Languages as Libraries

or, implementing the next 700 programming languages

Sam Tobin-Hochstadt
Vincent St-Amour   Ryan Culpepper
Matthew Flatt   Matthias Felleisen

PLT @ Northeastern & Utah

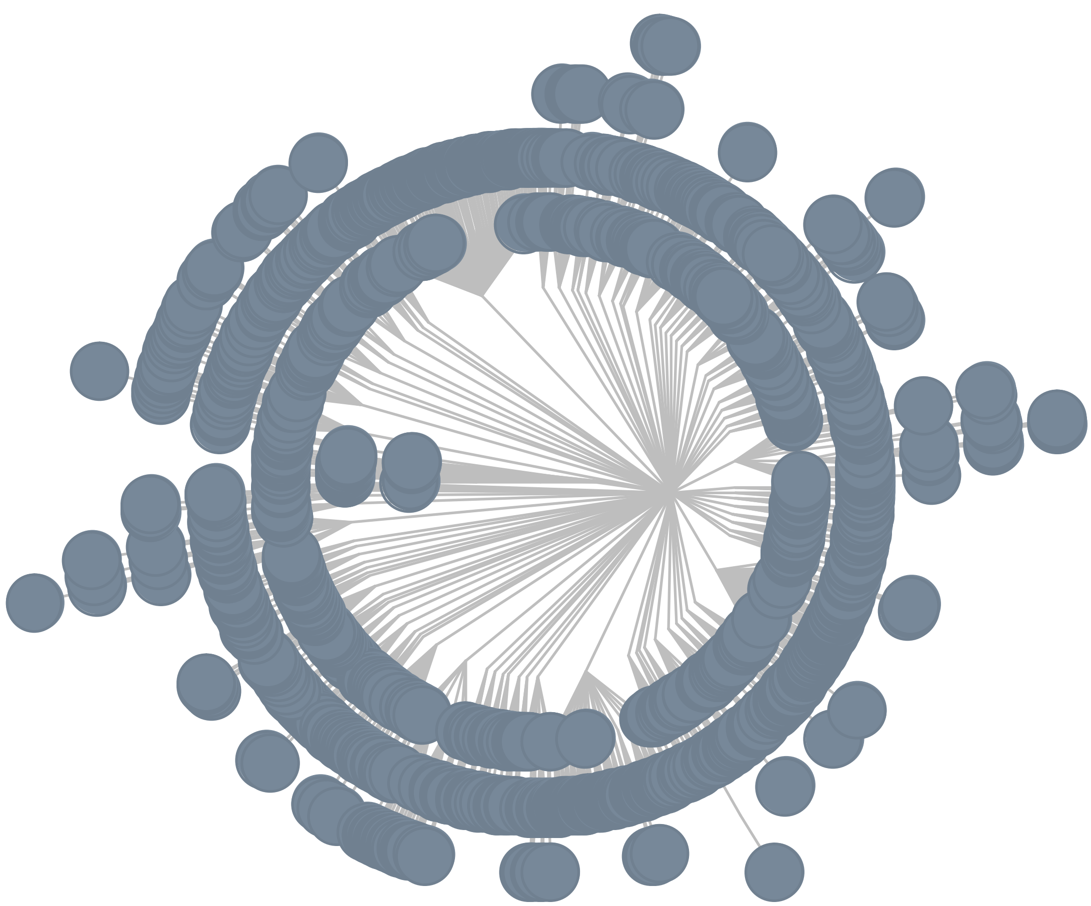
June 6, 2011   PLDI

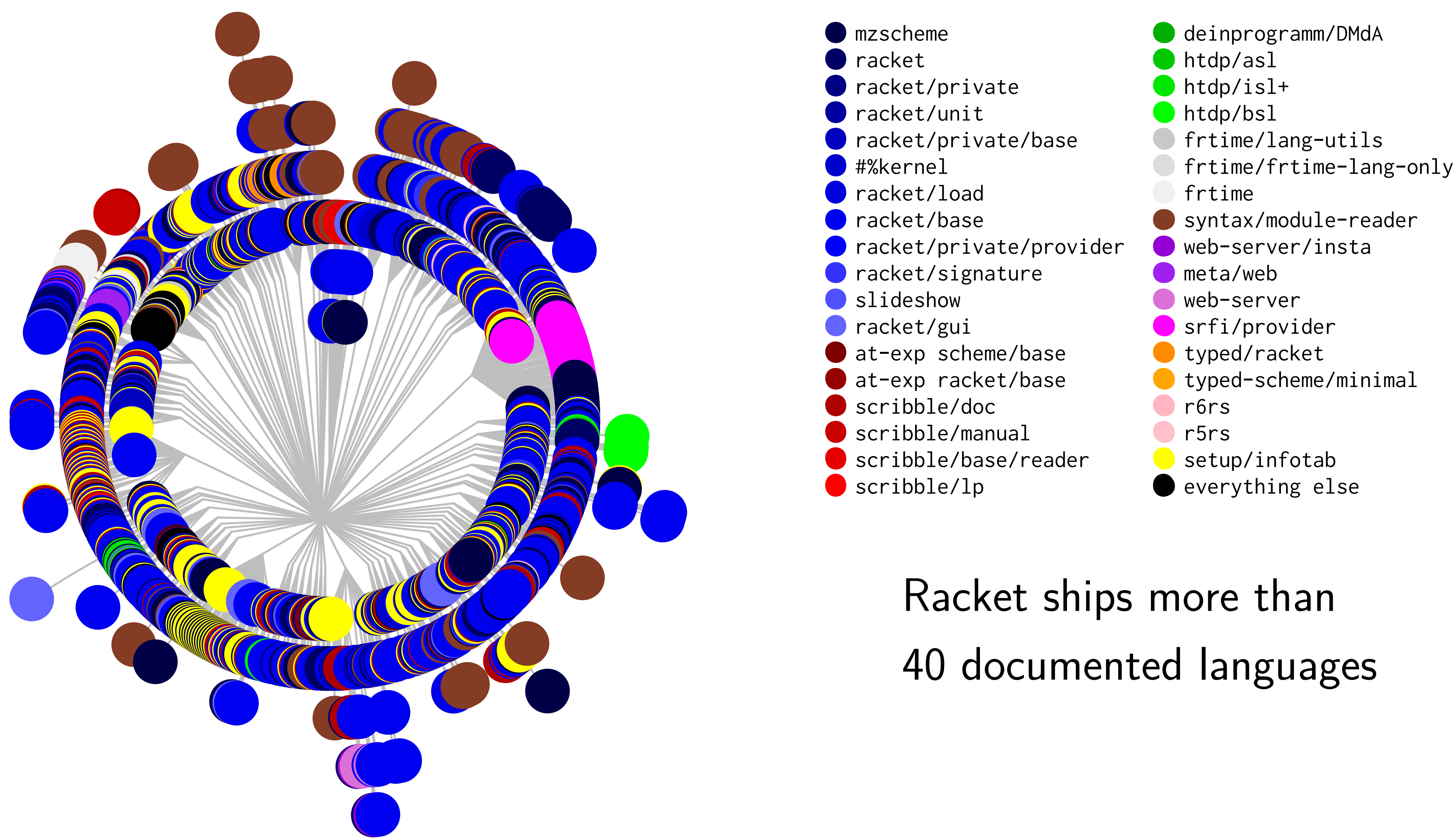
“A domain specific language is the ultimate abstraction.”

— Paul Hudak

“There will always be things we wish to say in our programs
that in all known languages can only be said poorly.”

— Alan Perlis



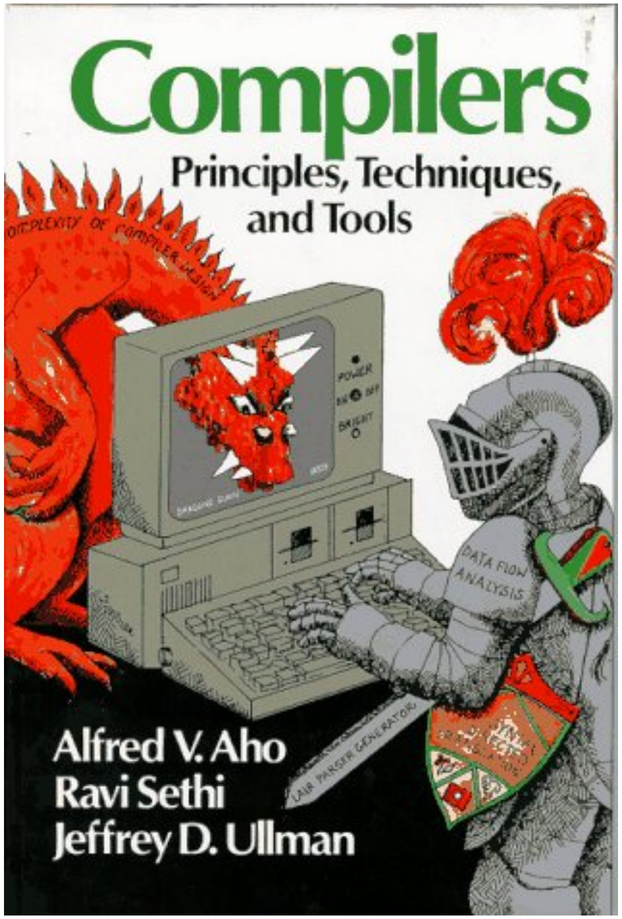


Racket ships more than

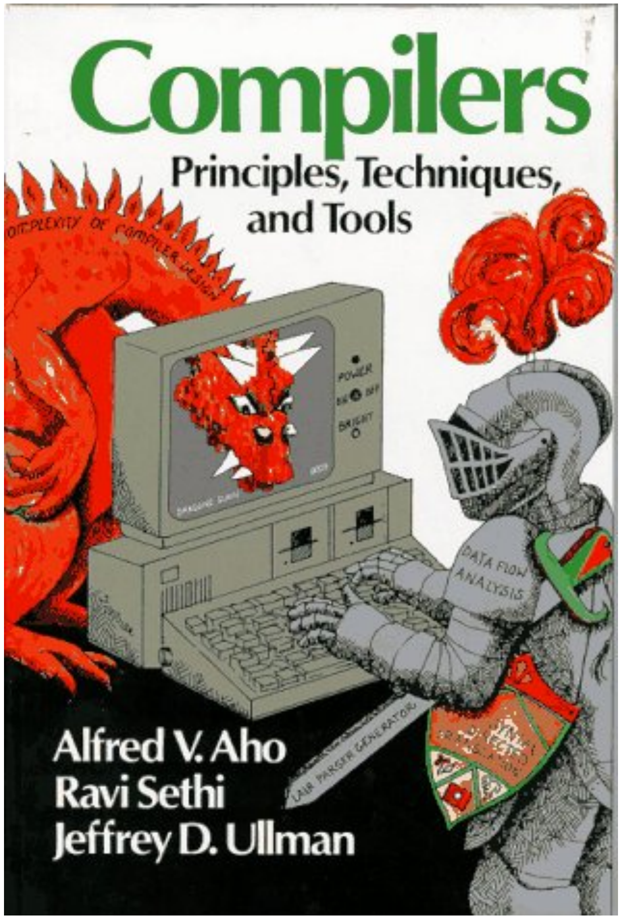
40 documented languages

How can we build so many languages?

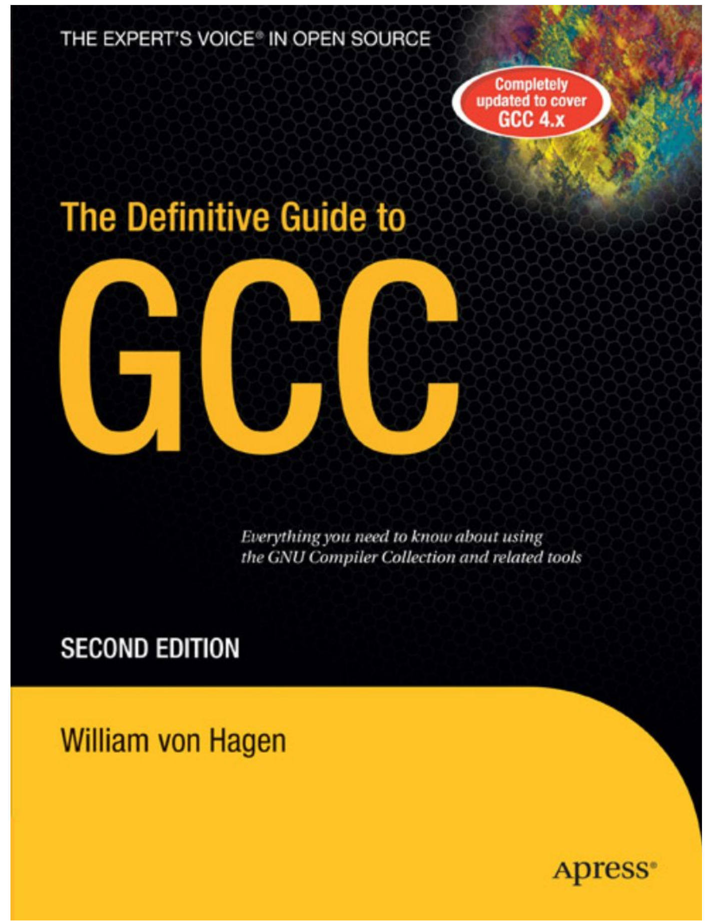
The Traditional Approach



The Traditional Approach



Produces impressive results



The Macro Approach

```
(define-syntax and
  (syntax-parser
   [(_ e1 e2)
    #'(if e1 e2 #f)]))
```

The Macro Approach

```
(define-syntax and
  (syntax-parser
   [(_ e1 e2)
    #'(if e1 e2 #f)]))
```

Supports linguistic reuse

Scoping

`if`

...

Functions

Classes

Modules

Our approach:

Linguistic reuse of the macro approach

Capabilities of the traditional approach

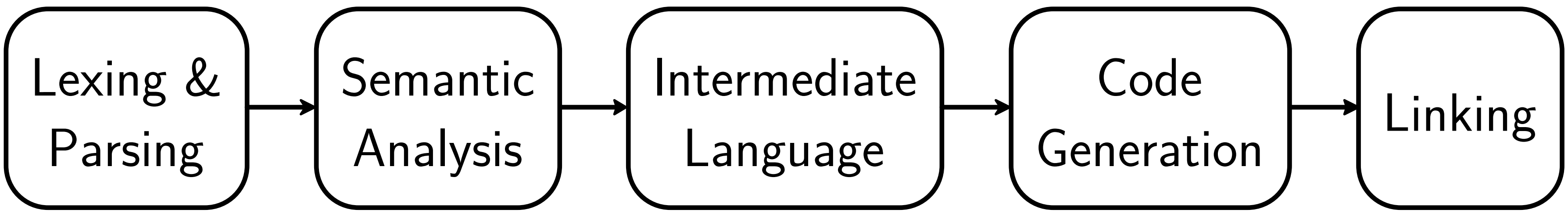
Our approach:

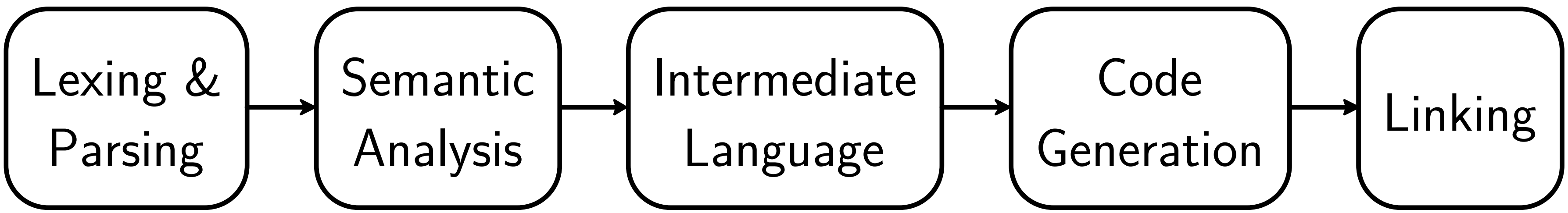
Linguistic reuse of the macro approach

Capabilities of the traditional approach

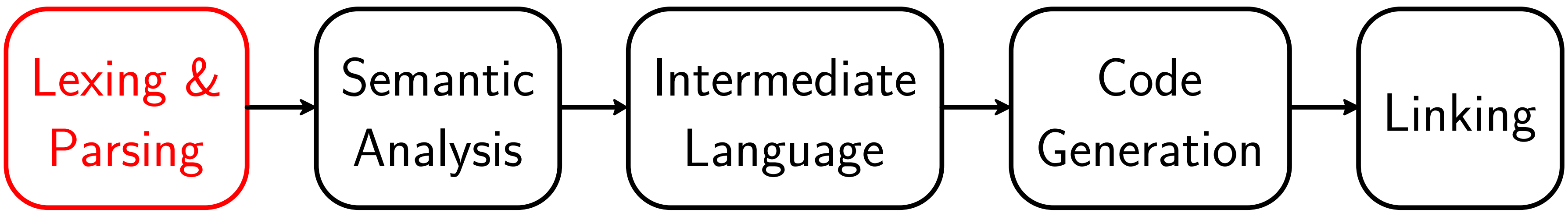
By exposing compiler tools to library authors

Providing the tools

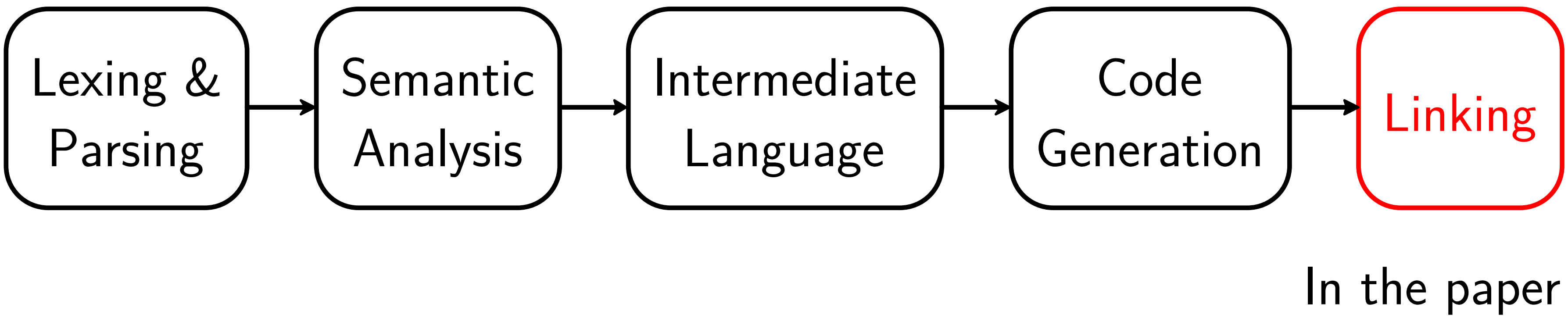

Lexing & Parsing
Semantic Analysis
Intermediate Language
Code Generation
Linking



Language authors control each stage



[Flatt et al, 2009]


Lexing & Parsing
Semantic Analysis
Intermediate Language
Code Generation
Linking
In the paper

Illustrated by Typed Racket

Semantic Analysis

Static Checking

ack

```
#lang        racket

;  ack : Integer Integer -> Integer
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))

(ack 2 3)
```

Static Checking

```
#lang typed/racket                                    ack


 (: ack : Integer Integer -> Integer)
 (define (ack m n)
   (cond [(<= m 0) (+ n 1)]
         [(<= n 0) (ack (- m 1) 1)]
         [else (ack (- m 1) (ack m (- n 1)))]))

 (ack 2 3)
```

Static Checking

ack

```
#lang typed/racket


(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))

(ack 2 3)
```

Type checking is a *global* process

module-begin

ack

```
#lang typed/racket

(module-begin
 (: ack : Integer Integer -> Integer)
 (define (ack m n)
   (cond [(<= m 0) (+ n 1)]
         [(<= n 0) (ack (- m 1) 1)]
         [else (ack (- m 1) (ack m (- n 1)))]))

 (ack 2 3))
```

Languages control the whole module

Implementing a language

`#lang racket` `typed/racket`

Module Semantics

```
(define-syntax module-begin ...)
```

Core Syntax

```
(define-syntax λ ...)
```

Standard Functions

```
(define + ...)
```

Implementing a language

typed/racket

```
#lang racket

(define-syntax module-begin
  (syntax-parser
   [(_ forms ...)
    (for ([form #'(forms ...)])
      (typecheck form))

    #'(forms ...)]))
```

The Typechecker

typechecker

```
#lang racket

(define (typecheck form)
  (syntax-parse form
    [v:identifier
     ...]
    [(λ args body)
     ...]
    [(define v body)
     ...]


    ... other syntactic forms ...))
```

Intermediate Language

Why Intermediate Languages?

"The compiler serves a broader set of programmers than it would if it only supported one source language"

— Chris Lattner

Why Intermediate Languages?

Most forms come from libraries

```
(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))
```

Why Intermediate Languages?

Most forms come from libraries

```
(: ack : Integer Integer -> Integer)
(define (ack m n)
  (cond [(<= m 0) (+ n 1)]
        [(<= n 0) (ack (- m 1) 1)]
        [else (ack (- m 1) (ack m (- n 1)))]))
```

Also: pattern matching, keyword arguments, classes,

loops, comprehensions, any many more

- Can't know static semantics ahead of time

Core Racket

Racket defines a common subset that expansion targets

```
expr ::= identifier
         (plain-lambda args expr)
         (app expr ...+)
         ...
```

a dozen core expressions

```
def ::= expr
        (define-values ids expr)
        (require spec)
        ...
```

local-expand

typed/racket

```
#lang racket

(define-syntax module-begin
  (syntax-parser
   [(_ forms ...)
    (define expanded-forms
          (local-expand #'(forms ...)))
    (for ([form expanded-forms])
      (typecheck form))

    expanded-forms]))
```

The Revised Typechecker

```
#lang racket                                    typechecker

(define (typecheck form)
  (syntax-parse form
    [v:identifier
     ...]
    [(plain-lambda args body)
     ...]
    [(define-values vs body)
     ...]

    ... two dozen core forms ...))
```

Communication between levels — see paper

Code Generation

Code generation

Problem: optimizing generic arithmetic

```
(: norm : Float Float -> Float)
(define (norm x y)
  (sqrt (+ (sqr x) (sqr y))))
```

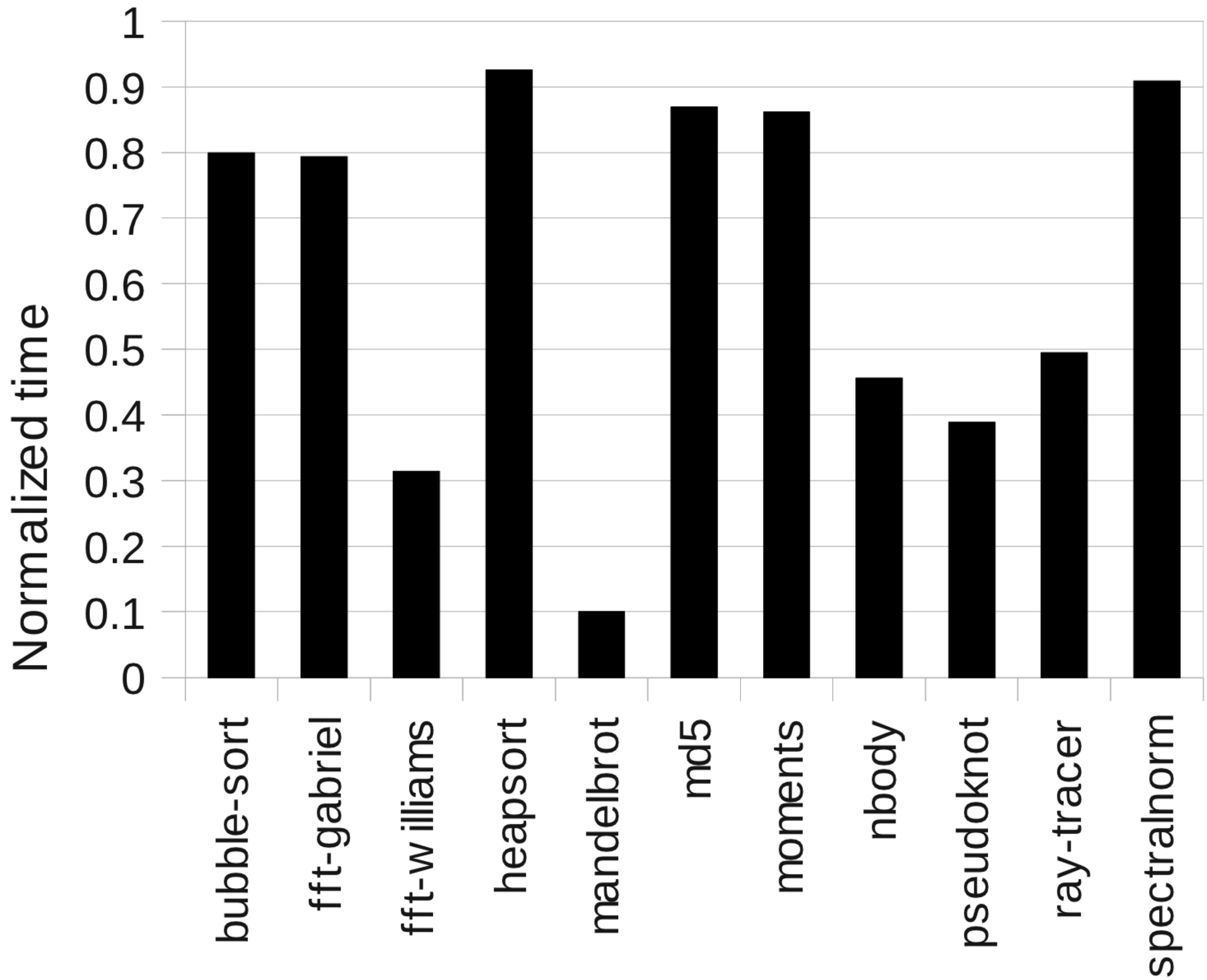
Code generation

Express guarantees as rewritings

```
(: norm : Float Float -> Float)
(define (norm x y)
  (unsafe-flsqrt
   (unsafe-fl+ (unsafe-fl* x x)
               (unsafe-fl* y y))))
```

Low-level operations expose code generation to libraries

Results


Normalized time
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
bubble-sort
fft-gabriel
fft-williams
heapsort
mandelbrot
md5
moments
nbody
pseudoknot
ray-tracer
spectralnorm

The take-away

- Languages are powerful abstractions
- Racket enables full-scale languages as libraries
- Key idea: expose compiler pipeline to language authors

The take-away

- Languages are powerful abstractions
- Racket enables full-scale languages as libraries
- Key idea: expose compiler pipeline to language authors

Thank you

`racket-lang.org`