Languages the Racket Way
2016 Language Workbench Challenge

Daniel Feltey
Northwestern University
daniel.feltey@eecs.northwestern.edu

Spencer P. Florence
Northwestern University
spencer.florence@eecs.northwestern.edu

Tim Knutson
University of Utah
tkkemo@gmail.com

Vincent St-Amour
Northwestern University
stamourv@eecs.northwestern.edu

Ryan Culpepper
Northeastern University
ryanc@ccs.neu.edu

Matthew Flatt
University of Utah
mflatt@cs.utah.edu

Robert Bruce Findler
Northwestern University
roppy@eecs.northwestern.edu

Matthias Felleisen
Northeastern University
matthias@ccs.neu.edu

Abstract
Racket espouses the view that full-fledged problem solving almost always calls for language design. In support of this view, it implements a notion of linguistic reuse, which allows programmers to rapidly develop and deploy new programming languages. Together with DrRacket, its IDE, the Racket ecosystem thus makes up a true language workbench.

This paper demonstrates Racket’s capabilities with an implementation of the 2016 Language Workbench Challenge. Building on a concise implementation of MiniJava, it shows how it is easy it is to add new notation, constrain constructs, and create IDE tools.

1. The Racket Manifesto in a Nutshell
Racket really is a programming-language programming language (Felleisen et al. 2015). It provides both unique linguistic support (Flatt and PLT 2010) for the rapid prototyping of languages as well as an ecosystem with unmatched elements (Findler et al. 2002). As such, it is a first step toward the idea of a language workbench (Erdweg et al. 2015) as originally imagined by Bill Scherlis and Dana Scott in the early 1980s (Scherlis and Scott 1983).

After explaining the linguistic elements of language development in Racket (section 2), this paper illustrates Racket as a language workbench starting with (section 3) an implementation of MiniJava (Roberts 2001).

Based on the MiniJava implementation, we present the results of tackling three benchmark problems from the 2016 language workbench challenge. First, we demonstrate a solution to the Tabular Notation problem in the Editing category by adding notation to MiniJava so that programmers can express finite-state machines as Unicode-based tables (section 4). Second, we solve the Beyond-Grammar Restrictions benchmark from the Evolution and Reuse category by showing how to constrain a break construct in MiniJava so that is is valid only within the scope of while (section 5). Third, we explain how to connect an implemented language, such as MiniJava, with DrRacket; specifically, we show how to add tools for program Restructuring, consistent with the Editing benchmark category (section 6).

Our MiniJava implementation with all of the extensions is available as a Racket package.¹ Using the current version of Racket², run

```
raco pkg install lwc2016
```

to install it. To view a selection of sample MiniJava programs and experiment with the language extensions, select the "Open Require Path..." entry under the "File" menu in DrRacket and type lwc2016/examples/programs/.

¹http://pkgs.racket-lang.org
²https://download.racket-lang.org/
2. The Racket Language Workbench

Racket promotes a language-oriented view of problem solving. Using Racket, programmers can quickly build languages to solve each aspect of a programming problem on its own linguistic terms. As such, Racket programs are composed of a number of modules, each implemented in the language that is best suited for the module’s task. To represent this module—language mapping, the first line of each module specifies the language in which it is written. The Racket ecosystem relies on this mapping for linguistic dispatch: a mechanism by which a language publishes its implementation and linguistic meta-information—syntax coloring, indentation, static analysis, etc.—to customize execution and user experience for programs written in that language.

To support this language-development idiom, Racket fully embraces the idea of linguistic reuse (Krishnamurthi 2000). According to this view, the development of a new language consists of adding, subtracting, and re-interpreting constructs and run-time facilities from a base language. Even the installation of a new language takes place within the Racket ecosystem: a language is itself a Racket component that provides certain services, and each module’s language specification (i.e., the initial #lang line) simply refers to another module that implements the language (Flatt 2002).

A language implementation can be as simple as a plain Racket module that exports specific constructs. A more sophisticated language variant consists of modules that implement a reader—a lexer and parser for any imaginable Unicode-based notation—and a semantics module. By using specific tools and following certain conventions, programmers can produce languages that work well together.

More broadly, a programmer can implement a Racket language in several different ways:

- as a plain interpreter that repeatedly traverses the code of a client program;
- as a compiler that maps a client program to a target language, either within or outside of the Racket ecosystem;
- as a Redex (Felleisen et al. 2010) reduction semantics; or
- as a linguistic derivative of an existing Racket language.

Racket strongly encourages this last approach, because it delivers results more quickly while remaining as general as any of the others. But, all of these approaches are useful in certain situations, and on occasion, as in the case of MiniJava, an implementation may borrow elements from several approaches.

Deriving one language from another means creating a translation of new linguistic constructs into those of the base (or “parent”) language and a run-time library. Such derivations inherit other elements of the run-time system (the VM, the JIT compiler, the garbage collector, etc.). We consider translation the critical part of language derivation.

Technically, the derivation works as follows. A language module may export a subset of the constructs and functions of some base language, which implicitly subtracts features from that language; it may export additional features and functions, which adds new capabilities; and it may re-interpret existing features, say, function applications or conditionals. The re-interpretation is accomplished by defining a new construct or function in a module and exporting it under the name of a feature that already exists in the base language.

A Racket programmer uses the syntax object system to create new linguistic constructs. This system is a descendant of Scheme and Lisp’s hygienic macro system (Clinger and Rees 1991; Hart 1963; Kohlbecker et al. 1986). The system represents syntactic terms via syntax objects, which include properties of the source syntax as well as those specified by a language implementor (Dybvig et al. 1992).

Like the Lisp macro system of lore, Racket’s syntax object system allows the specification of rewriting rules on syntax objects. An elaborator uses these rules to translate a module from any language into Racket core syntax on an incremental basis. Unlike Lisp or Scheme macros, Racket’s rewriting rules provide sophisticated services. For example, they automatically propagate source information so that they can report error in terms of the original source notation. Similarly, the rules almost automatically enforce context-free constraints so that error messages use the concepts of the surface language (Culpepper and Felleisen 2010). Lastly, these rewriting rules can also articulate transformations on complete modules and on existing linguistic constructs—giving them the expressive power to track context-sensitive constraints and to assign new meaning to old words.

3. MiniJava via Racket

The Racket language proper consists of a module-oriented, untyped functional language in the spirit of Lisp and Scheme. In contrast, MiniJava (section 3.1) is a class-based, object-oriented programming language in the spirit of Java. Using Racket’s syntax system, it is nevertheless possible to create a realistic implementation of MiniJava with a relatively small effort—thanks to linguistic reuse.

This section provides an overview of the MiniJava implementation (section 3.2) followed by a presentation of the individual building blocks. Where necessary, the section also explains Racket features on a technical basis.

3.1 MiniJava

Figure 1 displays a MiniJava program. Like any Java program, it consists of a series of class definitions, one of them designated as the “main” class. Classes have public methods and fields. Each of these comes with a type signature, where types are the names of classes plus the usual primitive types (e.g., int). The body of a method may use the familiar statements of an imperative language: assignments, conditionals, and loops. MiniJava expressions are also the familiar ones.
class Main {
   public static void main(String [] args) {
      System.out.println((new Runner()).run(10));
   }
}

class Runner {
   Parity check;
   public int run(int n) {
      int current;
      check = new Parity();
      current = 0;
      while (current < n) {
         if (check.is_even(current)) {
            System.out.println(current);
         } else {
            current = current + 1;
         }
      }
      return 0;
   }
}

class Parity {
   public boolean is_odd(int n) {
      return (!(n == 0)) && this.is_even(n - 1);
   }
   public boolean is_even(int n) {
      return (n == 0) || this.is_odd(n - 1);
   }
}

Figure 1: A sample MiniJava program

MiniJava lacks many of Java’s sophisticated features: abstract classes, interfaces, packages, etc. The goal is to help students in introductory courses, not to model the complexities of a real-world language.

3.2 From MiniJava to Core Racket: an Overview

Whereas a typical compiler’s front end parses a textual program into an abstract syntax tree (AST), an incremental Racket implementation separates this process into two distinct steps: the reader and the expander. The reader turns a textual program into a syntax object. The expander uses a suite of rewriting rules to elaborate this syntax object into Racket’s kernel syntax. This latter phase conceptually employs a tower of languages, and the elaboration gradually turns a program of one level into a language of the next lower level. In reality, the layers of this tower are not separated through sharp boundaries, and the intermediate programs may never exist in a pure form.

Figure 2 presents the overall pipeline of the MiniJava implementation in Racket. Step 1 turns the Java-like syntax into a syntax object, a combination of the symbolic source program and syntax properties; this object roughly corresponds to an abstract syntax tree. Step 2 elaborates this AST into a prefix variant of MiniJava in a conventional manner. The choice of type elaboration over checking allows the injection of type annotations that help implement efficient method calls (Flatt et al. 1998). The prefix variant of MiniJava is an untyped, parenthesized version of MiniJava.

Once a MiniJava program has been elaborated into prefix form, the regular expansion process takes over. Step 3 indicates how parenthesized MiniJava programs are rewritten into plain #lang racket constructs. This transformation is articulated with a (relatively small) suite of rewriting rules that map classes, method calls, and so on into functional Racket constructs.

As mentioned in the preceding section, the implementation of mini-java consists of a reader module, which implements steps 1 and 2, and a language module, which implements the syntactic translation of step 3. The former employs Racket’s lexing and parsing libraries. While lexing and parsing libraries require no explanation, the type elaboration needs some discussion (section 3.3). The language module consists of syntax rewriting rules and any functions in the target code that Racket does not provide already (section 3.4, but also section 3.5). Both of these modules are implemented in the ordinary racket language.

Finally, step 4 indicates that the existing Racket language elaborates the program into core Racket. The latter is known as #%kernel. This step is well-established and does not deserve any attention.

In contrast, the integration of #lang mini-java with DrRacket deserves a thorough explanation (section 3.6). Essentially, the pipeline of figure 2 preserves essential properties across the various transformations. In turn, DrRacket implements a protocol between the expanded syntax and its editor, which plug-in tools can exploit to implement an ecosystem for a language such as MiniJava.
3.3 \texttt{#lang mini-java}: Parsing and Type Elaboration

Racket's \texttt{#lang} reader mechanism wraps the entire content of a module (everything below the language specification) into a single syntactic object, a \texttt{#%module-begin} form. A language-implementation module may therefore opt in to linguistic dispatch by exporting its own \texttt{#%module-begin} macro and thus take over the interpretation of an entire module at once. The result of a \texttt{#%module-begin} expansion must be a \texttt{#%module-begin} form in some other language, usually the base language.

Figure 3 shows how mini-java exploits this mechanism. The module defines a “module begin” construct as \texttt{mj-module-begin}. The module's export specification says that \texttt{mj-module-begin} becomes the \texttt{#%module-begin} form for the mini-java language. The implementation of \texttt{mj-module-begin} expands to a \texttt{#%module-begin} that (through an import not shown in the figure) implements our prefix variant of MiniJava, but that \texttt{#%module-begin} is hidden from a module that is implemented in the surface mini-java language, which instead sees \texttt{mj-module-begin}.

Before expanding to an underlying \texttt{#%module-begin} form, \texttt{mj-module-begin} hands the list of class definitions (ASTs) to the auxiliary \texttt{typecheck-program} function. This syntax-level function implements an ordinary recursive-descent type elaboration mechanism on a MiniJava variant using infix operations, and the result is a MiniJava program using prefix forms.

The \texttt{#%module-begin} for prefix MiniJava forms turns out to be Racket’s usual \texttt{#%module-begin} form, so Racket’s usual macro expansion takes over the rest of the compilation pipeline. By linguistic reuse, MiniJava variables become Racket variables, MiniJava conditionals become Racket conditionals, and only forms without Racket analogs synthesize substantially new code.

3.4 \texttt{#lang mini-java}: Language Constructs

One form without a Racket precedent is MiniJava’s \texttt{while} construct. A use of the construct appears on line 7 in the left half of figure 5. The corresponding code in the right column of the same figure shows the code that is synthesized by expansion. Racket’s expander uses the relatively simple rewriting rule from figure 4 to effect this translation.

```
(define-syntx (while stx)
  (syntax-parse stx
    [(while test:expr body ...)
     #'(letrec ((loop (λ ()
                      (when test
                        body ...)
                      (loop))))))])
```

Figure 4: Definition of while in prefix MiniJava

The implementation of MiniJava’s \texttt{while} uses Racket’s \texttt{define-syntx} form to bind \texttt{while} to a \texttt{transformer function}. The latter implements the compilation step for \texttt{while} loops; the definition informs the macro expander that whenever a \texttt{while} AST node shows up, it must invoke the transformer on the node.

The definition of the \texttt{while} transformer function uses \texttt{syntax-parse}, a powerful pattern matcher for defining syntactic extensions (Culpepper and Felleisen 2010). This macro contains a single pattern,

```
(while test:expr body ...)
```

which indicates that it expects to see \texttt{while} followed by a \texttt{test} expression and any number of \texttt{body} elements. The constraint \texttt{:expr} forces \texttt{test} to be an expression, while \texttt{body} could be either a definition or an expression. If the expression constraint is violated, \texttt{syntax-parse} automatically synthesizes an error message in terms of source notation and source concepts.

The pattern matcher binds \texttt{test} and \texttt{body} for use within the code-generation template. Such a template is specified with \texttt{#'\ldots} or

```
(quasisyntax/loc ...)
```

This form resembles the \texttt{quasiquote} form that Racket inherits from Lisp. They differ in that \texttt{quasisyntax/loc} produces a syntax object instead of an S-expression and that it
(class Runner
  (define-field check)
  (define-method run (n)
    (if (send Parity check is_even current)
        (compound)
        (System.out.println current))
    (= current 0))
  (while (< current n)
    (if (send Parity is_even current)
        (compound)
        (= current (+ current 1))))
  )
)

(define Parity:runtime-method-table
  (vector
    (define current #f)
    (vector-set! this 1 (Parity:constructor))
    (set! current 0)
    (letrec ([loop
      (lambda ()
        (when (< current n)
          (let* ([receiver ([vector-ref this 1]])
            [meth-table ([vector-ref receiver meth-table 1])]
            [meth receiver current])
            (displayln current))
            (set! receiver current
              (loop)))]
          (loop)
          0)))))))
)

(define (Runner:constructor)
  (vector Runner:runtime-method-table #f))
)

(define-syntax Runner
  (syntax-rules ()
    [#f
      (make-immutable-free-id-table
        (list (cons #~'run 0))
        #~'Runner:runtime-method-table
        #~'Runner:constructor
        1))
    []
    ([meth-table meth receiver current]
      (displayln current)
      (set! receiver current
        (loop)))]
    [loop
case
    [1]
    [2]
    [3]
    [4]
    [5]
    [6]
    [7]
    [8]
    [9]
    [10]
    [11]
    [12]
    [13]
    [14]
    [15]
    [16]
    [17]
    [18]
    [19]
    [20]
    [21]
    [22]
    [23]
    [24]
    [25]
    [26]
    [27]
    [28]
    [29]
    [30]
    [31]
    [32]
    [33]
    [34]
    [35]
    [36]
    [37]
    [38]
    [39]
    [40]
    [41]
    [42]
    [43]
    [44]
    [45]
    [46]
    [47]
    [48]
    [49]
    [50]
    [51]
    [52]
    [53]
    [54]
    [55]
    [56]
    [57]
    [58]
    [59]
    [60]
    [61]
    [62]
    [63]
    [64]
    [65]
    [66]
    [67]
    [68]
    [69]
    [70]
    [71]
    [72]
    [73]
    [74]
    [75]
    [76]
    [77]
    [78]
    [79]
    [80]
    [81]
    [82]
    [83]
    [84]
    [85]
    [86]
    [87]
    [88]
    [89]
    [90]
    [91]
    [92]
    [93]
    [94]
    [95]
    [96]
    [97]
    [98]
    [99]
    [100]
    [101]
    [102]
    [103]
    [104]
    [105]
    [106]
    [107]
    [108]
    [109]
    [110]
    [111]
    [112]
    [113]
    [114]
    [115]
    [116]
    [117]
    [118]
    [119]
    [120]
    [121]
    [122]
    [123]
    [124]
    [125]
    [126]
    [127]
    [128]
    [129]
    [130]
    [131]
    [132]
    [133]
    [134]
    [135]
    [136]
    [137]
    [138]
    [139]
    [140]
    [141]
    [142]
    [143]
    [144]
    [145]
    [146]
    [147]
    [148]
    [149]
    [150]
    [151]
  )))
)

(define-syntax Parity
  (syntax-rules ()
    [#f
      (make-immutable-free-id-table
        (list (cons #~'is_odd 0) (cons #~'is_even 1))
        #~'Parity:runtime-method-table
        #~'Parity:constructor
        0))
    []
    ([meth-table meth receiver current]
      (displayln current)
      (set! receiver current
        (loop)))]
    [loop
case
    [1]
    [2]
    [3]
    [4]
    [5]
    [6]
    [7]
    [8]
    [9]
    [10]
    [11]
    [12]
    [13]
    [14]
    [15]
    [16]
    [17]
    [18]
    [19]
    [20]
    [21]
    [22]
    [23]
    [24]
    [25]
    [26]
    [27]
    [28]
    [29]
    [30]
    [31]
    [32]
    [33]
    [34]
    [35]
    [36]
    [37]
    [38]
    [39]
    [40]
    [41]
    [42]
    [43]
    [44]
    [45]
    [46]
    [47]
    [48]
    [49]
    [50]
    [51]
    [52]
    [53]
    [54]
    [55]
    [56]
    [57]
    [58]
    [59]
    [60]
    [61]
    [62]
    [63]
    [64]
    [65]
    [66]
    [67]
    [68]
    [69]
    [70]
    [71]
    [72]
    [73]
    [74]
    [75]
    [76]
    [77]
    [78]
    [79]
    [80]
    [81]
    [82]
    [83]
    [84]
    [85]
    [86]
    [87]
    [88]
    [89]
    [90]
    [91]
    [92]
    [93]
    [94]
    [95]
    [96]
    [97]
    [98]
    [99]
    [100]
    [101]
    [102]
    [103]
    [104]
    [105]
    [106]
    [107]
    [108]
    [109]
    [110]
    [111]
    [112]
    [113]
    [114]
    [115]
    [116]
    [117]
    [118]
    [119]
    [120]
    [121]
    [122]
    [123]
    [124]
    [125]
    [126]
    [127]
    [128]
    [129]
    [130]
    [131]
    [132]
    [133]
    [134]
    [135]
    [136]
    [137]
    [138]
    [139]
    [140]
    [141]
    [142]
    [143]
    [144]
    [145]
    [146]
    [147]
    [148]
    [149]
    [150]
    [151]
  )))
)
supports automatic interpolation. With interpolation, macro expansion splices the values of pattern variables into the template—indeed, this is what makes the expression a template. The /loc part means that it also moves along source-location information.

Figure 5 shows the prefix version of the Runner class from figure 1 and its expansion into Racket. The while form on the left of figure 5 expands into the letrec expression, spanning lines 7 through 25 on the right. As the definition of while specifies, the loop’s condition becomes the guard to Racket’s when and the body is copied into the body of when before further expansion occurs.

The while macro exemplifies the importance of hygiene. The macro relies on hygienic expansion to prevent the letrec-bound variable, loop, from conflicting with uses of identically-named variables in the source syntax of while forms. Hygiene eases the job of macro writers, allowing them to write macros without worrying that their bindings will conflict with those that appear at macro use sites. In addition to supporting local variables with hygiene, Racket’s expansion process also ensures that free variables in the syntax object in the template (letrec, λ, and when in this case) refer to the bindings in scope at the definition of the macro, not at the use of the macro.

3.5 Inter-macro Communication

Isolated macro definitions do not suffice to transform MiniJava into Racket. Consider a new expression, which instantiates a class. The name of the class is not enough to create the object. The construction of an object also needs to know how many slots to allocate for fields and how to connect with the “vtabelle,” i.e., the method dispatch table of the class. In short, the Racket form that defines a MiniJava class must communicate with the Racket form that implements a new expression. Racket builds on ordinary lexical scope to provide a communication channel for distinct macros for just this purpose (Flatt et al. 2012).

To make this idea concrete, consider line 5 of figure 5. It shows how our implementation of MiniJava expands

(define-syntax (new stx)  
  (syntax-parse stx  
    [(new the-class:id)  
     #`(#[,,(static-class-info-constructor-id  
           (syntax-local-value #'the-class))])]))

Figure 6: The implementation of new in prefix MiniJava

[...]

form. It escapes from the template, constructs code at compile time via an arbitrary computation, and inserts that code into the template in lieu of itself. The question is what this computation affects and how it works.

To this end, we turn our attention to the expansion of the Parity class from our running example. It is shown in figure 5 at line 40. MiniJava forms are compiled away to ordinary Racket code, which uses vectors to represent objects and method tables. Critically, though, the class form for the Parity class expands to three definitions:

- the run-time method table shared by all Parity instances,

- the constructor that creates instances, and

- a syntax definition of compile-time information about the Parity class used to guide the expansion of other forms that refer to Parity.

The first definition, Parity:runtime-method-table (figure 5 line 40), is a vector storing the methods is_odd and is_even. The second one, Parity:constructor on line 53, is a function of no arguments for creating new instances of the Parity class. Because the Parity class has no fields, its instance vectors contain only a reference to the method table. The third definition, that of Parity on line 56, uses define-syntax, but in a rather surprising and unusual manner.

While the preceding section employs define-syntax to bind syntax-transforming functions to a name, here it binds a variable to an ordinary value, specifically, a record. The record constructor, static-class-info, is a function of five values. Its instances thus store compile-time information about the class: an identifier bound to the parent class information (or #f if there is no parent class); a table mapping method names to vector offsets; a syntax object referring to the run-time method table; a syntax object that points to the constructor; and a count of the fields in the class.

Now that the variable Parity is compile-time bound to information, other macro computations may retrieve this in-

3 Racket uses [...] and (...) interchangeably.

4 The syntax system specially recognizes that while is bound to a function.
formation. Technically, these macros must use the `syntax-local-value` procedure for this purpose. And that explains the inner expression in the template of `new` in figure 6. It is passed a syntax object that points to a class identifier, and `syntax-local-value` uses this identifier to retrieve the `static-class-info` record. Next, the function `static-class-info-constructor-id` is simply a field accessor (which would be written as a "constructor_id" suffix in infix-notation languages) that returns the value of the second-to-last field in the record. In this case, the value is the identifier `#'Parity:constructor`.

A reader may wonder why the `new` macro does not just synthesize the name of the object constructor directly from the name of the given class. Doing so looks natural, but it may interfere with intermediate re-bindings of the class name. Splicing syntax objects into the syntax object for the template guarantees hygienic code synthesis and thus automatically creates correctly scoped code.

The `send` macro, used for method calls, also makes use of this technique. Our MiniJava type checker annotates `send` forms with the name of the class of the receiver (which it computes during type checking). The `send` macro uses the static information bound to that class name to determine the correct index into the class’s method table. The bodies of the methods stored in `Parity:runtime-method-table`, in figure 5, show the results of this expansion. Line 44 of figure 5 shows a method call to `is_even` on the left and the `let*` expression it transforms into on the right. The 1 on line 45 is computed by using `syntax-local-value` to get the `static-class-info` (as with `new`) and then looking up `is_even` in the table on line 59.

This technique highlights the distinction between Racket’s run-time and compile-time phases. The `new` and `send` macros must call `syntax-local-value` at compile-time to access static class information. In general, this means that arbitrary, possibly even side-effecting, code may need to run at compile-time to expand a syntactic form. Racket addresses the issue of mingling compile-time code with run-time code through a phase distinction (Flatt 2002) that makes explicit the execution time of a piece of code.

### 3.6 DrRacket Integration

Syntax bindings are but one of the communication mechanisms available to Racket macros. Syntax properties provide another one. Recall that Racket’s syntax objects are data structures that include symbolic representations of program fragments as well as additional syntax properties. Macros may attach additional key-value pairs to syntax objects. Just as syntax bindings allow communication between distinct macros, syntax properties open a communication channel between different processing passes, including external tools (Tobin-Hochstadt et al. 2011).

DrRacket’s `check syntax` tool exploits this information to provide a better user experience when editing source code. It draws arrows between binding and bound occurrences of variables and it renders information in tooltips. To get the binding information and tooltip information, it consults syntax properties, as well as just using the underlying lexical information in the fully expanded program.

To illustrate the idea, our implementation of MiniJava attaches type information to syntax objects via syntax properties. In the example shown in figure 7, programmers can see the type information via tool tips (`Parity` in this case), and see binding information via arrows connecting identifiers. Some MiniJava variables become Racket variables in the fully expanded program, e.g., the parameter `n` of the `run` method. Others, like `check`, are fields and are compiled into vector references. Figure 5 shows this explicitly: the definition of the `check` field on line 2 is absent from the expanded program and the reference to `check` on line 9 compiles into the expression `(vector-ref this 1)` on line 11 of the expanded program. DrRacket’s `check syntax` tool nonetheless infers the correct binding structure for such variables and displays binding arrows accordingly, as figure 7 shows.

### 4. Notation: Tabular Notation

Adding notation to our implementation of MiniJava is quite straightforward. To illustrate this idea with a rather extreme example, we present here the result of tackling the problem of `Tabular Notation` from the Notation category. Specifically, we explain how to add tabular notation to MiniJava for specifying state machines via tables.
Figure 8 presents an example state machine. The syntax is purely textual, relying on Racket’s Unicode integration. A programmer produces the table outline with Unicode characters. In the context of MiniJava, the table represents a two-dimensional grid of transitions. Our implementation understands it as an alternative notation for a class. This synthesized class implements the corresponding state machine.

In figure 8, the first row in the table specifies the names of the states: `wait_0` and `wait_1`. The first column specifies the names of the input symbols: `zero` and `one`. The cells in the middle portion of the diagram specify what happens in the given state when the given symbol is received; each cell contains some arbitrary MiniJava code that runs for its effect, followed by the name of a new state to transition to. For example, when in the `wait_0` state, if the `zero` input comes, then the state machine will print out 0 and transition to the `wait_1` state. The state machine is reified as a MiniJava class, named by the single name in the upper-left cell: `Receiver`. The inputs to the state machine are reified as nullary methods on the class.

Finally, the second class in figure 8 shows a client of the state machine class. The `StateMachineRunner` class is a textual class definition. It refers to the state machine by name and creates an instance. Following that, it sends this state machine four inputs via method calls.

### 4.2 Implementation
Implementing this extension requires changes to each phase of our MiniJava implementation. First, we extend the lexer with a new class of tokens for 2-dimensional tables, which begin with the token `#2dstate-machine`. For this token, the lexer uses Racket’s existing 2d syntax parser to find the bounds of the table and break it into separate cells. The original parser then handles the contents of each cell. After parsing a 2d table, lexing resumes as usual.

After parsing, each table is a syntax object that corresponds to an invocation of the `#2dstate-machine` macro. We then extend the following phases to recognize these objects and pass them along. The `#2dstate-machine` macro compiles tables to classes. Each transition becomes a method which dispatches on the state, represented by an integer field.

### 4.3 Variants
One possible variant of this problem would be a tabular `if` or `switch` form, which would dispatch on two scrutinees, one selecting a row, the other a column.

### 4.4 Usability
DrRacket provides special support for inputting and editing this tabular syntax. It is difficult to use outside of DrRacket.

---

5 See DrRacket’s manual for more: http://docs.racket-lang.org/drracket/Keyboard_Shortcuts.html#(idx._...gentag._219._...lib._scribblings/drracket/drracket.scrib))
4.5 Impact

In addition to the aforementioned changes to the lexer, each phase is extended to recognize and pass along the state machine and the macro 2dstate-machine is added to compile the state machines.

4.6 Composability

The 2dstate-machine form coexists with the solutions to the other benchmarks, and would also compose with syntactic extensions to statements. Future extensions using the same tabular notation can reuse the majority of the implementation.

The 2dstate-machine macro cooperates with DrRacket’s check syntax tool to recognize state names as bindings.

4.7 Limitations

Convenient usage of the 2d syntax is limited to DrRacket. Racket’s 2d parser relies on read tables to extend the reader. MiniJava’s reader does not support read tables, which led to using one of the 2d parser’s internal APIs.

4.8 Uses and Examples

This form of tabular notation is used by Racket’s 2dcond and 2dmatch forms. These forms are used in the implementation of Redex (Felleisen et al. 2010).

4.9 Effort

Adding the 2d capabilities to the existing parser took 1-2 hours. The remaining work required about 140 lines of code and took 2-3 hours. Work on editing support for the tabular notation is ongoing.

5. Evolution and Reuse: Beyond-Grammar Restrictions

This section presents our solution to the Beyond-Grammar Restrictions benchmark problem in the Evolution category. Our solution extends MiniJava with a break keyword that is valid only within while loops. The implementation uses Racket’s syntax parameters to control the meaning of a binding depending on its context (Barzilay et al. 2011).

5.1 Assumptions

Our implementation technique for break relies on our use of macros to implement MiniJava constructs, as syntax parameters interact with the expansion process.

5.2 Implementation

Syntax parameters are syntax bindings whose expansion can be controlled by macros in their context. Specifically, macros (such as while) may adjust the meaning of syntax parameters (such as break) to make the latter behave as if they had been lexically bound by the former. This is akin to the way lexical scoping lets programmers adjust the meaning of an identifier in some context by introducing a shadowing binding for it.

Figure 9 shows the definition of break, as well as an extended version of while which cooperates with it. Originally, break is bound to a transformer which always raises a syntax error, which statically rules out uses of break outside of while. The definition of while adjusts the meaning of break (using syntax-parameterize) within its body to instead call an escape continuation and break out of the loop.

5.3 Variants

One variant of this benchmark problem would be to add Java’s super keyword to MiniJava. The super keyword, like break, is valid only in certain contexts—in methods of child classes, specifically. This MiniJava implementation always breaks the nearest enclosing loop, but break could accept a (literal) number as a argument to allow breaking of nested loops.

5.4 Usability

Our addition of the break keyword to MiniJava is syntactically allowed wherever a statement is valid. The syntax parameter mechanism, however, disallows uses of break outside the body of while loops, as one would expect.

5.5 Impact

Beyond the changes to parenthesized MiniJava discussed previously, implementing break requires changes to our MiniJava lexer, parser, and type checker. We extend the lexer to recognize the break keyword and produce a corresponding token. Similarly, we modify the parser to produce abstract syntax representing a use of break. The type checking rule added for break always succeeds and produces a use of the break syntax parameter in parenthesized MiniJava. Overall, these changes are small and independent of the rest of our MiniJava implementation.

5.6 Composability

Our addition of break to MiniJava integrates seamlessly with our solutions to the other two benchmark problems. This implementation would compose well with other instances of the Beyond-Grammar Restrictions problem.

5.7 Limitations

Implementing language restrictions using syntax parameters necessitates a language that compiles via a set of macros. An alternate implementation of MiniJava that directly produced fully-expanded Racket programs would not be able to use this strategy. Our use of syntax parameters relies on the nested shape of Racket’s syntax. The syntax-parameterize form only adjusts the meaning of forms nested inside of it.
(define-syntax-parameter break
  (λ (stx)
    (raise-syntax-error 'break "used outside of `while`" stx)))

(define-syntax (while stx)
  (syntax-parse stx
    [(while test:expr body ...) #`(let/ec local-break
        (syntax-parameterize ([(break (λ (stx) #'(local-break)))]
          (letrec ([(loop (λ ()
              (when test body ...)
                (loop)))]
              (loop))))))])

---

**Figure 9:** The break syntax parameter and its use in while

### 5.8 Uses and Examples

Racket uses syntax parameters in many of its core libraries to restrict certain syntactic forms to specific contexts. For example, Racket implements this using syntax parameters to ensure that it is valid only within Racket’s class form.

### 5.9 Effort

Extending our implementation of MiniJava to support the break keyword requires fewer than twenty lines of code. The implementation took approximately thirty minutes, but required familiarity with our MiniJava implementation and the use of syntax parameters.

### 6. Editing: Restructuring

From the Editing category we tackled the Restructuring benchmark problem. Specifically, we built a refactoring tool for MiniJava which restructures if statements by swapping the then and else branches and negating the condition.

#### 6.1 Assumptions

We assume that we can modify the implementation of MiniJava to expose additional information about conditionals.

#### 6.2 Implementation

Our restructuring tool relies on source location information for conditionals, which it gets from our MiniJava implementation. To do this, we extended MiniJava’s if macro to attach a syntax property mapping the key ‘refactor to a list containing the source position and span of each piece of the if statement: the condition, the then branch, and the else branch. Figure 10 shows this extension.

We implement our tool as a plugin for DrRacket (Findler et al. 2002) which processes a program’s fully-expanded syntax to find syntax objects with the ‘refactor syntax property attached. These locations are where the refactoring may apply. When a user applies the refactoring, the tool rewrites the conditional within the editor’s buffer.

#### 6.3 Variants

A variant on this refactoring would be to transform between expressions that use && and || using De Morgan’s laws.

In addition, it is worth noting that the implementation of our refactoring tool is not MiniJava-specific. By parameterizing the refactoring rule over negation syntax, the tool generalizes across languages. To support the refactoring, a language simply needs to attach the relevant syntax property to its conditional form. As a proof of concept, we also extend Racket’s if in this fashion.

#### 6.4 Usability

As figure 11 shows, the if refactoring is accessed by right-clicking inside of an if statement or using a keyboard shortcut. The refactoring tool does not affect the usability of DrRacket or the MiniJava language otherwise.
6.5 Impact
The refactoring tool proper is a standalone piece of code. Otherwise, the only changes to our MiniJava implementation are to the `if` macro, as discussed earlier.

6.6 Composability
Our restructuring tool composes automatically with other DrRacket plug-ins. Additionally, it does not interfere with our solutions to the other benchmark problems, except that refactoring within a state-machine may break the alignment of the tabular syntax.

6.7 Limitations
This refactoring makes sense only for conditionals with exactly two branches. In a language such as Java, which also has single-branch `if` statements, the tool would distinguish these cases to determine where the refactoring applies.

6.8 Uses and Examples
Several tools built on top of Racket and DrRacket use syntax properties to facilitate communication between tools and language implementations. Specific examples include DrRacket’s `check syntax` utility, Typed Racket’s type tool-tips, and Racket’s feature-specific profiler (St-Amour et al. 2015).

6.9 Effort
The implementation of the `if` restructuring tool requires under 200 lines of code, including the small changes made to the MiniJava implementation. The implementation took approximately one day, assuming a basic understanding of the DrRacket plug-in system.

7. Conclusion
This paper introduces the key elements of the Racket language workbench via the MiniJava sample language and three benchmark challenges. Racket’s linguistic reuse capabilities allows programmers to easily build new languages and then extend and adapt them. Indeed, Racket’s unique approach to linguistic reuse extends to its ecosystem.

Racket’s language-building facilities are the result of a gradual evolution over its twenty year history: starting from Lisp’s macro system; adding hygiene to preserve lexical scope; introducing syntax objects to encapsulate hygiene, then extending those to carry arbitrary meta-information; and finally integrating with Racket’s module system to enable reader customization and linguistic dispatch. Throughout, Racket’s guiding principle of generalizing language features—lexical scope, modules, etc.—and giving programmers full access to them—on equal footing with Racket’s authors—led us to a powerful tool for building and extending languages (Felleisen et al. 2015).
Bibliography


