From FP to OOP

Start with data:

```
; A posn is
;  (make-posn num num)
(define-struct posn (x y))
```

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

From FP to OOP

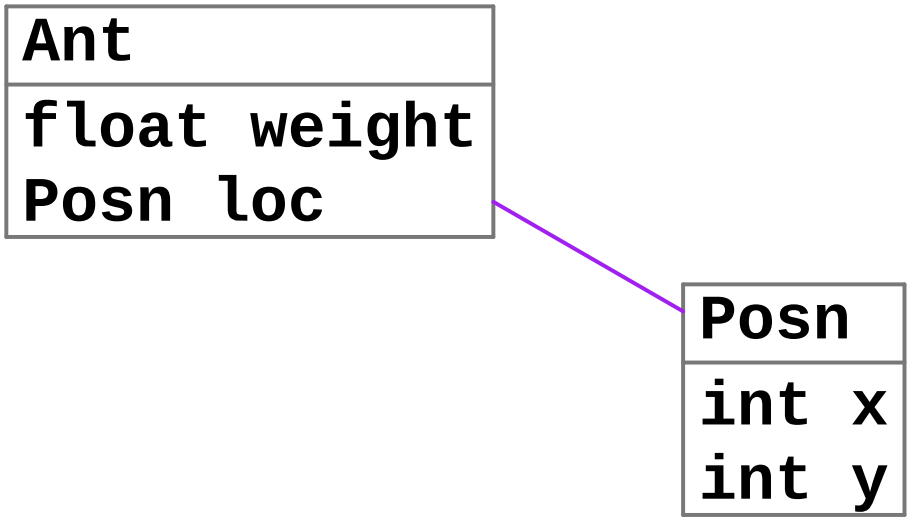
Start with data:

```
; A posn is
;  (make-posn num num)
(define-struct posn (x y))
```

Posn
int x int y

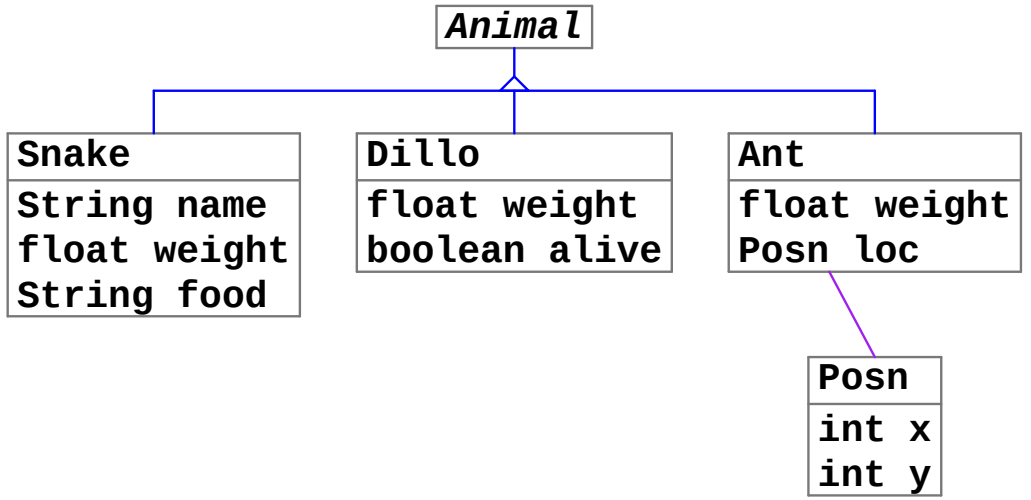
From FP to OOP

```
; An ant is
;  (make-ant num posn)

; A posn is
;  (make-posn num num)
```



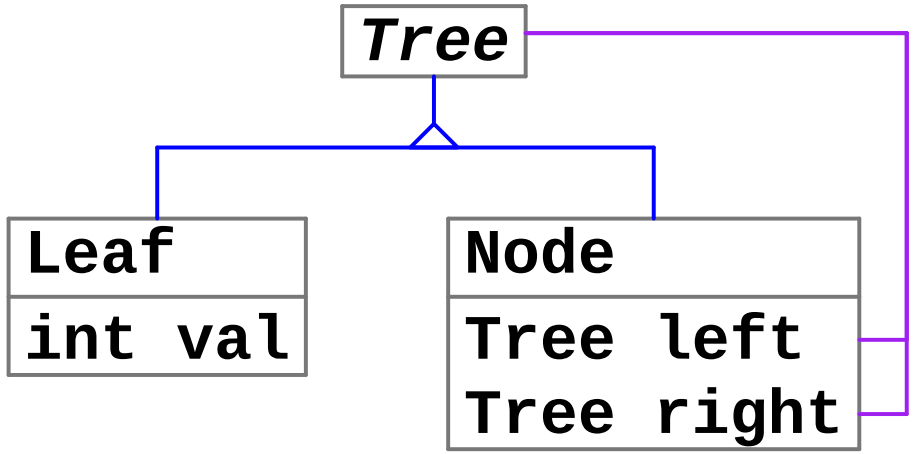
From FP to OOP

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

; A snake is
; (make-snake sym num sym)

; A dillo is
; (make-dillo num bool)

; An ant is
; (make-ant num posn)

; A posn is
; (make-posn num num)
```



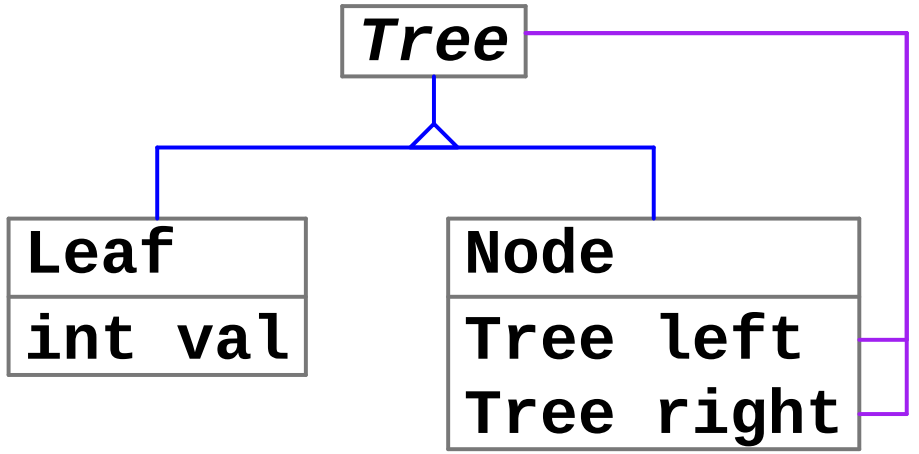
From FP to OOP

```
type tree =
    Leaf of int
  | Node of tree * tree
```



From FP to OOP

```
type tree =
    Leaf of int
  | Node of tree * tree
```



And so on (for mutually referential data definitions)...

From 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) ...)
```

```
interface Animal {
  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) { ... }
}
```

From 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) ...)
```

Data definition turns into class declarations

```
interface Animal {
  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) { ... }
}
```

From 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) ...)
```

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

```
interface Animal {
  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) { ... }
}
```

From 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

```
interface Animal {
  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) { ... }
}
```

Extensibility Problem

If we need new **`animal`** operations...

- FP: add a function (no change to old code)
- OOP: change interfaces & classes to add method

If we need new **`animal`** variants...

- FP: change all functions to add case
- OOP: add a subclass (no change to old code)

Design patterns in each style provide some benefits of the other

The Object Pattern for FP

```
(define-struct animal (lighter?-proc))

(define (animal-is-lighter? a n)
  ((animal-lighter?-proc a) n))

(define (make-snake name weight food)
  (make-animal (lambda (n) (< weight n))))

...
```

The Visitor Pattern for OOP

```
interface Animal {
  <T> accept(Visitor<T> v);
}
class Snake implements Animal {
  ...
  <T> accept(Visitor<T> v) { return v.visit(this); }
}
...
interface Visitor<T> {
  <T> visit(Snake s);
  ...
}

class IsLighter implements Visitor<boolean> {
   int n;
   ...
   boolean visit(Snake s) { return s.weight < n; }
   ...
}
```

Language Cores

FP core:

- Closures
- Datatype case dispatch
- Parametric polymorphsm

OOP core:

- Object creation
- Dynamic method dispatch
- Static method dispatch (e.g., **`super`**)

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 Dillo(...);

a.endangers(b);
```

dynamic
static

CAE Grammar

```
<prog>  ::=  <decl>* <CAE>
<decl>  ::=  {class <cid> <fid>* <meth>*}
<meth>  ::=  {<mid> <CAE>}
<CAE>   ::=  <num>
          |  {+ <CAE> <CAE>}
          |  {- <CAE> <CAE>}
          |  {if0 <CAE> <CAE> <CAE>}
          |  arg
          |  this
          |  {new <cid> <CAE>*}
          |  {get <CAE> <fid>}
          |  {dsend <CAE> <mid> <CAE>}
          |  {ssend <CAE> <cid> <mid> <CAE>}
```

```
{class posn
  x y
  {mdist {+ {get this x} {get this y}}}
  {addDist {+ {dsend arg mdist 0}
              {dsend this mdist 0}}}}
{dsend {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() + this.mdist();
  }
}
new Posn(1,2).addDist(new Posn(3,4))
```

CAE Grammar

```
<prog>  ::=  <decl>* <CAE>
<decl>  ::=  {class <cid> <fid>* <meth>*}
<meth>  ::=  {<mid> <CAE>}
<CAE>   ::=  <num>
          |  {+ <CAE> <CAE>}
          |  {- <CAE> <CAE>}
          |  {if0 <CAE> <CAE> <CAE>}
          |  arg
          |  this
          |  {new <cid> <CAE>*}
          |  {get <CAE> <fid>}
          |  {dsend <CAE> <mid> <CAE>}
          |  {ssend <CAE> <cid> <mid> <CAE>}
```

```
{class posn ...
  {addDist {+ {dsend arg mdist 0}
              {dsend 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();
  }
}
new Posn3D(1,2,3).addDist(new Posn(3,4))
```

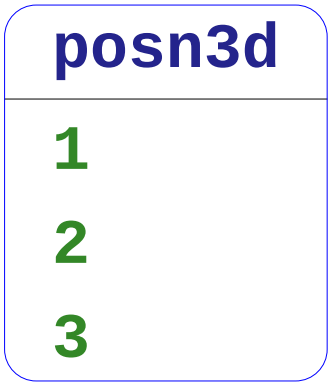
Object Values

How does

```
{send {new posn3D 1 2 3} mdist ...}
```

dispatch to the right `mdist`?

The result of `{new posn3D 1 2 3}` must contain a class tag and field values:



CAE Datatypes

```
type cae =
    Num of int
  | Add of cae * cae
  | Sub of cae * cae
  | IfZ of cae * cae * cae
  | Arg
  | This
  | New of string * cae list
  | Get of cae * string
  | DSend of cae * string * cae
  | SSend of cae * string * string * cae

and cdecl = Class of string * field list * meth list
and field = Field of string
and meth = Method of string * cae

and caeValue =
    NumV of int
  | ObjV of cdecl * caeValue list
```

CAE Interpreter

```
let rec interp : (cae * cdecl list * caeValue * caeValue
                    -> caeValue )
    = function (expr, cdecls, this, arg) ->
      let recur = fun e -> interp(e, cdecls, this, arg)
      in match expr with
      Num(n) -> NumV(n)
    | Add(l, r) -> numPlus(recur l, recur r)
    | Sub(l, r) -> numMinus(recur l, recur r)
    | IfZ(tst, thn, els) ->
        if (recur tst = NumV(0))
        then recur thn
        else recur els
    | This -> this
    | Arg -> arg
      …
      …
```

CAE Interpreter

```
let rec interp : (cae * cdecl list * caeValue * caeValue
                  -> caeValue )
    = function (expr, cdecls, this, arg) ->
      let recur = fun e -> interp(e, cdecls, this, arg)
      in match expr with
      …
    | New(name, exprs) ->
        let decl = findClass name cdecls
        in ObjV(decl, List.map recur exprs)
      …
```

CAE Interpreter

```
let rec interp : (cae * cdecl list * caeValue * caeValue
                  -> caeValue )
    = function (expr, cdecls, this, arg) ->
      let recur = fun e -> interp(e, cdecls, this, arg)
      in match expr with
      …
    | Get(expr, fname) ->
        (match recur expr with
          ObjV(Class(_, fields, _), vals) ->
            getField fname fields vals
        | _ -> raise (Failed "not an object for get"))
      …
```

CAE Interpreter

```
let rec interp : (cae * cdecl list * caeValue * caeValue
                  -> caeValue )
    = function (expr, cdecls, this, arg) ->
      let recur = fun e -> interp(e, cdecls, this, arg)
      in match expr with
      …
    | DSend(expr, mname, argExpr) ->
        (match recur expr with
          (ObjV(Class(_, _, methods), _) as this) ->
            let Method(name, body) = findMethod mname methods
            in interp(body, cdecls, this, recur argExpr)
        | _ -> raise (Failed "not an object for send"))
      …
```

CAE Interpreter

```
let rec interp : (cae * cdecl list * caeValue * caeValue
                  -> caeValue )
    = function (expr, cdecls, this, arg) ->
      let recur = fun e -> interp(e, cdecls, this, arg)
      in match expr with
      …
    | SSend(expr, cname, mname, argExpr) ->
        let this = recur expr
        in let Class(_, _, methods) = findClass cname cdecls
        in let Method(name, body) = findMethod mname methods
        in interp(body, cdecls, this, recur argExpr)
```

CAE Helpers

```
exception NoSuch of string * string

let rec find = fun what nameOf name vals ->
  match vals with
    [] -> raise (NoSuch (what, name))
  | a::rest ->
      if (name = nameOf(a))
      then a
      else find what nameOf name rest

let findClass = (find "class"
                   (fun (Class(name, _, _)) -> name))

let findMethod = (find "method"
                    (fun (Method(name, _)) -> name))

let getField = fun name fields vals ->
  snd (find "field" (fun (Field(name), _) -> name)
         name (List.map2 (fun x y -> (x,y)) fields vals))
```