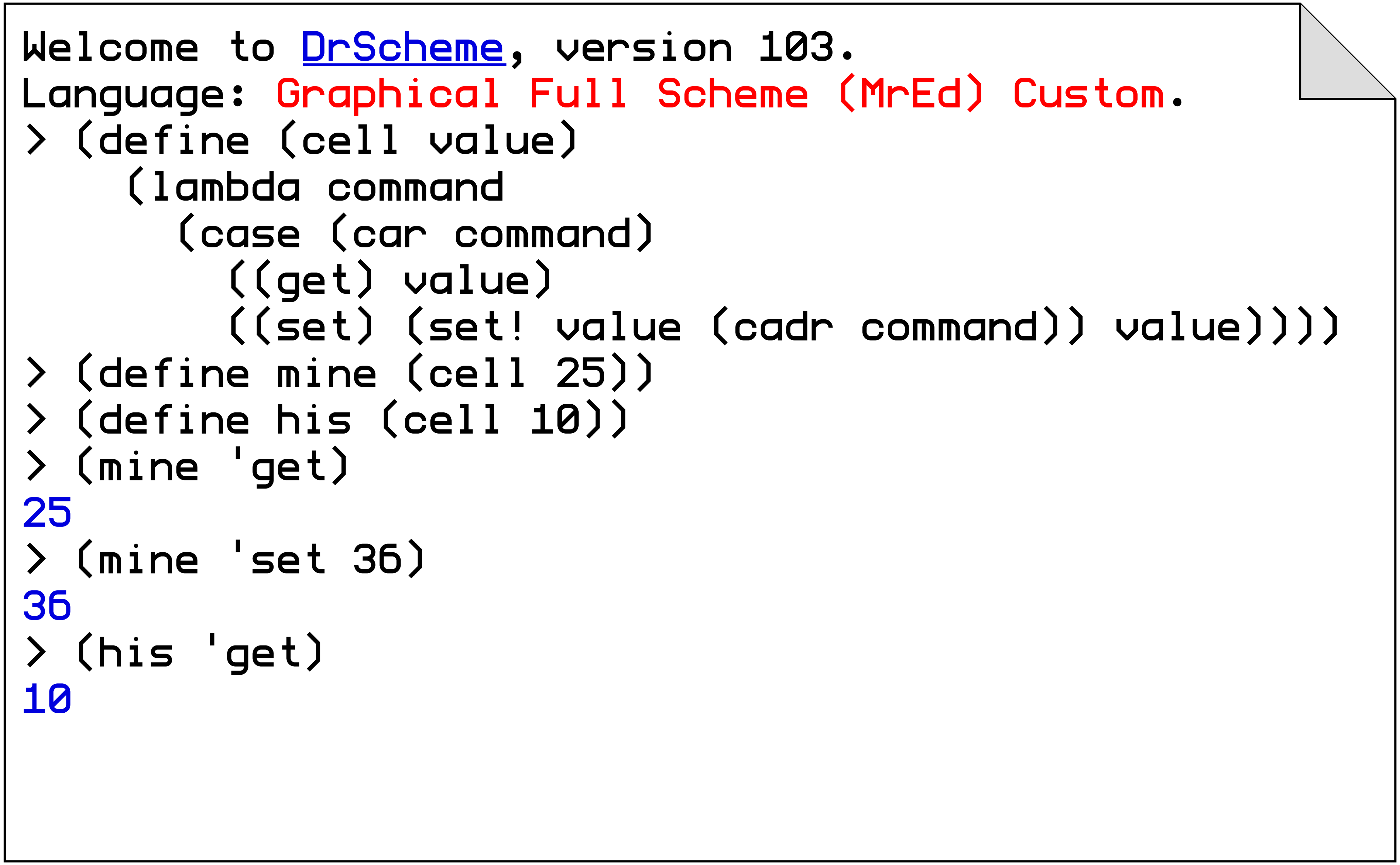
Vrije Universiteit Brussel
Faculty of Sciences
Computer Science Department

Principles of Object Oriented Languages

Lecture 2: Modeling Class-based objects

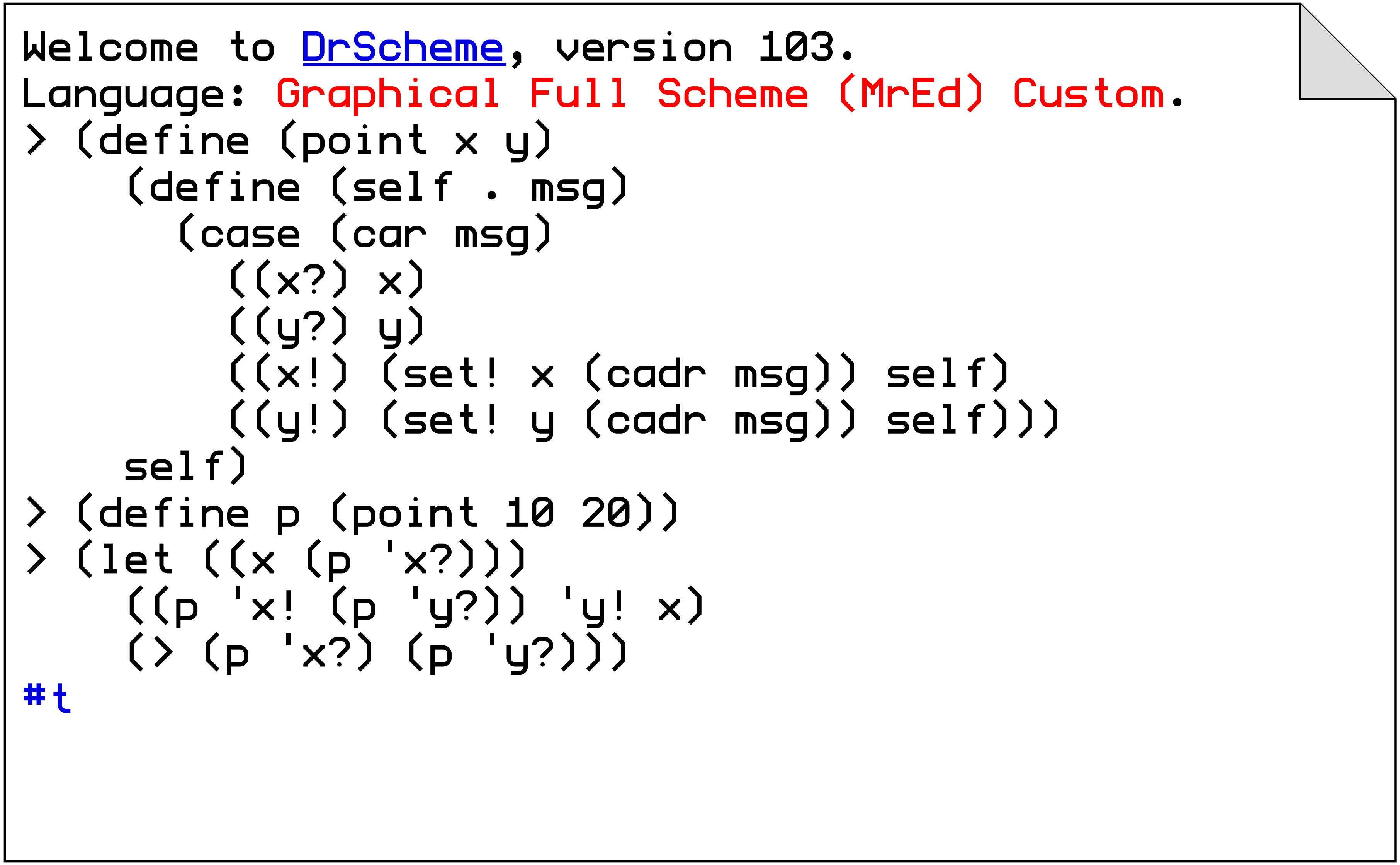
Theo D'Hondt
Programming Technology Lab

Closures



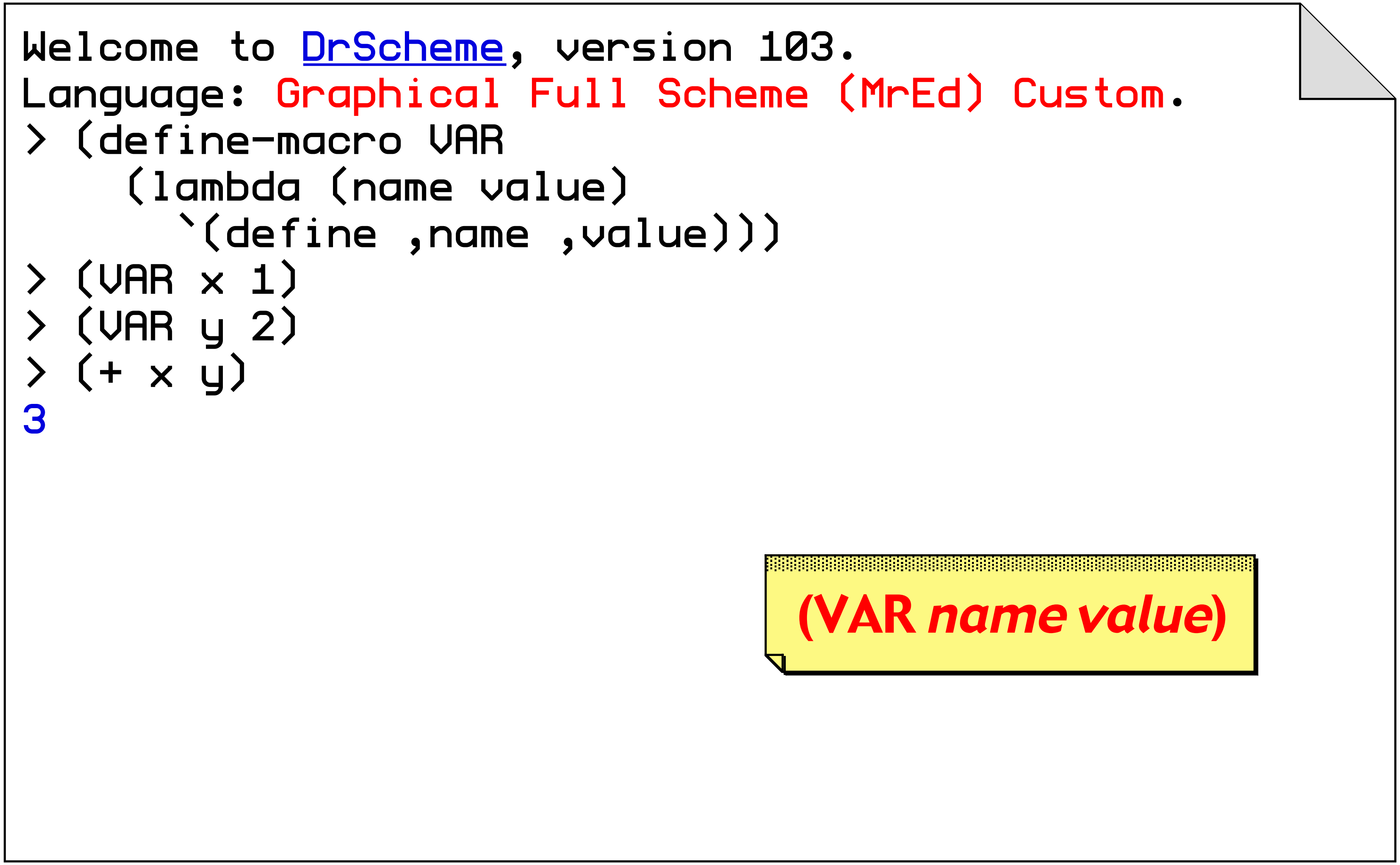
```
Welcome to DrScheme, version 103.
Language: Graphical Full Scheme (MrEd) Custom.
> (define (cell value)
    (lambda command
      (case (car command)
        ((get) value)
        ((set) (set! value (cadr command)) value))))
> (define mine (cell 25))
> (define his (cell 10))
> (mine 'get)
25
> (mine 'set 36)
36
> (his 'get)
10
```

Objects



```
Welcome to DrScheme, version 103.
Language: Graphical Full Scheme (MrEd) Custom.
> (define (point x y)
    (define (self . msg)
      (case (car msg)
        ((x?) x)
        ((y?) y)
        ((x!) (set! x (cadr msg)) self)
        ((y!) (set! y (cadr msg)) self)))
    self)
> (define p (point 10 20))
> (let ((x (p 'x?)))
    ((p 'x! (p 'y?)) 'y! x)
    (> (p 'x?) (p 'y?)))
#t
```

Object system macro's



```
Welcome to DrScheme, version 103.
Language: Graphical Full Scheme (MrEd) Custom.
> (define-macro VAR
    (lambda (name value)
      `(define ,name ,value)))
> (VAR x 1)
> (VAR y 2)
> (+ x y)
3
```

(VAR *name value*)

Object system macro's

```
> (define-macro METHOD
    (lambda (msg args . body)
      `(set! «TAB»
             (cons
              (cons ',msg
                    (lambda ,args ,@body))
              «TAB»))))
> (define «TAB» '())
> (VAR x 10)
> (METHOD x? () x)
> (METHOD x! (X) (set! x X))
> «TAB»
((x! . #<procedure>) (x? . #<procedure>))
> ((cdr (assoc 'x? «TAB»)))
10
> ((cdr (assoc 'x! «TAB»)) 30)
> ((cdr (assoc 'x? «TAB»)))
30
```

(METHOD *name arguments body*)

Object system macro's

```
> (define-macro OBJECT
    (lambda defs
      `(let ((«TAB» '()))
         (define (SELF msg . args)
           (apply (cdr (assoc msg «TAB»)) args))
         ,@defs
         SELF)))
> (define point
    (OBJECT
     (VAR x 0)
     (VAR y 0)
     (METHOD x? () x)
     (METHOD y? () y)
     (METHOD x! (X) (set! x X) SELF)
     (METHOD y! (Y) (set! y Y) SELF)))
> (+ (((point 'x! 10) 'y! 20) 'x?) (point 'y?))
30
```

(OBJECT *vars-and-methods*)

Object system macro's

```
(OBJECT
 (VAR x 0)
 (VAR y 0)
 (METHOD x? () x)
 (METHOD y? () y)
 (METHOD x! (X) (set! x X) SELF)
 (METHOD y! (Y) (set! y Y) SELF))
```

```
(let ((«TAB» '()))
  (define (SELF msg . args)
    (apply
     (cdr (assoc msg «TAB»))
     args))
  (VAR x 0)
  (VAR y 0)
  (METHOD x? () x)
  (METHOD y? () y)
  (METHOD x! (X) (set! x X) SELF)
  (METHOD y! (Y) (set! y Y) SELF)
  SELF)
```

Method tables

```
> (define («TABLE»)
    (define tab '())
    (lambda (op key . rest)
      (define entry (assoc key tab))
      (case op
        ((get)
         (if entry (cdr entry) #f))
        ((put)
         (let ((value (car rest)))
           (if entry
               (set-cdr! entry value)
               (set! tab (cons (cons key value)
                               tab))))))))
> (define-macro METHOD
    (lambda (msg args . body)
      `(«METHODS» 'put ',msg (lambda ,args ,@body))))
```

Method tables

```
> (define («TABLE»)
    (define tab '())
    (lambda (op key . rest)
      (define entry (assoc key tab))
      (case op
        ((get)
         (if entry (cdr entry) #f))
        ((put)
         (let ((value (car rest)))
           (if entry
               (set-cdr! entry value)
               (set! tab (cons (cons key value)
                               tab)))))))))
> (define-macro METHOD
    (lambda (msg args . body)
      `(«METHODS» 'put ',msg
```

retrieve value associated with "key"; "rest" is empty

store "key" with value held as single item in "rest"

Class-based system macro's

```
> (define-macro CLASS
    (lambda defs
      `(lambda ()
         (define «METHODS» («TABLE»))
         (define (SELF msg . args)
           (apply («METHODS» 'get msg) args))
         ,@defs SELF)))
> (define Point
    (CLASS
     (VAR x 0)
     (VAR y 0)
     (METHOD x? () x)
     (METHOD y? () y)
     (METHOD x! (X) (set! x X) SELF)
     (METHOD y! (Y) (set! y Y) SELF)))
> (define p (Point))
> (+ (((p 'x! 10) 'y! 20) 'x?) (p 'y?))
30
```

Class-based system macro's

```
(CLASS
 (VAR x 0)
 (VAR y 0)
 (METHOD x? () x)
 (METHOD y? () y)
 (METHOD x! (X) (set! x X) SELF)
 (METHOD y! (Y) (set! y Y) SELF)))
```

```
(lambda ()
  (define «METHODS» («TABLE»))
  (define (SELF msg . args)
    (apply («METHODS» 'get msg))
    args)
  (VAR x 0)
  (VAR y 0)
  (METHOD x? () x)
  (METHOD y? () y)
  (METHOD x! (X) (set! x X) SELF)
  (METHOD y! (Y) (set! y Y) SELF)
  SELF)
```

Class-based system macro's

```
> (define-macro NEW
    (lambda (class)
      `(,class)))
> (define p (NEW Point))
> (define-macro SEND
    (lambda (object msg . args)
      `(,object ',msg ,@args)))
> (SEND (SEND p x! 10) y! 20)
#<procedure:SELF>
> (define q (NEW Point))
> (SEND (SEND q x! (SEND p y?)) y! (SEND p x?))
#<procedure:SELF>
> (list (SEND q x?) (SEND q y?))
(20 10)
```

Polymorphism

```
(define PolarPoint
  (CLASS
   (VAR ro 0)
   (VAR theta 0)
   (METHOD x? () (* ro (cos theta)))
   (METHOD y? () (* ro (sin theta)))
   (METHOD ro? () ro)
   (METHOD theta? () theta)
   (METHOD ro! (Ro) (set! ro Ro) SELF)
   (METHOD theta! (Theta) (set! theta Theta) SELF)
   (METHOD Cartesian () (SEND (SEND (NEW CartesianPoint)
                                    x! (SEND SELF x?))
                          y! (SEND SELF y?)))
   (METHOD Polar () SELF)))
```

```
(define CartesianPoint
  (CLASS
   (VAR x 0)
   (VAR y 0)
   (METHOD x? () x)
   (METHOD y? () y)
   (METHOD ro? () (sqrt (+ (* x x) (* y y))))
   (METHOD theta? () (atan y x))
   (METHOD x! (X) (set! x X) SELF)
   (METHOD y! (Y) (set! y Y) SELF)
   (METHOD Cartesian () SELF)
   (METHOD Polar () (SEND (SEND (NEW PolarPoint)
                                ro! (SEND SELF ro?))
                          theta! (SEND SELF theta?)))))
```

Polymorphism

```
> (define p (NEW CartesianPoint))
> (SEND (SEND p x! 10) y! 20)
#<procedure:SELF>
> (define q (NEW PolarPoint))
> (SEND (SEND q ro! (SEND p ro?))
        theta! (SEND p theta?))
#<procedure:SELF>
> (list (SEND q x?) (SEND q y?))
(10.000000000000004 20.0)
>
```

Composition

```
> (define Point
    (CLASS
     (VAR x 0)
     (VAR y 0)
     (METHOD move (X Y)
             (set! x X)
             (set! y Y)
             SELF)
     (METHOD where () (x . y))
     (METHOD same? (p)
             (equal? (SEND SELF where)
                     (SEND p where)))))
> (define p (SEND (NEW Point) move 10 20))
> (define q (SEND (NEW Point) move 10 20))
> (SEND p same? q)
#t
```

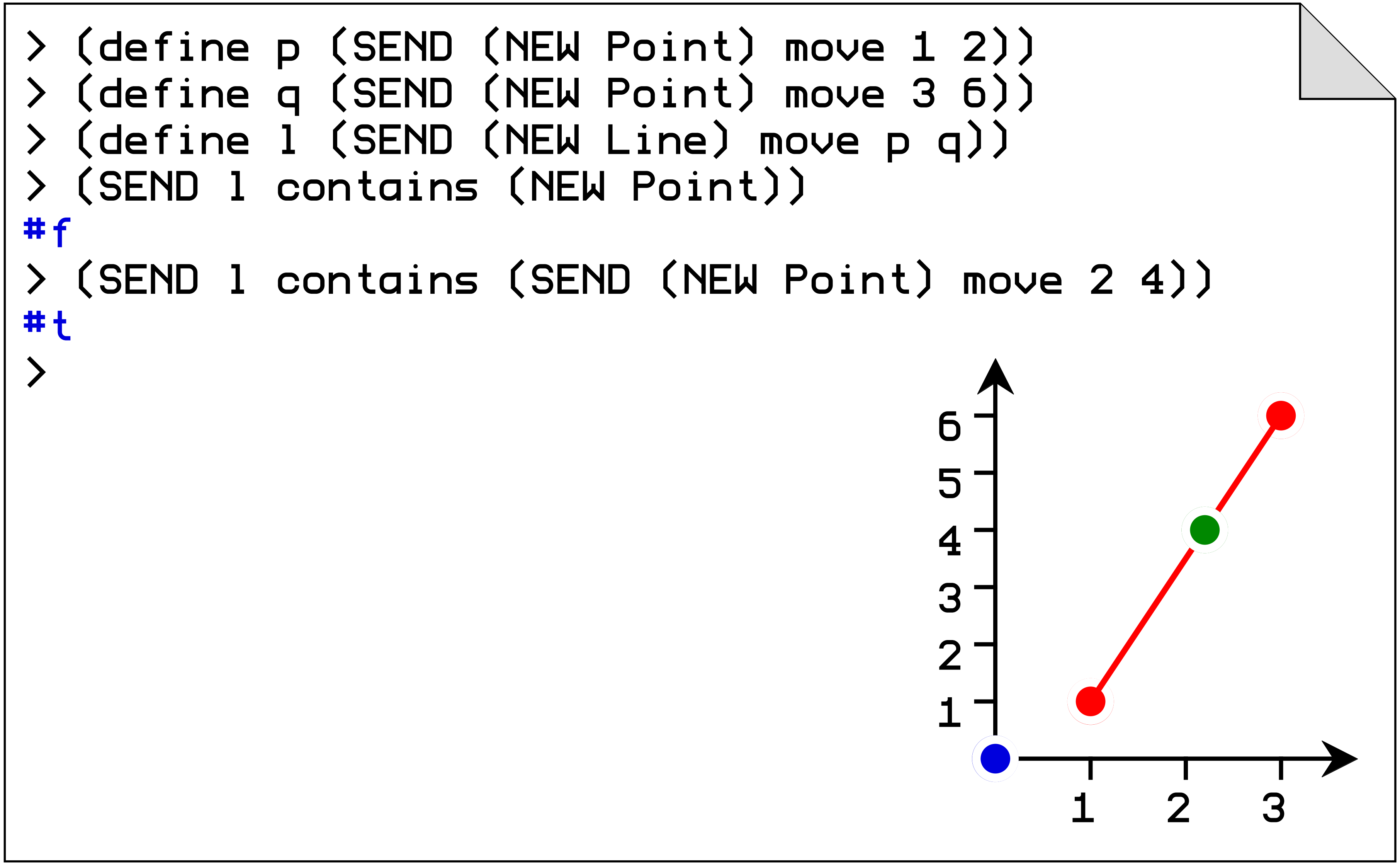
Composition

```
> (define Line
    (CLASS
     (VAR p (NEW Point))
     (VAR q (NEW Point))
     (METHOD move (P Q)
             (set! p P)
             (set! q Q)
             SELF)
     (METHOD where () (cons p q))
     (METHOD same? (L)
             (equal? (SEND SELF where)
                     (SEND L where)))
     (METHOD contains (P)
             (if (SEND p same? q)
                 (SEND p same? P)
                 (let*
```

Composition

```
(METHOD contains (P)
         (if (SEND p same? q)
             (SEND p same? P)
             (let*
                 ((u (SEND p where))
                  (v (SEND q where))
                  (w (SEND P where))
                  (convex?
                   (lambda (alfa)
                     (and (>= alfa 0) (<= alfa 1))))
                  (ratio
                   (lambda (opr)
                     (/ (- (opr w) (opr u))
                        (- (opr v) (opr u)))))
                  (a (ratio car))
                  (b (ratio cdr)))
               (and (< (abs (- a b)) 0.001)
                    (convex? a)))))))
```

Composition



```
> (define p (SEND (NEW Point) move 1 2))
> (define q (SEND (NEW Point) move 3 6))
> (define l (SEND (NEW Line) move p q))
> (SEND l contains (NEW Point))
#f
> (SEND l contains (SEND (NEW Point) move 2 4))
#t
>
```

Delegation

```
> (define-macro CLASS
    (lambda defs
      `(lambda ()
         (define «METHODS» («TABLE»))
         (define (PROXY msg . args)
           (error "method not found"))
         (define (SELF msg . args)
           (define entry («METHODS» 'get msg))
           (if entry
               (apply entry args)
               (apply PROXY (cons msg args))))
         ,@defs
         SELF)))
> (define-macro DELEGATE
    (lambda (object)
      `(set! PROXY ,object)))
```

Delegation

```
> (define Counter
    (CLASS
     (VAR count 0)
     (METHOD incr () (set! count (+ count 1)))
     (METHOD decr () (set! count (- count 1)))
     (METHOD value () count)
     (METHOD reset () (set! count 0))))
> (define ProtectedCounter
    (CLASS
     (VAR count (NEW Counter))
     (VAR max 10)
     (METHOD incr()
             (if (< (SEND count value) max)
                 (SEND PROXY incr)
                 (error "overflow")))
```

Delegation

```
    (METHOD decr()
             (if (> (SEND count value) 0)
                 (SEND PROXY decr)
                 (error "underflow")))
     (METHOD level (Max) (set! max Max))
     (DELEGATE count)))
> (define c (NEW ProtectedCounter))
> (SEND c incr)
> (SEND c reset)
> (SEND c decr)
underflow
```

Delegation

```
(define ProtectedCounter
  (CLASS
   (VAR count (NEW Counter))
   (VAR max 10)
   (METHOD incr ()
     (if (< (SEND count value) max)
       (SEND PROXY incr)
       (error "overflow")))
   (METHOD decr ()
     (if (> (SEND count value) 0)
       (SEND PROXY decr)
       (error "underflow")))
   (METHOD level (Max) (set! max Max))
   (DELEGATE count)))
```

```
(lambda ()
  (define «METHODS» («TABLE»))
  (define (PROXY msg . args)
    (error "method not found"))
  (define (SELF msg . args)
    (define entry («METHODS» 'get msg))
    (if entry
      (apply entry args)
      (apply PROXY (cons msg args))))
  (VAR count (NEW Counter))
  (VAR max 10)
  (METHOD incr ()
    (if (< (SEND count value) max)
      (SEND PROXY incr)
      (error "overflow")))
  (METHOD decr ()
    (if (> (SEND count value) 0)
      (SEND PROXY decr)
      (error "underflow")))
  (METHOD level (Max) (set! max Max))
  (DELEGATE count)
  SELF)
```

Reentrance

- ☑ factor out common behaviour
- ☑ parametrize behaviour with state
- ☑ introduce "state template"
- ☑ introduce "instantiation"
- ☑ qualified variable references

Re

```
(define («TABLE»)
  (define tab '())
  (lambda (op . rest)
    (case op
      ((instantiate)
       (let ((table («TABLE»)))
         (for-each
          (lambda (elt)
            (table 'put (car elt) (eval (cdr elt))))
          tab)
         table))
      ((get)
       (let* ((key (car rest))
              (entry (assoc key tab)))
         (if entry
             (cdr entry)
             #f)))
      ((put)
       (let* ((key (car rest))
              (entry (assoc key tab))
              (value (cadr rest)))
         (if entry
             (set-cdr! entry value)
             (set! tab (cons (cons key value) tab))))))))
```

(<table> 'instantiate)
returns instantiation

Reentrance

variables are stored in a template table

```
(define-macro VAR
  (lambda (name value)
    `(«VARS» 'put ',name ',value)))
```

variable initialization is not (yet) evaluated

a hidden context parameter is inserted

```
(define-macro METHOD
  (lambda (msg args . body)
    `(«METHODS» 'put ',msg
      (lambda («CONTEXT» ,@args) ,@body))))
```

Reentrance

```
(define-macro ?
  (lambda (name)
    `(«CONTEXT» 'get ' ,name)))

(define-macro !
  (lambda (name value)
    `(«CONTEXT» 'put ' ,name ,value)))
```

inside methods, variables are looked up in the hidden context parameter

Reentrance

method and variable template tables are defined outside the class constructor

```
(define-macro CLASS
  (lambda defs
    `(letrec
         ((«METHODS» («TABLE»))
          («VARS» («TABLE»))
          («CLASS»
           (lambda ()
             (define context («VARS» 'instantiate))
             (define (self msg . args)
               (define entry («METHODS» 'get msg))
               (apply entry (cons context args)))
             (context 'put '«SELF» self)
             self)))
       ,@defs
       «CLASS»)))
```

two scope layers are defined for "self" and "class"

Reentrance

context is instantiated: variables are initialized

```
(define-macro CLASS
  (lambda defs
    `(letrec
         ((«METHODS» («TABLE»))
          («VARS» («TABLE»))
          («CLASS»
           (lambda ()
             (define context («VARS» 'instantiate))
             (define (self msg . args)
               (define entry («METHODS» 'get msg))
               (apply entry (cons context args)))
             (context 'put '«SELF» self)
             self)))
       ,@defs
       «CLASS»)))
```

each method evaluation requires insertion of context

Reentrance

```
(define-macro CLASS
  (lambda defs
    `(letrec
        ((«METHODS» («TABLE»))
         («VARS» («TABLE»))
         («CLASS»
          (lambda ()
            (define context («VARS» 'instantiate))
            (define (self msg . args)
              (define entry («METHODS» 'get msg))
              (apply entry (cons context args)))
            (context 'put '«SELF» self)
            self)))
       ,@defs
       «CLASS»)))
```

self reference is also stored in context

```
(define-macro SELF
  (lambda ()
    `(«CONTEXT» 'get '«SELF»)))
```

Reentrance

```
(define Point
  (CLASS
   (VAR x 0)
   (VAR y 0)
   (METHOD move (X Y)
           (! x X)
           (! y Y)
           (SELF))
   (METHOD where () (cons (? x) (? y)))
   (METHOD same? (P)
           (equal? (SEND (SELF) where)
                   (SEND P where)))))
```

```
(letrec
  ((«METHODS» («TABLE»))
   («VARS» («TABLE»))
   («CLASS»
  (lambda ()
    (define context («VARS» 'instantiate))
    (define (self msg . args)
      (define entry («METHODS» 'get msg))
      (apply entry (cons context args)))
      (context 'put '«SELF» self)
      self)))
  (VAR x 0)
  (VAR y 0)
  (METHOD move (X Y)
    (! x X)
    (! y Y)
    (SELF))
  (METHOD where () (cons (? x) (? y)))
  (METHOD same? (P)
    (equal?
      (SEND (SELF) where)
      (SEND P where)))
  «CLASS»)
```

Delegation

```
> (define Point
    (CLASS
     (VAR x 0)
     (VAR y 0)
     (METHOD move (X Y)
             (! x X)
             (! y Y)
             (SELF))
     (METHOD where () (cons (? x) (? y)))
     (METHOD same? (P)
             (equal? (SEND (SELF) where)
                     (SEND P where)))))
> (define p (SEND (NEW Point) move 10 20))
> (SEND p same? (NEW Point))
#f
> (SEND p same? (SEND (NEW Point) move 10 20))
#t
```