From Macros to DSLs: The Evolution of Racket*

Ryan Culpepper
PLT
ryanc@racket-lang.org

Matthias Felleisen
PLT
matthias@racket-lang.org

Matthew Flatt
PLT
mflatt@racket-lang.org

Shriram Krishnamurthi
PLT
sk@racket-lang.org

Abstract

The Racket language promotes a language-oriented style of programming. Developers create many domain-specific languages, write programs in them, and compose these programs via Racket code. This style of programming can work only if creating and composing little languages is simple and effective. While Racket's Lisp heritage might suggest that macros suffice, its design team discovered significant shortcomings and had to improve them in many ways. This paper presents the evolution of Racket's macro system, including a false start, and assesses its current state.

2012 ACM Subject Classification Software and its engineering → Semantics

Keywords and phrases design principles, macros systems, domain-specific languages

Digital Object Identifier 10.4230/LIPIcs...0

* Over 20 years, this work was partially supported by our host institutions (Brown University, Northeastern University, Prague Technical University, and University of Utah) as well as several funding organizations (AFOSR, Cisco, DARPA, Microsoft, Mozilla, NSA, and NSF).

1 Macros and Domain-Specific Languages

The Racket manifesto [19, 20] argues for a *language-oriented programming* (LOP) [12, 43] approach to software development. The idea is to take Hudak's slogan of "languages [as] the ultimate abstractions" [32] seriously and to program with domain-specific languages (DSLs) as if they were proper abstractions within the chosen language. As with all kinds of abstractions, programmers wish to create DSLs, write programs in them, embed these programs in code of the underlying language, and have them communicate with each other.

According to the Lisp worldview, a language with macros supports this vision particularly well. Using macros, programmers can tailor the language to a domain. Because programs in such tailored languages sit within host programs, they can easily communicate with the host and each other. In short, creating, using, and composing DSLs looks easy.

Macros alone do not make DSLs, however, a lesson that the Racket team has learned over 20 years of working on a realization of language-oriented programming. This paper recounts Racket's history of linguistic mechanisms designed to support language-oriented programming; it formulates desiderata for DSL support based on, and refined by, the Racket team's experiences; it also assesses how well the desiderata are met by Racket's current capabilities. The history begins with Scheme's hygienic macros, which, in turn, derive from Lisp (see sec. 2). After a false start (see sec. 4), the Racket designers switched to procedural, hygienic macros and made them work across modules; they also strictly separated expansion time from run time (see sec. 5). Eventually, they created a meta-DSL for writing macros that could properly express the grammatical constraints of an extension, check them, and synthesize proper error messages (see sec. 6). A comparison between general DSL implementation desiderata (see sec. 3) and Racket's capabilities shows that the language's support for a certain class of DSLs still falls short in several ways (see sec. 7).

Note This paper does *not* address the safety issues of language-oriented programming. As a similar set of authors explained in the Racket Manifesto [19], language-oriented programming means programmers use a host language to safely compose code from many small pieces written in many different DSLs and use the very same host language to implement the DSLs themselves. Hence language-oriented programming clearly needs the tools to link DSLs safely (e.g., via contracts [21] or types [40]) and to incorporate systems-level protection features (e.g., sandboxes and resource custodians [29]).

Some Hints For Beginners on Reading Code

We use Racket-y constructs (e.g., `define-syntax-rule`) to illustrate Lisp and Scheme macros. Readers familiar with the original languages should be able to reconstruct the original ideas; beginners can experiment with the examples in Racket.

Lisp's S-expression construction		Racket's code construction	
`'`	S-expression quote	`#'`	code quote
`` ` ``	Quine quasiquote	`` #` ``	code quasiquote
`,`	Quine unquote	`#,`	code unquote
`@,`	list splicing	`#@,`	code splicing

Figure 1 Hints on strange symbols

For operations on S-expressions, i.e., the nested and heterogeneous lists that represent syntax trees, Lisp uses `car` for `first`, `cdr` for `rest`, and `cadr` for `second`. For the convenient construction of S-expressions, Lisp comes with an implementation of Quine's quasiquotation

idea that uses the symbols shown on the left of fig. 1 as short-hands: quote, quasiquote, unquote, and unquote-splicing. By contrast, Racket introduces a parallel datas tructure (syntax objects). To distinguish between the two constructions, the short-hand names are prefixed with `#`.

2 The Lisp and Scheme Pre-history

Lisp has supported macros since 1963 [31], and Scheme inherited them from Lisp in 1975 [41]. Roughly speaking, a Lisp or Scheme implementation uses a *reader* to turn the sequence of characters into a concrete tree representation: *S-expressions*. It then applies an *expander* to obtain the abstract syntax tree(s) (AST). The expander traverses the S-expression to find and eliminate uses of macros. A macro rewrites one S-expression into a different one, which the expander traverses afresh. Once the expander discovers a node with a syntax constructor from the core language, say `lambda`, it descends into its branches and recursively expands those. The process ends when the entire S-expression is made of core constructors.

A bit more technically, macros are functions of type

```
S-expression ⟶ S-expression
```

The `define-macro` form defines macros, which are written using operators on S-expressions. Each such definition adds a macro function to a table of macros. The expander maps an S-expression together with this table to an intermediate abstract syntax representation:

```
S-expression × TableOf[MacroId, (S-expression ⟶ S-expression)] ⟶ AST
```

The `AST` is an internal representation of an S-expression of only core constructors.

See the left-hand side of fig. 2 for the simple example of a `let` macro. As the comments above the code say, the `let` macro extends Racket with a block-structure construct for local definitions. The macro implementation assumes that it is given an S-expression of a certain shape. Once the definition of the `let` macro is recognized, the expander adds the symbol `'let` together with the specified transformer function to the macro table. Every time the macro expander encounters an S-expression whose head symbol is `'let`, it retrieves the macro transformer and calls it on the S-expression. The function deconstructs the given S-expression into four pieces: `decl`, `lhs`, `rhs`, `body`. From these, it constructs an S-expression that represents the immediate application of a `lambda` function.

```
;; PURPOSE extend Racket with block-oriented, local bindings
;;
;; ASSUME the given S-expression has the shape
;;  (let ((lhs rhs) ...) body ...)
;; FURTHERMORE ASSUME:
;; (1) lhs ... is a sequence of distinct identifiers
;; (2) rhs ..., body ... are expressions
;; PRODUCE
;;  ((lambda (lhs ...) body ...) rhs ...)
```

```
(define-macro (let e)
  (define decl (cadr e))
  (define lhs  (map car decl))
  (define rhs  (map cadr decl))
  (define body (cddr e))
  ;; return
  `((lambda ,lhs ,@body) ,@rhs))
```

```
(define-syntax-rule
  (let ((lhs rhs) ...) body ...)
  ;; rewrites above pattern to template below
  ((lambda (lhs ...) body ...) rhs ...))
```

Figure 2 Macros articulated in plain Lisp vs Kohlbecker's macro DSL

Macros greatly enhance the power of Lisp, but their formulation as functions on S-expressions is both error-prone and inconvenient. As fig. 2 shows, the creator of the function makes certain assumption about the shape of the given S-expression, which are not guaranteed by the macro expansion process. Yet even writing just a transformation from the assumed shape of the S-expression to the properly shaped output requires bureaucratic programming patterns, something a macro author must manage and easily causes omissions and oversights.

For concreteness, consider the following problems in the context of `let` macro:

1. The S-expression could be an improper list. The transformer, as written, does not notice such a problem, meaning the compilation process ignores this violation of the implied grammar of the language extension.
2. The S-expression could be too short. Its second part might not be a list. If it is a list, it may contain an S-expression without a `cadr` field. In these cases, the macro transformer raises an exception and the compilation process is aborted.
3. The S-expression has the correct length but its second part may contain lists that contain too many S-expressions. Once again, the macro transformer ignores this problem.
4. The S-expression may come with something other than an identifier as the `lhs` part of a local declaration. Or, it may repeat the same identifier as an `lhs` part of the second clause. In this case, the macro generates code anyways, relying on the rest of the compilation process to discover the problems. When these problems are discovered,
   a. it may have become impossible to report the error in terms of source code, meaning a programmer might not understand where to look for the syntax error.
   b. it has definitely become impossible to report errors in terms of the language extension, meaning a programmer might not comprehend the error message.
5. The author of the macro might forget the unquote `,` to the left of `lhs`. In many members of the Lisp family, the resulting code would be syntactically well formed but semantically rather different from the intended one. In particular, conventional Lisp would generate a function that binds all occurrences of `lhs` in `body` via this newly created `lambda`—a clear violation of the intended scoping arrangements expressed in the comments.

In short, if the S-expression fails to live up to the stated assumptions, the macro transformation may break, ignore code, or generate code that some later step in the compilation process recognizes as an error but describes in inappropriate terms. If the programmer makes even a small mistake, strange code may run and is likely to cause inexplicable run-time errors.

Kohlbecker's dissertation research on macros greatly improves this situation [34, 35, 36]. His macro system for Scheme 84 adds two elements to the macro writer's toolbox. The first is a DSL for articulating macro transformations as rewriting rules consisting of a pattern and a template. The revised macro expander matches S-expressions against the specified patterns; if there is a match, the template is instantiated with the resulting substitution. This DSL removes programming patterns from macro definitions and, to some extent, eliminates problems 1 through 3 from above. For an example, see the right-hand side of fig. 2.

Note We call Lisp-style macros *procedural* and Kohlbecker's approach *declarative*.

The second novel element is hygienic expansion. By default, Kohlbecker's macro expander assumes that identifiers contained in the source must be distinct from macro-generated identifiers in binding positions. As such, it eliminates the need to explicitly protect against accidental interference between the macro's lexical scopes and those of its use contexts—that is, yet another programming pattern from macro code. At a minimum, this hygienic expander would not bind `lhs` in `body` as indicated in problem 5 above.

Further work [2, 7, 14] refined and improved the pattern-oriented approach to specifying macros as well as hygienic macro expansion. The `define-syntax-rule` construct and

hygienic expansion became part of the Scheme standard by the late 1990s [1]. Starting in 1988 and in parallel to the Scheme standardization process, Dybvig et al. [14] designed and implemented a macro definition construct, `define-syntax-cases` (in actual code it requires a combination of `define-syntax` and `syntax-case`) that merged the procedural and declarative elements of the Lisp world. Dybvig et al. also switched from S-expressions to trees of syntax objects. These trees included source locations so that the error handling code could try to point back to the surface code (problem 4a above).

Starting in the late 80s, researchers explored other ways to facilitate the work of macro authors, including two relevant to creating DSLs from macros. Dybvig et al. [13] invented expander-passing macros. Macro authors would write their own expanders and use different ones in different macros. At an abstract level, expansion-passing style anticipates the need for checking static attributes. Blume [3] as well as Kelsey and Reese [33] added modules that could export and import macros. Such modules allow macro programmers to encapsulate bundles of macros, a first step towards encapsulating a DSL's design and implementation.

3 DSLs Require More than Bunches of Macros

Scheme-style macros greatly improve on Lisp's as far as the *extension* of an existing language is concerned. A developer can add concise and lexically correct macros to a program and may immediately use them, for writing either ordinary run-time code or additional macros. This immediacy is powerful and enticing because a programmer never has to leave the familiar programming environment, use external tools, or mess with scaffolding setups.

The idea of macros is also easy to comprehend at the abstract level. Conceptually, a macro definition adds a new alternative to one of Racket's grammatical productions: definitions or expressions. The declarative approach makes it easy to specify simple S-expression transformers in a straightforward manner; hygienic macro expansion guarantees the integrity of the program's lexical scope.

The problem is that a language extension provides only a false sense of a purpose-tailored language. On one hand, a programmer who uses a bunch of macro-based language extensions as if it were a self-contained DSL must code with an extreme degree of self-discipline. On the other hand, the macro system fails to support some of the traditional advantages of using DSLs: catching mistakes in the parsing or type-checking stage; exploiting constraints to generate optimized code; or link with/target tailor-made run-time functions.

Conventionally, the creation of DSLs demands a pipeline of compiler passes:

1. a *parser*, based on explicit specification of a domain-specific *vocabulary* and a *grammar*, that reports errors at the DSL's source level;
2. a *static semantics*, because one goal of migrating from an application interface to a DSL is to enforce certain constraints statically;
3. a *code generation* and *optimization* pass, because another goal of introducing DSLs is to exploit the static or linguistic constraints for improved performance; and,
4. a *run-time system*, because (1) the host language may lack pieces of functionality or (2) the target language might be distinct from the host language.

Scheme macros *per se* do not support the creation of such pipelines or its proper encapsulation.

The Racket designers noticed some of these problems when they created their first teaching languages [17, 18]. In response, they launched two orthogonal efforts to support the development of DSLs via language-extension mechanisms with the explicit goal of retaining the ease of use of the latter:

- One concerned the encapsulation of DSLs and support for some traditional passes. This idea was to develop a module system that allows the export and import of macros and functions while also retaining a notion of separate compilation for modules.
- The other aimed at a mechanism for easily expressing a macro's assumptions about its input and synthesizing error messages at the appropriate level, i.e., the problems from sec. 2. The results would also help with implementing DSLs via modules.

While sec. 4 reports on an ambitious, and abandoned, attempt to address these problems all at once, secs. 5 and 6 describe the tools that Racket eventually provided to DSL designers and implementors.

4 Ambitious Beginnings

When the Racket designers discovered the shortcomings of a traditional Scheme macro system, they decided to address them with three innovations. First, they decided to move beyond the traditional S-expression representation of syntax and instead use a richly structured one (see sec. 4.1). Second, they realized that macros needed to work together to implement context-sensitive checks. To this end, they supplemented declarative macros with procedural *micros* that could deal with attributes of the expansion context (see sec. 4.2). Finally they decided to use modules as the containers of macro-based DSL implementations as well as the units of DSL use (see sec. 4.3).

4.1 From S-expressions to Syntax Objects

To track source locations across macro expansion, Racket—like Dybvig's Chez Scheme—introduced a *syntax object* representation of the surface code, abandoning the conventional S-expression representation. Roughly speaking, a syntax object resembles an S-expression with a structure wrapped around every node. At a minimum, this structure contains source locations of the various tokens in the syntax. Using this information, a macro expander can often pinpoint the source location of a syntax error, partially solving problem 4a from sec. 3.

4.2 The Vocabularies of Micros

Recall that a macro is a function on the syntax representation. Once this representation uses structures instead of just S-expressions, the signature of a macro has to be adapted:

```
Syntax-Object ⟶ Syntax-Object
```

Of course, this very signature says that macros cannot naturally express[1] communication channels concerning attributes of the expansion context.

Krishnamurthi et al.'s work [38] supplements macros with *micros* to solve this issue. Like `define-macro`, `define-micro` specifies a function that consumes the representation of a syntax. Additionally, it may absorb any number of `Attribute` values so that collections of micros can communicate contextual information to each other explicitly:

```
Syntax-Object ⟶ (Attribute ... ⟶ Output)
```

As this signature shows, a micro also differs from a macro in that the result is some arbitrary type called `Output`. This type must be the same for all micros that collaborate but may

[1] A macro author could implement this form of communication via a protocol that encodes attributes as syntax objects. We consider an encoding unnatural and therefore use the phrase "naturally express."

differ from one collection of micros to another. For macro-like micros, `Output` would equal `Syntax-Object`. By contrast, for an embedded compiler `Output` would be `AST`, meaning the type of abstract syntax trees for the target language. This target language might be Racket, but it could also be something completely different, such as GPU assembly code. The Racket team did not explore this direction at the time.

As this explanation points out, micros for DSLs must be thought of as members of a collection. To make this notion concrete, Krishnamurthi et al. also introduce the notion of a *vocabulary*. Since collections of macros and micros determine the "words" and "sentence structure" of a DSL, a vocabulary represents the formal equivalent of a dictionary and grammar rules. The micros themselves transform "sentences" in an embedded language into meaningful—that is, executable—programs.

In Krishnamurthi et al.'s setting, a vocabulary is created with `(make-vocabulary)` and comes with two operations: `define-micro`, which adds a micro function to a given vocabulary, and `dispatch`, which applies a micro to an expression in the context of a specific vocabulary.

```
;; type Output = RacketAST

(define compiler (make-vocabulary))

_ _ _ elided _ _ _

(define-micro compiler
  (if cond then else)
  ==>
  (lambda ()
    (define (expd t)
      ((dispatch t compiler)))
    (define cond-ir (expd cond))
    (define then-ir (expd then))
    (define else-ir (expd else))
    (make-AST-if
      cond-ir then-ir else-ir)))

_ _ _ elided _ _ _

(define compiler-language
  (extend-vocabulary
    base-language
    compiler))
```

```
;; type Output = RacketType
(define type-check (make-vocabulary))

_ _ _ elided _ _ _

(define-micro type-check
  (if cond then else)
  ==>
  (lambda (Γ)
    ;; first block
    (define (tc t)
      ((dispatch t type-check) Γ))
    (define cond-type (tc cond))
    (unless (type-== cond-type Boolean)
      (error _ _ _ elided _ _ _))
    (define then-type (tc then))
    (define else-type (tc else))
    (unless (type-== then-type else-type)
      (error _ _ _ elided _ _ _))
    then-type))

_ _ _ elided _ _ _
```

Figure 3 Micros and vocabularies

Fig. 3 collects illustrative excerpts from a vocabulary-micro code base. The left-hand column sets up a `compiler` vocabulary, which expresses transformations from the surface syntax into Racket's core language. Among other micros, the `if` micro is added to `compiler` because it is a core construct. The final definition shows how to construct a complete language implementation by mixing in vocabularies into the common base language.

Like Scheme's macro definitions, micro definitions use a pattern DSL for specifying inputs. As for the `Attribute ...` sequence, micros consume those via an explicit `lambda`. To create its output, the `if` micro allocates an AST node via `make-AST-if`. The pieces of this node are the results of expanding the three pieces that make up the original `if` expression. The expansions of these sub-expressions employ `dispatch`, a function that consumes the expression to be expanded together with the contextual vocabulary and the attributes (none here) in a staged fashion.

The right-hand side of fig. 3 shows how to add an `if` micro for a type-checking variant of the DSL. The code introduces a second vocabulary for the type checker. The `if` micro for this additional vocabulary implements the type checking rule for `if` in a standard manner, reporting an error as soon it is discovered.

Once the `type-check` vocabulary is in place, a developer can use it independently or in combination with the `compiler` vocabulary. For example, Racket's soft typing system [23] needed a distinct interpretation for the language's `letrec` construct, i.e., a distinct `type-check` vocabulary unrelated to the actual compiler. A variant of Typed Racket [42] could be implemented via the composition of these two vocabularies; in this context, the composition would discard the result of the pass based on the `type-check` vocabulary.

In general, DSL creators get two advantages from vocabularies and micros. First, they can now specify the syntax of their languages via explicit collections of micros. Each micro denotes a new production in the language's expression language, and the input patterns describe its shape. Second, they can naturally express and implement static checking. The micro's secondary arguments represent "inherited" attribute, and the flexible `Output` type allows the propagation of "synthesized" ones.

Implementing complete DSLs from vocabularies becomes similar to playing with Legos: (1) vocabularies are like mixins [30], (2) languages resemble classes, and (3) `dispatch` is basically a method invocation. Hence creating a variety of similar DSLs is often a game of linking a number of pieces from a box of language-building blocks. For the team's rapid production and modification of teaching languages in the mid 1990s, vocabularies were a critical first step.

4.3 Languages for Semantic Modules

According to sec. 3 the implementation of any language combines a compiler with a run-time system. This dictum also applies to DSLs, whether realized with macros or micros. Both translate source code to target code, which refers to run-time values (functions, objects, constants, and so on). Such run-time values often collaborate "via conspiracy," meaning their uses in target code satisfies logical statements—invariants that would not hold if all code had free access to these values. That is, the implementor of a DSL will almost certainly wish to hide these run-time values and even some of the auxiliary compile-time transformations. All of this suggests that macros, micros and vocabularies should go into a module, and such modules should make up a DSL implementation.

Conversely, the implementors of DSLs do not think of deploying individual constructs but complete languages. Indeed, conventional language implementors imagine that DSL programmers create self-contained programs. By contrast, Lispers think of their language extensions and imagine that DSL programmers may wish to escape into the underlying host language or even integrate constructs from different DSL-like extensions at the expression level. The question is whether a macro-micro based approach can move away from the "whole program" thinking of ordinary DSLs and realize a Lisp-ish approach of deploying languages for small units of code.

Krishnamurthi's dissertation [37] presents answers to these two questions and thus introduces the first full framework for a macro-oriented approach to language-oriented programming. It combines macros with the first-class modules of the 1990s Racket, dubbed units [28], where the latter becomes both the container for DSL implementations as well as the one for DSL deployment. Technically, these units have the shape of fig. 4. They are parameterized over a `Language` and link-time imports, and they export values in response.

```
(unit/lang Language
  (ImportIdentifier ...)
  (ExportIdentfier ...)
  Definitions-and-Expressions ...)
```

Figure 4 Language-parameterized, first-class units

A DSL implementation is also just a `unit/lang` that combines macros, micros, and run-time values. It is not recognized as a valid `Language` until it is registered with a *language administrator*. The latter compiles `unit/lang` expressions separately to plain `unit`s. For this compilation, the language administrator expands all uses of macros and micros and then resolves all the names in the generated code—without exposing any of them to other code. In particular, the developer does not need to take any action, such as adding the names of run-time values to export specifications of `Language`s or to `unit/lang`s that use a `Language`. The result of a compilation is a collection of plain Racket `unit`s, and the Racket compiler turns this collection into a running program.

In principle, Krishnamurthi's `unit/lang` system addresses all four DSL criteria listed in sec. 3. The micro-vocabulary combination can enforce syntax constraints beyond what macros can do. They are designed to express static processing in several passes and explicitly accommodate target languages distinct from Racket. And, the implementations of DSLs as `unit/lang`s encapsulates the compiler component with a run-time component.

What this system fails to satisfy is the desire to synthesize DSL implementation techniques with Lisp's incremental language-extension approach. The main problem is that a programmer has to parameterize an entire unit over a complete language. It is impossible to selectively import individual macros and micros from a `unit/lang`, which is what Racket developers truly want from a modular macro system. After a few years of using plain `unit`s, the Racket team also realized that first-class units provided more expressive power than they usually needed, meaning the extra complexity of programming the linking process rarely ever paid off in the code base.

Additionally, the `unit/lang` system was a step too far on the social side. Racket—then called PLT Scheme—was firmly in the Scheme camp and, at the time, the Scheme community had developed and open-sourced a new syntax system [14] that quickly gained in popularity. This macro system combined the declarative form of Krishnamurthi's macros with the procedural form of his micros into a single `define-syntax-cases` form. Furthermore, this new macro system came with the same kind of syntax-object representation as Krishnamurthi's, allowing source tracking, hygienic expansion, and other cross-expansion communication. In other words, the new system seemed to come with all the positive aspects of Krishnamurthi's without its downsides. Hence, the Racket team decided to adapt this macro system and create a module system around it.

5 Languages From Syntactic Modules

The Racket designers started this rebuilding effort in 2000. The goal was to create a module system where a developer could write down each module in a DSL that fit the problem domain and where a module could export/import individual macros to/from other modules—and this second point forced them to reconsider the first-class nature of modules.

Flatt's "you want it when" module-macro system [24] realizes this goal. It introduces a module form, which at first glance looks like `unit/lang`. Like the latter, `module` explicitly specifies the language of a module body, as the grammar in fig. 5 shows. Otherwise the

```
(module Name Language
  {    ProvideSpeification
     | RequireSpecification
     | Definition
     | Expression }*
)
```

```
#lang Language                    Name.rkt

{    ProvideSpeification
   | RequireSpecification
   | Definition
   | Expression }*
```

Figure 5 Language-parameterized, first-order modules and their modern abbreviation

grammar appears to introduce a new expression form whose internals consist of a sequence of exports, imports, definitions and expressions. A small difference concerns the organization of the module body. The import and export specifications no longer need to show up as the first element of the module; they can appear anywhere in the module. Appearances are deceiving, however, and the `Name` part suggests the key difference. A module is *not* an expression but a first-order form, known to the expander.

When the expander encounters `module`, it imports the `Language`'s provided identifiers. This step establishes the base syntax and semantics of the module's expressions, definitions, imports, and exports. Next the expander finds the imported and locally-defined macros in the module body. The search for imported macros calls for the expansion and compilation of the referenced modules. It is this step that requires the restriction to first-order modules, because the expander must be able to identify the sources of imported macros and retrieve their full meaning. Finally, the expander adds those imported and local macros to the language syntax and then expands the module body properly, delivering an abstract-syntax representation in the Racket core language.

One consequence of this arrangment is that the expansion of one module may demand the evaluation of an entire tower of modules. The first module may import and use a macro from a second module, whose definition relies on code that also uses language extensions. Hence, this second module must be compiled after expanding and compiling the module that supplies these auxiliary macros.

```
#lang racket                      loop.rkt

(provide inf-loop)

(define-syntax-cases
  [(inf-loop e)
   (begin
     (displayln "generating inf-loop")
     #'(do-it (lambda () e)))])

(define (do-it th)
  (th)
  (do-it th))
```

```
#lang racket                      use-loop.rkt

(provide display-infinitely-often)

(require "loop.rkt")

(define (display-infinitely-often x)
  (inf-loop (do-it x)))

(define (do-it x)
  (displayln x))
```

Figure 6 Exporting macros from, and importing them into, modules

The right-hand side of fig. 5 also shows the modern, alternative syntax for modules. The first line of code specifies only the language of the module via a `#lang` specification; the name of the file (boxed) determines the name of the module. Fig. 6 illustrates how two modules interact at the syntax and run-time level. The module on the left defines the language extension `inf-loop`, whose implementation generates code with a reference to the function `do-it`. The module on the right imports this language extension via the `require` specification. The Racket compiler retrieves the macro during compile time and uses it to

expand the body of the `display-infinitely-often` function—including a reference to the `do-it` function in module `loop.rkt`. Cross-module hygienic expansion [24, 26] ensures that this macro-introduced name does not conflict in any way with the `do-it` function definition of the `use-loop.rkt` module. Conceptually, the expansion of `display-infinitely-often` looks like the following definition:

```
(define (display-infinitely-often x)
  (loop.rkt-do-it (lambda () (use-loop.rkt-do-it x))))
```

with the two distinct, fully-resolved names guaranteeing the proper functioning of the code according to the intuitive surface meaning.

Flatt's module-macro system allows the use of both declarative and procedural language extensions. To illustrate the latter kind, the `inf-loop` macro uses `define-syntax-cases`. If the expander can match a piece of syntax against one of the left-hand-side patterns of `define-syntax-cases`, it evaluates the expression on the right. This evaluation must produce code, which is often accomplished via the use of templates (introduced by `#'`) whose pattern variables are automatically replaced by matching pieces of syntax. But, as the definition of `inf-loop` suggests, the right-hand side may contain side-effecting expressions such as `displayln`. While these expressions do not become a part of the generated code as the above snippet shows, their side effects are observable during compile time.

To enable separate compilation, Racket discards the effects of the expansion phase before it moves on to running a module. Discarding such effects reflects the Racket designers' understanding that language-extensions are like compilers, which do not have to be implemented in the same language as the one that they compile and which are not run in the same phase as the program that they translate. Phase separation greatly facilitates reasoning about compilation, avoiding a lot of the pitfalls of Lisp's and Scheme's module systems with their explicit `eval-when` and `compile-when` instructions [24].

`math.rkt`

```
#lang racket

(provide Ack)

;; Number Number -> Number
(define (Ack x y)
  (cond
    [(zero? x) (+ y 1)]
    [(and (> x 0) (zero? y))
     (Ack (- x 1) 1)]
    [else
     (Ack (- x 1) (Ack x (- y 1)))]))
```

`use-acker.rkt`

```
#lang racket

(require (for-syntax "math.rkt")))

(define-syntax-cases ()
  [(static-Ack x y)
   ;; rewrites the pattern to a template
   ;; via some procedural processing
   (let* ((x-e (syntax-e #'x))
          (y-e (syntax-e #'y)))
     (unless (and (number? x-e) (number? y-e))
       (raise-syntax-error #f "not numbers"))
     (define ack (Ack x-e y-e))
     #`(printf "the Ack # is ~a" #,ack))])

(static-Ack 1 2)
```

Figure 7 Importing at a different phase

Phase separation imposes some cost on developers, however. If a module needs run-time functions for the definition of a language construct, the import specification must explicitly request a phase shift. For an example, see fig. 7. The module on the right defines `static-Ack`, which computes the Ackermann function of two numbers at compile time. Since the Ackermann function belongs into a different library module, say `math`, the `use-ack` module most import it from there. But, because this function must be used at compile time, the `require` specification uses the (underlined) `for-syntax` annotation to shift the import to this early phase. The Racket designers' experience shows that phase-shifting annotations are

still significantly easier to work with than Lisp's and Scheme's `expand-when` and `eval-when` annotations.

Like Krishnamurthi's `unit/lang`s, Flatt's `module`s allow developers to write different components in different languages. In the case of `module`s, the `Language` position points to a module itself. The exports of this `Language` module determine the initial syntax and semantics of a client module.

In contrast to an ordinary `module`, a `Language module` must export certain macros, called interposition points; it may export others. An interposition point is a keyword that the macro expander adds to some forms during its traversal of the source tree. Here are the two most important ones for `Language` modules:

- `#%module-begin` is the (invisible) keyword that introduces the sequence of definitions and expressions in a module body. A `Language` module must export this form.
- `#%top-interaction` enables the read-eval-print loop for a `Language`, i.e., dynamic loading of files and interactive evaluation of expressions.

Other interposition points control different aspects of a `Language`'s meaning:

- `#%app` is inserted into function applications. In source code, an application has the shape `(fun arg ...)`, which expands to the intermediate form `(#%app fun arg ...)`.
- `#%datum` is wrapped around every literal constant.
- `#%top` is used to annotate module-level variable occurrences.

In practice, a developer creates a `Language` by adding features to a base language, subtracting others (by not exporting them), and re-interpreting some. Here "features" covers both macros and run-time values. The `#%module-begin` macro is commonly re-interpreted for a couple of reasons. Its re-definition often helps with the elimination of boilerplate code but also the communication of context-sensitive information from one source-level S-expression (including `module`s) to another during expansion.

lazy.rkt

```
#lang racket

(provide
 (except-out (all-from-out racket) #%app)
 (rename-out [lazy-app #%app]))

(define-syntax-rule
  (lazy-app f a ...)
  (#%app f (lambda () a) ...))
```

no-error.rkt

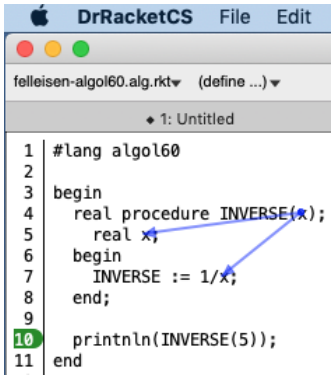
```
#lang "lazy.rkt"

; a constant function
(define (f x y)
  10)

; called on two erroneous terms
(f (/ 1 0) (first '()))
; evaluates to 10
```

Figure 8 Building an embedded DSL from modules and macros

Fig. 8 indicates how a developer could quickly build a language that looks like Racket but uses call-by-name instead of call-by-value. The module on the left is the language implementation. It starts from Racket and re-exports all of its features, including `#%module-begin`, except for function application. The module re-interprets function application via the second part of `provide`. Technically, a re-interpretation consists of a macro definition that is re-named in a `provide`. The `lazy` module comes with a `lazy-app` macro, which rewrites `(lazy-app fun arg ...)` to `(#%app fun (lambda () arg) ...)`. By static scope, the `#%app` in the expansion refers to the function application form of Racket. Since this macro is provided under the name `#%app`, a client module's function applications—into which the expander inserts `#%app`—eventually expand according to `lazy-app`. In particular, the two exception-raising expressions in the `no-error` module are wrapped in `lambda`; because `f` is a constant function that does not evaluate its arguments, these errors are never reported. (For additional details on `lazy`, see the last chapter of *Realm of Racket* [15].)

```
#lang racket                    all-in-one.rkt

(module lazy-impl racket

  (provide
   (except-out (all-from-out racket) #%app)
   (rename-out [lazy-app #%app]))

  (define-syntax-rule
    (lazy-app f a ...)
    (#%app f (lambda () a) ...)))

(module lazy-client (submod ".." lazy-impl)

  (define (f x y)
    10)

  (f (/ 1 0) (first '())))

(require (submod "." lazy-client))
```

(a) DSL development in one module



(b) Algol 60 as a Racket DSL

Figure 9 Developing and deploying DSLs in Racket

Modules and macros jointly make DSL development an interactive activity in the Racket ecosystem. A programmer can open two tabs or windows in an IDE to use one for the DSL implementation and another for a DSL program. Or, a programmer can place a DSL-implementing submodule [25] and a DSL-using submodule into a single file, which can then be edited and executed within a single editor window of the preferred IDE. Fig. 9a shows how to combine the modules of fig. 8 into a single file. This program consists of three pieces. The first one is a submodule that implements the lazy language, while the second uses the first one in the `Language` position. Hence the first submodule is the programming language of the second. The last piece of the program requires and thus evaluates the client module. Any change to the first submodule is immediately visible in the second.

A developer may also equip a DSL with any desired syntax, not just build on top of Racket's beautiful parentheses. To support this kind of syntax, a `Language` module may export a new reader. Recall from sec. 2 that a Lisp reader turns the stream of characters into a sequence of S-expressions (or `Syntax-Objects`, in the case of Racket). The rest of the implementation can then use the usual mix of macros and functions. Butterick's *Beautiful Racket* [4] is a comprehensive introduction to this strategy and comes with a powerful library package for lexing and parsing.

In the context of modular macros, a developer may also create a conventional compiler with the macro infrastructure. Instead of just expanding to Racket, a DSL implementation may use a combination of macros and compile-time functions to perform conventional type checking or other context-sensitive checks.

Fig. 9b presents a simple example of a Racket DSL program in conventional syntax. Except for the first line, the code represents a standard Algol 60 program. The first line turns this program into a Racket DSL and thus allows Racket to parse, type check, compile, and run this program. Because the DSL implementation turns the Algol 60 program into syntax objects and implements its semantics via macro expansion, DrRacket (the Racket IDE [22]) automatically adapts itself to this new language. For example, fig. 9b illustrates how DrRacket connects the binding occurrence of `INVERSE`'s parameter to its bound ones.

In sum, Racket's modules simultaneously allow the incremental and interactive construction of language extensions as well as the construction of complete DSLs with their own vocabulary. The key design decision is to turn macros into entities that first-order modules can export, import, hide, and re-interpret. It does necessitate the introduction of strict phase separation between the expansion phase and run-time phase to obtain separate compilation.

6 Syntax Done Properly With Parsing Macros

The implementation of a DSL's syntax consists of two essential parts: parsing syntactically legitimate sentences, and reporting violations of the syntactic rules. Both aspects are equally important, but for 40 years, the macro community mostly neglected the second one.

Sec. 2 lists five problems with parsing via Lisp-style macros. Kohlbecker's rewriting DSL—based on patterns and templates—eliminates all of them except for problem 4. To appreciate the complexity of this particular problem, consider the actual grammatical production of `let` expressions in classical BNF notation:

```
(let ({[id expression]}*) expression+)
```

Kohlbecker's pattern-based meta-DSL addresses this context-free shape specification with the elegant trick of using ellipses (...) for * and unrolling for +:

```
(let ([id expression] ...) expression expression ...)
```

What Kohlbecker's notation cannot express is the side condition of fig. 2:

```
id ... is a sequence of distinct identifiers
```

Indeed, Kohlbecker's notation cannot even specify that `id` must stand for an identifier.

So now imagine a programmer who writes

```
(let (((+ 1 2) x)) (* x 3))  ;; => ((lambda ((+ 1 2)) (* x 3)) x)
```

or

```
(let ((x 1) (x 2)) (* x 3))  ;; => ((lambda (x x) (* x 3)) 1 2)
```

In either case, a pattern-oriented language generates the `lambda` expression to the right of the `=>` arrow. Hence, the resulting syntax errors speak of `lambda` and parameters, concepts that the grammatical description of `let` never mentions. While a reader might be tempted to dismiss this particular error message as "obvious," it is imperative to keep in mind that this `let` expression might have been generated by the use of some other macro, which in turn might be the result of some macro-defining macro, and so on.

Dybvig's `define-syntax-cases` slightly improves on Kohlbecker's DSL. It allows the attachment of of *fenders*—Boolean expressions—to a macro's input patterns. With such fenders, a macro developer can manually formulate conditions that check such side conditions. Even in such simple cases as `let`, however, the error-checking code is many times the size of the rewriting specification. And this is why most macro authors fail to add this code or, if they do, fail to write comprehensive checks that also generates good error messages.

Culpepper's DSL for defining macros solves this problem with two innovations [8,9,10,11]. The first is an augmentation of the pattern-matching DSL with "words" for articulating classification constraints such as those of the `let` macro. The second is a DSL for specifying new classifications. Together, these innovations allow programmers to easily enforce assumptions about the surface syntax, synthesize error messages in terms of the specification, and deconstruct the inputs of a macro into relevant pieces.

```
(define-syntax-class distinct-bindings
 #:description "sequence of distinct binding pairs"
 (pattern (b:binding ...)
  #:fail-when (check-duplicate-id #'(b.lhs ..))
              "duplicate variable name"
  #:with (lhs* ...) #'(b.lhs ...)
  #:with (rhs* ...) #'(b.rhs ...)))
```

```
(define-syntax-class binding
 #:description "binding pair"
 (pattern (lhs:id rhs:expr)))
```

Figure 10 Syntax classifications

Following our discussion above, the specification of `let` needs two syntax classifications: one to say that the second part of `let`'s input is a sequence and another one to say that the elements of this sequence are identifier-expression pairs. Fig. 10 shows how a programmer can define these classifications in Culpepper's meta-DSL. A classification must come with at least one `pattern` clause, which spells out the context-free shape of the form and names its pieces. For example, the `binding` class uses the pre-defined classifications `id` (for identifier) and `expr` (for expression) to say that a binding has the shape `(id expr)` and that the name of the `id` is `lhs` and the name of `expr` is `rhs`. Any use of such a syntax class, for example the one in the definition of `distinct-bindings`, may refer to these attributes of the input via a dot notation. Thus, `b.lhs ...` in `distinct-bindings` denotes the sequence of identifiers. As this example also shows, a syntax-class definition may also defer to procedural code, such as `check-duplicate-id` to process the input. A `fail-when` clause allows macro developers to specify a part of the synthesized error message (when the default is not sufficiently clear).

```
(define-syntax-parser let
  [(_ bs:distinct-bindings body:expr ...+)
   ;; rewrites the pattern to a template
   #'((lambda (bs.lhs* ...) body ...) bs.rhs* ...)])
```

Figure 11 Macros via parsing macros

Using these two syntax classes, specifying the complete shape of `let` is straightforward; see fig. 11. The `:distinct-bindings` classification of `bs` introduces names for two pieces of the input syntax: a sequence of identifiers (`bs.lhs*`) and a sequence of right-hand-side expressions (`bs.rhs*`), one per variable. The syntax template uses these pieces to generate the same target code as the macros in fig. 2.

A comparison of figs. 2 and 11 illustrates the advantages as well as the disadvantages of Culpepper's DSL for writing macros. On the positive side, the size of the Culpepper-style macro definition appears to remain the same as the one for the Kohlbecker-style one. The revised definition merely adds classifications to the macro's pattern and attribute selections to the macro's template. This shallow size comparison camouflages that these small changes cause the macro to check *all* constraints on the shape of `let` and formulate syntax errors in terms of the specified surface syntax. As Culpepper [9, page 469] explains, implementing the same level of assumption checking and error reporting via procedural macros increases the code size by "several factors." Furthermore the "primary benefit [of this meta-DSL] ... is increased clarity" of a macro's input specification and its code template.

On the negative side, macro programmers are now expected to develop syntax classifications such as those in fig. 10 and use them properly in macro definitions, as in fig. 11. While the development of syntax classifications clearly poses a new obstacle, their use comes with a significant payoff *and* most end up as reusable elements in libraries. Hence the cost of developing them is recouped through reuse. As for the use of syntax classifications in

macro templates and patterns, experience shows that most macro programmers consider the annotations as type-like notions and the attribute selections as a natural by-product.

In short, Culpepper's meta-DSL completely replaces the `define-syntax-cases` meta-DSL for macro definitions. By now, the large majority of Racket programmers develop macros in Culpepper's DSL and contribute to the ever-expanding collection of syntax classifications.

	lang. extens. (sec. 3)	lexical scope (2)	classify syntax (1)	error messages (1)	separate compil. (4)	run-time encaps. (4)	code gen. opt. (3)
Lisp macros	✓	–	–	–	–	–	–
Scheme							
– syntax-rules	✓	✓	patterns	–	–	–	–
– syntax-case	✓	✓	patterns & fenders	–	–	–	–
Racket	✓	✓	patterns &	✓	✓&	✓	`module`
– syntax-parse			syn. classes		phases		only

– means programmers have the tools to design manual solutions

Figure 12 A concise overview of Lisp-family language extension features

7 DSL Creators Need Still More Than Modular, Parsing Macros

Racket has made great progress in improving the state of the art of macros with an eye toward both language extension and DSL implementation. Fig. 12 surveys the progress in roughly the terms of sec. 3's criteria. The `syntax-parse` DSL for defining macros can express almost every context-free and -sensitive constraint; macro developers get away with a few hints and yet get code that reports syntax errors in terms of the macro-defined variant. The `module` system supports both the fine-grained export/import of macros for language extensions and the whole-cloth implementation of DSLs.

At this point, implementing DSLs is well within reach for Racket beginners [4, 15] and easy for experts. While beginners may focus on `module`-based DSLs, experts use macros to create fluidly embedded DSLs. Examples are the DSL of pattern-matching for run-time values, the `syntax-parse` DSL itself, and Redex [16, 39]. In this domain, however, the macro framework falls short of satisfying the full list of desiderata for from sec. 3.

To explain this gap, let us concisely classify DSLs and characterize Racket's support:

- *stand-alone DSLs*
  These are the most recognized form in the real world. Racket supports those via `module` languages with at least the same conveniences as other DSL construction frameworks.
- *embedded DSLs with a fixed interface*
  All programming languages come with numerous such sub-languages. For example, `printf` interprets the format DSL—usually written as an embedded string—for rendering some number of values for an output device. In Racket, such DSLs instead come as a new set of expression forms with which programmers compose domain-specific programs. Even in Racket, though, such DSLs allow only restricted interactions with the host.
- *embedded and extensible DSLs with an expression-level interface*
  Racket's DSLs for pattern matching and structure declarations are illuminating examples of this kind. The former allows programmers to articulate complex patterns, with embedded Racket expressions. The latter may contain patterns, which contain expressions, etc. The pattern DSL is extensible so that, for example, the DSL of structure definitions

```
(define-typed-syntax (if cond then else)
  [⊢ cond >> cond-ir ⟹ cond-type]
  [⊢ then >> then-ir ⟹ then-type]
  [⊢ else >> else-ir ⟸ else-type]
  ------------------------------------------------
  [(AST-if cond-ir then-ir else-ir) ⟶ then-type])
```

Figure 13 Type-checking from macros

can automatically generate patterns for matching structure instances. Naturally, this DSL for structure declarations can also embed Racket expressions at a fine granularity.

With regard to the third kind of DSL, Racket's macro approach suffers from several problems. A comparison with the criteria in sec. 3 suggests three obvious ones.

The first concerns DSLs that demand new syntactic categories and, in turn, good parsing and error reporting techniques. While syntax classes allow DSL creators to enumerate the elements of a new syntactic category, this enumeration is fixed. Experienced DSL implementors can work around this restriction, just like programmers can create extensible visitor patterns in object-oriented languages to allow the blind-box extension of data types.

The second problem is about context-sensitive language processing. The existing macro framework makes it difficult to implement context-sensitive static checking, translations, and optimizing transformations—even for just Racket's macros, not to mention those that define new syntactic categories. Chang and his students [5, 6] have begun to push the boundaries in the realm of type checking, a particular rich form of context-sensitivity. Specifically, the team has encoded the rich domain of type checking as a meta-DSL. In essence, this meta-DSL enables DSL creators to formulate type checking in the form of type elaboration rules from the literature (see fig. 13), instead of the procedural approach of fig. 3. However, their innovation exploits brittle protocols to make macros work together [27]. As a result, it is difficult to extend their framework or adapt it to other domains without using design patterns for macro programming.

Finally, the DSL framework fails to accommodate languages whose compilation target is not Racket. Consider an embedded DSL for Cuda programming that benefits from a fluid integration with Racket. Such a DSL may need two interpretations: on computers with graphical co-processors it should compile to GPU code, while on a computer without such a processor it may need to denote a plain Racket expression. Implementing a dependent-type system in the spirit of Chang et al.'s work supplies a second concrete example. The language of types does not have the semantics of Racket's expressions and definitions. Although it is possible to expand such DSLs through Racket's core, it forces DSL developers to employ numerous macro-design patterns.

The proposed work-arounds for these three problems reveal why the Racket team does not consider the problem solved. Racket is all about helping programmers avoid syntactic design patterns. Hence, the appearance of design patterns at the macro level is antithetical to the Racket way of doing things, and the Racket team will continue to look for improvements.

Acknowledgements The authors thank Michael Ballantyne, Eli Barzilay, Stephen Chang, Robby Findler, Alex Knauth, Alexis King, and Sam Tobin-Hochstadt for contributing at various stages to the evolution of Racket's macro system and how it supports LOP. They also gratefully acknowledge the suggestions of the anonymous SNAPL '19 reviewers, Sam Caldwell, Ben Greenman for improving the presentation.

References

1. H. Abelson, R.K. Dybvig, C.T. Haynes, G.J. Rozas, N.I. Adams, D.P. Friedman, E. Kohlbecker, G.L. Steele, D.H. Bartley, R. Halstead, D. Oxley, G.J. Sussman, G. Brooks, C. Hanson, K.M. Pitman, and M. Wand. Revised5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, Aug 1998.
2. Alan Bawden and Jonathan Rees. Syntactic closures. In *Symposium on Lisp and Functional Programming*, pages 86–95, 1988.
3. Matthias Blume. Refining hygienic macros for modules and separate compilation. Technical report tr-h-171, ATR Human Information Processing Research Laboratories, Kyoto, Japan, 1995. `people.cs.uchicago.edu/~blume/papers/hygmac.pdf`.
4. Matthew Butterick. *Beautiful Racket.* 2013. URL: `https://beautifulracket.com/`.
5. Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. In *Symposium on Principles of Programming Languages*, pages 694–705, 2017.
6. Stephen Chang, Alex Knauth, and Emina Torlak. Symbolic types for lenient symbolic execution. In *Symposium on Principles of Programming Languages*, pages 40:1–40:29, 2018.
7. William Clinger and Jonathan Rees. Macros that work. In *Symposium on Principles of Programming Languages*, pages 155–162, 1991.
8. Ryan Culpepper. *Refining Syntactic Sugar: Tools for Supporting Macro Development.* PhD thesis, Northeastern University, 2010.
9. Ryan Culpepper. Fortifying macros. *Journal of Functional Programming*, 22(4–5):439–476, 2012.
10. Ryan Culpepper and Matthias Felleisen. Taming macros. In *Generative Programming and Component Engineering*, pages 225–243, 2004.
11. Ryan Culpepper and Matthias Felleisen. Fortifying macros. In *International Conference on Functional Programming*, pages 235–246, 2010.
12. Sergey Dmitriev. Language-oriented programming: the next programming paradigm, 2004.
13. R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, January 1988.
14. R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.
15. Matthias Felleisen, Forrest Bice, Rose DeMaio, Spencer Florence, Feng-Yun Mimi Lin, Scott Lindeman, Nicole Nussbaum, Eric Peterson, Ryan Plessner, David Van Horn, and Conrad Barski. *Realm of Racket.* No Starch Press, 2013. URL: `http://www.realmofracket.com/`.
16. Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* MIT Press, 2009.
17. Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs. Second Edition.* MIT Press, 2001–2018. URL: `http://www.htdp.org/`.
18. Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum. *Journal of Functional Programming*, 14(4):365–378, 2004.
19. Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In *First Summit on Advances in Programming Languages*, pages 113–128, 2015.
20. Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, February 2018.
21. R. Findler and M. Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming*, pages 48–59, 2002.
22. Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

23 Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Conference on Programming Language Design and Implementation*, pages 23–32, 1996.

24 Matthew Flatt. Composable and compilable macros: You want it *when?* In *International Conference on Functional Programming*, pages 72–83, 2002.

25 Matthew Flatt. Submodules in Racket: you want it when, again? In *Generative Programming and Component Engineering*, pages 13–22, 2013.

26 Matthew Flatt. Binding as sets of scopes. In *Symposium on Principles of Programming Languages*, pages 705–717, 2016.

27 Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that work together: Compile-time bindings, partial expansion, and definition contexts. *Journal of Functional Programming*, 22(2):181—216, March 2012.

28 Matthew Flatt and Matthias Felleisen. Cool modules for HOT languages. In *Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

29 Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems (or, Revenge of the Son of the Lisp Machine). In *International Conference on Functional Programming*, pages 138–147, September 1999.

30 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Symposium on Principles of Programming Languages*, pages 171–183, 1998.

31 Timothy P. Hart. MACROS for LISP. Technical Report 57, MIT Artificial Intelligence Laboratory, 1963.

32 Paul Hudak. Modular domain specific languages and tools. In *Fifth International Conference on Software Reuse*, pages 134–142, 1998.

33 Richard Kelsey and Jonathan Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 5(4):315–335, 1994.

34 Eugene E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, 1986.

35 Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *Symposium on Lisp and Functional Programming*, pages 151–161, 1986.

36 Eugene E. Kohlbecker and Mitchell Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In *Symposium on Principles of Programming Languages*, pages 77–84, 1987.

37 Shriram Krishnamurthi. *Linguistic Reuse*. PhD thesis, Rice University, 2001.

38 Shriram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. From macros to reusable generative programming. In *International Symposium on Generative and Component-Based Software Engineering*, pages 105–120, 1999.

39 Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Rewriting Techniques and Applications*, pages 2–16, 2004.

40 Daniel Patterson and Amal Ahmed. Linking types for multi-language software: Have your cake and eat it too. In *Summit on Advances in Programming Languages*, pages 12:1–12:15, 2017.

41 Gerald L. Sussman and Guy Lewis Steele Jr. Scheme: An interpreter for extended lambda calculus. Technical Report 349, MIT Artificial Intelligence Laboratory, 1975.

42 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Symposium on Principles of Programming Languages*, pages 395–406, 2008.

43 Martin P. Ward. Language oriented programming. *Software Concepts and Tools*, 15:147–161, April 1994.