Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators

Rijnard van Tonder
School of Computer Science
Carnegie Mellon University
USA
rvt@cs.cmu.edu

Claire Le Goues
School of Computer Science
Carnegie Mellon University
USA
clegoues@cs.cmu.edu

Abstract

Automatically transforming programs is hard, yet critical for automated program refactoring, rewriting, and repair. Multi-language syntax transformation is especially hard due to heterogeneous representations in syntax, parse trees, and abstract syntax trees (ASTs). Our insight is that the problem can be decomposed such that (1) a common grammar expresses the central context-free language (CFL) properties shared by many contemporary languages and (2) open extension points in the grammar allow customizing syntax (e.g., for balanced delimiters) and hooks in smaller parsers to handle language-specific syntax (e.g., for comments). Our key contribution operationalizes this decomposition using a Parser Parser combinator (PPC), a mechanism that generates parsers for matching syntactic fragments in source code by parsing declarative user-supplied templates. This allows our approach to detach from translating input programs to any particular abstract syntax tree representation, and lifts syntax rewriting to a modularly-defined parsing problem. A notable effect is that we skirt the complexity and burden of defining additional translation layers between concrete user input templates and an underlying abstract syntax representation. We demonstrate that these ideas admit efficient and declarative rewrite templates across 12 languages, and validate effectiveness of our approach by producing correct and desirable lightweight transformations on popular real-world projects (over 50 syntactic changes produced by our approach have been merged into 40+). Our declarative rewrite patterns require an order of magnitude less code compared to analog implementations in existing, language-specific tools.

CCS Concepts • **Software and its engineering → Syntax**; **Translator writing systems and compiler generators**; **Parsers**; *General programming languages*; Domain specific languages.

Keywords syntax, transformation, parsers, rewriting

ACM Reference Format:
Rijnard van Tonder and Claire Le Goues. 2019. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3314221.3314589

1 Introduction

Automatically transforming programs is hard, yet critical for automated program refactoring [1, 2, 45], rewriting [8, 44], and repair [37, 43, 52, 54]. The complexity of automatically transforming code has yielded a plethora of approaches and tools that strike different design points in expressivity, language-specificity, interfaces, and sophistication [1, 3, 4, 32, 44]. At one end of the spectrum, techniques based on regular expressions, like `sed` or `codemod` [3], perform simple textual substitutions. On the other, language-specific AST visitor frameworks, like `clang-reformat`, manipulate rich program structures and metadata. Unfortunately, many lightweight transformations that should apply to multiple languages are entangled in language-specific tools with complex interfaces, relying on a program's AST to work. For example, consider a simple list slicing "quick fix": both Python and Go can be simplified from α`[`β`:len(`α`)]` to α`[`β`:]` for a list assigned to variable α and sliced from initial index β. Visually, the transformation is lightweight and easy to describe. Unfortunately, *actually performing* this type of transformation generally boils down to separate implementations depending on language-specific ASTs.

We observe that a wide range of lightweight transformations rely on matching tokens structurally within well-formed delimiters (essential syntax for context free languages, and common to virtually all contemporary languages, [2, 4, 6, 10–12]). Our approach exploits this observation to enable short (yet nontrivial) rewrite specifications that elide the need for complex, language-specific implementations, syntax definitions, or translations from metasyntax to an

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314589

underlying abstract representation. We introduce a modular grammar that can be modified based on syntactic differences in languages (e.g., comment syntax), but which preserves the quintessential property of context-free languages. Our grammar extends Dyck languages [16], which capture the CFL constructs of interest in conventional programming languages. We develop a Dyck-Extended Language (DEL) grammar that can interpolate tokens (i.e., regular strings) between and inside balanced delimiters as a base representation for syntactic manipulation.

The crux of our approach implements a parser generator for a DEL; the generated parser matches concrete syntax of interest, enabling rewriting. Importantly, the parser generator is modular (reflecting the fact that DEL grammars are modularly defined), which we implement using parser combinators (we refer to our modular parser generators as "Parser Parser Combinators", or PPCs). Thus, the parser generator contains open extension points where smaller parsers handle language specific syntax (e.g., custom delimiters and comment syntax) in the generated match parser it produces.

The first advantage of using PPCs is that our approach does not rely on any source code translation at all. It instead detaches from representing input programs with any particular abstract syntax tree representation: the original source input *is* our concrete representation, and matching syntax relies only on how a parser interprets that input. Traditional approaches rely on first parsing syntax into a tree representation which is then manipulated. By contrast, our approach embeds the structural properties of syntax in the generated parsers alone; the parser records only matching syntax during parsing. PPCs enable easy multi-language extension for varying syntax across languages by exposing hooks for smaller parsers in a skeleton parser. Extending the parser to language-specific syntax (e.g., for comments) in the parsing routine is comparatively easy in our architecture as compared to modifying concrete AST definitions in code.

The second advantage of PPCs is that they interpret user-specified match templates *as* parser generators for a DEL. Small, custom syntactic parsers embed into the PPC to parameterize both how user-supplied templates are interpreted, *and* the parsers that those templates produce. The effect is that we avoid the complexity of implementing an additional translation layer from user-supplied metasyntax to an abstract syntax representation; the same custom syntax definitions embed seamlessly into the parser for user-supplied templates and the match procedure (a parser) produced from it. The result is rewrite templates that are syntactically close to the target patterns with minimal metasyntax. The main contributions of our work are:

- The Dyck-Extended Language representation, defined by a modular grammar for accommodating syntactic idiosyncrasies and ambiguity across multiple languages. We operationalize extensible DEL grammars using:
- Parser Parser Combinators for performing lightweight syntactic rewriting. PPCs enable an extensible parser-generating procedure that operates directly on source code without the need for an intermediate data structure. This enables:
- Declarative specification of rewrite templates (syntactically close to the target syntax) that are interpreted as parser definitions, eliding the need for additional translation layers converting the user-facing templates to an underlying representation and vice-versa.
- A large-scale empirical evaluation to demonstrate the effectiveness of our lightweight approach.

We evaluate our approach on 12 languages comprising 185 million lines of code using 30 transformations. We ran our transformations on the 100 most popular GitHub repositories for each language. By targeting actively developed projects, we had the opportunity to submit our changes upstream; our approach has produced over 50 syntactic changes that are merged into over 40 of the most popular open source projects on GitHub. Merged changes reflect an important additional validation that our transformation engine produces correct and desirable syntactic changes. Our evaluation on popular open-source projects also separately demonstrates the opportunities for lightweight transformation in practice, particularly for transformation tooling that may adapt to accommodate our representation or parsing approach. We also compare our multi-language approach to nine checkers and rewriters for individual languages. We demonstrate declarative specifications (typically less than 10 lines each) for existing real world transformations that otherwise require tens of lines of implemented in different languages in current real world tools.

We release our tool implementing the above ideas, our rewrite specifications, and over 100 real-world example transformations produced by our work.[1]

2 Motivating Example

The `staticcheck`[2] tool for Go detects buggy and redundant code patterns. Like many language style guides, it provides a practical description of "do X instead of Y" for code. An example from the documentation explains:

```
You can use range on nil slices and maps, the loop
will simply never execute. This makes an additional
nil check around the loop unnecessary.
```

[1] https://github.com/squaresLab/pldi-artifact-2019,
https://github.com/comby-tools/comby

[2] https://staticcheck.io/docs/#overview

Before:

```
if s != nil {
 for _, x := range s {
    ...
 }
}
```

After:

```
for _, x := range s {
  ...
}
```

Match template:

```
if :[var] != nil {
 for :[_] := range :[var] {
    :[body]
  }
}
```

Rewrite template:

```
for :[_] := range :[var] {
  :[body]
}
```

Figure 1. *Top:* A textual description for simplifying a `nil` check Go code, taken from the Go `staticcheck` tool. *Bottom:* Our match template and rewrite templates for the `nil-check` pattern above.

Regular expressions cannot generally recognize patterns like the `nil-check` simplification at the top of Fig. 1. The traditional solution is to write a checker that visits the program's AST to recognize the pattern. Checker mechanics are hidden in an implementation, diverging from Fig. 1's intuitive, syntactic before/after description. This programmatic implementation disconnects the checker code from the pattern, making it harder to understand and modify.

In our declarative approach, the user instead specifies a match-and-rewrite pattern that closely resembles and is as natural as the syntactic description. The *match template* for `nil-check` pattern is shown at the bottom of Fig. 1. The syntax `:[`*identifier*`]` denote *holes* with the name *identifier*. Every instance where the template matches source code produces an environment. An environment binds variables to string values (corresponding to syntax) in the source code. The identifier _ acts as an ordinary identifier where we don't particularly care to name the variable. Multiple occurrences of a variable in a template (like `var`) imply that matched values must be equal for matching to succeed.

The match template is expressive enough to match the syntactic structure of the general pattern in Fig. 1. Three core ideas make this possible:

1. The *built-in* understanding that delimiters `{...}` in the match template denote well-balanced (and possibly nested) syntax that must correspond to delimited syntax in the source.
2. Any syntax besides balanced delimiters and holes in the template (e.g., non-whitestape tokens like `if`, `!=`, `range`) *correspond to concrete syntax* in the source. Whitespace in the template is interpreted as a regular expression matching one or more consecutive whitespace characters.
3. Match *rules* govern whether the match succeeds. Rules act as additional constraints on values in the environment (e.g., we check that matches of `:[var]` are equal).

Section 4 explains the matching approach in detail. After the matching phase, we can use a *rewrite template* to perform a change to the program. A rewrite template takes as input an environment, and substitutes variables for values. Our rewrite template for `nil-check` is also declarative (cf. Fig. 1).

The match and rewrite templates are the only inputs needed to detect *and rewrite* instances of `nil-check`. Fig. 2a is a real-world example where we applied our `nil-check` rewrite specification, producing the simplified code as-is in Fig. 2b.[3] The rewrite is nontrivial: one match is contained inside another. Importantly, *each instance* matched by the template (which may be nested or occur in sequence) produces an environment to be instantiated using the rewrite template at that location. In this sense, our match-and-rewrite approach mimics traditional recursive, visitor-like tree traversals without a need for an actual AST visitor.

```
func (c *SymbolCollector) addContainer(...) {
  if fields.List != nil {
    for _, field := range fields.List {
      if field.Names != nil {
        for _, fieldName := range field.Names {
          c.addSymbol(field, fieldName.Name)
        }
      }
    }
  }
  ...
}
```

(a) Highlighted lines 2 and 4 contain redundant `nil` checks in Go code: iterating over a container in a for loop implies it is non-nil.

```
func (c *SymbolCollector) addContainer(...) {
    for _, field := range fields.List {
        for _, fieldName := range field.Names {
          c.addSymbol(field, fieldName.Name)
        }
    }
    ...
}
```

(b) Rewrite output simplifying the Go code above.

Figure 2. Redundant code pattern and simplification.

Indeed, presently, despite the fact that nothing about this approach is Go-specific, implementing the `nil-check` specification invariably requires creating or interfacing with a Go-specific tool operating on a parsed AST. One approach to multi-language support is an intermediate representation (e.g., `srcML` [44]), but this requires a translation layer and remains a programmatic approach. In language-specific approaches, tool design is dictated by the language's AST. This precludes, for example, the use of a general visitor framework for both Python and Go. This may be unavoidable due to

[3] This code was merged into a popular Go repository: https://github.com/sourcegraph/go-langserver/pull/324. We did reindent the code using `gofmt` so that it conforms to stylistic conventions. Reformatting is not always necessary.

language-specific features and tool application. However, for general syntactic transformations like Fig. 1, our approach avoids repeated implementation effort across different languages. Moreover, interfacing with the AST is programmatic in language-specific AST visitor frameworks. Similarly, language workbenches supporting transformation for multiple languages (e.g., [17, 38]) generally define an AST representation per language with particular sensitivity for matching program terms. In this design, making transformation declaratively accessible requires an additional translation layer from a surface syntax mapping to terms (e.g., [55]). In general, the specificity of terms in AST definitions determines the granularity at which such terms can be matched using generic tree matching procedures. This specificity (which can result in definitions spanning hundreds of lines)[4] can increase the complexity of the surface syntax for complex transformations (e.g., when disambiguating syntax [36]). By contrast, we propose a particularly coarse representation for lightweight transformations that is loose with respect to matching and rewriting specific terms. Our declarative transformation approach notably skirts the issue of defining a concrete representation for manipulating code (like ASTs) or defining additional translation layers (e.g., [55]) by creating on-demand parsers that match syntax of interest. The foundation of this idea lies in first identifying a suitable grammar for expressing syntax of multiple languages, discussed next.

3 Dyck-Extended Languages

We use the Dyck language as an initial building block to form a general and extensible representation for syntactic rewriting. The Dyck language is simply an abstraction of well-nested expressions over a single pair of symbols that form an open and closing delimiter pair. I.e., taking the symbols "`(`" and "`)`" as delimiters, `()()` and `(())` are valid expressions in the Dyck language. A generalization of the Dyck language maintains a well-nested structure over n pairs of delimiting symbols, denoted D_n. Like other literature, we refer to D_n as "Dyck Languages". The definition of D_n is:

$$S \longrightarrow \epsilon \mid SS \mid x_1\, S\, \overline{x}_1 \mid x_2\, S\, \overline{x}_2 \mid \cdots \mid x_n\, S\, \overline{x}_n$$

where ϵ represents the empty string, S is the only nonterminal symbol, and terminal symbols $(x_i, \overline{x}_i)$ for each $i \in n$ delimiter pairs. We denote the terminal alphabet for this grammar as $T = X \cup \overline{X}$ where $X = \{x_1, \ldots, x_n\}$ and $\overline{X} = \{\overline{x_1}, \ldots, \overline{x_n}\}$ are disjoint sets over opening and closing symbols, respectively.

Our key addition to the grammar allows interpolating strings between and inside balanced delimiters in D_n. We add the rule $S \longrightarrow c$, for terminal c, where $c \in \Sigma$ for a finite alphabet Σ disjoint from $X \cup \overline{X}$. We call this a general Dyck-Extended Language (DEL) grammar. To be practically useful for parsing conventional programs, we specialize this grammar and impose:

1. a finite set of matching delimiters, e.g., `(`,`)`, `{`, `}`, `[`, `]` as found in common languages.
2. a refinement that strings produced from the finite set Σ is regular, where strings correspond to tokens.

```
grammar ::= term EOF
term ::= '(' term ')' | '{' term '}' | '[' term ']'
       | term term | token
token ::= whitespace | string_literal
       | comment | any_token
---------------------------------------------------
whitespace ::= [ \n\t\r]+
string_literal ::= "<escaped string>"
comment ::= <single or multiline comment>
any_token ::= [...]*
```

Figure 3. A base grammar for parsing multi-language syntax. Productions above the dashed line are common to many contemporary languages. Below the dashed line, token structure may vary in minor ways (e.g., different string literal or comment syntax).

A DEL for multi-language transformation. In this paper, we use the DEL definition shown in Fig. 3 as a basis for conventional languages, which extends the grammar above. The extension defines concrete delimiters and partitions three kinds of tokens: whitespace, string literals, and comments. String literals can contain any character (with appropriate escaping, as usual). All other consecutive characters not already defined (i.e., ... indicates any character excluding whitespace characters, string delimiters and comments) form an `any_token`. In practice, the finite character set of the language is that of typical source code (i.e., ascii or unicode character sets). The distinction between string literals and comments is significant because these are the primary categories that, if ignored, break the ability to recognize well-balanced delimiters in code. String literals and comments are simultaneously a source of syntactic variability and ambiguity across languages. Handling these categories explicitly across languages allows our approach to correctly identify nested terms within those languages.

There are two notable properties of the representation. First, a parser for this language preserves all syntax (including whitespace) and partitions all characters coarsely into one of a few lexical constructs. This means that many separate lexical constructs in typical grammars are treated as one lexical construct under `any_token` (i.e., we don't distinguish keywords from variables). Second, hierarchical tree structure is determined solely by balanced delimiter syntax. Importantly, the representation preserves a well-formedness property of nested terms with respect to these delimiters when we rewrite code. Our intuition is that this representation models the essential concrete syntax for rewriting many conventional languages like C, Python, OCaml, etc., with expressivity to describe nested structures. At a high-level, our

[4] See, e.g., https://github.com/usethesource/rascal/blob/master/src/org/rascalmpl/library/lang/java/syntax/Java18.rsc

representation can be seen as imposing a kind of s-expression representation on contemporary language syntax, extending similar flexibility of lisp-like macro systems [14, 15, 25, 57] to many contemporary languages by accounting for varying syntactic idiosyncrasies (e.g., different sorts of balanced delimiter, comment, and string literal syntax) in a modular grammar.

In Fig. 3, the delimited terms and token structure above the dashed line denote commonly *shared* (though coarse) productions for many contemporary languages. Below the dashed line, token syntax may *differ* across languages in minor ways (e.g., quotes used for string literals, raw string literals, or differing comment syntax). The crux of our representation is that it allows a modular and extensible parsing strategy. Using parser combinators [35], we design a modular skeleton parser for the shared constructs (above the line Fig. 3) that is "open": parsers for token groups (below the line) plug into the skeleton parser. Our parser combinator architecture thus allows easy modification to handle syntactic idiosyncrasies at the token level, while preserving the essential CFL properties denoted by balanced delimiter syntax. In our experience, rewrite support for multiple languages with this architecture boiled down to tiny modifications in token parsers (typically one line of code) for handling individual language comment or string literal syntax. We now explain how the DEL representation, and the parse strategy for it, ties into matching and rewriting syntax.

4 The Rewrite Process

The rewrite process has two phases: (1) matching source code using a user-specified *match template* and (2) rewriting source code based on a user-specified *rewrite template*. We explain the intuition of our matching approach in Section 4.1. We introduce our key innovation in Section 4.2 that lifts the intuition of tree matching to a parser generator problem for modular grammars (as introduced in Section 3). We explain the rewrite phase in Section 4.4.

4.1 Declarative Matching: The Intuition

Suppose we have a DEL grammar parser for Go that produces DEL trees for Go code (i.e., the parser handles Go comments and string literals). The idea behind *declarative* match templates is that they translate directly to an underlying DEL parse tree to be matched against a DEL parse tree produced by source code. Match templates additionally contain holes that unify with syntax in the source parse tree. Successful matches produce environments binding variables to syntax, which are substituted into the rewrite template in the rewrite phase (Section 4.4)

Fig. 4 visualizes a match of template to source for our motivating example. On the top is a parse tree of the source code; on the bottom, a parse tree for the template. Dashed lines indicate matched tokens. Concrete tokens match syntactically in template and source. Balanced delimiters (e.g., `{}` braces) form nodes corresponding to terms that contain nested child terms; when encountered in the template, balanced delimiters must match correspondingly in the source.

Holes match to syntax based on (a) prefix and suffix matches of surrounding terms and (b) their level in the tree. As shown, the `:[_]` hole can only bind to sequences of terms inside the brace-delimited `if`-body. The matching semantics of `:[`*identifier*`]` is to match lazily up to its suffix. In practice, we do not need an explicit token class for the category of `any_token` in Fig. 3: we simply treat these as individual characters (i.e., each character can be considered its own term). Thus, holes can bind to any sequence of characters (akin to regular string matching), restricted within its current level in the tree. A hole's suffix is terminated by (a) another hole or (b) the end of the sequence of characters (preserving proper nesting) on its level. For example, the hole `:[_]` matches "`_,␣x`" up to its suffix "`␣:= range`", after which another hole occurs at the same level in the template. A hole immediately followed by another hole (e.g,. `:[1]:[2]`) implies that the first hole matches up to the empty string (i.e., `:[1]` will always be empty). We have found it useful to define additional hole syntax which binds only to alphanumeric characters, written `:[[`*identifier*`]]`.

We do not define additional metasyntactic operators (e.g., regex operators); *all* other syntax in the template relates to *concrete* matching.[5] The concrete specification of templates is separated from their *interpretation* which must be implemented in the matching procedure. For example, delimiter syntax implies matching must descend into the tree. For convenience, we also interpret a single space in the template as matching repetitions of whitespace in the source by default.[6] This matching behavior can be modified to expect exact whitespace matching in the tool configuration, if desired.

Implementing the match procedure as described raises two intertwined but separate concerns: (1) producing parse trees from match templates and source and (2) matching templates to source code by traversing the trees. These present two design challenges: defining a concrete data structure to represent DELs (i.e., a tree, the output of parsing), and implementing a configurable tree traversal strategy to enable configurable and flexible match interpretations.

Our insight is that both challenges are solved simultaneously by lifting tree construction and matching to a modular parsing problem. Doing so enables structural matching relating template and source (a) without any concrete representation at all and (b) with configurable matching represented directly by parsers, allowing for syntactic differences across

[5] Nothing stops us from complicating templates with more metasyntax; in practice we find doing so eventually limits declarative specification.

[6] This is desirable in our running example when we don't particularly care about the indentation size or other additional whitespace.

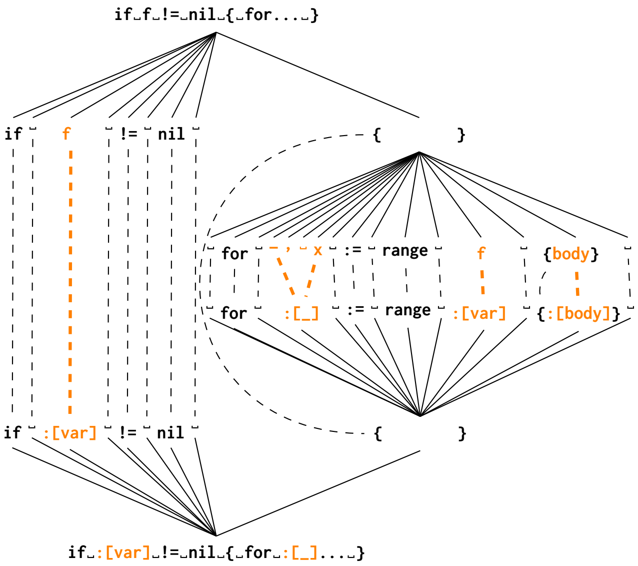

Figure 4. Visualization of matching for the `nil-check` pattern (Fig. 1) using the DEL representation. On the top is a parse tree of source code; on the bottom a parse tree for the match template. Dashed lines represent matched terms. Thick lines indicate assignment of terms to variables.

languages, and the ability to match character sequences (e.g., like regular string matching).

4.2 Parser Parser Combinators for Matching

The key innovation behind our approach is an extensible procedure for structural matching that relates match templates and source code without a concrete data structure via parser combinators [35]. The core idea of parser combinators is to model parsers as functions that can be composed using higher-order functions (combinators) to implement grammar constructions. Higher-order functions make the parsing process separable. Thus, composable parser combinators enable a modular parsing procedure for the modular grammar we developed in Section 3. We might, for example, use a modular parsing strategy to produce DEL parse trees for matching. However, this still leaves open the problems of (a) mapping parse results to a tree definition, then (b) defining a configurable match traversal strategy over this representation, and (c) creating an accessible declarative language to mapping user-specified templates to terms in the tree definition (as in [55]).

Instead, we use parser combinators in a new way. We first recognize that parser combinators can be composed to parse DEL languages. Such a DEL parser, when running on source code, performs the analogous role of matching valid parses of concrete syntax (to generate, in this case, a parse tree). Interestingly, a declarative match specification is nothing other than a description of particular concrete syntax (interpreted by a DEL parser), combined with holes, that is to be matched. Thus, a match specification can be interpreted as description of a parser for matching, rather than a concrete tree to unify against a parse tree of source code. In particular, it describes a valid parse for a DEL language fragment.

This insight leads to a solution where our approach generates on-demand parsers for DEL language fragments from match templates. We develop what we call a Parser Parser Combinator (PPC): a parser-generating parser using parser combinators. The approach works by implementing a parser for match templates *whose output is a parser* (constructed from parser combinators at runtime) that matches syntactic patterns according to the template. The match template PPC behaves as follows:

- When parsing a DEL token in the template (e.g., a string like `if`), generate a parser for that token.
- When parsing whitepsace, generate a parser to consume whitespace.
- When parsing nested syntax (indicated by delimiters), generate a parser for nested syntax between those same delimiters.
- When parsing holes (`:[ ]`), generate a parser that binds syntax to the identifier and continues lazily up to the suffix in the template (i.e., generate a parser implementing the match semantics described in Section 4.1 and store the result in the parser state).

The PPC automatically chains generated parsers in these categories *as it parses* the template at runtime. The resultant generated parser is lazy (i.e., it is a partial function) which encodes exactly the behavior for matching syntax of interest. The appeal of this approach is that we can now simply run the resultant parser directly on source code. The result of the generated parser is only the environment binding variables to syntax for successful matches. No intermediate data structure is needed to perform any matching. The PPC approach also offers key advantages for configuring matching. First, the *interpretation* of match template syntax (e.g., match semantics of holes, or whether to treat whitespace significantly) is easily modified by augmenting the parser generating behavior of the PPC when it encounters any particular kind of syntax in the template. Second, our approach allows matching sequences of characters familiar in regular expression-based code changes (e.g., [3]) as opposed to restricting matching to complete tokens or terms. In practice, match template content for non-whitespace characters maps to a matching parser character-for-character (analogous to lexing, but without classifying specific tokens). Importantly, well-formedness of nested delimiters is preserved when matching character sequences—a character cannot match, for example, an unbalanced parenthesis character unless it occurs inside a string literal or comment. Third, the PPC is extensible: it is a module exposing hooks in the parser-generation routine (which is modular, like the DEL grammar) so that parsing language-specific syntax (e.g., strings, comments) is expressed in the generated parser. The PPC is extended using small definitions that express language-specific syntax properties. For example, the three definitions below form a module defining the syntax of the base DEL parser:

```
let user_defined_delimiters =
    [ "(", ")"; "{", "}"; "[", "]"]
let string_literals = []
let comment_parser = []
```

The C parser extends the base parser It includes the base delimiter syntax, and adds the syntax for string literal and comment syntax using the prescribed definition names:

```
include Base.Syntax
let string_literals = ["\""; "'"]
let comment_parser =
    [Multiline ("/*", "*/"); Until_newline "//"]
```

Constructors like `Multiline` denote, e.g., that C-style block comment syntax should be used to parse multiline comments.[7] Parsers are created from these syntax definitions and embed into the PPC to parameterize *both* the significance of custom syntax in user-supplied match templates *and* the significance of the same custom syntax when the resulting matcher runs on source code. For example, the C definitions above reflect in the user-specified template (the parser now understands that holes inside string delimiters should match within string delimiters) and ensures that parentheses inside strings in the target source code (like "(") do not affect the parsing of balanced parentheses.

We implement our PPC using a left-to-right, top-down parsing strategy. Any ambiguity in the grammar is resolved by the ordered choice combinator (as in, e.g., [28, 41]), where terms are parsed similarly to the order of Figure 3.[8] The PPC implementation is roughly 500 lines of OCaml code. In practice, applying our PPC to 12 languages requires fewer than 12 lines of code to account for the syntactic differences in comments and string literals. Command-line flags activate parsers for a particular language.

Match contexts. Fig. 4 illustrates matching one instance of a template to an exact match of source code. Similarly, PPC's produce a parser that match one instance specified by the template. In practice, we typically want to find multiple matching instances in source code. We call each matching instance a *match context*. A match context describes (1) the range of characters in the source code matching the template in its entirety and (2) the unique environment binding holes to values for that instance.

For practical reasons, our objective is not to find *all* matching contexts, but rather non-overlapping matches over sequences. For example, using the match template `a␣b␣a` and a source `a␣b␣a␣b␣a` produces only the match `a␣b␣a␣b␣a` and not `a␣b␣a␣b␣a`. Disallowing sequentially overlapping match contexts allows an unambiguous rewrite result. We produce match contexts by shifting over the source code left-to-right and attempting to parse at each point. If the PPC-generated parser succeeds at any point, we record the match context (i.e, the match range and environment). We then shift to the end of the match range and continue trying to parse at each point until the end of source is reached. Our tool allows the same procedure to be applied successively on content in match environments, thereby recursively rewriting nested matches.[9]

4.3 Match Rules

Match rules apply additional constraints on matching. Match rules are high level predicates on environments in match contexts. Our motivating example demonstrates the utility of predicates on matching. For example, we can check that the variable ranged over is the same as the variable compared to `nil`. As in Fig. 1, using the same variable identifier for multiple holes adds the implicit constraint that values matched by these holes should be equal. This constraint is really syntactic sugar for a larger feature set of constraints that can be specified explicitly and declaratively in a small DSL-like notation.

A match rule may accompany a match template and has the form `where <...>` and a list of comma-separated expressions and evaluates to a boolean value. Commas are interpreted as logical conjunction over the expression list. If the value is true, a match succeeds. The default match rule is `where  true`. The grammar is:

```
grammar ::= "where" <expression> <"," expression>*
expression ::= "true" | "false"
  | <atom> "==" <atom> | <atom> "!=" <atom>
  | "match" <atom> "{" <branch> <"|" branch>* "}"
atom :: = ":["variable"]" | "<string literal>"
branch ::= atom "->" expression
```

Figure 5. A simple constraint grammar for match rules.

Operators `==` and `!=` check for syntactic equality (resp. inequality) on atoms. There are only two atoms: one for variables (expressed in hole syntax) and string literals. The `match` expression applies (disjunct) conditional constraints on values bound to holes. It is sometimes useful to treat values as templates in `match` expressions, rather than string values. When string literals contain hole syntax, they are interpreted as match templates. In this case, we may write *match templates* for the antecedent in match cases which produce match contexts when evaluating the branch expression. For example,

```
match :[expr] { ":[1] < :[2]" -> :[1] == :[2] }
```

[7] For brevity we elide other optional definitions that can further refine the PPC, e.g., raw string literals that have different escaping criterion.

[8] We overlooking the possibility of left-recursion in this grammar, which we gave to simplify presentation.

[9] Alternatively, the pass may be rerun to a fixpoint on the entire file.

evaluates `:[1] == :[2]` *if* the value bound to `:[expr]` produces a match context for the match template `:[1] < :[2]`. The branch expression must evaluate to true for all match contexts produced by the template.

4.4 The Rewrite Phase

The rewrite phase takes all match contexts produced by the match phase and substitutes the environment variables for values in a user-specified rewrite template. The rewrite template can be seen as a partial function. We say that a rewrite template is *instantiated* by an environment when all variables in the rewrite template are substituted for values. After a rewrite template is instantiated, it replaces, in-situ, the entirety of matched content for a given match context. Thus, there are two substitution passes: (1) substitution to instantiate rewrite templates and (2) contextual substitution of instantiated templates in the source. The rewrite procedure applies recursively for nested matching contexts, and are substituted bottom-up as in the motivating example.

Well-balanced output. During the match phase, only well-balanced terms can be matched by holes. By extension, rewrite output is well-balanced if and only if the rewrite template is well-balanced. Thus, rewrites preserve key syntactic properties of well-balanced delimiters found in most languages.

Rewriting comments. Associating comments across program manipulation is a known and long-standing problem, and not specific to our approach [46]. Holes *do* capture comments and can be recovered during rewriting. However, comments are ignored when interpreting match templates (otherwise, templates would require the user to anticipate comment location and content to match other significant syntax). Matched syntax is replaced according to the rewrite template as usual, and when matching on comments is not significant, those comments may be erased by the rewrite if they are not captured by holes.

The underlying problem is that we cannot anticipate how comments may be associated with syntax for arbitrary languages. In lieu of better language design (which appears inapplicable to contemporary languages) we note that inference techniques or explicit comment support (e.g., adding an extra dimension of comment contexts) could provide solutions for manipulating comments. We leave these areas open to future work.

5 Evaluation

We evaluate Comby, an implementation of our rewrite process (Section 4) for DELs (Section 3) using Parser Parser Combinators (Section 4.2). Section 5.1 evaluates Comby on 12 languages comprising 185 million lines of code using 30 transformations. We demonstrate that our approach is fast; enables simple, declarative specifications for nontrivial transformations; and that it is effective in *real* (not just realistic) settings, to date producing over 50 changes merged into 40+ of the most popular open source projects across 10 languages.

Table 1. We ran 1–3 rewrite patterns (**Pattern**) per language (**Lang**) on the top 100 most-starred GitHub projects for that language. **K Files** and **MLOC** is the aggregate thousands of files and millions of lines of code, respectively. **Proj** is the number of projects where patterns match; **#M** the number of matches; **Time**, the aggregate time to match/rewrite over all files.

Lang	**K Files**	**MLOC**	**Pattern**	**Proj**	**#M**	**Time**
Go	131.4	48.8	nil-check	40	372	7m56s
			str-contains	16	29	3m02s
Dart	28.2	4.9	slow-length	6	35	24s
			where-type	10	165	35s
Julia	5.2	0.9	simple-map	5	8	23s
			tweaks (3)	12	38	45s
			redun-bool	1	1	24s
JS	66.1	9.2	redun-bool (2)	3	89	7m14s
Rust	22.0	3.8	short-field	42	305	21s
			redun-pat (4)	2	12	1m15s
Scala	41.9	5.2	parens-guard	10	43	30s
			forall	2	5	23s
			count	13	46	23s
Elm	4.8	0.9	dot-access	2	3	13s
			pipe-left	4	10	13s
OCaml	25.6	6.3	include-mod	11	77	2m01s
			include-func	4	5	2m04s
C/C++	166.6	85.5	no-continue	2	5	5m58s
Clojure	5.1	0.7	simpl-check (4)	34	330	21s
Erlang	13.2	4.3	infix-append	25	187	25s
Python	34.5	15.2	dup-if-elif†	4	10	6m59s

In Section 5.2 we compare our approach to nine existing language-specific checker and rewrite tools. We show that our multi-language approach produces accurate syntactic checking and transformation for many syntactic properties covered by current tools, while requiring far less implementation effort. Our results indicate that Comby meets the demands of real-time development [24, 47]: transformation response time is in the 100-400ms response time per file. Section 5.3 provides further discussion and limitations.

5.1 Real, Large Scale Multi-Language Rewriting

Experimental setup. We cloned the top 100 most popular repositories on GitHub (ranked by stars) for each of 12 languages. We balanced imperative and functional languages (some mature, others more recent) to demonstrate broad applicability. The 1,200 repositories comprise over 185 million lines of code. We wrote 1–3 rewrite patterns per language (we elaborate below). We ran each rewrite pattern in parallel by file across 20 cores (Xeon E-2699 CPU, Ubuntu 16.04 LTS server, limiting experiments to 1GB of RAM). Matching was set to timeout after 2 seconds per file. Roughly 2% of Python files timed out (large, autogenerated code), while timeout for files across other languages accounted for less than 0.5%.

Choosing rewrite patterns. We chose patterns from existing linters, checkers, rewrite tools, and style guides that were likely to improve code readability or performance. Since we submitted many of our transformations to active code bases, we avoided stylistic patterns that were a matter of taste. The selected patterns either (1) remove clearly redundant code; (2) simplify readability; (3) replace functions with more performant versions; or (4) identify clearly buggy duplicate checks. We include nontrivial patterns that match on balanced delimiters for every language.

This part of our evaluation prioritizes (a) breadth of language application and (b) improving active real code. We therefore applied 1–3 choice patterns per language rather than a comprehensive catalogue of patterns for any particular one.

Results overview. Table 1 shows the results of running multiple rewrite patterns across 12 languages. The rewrite patterns are shown in the **Match Template** and **Rewrite Template** columns of Fig. 6. As written, these are the actual inputs to Comby (no extra metasyntax or programming required).[10] We applied 30 unique patterns to our data set. Some patterns, like "tweaks" and "redun-bool" check for similar patterns with syntactic variations (we show a representative transformation for these in the table). The number of variations is indicated in parentheses in the **Pattern** column, and the running time is an aggregate over variations. In aggregate, running all patterns takes about 42 minutes and parses 282 million lines of code (by common word count including newlines and comments).

Rewriting is fast: 13 out of 21 patterns terminate in under a minute for all 100 projects of that language, while the longest runtime for a pattern takes roughly 8 minutes (`nil-check` for Go code). Generally, longer runtimes correspond to larger codebases of popular languages (e.g., Go, Javascript, Python, and C). Match templates and source code content also affect runtime: The Go `nil-check` pattern takes more than twice as long compared to the `str-contains` pattern due to increased cases of matching the `if :[1]...` pattern.

The number of projects for which a rewrite pattern applies range from 1 to 42 (**Proj**) and the number of matches from 1 to 372 (**#M**). To demonstrate our multi-language rewrite approach in a real setting, we submitted a subset of these changes for merging into upstream repositories via GitHub pull requests (PR). This demonstrates both the opportunities for syntactic transformation in large, highly popular repositories and the ability of our approach to produce actionable, correct, and automatic syntactic changes in this setting. Submitting all changes was not appropriate because changes may touch code dependencies not part of the main project, or code that explicitly tests for bad patterns (e.g., linter tests). Unless stated otherwise, we used the following criteria to submit pull requests: (1) the project must be active (code has been committed in the last 30 days); (2) the changes should not affect files under a `test` or `vendor` directory; (3) we prefer projects where we can validate syntactic changes via a continuous integration build.

The results of our PR campaign are shown in Table 2. We submitted PRs to 50 projects. The **Merged** column shows the number of syntactic changes submitted and the outcome:

✓: 43 projects accepted more than 50 individual changes.
🕒: 3 projects have a PR in progress, or unresponded to.
–: 4 projects rejected changes unrelated to change content.
✗: Only two changes were outright wrong.

The number of PR changes in the **Merged** column are either fewer or equal to the number of **Matches**, since we filtered out matches in test or vendor directories. Across all projects, the maximum time for a project is 19 seconds (for `Kubernetes`, the largest project), while the median time per project is 0.55 seconds. We elaborate on patterns and results for each language below.

Go. The `nil-check` pattern identifies redundant `nil` checks, as in our motivating example. The `str-contains` rewrite uses a clearer `string.Contains(...)` call compared to checking string indices for substring containment (cf. Fig 6). Changes from all of our PRs are accepted except for one still in progress. Our PR to the `Go` compiler highlights an interesting case in that it is the only case where a syntactic change was outright wrong. The problem is that removing the check on `:[1]` in the pattern is only allowed on maps and slices, but not pointer types. In the `go` PR, two cases compare to pointers. In the remaining PRs, these comparisons were done against appropriate types, and safe to remove. We make two observations. First, purely syntactic transformations are vulnerable to language design where equivalent syntactic changes have different semantic implications. Thus, while our approach suffers from syntactic ambiguity, we note that that this ambiguity is a shortcoming that should be, in principle, avoided during language design for purposes of program manipulation [46]. Second, type information aids legal and complex syntactic changes (e.g., in refactorings [51]) and complements our approach. We note that recent uptake in the Language Server Protocol (LSP)[11] presents an elegant method for incorporating type information into our approach. With LSP, we can use external language servers (e.g., the `go-langserver`) to lookup types without having to parse, maintain, or persist type information for rewriting.

Dart. The `slow-length` pattern improves performance when checking the length of iterable containers (e.g., lists). The `where-type` pattern improves readability and removes redundant runtime checks. Both patterns are in the Effective Dart usage guide [11]. Because the `sdk` project does not have a CI

[10] Modulo newline stripping, for presentation.

[11] https://microsoft.github.io/language-server-protocol

Lang	Pattern	Match Template	Rewrite Template

Go	nil-check
	str-contains
		`if :[1] != nil { for :[2] := range :[3] {:[4]} }`
		`if strings.Index(:[1], :[2]) != -1 {:[3]}`
			`for :[2] := range :[3] {:[4]}`
			`if strings.Contains(:[1], :[2]) {:[3]}`

Dart	slow-length
	where-type
		`if (:[1]s.length != 0)`
		`.where((:[1]) => :[1] is :[[type]])`
			`if (:[1]s.isNotEmpty)`
			`.whereType<:[type]>()`

Julia	simple-map
	tweaks (3)
	redun-bool
		`map(:[1]->:[[x]](:[1]), :[y])`
		`ceil(:[1]/:[2])`
		`(:[1] || :[1])`
			`map(:[x], :[y])`
			`cld(:[1],:[2])`
			`(:[1]) # redundant expr`

JS	redun-bool (2)	`(:[1] && :[1])`	`(:[1]) // redundant expr`

Rust	short-field
	redun-pat (4)
		`{ :[[field]]: :[[field]] }`
		`if let Ok(_) = Ok:::[t](:[v])`
			`{ :[field] }`
			`if Ok:::[t](:[v]).is_ok()`

Scala	parens-guard
	forall
	count
		`for {:[body]if (:[1]) }`
		`.foldLeft(true)(:[1] && :[1])`
		`.filter(:[1]).size`
			`for {:[body]if :[1] }`
			`.forall(identity)`
			`.count(:[1])`

Elm	dot-access
	pipe-left
		`List.map (\:[[x]] -> :[[x]].:[[f]]) :[[vs]]`
		`|> List.map :[[fn1]] |> List.map :[[fn2]]`
			`List.map .:[f] :[vs]`
			`|> List.map (:[fn1] >> :[fn2])`

OCaml	include-mod
	include-func
		`module :[m] = struct include :[[x]] end`
		`module :[m] = struct include :[[x]](:[b]) end`
			`module :[m] = :[x]`
			`module :[m] = :[x](:[b])`

C/C++	no-continue	`for (:[1]) { continue; }`	`for (:[1]) {}`
Clojure	simpl-check (4)	`(= :[1] nil)`	`(nil? :[1])`
Erlang	infix-append	`lists:append(:[1], :[2])`	`:[1] ++ :[2]`
Python	dup-if-elif†	`if :[1]::[_] elif :[1]:`	`if :[1]::[_] # duplicate check`

Figure 6. The match-rewrite templates for the patterns in our large-scale evaluation. We performed additional preprocessing for the `dup-if-elif` Python pattern (†).

build on GitHub, we validated a subset of syntactic changes locally. The `slow-length` pattern is susceptible to syntactic ambiguity: if the object is a string, there is no corresponding `isNotEmpty` method. As a heuristic, we changed the pattern to match variables ending in 's' as a likely indication that an Iterable object was checked (e.g., matches included '`columns`', or '`argNames`'). This presents an interesting case where matching at the character granularity can be desirable. The changes using this heuristic were all successfully validated by CI builds or local compilation. Projects with matches for `where-type` were all inactive or deprecated, except for the Dart `sdk`. Not all objects implement the `whereType` method, so the rewrite may fail to compile (none of the changes we validated failed to compile). In both cases, additional type or interface information complements our approach.

Julia and Javascript. The `simple-map` pattern removes redundant code produced by anonymous function syntax. The `tweaks` patterns can improve performance for floating point operations in tight loops. Both of these patterns are taken from Julia's style guide and performance tips. We discovered 38 cases where `tweaks` could potentially improve performance (Table 1), but this required benchmarking individual project performance before and after the change. Due to the extra burden of validating performance effects, we skipped PRs for this pattern.

We also checked Julia and Javascript code for redundant boolean expressions. On the one hand, this check is desirable because the pattern clearly indicates redundant or (more often) wrong code. On the other, it can be hard to predict a correct rewrite output when the code is wrong. By default, our rewrite output produces a semantically equivalent change with a comment to note the problem. Overall, this pattern demonstrates fast matching capability with optional rewriting. The one issue in Julia code and three issues in Javascript needed manual fixing;[12] in the case of `angular`, the expression was redundant and corresponds to the rewrite template `:[1]`.

Rust. The `short-field` pattern removes redundant single named fields for data structures (note we use `:[[..]]` match notation to limit matching to single identifiers). The `redun-pat` eliminates redundant `let...` pattern matching for option and result types. Both checks are implemented in an existing rust linter. Our largest pull request updates 68 instances where the Rust compiler now uses shorthand notation for single field structures (additional matches exist inside test directories). The PR for `clap` was closed because the code is being removed. We did not find real cases of `redun-pat`.

Scala. All three Scala patterns improve readability, and are implemented in existing tools (`scalafmt` and `scapegoat`). The PR for `shapeless` was rejected because the original syntax explicitly tested `foldLeft` behavior. This was one of our initial submissions, and influenced our decision to ignore files in test directories. Three eligible changes for `no-guard-paren` broke whitespace indentation and required reformatting; we skipped these and submitted a single PR where indentation

[12] e.g., `(isnan(a[i]) || isnan(a[i]))` became `(isnan(a[i]) || isnan(b[i]))`

Table 2. Pull Request Results. **Merged** shows the number of changes submitted, and outcome: ✓means changes were merged; 🕒 means a PR is in progress; – means changes were rejected for reasons unrelated to their content; ✗ indicates incorrect changes. †: the `redun-bool` and `dup-if-elif` required manual fixes.

Lang	Pattern	Project	#KLOC	#M	Merged
Go	nil-check	go-langserver	247.5	2	✓ 2
		helm	50.2	1	✓ 1
		prometheus	902.4	2	✓ 1
		rclone	596.9	10	✓ 1
		go	1,703.2	3	✗ 2 ✓ 1
		cli	10.4	1	✓ 1
		rancher	2,695.2	14	✓ 1
		kubernetes	4,236.6	22	🕒10
	str-contains	gorm	14.6	2	✓ 2
		vault	1,553.1	1	✓ 1
		vitess	297.7	2	✓ 2
Dart	slow-length	sdk	2,361.3	18	✓ 3
		sqflite	6.9	1	✓ 1
		over_react	23.3	1	✓ 1
Julia	simple-map	Dagger.jl	3.7	1	✓ 1
		JLD.jl	4.5	1	✓ 1
		IJulia.jl	2.7	1	✓ 1
	redun-bool†	DataFramesMeta.jl	1.5	1	✓ 1
JS	redun-bool†	Rocket.Chat	149.2	3	✓ 3
		angular	297.8	1	✓ 1
Rust	short-field	rust	1,004.8	133	✓68
		clap	33.6	6	– 6
		conrod	34.1	5	🕒 5
		coreutils	45.5	1	✓ 1
		exa	9.7	1	✓ 1
		fd	3.9	1	✓ 1
		gluon	66.4	6	✓ 6
		hematite	3.9	2	✓ 2
Scala	forall	scala-js	190.6	1	✓ 1
		shapeless	47.1	4	– 4
	parens-guard	Ammonite	22.5	1	✓ 1
	count	FiloDB	44.9	7	✓ 7
Elm	pipe-left	elm-graphql	36.4	1	✓ 1
		node-test-runner	40.1	2	– 1
OCaml	include-func	bap	100.7	1	✓ 1
		pyre-check	79.4	1	✓ 1
		hhvm	213.5	1	✓ 1
	include-mod	base	43.1	1	✓ 1
		opium	1.6	1	✓ 1
		flow	147.8	2	✓ 2
		owl	131.0	12	✓12
C/C++	no-continue	radare2	745.3	4	✓ 4
		php-src	1,428.4	1	✓ 1
Clojure	simple-check	pedestal	18.7	12	✓ 1
Erlang	infix-append	zotonic	102.2	5	✓ 5
		lasp	15.7	4	✓ 4
		kazoo	345.8	1	🕒 1
Python	dup-if-elif†	matplotlib	229.3	1	✓ 1
		zulip	140.5	1	– 1
		powerline	30.0	1	✓ 1

was preserved. We discuss reformatting more generally in Section 5.3.

Elm. The Elm patterns simplify syntax by removing redundant anonymous functions (`dot-access`) and composing sequential map operations over list elements (`pipe-left`). We focused on `List` patterns, being a common data structure. Variants for these patterns can substitute `List` for `Array` or `Set`, but these are less common in practice. The `dot-access` pattern only rewrote code for two inactive projects (Table 1), so we did not submit pull requests for these. One `pipe-left` change was accepted, while the other was rejected because it touched files part of external dependencies.

OCaml. The `include-mod` and `include-func` removes redundant module include syntax for modules and functors respectively. The majority of `include-mod` matches are in compiler tests, which we did not submit in PRs. In some cases, the `include-func` pattern required reindentation because the module body (`:[b]`) captures syntax that may span multiple lines. All of these PRs were merged.

C/C++. The `no-continue` pattern removes redundant continue statements inside `for` loops. The pattern is based on a check in `clang-tidy`. A number of variants are possible (e.g., replacing `for` with `while` or `do-while` syntax). We found two real cases of redundant `continue`s in the C/C++ dataset. Both of these PRs have been accepted.

Clojure. The `simpl-check` pattern for clojure simplifies conditional checks. Syntactic variants include testing boolean values with `true?` and `false?` checks. All of the matches in the Clojure data set (Table 1) occurred in inactive projects, except for one. That PR was successfully merged.

Erlang. The `infix-append` pattern is part of the Erlang syntax tools suite. It replaces a `lists:append` function call with a terse ++ infix operator. Two PRs were accepted, while one remains open.

Python. The `dup-if-elif` pattern detects identical conditional checks on both branches of an `if-elif`. Matching on possibly nested and indentation-sensitive `if` statements is challenging compared to concrete syntax (e.g., braces) because whitespace is typically treated insensitively in many languages. We can implement an additional parser combinator hook to support layout-sensitive parsing in the PPC, but currently Comby treats indentation insensitively. However, our rewrite approach presents a convenient and interesting alternative via syntax preprocessing. We used Python's `pindent`[13] utility to annotate the end of indentation-sensitive blocks with comments. We then ran a rewrite rule to convert block annotations to braces compatible with existing delimiter parsers. The preprocess step took approximately 6 minutes and performed 600K rewrites on the Python data set.[14] We discovered three legitimate `dup-if-elif` matches. Like the `redun-bool` pattern, the pattern identifies clearly buggy code and required manual fixes for the two merged PRs. In the remaining PR, we were asked to perform a larger refactoring that removes references to the buggy function so that it can be removed entirely.

[13] https://github.com/python/cpython/blob/master/Tools/scripts/pindent.py

[14] The `pindent` preprocess step is also invertible.

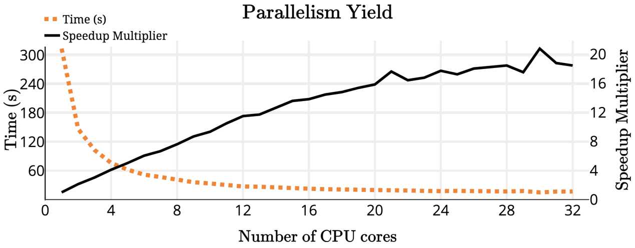

Figure 7. Parallelism yield running the `nil-check` pattern on Kubernetes (4.2 MLOC) up to 32 cores. Parallelism scales linearly up to 12 cores.

Parallel performance and memory use. To quantify memory use and parallelism performance for large projects, we benchmarked our approach on Kubernetes (4.2 MLOC, the largest project in our PR campaign) with the `nil-check` pattern. We ran a precise memory profile of our implementation using OCaml Spacetime,[15] on a single core. Maximum memory use was only 44 MB while the median was approximately 10 MB. Low memory use is attributed to the PPC generating a lazy parser that persists only matching syntax per file.

We evaluated speedup of our implementation up to 32 cores. Fig. 7 summarizes our result. Performance scales linearly up to 12 cores before tapering off. Performance deviates slightly for higher numbers of cores due to heuristic partitioning of work (i.e., files) per core.

5.2 Comparison with Existing Tools

Experimental setup. We compare Comby to nine language-specific checker and rewriting tools in terms of (1) syntax checking and rewrite accuracy, (2) implementation burden, and (3) runtime. The tools provide a variety of functions including refactoring, code simplification, and linter checks. Four tools only check syntax, and do not rewrite. We targeted real world projects where possible; otherwise, we applied patterns on the tool's test suite. We opted to use tests when isolating transformation behavior in tools was difficult. For example, some tools implement multiple syntactic checks under a single command line flag. Individual tool configuration also hinders a direct speed comparison. We provide the wall-clock times for running each pattern to indicate that our running time is competitive, but do not generally claim to be faster.

Experiments were run on a single core (2.2 GHz Core i7) with 1GB RAM. The hardware is comparable to a modern single-user developer machine, and supports our claim that our approach meets demands of real-time development response times.

We prioritized demonstrating breadth of application across languages, and include only a selection of patterns for eightn of the nine languages (many more patterns are possible). For one functional language (Clojure) we implemented multiple patterns to evaluate how comprehensively our approach can express syntactic transformations compared to a production rewrite tool.

Tool comparison. Table 3 shows results: Comby produces identical or similar functionality for detecting and rewriting the investigated patterns. The `2to3` tool (part of `cpython`) assists in refactoring Python 2 code to Python 3. Comby's output is identical to `2to3`'s. Comby also produces the same output for two patterns implemented by `Scalafmt`, a Scala formatter rewrite support tool. Comby's output is identical to `erl-tidy`, an Erlang refactoring tool, except for one match case of the `map` pattern. The output differs because `erl-tidy` creates unneeded fresh named variables in the output. Comby detects the same syntax issues found by a subset of linter checks implemented for Go, Dart, Elm, and Rust across respective projects and linter tests.

Implementation effort/size. We manually reviewed code in existing tools to determine the implementation size (**Impl.**), a proxy for implementation effort. Checks were generally implemented at the function level, or within a larger function. We manually isolated check functionality within these functions, stripped newlines, and counted the lines of code. Generally, the implementation burden is far lower with our declarative match and rewrite templates.

Speed. All Comby transformations complete in under 400ms per file (Python's `next` and `range` patterns run on multiple files); thus, syntactic transformation is responsive enough to integrate with real-time development.

Parsing robustness. Because existing tools require ASTs, some do not work if the target file cannot be correctly parsed. For example, `clang-tidy` may require header files to parse C++ files. We found that `clang-tidy` fails to parse C++ files in GCC-compiled projects (e.g., PHP) where the required header files contain GCC extensions like `asm goto`. `elm-lint` may also fail to parse files due to irregular comments. In contrast, our approach was robust to syntactic irregularities.

Clojure. To evaluate expressiveness more deeply, we implemented multiple patterns based on a dedicated syntax rewriter for Clojure. `Kibit` replaces Clojure code patterns with more idiomatic or terse versions (cf. `simpl-check` in Table 1). `Kibit` is an apt tool for comparison because it uses logic programming to specify rewrite patterns that are syntactically close to Clojure syntax. For this reason, we were able to easily translate `Kibit` rewrite patterns into our match and rewrite template format (some are identical, modulo hole syntax). We implemented 81 templates for simplifying arithmetic, equality, collection, and control flow syntax for Clojure. Comby's output is identical to `Kibit`'s on `Kibit`'s tests for all of these patterns. We were unable to implement one pattern that `Kibit` supports, which relies on detecting class names for static methods versus object methods. `Kibit`'s logic programming pattern representation is comparatively

[15] https://caml.inria.fr/pub/docs/manual-ocaml/spacetime.html

Table 3. Tool Comparison. We compare against 9 existing checker and rewrite tools for multiple **Patterns**. **(R)** in the **Tool** column means that the tool offers rewrite capabilities; remaining tools only emit warnings. **Target Kind** describes the transformation target: either a selection of code extracted from a popular open source project, or tests that are part of the tool. In general, the implementation burden for rewriting in our approach (**Us, Impl.**) is less compared to that of language-specific implementations for the set of syntactic properties considered. The exception is `kibit`, which offers comparable terseness due to a logic programming representation for rewrite rules.

Lang	**Tool**	**Pattern**	**LOC**	**Matches**	**Time (ms)**		**Impl. (LOC)**		**Target**
					Tool	**Us**	**Tool**	**Us**	**Kind**
Python	2to3 **(R)**	next	6,908	11	911	486	76	4	osf.io
		range	13,057	14	1,309	895	53	8	
Erlang	erl-tidy **(R)**	append	4,891	2	1,426	277	8	2	ejabberd
		map	11,389	16	1,633	389	20	2	
Elm	elm-lint	map	64	1	1,712	124	67	8	graphql
		dot-access	153	2	2,727	130	51	2	take-home
Go	gosimple	nil-check	236	1	1,315	130	39	5	helm
		str-contains	467	1	2,409	132	65	2	gitea
Dart	linter	where-type	43	3	1,718	122	74	9	tests
		slow-length	518	1	1,808	134	89	2	sqflite
Rust	clippy	redun-pat	71	4	115	137	74	11	tests
		short-field	74	5	148	141	38	5	
Scala	Scalafmt **(R)**	curly-fors	32	2	1,312	154	80	6	tests
		parens-guard	41	2	1,451	270	32	2	
C++	clang-tidy **(R)**	no-continue	2,143	1	325	155	15	2	radare2
Clojure	kibit **(R)**	various (81)	182	97	6,840	164	215	240	tests

short and closely matches our declarative syntax (many specifications fit on one line, which accounts for a lower number of LOC in Table 3).

5.3 Discussion

Known limitations. Our approach enables lightweight, purely syntactic rewrite patterns. We emphasize simple and declarative specification, and have shown that patterns are particularly effective at, e.g., removing redundant code. However, we do not currently incorporate type information, which limits our ability to implement richer rewrite patterns found in tools like `clang-tidy`. This is problematic for languages with syntactic ambiguity with respect to program manipulation [46], as we noted for the Go `nil-check` pattern. However, this problem is not unique to our approach: linters can suffer false positives for the same reason. Further, redundant syntax, like redundantly parenthesized expressions, may require additional match templates (or rules) to account for syntactic variation.

For transformations that change nesting levels or whitespace, code may need reindentation. We treat whitespace liberally by default; enabling finer-grained control over whitespace complicates matching on non-whitespace syntax. We found that patterns sometimes do preserve desired whitespace, but in general a post-process formatting tool is effective for (re)applying whitespace style.

In our experience, declarative templates are well suited to patterns that describe a syntactic change that does one thing well. More complex transformations may be difficult to express in a single rewrite pattern, and could require running transformations in succession (consistent with work using rewrite stages for transformation [49, 50, 53]). There may otherwise be syntactic transformations that are overly difficult or impossible to specify declaratively. Such patterns are hard for us to qualify generally; we mitigate by having shown real world value using existing patterns.

We have not yet implemented support for layout-sensitive or context-sensitive language properties, but parser combinators are sufficiently powerful to do so. We thus believe our approach generalizes.

Experience compared to existing tools. We noticed the following undesirable behavior in existing tools: (1) failure to handle files due to parsing errors or inadequate parsing support (2) false negatives where tools miss rewriting opportunities (3) hanging on certain files. From a usability perspective, we found Comby to be more robust: we never encountered runtime parsing errors, were able to transform code that other tools could not parse, and performed transformations that other tools missed.

6 Related Work

Parsing expression grammars (PEGs) [28] were developed to address shortcomings in regex matching, and can match non-regular constructs (e.g., balanced parentheses). PEGs describe a top-down parser for a language using the ordered choice operator. Existing tools implement PEGs to enable non-regular matching (`Rosie` [9], `instaparse` [5]) and modular syntax extension (`Rats!` [30]). Match specifications for these tool resemble grammar definitions and include, e.g., explicit metasyntax for parsing with ordered choice. Our match approach can be seen as automatically generating PEG-like parsers from declarative match templates, where templates lack metasyntax besides holes, and abstract from the underlying grammar interpretation. TXL [23] is a multi-language transformation tool where users specify both a grammar specification and transformation rules in the TXL language. Learning the TXL language (where, e.g., the grammar nonterminals are typically referenced or defined in the rewrite specification) is expensive compared to our approach. Similarly, language workbench tools [26] like Rascal [38] expose metaprogramming languages with grammar metasyntax to enable transformation. The `srcML` [21, 22, 44] platform features multi-language transformation, but uses an intermediate XML representation requiring programming to manipulate syntax, rather than declarative specification. Cobra [32, 33] is a lightweight approach for implementing syntactic checks (and optional transformation) primarily through programming and a query language.

Compared generally to these approaches, our design disentangles match and rewrite specification from grammar definition through a predefined (but configurable) DEL grammar operationalized as a parser generator. Template specification is declarative as opposed to programmatic. Matching is a function of parser generation that acts directly on syntax, without intermediate tree representations. This notably removes the complexity of translating a declarative concrete syntax to an underlying abstract syntax.

Cubix is a multi-language transformation tool using compositional data types and is a heavyweight solution intended for complex rewrites (like lifting variable declarations) with strong, type-based guarantees [39]. We prioritize lightweight transformation (templates take seconds or minutes to write), whereas Cubix is not intended for lay programmers.

Coccinelle [40] is notable for using a declarative approach with similar specifications to ours to transform syntax, although it introduces additional metasyntax and uses a fixed grammar that only parses C code. Refaster [56] introduces a template-based approach restricted to Java, but can additionally draw on type information. D-expressions derive from Lisp macros [25] and use a skeleton syntax tree to support a simple core grammar definition for the Dylan language [14], later adapted for Java [15]. In our work, an extensible skeleton parser encodes handling syntactic variations to enable macro-like functionality for many languages.

Parser combinators [34] are central for enabling our modular transformation approach for flexibly handling multiple language syntax. In this sense, we relate to multi-language static checking using parser combinators [18]. Our focus, however, is on transformation. The relation between structural tree matching and parsing is observed by [31]; in this sense, our approach operationalizes configurable tree-like matching as a function of modular, configurable parser combinator generation via declarative specification. Okasaki observes the phenomenon of parsers-producing-parsers [48] and notes how a "prelude" parser can define another parser for parsing the remainder of a program. Templates in our approach correspond to a kind of extensible "prelude" parser; the resulting parser runs on the complete program and records only syntactic fragments of interest for rewriting.

Our approach extends Dyck languages (being simple yet fundamental to CFL properties [27, 29]) as a foundation for representing contemporary language syntax and structure. Visibly pushdown languages [13] also build on the essential CFL properties of Dyck languages toward term rewriting [19] and program analysis [20].

Refal [7, 53] is a pattern matching language and tool for supercompilation. Refal's operation and application emphasize our own: matching and rewriting patterns is valuable for removing redundancy and improving performance. Our tools presents further potential for large-scale cleanup and code changes, such as those performed by Google's Rosie [42].

7 Conclusion

We presented a lightweight approach to syntactic transformation for multiple languages. In general, defining a universal AST for transformation is problematic: concrete definitions must be referred to, modified, or extended to support multiple languages. Our approach skirts this issue by focusing on universal CFL properties of languages (i.e., nested constructs) rather than defining a concrete representation. We lift the matching problem to declaratively generate parsers that detect syntax of interest directly. We use PPCs to modularize the behavior of parser generation to (a) preserve essential CFL properties while (b) allowing hooks to customize the parsing of language-specific syntax. Matched syntax is recorded during parsing and transformed contextually and hierarchically based on rewrite templates.

We evaluated our approach by implementing and applying 30 rewrite patterns to the 100 most popular GitHub projects for 12 languages totaling 185 million lines of code. Over 50 changes have been merged into upstream repositories. We also showed that our declarative approach requires significantly less effort to specify rewrite patterns compared to language-specific tools used in practice.

Acknowledgments

This work is partially supported under National Science Foundation Grant No. CCF-1750116 and CCF-1563797. All statements are those of the authors, and do not necessarily reflect the views of the funding agency.

References

[1] Online. Clang's refactoring engine. https://clang.llvm.org/docs/RefactoringEngine.html. Accessed 16 April 2019.

[2] Online. decaffeinate. https://github.com/decaffeinate/decaffeinate. Accessed 16 April 2019.

[3] Online. Facebook's Codemod for large-scale codebase refactors. https://github.com/facebook/codemod. Accessed 16 April 2019.

[4] Online. gofmt Go code formatter. https://golang.org/cmd/gofmt. Accessed 16 April 2019.

[5] Online. Instaparse. https://github.com/Engelberg/instaparse. Accessed 16 April 2019.

[6] Online. Pylint: a Python linter. https://media.readthedocs.org/pdf/pylint/latest/pylint.pdf. Accessed 16 April 2019.

[7] Online. Refal: REcursive Functions Algorithmic Language. http://www.refal.net/~arklimov/refal6/. Accessed 16 April 2019.

[8] Online. Rewrite Transforms in Facebook's Hack. https://github.com/facebook/hhvm/blob/4abea3/hphp/hack/src/parser/coroutine/coroutine_state_machine_generator.ml#L411-L431. Accessed 16 April 2019.

[9] Online. Rosie Pattern Language. https://developer.ibm.com/code/open/projects/rosie-pattern-language/. Accessed 16 April 2019.

[10] Online. staticcheck for Go. http://staticcheck.io/docs/#overview. Accessed 16 April 2019.

[11] Online. The Effective Dart style guide. https://www.dartlang.org/guides/language/effective-dart/style. Accessed 16 April 2019.

[12] Online. TSLint: a linter for TypeScript. https://palantir.github.io/tslint/rules/. Accessed 16 April 2019.

[13] Rajeev Alur and P. Madhusudan. 2004. Visibly pushdown languages. In *Symposium on Theory of Computing*. 202–211.

[14] Jonathan Bachrach and Keith Playford. 1999. D-expressions: Lisp power, Dylan style. (1999).

[15] Jonathan Bachrach and Keith Playford. 2001. The Java Syntactic Extender. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA '01)*. 31–42.

[16] Jean Berstel and Luc Boasson. 2002. Balanced Grammars and Their Languages. In *Formal and Natural Computing*. 3–25.

[17] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.* 72, 1-2 (2008), 52–70.

[18] Fraser Brown, Andres Nötzli, and Dawson R. Engler. 2016. How to Build Static Checking Systems Using Orders of Magnitude Less Code. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. 143–157.

[19] Jacques Chabin and Pierre Réty. 2007. Visibly Pushdown Languages and Term Rewriting. In *Frontiers of Combining Systems*. 252–266.

[20] Swarat Chaudhuri and Rajeev Alur. 2007. Instrumenting C Programs with Nested Word Monitors. In *International SPIN Symposium on Model Checking of Software*. 279–283.

[21] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 2011. Lightweight Transformation and Fact Extraction with the srcML Toolkit. In *International Working Conference on Source Code Analysis and Manipulation (SCAM '11)*. 173–184.

[22] Michael L. Collard and Jonathan I. Maletic. 2016. srcML 1.0: Explore, Analyze, and Manipulate Source Code. In *International Conference on Software Maintenance and Evolution (ICSME '16)*. 649.

[23] James R. Cordy. 2006. The TXL source transformation language. *Sci. Comput. Program.* 61, 3 (2006), 190–210.

[24] Walter J Doherty and Arvind J Thadhani. 1982. The economic value of rapid response time. *IBM Report* (1982).

[25] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. 1992. Syntactic Abstraction in Scheme. *Lisp and Symbolic Computation* 5, 4 (1992), 295–326.

[26] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In *Software Language Engineering (SLE '13)*. 197–217.

[27] Eldar Fischer, Frédéric Magniez, and Tatiana A. Starikovskaya. 2018. Improved bounds for testing Dyck languages. In *Symposium on Discrete Algorithms (SODA '18)*. 1529–1544.

[28] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Symposium on Principles of Programming Languages (POPL '04)*. 111–122.

[29] Seymour Ginsburg and Michael A. Harrison. 1967. Bracketed Context-Free Languages. *J. Comput. Syst. Sci.* 1, 1 (1967), 1–23.

[30] Robert Grimm. 2006. Better extensibility through modular syntax. In *Programming Language Design and Implementation (PLDI '06)*. 38–51.

[31] Christoph M Hoffmann and Michael J O'Donnell. 1982. Pattern matching in trees. *J. ACM* 29, 1 (1982), 68–95.

[32] Gerard J. Holzmann. 2017. Cobra: a light-weight tool for static and dynamic program analysis. *ISSE* 13, 1 (2017), 35–49.

[33] Gerard J. Holzmann. 2017. Cobra: fast structural code checking (keynote). In *International SPIN Symposium on Model Checking of Software*. 1–8.

[34] Graham Hutton. 1992. Higher-Order Functions for Parsing. *Journal of Functional Programmming* 2, 3 (1992), 323–343.

[35] Graham Hutton and Erik Meijer. 1996. Monadic Parser Combinators.

[36] Lennart C. L. Kats, Karl Trygve Kalleberg, and Eelco Visser. 2010. Interactive Disambiguation of Meta Programs with Concrete Object Syntax. In *Software Language Engineering*. 327–336.

[37] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-written Patches. In *International Conference on Software Engineering (ICSE '13)*. 802–811.

[38] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *International Working Conference on Source Code Analysis and Manipulation (SCAM '09)*. 168–177.

[39] James Koppel, Varot Premtoon, and Armando Solar-Lezama. 2018. One tool, many languages: language-parametric transformation with incremental parametric syntax. *PACMPL* 2, OOPSLA (2018), 122:1–122:28.

[40] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 Years of Automated Evolution in the Linux Kernel. In *USENIX Annual Technical Conference*. 601–614.

[41] Daan Leijen and Erik Meijer. 2001. *Parsec: Direct style monadic parser combinators for the real world.* Technical Report. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht.

[42] Josh Levenberg. 2016. Why Google stores billions of lines of code in a single repository. *Commun. ACM* 59, 7 (2016), 78–87.

[43] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Principles of Programming Languages (POPL '16)*. 298–31.

[44] Jonathan I. Maletic and Michael L. Collard. 2015. Exploration, Analysis, and Manipulation of Source Code Using srcML. In *International Conference on Software Engineering (ICSE '15)*. 951–952.

[45] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Trans. Software Eng.* 44, 5 (2018), 453–469.

[46] Eduardus A. T. Merks, J. Michael Dyck, and Robert D. Cameron. 1992. Language Design For Program Manipulation. *IEEE Trans. Software Eng.* 18, 1 (1992), 19–32.

[47] Jakob Nielsen. 1994. *Usability engineering*. Elsevier.

[48] Chris Okasaki. 1998. Functional Pearl: Even Higher-Order Functions for Parsing. *J. Funct. Program.* 8, 2 (1998), 195–199.

[49] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Quoted staged rewriting: a practical approach to library-defined optimizations. In *International Conference on Generative Programming: Concepts and Experiences (GPCE '17)*. 131–145.

[50] Sukyoung Ryu. 2016. Scalable framework for parsing: from Fortress to JavaScript. *Softw., Pract. Exper.* 46, 9 (2016), 1219–1238.

[51] Konstantinos Sagonas and Thanassis Avgerinos. 2009. Automatic refactoring of Erlang programs. In *International Conference on Principles and Practice of Declarative Programming (PPDP '09)*. 13–24.

[52] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *International Conference on Mining Software Repositories (MSR '16)*. 512–515.

[53] Valentin F. Turchin. 1986. The Concept of a Supercompiler. *ACM Trans. Program. Lang. Syst.* 8, 3 (1986), 292–325.

[54] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *International Conference on Software Engineering (ICSE '18)*. 151–162.

[55] Eelco Visser. 2002. Meta-programming with Concrete Object Syntax. In *Generative Programming and Component Engineering (GPCE '02)*. 299–315.

[56] Louis Wasserman. 2013. Scalable, example-based refactorings with refaster. In *Workshop on Refactoring Tools (WRT@SPLASH '13)*. 25–28.

[57] Daniel Weise and Roger F. Crew. 1993. Programmable Syntax Macros. In *Programming Language Design and Implementation (PLDI '93)*. 156–165.