Part I

Functional Programming

Functional programming is avoiding state

```
(define (area [s : Shape]) : Number
  ....)
(define (scale [s : Shape] [n : Number]) : Shape
  ....)

(define r (rectangle 10 15))
(test (area r)
      150)
(test (area (scale r 2))
      600)
(test (area r)
      150)
```

The alternative: ***imperative programming***

Functional Programming

Functional programming often means using functions as values

```
(map (lambda (s) (> (area s) 40))
     (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

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

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

```
(define (perim
  (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

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

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

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

Object-oriented: call a variant with an operation

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

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

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) ⇒ 150


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

Part 5

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 plait #:untyped ...

; A Shape is
;   (list (values 'area (-> Number))
;         (values 'bigger-than? (Number -> Boolean))
;         ....)

(define (find [l : (Listof (Symbol * 'a))] [name : Symbol]) : 'a
  (type-case (Listof (Symbol * 'a)) l
    [empty
     (error 'find (string-append "not found: " (symbol->string name)))]
    [(cons p rst-l)
     (if (symbol=? (fst p) name)
         (snd p)
         (find rst-l name))]))
```

Representing Objects with Functions

```
#lang plait #:untyped ...

; A Shape is
;  (list (values 'area (-> Number))
;        (values 'bigger-than? (Number -> Boolean))
;        ....)

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

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

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

Part 7

Objects without Functions

In some contexts:

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

... choice of organization with implications for extensibility

In other contexts:

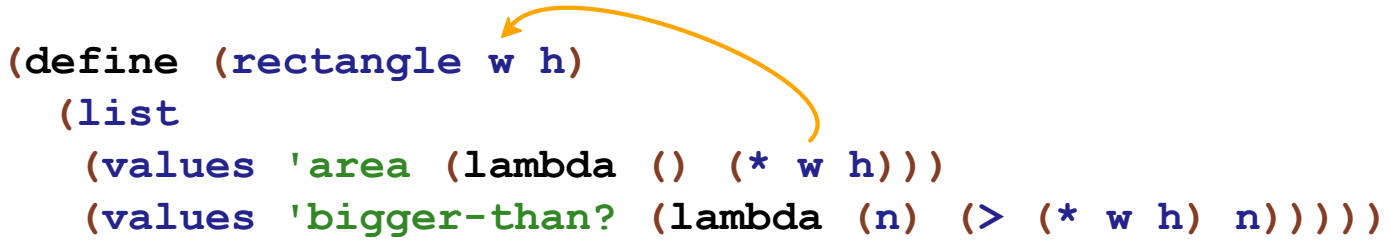
functional vs. ***object-oriented***

... choice of language primitives

Representing Objects with Higher-Order Functions

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

Representing Objects with Higher-Order Functions



```
(define (rectangle w h)
  (list
   (values 'area (lambda () (* w h)))
   (values '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)
  (values (list (values 'w init-w) (values 'h init-h))
          (list (values 'area (lambda (this)
                                (* (get this w)
                                   (get this h))))
                (values '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)
  (values (list (values 'w init-w) (values 'h init-h))
          (list (values 'area r-area)
                (values 'bigger-than? r-bigger-than?))))
```

Representing Objects with First-Order Functions

```
; A Shape is
;  (values (list ....)
;          (list (values 'area (Shape -> Number))
;                (values 'bigger-than? (Shape Number -> Number))))

(define-syntax-rule (get o-expr f-id)
  (find (fst o-expr) 'f-id))

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

(define-syntax-rule (send o-expr m-id arg-expr ...)
  (let ([o o-expr])
    ((find (snd o) 'm-id) o arg-expr ...)))

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

Part 8

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 (arg) body-expr] ...)
  ....)

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

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

Part 9

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

```
<Exp>     ::=  …
            |  {object {<Field>*} <Method>*}
            |  {get <Exp> <Symbol>}
            |  {send <Exp> <Symbol> <Exp>}
<Field>   ::=  [<Symbol> <Exp>]
<Method>  ::=  [<Symbol> {arg} <Exp>]
```

Objects Instead of Functions

```
<Exp>      ::=  ...
             |  {object {<Field>*} <Method>*}
             |  {get <Exp> <Symbol>}
             |  {send <Exp> <Symbol> <Exp>}
<Field>    ::=  [<Symbol> <Exp>]
<Method>   ::=  [<Symbol> {ar
```

evaluated when object is created

Objects Instead of Functions

```
<Exp>      ::=  …
             |  {object {<Field>*} <Method>*}
             |  {get <Exp> <Symbol>}
             |  {send <Exp> <Symbol> <Exp>}
<Field>    ::=  [<Symbol> <Exp>]
<Method>   ::=  [<Symbol> {arg} <Exp>]
```

delayed until method is called

Objects Instead of Functions

```
<Exp>      ::=  ...
             |  {object {<Field>*} <Method>*}
             |  {get <Exp> <Symbol>}
             |  {send <Exp> <Symbol> <Exp>}
<Field>    ::=  [<Symbol> <Exp>]
<Method>   ::=  [<Symbol> {arg} <Exp>]
```

method always takes one argument

Objects Instead of Functions

```
<Exp>     ::=  …
            |  {object {<Field>*} <Method>*}
            |  {get <Exp> <Symbol>}
            |  {send <Exp> <Symbol> <Exp>}
<Field>   ::=  [<Symbol> <Exp>]
<Method>  ::=  [<Symbol> {arg} <Exp>]
```

use `arg` to access method argument

Objects Instead of Functions

```
<Exp>      ::=  …
             |  {object {<Field>*} <Method>*}
             |  {get <Exp> <Symbol>}
             |  {send <Exp> <Symbol> <Exp>}
<Field>    ::=  [<Symbol> <Exp>]
<Method>   ::=  [<Symbol> {arg} <Exp>]
```

use `this` to access enclosing object

Objects Instead of Functions

```
<Exp>      ::=  ...
             |  {object {<Field>*} <Method>*}
             |  {get <Exp> <Symbol>}
             |  {send <Exp> <Symbol> <Exp>}
<Field>    ::=  [<Symbol> <Exp>]
<Method>   ::=  [<Symbol> {ar
```

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

Objects Instead of Functions

```
<Exp>     ::=  ...
            |  {object {<Field>*} <Method>*}
            |  {get <Exp> <Symbol>}
            |  {send <Exp> <Symbol> <Exp>}
<Field>   ::=  [<Symbol> <Exp>]
<Method>  ::=  [<Symbol> {arg} <Exp>]
```

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

Objects Instead of Functions

```
<Exp>      ::=  ...
             |  {object {<Field>*} <Method>*}
             |  {get <Exp> <Symbol>}
             |  {send <Exp> <Symbol> <Exp>}
<Field>    ::=  [<Symbol> <Exp>]
<Method>   ::=  [<Symbol> {arg} <Exp>]
```

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

Objects Instead of Functions

```
<Exp>      ::=  ...
             |  {object {<Field>*} <Method>*}
             |  {get <Exp> <Symbol>}
             |  {send <Exp> <Symbol> <Exp>}
<Field>    ::=  [<Symbol> <Exp>]
<Method>   ::=  [<Symbol> {arg} <Exp>]
```

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

Objects Instead of Functions

```
<Exp>      ::=  ...
             |  {object {<Field>*} <Method>*}
             |  {get <Exp> <Symbol>}
             |  {send <Exp> <Symbol> <Exp>}
<Field>    ::=  [<Symbol> <Exp>]
<Method>   ::=  [<Symbol> {arg} <Exp>]
```

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

Objects Instead of Functions

```
<Exp>     ::=  ...
            |  {object {<Field>*} <Method>*}
            |  {get <Exp> <Symbol>}
            |  {send <Exp> <Symbol> <Exp>}
<Field>   ::=  [<Symbol> <Exp>]
<Method>  ::=  [<Symbol> {arg} <Exp>]
```

```
{send {object
       {}
       [inc {arg} {+ arg 1}]}
      inc
      2}
⇒
3
```

Objects Instead of Functions

```
<Exp>      ::=  …
              |  {object {<Field>*} <Method>*}
              |  {get <Exp> <Symbol>}
              |  {send <Exp> <Symbol> <Exp>}
<Field>    ::=  [<Symbol> <Exp>]
<Method>   ::=  [<Symbol> {arg} <Exp>]
```

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

Objects Instead of Functions

```
<Exp>     ::=  ...
            |  {object {<Field>*} <Method>*}
            |  {get <Exp> <Symbol>}
            |  {send <Exp> <Symbol> <Exp>}
<Field>   ::=  [<Symbol> <Exp>]
<Method>  ::=  [<Symbol> {arg} <Exp>]
```

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

Objects Instead of Functions

```
<Exp>      ::=  …
              |  {object {<Field>*} <Method>*}
              |  {get <Exp> <Symbol>}
              |  {send <Exp> <Symbol> <Exp>}
<Field>    ::=  [<Symbol> <Exp>]
<Method>   ::=  [<Symbol> {arg} <Exp>]
```

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

Part 10

Expressive Power

Functions can encode objects

```
(list (values 'area (lambda () (* w h)))
      (values 'bigger-than? (lambda (n) (> (* w h) n))))
```

Objects can encode functions?

Objects Instead of Functions

Objects can encode functions:

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

≈

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

Objects Instead of Functions

Objects can encode functions:

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

≈

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

Part II

Object Language

```
<Exp>  ::=  <Number>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  arg
         |  this
         |  {object {<Field>*} <Method>*}
         |  {get <Exp> <Symbol>}
         |  {send <Exp> <Symbol> <Exp>}
```

```
{send {object
       {[x 3] [y 4]}
       [mdist {arg} {+ {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

```
<Exp>  ::=  <Number>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  arg
         |  this
         |  {object {<Field>*} <Method>*}
         |  {get <Exp> <Symbol>}
         |  {send <Exp> <Symbol> <Exp>}
```

```
{send {object
       {[x 1] [y 2] [z 3]}
       [mdist {arg} {+ {get this x}
                       {+ {get this y}
                          {get this z}}}]
       [addDist {arg} {+ {send arg mdist 0}
                         {send this mdist 0}}]}
      addDist
      {object
       {[x 3] [y 4]}
       [mdist {arg} {+ {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.z;
  }
  int addDist(Posn p) {
    return p.mdist() + this.mdist();
  }
}
new Posn3D(1,2,3).addDist(new Posn(3,4))
```

Expressions

```
(define-type Exp
  (numE [n : Number])
  (plusE [lhs : Exp]
         [rhs : Exp])
  (multE [lhs : Exp]
         [rhs : Exp])
  (argE)
  (thisE)
  (objectE [fields : (Listof (Symbol * Exp))]
           [methods : (Listof (Symbol * Exp))])
  (getE [obj-expr : Exp]
        [field-name : Symbol])
  (sendE [obj-expr : Exp]
         [method-name : Symbol]
         [arg-expr : Exp]))
```

Values

```
(define-type Value
  (numV [n : Number])
  (objV [fields : (Listof (Symbol * Value))]
        [methods : (Listof (Symbol * Exp))]))
```

Examples

```
interp : (Exp -> Value)

(test (interp (plusE (numE 1) (numE 2)))
      (numV 3))
```

Examples

```
(test (interp (objectE empty empty))
      (objV empty empty))
```

Examples

```
(test (interp (objectE (list
                        (values 'x (plusE (numE 1)
                                          (numE 2))))
                       (list
                        (values 'inc (plusE (argE)
                                            (numE 1))))))
      (objV (list (values 'x (numV 3)))
            (list (values 'inc (plusE (argE)
                                      (numE 1))))))
```

Examples

```
(test (interp (getE
               (objectE (list
                         (values 'x (plusE (numE 1)
                                           (numE 2))))
                        (list
                         (values 'inc (plusE (argE)
                                             (numE 1)))))
               'x))
      (numV 3))
```

Examples

```
(test (interp (sendE
               (objectE (list
                         (values 'x (plusE (numE 1)
                                           (numE 2))))
                        (list
                         (values 'inc (plusE (argE)
                                             (numE 1)))))
               'inc
               (numE 7)))
      (numV 8))
```

Examples

```
(test (interp (plusE (argE) (numE 1)))
      ???)
```

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

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

Examples

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

(test (interp (plusE (argE) (numE 1))
              (objV empty empty)
              (numV 7))
      (numV 8))
```

Examples

```
(test (interp (getE (thisE) 'x)
              (objV (list (values 'x (numV 9)))
                    empty)
              (numV 7))
      (numV 9))
```

Examples

```
(test (interp (plusE (numE 1) (numE 2))
              (objV empty empty)
              (numV 0))
      (numV 3))
```

Part 12

Interpreter

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

Interpreter

```
(define interp : (Exp Value Value -> Value)
  (lambda (a this-val arg-val)
    (type-case Exp a
      ...
      [(objectE fields methods)
       .... (map (lambda (f)
                   (let ([name (fst f)]
                         [exp (snd f)])
                     .... (interp exp this-val arg-val) ....))
                 fields)
       ....]
      ...)))
```

Interpreter

```
(define interp : (Exp Value Value -> Value)
  (lambda (a this-val arg-val)
    (type-case Exp a
      ...
      [(objectE fields methods)
       (objV (map (lambda (f)
                    (let ([name (fst f)]
                          [exp (snd f)])
                      .... (interp exp this-val arg-val) ....))
                  fields)
             methods)]
      ...)))
```

Interpreter

```
(define interp : (Exp Value Value -> Value)
  (lambda (a this-val arg-val)
    (type-case Exp a
      ...
      [(objectE fields methods)
       (objV (map (lambda (f)
                    (let ([name (fst f)]
                          [exp (snd f)])
                      (values name
                              (interp exp this-val arg-val))))
                  fields)
             methods)]
      ...)))
```

Interpreter

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

Interpreter

```
(define interp : (Exp Value Value -> Value)
  (lambda (a this-val arg-val)
    (type-case Exp a
      ...
      [(sendE 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 fields methods)
            (let ([body-expr (find methods method-name)])
              (interp body-expr
                      obj
                      arg-val))]
           [else (error 'interp "not an object")]))]
      ...)))
```