Part I

Objects and Constructors

```
(define (numV n)
  (list (values 'apply (lambda (arg-val) ....))
        (values 'number (lambda () ...))))

(define (closV n body c-env)
  (list (values 'apply (lambda (arg-val) ....))
        (values 'number (lambda () ....))))
```

- result of `numV` or `closV` is an ***object***
- functions for `'apply` and `'number` are ***methods***
- `n`, `body`, and `c-env` are ***fields***
- `numV` and `closV` themselves are ***constructors***

... and not far from ***classes***

Classes

Classes play two (dynamic) roles:

- Object construction

```
class Snake {
  ...
}

new Snake("Slinky", 10);
```

- Implementation inheritance

```
class Rattlesnake extends Snake {
  ...
}
```

  - Inheritance of methods
  - Static method dispatch

Classes: Static and Dynamic Dispatch

```
class Snake implements Animal {
  ...
  boolean endangers(Animal a) {
    return (a.slowerThan(100)
            && a.isLighter(this.weight/2));
  }
}

class Rattlesnake extends Snake {
   ...
   boolean endangers(Animal a) {
     return (!a.hasThickSkin()
             || super.endangers(a))
   }
}

Animal a = new Rattlesnake(...);
Animal b = new Mouse(...);

a.endangers(b);
```

dynamic

static

Part 2

Class Language with Explicit Static Calls

```
<Class>   ::=  {class <Symbol>                    NEW
                  {<Field>*}
                  <Method>*}
<Field>   ::=  <Symbol>
<Method>  ::=  [<Symbol> {arg} <Exp>]
```

```
<Exp>  ::=  <Number>
        |   {+ <Exp> <Exp>}
        |   {* <Exp> <Exp>}
        |   arg
        |   this
        |   {new <Symbol> <Exp>*}                  NEW
        |   {get <Exp> <Symbol>}
        |   {send <Exp> <Symbol> <Exp>}
        |   {ssend <Exp> <Symbol> <Symbol> <Exp>}   NEW
```

```
{class Posn
  {x y}
  [mdist {arg} {+ {get this x} {get this y}}]
  [addDist {arg} {+ {send arg mdist 0}
                    {send this mdist 0}}]}

{send {new Posn 1 2}
      addDist
      {new Posn 3 4}}
```

Analogous Java code

```
class Posn {
  int x, y;
  int mdist() {
    return this.x + this.y;
  }
  int addDist(Posn p) {
    return p.mdist() + mdist();
  }
}
new Posn(1,2).mdist(new Posn(3,4))
```

Class Language with Explicit Static Calls

```
<Class>   ::= {class <Symbol>
                {<Field>*}
                <Method>*}
<Field>   ::= <Symbol>
<Method>  ::= [<Symbol> {arg} <Exp>]
```

```
<Exp> ::= <Number>
       |  {+ <Exp> <Exp>}
       |  {* <Exp> <Exp>}
       |  arg
       |  this
       |  {new <Symbol> <Exp>*}
       |  {get <Exp> <Symbol>}
       |  {send <Exp> <Symbol> <Exp>}
       |  {ssend <Exp> <Symbol> <Symbol> <Exp>}
```

```
{class Posn ...
  [addDist {arg} {+ {send arg mdist 0}
                    {send this mdist 0}}]}

{class Posn3D
  {x y z}
  [mdist {arg} {+ {get this z}
                  {ssend this Posn mdist arg}}]
  [addDist {arg} {ssend this Posn addDist arg}]}

{send {new Posn3D 1 2 3}
      addDist
      {new Posn 3 4}}
```

Analogous Java code

```
class Posn {
  ... as before ...
}
class Posn3D extends Posn {
  int z; ...
  int mdist() {
    return this.z + super.mdist();
  }
  int addDist(Posn p) {
    return super.addDist(p);
  }
}
new Posn3D(1,2,3).addDist(new Posn(3,4))
```

Part 3

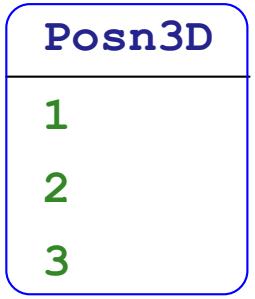
Object Values

How does

```
{send {new Posn3D 1 2 3} mdist ...}
```

dispatch to the right `mdist`?

The result of `{new Posn3D 1 2 3}` can now hold a class tag and field values:



Look for field names and methods in the class

Classes and Object Values

```
(define-type Value
  (numV [n : Number])
  (objV [class-name : Symbol]
        [field-values : (Listof Value)]))

(define-type Class
  (classC [field-names : (Listof Symbol)]
          [methods : (Listof (Symbol * Exp))]))

interp : (Exp (Listof (Symbol * Class)) Value Value
              -> Value)
```

Examples

```
(test (interp (numE 10)
              empty
              (objV 'Object empty)
              (numV 0))
      (numV 10))
```

Examples

```
(define posn-class
  (values 'Posn
          (classC (list 'x 'y)
                  (list
                   (values 'mdist
                           (plusE (getE (thisE) 'x) (getE (thisE) 'y)))
                   (values 'addDist
                           (plusE (sendE (thisE) 'mdist (numE 0))
                                  (sendE (argE) 'mdist (numE 0)))))))))
(define posn3D-class
  (values 'Posn3D
          (classC (list 'x 'y 'z)
                  (list
                   (values 'mdist
                           (plusE (getE (thisE) 'z)
                                  (ssendE (thisE) 'Posn 'mdist (argE))))
                   (values 'addDist
                           (ssendE (thisE) 'Posn 'addDist (argE)))))))

(define (interp-posn a)
  (interp a (list Posn-class Posn3D-class) (objV 'Object empty) (numV 0)))
```

Examples

```
(test (interp-posn (newE 'Posn
                         (list (numE 2)
                               (numE 7))))
      (objV 'Posn
            (list (numV 2)
                  (numV 7))))
```

```
(define new-posn27
  (newE 'Posn (list (numE 2) (numE 7))))
```

Examples

```
(define posn-class
  (values 'Posn
          (classC (list 'x 'y)
                  (list
                   (values 'mdist
                           (plusE (getE (thisE) 'x) (getE (thisE) 'y)))
                   (values 'addDist
                           (plusE (sendE (thisE) 'mdist (numE 0))
                                  (sendE (argE) 'mdist (numE 0)))))))))

(define new-posn27
  (newE 'Posn (list (numE 2) (numE 7))))
```

```
(test (interp-posn (sendE new-posn27 'mdist (numE 0)))
      (numV 9))
```

Examples

```
(define posn-class
  (values 'Posn
          (classC (list 'x 'y)
                  (list
                   (values 'mdist
                           (plusE (getE (thisE) 'x) (getE (thisE) 'y)))
                   (values 'addDist
                           (plusE (sendE (thisE) 'mdist (numE 0))
                                  (sendE (argE) 'mdist (numE 0)))))))))
(define posn3D-class
  (values 'Posn3D
          (classC (list 'x 'y 'z)
                  (list
                   (values 'mdist
                           (plusE (getE (thisE) 'z)
                                  (ssendE (thisE) 'Posn 'mdist (argE))))
                   (values 'addDist
                           (ssendE (thisE) 'Posn 'addDist (argE)))))))
(define new-posn27
  (newE 'Posn (list (numE 2) (numE 7))))
(define new-posn531
  (newE 'Posn3D (list (numE 5) (numE 3) (numE 1))))
```

```
(test (interp-posn (sendE new-posn531 'addDist new-posn27))
      (numV 18))
```

Part 4

Interpreter

```
(define interp : (Exp (Listof (Symbol * Class)) Value Value -> Value)
  (lambda (a classes this-val arg-val)
    (local [(define (recur expr)
              (interp expr classes this-val arg-val))]
      (type-case Exp a
        ...
        [(numE n) (numV n)]
        [(plusE l r) (num+ (recur l) (recur r))]
        [(multE l r) (num* (recur l) (recur r))]
        [(thisE) this-val]
        [(argE) arg-val]
        ...))))
```

Interpreter

```
(define interp : (Exp (Listof (Symbol * Class)) Value Value -> Value)
  (lambda (a classes this-val arg-val)
    (local [(define (recur expr)
              (interp expr classes this-val arg-val))]
      (type-case Exp a
        ...
        [(newE class-name field-exprs)
         (local [(define c (find classes class-name))
                 (define vals (map recur field-exprs))]
           (if (= (length vals) (length (classC-field-names c)))
               (objV class-name vals)
               (error 'interp "wrong field count")))]
        ...))))
```

Interpreter

```
(define interp : (Exp (Listof (Symbol * Class)) Value Value -> Value)
  (lambda (a classes this-val arg-val)
    (local [(define (recur expr)
              (interp expr classes this-val arg-val))]
      (type-case Exp a
        ...
        [(getE obj-expr field-name)
         (type-case Value (recur obj-expr)
           [(objV class-name field-vals)
            (type-case Class (find class-name classes)
              [(classC field-names methods)
               (find (map2 (lambda (n v) (values n v))
                           field-names
                           field-vals)
                     field-name)])]
           [else (error 'interp "not an object")])]
        ...))))
```

Interpreter

```
(define interp : (Exp (Listof (Symbol * Class)) Value Value -> Value)
  (lambda (a classes this-val arg-val)
    (local [(define (recur expr)
              (interp expr classes this-val arg-val))]
      (type-case Exp a
        ...
        [(sendE obj-expr method-name arg-expr)
         (local [(define obj (recur obj-expr))
                 (define arg-val (recur arg-expr))]
           (type-case Value obj
             [(objV class-name field-vals)
              (call-method class-name method-name classes
                           obj arg-val)]
             [else (error 'interp "not an object")]))]
        ...))))
```

Calling a Method

```
(define (call-method class-name method-name classes
                     obj arg-val)
  (type-case Class (find classes class-name)
    [(classC field-names methods)
     (let ([body-expr (find methods method-name)])
       (interp body-expr
               classes
               obj
               arg-val))]))
```

Interpreter

```
(define interp : (Exp (Listof (Symbol * Class)) Value Value -> Value)
  (lambda (a classes this-val arg-val)
    (local [(define (recur expr)
              (interp expr classes this-val arg-val))]
      (type-case Exp a
        ...
        [(ssendE obj-expr class-name method-name arg-expr)
         (local [(define obj (recur obj-expr))
                 (define arg-val (recur arg-expr))]
           (call-method class-name method-name classes
                        obj arg-val))]
        ...))))
```

Part 5

Subclasses

Subclasses with **`Exp`**:

```
{class Posn
  {x y}
  [mdist {arg} {+ {get this x} {get this y}}]
  [addDist {arg} {+ {send arg mdist 0}
                    {send this mdist 0}}]}

{class Posn3D
  {x y z}
  [mdist {arg} {+ {get this z}
                  {ssend this Posn mdist arg}}]
  [addDist {arg} {+ {send arg mdist 0}
                    {send this mdist 0}}]}

{send {new Posn3D 1 2 3} addDist {new Posn 3 4}}
```

Programmer manually

- duplicates fields
- implements method inheritance

Subclasses

`ExpI` adds ***implementation inheritance***:

```
{class Posn extends Object
  {x y}
  [mdist {arg} {+ {get this x} {get this y}}]
  [addDist {arg} {+ {send arg mdist 0}
                    {send this mdist 0}}]}

{class Posn3D extends Posn
  {z}
  [mdist {arg} {+ {get this z}
                  {super mdist arg}}]}

{send {new Posn3D 1 2 3} addDist {new Posn 3 4}}
```

Class Language with Inheritance

```
<Class>   ::=  {class <Symbol> extends <Symbol>          NEW
                  {<Field>*}
                  <Method>*}
<Field>   ::=  <Symbol>
<Method>  ::=  [<Symbol> {arg} <Exp>]

<Exp>  ::=  <Number>
        |   {+ <Exp> <Exp>}
        |   {* <Exp> <Exp>}
        |   arg
        |   this
        |   {new <Symbol> <Exp>*}
        |   {get <Exp> <Symbol>}
        |   {send <Exp> <Symbol> <Exp>}
        |   {super <Symbol> <Exp>}                     NEW
```

Compiling Inheritance

```
{class Posn extends Object
  {x y}
  [mdist {arg} {+ {get this x} {get this y}}]
  [addDist {arg} {+ {send arg mdist 0}
                    {send this mdist 0}}]}

{class Posn3D extends Posn
  {z}
  [mdist {arg} {+ {get this z}
                  {super mdist arg}}]}

{send {new Posn3D 1 2 3} addDist {new Posn 3 4}}
```

```
{class Posn
  {x y}
  [mdist {arg} {+ {get this x} {get this y}}]
  [addDist {arg} {+ {send arg mdist 0}
                    {send this mdist 0}}]}

{class Posn3D
  {x y z}
  [mdist {arg} {+ {get this z}
                  {ssend this Posn mdist arg}}]
  [addDist {arg} {+ {send arg mdist 0}
                    {send this mdist 0}}]}

{send {new Posn3D 1 2 3} addDist {new Posn 3 4}}
```

- merge fields from superclasses
- change **`super`** to **`ssend`**
- merge/override methods

Part 6

Classes

```
(define-type ClassI
  (classI [super-name : Symbol]
          [field-names : (Listof Symbol)]
          [methods : (Listof (Symbol * ExpI))]))
```

Expressions

```
(define-type ExpI
  (numI [n : Number])
  (plusI [lhs : ExpI]
         [rhs : ExpI])
  (multI [lhs : ExpI]
         [rhs : ExpI])
  (argI)
  (thisI)
  (newI [class-name : Symbol]
        [args : (Listof ExpI)])
  (getI [obj-expr : ExpI]
        [field-name : Symbol])
  (sendI [obj-expr : ExpI]
         [method-name : Symbol]
         [arg-expr : ExpI])
  (superI [method-name : Symbol]
          [arg-expr : ExpI]))
```

Examples

```
(test (exp-i->c (numI 10))
      (numE 10))
```

Examples

```
(test (exp-i->c (thisI))
      (thisE))
```

Examples

```
(test (exp-i->c (superI 'mdist (numI 0)))
      (ssendE (thisE) ??? 'mdist (numE 0)))
```

Examples

```
exp-i->c : (ExpI Symbol -> Exp)

(test (exp-i->c (superI 'mdist (numI 0)) 'Posn)
      (ssendE (thisE) 'Posn 'mdist (numE 0)))
```

Compiling Expressions

```
(define (exp-i->c [a : ExpI] [super-name : Symbol]) : Exp
  (local [(define (recur expr)
            (exp-i->c expr super-name))]
    (type-case ExpI a
      [(numI n) (numE n)]
      [(plusI l r) (plusE (recur l) (recur r))]
      [(multI l r) (multE (recur l) (recur r))]
      ...
      [(superI method-name arg-expr)
       (ssendE (thisE)
               super-name
               method-name
               (recur arg-expr))])))
```

Compiling Class Methods

```
(define (class-i->c-not-flat [c : ClassI]) : Class
  (type-case ClassI c
    [(classI super-name field-names methods)
     (classC
      field-names
      (map (lambda (m)
             (values (fst m)
                     (exp-i->c (snd m) super-name)))
           methods))]))
```

Flattening a Class

```
(define (flatten-class [name : Symbol]
                       [classes-not-flat : (Listof (Symbol * Class))]
                       [i-classes : (Listof (Symbol * ClassI))]) : Class
  ...)
```

Flattening a Class

```
(define (flatten-class [name : Symbol]
                       [classes-not-flat : (Listof (Symbol * Class))]
                       [i-classes : (Listof (Symbol * ClassI))]) : Class
  ... (find classes-not-flat name) ...)
```

Flattening a Class

```
(define (flatten-class [name : Symbol]
                       [classes-not-flat : (Listof (Symbol * Class))]
                       [i-classes : (Listof (Symbol * ClassI))]) : Class
  (type-case Class (find classes-not-flat name)
    [(classC field-names methods)
     ...]))
```

Flattening a Class

```
(define (flatten-class [name : Symbol]
                       [classes-not-flat : (Listof (Symbol * Class))]
                       [i-classes : (Listof (Symbol * ClassI))]) : Class
  (type-case Class (find classes-not-flat name)
    [(classC field-names methods)
     ... (flatten-super name classes-not-flat i-classes) ...]))
```

Flattening a Class

```
(define (flatten-class [name : Symbol]
                       [classes-not-flat : (Listof (Symbol * Class))]
                       [i-classes : (Listof (Symbol * ClassI))]) : Class
  (type-case Class (find classes-not-flat name)
    [(classC field-names methods)
     (type-case Class (flatten-super name classes-not-flat i-classes)
       [(classC super-field-names super-methods)
        ...])]))
```

Flattening a Class

```
(define (flatten-class [name : Symbol]
                       [classes-not-flat : (Listof (Symbol * Class))]
                       [i-classes : (Listof (Symbol * ClassI))]) : Class
  (type-case Class (find classes-not-flat name)
    [(classC field-names methods)
     (type-case Class (flatten-super name classes-not-flat i-classes)
       [(classC super-field-names super-methods)
        (classC ....
                ....)])]))
```

Flattening a Class

```
(define (flatten-class [name : Symbol]
                       [classes-not-flat : (Listof (Symbol * Class))]
                       [i-classes : (Listof (Symbol * ClassI))]) : Class
  (type-case Class (find classes-not-flat name)
    [(classC field-names methods)
     (type-case Class (flatten-super name classes-not-flat i-classes)
       [(classC super-field-names super-methods)
        (classC (add-fields super-field-names
                            field-names)
                ....)])]))
```

Flattening a Class

```
(define (flatten-class [name : Symbol]
                       [classes-not-flat : (Listof (Symbol * Class))]
                       [i-classes : (Listof (Symbol * ClassI))]) : Class
  (type-case Class (find classes-not-flat name)
    [(classC field-names methods)
     (type-case Class (flatten-super name classes-not-flat i-classes)
       [(classC super-field-names super-methods)
        (classC (add-fields super-field-names
                            field-names)
                (add/replace-methods super-methods
                                     methods))])]))
```

Flattening a Class

```
(define (flatten-super [name : Symbol]
                       [classes-not-flat : (Listof (Symbol * Class))]
                       [i-classes : (Listof (Symbol * ClassI))]) : Class
  ...)
```

Flattening a Class

```
(define (flatten-super [name : Symbol]
                       [classes-not-flat : (Listof (Symbol * Class))]
                       [i-classes : (Listof (Symbol * ClassI))]) : Class
  ... (find i-classes name) ...)
```

Flattening a Class

```
(define (flatten-super [name : Symbol]
                       [classes-not-flat : (Listof (Symbol * Class))]
                       [i-classes : (Listof (Symbol * ClassI))]) : Class
  (type-case ClassI (find i-classes name)
    [(classI super-name field-names i-methods)
     ...]))
```

Flattening a Class

```
(define (flatten-super [name : Symbol]
                       [classes-not-flat : (Listof (Symbol * Class))]
                       [i-classes : (Listof (Symbol * ClassI))]) : Class
  (type-case ClassI (find i-classes name)
    [(classI super-name field-names i-methods)
     ... (flatten-class super-name
                        classes-not-flat
                        i-classes) ...]))
```

Flattening a Class

```
(define (flatten-super [name : Symbol]
                       [classes-not-flat : (Listof (Symbol * Class))]
                       [i-classes : (Listof (Symbol * ClassI))]) : Class
  (type-case ClassI (find i-classes name)
    [(classI super-name field-names i-methods)
     (if (equal? super-name 'Object)
         (classC empty empty)
         (flatten-class super-name
                        classes-not-flat
                        i-classes))]))
```

Interpreter

```
(define (interp-i [i-a : ExpI]
                  [i-classes : (Listof (Symbol * ClassI))]) : Value
  (local [(define a (exp-i->c i-a 'Object))
          (define classes-not-flat
            (map (lambda (i)
                   (values (fst i)
                           (class-i->c-not-flat (snd i))))
                 i-classes))
          (define classes
            (map (lambda (c)
                   (let ([name (fst c)])
                     (values
                      name
                      (flatten-class name classes-not-flat i-classes))))
                 classes-not-flat))]
    (interp a classes (objV 'Object empty) (numV 0))))
```