- **Design: Functional to Object-Oriented**
- **Semantics of Local Definitions**

Compound Data in Java

Beginner Scheme:

```
; A snake is
;  (make-snake sym num sym)
(define-struct snake (name weight food))
```

Beginner Java:

```
class Snake {
  String name;
  double weight;
  String food;
  Snake(String name, double weight, String food) {
    this.name = name;
    this.weight = weight;
    this.food = food;
  }
}
```

Instances of Compound Data Types

Beginner Scheme:

```
(make-snake 'Slinky 12 'rats)
(make-snake 'Slimey 5 'grass)
```

Beginner Java:

```
new Snake("Slinky", 12, "rats")
new Snake("Slimey", 5, "grass")
```

Armadillos

```
class Dillo {
  double weight;
  boolean alive;
  Dillo(double weight, boolean alive) {
    this.weight = weight;
    this.alive = alive;
  }
}

new Dillo(2, true)
new Dillo(3, false)
```

Posns

```
class Posn {
  int x;
  int y;
  Posn(int x, int y) {
   this.x = x;
   this.y = y;
  }
}

new Posn(0, 0)
new Posn(1, -2)
```

Ants

```
class Ant {
  double weight;
  Posn loc;
  Ant(double weight, Posn loc) {
    this.weight = weight;
    this.loc = loc;
  }
}

new Ant(0.0001, new Posn(0, 0))
new Ant(0.0002, new Posn(1, -2))
```

Data with Variants

Beginner Scheme:

```
; An animal is either
;    - snake
;    - dillo
;    - ant
```

```
abstract class Animal {
}

class Snake extends Animal {
 ... as before ...
}
class Dillo extends Animal {
 ... as before ...
}
class Ant extends Animal {
 ... as before ...
}
```

Variants in Java

- A data definition with variants must refer only to other data definitions (which are not built in)

```
; A grade is either        ⇒     ; A grade is either
;  - false                       ;  - no-grade
;  - num                         ;  - num-grade

                                 ; A no-grade is
                                 ;  (make-no-grade)
                                 (define-struct no-grade ())

                                 ; A num-grade is
                                 ;  (make-num-grade num)
                                 (define-struct num-grade (n))
```

- A data definition can be a variant in at most one other data definition

From Scheme to Java

So far, we’ve translated data definitions:

```
; A snake is
;  (make-snake sym num sym)
(define-struct snake (name weight food))
```

⇒

```
class Snake {
  String name;
  double weight;
  String food;
  Snake(String name, double weight, String food) {
    this.name = name;
    this.weight = weight;
    this.food = food;
  }
}
```

Functions in Scheme

```
; A snake is
;  (make-snake sym num sym)
(define-struct snake (name weight food))

; snake-lighter? : snake num -> bool
;  Determines whether s is < n lbs
(define (snake-lighter? s n)
  (< (snake-weight s) n))

(snake-lighter? (make-snake 'Slinky 10 'rats) 10)
"should be" false
(snake-lighter? (make-snake 'Slimey 5 'grass) 10)
"should be" false
```

Functions in Java

```
class Snake {
  String name;
  double weight;
  String food;
  Snake(String name, double weight, String food) {
    this.name = name;
    this.weight = weight;
    this.food = food;
  }

  // Determines whether it's < n lbs
  boolean isLighter(double n) {
    return this.weight < n;
  }
}

new Snake("Slinky", 10, "rats").isLighter(10)
"should be" false
```

Functions in Java

```
class Snake {
  String name;
  double weight;
  String food;
  Snake(String na                String food) {
    this.name = r
    this.weight =
    this.food = f
  }

  // Determines
  boolean isLighter(double n) {
    return this.weight < n;
  }
}

new Snake("Slinky", 10, "rats").isLighter(10)
"should be" false
```

A method in **Snake** has an implicit **Snake this** argument

Method Calls in Java

Original tests:

Scheme:

```
(snake-lighter? (make-snake 'Slinky 10 'rats) 10)
"should be" false
```

Java:

```
new Snake("Slinky", 10, "rats").isLighter(10)
"should be" false
```

Templates

In Scheme:

```
; A snake is
;  (make-snake sym num sym)
(define-struct snake (name weight food))

; func-for-snake : snake -> ...
(define (func-for-snake s)
  ... (snake-name s)
  ... (snake-weight s)
  ... (snake-food s) ...)
```

Templates

Same idea works for Java:

```
class Snake {
  String name;
  double weight;
  String food;
  Snake(String name, double weight, String food) {
    this.name = name;
    this.weight = weight;
    this.food = food;
  }

  ... methodForSnake(...) {
    ... this.name
    ... this.weight
    ... this.food ...
  }
}
```

Functions with Variants

```
; An animal is either
;  - snake
;  - dillo
;  - ant

; animal-is-lighter? : animal num -> bool
(define (animal-is-lighter? a n)
  (cond
    [(snake? a) (snake-is-lighter? s n)]
    [(dillo? a) (dillo-is-ligheter? s n)]
    [(ant? a) (ant-is-lighter? s n)]))

; snake-is-lighter? : snake num -> bool
(define (snake-is-lighter? s n) ...)

; dillo-is-lighter? : dillo num -> bool
(define (dillo-is-lighter? d n) ...)

; ant-is-lighter? : ant num -> bool
(define (ant-is-lighter? a n) ...)
```

Methods with Variants

```
abstract class Animal {
  abstract boolean isLighter(double n);
}

class Snake extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Dillo extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Ant extends Animal {
  ...
  boolean isLighter(double n) { ... }
}
```

Translating Functions to Methods

```
; An animal is either
;  - snake
;  - dillo
;  - ant

; animal-is-lighter? : animal num -> bool
(define (animal-is-lighter? a n)
  (cond
    [(snake? a) (snake-is-lighter? s n)]
    [(dillo? a) (dillo-is-ligheter? s n)]
    [(ant? a) (ant-is-lighter? s n)]))

; snake-is-lighter? : snake num -> bool
(define (snake-is-lighter? s n) ...)

; dillo-is-lighter? : dillo num -> bool
(define (dillo-is-lighter? d n) ...)

; ant-is-lighter? : ant num -> bool
(define (ant-is-lighter? a n) ...)
```

```
abstract class Animal {
  abstract boolean isLighter(double n);
}

class Snake extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Dillo extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Ant extends Animal {
  ...
  boolean isLighter(double n) { ... }
}
```

Translating Functions to Methods

Data definition turns into class declarations

```
; An animal is either
;  - snake
;  - dillo
;  - ant

; animal-is-lighter? : animal num -> bool
(define (animal-is-lighter? a n)
  (cond
    [(snake? a) (snake-is-lighter? s n)]
    [(dillo? a) (dillo-is-ligheter? s n)]
    [(ant? a) (ant-is-lighter? s n)]))

; snake-is-lighter? : snake num -> bool
(define (snake-is-lighter? s n) ...)

; dillo-is-lighter? : dillo num -> bool
(define (dillo-is-lighter? d n) ...)

; ant-is-lighter? : ant num -> bool
(define (ant-is-lighter? a n) ...)
```

```
abstract class Animal {
  abstract boolean isLighter(double n);
}

class Snake extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Dillo extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Ant extends Animal {
  ...
  boolean isLighter(double n) { ... }
}
```

Translating Functions to Methods

Variant functions turn into variant methods — all with the same contract after the implicit argument

```
; An animal is either
;  - snake
;  - dillo
;  - ant

; animal-is-lighter? : animal num -> bool
(define (animal-is-lighter? a n)
  (cond
    [(snake? a) (snake-is-lighter? s n)]
    [(dillo? a) (dillo-is-ligheter? s n)]
    [(ant? a) (ant-is-lighter? s n)]))

; snake-is-lighter? : snake num -> bool
(define (snake-is-lighter? s n) ...)

; dillo-is-lighter? : dillo num -> bool
(define (dillo-is-lighter? d n) ...)

; ant-is-lighter? : ant num -> bool
(define (ant-is-lighter? a n) ...)
```

```
abstract class Animal {
  abstract boolean isLighter(double n);
}

class Snake extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Dillo extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Ant extends Animal {
  ...
  boolean isLighter(double n) { ... }
}
```

Translating Functions to Methods

```
; An animal is either
;  - snake
;  - dillo
;  - ant

; animal-is-lighter? : animal num -> bool
(define (animal-is-lighter? a n)
  (cond
    [(snake? a) (snake-is-lighter? s n)]
    [(dillo? a) (dillo-is-ligheter? s n)]
    [(ant? a) (ant-is-lighter? s n)]))

; snake-is-lighter? : snake num -> bool
(define (snake-is-lighter? s n) ...)

; dillo-is-lighter? : dillo num -> bool
(define (dillo-is-lighter? d n) ...)

; ant-is-lighter? : ant num -> bool
(define (ant-is-lighter? a n) ...)
```

Function with variant-based **cond** turns into just an **abstract** method declaration

```
abstract class Animal {
  abstract boolean isLighter(double n);
}

class Snake extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Dillo extends Animal {
  ...
  boolean isLighter(double n) { ... }
}

class Ant extends Animal {
  ...
  boolean isLighter(double n) { ... }
}
```

Lists of Things

```
abstract class ListOfThing {
  abstract int length();
}

class EmptyListOfThing extends ListOfThing {
   EmptyListOfThing() { }
   int length() { return 0; }
}

class ConsListOfThing extends ListOfThing {
   Thing first;
   ListOfThing rest;
   ConsListOfThing(Thing first, ListOfThing rest) {
     this.first = first;
     this.rest = rest;
   }
   int length() { return 1 + this.rest.length(); }
}
```

Trees of Things

```
abstract class TreeOfThing {
  abstract int count();
}

class EmptyTreeOfThing extends TreeOfThing {
   EmptyTreeOfThing() { }
   int count() { return 0; }
}

class ConsTreeOfThing extends TreeOfThing {
   Thing v;
   TreeOfThing left;
   TreeOfThing right;
   ConsTreeOfThing(Thing v, TreeOfThing left, TreeOfThing right) {
     this.v = v;
     this.left = left;
     this.right = right;
   }
   int count() { return 1 + this.left.count()
                           + this.right.count(); }
}
```

Implementing Methods Directly

Some Scheme methods on **animal** can be implemented with other **animal** functions:

```
; animal-light? : animal -> bool
; Determines whether a is less than 10 lbs
(define (animal-light? a)
  (animal-lighter? a 10))
```

In Java, this corresponds to a non-**abstract** method in an **abstract** class:

```
abstract class Animal {
   ...
   boolean isLight() {
     return this.isLighter(10);
   }
}
```

- Design: Functional to Object-Oriented
- **Semantics of Local Definitions**

Random Numbers

The `random` operator returns a different result for different calls with the same input:

```
> (random 3)
0
> (random 3)
2
> (random 3)
1
> (random 3)
2
```

Random Symbols

Suppose we need a `random-symbol` function

```
> (random-symbol 'huey 'dewey 'louie)
'dewey
> (random-symbol 'huey 'dewey 'louie)
'huey
> (random-symbol 'huey 'dewey 'louie)
'dewey
> (random-symbol 'huey 'dewey 'louie)
'louie
```

Can we implement it with `random`?

Random Symbols

```
; random-symbol : sym sym sym -> sym
(define (random-symbol a b c)
  (cond
    [(= (random 3) 0) a]
    [(= (random 3) 1) b]
    [(= (random 3) 2) c]))
```

This doesn't work, because `random` produces a different result each time

Saving a Random Number

On the other hand...

```
(define n (random 3))
(list n n n)
```

produces `(list 0 0 0)`, `(list 1 1 1)`, or `(list 2 2 2)`

Constant definitions name constants, so `(random 3)` must be evaluted when defining `n`

Try it in the stepper

A Random Constant

Does this work?

```
(define n (random 3))

; random-symbol : sym sym sym -> sym
(define (random-symbol a b c)
  (cond
    [(= n 0) a]
    [(= n 1) b]
    [(= n 2) c]))
```

Not quite, because it always picks the same symbol

We want `(define n (random 3))` that is local to `random-symbol`'s body

Local Definitions

This works, in the ***Intermediate*** language

```
; random-symbol : sym sym sym -> sym
(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond
      [(= n 0) a]
      [(= n 1) b]
      [(= n 2) c])))
```

- The **`local`** form has definitions and a body
- Local definitions are only visible in the body
- Local definitions are evaluated only when the **`local`** is evaluated
- The result of **`local`** is the result of its body

Evaluation with Local

```
(define (random-symbol a b c)
  (local [(define n (random 3))]
    (cond
      [(= n 0) a]
      [(= n 1) b]
      [(= n 2) c])))
(random-symbol 'huey 'dewey 'louie)
(random-symbol 'huey 'dewey 'louie)
```

→

```
(define (random-symbol ...) ...)
(local [(define n (random 3))]
  (cond
    [(= n 0) 'huey]
    [(= n 1) 'dewey]
    [(= n 2) 'louie]))
(random-symbol 'huey 'dewey 'louie)
```

Evaluation with Local

```
(define (random-symbol ...) ...)
(local [(define n (random 3))]
  (cond
    [(= n 0) 'huey]
    [(= n 1) 'dewey]
    [(= n 2) 'louie]))
(random-symbol 'huey 'dewey 'louie)
```

→

```
(define (random-symbol ...) ...)
(define n17 (random 3))
(cond
  [(= n17 0) 'huey]
  [(= n17 1) 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

Evaluation of **local** lifts and renames the definition

Evaluation with Local

```
(define (random-symbol ...) ...)
(define n17 (random 3))
(cond
  [(= n17 0) 'huey]
  [(= n17 1) 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

→

```
(define (random-symbol ...) ...)
(define n17 1)
(cond
  [(= n17 0) 'huey]
  [(= n17 1) 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

Evaluation with Local

```
(define (random-symbol ...) ...)
(define n17 1)
(cond
  [(= n17 0) 'huey]
  [(= n17 1) 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

→

```
(define (random-symbol ...) ...)
(define n17 1)
(cond
  [(= 1 0) 'huey]
  [(= n17 1) 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

Evaluation of a constant name finds the value

Evaluation with Local

```
(define (random-symbol ...) ...)
(define n17 1)
(cond
  [(= 1 0) 'huey]
  [(= n17 1) 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

→

```
(define (random-symbol ...) ...)
(define n17 1)
(cond
  [false 'huey]
  [(= n17 1) 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

Evaluation with Local

```
(define (random-symbol ...) ...)
(define n17 1)
(cond
  [false 'huey]
  [(= n17 1) 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

→

```
(define (random-symbol ...) ...)
(define n17 1)
(cond
  [(= n17 1) 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

Evaluation with Local

```
(define (random-symbol ...) ...)
(define n17 1)
(cond
  [(= n17 1) 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

→

```
(define (random-symbol ...) ...)
(define n17 1)
(cond
  [(= 1 1) 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

Evaluation with Local

```
(define (random-symbol ...) ...)
(define n17 1)
(cond
  [(= 1 1) 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

→

```
(define (random-symbol ...) ...)
(define n17 1)
(cond
  [true 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

Evaluation with Local

```
(define (random-symbol ...) ...)
(define n17 1)
(cond
  [true 'dewey]
  [(= n17 2) 'louie])
(random-symbol 'huey 'dewey 'louie)
```

→

```
(define (random-symbol ...) ...)
(define n17 1)
'dewey
(random-symbol 'huey 'dewey 'louie)
```

Evaluation with Local

```
(define (random-symbol ...) ...)
(define n17 1)
'dewey
(random-symbol 'huey 'dewey 'louie)
```

→

```
(define (random-symbol ...) ...)
(define n17 1)
'dewey
(local [(define n (random 3))]
  (cond
    [(= n 0) 'huey]
    [(= n 1) 'dewey]
    [(= n 2) 'louie]))
```

Evaluation with Local

```
(define (random-symbol ...) ...)
(define n17 1)
'dewey
(local [(define n (random 3))]
  (cond
    [(= n 0) 'huey]
    [(= n 1) 'dewey]
    [(= n 2) 'louie]))
```

→

```
(define (random-symbol ...) ...)
(define n17 1)
'dewey
(define n45 (random 3))
(cond
  [(= n45 0) 'huey]
  [(= n45 1) 'dewey]
  [(= n45 2) 'louie])
```

Evaluation of `local` picks a new name each time

Evaluation with Local

```
(define (random-symbol ...) ...)
(define n17 1)
'dewey
(define n45 (random 3))
(cond
  [(= n45 0) 'huey]
  [(= n45 1) 'dewey]
  [(= n45 2) 'louie])
```

→

```
(define (random-symbol ...) ...)
(define n17 1)
'dewey
(define n45 0)
(cond
  [(= n45 0) 'huey]
  [(= n45 1) 'dewey]
  [(= n45 2) 'louie])
```

Another Example

```
; kind-of-blue? : image -> bool
(define (kind-of-blue? i)
  (and
   (> (total-blue (image->color-list i))
      (total-red (image->color-list i)))
   (> (total-blue (image->color-list i))
      (total-green (image->color-list i)))))
```

Easier to read, converts image only once:

```
(define (kind-of-blue? i)
  (local [(define colors
            (image->color-list i))]
    (and (> (total-blue colors)
            (total-red colors))
         (> (total-blue colors)
            (total-green colors)))))
```

Another Example

```
(define (eat-apples l)
  (cond
    [(empty? l) empty]
    [(cons? l)
     (cond
       [(symbol=? (first l) 'apple)
        (eat-apples (rest l))]
       [else
        (cons (first l) (eat-apples (rest l)))])]))
```

Better:

```
(define (eat-apples l)
  (cond
    [(empty? l) empty]
    [(cons? l)
     (local [(define ate-rest (eat-apples (rest l)))]
       (cond
         [(symbol=? (first l) 'apple) ate-rest]
         [else (cons (first l) ate-rest)]))]))
```

Another Use for Local

`local` can define functions as well as constants

This is useful for making a function private

```
(define (random-symbol a b c)
  (local [(define (real-random-symbol a b c)
            (local [(define n (random 3))]
              (cond
                [(= n 0) a]
                [(= n 1) b]
                [(= n 2) c])))]
    (cond
      [(and (symbol? a) (symbol? b) (symbol? c))
       (real-random-symbol a b c)]
      [else (error 'random-symbol "not a symbol")])))
```

Use Check Syntax and mouse over variables

Lexical Scope

```
(define (random-symbol a b c)
  (local [(define (real-random-symbol a b c)
            (local [(define n (random 3))]
              (cond
                [(= n 0) a]
                [(= n 1) b]
                [(= n 2) c])))]
    (cond
      [(and (symbol? a) (symbol? b) (symbol? c))
       (real-random-symbol a b c)]
      [else (error 'random-symbol "not a symbol")])))
```

(In the inner definition, the parameters *a b c* and the *a*, *b*, *c* results of the inner cond are printed in italic.)

Italic ***a*** could be changed to **z** without affecting non-italic **a**, no matter how the code runs

In other words, bindings are static; this is ***lexical scope***