5.4.1: Implementation
Method Invocation

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      ...
      (method-app-exp (obj-exp method-name rands)
        (let ((args (eval-rands rands env)
              (obj (eval-expression obj-exp env)))
          (find-method-and-apply
             method-name
             (object->class-name obj) ;obj.class-name
             obj
             args)))
```

E.g., **`send circle2 move(-3,10)`**

super Call

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      ...
      (super-call-exp (method-name rands)
        (let ((args (eval-rands rands env)
              (obj (apply-env env 'self)))
          (find-method-and-apply
            method-name
            (apply-env env '%super) ; special name
            obj
            args)))
```

Note that identity of **super** is determined by calling environment, not object! Otherwise...

`super` Call

… otherwise, **`super`** wouldn't work as expected:

```
class c1 extends object
  method initialize() 1
  method m () 99
class c2 extends c1
  method m () super m (); what happens here?
class c3 extends c2
    method m () super m ()
let o = new c3 ()
in send o m ()
```

We'd have an infinite regress at **`c2.m`** if we called **`super`** on o's parent class (**`c2`**)

Object Creation via **new**

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      ...
      (new-object-exp (class-name rands)
        (let ((args (eval-rands rands env))
              (obj (new-object class-name)))
          (find-method-and-apply ; side-effect
            'initialize class-name obj args)
          obj))                          ; return obj
```

E.g., `new circle(x, y, (sqrt 22))`

The Language : (Simple) Implementation

- Need to write **`elaborate-class-decls!`** and **`find-method-and-apply.`**
- For simple implementation, first is easy:

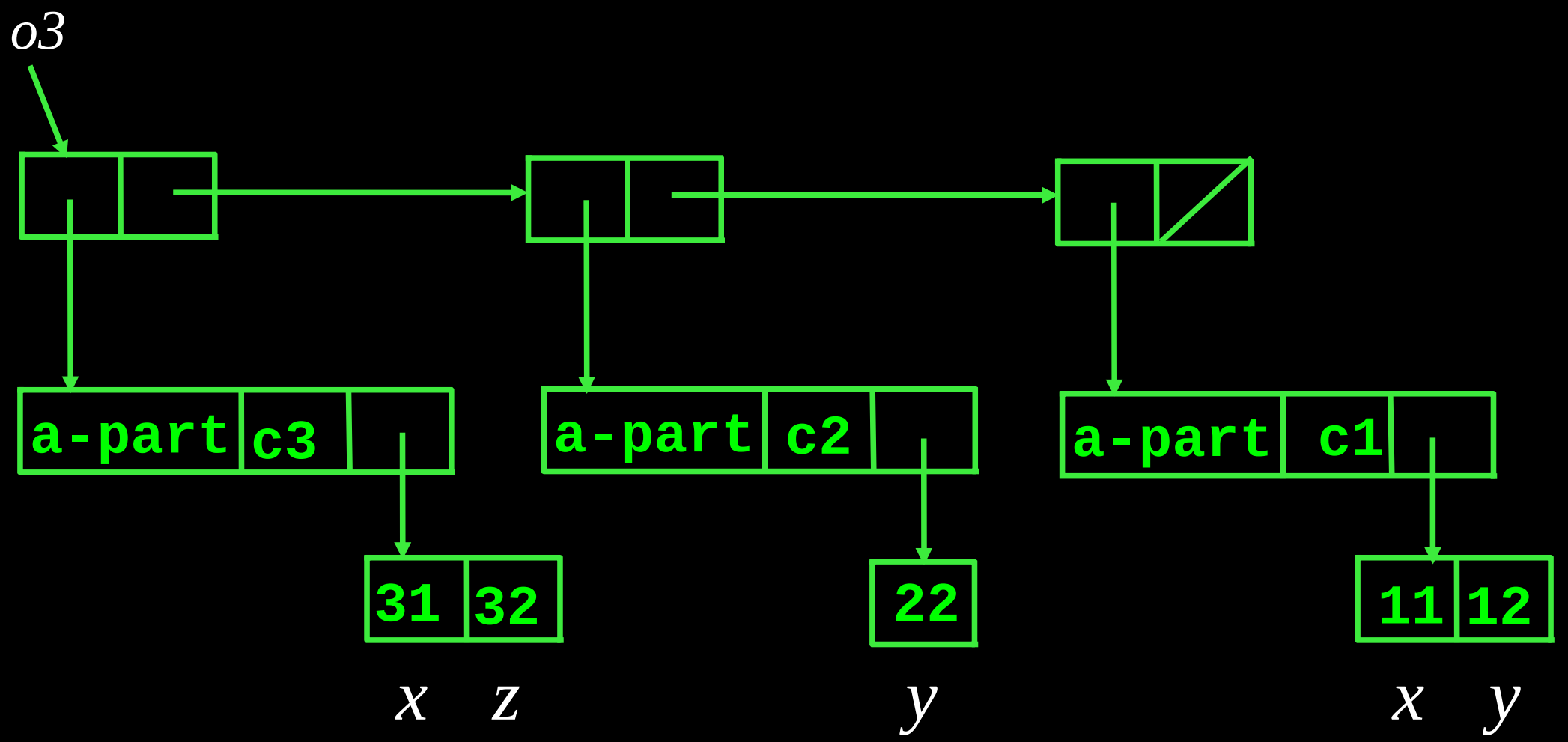
```
(define the-class-env '()) ; why?

(define elaborate-class-decls!
   (lambda (c-decls)
      (set! the-class-env c-decls)))
```

- To understand the rest, we need an example....

```
class c1 extends object
  field x
  field y
  method initialize()
    begin
      set x = 11;
      set y = 12
    end
  method m1 () ... x ... y ...
  method m2 () ... send self m3() ... ; why can we do this?
class c2 extends c1
  field y
  method initialize ()
    begin
      super initialize();
      set y = 22
    end
  method m1(u, v) ... x ... y ...
  method m3() ...
class c3 extends c2
  field x
  field z
  method initialize ()
    begin
      super initialize();
      set x = 31;
      set z = 32
    end
  method m3 () ... x ... y ... z
  let o3 = new c3()
  in send o3 m1(7,8)
```

Can represent **o3** as a “class-chain” of *parts*:



- So we need a datatype for parts:

```
(define-dataype part part?
  (a-part
    (class-name symbol?)
    (fields vector?)))
```

- Use this to build objects recursively:

```
(define new-object
  (lambda (class-name)
    (if (eqv? class-name 'object)
        '() ; why?
        (let ((c-decl (lookup-class ; c-decls is
                        class-name)); global
              (cons (make-first-part c-decl)
                    (new-object       ; recur up
                      (class-decl->super-name
                       c-decl))))))))
```

- Piece-by-piece...

```
(define make-first-part
  (lambda (c-decl)
    (a-part ; constructor
      (class-decl->class-name c-decl)
      (make-vector (length (class-decl->field-ids c-decl)..
```

```
(define class-decl->class-name
  (lambda (c-decl)
    (cases class-decl c-decl
      (a-class-decl
        (class-name super-name field-ids m-decls)
       class-name)))) ; return class_decl.class_name
```

- Now we can write **find-method-and-apply**

```
(define find-method-and-apply
  (lambda (m-name host-name self args)
    (if (eqv? host-name 'object)
        (eopl:error 'find-method-and-apply ; why?
                    "No method for name ~s" m-name)
        (let ((m-decl (lookup-method-decl m-name
                        (class-name->method-decls
                          host-name))))
          (if (method-decl? m-decl) ; if found
              (apply-method m-decl host-name self args)
              (find-method-and-apply m-name
                (class-name->super-name host-name)
                self args)))))) ; recur up class hierarchy
```