Part I

Functional Programming

Functional programming is avoiding state

```
(define (area [s : Shape]) : number
  ....)

(define r (rectangle 10 15))
(test (area r)
      150)
(test (area r)
      150)
...
```

The alternative: ***imperative programming***

Functional Programming

Functional programming often means using functions as values

```
(map area (list (rectangle 10 5)
                (square 6)
                (equilateral-triangle 7)))
```

The alternative: ***first-order programming***?

Functional Programming

Functional programming often means
datatype-oriented programming

```
(define-type Shape
  [rectangle (w : number) (h : number)]
  [square (side : number)]
  [equilateral-triangle (side : number)])

(define (area [s : Shape]) : number
  (type-case Shape s
    [rectangle (w h) ...]
    [square (s) ...]
    [equilateral-triangle (s) ...]))
```

Functional Programming

Functional programming often means
datatype-oriented programming

```
(define (sum-of-areas [l : (listof Shape)])
  (cond
   [(empty? l) 0]
   [(cons? l) (+ (area (first l))
                 (sum-of-areas (rest l)))]))
```

The alternative: ***object-oriented programming***

Datatype-Oriented versus Object-Oriented

Datatype-oriented: call an operation with a variant

```
(define (area [s : Shape]) : number
  (type-case Shape s
    [rectangle (w h) ...]
    [square (s) ...]
    [equilateral-triangle (s) ...]))
```

```
(define (perimeter [s : Shape]) : number
  (type-case Shape s
    [rectangle (w h) ...]
    [square (s) ...]
    [equilateral-triangle (s) ...]))
```

Datatype-Oriented versus Object-Oriented

Datatype-oriented: call an operation with a variant

```
(define (area [s : Shape]) : number
  (type-case Shape s
    [rectangle (w h) ...]
    [square (s) ...]
    [equilateral-triangle (s) ...]))
```

```
(define (perimeter [s : Shape]) : number
  (type-case Shape s
    [rectangle (w h) ...]
    [square (s) ...]
    [equilateral-triangle (s) ...]))
```

Object-oriented: call a variant with an operation

```
class Rectangle extends Shape { ...
  int area() { ... }
  int perimeter() { ... }
}
class Square extends Shape { ...
  int area() { ... }
  int perimeter() { ... }
}
class EquilateralTriangle extends Shape { ...
  int area() { ... }
  int perimeter() { ... }
}
```

Datatype-Oriented versus Object-Oriented

Datatype-oriented: call an operation with a variant

```
(define (area [s : Shape]) : number
  (type-case Shape s
    [rectangle (w h) ...]
    [square (s) ...]
    [equilateral-triangle (s) ...]))
```

```
(define (perimeter [s : Shape]) : number
```

- new **operation** ⇒ new function
- new **variant** ⇒ change functions

Object-oriented: call a variant with an operation

```
class Rectangle extends Shape { ...
  int area() { ... }
  int perimeter() { ... }
}
class Square extends Shape { ...
  int area() { ... }
  int perimeter() { ... }
}
class EquilateralTriangle extends Shape { ...
  int area() { ... }
  int perimeter() { ... }
}
```

Datatype-Oriented versus Object-Oriented

Datatype-oriented: call an operation with a variant

```
(define (area [s : Shape]) : number
  (type-case Shape s
    [rectangle (w h) ...]
    [square (s) ...]
    [equilateral-triangle (s) ...]))
```

```
(define (perimeter [s : Shape]) : number
```

- new **operation** ⇒ new function
- new **variant** ⇒ change functions

Object-oriented: call a variant with an operation

```
class Rectangle extends Shape { ...
  int area() { ... }
  int perimeter() { ... }
}
class Square extends Shape { ...
  int area() { ... }
  int perimeter() { ... }
}
cl
```

```
}
```

- new **operation** ⇒ change objects
- new **variant** ⇒ new objects

Part 2

Datatype-Oriented versus Object-Oriented

Functional programming can be
datatype-oriented or object-oriented

We can use functions to represent objects...

Representing Objects with Functions

```
(define-type-alias Shape (-> number))


(define (rectangle w h) : Shape
  (lambda ()
    (* w h)))

(define (square s) : Shape
  (lambda ()
    (* s s)))

(define r (rectangle 10 15))
(r) ⇒ 15


(define c (let ([r 10])
            (lambda () (* pi (* r r)))))
(c) ⇒ 314.179
```

Part 3

Objects and Multiple Operations

Simple function implements an object with a single operation:

```
(define-type-alias Shape (-> number))

(define r (rectangle 10 15))
(r)
```

For multiple operations, could pass a symbol to select:

```
(define-type-alias Shape (symbol -> number))

(define r (rectangle 10 15))
(r 'area)
(r 'perimeter)
```

Representing Objects with Functions

```
(define-type-alias Shape (symbol -> number))

(define (rectangle w h) : Shape
  (lambda (op)
    (cond
     [(equal? op 'area) (* w h)]
     [(equal? op 'perimeter) (* 2 (+ w h))])))

(define (square s) : Shape
  (lambda (op)
    (cond
     [(equal? op 'area) (* s s)]
     [(equal? op 'perimeter) (* 4 s)])))

(define r (rectangle 10 15))
(r 'area) ⇒ 150
(r 'perimeter) ⇒ 50
```

Representing Objects with Functions

```
#lang plai ...

; A Shape is
;   (hash 'area (-> number)
;         'bigger-than? (number -> boolean))
;         ....)

(define (rectangle w h)
  (hash 'area (lambda () (* w h))
        'bigger-than? (lambda (n) (> (* w h) n))))

(define (square s)
  (hash 'area (lambda () (* s s))
        'bigger-than? (lambda (n) (> (* s s) n))))

(define r (rectangle 10 15))
((hash-ref r 'area)) ⇒ 150
((hash-ref r 'bigger-than?) 100) ⇒ #t
```

Part 4

Objects without Functions

In some contexts:

datatype-oriented vs. ***object-oriented***

... choice of organization with implications for extensibility

In other contexts:

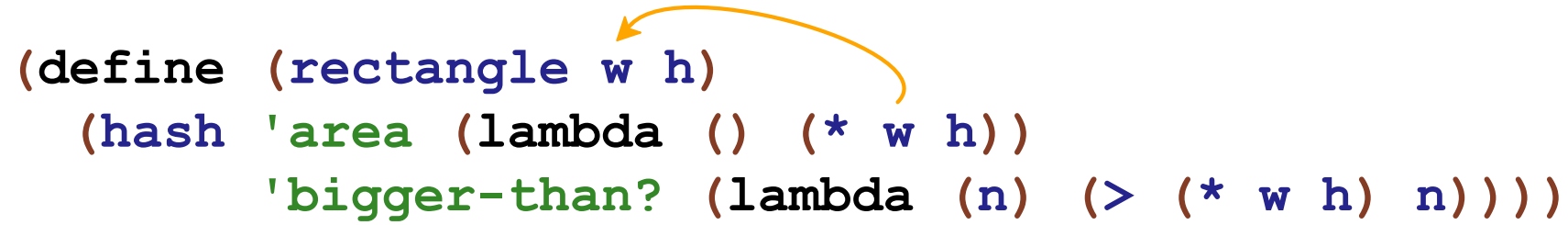
functional vs. ***object-oriented***

... choice of language primitives

Representing Objects with Higher-Order Functions

```
(define (rectangle w h)
  (hash 'area (lambda () (* w h))
        'bigger-than? (lambda (n) (> (* w h) n))))
```

Representing Objects with Higher-Order Functions



```
(define (rectangle w h)
  (hash 'area (lambda () (* w h))
        'bigger-than? (lambda (n) (> (* w h) n))))
```

Relies on nested functions

... implemented as closures

Representing Objects with First-Order Functions

```
(define-syntax-rule (rectangle init-w init-h)
  (list (hash 'w init-w  'h init-h)
        (hash 'area (lambda (this)
                      (* (get this w)
                         (get this h)))
              'bigger-than? (lambda (this n)
                              (> (send this area)
                                 n)))))
```

Could be written as

```
(define (r-area this)
  (* (get this w) (get this h)))

(define (r-bigger-than? this n)
  (> (send this area) n))

(define-syntax-rule (rectangle init-w init-h)
  (list (hash 'w init-w  'h init-h)
        (hash 'area r-area
              'bigger-than? r-bigger-than?)))
```

Representing Objects with First-Order Functions

```
; A shape is
;  (list (hash ....)
;        (hash 'area (shape -> number)
;              'bigger-than? (shape number -> number)))

(define-syntax-rule (get obj-expr field-id)
  (hash-ref (first obj-expr) 'field-id))

(define-syntax-rule (rectangle init-w init-h)
  (list (hash 'w init-w 'h init-h)
        (hash 'area (lambda (this)
                      (* (get this w)
                         (get this h)))
              'bigger-than? (lambda (this arg)
                              (> (send this area)
                                 arg)))))

(define-syntax-rule (send obj-expr method-id arg ...)
  (let ([o obj-expr])
    ((hash-ref (second o) 'method-id) o arg ...)))

(define r (rectangle 10 15))
(send r bigger-than? 200) ⇒ #f
```

Representing Objects with First-Order Functions

Simplifiction: assume that all methods take one argument, and the argument is named **`arg`**

```
(define-syntax-rule (object ([field-id field-expr] ...)
                            [method-id body-expr] ...)
  ....)

(define-syntax-rule (rectangle init-w init-h)
  (object ([w init-w]
           [h init-h])
          [area (* (get this w)
                   (get this h))]
          [bigger-than? (> (send this area 0)
                           arg)]))

(define r (rectangle 10 15))
(send r area 0) ⇒ 150
(send r bigger-than? 100) ⇒ #t
```

Part 5

Functions/Datatypes versus Objects

So far:

object-oriented interpreter of a functional language

Now:

functional interpreter of an object-oriented language

Objects Instead of Functions

```
<Expr>    ::=  …
            |  {object {<Field>*} <Method>*}
            |  {get <Expr> <Sym>}
            |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Expr>}
```

Objects Instead of Functions

```
<Expr>    ::=  ...
            |  {object {<Field>*} <Method>*}
            |  {get <Expr> <Sym>}
            |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Ex
```

evaluated when object is created

Objects Instead of Functions

```
<Expr>    ::=  …
             |  {object {<Field>*} <Method>*}
             |  {get <Expr> <Sym>}
             |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Expr>}
```

delayed until method is called

Objects Instead of Functions

```
<Expr>    ::=  …
             |  {object {<Field>*} <Method>*}
             |  {get <Expr> <Sym>}
             |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Expr>}
```

method always takes one argument

Objects Instead of Functions

```
<Expr>    ::=  …
             |  {object {<Field>*} <Method>*}
             |  {get <Expr> <Sym>}
             |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Expr>}
```

use `arg` to access method argument

Objects Instead of Functions

```
<Expr>    ::=  …
            |  {object {<Field>*} <Method>*}
            |  {get <Expr> <Sym>}
            |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Expr>}
```

use `this` to access enclosing object

Objects Instead of Functions

```
<Expr>    ::=  ...
             |  {object {<Field>*} <Method>*}
             |  {get <Expr> <Sym>}
             |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Ex
```

`arg` and `this` refer to outside object, if any

Objects Instead of Functions

```
<Expr>    ::=  ...
            |  {object {<Field>*} <Method>*}
            |  {get <Expr> <Sym>}
            |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Expr>}
```

```
{object
 {{x 1} {y 2}}}
```

Objects Instead of Functions

```
<Expr>    ::=  …
            |  {object {<Field>*} <Method>*}
            |  {get <Expr> <Sym>}
            |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Expr>}
```

```
{object
 {{x 1} {y {+ 1 1}}}}
```

Objects Instead of Functions

```
<Expr>    ::=  ...
            |  {object {<Field>*} <Method>*}
            |  {get <Expr> <Sym>}
            |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Expr>}
```

```
{get {object
       {{x 1} {y 2}}}
     x}
⇒
1
```

Objects Instead of Functions

```
<Expr>    ::=  …
            |  {object {<Field>*} <Method>*}
            |  {get <Expr> <Sym>}
            |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Expr>}
```

```
{object
 {}
 {inc {+ arg 1}}}
```

Objects Instead of Functions

```
<Expr>    ::=  …
            |  {object {<Field>*} <Method>*}
            |  {get <Expr> <Sym>}
            |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Expr>}
```

```
{send {object
        {}
        {inc {+ arg 1}}}
      inc
      2}
```

⇒

```
3
```

Objects Instead of Functions

```
<Expr>    ::=  …
            |  {object {<Field>*} <Method>*}
            |  {get <Expr> <Sym>}
            |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Expr>}
```

```
{object
 {}
 {inc {+ arg 1}}
 {dec {+ arg -1}}}
```

Objects Instead of Functions

```
<Expr>    ::=  …
             |  {object {<Field>*} <Method>*}
             |  {get <Expr> <Sym>}
             |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Expr>}
```

```
{object
 {{x 1} {y 2}}
 {mdist {+ {get this x}
           {get this y}}}}
```

Objects Instead of Functions

```
<Expr>    ::=  …
            |  {object {<Field>*} <Method>*}
            |  {get <Expr> <Sym>}
            |  {send <Expr> <Sym> <Expr>}
<Field>   ::=  {<Sym> <Expr>}
<Method>  ::=  {<Sym> <Expr>}
```

```
{send {object
        {{x 1} {y 2}}
        {mdist {+ {get this x}
                  {get this y}}}}
      mdist
      0}
⇒
3
```

Part 6

Objects Instead of Functions

Objects can encode functions:

```
{{lambda {x} x}
 5}
```

≈

```
{send {object {} {call arg}}
      call
      5}
```

Objects Instead of Functions

Objects can encode functions:

```
{{{lambda {x} {lambda {y} {+ x y}}}
  5}
 6}

≈

{send {send {object
              {}
              {call {object
                      {{x arg}}
                      {call {+ arg
                               {get this x}}}}}}
            call
            5}
      call
      6}
```

Part 7

Object Language

```
<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  arg
          |  this
          |  {object {<Field>*} <Method>*}
          |  {get <Expr> <Sym>}
          |  {send <Expr> <Sym> <Expr>}
```

```
{send {object
       {{x 3} {y 4}}
       {mdist {+ {get this x}
                 {get this y}}}}
      call
      mdist
      0}
```

Analogous Java code

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

Object Language

```
<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  arg
          |  this
          |  {object {<Field>*} <Method>*}
          |  {get <Expr> <Sym>}
          |  {send <Expr> <Sym> <Expr>}
```

```
{send {object
       {{x 1} {y 2} {z 3}}
       {mdist {+ {get this x}
                 {+ {get this y}
                    {get this z}}}}
       {addDist {+ {send arg mdist 0}
                   {send this mdist 0}}}}
      addDist
      {object
       {{x 3} {y 4}}
       {mdist {+ {get this x}
                 {get this y}}}}}
```

Analogous Java code

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

Expressions

```
(define-type ExprC
  [numC (n : number)]
  [plusC (lhs : ExprC)
         (rhs : ExprC)]
  [multC (lhs : ExprC)
         (rhs : ExprC)]
  [argC]
  [thisC]
  [objectC (field-names : (listof symbol))
           (field-exprs : (listof ExprC))
           (methods : (listof Method))]
  [getC (obj-expr : ExprC)
        (field-name : symbol)]
  [sendC (obj-expr : ExprC)
         (method-name : symbol)
         (arg-expr : ExprC)])

(define-type Method
  [method (name : symbol)
          (body-expr : ExprC)])
```

Values

```
(define-type Value
  [numV (n : number)]
  [objV (field-names : (listof symbol))
        (field-values : (listof Value))
        (methods : (listof Method))])
```

Examples

```
interp : (ExprC -> Value)

(test (interp (plusC (numC 1) (numC 2)))
      (numV 3))
```

Examples

```
(test (interp (objectC empty empty empty))
      (objV empty empty empty))
```

Examples

```
(test (interp (objectC (list 'x)
                       (list (plusC (numC 1)
                                    (numC 2)))
                       (list
                        (method 'inc (plusC (argC)
                                            (numC 1))))))
      (objV (list 'x)
            (list (numV 3))
            (list (method 'inc (plusC (argC)
                                      (numC 1))))))
```

Examples

```
(test (interp (getC
               (objectC (list 'x)
                        (list (plusC (numC 1)
                                     (numC 2)))
                        (list
                         (method 'inc (plusC (argC)
                                             (numC 1)))))
               'x))
      (objV (numV 3)))
```

Examples

```
(test (interp (sendC
               (objectC (list 'x)
                        (list (plusC (numC 1)
                                     (numC 2)))
                        (list
                         (method 'inc (plusC (argC)
                                             (numC 1)))))
               'inc
               (numC 7)))
      (objV (numV 8)))
```

Examples

```
(test (interp (plusC (argC) (numC 1)))
      ???)
```

Need **`arg`** and **`this`** values...

Instead of an environment, just provide 2 values to **`interp`**

Examples

```
(define interp : (ExprC Value Value -> Value)
  (lambda (a this-val arg-val)
    ...))

(test (interp (plusC (argC) (numC 1))
              (objV empty empty empty)
              (numV 7))
      (numV 8))
```

Examples

```
(test (interp (getC (thisC) 'x)
              (objV (list 'x)
                    (list (numV 9))
                    empty)
              (numV 7))
      (numV 9))
```

Examples

```
(test (interp (plusC (numC 1) (numC 2))
              (numV -1)
              (numV -1))
      (numV 3))
```

Part 8

Interpreter

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

Interpreter

```
(define interp : (ExprC Value Value -> Value)
  (lambda (a this-val arg-val)
    (type-case ExprC a
      ...
      [objectC (field-names field-exprs methods)
               (objV field-names
                     (map (lambda (e)
                            (interp e this-val arg-val))
                          field-exprs)
                     methods)]
      ...)))
```

Interpreter

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

Interpreter

```
(define interp : (ExprC Value Value -> Value)
  (lambda (a this-val arg-val)
    (type-case ExprC a
      ...
      [sendC (obj-expr method-name arg-expr)
             (local [(define obj
                       (interp obj-expr this-val arg-val))
                     (define arg-val
                       (interp arg-expr this-val arg-val))]
               (type-case Value obj
                 [objV (field-names field-vals methods)
                       (type-case Method (find-method method-name
                                                      methods)
                         [method (name body-expr)
                                 (interp body-expr
                                         obj
                                         arg-val)])]
                 [else (error 'interp "not an object")]))]
      ...)))
```

Helpers

```
(define (make-find [name-of : ('a -> symbol)])
  (lambda ([name : symbol] [vals : (listof 'a)]) : 'a
    (cond
     [(empty? vals)
      (error 'find "not found")]
     [else (if (equal? name (name-of (first vals)))
               (first vals)
               ((make-find name-of) name (rest vals)))])))

(define find-method : (symbol (listof Method) -> Method)
  (make-find method-name))
```

Helpers

```
; A non-list pair:
(define-type (Pair 'a 'b)
  [kons (first : 'a) (rest : 'b)])


(define (get-field [name : symbol]
                   [field-names : (listof symbol)]
                   [vals : (listof Value)])
  ; Pair fields and values, find by field name,
  ; then extract value from pair
  (kons-rest ((make-find kons-first)
              name
              (map2 kons field-names vals))))
```