

Hackett

a Haskell for Racketeers

What is **Hackett**?

What is **Hackett**?

Haskell + **Racket**

Haskell

pure, functional programming language

static types, strong type inference

algebraic datatypes + pattern-matching

```
(data (Maybe a)  
  (just a) nothing)
```

```
(case x  
  [(just v) (+ v 10)]  
  [nothing 0])
```

Haskell

pure, functional programming language

static types, strong type inference

algebraic datatypes + pattern-matching

typeclasses

```
(class (Show a)  
  [show : {a -> String}])
```

Haskell

pure, functional programming language

static types, strong type inference

algebraic datatypes + pattern-matching

typeclasses

higher-rank polymorphism

```
(forall [b] {(forall [a] {a -> a})  
-> b -> b})
```

Racket

a Racket #lang

hygienic macros

```
(define-syntax-parser do
  #:datum-literals [<-]
  [(_ [~brackets ~! x:id <- e:expr] rest ...+)
   (>>= e (lambda [x] (do rest ...)))])
[(_ e:expr)
 #'e]
[(_ e:expr rest ...+)
 (syntax/loc #'e
  (>>= e (lambda [x] (do rest ...))))])
```

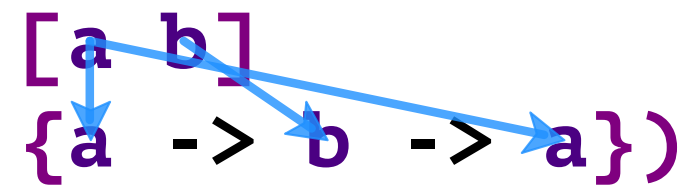
Racket

a Racket #lang

hygienic macros

editor/tooling integration

```
(def const : (forall [a b]
  {a -> b -> a})
  (lambda [x _] x))
```



Racket

a Racket #lang

hygienic macros

editor/tooling integration

access to the Racket ecosystem

```
(require hackett/demo/pict  
         hackett/demo/pict/universe)
```

Hackett has...

...the **runtime**, **value**, and **type** model of
Haskell...

...but the **syntax** model of **Racket**.

Why types?

Static verification to catch bugs?

Type Erasure

```
#lang typed/racket
```

```
(define (f [x : Integer]) : Integer  
  (+ x 1))
```

```
(f 5) ; => 6
```

Type Erasure

```
#lang typed/racket
```

```
(define (f [x : Integer]) : Integer  
  (+ x 1))
```

```
(f 5) ; => 6
```

Type Erasure

`(pure 5) ; => ?`

`(: (pure 5) (Maybe Integer))`

`; => (just 5)`

`(: (pure 5) (Either String Integer))`

`; => (right 5)`

Type Erasure

mzero ; => ?

(: mzero String) ; => ""

(: mzero (Maybe Integer)) ; => nothing

(: mzero (List Bool)) ; => nil

Why types?

A principled system for propagating information at compile-time.

Racket has *macros that communicate*.

```
#lang racket
```

```
(define-enum animal  
  [dog cat deer])
```

```
(define (animal->baby-name a)  
  (enum-case animal a  
    [dog "puppy"]  
    [cat "kitten"]  
    [deer "fawn"]))
```

Racket has *macros that communicate*.

```
#lang racket
```

```
(define-enum animal  
  [dog cat deer])
```

```
(define (animal->baby-name a)  
  (enum-case animal a  
    [dog "puppy"]  
    [cat "kitten"]  
    [deer "fawn"]))
```

Racket has *macros that communicate*.

```
#lang racket
```

```
(define-enum animal  
  [dog cat deer bird])
```

```
(define (animal->baby-name a)  
  (enum-case animal a  
    [dog "puppy"]  
    [cat "kitten"]  
    [deer "fawn"])))
```

Racket has *macros that communicate*.

```
(define-syntax-parser define-enum
  [(_ name:id [elem:id ...])]
  #'(define-syntax name
      (set 'elem ...)))
```

```
(define-syntax-parser enum-case
  [(_ enum-name:id val:expr
      [elem:id body:expr ...+])]
  ... (syntax-local-value #'enum-name) ...
  #'(case val
      [(elem) body] ...))
```

Racket has *macros that communicate*.

```
#lang racket
```

```
(define-enum animal  
  [dog cat deer bird])
```

```
(define (animal->baby-name a)  
  (enum-case animal a  
    [dog "puppy"]  
    [cat "kitten"]  
    [deer "fawn"]))
```

=> enum-case: no clause for 'bird'


Racket has *macros that communicate*.

```
#lang racket
```

```
(define-enum animal  
  [dog cat deer bird])
```

```
(define (animal->baby-name a)  
  (enum-case animal a  
    [dog "puppy"]  
    [cat "kitten"]  
    [deer "fawn"])))
```

annoying!



Racket has *macros that communicate...*
but the communication is ad-hoc.

Type inference is a *principled* system that can propagate compile-time information.

(lambda [(just x)] x)

=> **non-exhaustive pattern-match
unmatched case:
nothing**

Note that exhaustiveness checking is not fundamentally part of typechecking!

SQL DSL

Problem: SQL is error-prone.

```
(define-persistent-struct
  (user [name sql:string]
        [email sql:string]
        [password-hash sql:bytes]
        [created-at sql:moment])
  #:table-name "users")
```

Solution: teach the language about your schema.

SQL DSL

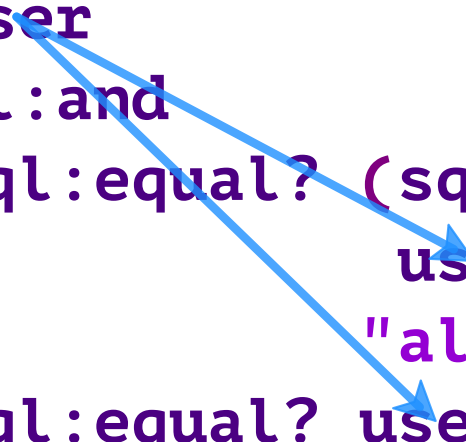
Querying simple data is easy.

```
(struct id (table value))  
(get (id table:user 1))
```

SQL DSL

Problem: sometimes we want complex queries.

```
(find table:user
  #:where (sql:and
    (sql:equal? (sql:string-downcase
      user.email)
      "alyssa@example.com")
    (sql:equal? user.password "1234")))
```



Solution: provide a schema-aware query language.

SQL DSL

```
(find table:user  
  #:where (sql:and  
    (sql:equal? (sql:string-downcase  
      user.email)  
      "alyssa@example.com")  
    (sql:equal? user.password "1234")))
```

← this again!

SQL DSL

```
(defn handle-login
  [[email password]
   (do [maybe-user
        <- (find (sql:and
                  (sql:equal? (sql:string-downcase
                               user.email)
                               email)
                            (sql:equal? user.password password)))]

        (case maybe-user
          [(just user) (set-session/redirect user)]
          [nothing     render-login-403]))))
```

SQL DSL

```
(defn handle-login
  [[email password]
   (do [maybe-user
        <- (find (sql:and
                  (sql:equal? (sql:string-downcase
                               user.email)
                               email)
                            (sql:equal? user.password password)))]

        (case maybe-user
          [(just user) (set-session/redirect user)]
          [nothing     render-login-403]))))
```

set-session/redirect : {User -> (IO Response)}

SQL DSL

```
(defn handle-login
  [[email password]
   (do [maybe-user
        <- (find (sql:and
                  (sql:equal? (sql:string-downcase
                               user.email)
                               email)
                            (sql:equal? user.password password)))]

        (case maybe-user
          [(just user) (set-session/redirect user)]
          [nothing     render-login-403]))))
```

user ⇒ **User**

SQL DSL

```
(defn handle-login
  [[email password]
   (do [maybe-user
        <- (find (sql:and
                  (sql:equal? (sql:string-downcase
                               user.email)
                               email)
                  (sql:equal? user.password password)))]

        (case maybe-user
          [(just user) (set-session/redirect user)]
          [nothing     render-login-403]))))
```

maybe-user ⇒ (Maybe User)

SQL DSL

```
(defn handle-login
  [[email password]
   (do [maybe-user
        <- (find (sql:and
                  (sql:equal? (sql:string-downcase
                               user.email)
                               email)
                  (sql:equal? user.password password)))]

        (case maybe-user
          [(just user) (set-session/redirect user)]
          [nothing     render-login-403]))))
```

(find ...) \Leftarrow **(IO (Maybe User))**

SQL DSL

```
(defn handle-login
  [[email password]
   (do [maybe-user
        <- (find (sql:and
                  (sql:equal? (sql:string-downcase
                               user.email)
                               email)
                  (sql:equal? user.password password)))]

        (case maybe-user
          [(just user) (set-session/redirect user)]
          [nothing     render-login-403]))))
```

But this “backwards inference” is tricky.

The Constraint Solver

Typeclasses already require this sort of backwards inference.

```
{"hello, " ++ "world!"}
```

⇒

```
{"hello, " ++String "world!"}
```

The Constraint Solver

Solution: perform two passes of macroexpansion.

```
{"hello, " ++ "world!"}
```

⇒

```
(#%app (typeclass-dict  
      (Semigroup t))  
      "hello, " "world!")
```

⇒

```
t ~ String
```

```
(#%app ++String "hello, " "world!")
```

Programmable constraint solver?

Lots of possibilities.

Recap

Hackett is a Haskell that runs on Racket.

Today: Hackett cooperates with Racket's hygienic macroexpander.

Today: Hackett supports Haskell idioms, including typeclasses.

Today: Hackett integrates with Racket's language tooling.

Soon: Use Hackett to write real programs.

Soon: More Haskell features (e.g. multi-param typeclasses).

Eventually: Contract-protected Racket/Hackett FFI.

Eventually: Programmable constraint solver.

Questions?

Install: `raco pkg install hackett`

Docs: <http://docs.racket-lang.org/hackett/>

Source: <https://github.com/lexi-lambda/hackett>