Part I

Functional Programming

Functional programming is avoiding state

```
fun area(s :: Shape) :: Int:
  ....
fun scale(s :: Shape, n :: Int) :: Shape:
  ....

def r = rectangle(10, 15)
check: area(r)
       ~is 150
check: area(scale(r, 2))
       ~is 600
check: area(r)
       ~is 150
```

The alternative: ***imperative programming***

Functional Programming

Functional programming often means using functions as values

```
map(fun (s): area(s) > 40,
    [rectangle(10, 5),
     square(6),
     equilateral_triangle(7)])
```

The alternative: ***procedural programming***

Functional Programming

Functional programming often means
datatype-oriented programming

```
type Shape
| rectangle(w :: Int, h :: Int)
| square(side :: Int)
| equilateral_triangle(side :: Int)

fun area(s :: Shape) :: Int:
  match s
   | rectangle(w, h): ....
   | square(s): ....
   | equilateral_triangle(s): ....
```

Functional Programming

Functional programming often means
datatype-oriented programming

```
fun sum_of_areas(l :: Listof(Shape)):
  match l
  | []: 0
  | cons(s, rst_l): area(s)
                    + sum_of_areas(rst_l)
```

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

Datatype-Oriented versus Object-Oriented

Datatype-oriented: call an operation with a variant

```
fun area(s :: Shape) :: Int:
  match s
  | rectangle(w, h): ....
  | square(s): ....
  | equilateral_triangle(s): ....
```

```
fun perimeter(s :: Shape) :: Int:
  match s
  | rectangle(w, h): ....
  | square(s): ....
  | equilateral_triangle(s): ....
```

Datatype-Oriented versus Object-Oriented

Datatype-oriented: call an operation with a variant

```
fun area(s :: Shape) :: Int:
  match s
  | rectangle(w, h): ....
  | square(s): ....
  | equilateral_triangle(s): ....
```

```
fun perimeter(s :: Shape) :: Int:
  match 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

```
fun area(s :: Shape) :: Int:
  match s
  | rectangle(w, h): ....
  | square(s): ....
  | equilateral_triangle(s): ....
```

```
fun perimete
  match 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

```
fun area(s :: Shape) :: Int:
  match s
  | rectangle(w, h): ....
  | square(s): ....
  | equilateral_triangle(s): ....
```

```
fun perimete
  match 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

```
type Shape = () -> Int


fun rectangle(w, h) :: Shape:
  fun ():
    w * h

fun square(s) :: Shape:
  fun ():
    s * s

def r = rectangle(10, 15)
r() ⇒ 150


def c = (let r = 10:
           fun (): 3 * r * r)
c() ⇒ 300
```

Part 5

Objects and Multiple Operations

Simple function implements an object with a single operation:

```
type Shape = () -> Int

def r = rectangle(10, 15)
r()
```

For multiple operations, could pass a symbol to select:

```
type Shape = Symbol -> Int

def r = rectangle(10, 15)
r(#'area)
r(#'perimeter)
```

Representing Objects with Functions

```
type Shape = Symbol -> Int

fun rectangle(w, h) :: Shape:
  fun (op):
    cond
    | op == #'area: w * h
    | op == #'perimeter: 2 * (w + h)

fun square(s) :: Shape:
  fun (op):
    cond
    | op == #'area: s * s
    | op == #'perimeter: 4 * s

def r = rectangle(10, 15)
r(#'area) ⇒ 150
r(#'perimeter) ⇒ 50
```

Representing Objects with Functions

```
#lang shplait
~untyped

// A Shape is
//   [values(#'area, -> Int),
//    values(#'is_bigger_than, Int -> Boolean),
//    ....]

fun find(l :: Listof(Symbol * ?a), name :: Symbol) :: ?a:
  match l
  | []: error(#'find, "not found: " +& name)
  | cons(p, rst_l): if fst(p) == name
                    | snd(p)
                    | find(rst_l, name)
```

Representing Objects with Functions

```
#lang shplait
~untyped

// A Shape is
//  [values(#'area, -> Int),
//   values(#'is_bigger_than, Int -> Boolean),
//   ....]

fun rectangle(w, h):
  [values(#'area, fun (): w * h),
   values(#'is_bigger_than, fun (n): w * h > n)]

fun square(s):
  [values(#'area, fun (): s * s),
   values(#'is_bigger_than, fun (n): s * s > n)]

def r = rectangle(10, 15)
find(r, #'area)() ⇒ 150
find(r, #'is_bigger_than)(100) ⇒ #true
```

Representing Objects with Functions

```
fun rectangle(w, h):
  [values(#'area, fun (): w * h),
   values(#'is_bigger_than, fun (n): w * h > n)]


macro '$o_expr . $(id :: Identifier) ($arg, ...)':
  'find($o_expr, #' $id)($arg, ...)'


def r = rectangle(10, 15)
r.area() ⇒ 150
r.is_bigger_than(100) ⇒ #true
```

Part 7

Objects without Functions

In some contexts:

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

... choice of organization with implications for extensibility

In other contexts:

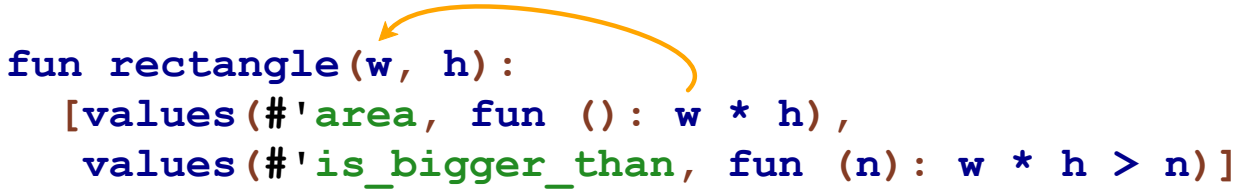
functional vs. ***object-oriented***

... choice of language primitives

Representing Objects with Higher-Order Functions

```
fun rectangle(w, h):
  [values(#'area, fun (): w * h),
   values(#'is_bigger_than, fun (n): w * h > n)]
```

Representing Objects with Higher-Order Functions



```
fun rectangle(w, h):
  [values(#'area, fun (): w * h),
   values(#'is_bigger_than, fun (n): w * h > n)]
```

Relies on nested functions

... implemented as closures

Representing Objects with First-Order Functions

```
macro 'rectangle($init_w, $init_h)':
  'values([values(#'w, $init_w),
           values(#'h, $init_h)],
          [values(#'area, fun (this):
                            this.w * this.h),
           values(#'is_bigger_than, fun (this, n):
                                      this.area() > n)])'
```

Use two lists: fields and methods

Pass "this" to methods to access fields

Representing Objects with First-Order Functions

```
macro 'rectangle($init_w, $init_h)':
  'values([values(#'w, $init_w),
           values(#'h, $init_h)],
          [values(#'area, fun (this):
                            this.w * this.h),
           values(#'is_bigger_than, fun (this, n):
                                      this.area() > n)])'
```

Could be written as

```
fun r_area(this):
  this.w * this.h

fun r_is_bigger_than(this, n):
  this.area() > n

macro 'rectangle($init_w, $init_h)':
  'values([values(#'w, $init_w),
           values(#'h, $init_h)],
          [values(#'area, r_area),
           values(#'is_bigger_than, r_is_bigger_than)])'
```

Representing Objects with First-Order Functions

```
// A Shape is
//  values([....],
//         [values(#'area, Shape -> Int),
//          values(#'is_bigger_than, Shape Int -> Int)])

macro 'rectangle($init_w, $init_h)':
  'values([values(#'w, $init_w),
           values(#'h, $init_h)],
          [values(#'area, fun (this):
                            this.w * this.h),
           values(#'is_bigger_than, fun (this, arg):
                                      this.area() > arg)])'

macro
| '$o_expr . $(id :: Identifier) ($arg, ...)':
    'let o = $o_expr:
       find(snd(o), #' $id)(o, $arg, ...)'
| '$o_expr . $(id :: Identifier)':
    'find(fst($o_expr), #' $id)'

def r = rectangle(10, 15)
r.w ⇒ 10
r.is_bigger_than(200) ⇒ #false
```

Part 8

Representing Objects with First-Order Functions

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

```
macro 'object ($field_id = $field_expr, ...):
         method $method_id(arg): $body_expr
         ...':
  '....'

macro 'rectangle($init_w, $init_h)':
  'object (w = $init_w,
           h = $init_h):
     method area(arg): this.w * this.h
     method is_bigger_than(arg): this.area(0) > arg'

def r = rectangle(10, 15)
r.area(0) ⇒ 150
r.is_bigger_than(100) ⇒ #true
```

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>
                  ...
             |  object (<Field>, ...)
             |  <Exp>.<Symbol>
             |  <Exp>.<Symbol>(<Exp>)
<Field>    ::=  <Symbol> = <Exp>
<Method>   ::=  method <Symbol>(arg):
                  <Exp>
```

Objects Instead of Functions

```
<Exp>      ::=  ....
              |  object (<Field>, ...):
                   <Method>
                   ...
              |  object (<Field>, ...)
              |  <Exp>.<Symbol>
              |  <Exp>.<Symbol>(<Exp>)
<Field>    ::=  <Symbol> = <Exp>
<Method>   ::=  method <Symbol
                   <Exp>
```

evaluated when object is created

Objects Instead of Functions

```
<Exp>      ::=  ....
              |  object (<Field>, ...):
                   <Method>
                   ...
              |  object (<Field>, ...)
              |  <Exp>.<Symbol>
              |  <Exp>.<Symbol>(<Exp>)
<Field>    ::=  <Symbol> = <Exp>
<Method>   ::=  method <Symbol>(arg):
                   <Exp>
```

delayed until method is called

Objects Instead of Functions

```
<Exp>      ::=  ....
              |  object (<Field>, ...):
                   <Method>
                   ...
              |  object (<Field>, ...)
              |  <Exp>.<Symbol>
              |  <Exp>.<Symbol>(<Exp>)
<Field>    ::=  <Symbol> = <Exp>
<Method>   ::=  method <Symbol>(arg):
                   <Exp>
```

method always takes one argument

Objects Instead of Functions

```
<Exp>      ::=  ....
              |  object (<Field>, ...):
                   <Method>
                   ...
              |  object (<Field>, ...)
              |  <Exp>.<Symbol>
              |  <Exp>.<Symbol>(<Exp>)
<Field>    ::=  <Symbol> = <Exp>
<Method>   ::=  method <Symbol>(arg):
                   <Exp>
```

use **arg** to access method argument

Objects Instead of Functions

```
<Exp>      ::=  ....
              |  object (<Field>, ...):
                   <Method>
                   ...
              |  object (<Field>, ...)
              |  <Exp>.<Symbol>
              |  <Exp>.<Symbol>(<Exp>)
<Field>    ::=  <Symbol> = <Exp>
<Method>   ::=  method <Symbol>(arg):
                   <Exp>
```

use **this** to access enclosing object

Objects Instead of Functions

```
<Exp>      ::=  ....
             |  object (<Field>, ...):
                  <Method>
                  ...
             |  object (<Field>, ...)
             |  <Exp>.<Symbol>
             |  <Exp>.<Symbol>(<Exp>)
<Field>    ::=  <Symbol> = <Exp>
<Method>   ::=  method <Symbol
                  <Exp>
```

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

Objects Instead of Functions

```
<Exp>      ::=  ....
             |  object (<Field>, ...):
                  <Method>
                  ...
             |  object (<Field>, ...)
             |  <Exp>.<Symbol>
             |  <Exp>.<Symbol>(<Exp>)
<Field>    ::=  <Symbol> = <Exp>
<Method>   ::=  method <Symbol>(arg):
                  <Exp>
```

```
object (x = 1, y = 2)
```

Objects Instead of Functions

```
<Exp>      ::=  ....
              |  object (<Field>, ...):
                   <Method>
                   ...
              |  object (<Field>, ...)
              |  <Exp>.<Symbol>
              |  <Exp>.<Symbol>(<Exp>)
<Field>    ::=  <Symbol> = <Exp>
<Method>   ::=  method <Symbol>(arg):
                   <Exp>
```

```
object (x = 1, y = 1 + 1)
```

Objects Instead of Functions

```
<Exp>     ::=  ....
           |   object (<Field>, ...):
                 <Method>
                 ...
           |   object (<Field>, ...)
           |   <Exp>.<Symbol>
           |   <Exp>.<Symbol>(<Exp>)
<Field>   ::=  <Symbol> = <Exp>
<Method>  ::=  method <Symbol>(arg):
                 <Exp>
```

```
(object (x = 1, y = 2)).x
⇒
1
```

Objects Instead of Functions

```
<Exp>      ::=  ....
             |  object (<Field>, ...):
                  <Method>
                  ...
             |  object (<Field>, ...)
             |  <Exp>.<Symbol>
             |  <Exp>.<Symbol>(<Exp>)
<Field>    ::=  <Symbol> = <Exp>
<Method>   ::=  method <Symbol>(arg):
                  <Exp>
```

```
object ():
  method inc(arg):
    arg + 1
```

Objects Instead of Functions

```
<Exp>      ::=  ....
             |  object (<Field>, ...):
                  <Method>
                  ...
             |  object (<Field>, ...)
             |  <Exp>.<Symbol>
             |  <Exp>.<Symbol>(<Exp>)
<Field>    ::=  <Symbol> = <Exp>
<Method>   ::=  method <Symbol>(arg):
                  <Exp>
```

```
(object ():
   method inc(arg):
     arg + 1).inc(2)
⇒
3
```

Objects Instead of Functions

```
<Exp>      ::=  ....
              |  object (<Field>, ...):
                   <Method>
                   ...
              |  object (<Field>, ...)
              |  <Exp>.<Symbol>
              |  <Exp>.<Symbol>(<Exp>)
<Field>    ::=  <Symbol> = <Exp>
<Method>   ::=  method <Symbol>(arg):
                   <Exp>
```

```
object ():
  method inc(arg):
    arg + 1
  method dec(arg):
    arg + -1
```

Objects Instead of Functions

```
<Exp>      ::=  ....
             |  object (<Field>, ...):
                  <Method>
                  ...
             |  object (<Field>, ...)
             |  <Exp>.<Symbol>
             |  <Exp>.<Symbol>(<Exp>)
<Field>    ::=  <Symbol> = <Exp>
<Method>   ::=  method <Symbol>(arg):
                  <Exp>
```

```
object (x = 1, y = 2):
  method mdist(arg):
    this.x + this.y
```

Objects Instead of Functions

```
<Exp>     ::=  ....
            |  object (<Field>, ...):
                 <Method>
                 ...
            |  object (<Field>, ...)
            |  <Exp>.<Symbol>
            |  <Exp>.<Symbol>(<Exp>)
<Field>   ::=  <Symbol> = <Exp>
<Method>  ::=  method <Symbol>(arg):
                 <Exp>
```

```
(object (x = 1, y = 2):
   method mdist(arg):
     this.x + this.y).mdist(0)
⇒
3
```

Part 10

Expressive Power

Functions can encode objects

```
[values(#'area, fun (): w * h),
 values(#'is_bigger_than, fun (n): w * h > n)]
```

Objects can encode functions?

Objects Instead of Functions

Objects can encode functions:

```
(fun (x): x)(5)
```

≈

```
(object ():
   method call(arg): arg).call(5)
```

Objects Instead of Functions

Objects can encode functions:

```
(fun (x): fun (y): x + y)(5)(6)
```

≈

```
(object ():
   method call(arg):
     object (x = arg):
       method call(arg):
         arg + this.x).call(5).call(6)
```

Part II

Object Language

```
<Exp>  ::=  <Int>
        |   <Exp> + <Exp>
        |   <Exp> * <Exp>
        |   arg
        |   this
        |   object (<Field>, ...)
        |   object (<Field>, ...):
              <Method>
              ...
        |   <Exp>.<Symbol>
        |   <Exp>.<Symbol>(<Exp>)
```

```
(object (x = 3, y = 4):
   method mdist(arg):
     this.x + this.y).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>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  arg
         |  this
         |  object (<Field>, ...)
         |  object (<Field>, ...):
              <Method>
              ...
         |  <Exp>.<Symbol>
         |  <Exp>.<Symbol>(<Exp>)
```

```
(object (x = 1, y = 2, z = 3):
   method mdist(arg):
     this.x + this.y + this.z
   method addDist(arg):
     arg.mdist(0)
       + this.mdist(0)).addDist(object (x = 3, y = 4):
                                  method mdist(arg):
                                    this.x + 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

```
type Exp
| intE(n :: Int)
| 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

```
type Value
| intV(n :: Int)
| objV(fields :: Listof(Symbol * Value),
       methods :: Listof(Symbol * Exp))
```

Examples

```
interp :: Exp -> Value

check: interp(plusE(intE(1), intE(2)))
       ~is intV(3)
```

Examples

```
check: interp(objectE([], []))
       ~is objV([], [])
```

Examples

```
check: interp(objectE([values(#'x, plusE(intE(1),
                                         intE(2)))],
                      [values(#'inc, plusE(argE(),
                                           intE(1)))]))
       ~is objV([values(#'x, intV(3))],
                [values(#'inc, plusE(argE(),
                                     intE(1)))])
```

Examples

```
check: interp(getE(objectE([values(#'x, plusE(intE(1),
                                              intE(2)))],
                           [values(#'inc, plusE(argE(),
                                                intE(1)))]),
                    #'x))
       ~is intV(3)
```

Examples

```
check: interp(sendE(objectE([values(#'x, plusE(intE(1),
                                               intE(2)))],
                            [values(#'inc, plusE(argE(),
                                                 intE(1)))]),
                    #'inc,
                    intE(7)))
       ~is intV(8)
```

Examples

```
check: interp(plusE(argE(), intE(1)))
       ~is ???
```

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

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

Examples

```
def interp :: (Exp, Value, Value) -> Value:
  fun (a, this_val, arg_val):
    ....

check: interp(plusE(argE(), intE(1)),
              objV([], []),
              intV(7))
       ~is intV(8)
```

Examples

```
check: interp(getE(thisE(), #'x),
              objV([values(#'x, intV(9))],
                   []),
              intV(7))
       ~is intV(9)
```

Examples

```
check: interp(plusE(intE(1), intE(2)),
              objV([], []),
              intV(0))
       ~is intV(3)
```

Part 12

Interpreter

```
def interp :: (Exp, Value, Value) -> Value:
  fun (a, this_val, arg_val):
    match a
    | ....
    | intE(n): intV(n)
    | plusE(l, r): num_plus(interp(l, this_val, arg_val),
                            interp(r, this_val, arg_val))
    | multE(l, r): num_mult(interp(l, this_val, arg_val),
                            interp(r, this_val, arg_val))
    | thisE(): this_val
    | argE(): arg_val
    | ....
```

Interpreter

```
def interp :: (Exp, Value, Value) -> Value:
  fun (a, this_val, arg_val):
    match a
    | ....
    | objectE(fields, methods):
        .... map(fun (f):
                   def name = fst(f)
                   def exp = snd(f)
                   .... interp(exp, this_val, arg_val) ....,
                 fields)
        ....
    | ....
```

Interpreter

```
def interp :: (Exp, Value, Value) -> Value:
  fun (a, this_val, arg_val):
    match a
    | ....
    | objectE(fields, methods):
        objV(map(fun (f):
                   def name = fst(f)
                   def exp = snd(f)
                   .... interp(exp, this_val, arg_val) ....,
                 fields),
             methods)
    | ....
```

Interpreter

```
def interp :: (Exp, Value, Value) -> Value:
  fun (a, this_val, arg_val):
    match a
    | ....
    | objectE(fields, methods):
        objV(map(fun (f):
                   def name = fst(f)
                   def exp = snd(f)
                   values(name,
                          interp(exp, this_val, arg_val)),
                 fields),
             methods)
    | ....
```

Interpreter

```
def interp :: (Exp, Value, Value) -> Value:
  fun (a, this_val, arg_val):
    match a
    | ....
    | getE(obj_expr, field_name):
        match interp(obj_expr, this_val, arg_val)
        | objV(fields, methods):
            find(fields, field_name)
        | ~else: error(#'interp, "not an object")
    | ....
```

Interpreter

```
def interp :: (Exp, Value, Value) -> Value:
  fun (a, this_val, arg_val):
    match a
    | ....
    | sendE(obj_expr, method_name, arg_expr):
        def obj:
          interp(obj_expr, this_val, arg_val)
        def next_arg_val:
          interp(arg_expr, this_val, arg_val)
        match obj
        | objV(fields, methods):
            let body_expr = find(methods, method_name):
              interp(body_expr,
                     obj,
                     next_arg_val)
        | ~else: error(#'interp, "not an object")
    | ....
```