Part I

Method Macro

Suppose that we want

```
(method (+ arg (get this x)))
```

to expand to

```
(lambda (this arg)
  (+ arg (get this x)))
```

Method Macro

```
(define-syntax-rule (method expr)
  (lambda (this arg)
    expr))
```

Doesn't work:

```
(method (+ arg (get this x)))
```

⇒

```
(lambda¹ (this¹ arg¹)
  (+ arg (get this x)))
```

Lexical Context

```
(lambda¹ (this¹ arg¹)
  (+ arg (get this x)))
```

The superscripts correspond to ***lexical context***

We can use `datum->syntax` to manipulate lexical context

Non-Hygienic Macros

```
(define-syntax (method stx)
  (syntax-parse stx
    [(method expr)
     (define this-var (datum->syntax #'expr 'this))
     (define arg-var (datum->syntax #'expr 'arg))
     #`(lambda (#,this-var #,arg-var)
         expr)]))
```

Using `(datum->syntax #'expr 'this)` gives

`this` the lexical context of `#'expr`

Non-Hygienic Macros

```
(define-syntax (method stx)
  (syntax-parse stx
    [(method expr)
     (define this-var (datum->syntax #'expr 'this))
     (define arg-var (datum->syntax #'expr 'arg))
     #`(lambda (#,this-var #,arg-var)
         expr)]))
```

```
(method (+ arg (get this x)))
```

⇒

```
(lambda¹ (this arg)
  (+ arg (get this x)))
```

Part 2

Non-Hygienic Macros and Composition

```
(define-syntax-rule (method-plus-one expr)
  (method (+ 1 expr)))

(method-plus-one arg)
```

⇒

```
(method (+² 1² arg))
```

⇒

```
(lambda³ (this² arg²)
  (+² 1² arg))
```

Part 3

Alternative to Non-Hygienic Macros

Bind `this` and `arg` as macros in the same scope as `method`:

```
(define-syntax method ....)
(define-syntax this ....)
(define-syntax arg ....)
```

Make the `this` and `arg` macros cooperate with the `method` macro

Syntax Parameters

The **`racket/stxparam`** library provides

- **`define-syntax-parameter`** — to introduce a cooperation channel
- **`syntax-parameterize`** — to send through the channel
- **`syntax-parameter-value`** — to receive from the channel

Syntax Parameters

```
(define-syntax-parameter this-name #f)
(define-syntax-parameter arg-name #f)

(define-syntax this
  (lambda (stx)
    (define v (syntax-parameter-value #'this-name))
    (if v
        v
        (raise-syntax-error 'this
                            "illegal outside of method"
                            stx))))
(define-syntax arg ....)

(define-syntax-rule (method expr)
  (lambda (this-n arg-n)
    (syntax-parameterize ([this-name #'this-n]
                          [arg-name #'arg-n])
      expr)))
```

Part 4

Class Macro

Let's implement `class` as a macro:

```
{class posn
  {x y}
  {mdist {+ {get this y}
            {get this x}}}
  {addDist {+ {send this mdist 0}
              {send arg mdist 0}}}}
```

Pattern:

```
(define-syntax (class stx)
  (syntax-parse stx
    [{class name:id
       {fld:id ...}
       {mthd-name:id mthd-expr} ...}
     ....]))
```

Class Macro

Template:

```
(define-syntax (class stx)
  (syntax-parse stx
    [(class name:id
       {fld:id ...}
       {mthd-name:id mthd-expr} ...}
     ....]))
```

Class Macro

Template:

```
(define-syntax (class stx)
  (syntax-parse stx
    [(class name:id
       {fld:id ...}
       {mthd-name:id mthd-expr} ...)
     #'(...
        (record [fld ???]
                ...)
        (record [mthd-name (method mthd-expr)]
                ...))]))
```

Class Macro

Template:

```
(define-syntax (class stx)
  (syntax-parse stx
    [(class name:id
       {fld:id ...}
       {mthd-name:id mthd-expr} ...)
     #'(...
        (lambda (fld ...)
          (record [fld fld]
                  ...))
        (record [mthd-name (method mthd-expr)]
                ...))]))
```

Class Macro

Template:

```
(define-syntax (class stx)
  (syntax-parse stx
    [(class name:id
       {fld:id ...}
       {mthd-name:id mthd-expr} ...)
     #'(classV
        (lambda (fld ...)
          (record [fld fld]
                  ...))
        (record [mthd-name (method mthd-expr)]
                ...))]))

(struct classV (constructor methods))
```

Class Macro

Template:

```
(define-syntax (class stx)
  (syntax-parse stx
    [{class name:id
       {fld:id ...}
       {mthd-name:id mthd-expr} ...}
     #'(define name
         (classV
          (lambda (fld ...)
            (record [fld fld]
                    ...))
          (record [mthd-name (method mthd-expr)]
                  ...)))]))

(struct classV (constructor methods))
```

`new` Macro

```
{new posn 1 2}
```

```
(define-syntax (new stx)
  (syntax-parse stx
    [{new cls:id expr ...}
     ....]))
```

`new` Macro

```
{new posn 1 2}
```

```
(define-syntax (new stx)
  (syntax-parse stx
    [{new cls:id expr ...}
     .... (classV-constructor cls)
     ....]))
```

`new` Macro

```
{new posn 1 2}
```

```
(define-syntax (new stx)
  (syntax-parse stx
    [{new cls:id expr ...}
     .... ((classV-constructor cls) expr ...)
     ....]))
```

`new` Macro

```
{new posn 1 2}
```

```
(define-syntax (new stx)
  (syntax-parse stx
    [{new cls:id expr ...}
     #'(objectV
        ....
        ((classV-constructor cls) expr ...))]))
```

`new` Macro

```
{new posn 1 2}
```

```
(define-syntax (new stx)
  (syntax-parse stx
    [{new cls:id expr ...}
     #'(objectV
        cls
        ((classV-constructor cls) expr ...))]))

(struct objectV (class fields))
```

Part 5

`get` Macro

```
{get this x}
```

```
(define-syntax (get stx)
  (syntax-parse stx
    [{get obj-expr fld:id}
     ....]))
```

`get` Macro

```
{get this x}
```

```
(define-syntax (get stx)
  (syntax-parse stx
    [{get obj-expr fld:id}
     .... (objectV-fields obj-expr) ....]))
```

`get` Macro

```
{get this x}
```

```
(define-syntax (get stx)
  (syntax-parse stx
    [{get obj-expr fld:id}
     #'(hash-ref (objectV-fields obj-expr)
                 'fld)]))
```

`send` Macro

```
{send this mdist 0}
```

```
(define-syntax (send stx)
  (syntax-parse stx
    [{send obj-expr mthd:id arg-expr}
     ....]))
```

`send` Macro

```
{send this mdist 0}
```

```
(define-syntax (send stx)
  (syntax-parse stx
    [{send obj-expr mthd:id arg-expr}
     #'(let ([obj obj-expr])
         ....)]))
```

`send` Macro

```
{send this mdist 0}
```

```
(define-syntax (send stx)
  (syntax-parse stx
    [{send obj-expr mthd:id arg-expr}
     #'(let ([obj obj-expr])
         .... (objectV-class obj)
         ....)]))
```

`send` Macro

```
{send this mdist 0}
```

```
(define-syntax (send stx)
  (syntax-parse stx
    [{send obj-expr mthd:id arg-expr}
     #'(let ([obj obj-expr])
         .... (classV-methods
               (objectV-class obj))
         ....)]))
```

`send` Macro

```
{send this mdist 0}
```

```
(define-syntax (send stx)
  (syntax-parse stx
    [{send obj-expr mthd:id arg-expr}
     #'(let ([obj obj-expr])
         ....
         (hash-ref (classV-methods
                    (objectV-class obj))
                   'mthd)
         ....)]))
```

`send` Macro

```
{send this mdist 0}
```

```
(define-syntax (send stx)
  (syntax-parse stx
    [{send obj-expr mthd:id arg-expr}
     #'(let ([obj obj-expr])
         ((hash-ref (classV-methods
                     (objectV-class obj))
                    'mthd)
          obj
          arg-expr))]))
```

`ssend` Macro

```
{ssend this posn mdist 0}
```

```
(define-syntax (ssend stx)
  (syntax-parse stx
    [{ssend obj-expr cls:id mthd:id arg-expr}
     ....]))
```

`ssend` Macro

```
{ssend this posn mdist 0}
```

```
(define-syntax (ssend stx)
  (syntax-parse stx
    [{ssend obj-expr cls:id mthd:id arg-expr}
     #'(let ([obj obj-expr])
         ....)]))
```

`ssend` Macro

```
{ssend this posn mdist 0}
```

```
(define-syntax (ssend stx)
  (syntax-parse stx
    [{ssend obj-expr cls:id mthd:id arg-expr}
     #'(let ([obj obj-expr])
         .... (classV-methods cls)
         ....)]))
```

`ssend` Macro

```
{ssend this posn mdist 0}
```

```
(define-syntax (ssend stx)
  (syntax-parse stx
    [{ssend obj-expr cls:id mthd:id arg-expr}
     #'(let ([obj obj-expr])
         ....
         (hash-ref (classV-methods cls)
                   'mthd)
         ....)]))
```

`ssend` Macro

```
{ssend this posn mdist 0}
```

```
(define-syntax (ssend stx)
  (syntax-parse stx
    [{ssend obj-expr cls:id mthd:id arg-expr}
     #'(let ([obj obj-expr])
         ((hash-ref (classV-methods cls)
                    'mthd)
          obj
          arg-expr))]))
```

Part 6

Class Language

posn.rkt

```
#lang s-exp "class-lang.rkt"

{class posn
  {x y}
  {mdist {+ {get this y}
            {get this x}}}
  {addDist {+ {send this mdist 0}
              {send arg mdist 0}}}}

{class posn3D
  {x y z}
  {mdist {+ {get this z}
            {ssend this posn mdist arg}}}
  {addDist {ssend this posn addDist arg}}}

{send {new posn3D 5 3 1} addDist {new posn 2 7}}
```

Class Language

class-lang.rkt

```
#lang racket
(require "method-macro.rkt"
         "class-macro.rkt")

(provide this arg class new get send ssend)

....
```

Still need:

- `+` and `*`
- numbers
- program is `class`es then one expression

`+` and `*`

```
{+ 1 2}
```

We don't want to provide `#%app`

+ and *

```
{+ 1 2}
```

```
(provide (rename-out [+-form +]
                     [*-form *]))

(define-syntax-rule (+-form l r) (+ l r))
(define-syntax-rule (*-form l r) (* l r))
```

Numbers

```
1
```

We don't want to allow strings, booleans, etc.

Numbers

A literal is implicitly wrapped by **`#%datum`**:

```
1
⇒
(#%datum . 1) ⇒ '1

"abc"
⇒
(#%datum . "abc") ⇒ '"abc"
```

Numbers

```
(provide (rename-out [datum #%datum]))

(define-syntax (datum stx)
  (syntax-parse stx
    [(datum . v:number)
     #'(#%datum . v)]
    [(datum . other)
     (raise-syntax-error #f
                         "bad syntax"
                         #'other)]))
```

Part 7

Module Body

```
#lang s-exp "class-lang.rkt"

{class name_1 ....}

{class name_2 ....}

expr
```

Module Body

```
#lang s-exp "class-lang.rkt"
....
```

⇒

```
(module name "class-lang.rkt"
  (#%module-begin
   ....))
```

Module Body

```
(provide (rename-out [module-begin
                      #%module-begin]))

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

Module Body

```
(provide (rename-out [module-begin
                      #%module-begin]))

(define-syntax (module-begin stx)
  (syntax-parse stx
    #:literals (class)
    [(module-begin {class _ ...} ... expr)
     ....]))
```

Module Body

```
(provide (rename-out [module-begin
                      #%module-begin]))

(define-syntax (module-begin stx)
  (syntax-parse stx
    #:literals (class)
    [(module-begin {class _ ...} ... expr)
     (syntax-parse #'expr
       #:literals (class)
       [{class _ ...}
        ....]
       [_
        ....])]))
```

Module Body

```
(provide (rename-out [module-begin
                      #%module-begin]))

(define-syntax (module-begin stx)
  (syntax-parse stx
    #:literals (class)
    [(module-begin {class _ ...} ... expr)
     (syntax-parse #'expr
       #:literals (class)
       [{class _ ...}
        (raise-syntax-error
         'module-begin
         "need an expression after classes")]
       [_
        ....])]))
```

Module Body

```
(provide (rename-out [module-begin
                      #%module-begin]))

(define-syntax (module-begin stx)
  (syntax-parse stx
    #:literals (class)
    [(module-begin {class _ ...} ... expr)
     (syntax-parse #'expr
       #:literals (class)
       [{class _ ...}
        (raise-syntax-error
         'module-begin
         "need an expression after classes")]
       [_
        (datum->syntax
         #'here
         (cons #'#%module-begin
               (rest (syntax-e stx))))])]))
```