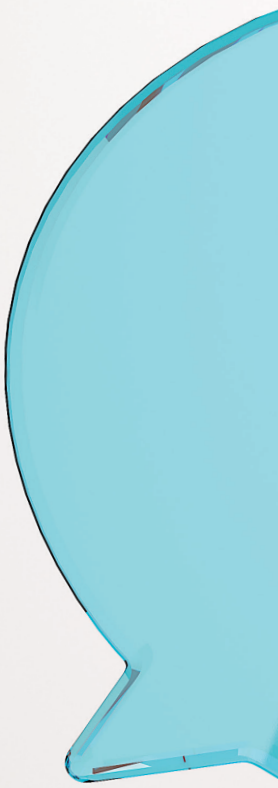
DOI:10.1145/3127323

**As the software industry enters the era of language-oriented programming, it needs programmable programming languages.**

BY MATTHIAS FELLEISEN, ROBERT BRUCE FINDLER, MATTHEW FLATT, SHRIRAM KRISHNAMURTHI, ELI BARZILAY, JAY MCCARTHY, AND SAM TOBIN-HOCHSTADT

# A Programmable Programming Language

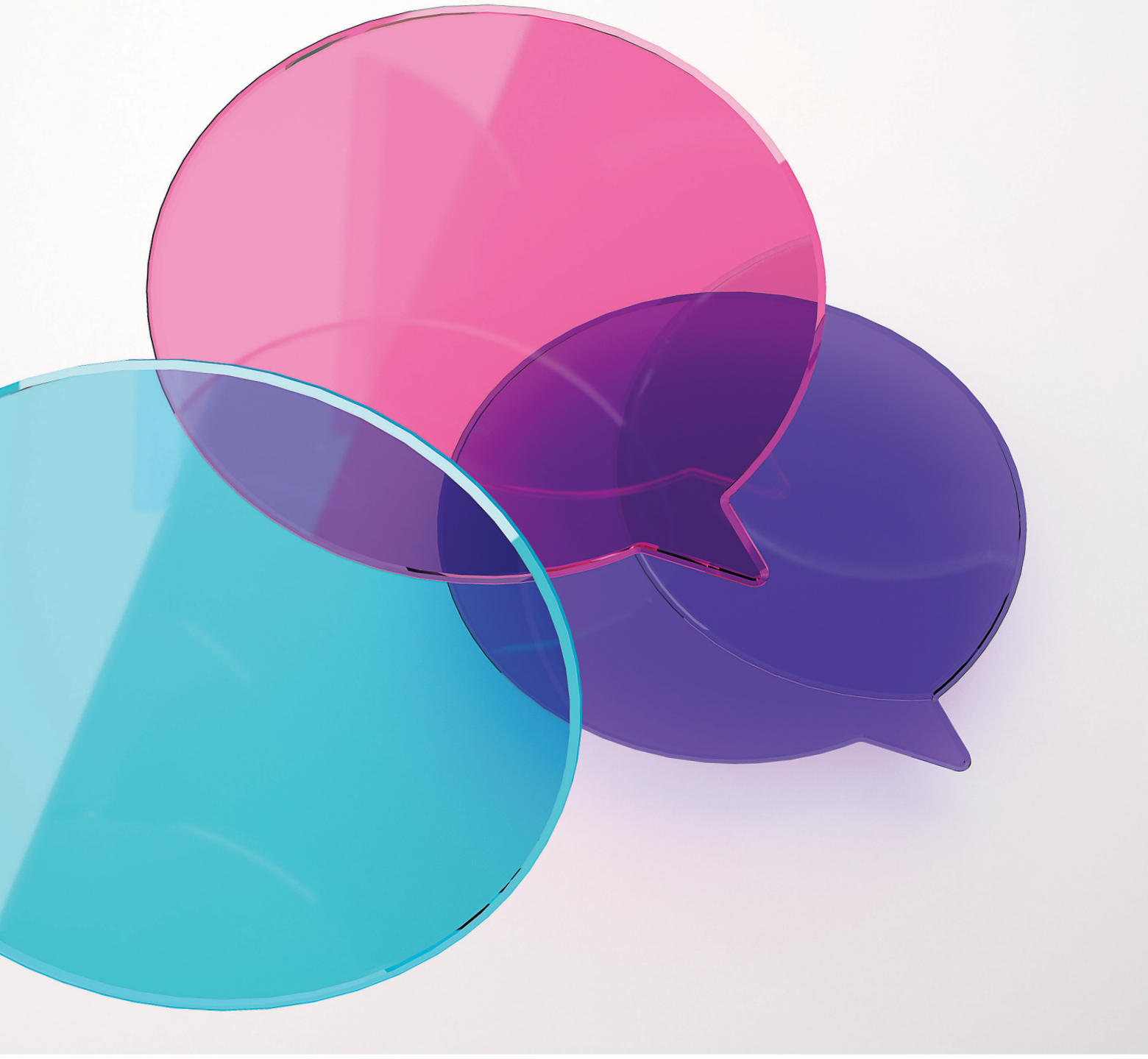


IN THE IDEAL world, software developers would analyze each problem in the language of its domain and then articulate solutions in matching terms. They could thus easily communicate with domain experts and separate problem-specific ideas from the details of general-purpose languages and specific program design decisions.

In the real world, however, programmers use a mainstream programming language someone else picked for them. To address this conflict, they resort to—and on occasion build their own—domain-specific languages embedded in the chosen language (embedded domain-specific languages, or eDSLs). For example, JavaScript programmers employ jQuery for interacting with the Document Object Model and React for dealing with events and concurrency.

## » key insights

- Language-oriented programming is an emerging software-development paradigm likely to revolutionize the way people build software.
- It elevates "language" itself to a software building block, with the same status as objects, modules, and components.
- As with other paradigms, language orientation thrives when the base language supports it directly; the Racket project has worked on support for language-oriented programming for 20 years, providing a platform for exploring this exciting new development in depth.



As developers solve their problems in appropriate eDSLs, they compose these solutions into one system; that is, they effectively write multilingual software in a common host language.[a]

Sadly, multilingual eDSL programming is done today on an ad hoc basis and is rather cumbersome. To create and deploy a language, programmers usually must step outside the chosen language to set up configuration files and run compilation tools and link-in the resulting object-code files. Worse, the host languages fail to support the proper and sound integration of components in different eDSLs. Moreover, most available integrated development environments (IDEs) do not even understand eDSLs or perceive the presence of code written in eDSLs.

The goal of the Racket project is to explore this emerging idea of language-oriented programming, or LOP, at two different levels. At the practical level, the goal is to build a programming language that enables language-oriented software design. This language must facilitate easy creation of eDSLs, immediate development of components in these newly created languages, and integration of components in distinct eDSLs; Racket is available at http://racket-lang.org/

At the conceptual level, the case for LOP is analogous to the ones for object-oriented programming and for concurrency-oriented programming.[3] The

a The numerous language-like libraries in scripting languages (such as JavaScript, Python, and Ruby), books (such as Fowler and Parson),[20] and websites (such as Federico Tomassetti's, https://tomassetti.me/resources-create-programming-languages/) are evidence of the desire by programmers to use and develop eDSLs.

ILLUSTRATION BY CHRIS LABROOY

**Figure 1. Small language-oriented programming example.**

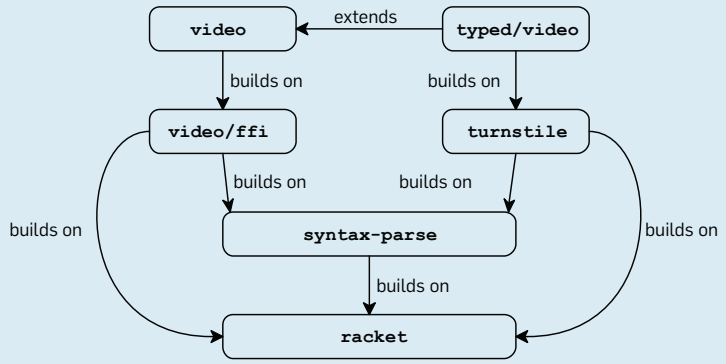



**Figure 2. A plain Racket module.**

demo

```
#lang racket/base
(provide
  ;; type MaxPath = [Listof Edge]
  ;; Natural -> MaxPath
  walk-simplex)

(require "constraints" graph)

;; Natural -> MaxPath
(define (walk-simplex timing)
  ··· (maximizer #:x 2)···)
```

**Figure 3. A module for describing a simplex shape.**

constraints

```
#lang simplex

;; implicitly provides synthesized function maximizer:
;;    #:x Real -> Real
;;    #:y Real -> Real

#:variables x y

3 * x + 5 * y <= 10
3 * x - 5 * y <= 20
```

**Figure 4. Lambda, redefined.**

new-lam

```
01 #lang racket
02
03 (provide (rename-out [new-lambda lambda]))
04
05 (require (for-syntax syntax/parse))
06 ...
07 ;; Syntax -> Syntax
08 (define-syntax (new-lambda stx)
09   (syntax-parse stx
10     [(new-lambda (x:id (~literal ::) predicate:id) body:expr)
11      (syntax
12       (lambda (x)
13         (unless (predicate x)
14           (define name (object-name predicate))
15           (error 'lambda "~a expected, given: ~e" name x))
16         body))]))
17 ...
```

former arose from making the creation and manipulation of objects syntactically simple and dynamically cheap, the latter from Erlang's inexpensive process creation and message passing. Both innovations enabled new ways to develop software and triggered research projects. The question is how our discipline will realize LOP and how it will affect the world of software.

Our decision to develop a new language—Racket—is partly an historical artifact and partly due to our desire to free ourselves from any unnecessary constraints of industrial mainstream languages as we investigate LOP. The next section spells out how Racket got started, how we honed in on LOP, and what the idea of LOP implies.

## Principles of Racket

The Racket project dates to January 1995 when we started it as a language for experimenting with pedagogic programming languages.[15] Working on them quickly taught us that a language itself is a problem-solving tool. We soon found ourselves developing different languages for different parts of the project: a (meta-) language for expressing many pedagogic languages, another for specializing the DrRacket IDE,[15] and a third for managing configurations. In the end, the software was a multilingual system, as outlined earlier.

Racket's guiding principle reflects the insight we gained: empower programmers to create new programming languages easily and add them with a friction-free process to a codebase. By "language," we mean a new syntax, a static semantics, and a dynamic semantics that usually maps the new syntax to elements of the host language and possibly external languages via a foreign-function interface (FFI). For a concrete example, see Figure 1 for a diagram of the architecture of a recently developed pair of scripting languages for video editing[2,b] designed to assist people who turn recordings of conference presentations into YouTube videos and channels. Most of that work is

b The `video` language, including an overview of the implementation, is available as a use-case artifact at https://www2.ccs.neu.edu/racket/pubs/#icfp17-acf

repetitive—adding preludes and postludes, concatenating playlists, and superimposing audio—with few steps demanding manual intervention. This task calls for a domain-specific scripting language; video is a declarative eDSL that meets this need.

The `typed/video` language adds a type system to `video`. Clearly, the domain of type systems comes with its own language of expertise, and `typed/video`'s implementation thus uses `turnstile`,[6] an eDSL created for expressing type systems. Likewise, the implementation of `video`'s rendering facility calls for bindings to a multimedia framework. Ours separates the binding definitions from the repetitive details of FFI calls, yielding two parts: an eDSL for multimedia FFIs, dubbed `video/ffi`, and a single program in the eDSL. Finally, in support of creating all these eDSLs, Racket comes with the `syntax parse` eDSL,[7] which targets eDSL creation.

The LOP principle implies two subsidiary guidelines:

*Enable creators of a language to enforce its invariants.* A programming language is an abstraction, and abstractions are about integrity. Java, for example, comes with memory safety and type soundness. When a program consists of pieces in different languages, values flow from one context into another and need protection from operations that might violate their integrity, as we discuss later; and

*Turn extra-linguistic mechanisms into linguistic constructs.* A LOP programmer who resorts to extra-linguistic mechanisms effectively acknowledges that the chosen language lacks expressive power.[13,c] The numerous external languages required to deal with Java projects—a configuration language, a project description language, and a `makefile` language—represent symptoms of this problem. We treat such gaps as challenges later in the article.

They have been developed in a feedback loop that includes DrRacket[15] plus typed,[36] lazy,[4] and pedagogical languages.[15]

c Like many programming-language researchers, we subscribe to a weak form of the Sapir-Whorf hypothesis; see http://docs.racket-lang.org/algol60/ and https://www.hashcollision.org/brainfudge/ showing how Racket copes with obscure syntax.

**Most notably, Racket eliminates the hard boundary between library and language, overcoming a seemingly intractable conflict.**

### Libraries and Languages Reconciled

Racket is an heir of Lisp and Scheme. Unlike these ancestors, however, Racket emphasizes functional over imperative programming without enforcing an ideology. Racket is agnostic when it comes to surface syntax, accommodating even conventional variants (such as Algol 60).[d] Like many languages, Racket comes with "batteries included."

Most notably, Racket eliminates the hard boundary between library and language, overcoming a seemingly intractable conflict. In practice, this means new linguistic constructs are as seamlessly imported as functions and classes from libraries and packages. For example, Racket's class system and `for` loops are imports from plain libraries, yet most programmers use these constructs without ever noticing their nature as user-defined concepts.

Racket's key innovation is a modular syntax system,[17,26] an improvement over Scheme's macro system,[11,24,25] which in turn improved on Lisp's tree-transformation system. A Racket module provides such services as functions, classes, and linguistic constructs. To implement them, a module may require the services of other modules. In this world of modules, creating a new language means simply creating a module that provides the services for a language. Such a module may subtract linguistic constructs from a base language, reinterpret others, and add a few new ones. A language is rarely built from scratch.

Like Unix shell scripts, which specify their dialect on the first line, every Racket module specifies its language on the first line, too. This language specification refers to a file that contains a language-defining module. Creating this file is all it takes to install a language built in Racket. Practically speaking, a programmer may develop a language in one tab of the IDE, while another tab may be a module written in the language of the first. Without ever leaving the IDE to run compilers, linkers, or other tools, the developer can modify the language implementation in the first tab and immediately experience the modification in the second;

d See http://docs.racket-lang.org/algol60/, as well as well as https://www.hashcollision.org/brainfudge/, which shows how Racket copes with obscure syntax.

that is, language development is a friction-free process in Racket.

In the world of shell scripts, the first-line convention eventually opened the door to a slew of alternatives to shells, including Perl, Python, and Ruby. The Racket world today reflects a similar phenomenon, with language libraries proliferating within its ecosystem: `racket/base`, the Racket core language; `racket`, the "batteries included" variant; and `typed/racket`, a typed variant. Some lesser-known examples are `datalog` and a `web-server` language.[27,30] When precision is needed, we use the lowercase name of the language in typewriter font; otherwise we use just "Racket."

Figure 2 is an illustrative module. Its first line—pronounced "hash lang racket base"—says it is written in `racket/base`. The module provides a single function, `walk-simplex`. The accompanying line comments—introduced with semicolons—informally state a type definition and a function signature in terms of this type definition; later, we show how developers can use `typed/racket` to replace such comments with statically checked types, as in Figure 5. To implement this function, the module imports functionality from the `constraints` module outlined in Figure 3. The last three lines of Figure 2 sketch the definition of the `walk-simplex` function, which refers to the `maximizer` function imported from constraints.

The `"constraints"` module in Figure 3 expresses the implementation of its only service in a domain-specific language because it deals with simplexes, which are naturally expressed through a system of inequalities. The module's `simplex` language inherits the line-comment syntax from `racket/base` but uses infix syntax otherwise. As the comments state, the module exports a single function, `maximizer`, which consumes two optional keyword parameters. When called as (`maximizer #:x n`), as in Figure 2, it produces the maximal `y` value of the system of constraints. As in the lower half of Figure 3, these constraints are specified with conventional syntax.

In support of this kind of programming, Racket's modular syntax system benefits from several key innovations. A particularly illustrative one is the ability to incrementally redefine the meaning of existing language constructs via the module system. It allows eDSL creators to ease their users into a new language by reusing familiar syntax, but reinterpreted.

**In general, cooperating multilingual components must respect the invariants established by each participating language.**

Consider `lambda` expressions, for example. Suppose a developer wishes to equip a scripting language (such as `video`) with functions that check whether their arguments satisfy specified predicates. Figure 4 shows the basic idea:

*line 01* The module uses the `racket` language.

*line 03* It exports a defined compile-time function, `new-lambda`, under the name `lambda`, which is overlined in the code to mark its origin as this module.

*line 05* Here, the module imports tools from a library for creating robust compile-time functions conveniently.[7]

*line 07* The comment says a function on syntax trees follows.

*line 08* While (`define (f x) . . .`) introduces an ordinary function `f` of `x`, (`define-syntax (c stx) . . .`) creates the compile-time function `c` with a single argument, `stx`.

*line 09* As with many functional languages, Racket comes with pattern-matching constructs. This one uses `syntax-parse` from the library mentioned earlier. Its first piece specifies the to-be-matched tree (`stx`); the remainder specifies a series of pattern-responses clauses.

*line 10* This pattern matches any syntax tree with first token as `new-lambda` followed by a parameter specification and a body. The annotation `:id` demands that the pattern variables `x` and `predicate` match only identifiers in the respective positions. Likewise, `:expr` allows only expressions to match the body pattern variable.

*line 11* A compile-time function synthesizes new trees with `syntax`.

*line 12* The generated syntax tree is a `lambda` expression. Specifically, the function generates an expression that uses `lambda`. The underline in the code marks its origin as the ambient language, here `racket`.

*other lines* Wherever the syntax system encounters the pattern variables `x`, `predicate`, and `body`, it inserts the respective subtrees that match `x`, predicate, and body.

When another module uses `"new-lam"` as its language, the compiler

elaborates the surface syntax into the core language like this

```
(lambda (x :: integer?) (+ x 1))
–elaborates to⟶
lambda (x :: integer?) (+ x 1))
–elaborates to⟶
(new-lambda (x :: integer?) (+ x 1))
–elaborates to⟶
(lambda (x)
  (unless (integer? x)
    <elided error reporting>)
  (+ x 1))
```

The first elaboration step resolves `lambda` to its imported meaning,[18] or `lambda`. The second reverses the "rename on export" instruction. Finally, the `new-lambda` compile-time function translates the given syntax tree into a `racket` function.

In essence, Figure 4 implements a simplistic precondition system for one-argument functions. Next, the language developer might wish to introduce multiargument `lambda` expressions, add a position for specifying the post-condition, or make the annotations optional. Naturally, the compile-time functions could then be modified to check some or all of these annotations statically, eventually resulting in a language that resembles `typed/racket`.

## Sound Cooperation Between Languages

A LOP-based software system consists of multiple cooperating components, each written in domain-specific languages. Cooperation means the components exchange values, while "multiple languages" implies these values are created in distinct languages. In this setting, things can easily go wrong, as demonstrated in Figure 5 with a toy scenario. On the left, a module written in `typed/racket` exports a numeric differentiation function. On the right, a module written in `racket` imports this function and applies it in three different ways, all illegal. If such illegal uses of the function were to go undiscovered, developers would not be able to rely on type information for designing functions or for debugging, nor could compilers rely on them for optimizations. In general, cooperating multilingual components must respect the invariants established by each participating language.

In the real world, programming languages satisfy a spectrum of guarantees about invariants. For example, C++ is unsound. A running C++ program may apply any operation to any bit pattern and, as long as the hardware does not object, program execution continues. The program may even terminate "normally," printing all kinds of output after the misinterpretation of the bits. In contrast, Java does not allow the misrepresentation of bits but is only somewhat more sound than C++.[1] ML improves on Java again and is completely sound, with no value ever manipulated by an inappropriate operation.

Racket aims to mirror this spectrum of soundness at two levels: language implementation itself and cooperation between two components written in different embedded languages. First consider the soundness of languages. As the literature on domain-specific languages suggests,[20] such languages normally evolve in a particular manner, as is true for the Racket world, as in Figure 6. A first implementation is often a thin veneer over an efficient C-level API. Racket developers create such a veneer with a foreign interface that allows parenthesized C-level programming.[5] Programmers can refer to a C library, import functions and data structures, and wrap these imports in Racket values. Figure 7 illustrates the idea with a sketch of a module; `video`'s initial implementation consisted of just such a set of bindings to a video-rendering framework. When a `racket/base` module imports the `ffi/unsafe` library, the language of the module is unsound.

A language developer who starts with an unsound eDSL is likely to make it sound as the immediate next step. To this end, the language is equipped with runtime checks similar to those found in dynamically typed scripting languages to prevent the flow of bad values to unsound primitives. Unfortunately, such protection is ad hoc, and, unless developers are hypersensitive, the error messages may originate from inside the library, thus blaming some `racket/base` primitive operation for the error. To address this problem, Racket comes with higher-order contracts[16] with which a language developer might uniformly protect the API of a library from bad values. For example, the `video/ffi` language provides language constructs for making the bindings to the video-rendering framework safe. In addition to plain logical assertions, Racket's developers are also experimenting with contracts for checking protocols, especially temporal ones.[9] The built-in blame mechanism of the contract library ensures sound blame assignment.[10]

Finally, a language developer may wish to check some logical invariants *before* the programs run. Checking simple types is one example, though

**Figure 5. Protecting invariants.**

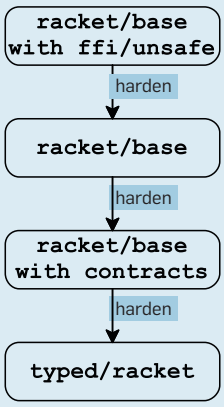
TR

```
#lang typed/racket

(provide diff)

(: diff
   ((Real -> Real)
    ->
    (Real -> Real)))
(define (diff f)
  (lambda (x)
    (define lo (f (- x eps)))
    (define hi (f (+ x eps)))
    (/ (- hi lo)
       (* 2 eps))))
```

RR

```
#lang racket

(require "TR.rkt")

;; scenario 1
(diff 1)
;; scenario 2
(define (f-bool x)
  #true)
(diff f-bool)
;; scenario 3
(define (f-char x)
  (string x x))
(diff f-str)
```

**Figure 6. Hardening a module.**



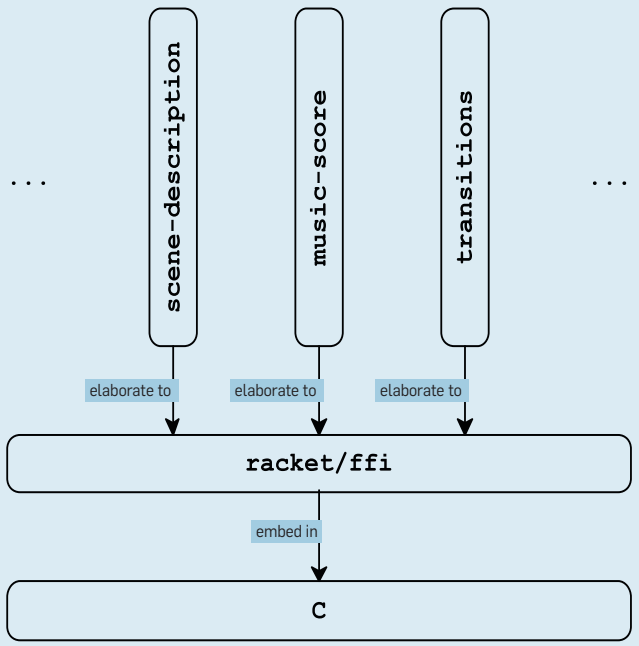
other forms of static checking are also possible. The `typed/video` language illustrates this point with a type system that checks the input and output types of functions that may include numeric constraints on the integer arguments; as a result, no script can possibly render a video of negative length. Likewise, `typed/racket` is a typed variant of (most of) racket.

Now consider the soundness of cooperating languages. It is again up to the language developer to anticipate how programs in this language interact with others. For example, the creator of `typed/video` provides no protection for its programs. In contrast, the creators of `typed/racket` intended the language to be used in a multilingual context; `typed/racket` thus compiles the types of exported functions into the higher-order contracts mentioned. When, for example, an exported function must always be applied to integer values, the generated contract inserts a check that ensures the "integerness" of the argument at every application site for this function; there is no need to insert such a check for the function's return points because the function is statically type checked. For a function that consumes an integer-valued function, the contract must ensure the function argument always returns an integer. In general, a contract wraps exported values with a proxy[31] that controls access to the value. The idea is due to Matthews and Findler,[29] while Tobin-Hochstadt's and Felleisen's Blame Theorem[35] showed that if something goes wrong with such a mixed system, the runtime exception points to two faulty components and their boundary as the source of the problem.[10] In general, Racket supplies a range of protection mechanisms, and a language creator can use them to implement a range of soundness guarantees for cooperating eDSLs.

**Figure 7. A Racket module using the foreign-function interface.**

```
#lang racket/base                                              ffi

(provide
 ;; [Vectorof [Vectorof Real]] -> [Vectorof Real]
 simplex)

(require ffi/unsafe)

(define (simplex M)
  ··· (ffi-simplex-set ···) ···)

(define lib-simplex (ffi-lib "./coin-Clp/lib/libClp"))

(define ffi-simplex-set
  (get-ffi-obj "simplex" lib-simplex (_fun _bytes -> _void)))
```

**Figure 8. A sketch of an industrial example of language-oriented programming.**




## Universality vs. Expressiveness

Just because a general-purpose language can compute all partial-recursive functions, programmers cannot necessarily express all their ideas about programs in this language.[13] This point is best illustrated through an example. So, imagine the challenge of building an IDE for a new programming language in the very same language. Like any modern IDE, it is supposed to enable users to compile and run their code. If the code goes into an infinite loop, the user must be able to terminate it with a simple mouse click. To implement this capability in a natural[e] manner, the language must internalize the idea of a controllable process, a thread. If it does not internalize such a notion, the implementer of the IDE must step outside the language and somehow re-use processes from the underlying operating system.

For a programming language researcher, "stepping outside the language" signals failure. Or, as Ingalls[21] said, "[an] operating system is a collection of things that don't fit into a language[; t]here shouldn't be one." We, Racket creators, have sought to identify services Racket borrows from the surrounding operating system and assimilate them into the language itself.[19] Here are three sample constructs for which programmers used to step outside of Racket but no longer need to:

*Sandboxes.* That restrict access to resources;

*Inspectors.* That control reflective capabilities; and

e An alternative is to rewrite the entire program before handing it to the given compiler, exactly what distinguishes "expressiveness" from "universality."

*Custodians.* That manage resources (such as threads and sockets).

To understand how inclusion of such services helps language designers, consider a 2014 example, the `shill` language.[32] Roughly speaking, `shill` is a secure scripting language in Racket's ecosystem. With `shill`, a developer articulates fine-grain security and resource policies—along with, say, what files a function may access or what binaries the script may run—and the language ensures these constraints are satisfied. To make this concrete, consider a homework server to which students can submit their programs. The instructor might wish to run an autograde process for all submissions. Using a `shill` script, the homework server can execute student programs that cannot successfully attack the server, poke around in the file system for solutions, or access external connections to steal other students' solutions. Naturally, `shill`'s implementation makes extensive use of Racket's means of running code in sandboxes and harvesting resources via custodians.

**Racket borrows from the surrounding operating system and assimilates such extra-linguistic mechanisms into the language itself.**

## State of Affairs

The preceding sections explained how Racket enables programmers to do the following:

*Create languages.* Create by way of linguistic reuse for specific tasks and aspects of a problem;

*Equip with soundness.* Equip a language with almost any conventional level of soundness, as found in ordinary language implementations; and

*Exploit services.* Exploit a variety of internalized operating system services for constructing runtime libraries for these embedded languages.

What makes such language-oriented programming work is "incrementality," or the ability to develop languages in small pieces, step by step. If conventional syntax is not a concern, developers can create new languages from old ones, one construct at a time. Likewise, they do not have to deliver a sound and secure product all at once; they can thus create a new language as a wrapper around, say, an existing C-level library, gradually tease out more of the language from the interface, and make the language as sound or secure as time permits or a growing user base demands.

Moreover, the entire process takes place within the Racket ecosystem. A developer creates a language as a Racket module and installs it by "importing" it into another module. This tight coupling has two implications: the development tools of the ecosystem can be used for creating language modules and their clients; and the language becomes available for creating more languages. Large projects often employ a tower involving a few dozen languages, all helping manage the daunting complexity in modern software systems.

Sony's *Naughty Dog* game studio has created just such a large project, actually a framework for creating projects. Roughly speaking, Sony's Racket-based architecture provides languages for describing scenes, transitions between scenes, scores for scenes, and more. Domain specialists use the languages to describe aspects of the game. The Racket implementation composes these domain-specific programs, then compiles them into dynamically linked libraries for a C-based game engine; Figure 8 sketches the arrangement graphically.

Racket's approach to language-oriented programming is by no means perfect. To start with, recognizing when a library should become a language requires a discriminating judgment call. The next steps require good choices in terms of linguistic constructs, syntax, and runtime primitives.

As for concrete syntax, Racket currently has strong support for typical, incremental Lisp-style syntax development, including traditional support for conventional syntax, or generating lexers and parsers. While traditional parsing introduces the natural separation between surface syntax and meaning mentioned earlier, it also means the development process is no longer incremental. The proper solution would be to inject Racket ideas into a context where conventional syntax is the default.[f]

f Language workbenches (such as Spoofax[22]) deal with conventional syntax for DSLs but do not support the incremental modification of existing languages. A 2015 report[12] suggests, however, these tool chains are also converging toward the idea of language creation as language modification. We conjecture that, given sufficient time, development of Racket and language workbenches will converge on similar designs.

As for static checking, Racket forces language designers to develop such checkers wholesale, not incrementally. The type checker for `typed/racket` looks like, for example, the type checker for any conventionally typed language; it is a complete recursive-descent algorithm that traverses the module's representation and algebraically checks types. What Racket developers really want is a way to attach type-checking rules to linguistic constructs, so such algorithms can be synthesized as needed.

Chang et al.[6] probably took a first step toward a solution for this problem and have thus far demonstrated how their approach can equip a DSL with any structural type system in an incremental and modular manner. A fully general solution must also cope with substructural type systems (such as the Rust programming language) and static program analyses (such as those found in most compilers).

As for dynamic checking, Racket suffers from two notable limitations: On one hand, it provides the building blocks for making language cooperation sound, but developers must create the necessary soundness harnesses on an ad hoc basis. To facilitate the composition of components in different languages, Racket developers need both a theoretical framework and abstractions for the partial automation of this task. On the other hand, the available spectrum of soundness mechanisms lacks power at both ends, and how to integrate these powers seamlessly is unclear. To achieve full control over its context, Racket probably needs access to assembly languages on all possible platforms, from hardware to browsers. To realize the full power of types, `typed/racket` will have to be equipped with dependent types. For example, when a Racket program uses vectors, its corresponding typed variant type-checks what goes into them and what comes out, but like ML or Haskell, indexing is left to a (contractual) check in the runtime system. Tobin-Hochstadt and his Typed Racket group are working on first steps in this direction, focusing on numeric constraints,[23] similar to Xi's and Pfenning's research.[37]

As for security, the Racket project is still looking for a significant breakthrough. While the `shill` team was able to construct the language inside the Racket ecosystem, its work exposed serious gaps between Racket's principle of language-oriented programming and its approach to enforcing security policies. It thus had to alter many of Racket's security mechanisms and invent new ones. Racket must clearly make this step much easier, meaning more research is needed to turn security into an integral part of language creation.

**To achieve full control over its context, Racket probably needs access to assembly languages on all possible platforms, from hardware to browsers.**

Finally, LOP also poses brand-new challenges for tool builders. An IDE typically provides tools for a single programming language or a family of related languages, including debuggers, tracers, and profilers. Good tools communicate with developers in terms of the source language. Due to its very nature, LOP calls for customization of such tools to many languages, along with their abstractions and invariants. We have partially succeeded in building a tool for debugging programs in the syntax language,[8] have the foundations of a debugging framework,[28] and started to explore how to infer scoping rules and high-level semantics for newly introduced, language-level abstractions.[33,34] Customizing these tools automatically to newly created (combinations of) languages remains an open challenge.

## Conclusion

Programming language research is short of its ultimate goal—provide software developers tools for formulating solutions in the languages of problem domains. Racket is one attempt to continue the search for proper linguistic abstractions. While it has achieved remarkable success in this direction, it also shows that programming-language research has many problems to address before the vision of language-oriented programming becomes reality.

## Acknowledgments

We thank Claire Alvis, Robert Cartwright, Ryan Culpepper, John Clements, Stephen Chang, Richard Cobbe, Greg Cooper, Christos Dimoulas, Bruce Duba, Carl Eastlund, Burke Fetscher, Cormac Flanagan, Kathi Fisler, Dan Friedman, Tony Garnock

Jones, Paul Graunke, Dan Grossman, Kathy Gray, Casey Klein, Eugene Kohlbecker, Guillaume Marceau, Jacob Matthews, Scott Owens, Greg Pettyjohn, Jon Rafkind, Vincent St-Amour, Paul Steckler, Stevie Strickland, James Swaine, Asumu Takikawa, Kevin Tew, Neil Toronto, and Adam Wick for their contributions.

A preliminary version of this article appeared in the *Proceedings of the First Summit on Advances in Programming Languages* conference in 2015.[14] In addition to its reviewers, Sam Caldwell, Eduardo Cavazos, John Clements, Byron Davies, Ben Greenman, Greg Hendershott, Manos Renieris, Marc Smith, Vincent St-Amour, and Asumu Takikawa suggested improvements to the presentation of this material. The anonymous *Communications* reviewers challenged several aspects of our original submission and thus forced us to greatly improve the exposition.

Since the mid-1990s, this work has been generously supported by our host institutions—Rice University, University of Utah, Brown University, University of Chicago, Northeastern University, Northwestern University, Brigham Young University, University of Massachusetts Lowell, and Indiana University—as well as a number of funding agencies, foundations, and companies, including the Air Force Office of Scientific Research, Cisco Systems Inc., the Center for Occupational Research and Development, the Defense Advanced Research Projects Agency, the U.S. Department of Education's Fund for the Improvement of Postsecondary Education, the ExxonMobil Foundation, Microsoft, the Mozilla Foundation, the National Science Foundation, and the Texas Advanced Technology Program. C

**References**

1. Amin, N. and Tate, R. Java and Scala's type systems are unsound: The existential crisis of null pointers. In *Proceedings of ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages & Applications*, 2016, 838–848.
2. Andersen, L., Chang, S., and Felleisen, M. Super 8 languages for making movies. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2017, 1–29.
3. Armstrong, J. Concurrency-oriented programming. In *Frühjahrsfachgespräch der German Unix User Group*, 2003; http://guug.de/veranstaltungen/ffg2003/papers/
4. Barzilay, E. and Clements, J. Laziness without all the hard work. In *Proceedings of the Workshop on Functional and Declarative Programming in Education*, 2005, 9–13.
5. Barzilay, E. and Orlovsky, D. Foreign interface for PLT Scheme. In *Proceedings of the Ninth ACM SIGPLAN Workshop on Scheme and Functional Programming*, 2004, 63–74.
6. Chang, S., Knauth, A., and Greenman, B. Type systems as macros. In *Proceedings of the 44th ACM SIGPLAN Principles of Programming Languages*, 2017, 694–705.
7. Culpepper, R. Fortifying macros. *Journal of Functional Programming 22*, 4–5 (Aug. 2012), 439–476.
8. Culpepper, R. and Felleisen, M. Debugging macros. *Science of Computer Programming 75*, 7 (July 2010), 496–515.
9. Dimoulas, C., New, M., Findler, R., and Felleisen, M. Oh Lord, please don't let contracts be misunderstood. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2016, 117–131.
10. Dimoulas, C., Tobin-Hochstadt, S., and Felleisen, M. Complete monitors for behavioral contracts. In *Proceedings of the European Symposium on Programming*, 2012, 214–233.
11. Dybvig, R., Hieb, R., and Bruggeman, C. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation 5*, 4 (Dec. 1993), 295–326.
12. Erdweg, S., van der Storm, T., Vlter, M., Tratt, L., Bosman, R., Cook, W.R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P.J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., and van derWoning, J. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems and Structures 44*, Part A (Dec. 2015), 24–47.
13. Felleisen, M. On the expressive power of programming languages. *Science of Computer Programming 17*, 1–3 (Dec. 1991), 35–75.
14. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., and Tobin-Hochstadt, S. The Racket Manifesto. In *Proceedings of the First Summit on Advances in Programming Languages*, T. Ball, R. Bodik, S. Krishnamurthi, B.S. Lerner, and G. Morrisett, Eds. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2015, 113–128.
15. Findler, R., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., and Felleisen, M. DrScheme: A programming environment for Scheme. *Journal of Functional Programming 12*, 2 (Mar. 2002), 159–182.
16. Findler, R.B. and Felleisen, M. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, 2002, 48–59.
17. Flatt, M. Composable and compilable macros: You want it when? In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, 2002, 72–83.
18. Flatt, M. Bindings as sets of scopes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, 705–717.
19. Flatt, M., Findler, R.B., Krishnamurthi, S., and Felleisen, M. Programming languages as operating systems (or revenge of the son of the Lisp machine). In *Proceedings of the International Conference on Functional Programming*, 1999, 138–147.
20. Fowler, M. and Parsons, R. *Domain-Specific Languages*. Addison-Wesley, Boston, MA, 2010.
21. Ingalls, D.H. Design principles behind Smalltalk. *Byte Magazine 6*, 8 (Aug. 1981), 286–298.
22. Kats, L.C.L. and Visser, E. The Spoofax language workbench. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 2010, 444–463.
23. Kent, A.M., Kempe, D., and Tobin-Hochstadt, S. Occurrence typing modulo theories. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, 296–309.
24. Kohlbecker, E.E., Friedman, D.P., Felleisen, M., and Duba, B.F. Hygienic macro expansion. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1986, 151–161.
25. Kohlbecker, E.E. and Wand, M. Macros-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the 14th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1987, 77–84.
26. Krishnamurthi, S. *Linguistic Reuse*. Ph.D. Thesis, Rice University, Houston, TX, 2001; https://www2.ccs.neu.edu/racket/pubs/#thesis-shriram
27. Krishnamurthi, S., Hopkins, P.W., McCarthy, J., Graunke, P.T., Pettyjohn, G., and Felleisen, M. Implementation and use of the PLT Scheme Web server. *Higher-Order and Symbolic Computation 20*, 4 (Apr. 2007), 431–460.
28. Marceau, G., Cooper, G.H., Spiro, J.P., Krishnamurthi, S., and Reiss, S.P. The design and implementation of a dataflow language for scriptable debugging. In *Proceedings of the Annual ACM SIGCSE Technical Symposium on Computer Science Education*, 2007, 59–86.
29. Matthews, J. and Findler, R.B. Operational semantics for Multilanguage programs. *ACM Transactions on Programming Languages and Systems 31*, 3 (Apr. 2009), 1–44.
30. McCarthy, J. The two-state solution. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 2010, 567–582.
31. Miller, M.S. *Robust Composition: Towards a United Approach to Access Control and Concurrency Control*. Ph.D. Thesis, Johns Hopkins University, Baltimore, MD, May 2006; http://www.erights.org/talks/thesis/
32. Moore, S., Dimoulas, C., King, D., and Chong, S. `Shill`: A secure shell scripting language. In *Proceedings of the Conference on Operating Systems Design and Implementation*, 2014, 183–199.
33. Pombrio, J. and Krishnamurthi, S. Resugaring: Lifting evaluation sequences through syntactic sugar. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2014, 361–371.
34. Pombrio, J., Krishnamurthi, S., and Wand, M. Inferring scope through syntactic sugar. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2017, 1–28.
35. Tobin-Hochstadt, S. and Felleisen, M. Interlanguage migration: From scripts to programs. In *Proceedings of the ACM SIGPLAN Dynamic Language Symposium*, 2006, 964–974.
36. Tobin-Hochstadt, S. and Felleisen, M. The design and implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Conference on the Principles of Programming Languages*, 2008, 395–406.
37. Xi, H. and Pfenning, F. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998, 249–257.

**Matthias Felleisen** (matthias@ccs.neu.edu) is a Trustee Professor in the College of Computer Science at Northeastern University, Boston, MA, USA.

**Robert Bruce Findler** (robby@eecs.northwestern.edu) is a professor of computer science at Northwestern University, Evanston, IL, USA.

**Matthew Flatt** (mflatt@cs.utah.edu) is a professor of computer science at the University of Utah, Salt Lake City, UT, USA.

**Shriram Krishnamurthi** (sk@cs.brown.edu) is a professor of computer science at Brown University, Providence, RI, USA.

**Eli Barzilay** (eli@barzilay.org) is a research scientist at Microsoft Research, Cambridge, MA, USA.

**Jay McCarthy** (jay.mccarthy@gmail.com) is an associate professor of computer science at the University of Massachusetts, Lowell, MA, USA.

**Sam Tobin-Hochstadt** (samth@cs.indiana.edu) is an assistant professor of computer science at Indiana University, Bloomington, IN, USA.



Watch the authors discuss their work in this exclusive *Communications* video. https://cacm.acm.org/videos/a-programmable-programming-language