THE RACKET MANIFESTO

MATTHIAS, RACKETEER

Robby Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, Sam Tobin-Hochstadt
SNAPL 2015
ccs.neu.edu/matthias/manifesto/
(define UFO)

(big-bang 0
  (to-draw (lambda (y) (place-image UFO 100 y (empty-scene 200 200))))
  (on-tick add1)
  (stop-when (lambda (y) (> y 180))))
Composable and Compilable Macros

You Want it When?

Matthew Flatt
University of Utah

Abstract

Many macro systems, especially for Lisp and Scheme, allow macro transformers to perform general computation. Moreover, the language for implementing compile-time macro transformers is usually the same as the language for implementing run-time functions. As a side effect of this sharing, implementations tend to allow the mingling of compile-time values and run-time values, as well as values from separate compilations. Such mingling breaks programming tools that must parse code without executing it. Macro implementors avoid harmful mingling by obeying certain macro-definition protocols and by inserting phase-distinguishing annotations into the code. However, the annotations are fragile, the protocols are not enforced, and programmers can only reason about the result in terms of the compiler’s implementation. M$\$cheme—the language of the PLT Scheme tool suite—addresses the problem through a macro system that separates compilation without sacrificing the expressiveness of macros.

Categories and Subject Descriptors

D.3.3 [Software]: Programming Languages—language constructs and features; Scheme; D.3.4 [Software]: Processors—parsing, pre-processors; D.2.12 [Software Engineering]: Interoperability

General Terms

Languages, Design

pattern-matching transformations, but may perform arbitrary computation during expansion [12, 17, 3, 24, 28, 6, 1]. In addition, macros may manipulate abstract syntax enriched with lexical information instead of manipulating raw source text [5, 2, 8, 48], which means that macro-defined constructs can be assigned a meaning independent of details of the macro’s expansion (e.g., whether the macro introduces a local variable named temp or happens to call the car function). Finally, in the Lisp and Scheme tradition where macros are themselves defined in a macro-extensible language, extensions can be stacked in a “language tower.” Each extension of the language can be used in implementing the next extension.

Trouble with Expressive Macro Systems. In a typical Scheme system, however, language towers cause trouble [19]. Advances in macro technology have simplified the creation of individual blocks for a tower, but they have not delivered a reliable mortar for assembling the blocks. For example, suppose “P.scm” is implemented in an extension of Scheme $E$, where $E$ is implemented by “E.scm” directly in Scheme. A typical load sequence for $P$ is:

```
(load "E.scm")
(load "P.scm")
```

The above statements might be placed in a file “loadP.scm,” which can then be submitted to a Scheme interpreter to execute “P.scm” successfully. The problem starts when the programmer tries to coodle the program for later execution. Supplying “loadP.scm” to the compiler is useless, because the result is simply the compiled form of two load statements. A full compiler will be needed at run-time when “P.scm” is actually loaded.
Composable and Compilable Macros

You Want it When?

Matthew Flatt
University of Utah


General Terms
Languages, Design

Pattern matching transformations, but may perform arbitrary computation during expansion [12, 17, 3, 24, 26, 1]. In addition, macros may manipulate abstract syntax enriched with lexical information instead of manipulating raw source text [15, 2, 4, 8], which means that macro-defined constructs can be assigned a meaning independent of details of the macro's expansion (e.g., whether the macro introduces a local variable named 'aexp' or happens to call the car function). Finally, in the Lisp and Scheme tradition where macros are themselves defined in a macro-extendable language, extensions can be stacked in a "language tower." Each extension of the language can be used in implementing the next extension.

Trouble with Expressive Macro Systems. In a typical Scheme system, however, language towers cause trouble [19]. Advances in macro technology have simplified the creation of individual blocks for a tower, but they have not delivered a reliable mortar for assembling the blocks. For example, suppose "P.scm" is implemented in an extension of Scheme R, where R is implemented by "E.scm" directly in Scheme. A typical load sequence for P is:

(load "E.scm")
(load "P.scm")

The above statements might be placed in a file "loadP.scm," which can then be submitted to a Scheme interpreter to execute "P.scm" successfully. The problem starts when the programmer tries to compile the program for later execution. Suppling "loadP.scm" to the compiler is useless, because the result is simply the compiled form of two load statements. A full compiler will be needed at run-time when "P.scm" is actually loaded.
Composable and Compilable Macros

You Want it When?

Matthew Flatt
University of Utah

...
Racket Lives in Three Worlds: Education, Research & Industry
On 26 January 1995, Racket was a …
On 26 January 1995, Racket was a …
On 26 January 1995, Racket was a …
On 26 January 1995, Racket was a …

… and then,

a eureka!

a deep thought
On 26 January 1995, Racket was a …

… and then,
So, what is Racket?
So, what is Racket?

Haskell is a purely functional, lazy language.
So, what is Racket?

Haskell is a purely functional, lazy language.

Python is about the one way, the obvious way.
So, what is Racket?

Haskell is a purely functional, lazy language.

Python is about the one way, the obvious way.

Is there a one-liner for Racket?
Racket is a programming language
Racket

... is a full spectrum programming language
Racket

... is a programming language

... is a full spectrum programming language

... is a linguist's language
... is a full spectrum programming language

Racket

... is a linguist’s language

... is a programming language programming language
RACKET IS A PROGRAMMING-LANGUAGE PROGRAMMING LANGUAGE
The next 700 languages?
The next 7,000 languages?
The next 70,000 languages?
The next 700,000 languages?
The next 700 languages?
The next 7,000 languages?
The next 70,000 languages?
The next 700,000 languages?

Why many languages? Isn’t Racket enough?
Imagine Playstation Game Programming
Imagine Playstation Game Programming

Problems to be solved:
— physics
— scenery
— transitions
— music score
— …
Problems to be solved:
— physics
— scenery
— transitions
— music score
— …

Features supported:
— for and while loops
— methods
— classes
— packages
…
People don’t speak one English. They speak many.
execute
accounts
technology
operations
information
legalese
The diagram illustrates the flow of information across different departments:

- **Operations**
- **Information**
- **Technology**
- **Legalese**
- **Financials**

Each department is connected to English, indicating the flow of information in that language. The financials section is highlighted with a green circle, suggesting its importance or unique role in the flow.
Embedded DSL

“Fluent” Interfaces

Racket
Embedded DSL

“Fluent” Interfaces

Racket

“Hard” Interfaces

“Little Language” DSL
We want to be able to *build* all these things and we want to be able to *compose* programs in these “DSLs”.
How do you build these “DSLs”?  
(compilation)

How do you compose programs in these “DSLs”?  
(run time)
How do you build these “DSLs”?
(compilation)
Hygienic Macros!
We will transfer $1,000 to your account, nicknamed ____, on 18 July 2016.
We will transfer $1,000 to your account, nicknamed ____ , on 18 July 2016.
We will transfer $1,000 to your account, nicknamed ____ , on 18 July 2016.
We will transfer $1,000 to your account, nicknamed ____ , on 18 July 2016.
We will transfer $1,000 to your account, nicknamed unixE, on 18 July 2016.
We will transfer $1,000 to your account, nicknamed unixE, on 18 July 2016.

The words from CFO must preserve their meaning as they change context, and ditto for the words from the COO component.
We will transfer $1,000 to your account, nicknamed _____, on 18 July 2016.
We will transfer $1,000 to your account, nicknamed ____ , on 18 July 2016.
We will transfer $1,000 to your account, nicknamed ____, on 18 July 2016.
The words from a foreign executive must be parsed before they enter our linguistic universe.

We will transfer $1,000 to your account, nicknamed ____, on 18 July 2016.

Wir werden $1,000 in ihr Konto, namens ____, on 18 July 2016, einbezahlen.
Hygienic Macros!
Hygienic Macros!
#lang setup/infotab
#lang scribble/manual
#lang typed/racket
#lang datalog
Embedded DSL Compilers

- Features from Macros
- Scope from Hygiene
- “Nativeness” from Syntax Objects
- “Errors” from “Parse”
- Flexibility from Reinterpretation
- Linguistic Reuse from Modules
- Integrity from Phases
- Surface Syntax from parser-tools
Embedded DSL Compilers

- **Features from Macros**
  ```scheme
  (provide
    (except-out (all-from-out racket) #%app add1)
    (rename-out
      [lazy-application #%app]
      [strict-add1 add1]))
  ```

- **Scope from Hygiene**
  ```scheme
  (define-syntax-rule
    (lazy-application f arg …)
  ```

- **“Nativeness” from Syntax Objects**
  ```scheme
  (;; rewrite the above to:
    (%app f (lambda () arg) …))
  ```

- **“Errors” from “Parse”**
  ```scheme
  (define (strict-add1 arg)
    (add1 [arg]))
  ```

- **Flexibility from Reinterpretation**
  ```scheme
  ```

- **Linguistic Reuse from Modules**
  ```scheme
  ```

- **Integrity from Phases**
  ```scheme
  ```

- **Surface Syntax from parser-tools**
  ```scheme
  ```
Embedded DSL Compilers

Features from Macros
Scope from Hygiene
“Nativeness” from Syntax Objects
“Errors” from “Parse”
Flexibility from Reinterpretation
Linguistic Reuse from Modules
Integrity from Phases
Surface Syntax from parser-tools

(provide
(except-out (all-from-out racket) #%app add1)
(rename-out
[lazy-application #%app]
[strict-add1 add1]))

(define-syntax-rule
(lazy-application f arg …)
;; rewrite the above to:
(#%app f (lambda () arg) …))

(define (strict-add1 arg)
(add1 [arg]))
Embedded DSL Compilers

- Features from Macros
- Scope from Hygiene
- "Nativeness" from Syntax Objects
- "Errors" from "Parse"
- Flexibility from Reinterpretation
- Linguistic Reuse from Modules
- Integrity from Phases
- Surface Syntax from parser-tools

```
(provide
  (except-out (all-from-out racket) #%app add1)
  (rename-out
    [lazy-application #%app]
    [strict-add1    add1]))

(define-syntax-rule
  (lazy-application f arg …)
  ;; rewrite the above to:
  (#%app            f (lambda () arg) …))

(define (strict-add1 arg)
  (add1  [arg]))
```
#lang datalog

edge(a, b).
edge(b, c).
edge(c, d).
edge(d, a).

path(X, Y) :- edge(X, Y).
path(X, Y) :- edge(X, Z), path(Z, Y).
path(X, Y)?
How do you compose (programs in) these “DSLs”? (run time)
COO invariants

CFO invariants

DSL COO
DSL CIO
DSL CTO
DSL CFO

Racket
how do you protect the meaning?
Concrete: Lazy Racket vs Strict Racket
Concrete: Lazy Racket vs Strict Racket

Strict invariants

DSL STRICT
DSL CIO
DSL CTO

Racket

DSL LAZY
DSL Counsel
Concrete: Lazy Racket vs Strict Racket

- Lazy invariants
- Strict invariants

Diagram showing different DSLs (DSL STRICT, DSL CIO, DSL CTO, DSL LAZY) with Racket at the bottom.
Concrete: Lazy Racket vs Strict Racket

- Lazy invariants
- Strict invariants

how do you protect these values?
Good Luck
Good Luck

think positive!!
HOPE FOR THE BEST

Good Luck

think positive!!
Composing DSL Components

- inspectors and code control
- events and event spaces
- impersonators & chaperones
- threads and time
- custodians and resources
- sandboxes and access
- wills and executors
Composing DSL Components

- inspectors and code control
- events and event spaces
- impersonators & chaperones
- threads and time
- custodians and resources
- sandboxes and access
- wills and executors

```scheme
#lang racket
(provide
 (contract-out
  (open
    ;; pops up a currently invisible area
    (-> (and/c window? invisible?)
        (and/c window? top-level?))))
```
Are all “DSL” problems solved?
RACKET IS A FULL-SPECTRUM PROGRAMMING LANGUAGE
Racket
ASM Racket
VeriRacket
Typed Racket
Contract Racket
Racket Racket
FFI Racket
ASM Racket (?)
VeriRacket
Typed Racket
Contract Racket
Racket Racket
FFI Racket
ASM Racket (?)

full-stack language
#lang racket

(provide
  ;; Image Number Number Image -> Image
  ;; (place obj x y bg) puts obj at (x,y) on bg
  place)

...

(define (place obj x y background)
  (define width (width background))
  (define height (height background))
  (unless (and (<= 0 x) (< x width))
    (error 'place "bad x")
  (unless (and (<= 0 y) (< x height))
    (error 'place "bad y")
  (place-proper obj x y background))
(provide
  (contract-out
    ;; (place obj x y bg) puts obj at (x,y) on bg
    (place
      (->i ((obj image?)
        (x (bg) (and/c number?
          (>=/c 0) (</c (width bg))))
        (y (bg) (and/c number?
          (>=/c 0) (</c (height bg))))
        (bg image?))
      (result image?)))))

(define (place obj x y background)
  (place-proper obj x y background))
;; with contracts

(provide
 (contract-out

 ;; (place obj x y bg) puts obj at (x,y) on bg
 (place
  (->i ((obj image?)
     (x (bg) (and/c number? (>=/c 0) (</c (width bg)))))
  (y (bg) (and/c number? (>=/c 0) (</c (height bg)))))
  (bg image?))
  (result image?))))

(define (place obj x y background)
 (place-proper obj x y background))

#lang racket
#lang typed/racket

(provide
  (contract-out

    ;; (place obj x y bg) puts obj at (x,y) on bg
    (place
      (place
        (-> Image Number Number Number Image)
        (->i ((x (bg) (and/c (>=/c 0) (</c (width bg))))
          (y (bg) (and/c (>=/c 0) (</c (height bg))))
          any))))

    (define (place obj x y background)
      (place-proper obj x y background)))
(provide
  (contract-out
    ;; (place obj x y bg) puts obj at (x,y) on bg
    (place
      (-> Image Number Number Image)
      (->i ((x bg) (and/c (>=/c 0) (</c (width bg))))
        (y bg) (and/c (>=/c 0) (</c (height bg))))
        any)))))

(define (place obj x y background)
  (place-proper obj x y background))
And we can also go in the other direction.
#lang racket

(require unsafe)

(provide
  ;; Image Number Number Image -> Image
  ;; (place obj x y bg) puts obj at (x,y) on bg place)

...

(define (place obj x y background)
  ;; FFI Calls for Speed
)
)
More work to be done.
Coming to a RacketCon near you real soon.

18 Sept 2016 · St. Louis: Union Station hotel · Speakers · Register

SIXTH
RACKET
CON

(sixth RacketCon) is a public meeting for everyone interested in Racket: developers, contributors, programmers, educators, and bystanders. It’s an opportunity for all members of the community to come together to share plans, ideas, and enthusiasm. RacketCon will enable the entire Racket community to mingle: to update each other, to exchange ideas, to collaborate, and to help shape the future of Racket.

KEYNOTE

Emina Torlak
There is a competition for resources, so we need means to articulate resource claims.
Contextual service concepts migrate into Racket.
Welcome to DrRacket version 6.2.901.17-2015-09-19(11070/2a/d)
Language: http/dsl+ [custom].

(define-struct db (name type) db)
(define-struct schema (content) schema)
(define (make-db name type) (list name type))
(define (make-schema content) (list content))
(define (list db s c r) (list db s c r))
(define spec (list label predicate))
(define make-spec (define spec-label First) (define spec-first (define db...)})

CC: 0:min @ 414,696K(+100,155K)[+34,028K]; free 27,195K(-27,196K) 26ms @ 46569
CC: 0:min @ 420,454K(+94,398K)[+34,028K]; free 27,390K(-27,391K) 26ms @ 46691
CC: 0:min @ 426,329K(+88,525K)[+34,040k]; free 22,560K(-22,563K) 44ms @ 46624
CC: 0:min @ 437,775K(+77,107K)[+34,460k]; free 28,086K(-29,194K) 56ms @ 47334
What does it take to build DrRacket in Racket?
Racket internalizes features of IDEs and Operating Systems.

- NO PROJECTS
- inspectors provide access rights
- composing classes at run time
- linking modules at run time
Racket internalizes features of IDEs and Operating Systems.

From MATTHIAS to ROBBY, MATTHEW:

Could DrRacket link the name of the primitive to its docs in error messages, especially in *SL?

From ROBBY to MATTHIAS, MATTHEW:

That would require keeping compile time info around at run time.

From MATTHIAS to ROBBY, MATTHEW:

Why? Can’t DrRacket just map the primitives.

From ROBBY to MATTHEW, MATTHIAS:

I want to make sure it works inside of Racket.
RACKET DESIGN NEEDS A FEEDBACK LOOP ...
(require 2hawk/image)

;; Number -> Image
(check-equal (image? (tree 2)) #true)
(define (tree n)
  (if (= n 0)
      #false
      (cons (circle 50 "solid" "green")
            (cons (rectangle 40 "solid" "brown")
                  (tree (- n 1))))))

Welcome to DrRacket, version 5.2.903.17~2015-09-18[110739au] [rn].
Language: Beginning Student
The test passed!
> (tree 2)

>
;; Number -> Image
(check-expect (image? (tree 2)) true)
(define (tree n)
  (+
    (show
     (circle 20 "solid" "green")
     (rectangle 40 "solid" "brown")))
)

Welcome to DrRacket, version 5.2.900.17-2015-08-18[11:15:13]
[en].
Language: Beginning Student
The test passed!
> (tree 2)

BEGINNING STUDENT
TAKE AWAY
1. Racket is a programming-language programming language.
2. Racket is a full-spectrum programming language.
3. Racket is a linguist’s language.

Racket lives inside a feedback loop, and it needs you.