Part I

Pattern-Based Macros

With **`gensym.rkt`**:

```
(add-let
 `{let-macro {[case {lambda {s}
                     ,(add-let
                       `{let {[tmp {gensym}]}
                         {cons 'let
                          {cons {cons {cons tmp {cons {first {rest s}}
                                                      '{}}}
                                      '{}}
                                {cons {if0 {if0 {symbol? {first {first {rest {rest s}}}}}
                                                {symbol=? 'else {first {first {rest {rest s}}}}}
                                                1}
                                           {first {rest {first {rest {rest s}}}}}
                                           {cons 'if0
                                            {cons {cons '+
                                                   {cons tmp
                                                    {cons {* -1 {first {first {first {rest {rest s}}}}}}
                                                     '{}}}}}
                                             {cons
                                              {first {rest {first {rest {rest s}}}}}
                                              {cons {cons 'case
                                                     {cons tmp
                                                      {rest {rest {rest s}}}}}
                                               '{}}}}}}
                                      '{}}}}}}}}]}
   {let {[f {lambda {n}
              {case n
               [{1} 10]
               [{2} 100]
               [else 1]}}]}
    {+ {+ {f 1} {f 2}}
       {* -1 {f 3}}}}}})
```

Pattern-Based Macros

In Racket:

```
(let-syntax ([case (syntax-rules (else)
                     [(case expr [else rhs])
                      (let ([tmp expr])
                        rhs)]
                     [(case expr [(num) rhs] clause ...)
                      (let ([tmp expr])
                        (if0 (+ tmp (* -1 num))
                             rhs
                             (case expr clause ...)))])])
  (let ([f (lambda (n)
             (case n
               [(1) 10]
               [(2) 100]
               [else 1]))])
    (+ (+ (f 1) (f 2))
       (* -1 (f 3)))))
```

Pattern-Based Macros

In Racket:

```
(let-syntax ([case (syntax-rules (else)
                     [(case expr [else rhs])
                      (let ([tmp expr])
                        rhs)]
                     [(case expr [(num) rhs] clause ...)
                      (let ([tmp expr])
                        (if0 (+ tmp (* -1 num))
                             rhs
                             (case expr clause ...)))])])
                 n)
                 10]
                 100]
              [else 1]))])
  (+ (+ (f 1) (f 2))
     (* -1 (f 3)))))
```

Expands to

```
(lambda (stx)
  ....
```

pull expression apart

and

rearrange the pieces

```
  ....)
```

Pattern-Based Macros

In Racket:

```
(let-syntax ([delay (syntax-rules ()
                      [(delay expr)
                       (lambda (dummy) expr)])])
  (let-syntax ([force (syntax-rules ()
                        [(force expr)
                         (expr 0)])])
    (force (delay 7))))
```

Pattern-Based Macros

In Racket:

```
(define-syntax delay (syntax-rules ()
                       [(delay expr)
                        (lambda (dummy) expr)]))

(define-syntax force (syntax-rules ()
                       [(force expr)
                        (expr 0)]))

(force (delay 7))
```

Pattern-Based Macros

In Racket:

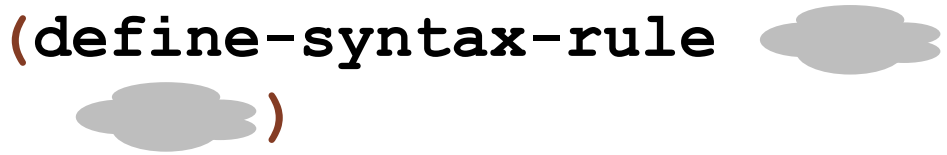
```
(define-syntax-rule (delay expr)
  (lambda (dummy) expr))

(define-syntax-rule (force expr)
  (expr 0))

(force (delay 7))
```

Part 2

Simple Pattern-Based Macros

```
(define-syntax-rule
  )
```

Simple Pattern-Based Macros



- **`define-syntax-rule`** indicates a simple-pattern macro definition

Simple Pattern-Based Macros

```
(define-syntax-rule pattern
  template)
```

- A *pattern* to match
- Produce result from *template*

Simple Pattern-Based Macros

```
(define-syntax-rule (swap a b)
  (                 )
```

Simple Pattern-Based Macros

```
(define-syntax-rule (swap a b)
  (          ))
```

- Pattern for this macro: `(swap a b)`

Simple Pattern-Based Macros

```
(define-syntax-rule (swap a b)
  (                 )
```

- Pattern for this macro: `(swap a b)`
- Each pattern identifier matches anything

`(swap x y)` ⇒ `a` is `x`, `b` is `y`

`(swap 9 (+ 1 7))` ⇒ `a` is `9`, `b` is `(+ 1 7)`

Simple Pattern-Based Macros

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

Simple Pattern-Based Macros

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

Matches substituted into template to generate the result

```
(swap x y)          ⇒ (let ([tmp y])
                        (set! y x)
                        (set! x tmp))

(swap 9 (+ 1 7))    ⇒ (let ([tmp (+ 1 7)])
                        (set! (+ 1 7) 9)
                        (set! 9 tmp))
```

Part 3

General Pattern-Based Macros

```
(define-syntax shift
  [illegible])
```

```
(let ([x 0]
      [y 1]
      [z 2])
  (shift x y z))
```

```
(let ([x 0]
      [y 1]
      [z 2])
  (shift back x y z))
```

General Pattern-Based Macros

```
(define-syntax shift
  )
```

- **`define-syntax`** indicates a macro definition

General Pattern-Based Macros

```
(define-syntax shift
  (syntax-rules (back)
    ))
```

- **`syntax-rules`** means a pattern-matching macro
- **`(back)`** means that **`back`** is literal in patterns

General Pattern-Based Macros

```
(define-syntax shift
  (syntax-rules (back)
    [pattern template]
    ...
    [pattern template]))
```

- Any number of *pattern*s to match
- Produce result from *template* of first match

General Pattern-Based Macros

```
(define-syntax shift
  (syntax-rules (back)
    [(shift a b c) ]
    [(shift back a b c) ]))
```

Two patterns for this macro

- `(shift x y z)` matches first pattern
- `(shift back x y z)` matches second pattern
- `(shift rev x y z)` does not match

General Pattern-Based Macros

```
(define-syntax shift
  (syntax-rules (back)
    [(shift a b c) (begin
                     (swap a b)
                     (swap b c))]
    [(shift back a b c) (begin
                          (swap c b)
                          (swap b a))]))
```

```
(shift x y z)        ⇒  (begin
                          (swap x y)
                          (swap y z))

(shift back x y z)   ⇒  (begin
                          (swap z y)
                          (swap y x))
```

Part 4

Matching Sequences

Some macros need to match sequences

```
(rotate x y)

(rotate red green blue)

(rotate front-left
        rear-right
        front-right
        rear-left)
```

Matching Sequences

```
(define-syntax rotate
  (syntax-rules ()
    [(rotate a) (void)]
    [(rotate a b c ...) (begin
                          (swap a b)
                          (rotate b c ...))]))
```

- `...` in a pattern: 0 or more of previous sub-pattern

  `(rotate x y z w)` ⇒ `c` is `z w`

- `...` in a template: 0 or more of previous sub-template

```
(rotate x y z w) ⇒ (begin
                     (swap x y)
                     (rotate y z w))
```

Matching Sequences

```
(define-syntax rotate
  (syntax-rules ()
    [(rotate a c ...)
     (shift-to (c ... a) (a c ...))]))

(define-syntax shift-to
  (syntax-rules ()
    [(shift-to (to0 to ...) (from0 from ...))
     (let ([tmp from0])
       (set! to from) ...
       (set! to0 tmp))]))
```

- `...` maps over same-sized sequences
- `...` duplicates constants paired with sequences

Part 5

Macro Scope

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

What if we **swap** a variable named **tmp**?

```
(let ([tmp 5]                 ?   (let ([tmp 5]
      [other 6])              ⇒         [other 6])
  (swap tmp other))                 (let ([tmp other])
                                      (set! other tmp)
                                      (set! tmp tmp)))
```

Macro Scope

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

What if we **swap** a variable named **tmp**?

```
(let ([tmp 5]                 ?   (let ([tmp 5]
      [other 6])              ⇒         [other 6])
  (swap tmp other))                 (let ([tmp other])
                                      (set! other tmp)
                                      (set! tmp tmp)))
```

This expansion would break scope

Macro Scope

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

What if we **swap** a variable named **tmp**?

```
(let ([tmp 5]                    ⇒  (let ([tmp 5]
      [other 6])                         [other 6])
  (swap tmp other))                  (let ([tmp1 other])
                                       (set! other tmp)
                                       (set! tmp tmp1)))
```

Hygienic macros rename the introduced binding

Hygienic Macros: Local Bindings

Hygiene means that local macros work, too:

```
(define (f x)
  (define-syntax swap-with-arg
    (syntax-rules ()
      [(swap-with-arg y) (swap x y)]))

  (let ([z 12]
        [x 10])
    ; Swaps z with original x:
    (swap-with-arg z))
)
```

Part 6

Identifier Macros

The `swap` and `rotate` names work only in an "application" position

```
(swap x y)  ⇒  (let ([tmp y]) ...)
(+ swap 2)  ⇒  syntax error
```

An ***identifier macro*** works in any expression position

```
clock          ⇒  (get-clock)
(+ clock 10)   ⇒  (+ (get-clock) 10)
(clock 5)      ⇒  ((get-clock) 5)
```

...or as a `set!` target

```
(set! clock 10)  ⇒  (set-clock! 10)
```

Identifier Macros

Using **`syntax-id-rules`**:

```
(define-syntax clock
  (syntax-id-rules (set!)
    [(set! clock e) (put-clock! e)]
    [(clock a ...) ((get-clock) a ...)]
    [clock (get-clock)]))
```

- **`set!`** is designated as a literal
- **`syntax-rules`** is a special case of **`syntax-id-rules`** with errors in the first and third cases

Part 7

Macro-Generating Macros

If we have many identifiers like `clock`...

```
(define-syntax define-get/put-id
  (syntax-rules ()
    [(define-get/put-id id get put!)
     (define-syntax id
       (syntax-id-rules (set!)
        [(set! id e) (put! e)]
        [(id a (... ...)) ((get) a (... ...))]
        [id (get)])) ]))

(define-get/put-id clock get-clock put-clock!)
```

where `(... ...)` in a template gets replaced by `...`

Part 8

Extended Example

Let's add call-by-reference definitions to Racket

```
(define-cbr (f a b)
  (swap a b))

(let ([x 1] [y 2])
  (f x y)
  x)
; should produce 2
```

Extended Example

Expansion of first half:

```
(define-cbr (f a b)
  (swap a b))
```

⇒

```
(define (do-f get-a get-b put-a! put-b!)
  (define-get/put-id a get-a put-a!)
  (define-get/put-id b get-b put-b!)
  (swap a b))
```

Extended Example

Expansion of second half:

```
(let ([x 1] [y 2])
  (f x y)
  x)
```

⇒

```
(let ([x 1] [y 2])
  (do-f (lambda () x)
        (lambda () y)
        (lambda (v) (set! x v))
        (lambda (v) (set! y v)))
  x)
```

Call-by-Reference Setup

How the first half triggers the second half:

```
(define-syntax define-cbr
  (syntax-rules ()
    [(_ (id arg ...) body)
     (begin
       (define-for-cbr do-f (arg ...)
         () body)
       (define-syntax id
         (syntax-rules ()
           [(id actual (... ...))
            (do-f (lambda () actual)
                  (... ...)
                  (lambda (v)
                    (set! actual v))
                  (... ...))]))))]))
```

Call-by-Reference Body

Remaining expansion to define:

```
(define-for-cbr do-f (a b)
  () (swap a b))
```

⇒

```
(define (do-f get-a get-b put-a! put-b!)
  (define-get/put-id a get-a put-a!)
  (define-get/put-id b get-b put-b!)
  (swap a b))
```

How can `define-for-cbr` make `get-` and `put-!` names?

Call-by-Reference Body

A name-generation trick:

```
(define-syntax define-for-cbr
  (syntax-rules ()
    [(define-for-cbr do-f (id0 id ...)
       (gens ...) body)
     (define-for-cbr do-f (id ...)
       (gens ... (id0 get put)) body)]
    [(define-for-cbr do-f ()
       ((id get put) ...) body)
     (define (do-f get ... put ...)
       (define-get/put-id id get put) ...
       body)]))
```

Call-by-Reference Body

More accurate description of the expansion:

```
(define-for-cbr do-f (a b)
  () (swap a b))
```

⇒

```
(define (do-f get¹ get² put¹ put²)
  (define-get/put-id a get¹ put¹)
  (define-get/put-id b get² put²)
  (swap a b))
```

Complete Code to Add Call-By-Reference

```
(define-syntax define-cbr
  (syntax-rules ()
    [(_ (id arg ...) body)
     (begin
       (define-for-cbr do-f (arg ...)
         () body)
       (define-syntax id
         (syntax-rules ()
           [(id actual (... ...))
            (do-f (lambda () actual)
                  (... ...)
                  (lambda (v)
                    (set! actual v))
                  (... ...))]))))]))
```

```
(define-syntax define-for-cbr
  (syntax-rules ()
    [(define-for-cbr do-f (id0 id ...)
       (gens ...) body)
     (define-for-cbr do-f (id ...)
       (gens ... (id0 get put)) body)]
    [(define-for-cbr do-f ()
       ((id get put) ...) body)
     (define (do-f get ... put ...)
       (define-get/put-id id get put) ...
       body)]))
```

```
(define-syntax define-get/put-id
  (syntax-rules ()
    [(define-get/put-id id get put!)
     (define-syntax id
       (syntax-id-rules (set!)
         [(set! id e) (put! e)]
         [(id a (... ...)) ((get) a (... ...))]
         [id (get)]))]))
```

Part 9

Modules

Modules can export and import macros

cbr.rkt

```
(provide define-cbr)

....
```

f.rkt

```
(require "cbr.rkt")

(define-cbr (f x y)
  ....)

(let ([a 0] [b 1])
  (f a b))
```

Modules

Modules can export and import macros

cbr.rkt

```
(provide define-cbr)

....
```

f.rkt

```
(require "cbr.rkt")

(define-cbr (f x y)
  ....)

(provide f)
```

g.rkt

```
(require "f.rkt")

(let ([a 0] [b 1])
  (f a b))
```

Renaming Exports

The **`provide`** form supports renaming

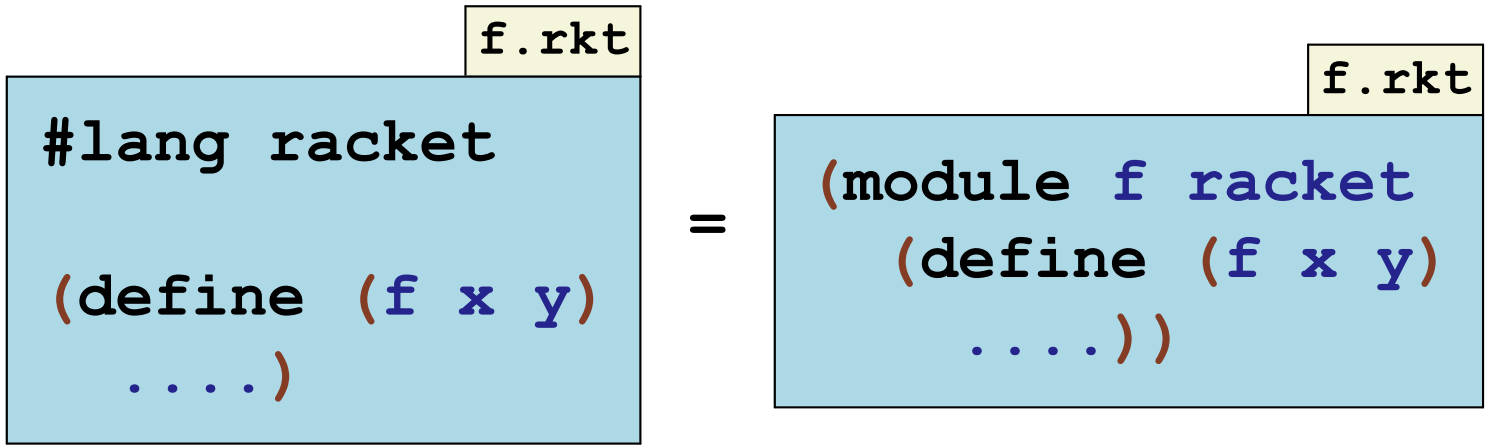
`cbr.rkt`

```
(provide
 (rename-out
  [define-cbr define]))

....
```

`f.rkt`

```
(require "cbr.rkt")

(define (f x y)
  ....)

(let ([a 0] [b 1])
  (f a b))
```

Modules and `#lang`



Adjusting a Language

The **`provide`** form has its own sublanguage...

racket-cbr.rkt

```
#lang racket

(provide (except-out (all-from-out racket)
                     define)
         (rename-out [define-cbr define]))

...
```

f.rkt

```
(module f "racket-cbr.rkt"
  (define (f x y)
    ....))
```

Adjusting a Language

The **`provide`** form has its own sublanguage...

racket-cbr.rkt

```
#lang racket

(provide (except-out (all-from-out racket)
                     define)
         (rename-out [define-cbr define]))

...
```

f.rkt

```
#lang s-exp "racket-cbr.rkt"

(define (f x y)
  ....)
```