Part 1

Objects and Constructors

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

(define (closV n body c-env)
  (hash 'apply (lambda (arg-val) ....)
        '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 <Sym>              NEW
                {<Field>*}
                <Method>*}
<Field>   ::= <Sym>
<Method>  ::= {<Sym> <Expr>}
```

```
<Expr> ::= <Num>
        |  {+ <Expr> <Expr>}
        |  {* <Expr> <Expr>}
        |  arg
        |  this
        |  {new <Sym> <Expr>*}                    NEW
        |  {get <Expr> <Sym>}
        |  {send <Expr> <Sym> <Expr>}
        |  {ssend <Expr> <Sym> <Sym> <Expr>}      NEW
```

```
{class posn
  {x y}
  {mdist {+ {get this x} {get this y}}}
  {addDist {+ {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 <Sym>                NEW
                 {<Field>*}
                 <Method>*}
<Field>   ::=  <Sym>
<Method>  ::=  {<Sym> <Expr>}
```

```
<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  arg
          |  this
          |  {new <Sym> <Expr>*}                    NEW
          |  {get <Expr> <Sym>}
          |  {send <Expr> <Sym> <Expr>}
          |  {ssend <Expr> <Sym> <Sym> <Expr>}      NEW
```

```
{class posn ...
  {addDist {+ {send arg mdist 0}
              {send this 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 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:

posn3D
1
2
3

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 ClassC
  [classC (name : symbol)
          (field-names : (listof symbol))
          (methods : (listof Method))])
```

```
interp : (ExprC (listof ClassC) Value Value
                -> Value)
```

Examples

```
(test (interp (numC 10)
              empty
              (numV -1)
              (numV -1))
      (numV 10))
```

Examples

```
(define posn-class
  (classC 'posn
          (list 'x 'y)
          (list
           (methodC 'mdist
                    (plusC (getC (thisC) 'x) (getC (thisC) 'y)))
           (methodC 'addDist
                    (plusC (sendC (thisC) 'mdist (numC 0))
                           (sendC (argC) 'mdist (numC 0)))))))
(define posn3D-class
  (classC 'posn3D
          (list 'x 'y 'z)
          (list
           (methodC 'mdist
                    (plusC (getC (thisC) 'z)
                           (ssendC (thisC) 'posn 'mdist (argC))))
           (methodC 'addDist
                    (ssendC (thisC) 'posn 'addDist (argC))))))

(define (interp-posn a)
  (interp a (list posn-class posn3D-class) (numV -1) (numV -1)))
```

Examples

```
(test (interp-posn (newC 'posn
                         (list (numC 2)
                               (numC 7))))
      (objV 'posn
            (list (numV 2)
                  (numV 7))))
```

```
(define posn27
  (newC 'posn (list (numC 2) (numC 7))))
```

Examples

```
(define posn-class
  (classC 'posn
          (list 'x 'y)
          (list
           (methodC 'mdist
                    (plusC (getC (thisC) 'x) (getC (thisC) 'y)))
           (methodC 'addDist
                    (plusC (sendC (thisC) 'mdist (numC 0))
                           (sendC (argC) 'mdist (numC 0)))))))

(define posn27
  (newC 'posn (list (numC 2) (numC 7))))
```

```
(test (interp-posn (sendC posn27 'mdist (numC 0)))
      (numV 9))
```

Examples

```
(define posn-class
  (classC 'posn
          (list 'x 'y)
          (list
           (methodC 'mdist
                    (plusC (getC (thisC) 'x) (getC (thisC) 'y)))
           (methodC 'addDist
                    (plusC (sendC (thisC) 'mdist (numC 0))
                           (sendC (argC) 'mdist (numC 0)))))))
(define posn3D-class
  (classC 'posn3D
          (list 'x 'y 'z)
          (list
           (methodC 'mdist
                    (plusC (getC (thisC) 'z)
                           (ssendC (thisC) 'posn 'mdist (argC))))
           (methodC 'addDist
                    (ssendC (thisC) 'posn 'addDist (argC))))))
(define posn27
  (newC 'posn (list (numC 2) (numC 7))))
(define posn531
  (newC 'posn3D (list (numC 5) (numC 3) (numC 1))))
```

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

Part 4

Interpreter

```
(define interp : (ExprC (listof ClassC) Value Value -> Value)
  (lambda (a classes this-val arg-val)
    (local [(define (recur expr)
              (interp expr classes this-val arg-val))]
      (type-case ExprC a
        ...
        [numC (n) (numV n)]
        [plusC (l r) (num+ (recur l) (recur r))]
        [multC (l r) (num* (recur l) (recur r))]
        [thisC () this-val]
        [argC () arg-val]
        ...))))
```

Interpreter

```
(define interp : (ExprC (listof ClassC) Value Value -> Value)
  (lambda (a classes this-val arg-val)
    (local [(define (recur expr)
              (interp expr classes this-val arg-val))]
      (type-case ExprC a
        ...
        [newC (class-name field-exprs)
              (local [(define c (find-class class-name classes))
                      (define vals (map recur field-exprs))]
                (if (= (length vals) (length (class-field-names c)))
                    (objV class-name vals)
                    (error 'interp "wrong field count")))]
        ...))))
```

Helper

```
(define find-class : (symbol (listof ClassC) -> ClassC)
  (make-find class-name))
```

Interpreter

```
(define interp : (ExprC (listof ClassC) Value Value -> Value)
  (lambda (a classes this-val arg-val)
    (local [(define (recur expr)
              (interp expr classes this-val arg-val))]
      (type-case ExprC a
        ...
        [getC (obj-expr field-name)
              (type-case Value (recur obj-expr)
                [objV (class-name field-vals)
                      (type-case ClassC (find-class class-name
                                                     classes)
                        [classC (name fields methods)
                                (get-field field-name fields
                                           field-vals)])]
                [else (error 'interp "not an object")])]
        ...))))
```

Interpreter

```
(define interp : (ExprC (listof ClassC) Value Value -> Value)
  (lambda (a classes this-val arg-val)
    (local [(define (recur expr)
              (interp expr classes this-val arg-val))]
      (type-case ExprC a
        ...
        [sendC (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 ClassC (find-class class-name classes)
    [classC (name field-names methods)
            (type-case MethodC (find-method method-name
                                            methods)
              [methodC (name body-expr)
                       (interp body-expr
                               classes
                               obj
                               arg-val)])]))
```

Interpreter

```
(define interp : (ExprC (listof ClassC) Value Value -> Value)
  (lambda (a classes this-val arg-val)
    (local [(define (recur expr)
              (interp expr classes this-val arg-val))]
      (type-case ExprC a
        ...
        [ssendC (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 **ExprC**:

```
{class posn
  {x y}
  {mdist {+ {get this x} {get this y}}}
  {addDist {+ {send arg mdist 0}
              {send this 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 1 2 3} addDist {new posn 3 4}}
```

Programmer manually

- duplicates fields
- implements method inheritance

Subclasses

`ExprI` adds ***implementation inheritance***:

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

{class posn3D extends posn
  {z}
  {mdist {+ {get this z}
            {super mdist arg}}}}

{send {new posn3D 1 2 3} addDist {new posn 3 4}}
```

Class Language with Inheritance

```
<Class>   ::=  {class <Sym> extends <Sym>      NEW
                 {<Field>*}
                 <Method>*}
<Field>   ::=  <Sym>
<Method>  ::=  {<Sym> <Expr>}

<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  arg
          |  this
          |  {new <Sym> <Expr>*}
          |  {get <Expr> <Sym>}
          |  {send <Expr> <Sym> <Expr>}
          |  {super <Sym> <Expr>}           NEW
```

Compiling Inheritance

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

{class posn3D extends posn
  {z}
  {mdist {+ {get this z}
            {super mdist arg}}}}

{send {new posn3D 1 2 3} addDist {new posn 3 4}}
```

⇒

```
{class posn
  {x y}
  {mdist {+ {get this x} {get this y}}}
  {addDist {+ {send arg mdist 0}
              {send this 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 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 (name : symbol)
          (super-name : symbol)
          (field-names : (listof symbol))
          (methods : (listof MethodI))])

(define-type MethodI
  [methodI (name : symbol)
           (body-expr : ExprI)])
```

Expressions

```
(define-type ExprI
  [numI (n : number)]
  [plusI (lhs : ExprI)
         (rhs : ExprI)]
  [multI (lhs : ExprI)
         (rhs : ExprI)]
  [argI]
  [thisI]
  [newI (class-name : symbol)
        (args : (listof ExprI))]
  [getI (obj-expr : ExprI)
        (field-name : symbol)]
  [sendI (obj-expr : ExprI)
         (method-name : symbol)
         (arg-expr : ExprI)]
  [superI (method-name : symbol)
          (arg-expr : ExprI)])
```

Examples

```
(test (expr-i->c (numI 10))
      (numC 10))
```

Examples

```
(test (expr-i->c (thisI))
      (thisC))
```

Examples

```
(test (expr-i->c (superI 'mdist (numI 0)))
      (ssendC (thisC) ??? 'mdist (numC 0)))
```

Examples

```
expr-i->c : (ExprI symbol -> ExprC)

(test (expr-i->c (superI 'mdist (numI 0)) 'posn)
      (ssendC (thisC) 'posn 'mdist (numC 0)))
```

Compiling Expressions

```
(define expr-i->c : (ExprI symbol -> ExprC)
  (lambda (a super-name)
    (local [(define (recur expr)
                (expr-i->c expr super-name))]
      (type-case ExprI a
        [numI (n) (numC n)]
        [plusI (l r) (plusC (recur l) (recur r))]
        [multI (l r) (multC (recur l) (recur r))]
        ...
        [superI (method-name arg-expr)
                (ssendC (thisC)
                        super-name
                        method-name
                        (recur arg-expr))]))))
```

Compiling Methods

```
(define method-i->c : (MethodI symbol -> MethodC)
  (lambda (m super-name)
    (type-case MethodI m
      [methodI (name body-expr)
               (methodC name
                        (expr-i->c body-expr
                                   super-name))])))
```

Compiling Class Methods

```
(define class-i->c-not-flat : (ClassI -> ClassC)
  (lambda (c)
    (type-case ClassI c
      [classI (name super-name field-names methods)
              (classC name
                      field-names
                      (map (lambda (m)
                             (method-i->c m super-name))
                           methods))])))
```

Flattening a Class

```
(define flatten-class : (ClassC (listof ClassC) (listof ClassI)
                                 -> ClassC)
  (lambda (c classes i-classes)
    ...))
```

Flattening a Class

```
(define flatten-class : (ClassC (listof ClassC) (listof ClassI)
                                -> ClassC)
  (lambda (c classes i-classes)
    (type-case ClassC c
      [classC (name field-names methods)
              ...])))
```

Flattening a Class

```
(define flatten-class : (ClassC (listof ClassC) (listof ClassI)
                                -> ClassC)
  (lambda (c classes i-classes)
    (type-case ClassC c
      [classC (name field-names methods)
              ... (flatten-super name
                                 classes
                                 i-classes) ...])))
```

Flattening a Class

```
(define flatten-class : (ClassC (listof ClassC) (listof ClassI)
                                -> ClassC)
  (lambda (c classes i-classes)
    (type-case ClassC c
      [classC (name field-names methods)
              (type-case ClassC (flatten-super name
                                               classes
                                               i-classes)
                [classC (super-name super-field-names super-methods)
                        ...])])))
```

Flattening a Class

```
(define flatten-class : (ClassC (listof ClassC) (listof ClassI)
                                -> ClassC)
  (lambda (c classes i-classes)
    (type-case ClassC c
      [classC (name field-names methods)
              (type-case ClassC (flatten-super name
                                              classes
                                              i-classes)
                [classC (super-name super-field-names super-methods)
                        (classC name
                                ....
                                ....)])]))))
```

Flattening a Class

```
(define flatten-class : (ClassC (listof ClassC) (listof ClassI)
                                -> ClassC)
  (lambda (c classes i-classes)
    (type-case ClassC c
      [classC (name field-names methods)
              (type-case ClassC (flatten-super name
                                               classes
                                               i-classes)
                [classC (super-name super-field-names super-methods)
                        (classC name
                                (add-fields super-field-names
                                            field-names)
                                ....)])])))
```

Flattening a Class

```
(define flatten-class : (ClassC (listof ClassC) (listof ClassI)
                                 -> ClassC)
  (lambda (c classes i-classes)
    (type-case ClassC c
      [classC (name field-names methods)
              (type-case ClassC (flatten-super name
                                               classes
                                               i-classes)
                [classC (super-name super-field-names super-methods)
                        (classC name
                                (add-fields super-field-names
                                            field-names)
                                (add/replace-methods super-methods
                                                     methods))])])))
```

Flattening a Class

```
(define flatten-super : (symbol (listof ClassC) (listof ClassI)
                                 -> ClassC)
  (lambda (name classes i-classes)
    ...))
```

Flattening a Class

```
(define flatten-super : (symbol (listof ClassC) (listof ClassI)
                                -> ClassC)
  (lambda (name classes i-classes)
    ... (find-i-class name i-classes) ...))
```

Flattening a Class

```
(define flatten-super : (symbol (listof ClassC) (listof ClassI)
                                -> ClassC)
  (lambda (name classes i-classes)
    (type-case ClassI (find-i-class name i-classes)
      [classI (name super-name field-names i-methods)
              ...])))
```

Flattening a Class

```
(define flatten-super : (symbol (listof ClassC) (listof ClassI)
                                -> ClassC)
  (lambda (name classes i-classes)
    (type-case ClassI (find-i-class name i-classes)
      [classI (name super-name field-names i-methods)
              ... (find-class super-name classes) ...])))
```

Flattening a Class

```
(define flatten-super : (symbol (listof ClassC) (listof ClassI)
                                -> ClassC)
  (lambda (name classes i-classes)
    (type-case ClassI (find-i-class name i-classes)
      [classI (name super-name field-names i-methods)
              (if (equal? super-name 'object)
                  (classC 'object empty empty)
                  ... (find-class super-name classes) ...)])))
```

Flattening a Class

```
(define flatten-super : (symbol (listof ClassC) (listof ClassI)
                                -> ClassC)
  (lambda (name classes i-classes)
    (type-case ClassI (find-i-class name i-classes)
      [classI (name super-name field-names i-methods)
              (if (equal? super-name 'object)
                  (classC 'object empty empty)
                  (flatten-class (find-class super-name classes)
                                 classes
                                 i-classes))])))
```

Interpreter

```
(define interp-i : (ExprI (listof ClassI) -> Value)
  (lambda (i-a i-classes)
    (local [(define a (expr-i->c i-a 'object))
            (define classes-not-flat
              (map class-i->c-not-flat i-classes))
            (define classes
              (map (lambda (c)
                     (flatten-class c classes-not-flat i-classes))
                   classes-not-flat))]
      (interp a classes (numV -1) (numV -1)))))
```