify Move Module Entities, for which this refactoring is a prerequisite. Instead, Move Module Entities could rename references according to the new local names.

The first precondition can be eliminated in a differential refactoring engine since introducing a circular dependency will result in a compilation error. The checks for precondition [IN] can also be eliminated as described in its description. Adding the USE statements will introduce outgoing name bindings (to the imported entities), but the used names should be unreferenced; therefore, the USE statements should be inserted according to rule $N\overrightarrow{\subseteq}$.

Move Module Entities Requires: [OU],[Ou],[LC],[RN],[PP],[PR],[Pr],[Us],[Rn]

This refactoring moves a set of entities from one module to another, updating USE *statements as necessary. It fails if the changes will result in a naming conflict, a visibility problem, or the introduction of circular dependencies between modules.*

Allowing the user to move a set of entities often simplifies the refactoring process since it allows a PRIVATE *module variable and all of the procedures that use it to be moved at once. If they are moved one at a time, it becomes necessary to temporarily increase the visibility of the module variables in the interim.*

There are 21 different declaration constructs that can appear in a ‹module›. To keep this specification to a resonable length, we require the entities to move to be referenced only in ‹type-declaration-stmt›s, ‹access-stmt›s, and procedure definitions (see Precondition 1a). Extending it to support other constructs should be straightforward.

Input.

1. A set D of Named Entity Declarations in a Module M.
2. A distinct Module M' into which the entities will be moved.

Preconditions.

1. For each Named Entity N in D...
   (a) For each reference R to N which occurs in the context of M...
      i. If R does *not* occur in the context of any of the following, FAIL:
      - ‹*type-declaration-stmt*›
      - ‹*access-stmt*›
      - ‹*subroutine-subprogram*›
      - ‹*function-subprogram*›

   (b) For each Named Entity N' declared in M'...
      i. Introducing N into M' must not introduce a local conflict [LC] with N'.

   (c) Introducing N into M' must not skew references [SK] in M'.
2. M' must not rename entities D from M [RN].
3. D must partition private references in M [PP].

Transformation.

1. *(If any of the entities being moved from M use other entities in M, add* `use M` *to M′.)* If D references Named Entities in M outside D [OU]...

   Let E denote the set of Named Entities in M outside D that are referenced by D.

   (a) ♦ Add Use of Entities E in M to M' [Prerequisite].

   (b) Construct a Set U_E of Pairs from the USE Statement [Pr] created in the previous step. Let X denote the set $\{(N, N') \in U_E \mid N \neq N'\}$.

Let $\overline{D}$ denote the set of all module entities declared in M that are not members of D.

2. *(If any of the entities being moved from M are used by other entities in M that are not being moved, add* `use M'` *to M.)* If $\overline{D}$ references Named Entities in M outside $\overline{D}$ [OU]...

   Let E denote the set of Named Entities in M outside $\overline{D}$ that are referenced by D.

   (a) ♦ Add Use of Entities E in M' to M [Prerequisite].

3. *(If M' already contained a ‹use-stmt›, remove any of the references to the entities that are being moved, since they will no longer be in M.)* If M' contains a ‹*use-stmt*› U' with a ‹*module-name*› naming M...

   (a) If U' contains an ‹*only-list*›...

      i. ♦ If every ‹*only-use-name*› in the ‹*only-list*› is in D, remove the ‹*use-stmt*› U'.

      ii. ♦ Otherwise, remove from the ‹*only-list*› every ‹*only*› whose ‹*use-name*› is in D (also removing an appropriate adjacent comma).

4. *(Update* USE *statements.)* For each ‹*use-stmt*› U with a ‹*module-name*› naming M...

   Let S denote the Lexical Scope containing the ‹*use-stmt*›.

   If S contains a ‹*use-stmt*› whose ‹*module-name*› names M', let U' denote this ‹*use-stmt*›.

   (a) Construct a set U_M of pairs from U [Pr].

   (b) If U' does not exist, define $U_{M'} \coloneqq \varnothing$; otherwise, Construct a set $U_{M'}$ of pairs from U' [Pr].

   Let U_D denote the subset of U_M consisting of pairs whose first component names an entity in D. $U_D \coloneqq \{(Q, Q') \mid Q \in D \wedge (Q, Q') \in U_M\}$.

   Let P_M denote the set of pairs of public entities in M and P_D denote the subset of P_M consisting of pairs whose first component names an entity in D. $P_D \coloneqq \{(C, C') \mid C \in D\}$.

   (c) Construct a USE Statement K for Module M with $X \coloneqq U_M - U_D$ and $Y \coloneqq P_M - P_D$ [Us].

   (d) Construct a USE Statement K' for Module M' where $X \coloneqq U_{M'} \cup U_D$ and $Y \coloneqq P_M \cup P_D$ [Us].

   (e) i. ♦ If K does not have an empty ‹*only-list*›, replace U with K.

      ii. ♦ If K has an empty ‹*only-list*›, remove U.

   (f) i. ♦ If U' exists, then remove U'.

ii. ♦ If K' does not have an empty ‹*only-list*›, insert K' into S.

5. *(Move the declarations from M to M′.)* For each Named Entity N in D...

   (a) If N is a variable, and its Declaration is a ‹*type-declaration-stmt*› T...

      i. ♦ If X is defined (from Step 1b), replace references in T according to X [Rn].

      ii. ♦ If T's ‹*entity-decl-list*› contains only one ‹*entity-decl*›, (i.e., an ‹*entity-decl*› with the name of N), move T into the list of ‹*declaration-construct*›s in M'.

      iii. If T's ‹*entity-decl-list*› contains more than one ‹*entity-decl*›...

         Let E denote the ‹*entity-decl*› with the name of N in T's ‹*entity-decl-list*›.

         A. Create a copy T' of T.

         B. Replace T''s ‹*entity-decl-list*› with a list containing the single entry E.

         C. ♦ Remove E and an appropriate adjacent comma from T.

         D. ♦ Insert T' into the list of ‹*declaration-construct*›s in M'.

   (b) If N is a Subprogram whose Definition occurs in the context of a ‹*module-subprogram*› S...

      i. ♦ If X is defined (from Step 1b), replace references in S according to X [Rn].

      ii. ♦ If M' does not contain a ‹*module-subprogram-part*›, move S to construct the ‹*module-subprogram-part*› of M':

         ‹*module-subprogram-part*› ← `contains` ↷ S

      iii. ♦ If M' contains a ‹*module-subprogram-part*› P, move S into P.

   (c) For each Reference R to N...

      i. If R occurs in the context of an ‹*access-stmt*› A and A has not been moved into M' by the following step...

         A. ♦ If every ‹*access-id*› references a Named Entity in D, move A into the list of ‹*declaration-construct*›s in M'.

         B. ♦ Otherwise...

            Let S denote the ‹*access-spec*› of A.

            (1) Remove the ‹*use-name*› of R and an appropriate

adjacent comma.

(2) Insert a new ‹*access-stmt*›

$$\text{‹access-stmt›} \quad \leftarrow \quad S :: R_{\curvearrowright}$$

into the list of ‹*declaration-construct*›s in M'.

6. ♦ If, after completing Step 5, the ‹*module-subprogram-part*› of M is empty but M still contains a ‹*contains-stmt*›, remove the ‹*contains-stmt*› from M.

Notes. In a differential refactoring engine, precondition [PP] can be eliminated: When entities are moved from M to M', name bindings to PRIVATE entities in M will be eliminated (or skewed), resulting in a preservation failure. Precondition [RN] can also be eliminated, since the renamed entities will no longer exist, resulting in a compilation error and/or skewed bindings. Checks [LC] and [SK] can be eliminated from Step 1 as described in their descriptions. Step 1(a)(i) cannot be eliminated since it restricts the number of constructs on which the transformation can operate. The preservation analysis is only applied in Step 5 (after the USE statements have been updated): new incoming name binding edges (from the updated USE statements) will appear, but, otherwise, name bindings should be preserved. Therefore, this step proceeds according to rule $N^{\subseteq}_{\supseteq}$.

B.3 BC

B.3.1 Definitions

Array Declaration. A declaration of an array variable in a ‹*define_list*›: LETTER `[ ]`

Global Variable. A LETTER.

Name. A LETTER.

Scalar Declaration. A declaration of a scalar variable in a ‹*define_list*›: LETTER

Variable Declaration. An Array Declaration or a Scalar Declaration.

B.3.2 Predicates, Preconditions, & Procedures

Procedure [Ds]: Compute Dynamic Shadowing for Program P

Since BC is dynamically scoped, this procedure uses a simple, interprocedural data flow analysis to determine what local variables may be accessed by other functions.

Input. A ‹*program*› P.

Output. A function *uses*, which maps a ‹*function*› to a set of Variable Declarations.

Procedure. 1. *(Compute the call graph for P.)* Construct a directed graph whose node set consists of all ‹*function*›s in P and the whose edges are determined by the **calls** relation defined as follows:

(a) For each ‹*function*› F in P...

i. For each ‹*expression*› in the context of F which has the form $G(A)$ for some LETTER G and ‹*opt_argument_list*› A...

A. Define F **calls** G.

2. *(Compute the solution to the reaching definitions problem on the call graph.)*

Let X denote the set of all Variable Declarations in P.

(a) For each ‹*function*› F in P...

i. Define $gen(F)$ to be the set of all Variable Declarations in F. (Note that this is a subset of X.)

ii. Define $kill(F)$ to be the set of all Variable Declarations in X which have the same name as a Variable Declaration in F.

iii. Initially, let $reaches(F) \coloneqq \varnothing$.

(b) For each ‹*function*› G in P...

i. Let $reaches(G)$ be the least solution to the equation

$$reaches(G) = \bigcup_{F \in \mathbf{calls}^{-1}(G)} (gen(F) \cup (reaches(F) \cap \neg kill(F))) .$$

3. *(Compute du-chains on the call graph.)* Define a function *uses* as follows. For each ‹*function*› F in P...

(a) For each reference in F to a variable V...

i. If F does not contain a Variable Declaration for V...

A. every Variable Declaration in $reaches(F)$ with the name V is included in $uses(F)$.

B. every Global Variable with the name V is included in $uses(F)$.

4. Return the function *uses*.

Precondition [IN]: Introducing Variable Declaration V into Function F must be legal and name binding-preserving

This precondition makes two guarantees: (1) if a particular declaration is added to a program, the it will not introduce duplicate local variables, and (2) if the declaration will shadow another declaration, it will not inadvertently change references to the shadowed declaration.

Input. A new Variable Declaration V and a Function F.

Procedure. Compute dynamic shadowing for F [Ds] to obtain the function *uses*.

1. FAIL if the ‹*opt_auto_define_list*› in the context of F contains a declaration matching V.

2. Fail if *uses*(F) contains a Variable Declaration with the same name as V which does not occur in the context of F.
3. Pass.

Procedure [Cv]: Classify Local Variables in Statement Sequence S

This procedure is used by Extract Function to determine which local variables need to be passed as parameters to, and/or returned from, the extracted function.

Input.
1. A sequence $S := S_1, S_2, \ldots, S_n$ of consecutive ‹*statement*›s from a ‹*statement_list*› in the immediate context of a ‹*function*› F.

Output.
1. A set X of Variable Declarations.
2. A function *isParam* : $X \to \{\text{TRUE}, \text{FALSE}\}$.
3. A function *isReturn* : $X \to \{\text{TRUE}, \text{FALSE}\}$.

Procedure.
1. Initially, define $X := \varnothing$.
2. For each local variable V declared in F...
   (a) If V is referenced in S...
      i. Define $X := X \cup \{V\}$.
      ii. If there is a du-chain for V whose definition lies outside S and whose use lies inside S, define *isParam*(V) := TRUE. Otherwise, define *isParam*(V) := FALSE.
      iii. If there is a du-chain for V whose definition lies inside S and whose use lies outside S, define *isReturn*(V) := TRUE. Otherwise, define *isReturn*(V) := FALSE.
3. Return X, *isParam*, and *isReturn*.

B.3.3 Refactorings

Add Unreferenced Local Variable Declaration [Prerequisite]

Requires: [IN] [Ds]

This refactoring adds a declaration for an (unused) local variable.

Input. 1. A Variable Declaration V.

2. A Function F in which V will be declared as a local variable.

Preconditions. 1. Introducing V into F must be legal and name binding-preserving [IN].

Transformation. 1. If F contains an ‹*opt_auto_define_list*› L,

(a) ♦ If L is nonempty, append

list element ← , V

to the ‹*define_list*› of L.

(b) ♦ If L is empty, append

list element ← V

to the empty ‹*define_list*› of L.

2. ♦ If F does not contain an ‹*opt_auto_define_list*›, insert

‹*opt_auto_define_list*› ← `auto` V

to the ‹*define_list*› of L.

Notes. In a differential refactoring engine, Precondition [IN] can be eliminated, since, depending on the implementation, a conflict will either result in a compilability error or the introduction of additional (ambiguous) name binding edges, and shadowing will result in skewed name binding edges. The new variable should have no incoming name bindings. Therefore, this refactoring should preserve the program graph in its entirety.

Replace Statement with Block [Prerequisite]

Requires: (none)

This refactoring replaces a statement S with a block { S }.

Input. A ‹*statement*› S.

Preconditions. None.

Transformation. ♦ Replace S with

‹*statement*› ← { S }

Notes. This refactoring is always legal: If S is a ‹*statement*›, { S } is also a ‹*statement*›, according to the BC grammar. The BC specification does not contain any extra-grammatical restrictions on where particular statements may or may not occur.

Insert Assignment to Unreferenced Local Variable [Prerequisite] Requires: none

This refactoring inserts an assignment statement which assigns the value 0 to an otherwise unreferenced local variable.

Input.
1. A Scalar Variable V to be assigned.
2. A ‹*statement_list*› into which an assignment statement will be inserted, and the position at which it should be inserted.

Preconditions. There must be no references to V.

Transformation. ♦ Insert

‹*statement*› ← V = 0 ;

at the given position in the given ‹*statement_list*›.

Notes. The precondition for this refactoring is automatically satisfied when this refactoring is part of the Extract Function refactoring. Nevertheless, it is unnecessarily strong: The purpose of the precondition is to avoid introducing an assignment that would change the behavior of the program, i.e., to avoid introducing a new def-use edge. The assignment statement will have an outgoing name binding edge to the variable declaration, and control flow will not be preserved, but def-use edges should be preserved. Therefore, this refactoring proceeds according to the rule $N \overrightarrow{\supseteq} C \overleftarrow{\supseteq} \overrightarrow{\supseteq} \overleftarrow{\subseteq}^{\times} D$.

Move Expression Into Assignment [Prerequisite] Requires: none

This refactoring moves an expression from its original context into an assignment statement and then replaces the original expression with a use of the assigned variable.

Input.
1. An ‹*expression*› E occuring in the context of a ‹*function*›.
2. An assignment statement A of the form V = 0 for a Scalar Variable V.

Preconditions. Let S denote the least ‹*statement*› containing E.
1. S must exist in the immediate context of a ‹*statement_list*›. Let L denote this ‹*statement_list*›.
2. There must be no references to V except for the reference in A.
3. The assignment statement A must exist in the immediate context of L.
4. A must immediately precede S in L.
5. FAIL if E occurs in the context of the test or update expression in a for-loop.

Transformation.
1. ♦ In the assignment statement, replace the RHS expression 0 with E, removing E from its current context.

2. ♦ In E's original context, insert

$$\langle expression \rangle \leftarrow V$$

Notes. Observe that E is an ‹*expression*›—it cannot be a ‹*named-expr*› in a context like `e++` or `e += 5`. Preconditions 2 and 5 can be eliminated in a differential refactoring engine since they are effectively preserving def-use edges. The affected forest should include both of the replaced expressions (in the assignment statement and in the original context). Then, there will be one internal def-use edge introduced (from the new variable to the assignment statement), but otherwise no def-use edges should be introduced. Introducing the reference to the local variable will add one name binding. Therefore, this refactoring should proceed according to the rule $N^{\supseteq}_{\supseteq} D^{\circlearrowleft}_{\supseteq}$.

Extract Local Variable

Requires: (prerequisites)

Extract Local Variable removes an expression or subexpression from a statement, assigns it to a local variable, and replaces the original expression with a reference to that local variable.

Although the refactoring ensures that du-chains for local variables are preserved, it is the user's responsibility to ensure that the extracted expression is side effect-free or that the program will exhibit the correct behavior if it is not.

Input.
1. An ‹*expression*› E in a ‹*function*› F.
2. A new Name N for the local variable that will be created.

Preconditions.
1. E must have scalar type.
2. F must *not* declare or reference a scalar named N.

Transformation.
1. Add an Unreferenced Local Variable Declaration for N [Prerequisite].

   Let S denote the least ‹*statement*› in which E occurs.
2. ♦ If S is the ‹*statement*› providing the body of a for-statement, if-statement, or while-statement, Enclose S in a Block [Prerequisite], and, in the remaining steps, assume that S exists in this new context.

   (Note that, by construction, S must now exist in the immediate context of a ‹ statement_list ›.)
3. ♦ Insert an Assignment to the Unreferenced Local Variable N [Prerequisite] immediately before S.
4. ♦ Move E Into the Assignment statement inserted in the previous step [Prerequisite].

Notes. —

Add Empty Function

Requires: none

This refactoring adds a new ‹function› to a ‹program›. The ‹function› initially has an empty body. The refactoring fails if a ‹function› with the same name already exists.

Input.
1. A ‹*program*› P.
2. A new name (LETTER) N for the function.

Preconditions. FAIL if any ‹*input_item*› in P is a ‹*function*› whose name (LETTER) matches N.

Transformation. ♦ Append to P

‹*input_item*› ← `define` N`() {`↵
`}`↵

Notes. In a differential refactoring engine, the precondition can be eliminated: Depending on the implementation, introducing a function with the same name as an existing function will either result in a compilability error or the introduction of new def-use or name binding edges. Therefore, this refactoring should preserve the program graph in its entirety.

Populate Unreferenced Function

Requires: [Cv]

This refactoring copies statements from one function into another, replacing local variables with function arguments and returning the value of a variable if necessary. The refactoring fails if more than one value must be returned.

Input.
1. A sequence $S := S_1, S_2, \ldots, S_n$ of consecutive ‹*statement*›s from a ‹*statement_list*› in the immediate context of a ‹*function*›.

Preconditions.
1. There must not be a `return` statement in the context of S.
2. If there is a *break* statement in the context of S, then S must contain the least for-loop or while-loop containing the *break* statement.
3. (Checked during transformation)

Transformation.
1. Classify local variables in S [Cv] to obtain the set X and the functions *isParam* and *isReturn*.
2. ◊ FAIL if $|isReturn(X)| > 1$.
3. *(Construct an ‹opt_auto_define_list› A and an ‹opt_parameter_list› P.)*
   (a) Initially, let A and P be empty.
   (b) For each V in X...

i. If *isParam*(V) = TRUE, append V (and a comma, if necessary) to P.

ii. If *isParam*(V) = FALSE...

A. If A is empty, define A to be

‹*opt_auto_define_list*› ← `auto` V

B. Otherwise, append V (and a comma, if necessary) to A's ‹*define-list*›.

4. *(Construct a return statement R.)*

(a) Initially, let R be empty.

(b) If *isReturn*(X) = {V} for some V...

i. Define R to be

‹*statement*› ← `return` N`(`V`)` ↵

5. ♦ Replace F with

‹*function*› ← `define` N`(`P`) {` ↵
A ↵
S_1 ↵
S_2 ↵
...
S_n ↵
R ↵
`}` ↵

where A and R are omitted if they are empty.

Notes. In a differential refactoring engine, all preconditions can be eliminated, as long as this refactoring is being used only in the Extract Function composite: This is because a failure to meet these preconditions will cause Replace Statement Sequence to fail. If the statement sequence includes a *return* statement, this control flow will be lost when the statement sequence is replaced. Similarly, if more than one value needs to be returned, a def-use chain will be lost when the statement sequence is replaced. If the statement sequence includes a *break* statement but not the entire enclosing loop, this will raise a compilability error.

Replace Statement Sequence S

Requires: [Cv]

This refactoring replaces a sequence of statements with an equivalent function call. This refactoring is intended to be used only as part of Extract Function.

Input. 1. A sequence $S := S_1, S_2, \ldots, S_n$ of consecutive ‹*statement*›s from a ‹*statement_list*› in the immediate context of a ‹*function*›.

2. A new Name N.

Preconditions. (none)

Transformation.

1. Classify local variables in S [Cv] to obtain the set X and the functions *isParam* and *isReturn*.
2. *(Construct an ‹opt_parameter_list› P.)*
   (a) Initially, let P be empty.
   (b) For each V in X...
      i. If $isParam(V)$ = TRUE, append V (and a comma, if necessary) to P.
3. ♦ If $isReturn(X) = \{V\}$ for some V, replace S with

$$\text{‹statement\_list›} \quad \leftarrow \quad V = N(P)$$

4. ♦ Otherwise, replace S with

$$\text{‹statement\_list›} \quad \leftarrow \quad N(P)$$

Notes. In a differential refactoring engine, this replacement is expected to preserve incoming and outgoing control flow and du-chains if it is to preserve behavior. Clearly, the set of name bindings will change. Therefore, this refactoring proceeds according to the rule $C \overset{\cup}{\subseteq} D \overset{\cup}{\subseteq}$.

Extract Function

Requires: (prerequisites)

Extract Function creates a new method from a sequence of statements and replaces the original statements with a call to that method.

Input.

1. A sequence $S := S_1, S_2, \ldots, S_n$ of consecutive ‹*statement*›s from a ‹*statement_list*› in the immediate context of a ‹*function*›.
2. A new Name N.

Preconditions. (none)

Transformation.

1. ♦ Add an empty function named N [Prerequisite]. Call this function F.
2. ♦ Populate F according to S [Prerequisite].
3. ♦ Replace S with a call to F [Prerequisite].

Notes. —

B.4 PHP

B.4.1 Definitions

Class Declaration. An ‹*unticked-class-declaration-statement*›.

Class Name. The T_STRING in an ‹*unticked-class-declaration-statement*›.

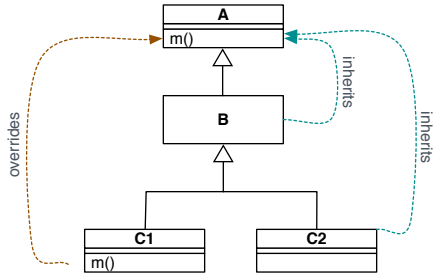
Method Declaration. A ‹*class-statement*› matching

‹*method-modifiers*› ‹*function*› ‹*is-reference*› T_STRING (‹*parameter-list*›) ‹*method-body*›

Method Name. The T_STRING in a Method Declaration.

B.4.2 Preconditions

Precondition [II]: Introducing M into Class C' must not introduce unexpected inheritance



*In a situation such as the one illustrated above, method m cannot be pulled up from C*1 *into B because this would cause C*2 *to inherit the pulled up method. This precondition prevents situations like this, where a class would inherit the "wrong" override of a method.*

Input. A Method Declaration M in a Class Declaration C with a direct superclass C'.

Procedure.

1. If M does not override a concrete superclass method, PASS.

   Otherwise, suppose M overrides M', which is defined in class P.

2. For each (direct or indirect) subclass D of P...
   (a) FAIL if all of the following hold:
      i. D inherits M' from P.
      ii. D is a (direct or indirect) subclass of C'.
      iii. $D \neq C$.
3. PASS.

B.4.3 Refactorings

Copy Up Method [Prerequisite]

Requires: [II]

Copy Up Method copies a method from one class into its immediate superclass.

Input. A Method Declaration M in the context of a Class Declaration C.

Preconditions.

1. There must be an ‹*extends-from*› node in the immediate context of C, and its ‹*fully-qualified-class-name*› must (uniquely) identify a Class Declaration C' in the same file as C.
2. M's ‹*method-modifiers*› must not contain T_ABSTRACT or T_STATIC.
3. C' must not contain a Method Declaration with the same name as M.
4. If there are any references to M that are not recursive references contained in M, then M must not have private visibility.
5. M must not contain any references to `self` or `__CLASS__`.
6. M must not contain any references to private members of C.
7. Moving M to C' must not introduce unexpected inheritance [II].
8. If M overrides a concrete method, and C' is not an abstract class, WARN the user that M will replace the overridden method in C', possibly changing the behavior of objects of that type.
9. If C' defines or inherits `__call`, and M does not override a superclass method, WARN the user: the program's behavior may change, since M will be invoked instead of `__call` for objects of type C'.

Transformation. ♦ Move the ‹*class-statement*› containing M from C's ‹*class-statement-list*› into C''s ‹*class-statement-list*›, replacing all references to `parent` with `self`.

Notes. We require the superclass to be in the same file as C in order to avoid dealing with `include` directives.

In a differential refactoring engine, precondition 3 will be caught by a compilability check. Preconditions 4–6 are simply preserving name bindings. A program that failed precondition 7 would introduce an incoming inheritance edge. If a program failed precondition 8, an outgoing inheritance edge from C' would vanish. Preconditions 1 and 2 cannot be eliminated because they perform input validation; precondition 9 checks for behavior that is not modeled by a program graph. This refactoring proceeds with preservation rule $NO^{\hookleftarrow}_{\supseteq}{}^{\cup}_{\supseteq}I^{\times}_{\subseteq}$.

Pull Up Method Requires: (prerequisites)

Pull Up Method moves a method from one class into its immediate superclass.

Input. A Method Declaration M in the context of a Class Declaration C.

Preconditions. None.

Transformation.

1. ♦ Copy Up M [Prerequisite].
2. ♦ Delete the ‹*class-statement*› containing M from the Class Declaration C's ‹*class-statement-list*›

Notes. All of the preconditions for this refactoring are handled by Copy Up Method. The delete operation proceeds with preservation rule $NO^{\cup}_{\subseteq}I^{\subseteq}_{\supseteq}$.

References

[1] ANTLR. http://www.antlr.org/.

[2] Eclipse C/C++ development tooling. http://www.eclipse.org/cdt.

[3] Eclipse Java development tools. http://www.eclipse.org/jdt.

[4] LALR parser generator. http://sourceforge.net/projects/lpg/.

[5] Ludwig. http://ludwig.jeff.over.bz.

[6] PHP: Hypertext preprocessor. http://www.php.net/.

[7] Usage data collector: Reports: Commands. http://www.eclipse.org/org/usagedata/reports/data/commands.csv.

[8] J.C. Adams, W.S. Brainerd, J.T. Martin, B.T. Smith, and J.L. Wagener. Fortran 95 Handbook: Complete ISO/ANSI Reference. MIT Press, Cambridge, MA, 1997.

[9] F. E. Allen. A technological review of the FORTRAN I compiler. In AFIPS '82: Proceedings of the June 7-10, 1982, National Computer Conference, pages 805–809, New York, NY, USA, 1982. ACM.

[10] J. R. Allen, D. Baumgartner, K. Kennedy, and A. Porterfield. PTOOL: A semi-automatic parallel programming assistant. In Proceedings of the 1986 International Conference on Parallel Processing. IEEE Computer Society Press, August 1986.

[11] American National Standards Institute. ANSI INCITS 319-1998 (R2007): Information Technology – Programming Languages – Smalltalk. 2007.

[12] Edward A. Ashcroft and Zohar Manna. The translation of 'go to' programs to 'while' programs. In World Computer Congress – IFIP, pages 250–255, 1971.

[13] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Subhlok. The ParaScope editor: An interactive parallel programming tool. In Supercomputing '89: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, pages 540–550, New York, NY, USA, 1989. ACM.

[14] Jan Bečička, Petr Hřebejk, and Petr Zajac. Using Java 6 compiler as a refactoring and an analysis engine. In Proceedings of the 1st Workshop on Refactoring Tools (WRT'07), pages 57–58. Technical Report 2007-08, Technische Universität Berlin, 2007.

[15] Robert Bowdidge. Performance trade-offs implementing refactoring support for Objective-C. In WRT '09: Proceedings of the 3rd Workshop on Refactoring Tools.

[16] Robert Bowdidge. Personal communication, 2007.

[17] M. Van Den Brand, A. Van Deursen, J. Heering, H. De Jong, M. De Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: A component-based language development environment. In Compiler Construction 2001, volume 2027 of Lecture Notes in Computer Science, pages 365–370, 2001.

[18] John Brant. Personal communication, 2009.

[19] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-Oriented Software Architecture: A System of Patterns, volume 1. John Wiley & Sons, 1996.

[20] C. R. Calidonna, M. Giordano, and M. Mango Furnari. A graphic parallelizing environment for user-compiler interaction. In ICS '99: Proceedings of the 13th International Conference on Supercomputing, pages 238–245, New York, NY, USA, 1999. ACM.

[21] Dustin Campbell and Mark Miller. Designing refactoring tools for developers. In WRT '08: Proceedings of the 2nd Workshop on Refactoring Tools, pages 9:1–9:2, New York, NY, USA. ACM.

[22] Alan Carle, Keith D. Cooper, Robert T. Hood, Ken Kennedy, and Linda Torczon. A practical environment for scientific programming. IEEE Computer, 20(11):75–89, 1987.

[23] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. IEEE Software, 7(1):13–17, 1990.

[24] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. Proceedings of the IEEE, 81(2):244–263, February 1993.

[25] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization on the design of a software development environment. In Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, pages 107–116, New York, NY, USA, 1985. ACM.

[26] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment. ACM Transactions on Programming Languages and Systems, 8(4):491–523, 1986.

[27] Thomas Corbat, Lukas Felber, Mirko Stocker, and Peter Sommerlad. Ruby refactoring plug-in for Eclipse. In OOPSLA '07: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 779–780, New York, NY, USA, 2007. ACM.

[28] J. Cordy. The TXL source transformation language. Science of Computer Programming, 61:190–210, 2006.

[29] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to Algorithms. MIT Press, second edition, 2001.

[30] S. Counsell, Y. Hassoun, G. Loizou, and R. Najjar. Common refactorings, a dependency graph and some code smells: An empirical study of Java OSS. In ISESE '06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, pages 288–296, New York, NY, USA, 2006. ACM.

[31] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 185–194, New York, NY, USA, 2007. ACM.

[32] Danny Dig and Ralph Johnson. How do APIs evolve? A story of refactoring. Journal of Software Maintenance and Evolution: Research and Practice, 18(2):83–107, 2006.

[33] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In ICSE '09, pages 397–407, 2009.

[34] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. Technical Report RC-10208, IBM Research, August 1983.

[35] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems, 9(3):319–349, 1987.

[36] Martin Fowler. Crossing refactoring's rubicon. `http://www.martinfowler.com/articles/refactoringRubicon.html`.

[37] Martin Fowler. Refactoring: Improving the design of existing code. Addison Wesley, Boston, MA, USA, 1999.

[38] Martin Fowler. MF Bliki: RefactoringMalapropism. `http://martinfowler.com/bliki/RefactoringMalapropism.html`, January 2004.

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Massachusetts, January 1995.

[40] Alejandra Garrido. Program refactoring in the presence of preprocessor directives. PhD thesis, Champaign, IL, USA, 2005.

[41] Emanuel Graf, Guido Zgraggen, and Peter Sommerlad. Refactoring support for the C++ development tooling. In OOPSLA '07: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 781–782, New York, NY, USA, 2007. ACM.

[42] William G. Griswold. Program Restructuring as an Aid to Software Maintenance. PhD thesis, University of Washington, 1991.

[43] V. A. Guarna, D. Gannon, D. Jablonowski, A. D. Malony, and Y. Gaur. Faust: An integrated environment for parallel programming. IEEE Software, 6(4):20–27, July 1989.

[44] Mary W. Hall, Timothy J. Harvey, Ken Kennedy, Nathaniel McIntosh, Kathryn S. McKinley, Jeffrey D. Oldham, Michael H. Paleczny, and Gerald Roth. Experiences using the ParaScope Editor: An interactive parallel programming tool. In PPOPP '93: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 33–43, New York, NY, USA, 1993. ACM.

[45] Seema Hiranandani, Ken Kennedy, Chau-Wen Tseng, and Scott Warren. The D editor: A new interactive parallel programming tool. In Supercomputing '94: Proceedings of the 1994 Conference on Supercomputing, pages 733–742, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[46] Institut für software. `http://ifs.hsr.ch/`.

[47] Institute of Electrical and Electronics Engineers. IEEE Std 1003.1-2008: IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 7. January 2008.

[48] International Organization for Standardization and International Electrotechnical Commission. ISO/IEC 1539-1:1997: International Standard: Information Technology, Programming Languages, Fortran. Second edition, 1997.

[49] Stephen C. Johnson. Yacc – Yet another compiler-compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[50] M. De Jonge, E. Visser, and J. Visser. XT: A bundle of program transformation tools. In Proc. Lang. Descriptions, Tools and Applications 2001, volume 44 of Elec. Notes in Theoretical Comp. Sci., pages 211–218, 2001.

[51] K. Kennedy, K.S. McKinley, and C.W. Tseng. Interactive parallel programming using the ParaScope Editor. IEEE Transactions on Parallel and Distributed Systems, 2(3):329–341, 1991.

[52] Ken Kennedy and John R. Allen. Optimizing compilers for modern architectures: A dependence-based approach. Morgan Kaufmann, San Francisco, CA, USA, 2002.

[53] Ken Kennedy, Kathryn S. McKinley, and Chau-Wen Tseng. Analysis and transformation in the ParaScope editor. In ICS '91: Proceedings of the 5th International Conference on Supercomputing, pages 433–447, New York, NY, USA, 1991. ACM.

[54] Adam Kiezun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. Refactoring for parameterizing Java classes. In ICSE '07: Proceedings of the 29th International Conference on Software Engineering, pages 437–446, Washington, DC, USA, 2007. IEEE Computer Society.

[55] Michael Klenk, Reto Kleeb, Martin Kempf, and Peter Sommerlad. Refactoring support for the Groovy-Eclipse plug-in. In OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 727–728, New York, NY, USA, 2008. ACM.

[56] Raghavan Komondoor and Susan Horwitz. Semantics-preserving procedure extraction. In Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 155–169, New York, NY, USA, 2000. ACM.

[57] Arun Lakhotia and Jean-Christophe Deprez. Restructuring programs by tucking statements into functions. Information and Software Technology, 40(11–12):677–689, November 1998.

[58] Shih-Wei Liao, Amer Diwan, Robert P. Bosch, Jr., Anwar Ghuloum, and Monica S. Lam. SUIF Explorer: An interactive and interprocedural parallelizer. In PPoPP '99: Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 37–48, New York, NY, USA, 1999. ACM.

[59] Yuichi Matsuo. On emerging trends and future challenges in aerospace CFD using the CeNSS system of JAXA NS-III. International Conference on High Performance Computing and Grid in Asia Pacific Region, 0:388–395, 2004.

[60] Steve McConnell. Code Complete. Microsoft Press, Redmond, WA, USA, second edition, 2004.

[61] Robert J. McEliece, Robert B. Ash, and Carol Ash. Introduction to Discrete Mathematics. Random House, New York, NY, 1989.

[62] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. Journal of Software Maintainence and Evolution, 17(4):247–276, 2005.

[63] John David Morgenthaler. Static Analysis for a Software Transformation Tool. PhD thesis, University of California, San Diego, 1997.

[64] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Eclipse IDE? IEEE Software, 23(4):76–83, 2006.

[65] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[66] 永井, 亨. XPFortran 入門. 情報連携基盤センターニュース, 5(2):129-168, 2006.

[67] William F. Opdyke. Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1992.

[68] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA). ACM, September 1990.

[69] Jeffrey L. Overbey, Matthew J. Fotzler, Ashley J. Kasza, and Ralph E. Johnson. A collection of refactoring specifications for Fortran 95. SIGPLAN Fortran Forum, 29:11–25, November 2010.

[70] Jeffrey L. Overbey, Matthew J. Fotzler, Ashley J. Kasza, and Ralph E. Johnson. A collection of refactoring specifications for Fortran 95, BC, and PHP 5. Technical Report `http://jeff.over.bz/papers/2011/tr-refacs.pdf`, 2011.

[71] Jeffrey L. Overbey and Ralph E. Johnson. Generating rewritable abstract syntax trees. In Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers, volume 5452 of Lecture Notes in Computer Science, pages 114–133, Berlin, Heidelberg, 2009. Springer-Verlag.

[72] Jeffrey L. Overbey, Matthew D. Michelotti, and Ralph E. Johnson. Toward a language-agnostic, syntactic representation for preprocessed code. In WRT '09: Proceedings of the 3rd Workshop on Refactoring Tools.

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

[74] Terence Parr. The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Programmers. Pragmatic Bookshelf, May 2007.

[75] Photran – an integrated development environment for Fortran. `http://www.eclipse.org/photran/`.

[76] Chris Recoskie. Handling conditional compilation in CDT's core. `http://wiki.eclipse.org/images/b/b3/Handling_Conditional_Compilation_In_CDT's_Core.pdf`, 2007.

[77] Refactoring benchmarks for Extract Method. `http://c2.com/cgi/wiki/wiki?RefactoringBenchmarksForExtractMethod`.

[78] Refactoring benchmarks for Pull Up Method. `http://c2.com/cgi/wiki/wiki?RefactoringBenchmarksForPullUpMethod`.

[79] Christoph Reichenbach, Devin Coughlin, and Amer Diwan. Program metamorphosis. In Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP 2009), pages 394–418, Berlin, Heidelberg, 2009. Springer-Verlag.

[80] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. Theory and Practice of Object Systems, 3(4):253–263, 1997.

[81] Donald Bradley Roberts. Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[82] Max Schäfer and Oege de Moor. Specifying and implementing refactorings. In Martin Rinard, editor, Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH). ACM, 2010.

[83] Max Schäfer, Julian Dolby, Manu Sridharan, Frank Tip, and Emina Torlak. Correct refactoring of concurrent Java code. In Theo D'Hondt, editor, European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, 2010.

[84] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for Java. In OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 277–294, New York, NY, USA, 2008. ACM.

[85] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. Stepping stones over the refactoring rubicon – lightweight language extensions to easily realise refactorings. In Sophia Drossopoulou, editor, European Conference on Object-Oriented Programming (ECOOP), pages 369–393. Springer-Verlag, 2009.

[86] Mathieu Verbaere. A Language to Script Refactoring Transformations. PhD thesis, University of Oxford, 2008.

[87] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. JunGL: A scripting language for refactoring. In Dieter Rombach and Mary Lou Soffa, editors, ICSE'06: Proceedings of the 28th International Conference on Software Engineering, pages 172–181, New York, NY, USA, 2006. ACM.

[88] Mathieu Verbaere, Arnaud Payement, and Oege de Moor. Scripting refactorings with JunGL. In OOPSLA '06: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming, Systems, Languages, and Applications, pages 651–652, New York, NY, USA, 2006. ACM.

[89] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In Rewriting Techniques and Applications, volume 2051 of Lecture Notes in Computer Science, pages 357–361. Springer-Verlag, 2001.

[90] John Vlissides. Pattern Hatching: Design Patterns Applied. Addison Wesley, 1998.

[91] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In DSL'97: Proceedings of the Conference on Domain-Specific Languages (DSL), 1997, pages 17–17, Berkeley, CA, USA, 1997. USENIX Association.

[92] Tom Way and Lori Pollock. Evaluation of a region-based partial inlining algorithm for an ILP optimizing compiler. In Proceedings of the Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA2002), pages 552–556, 2002.

[93] Xrefactory. `http://www.xref.sk/xrefactory/main.html`.

[94] Peng Zhao and Jose Nelson Amaral. Function outlining and partial inlining. In Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing, pages 101–108, Washington, DC, USA, 2005. IEEE Computer Society.