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

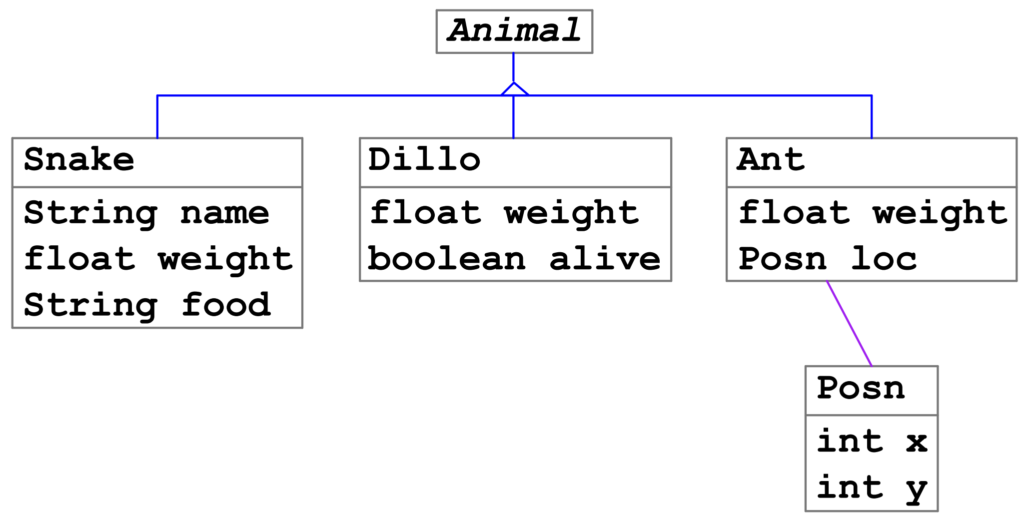
```
; An ant is
;   (make-ant num posn)

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

```
Ant
----------------
float weight
Posn loc
```

```
Posn
----------------
int x
int y
```

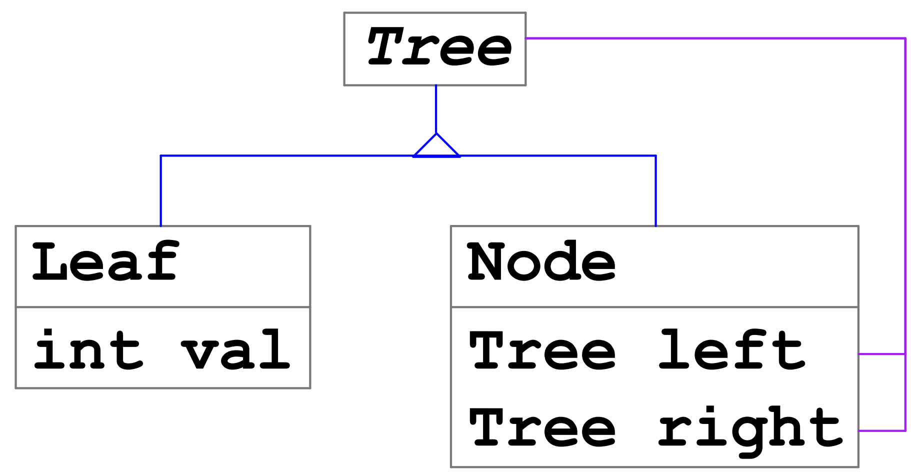
From FP to OOP

```
; 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)
```



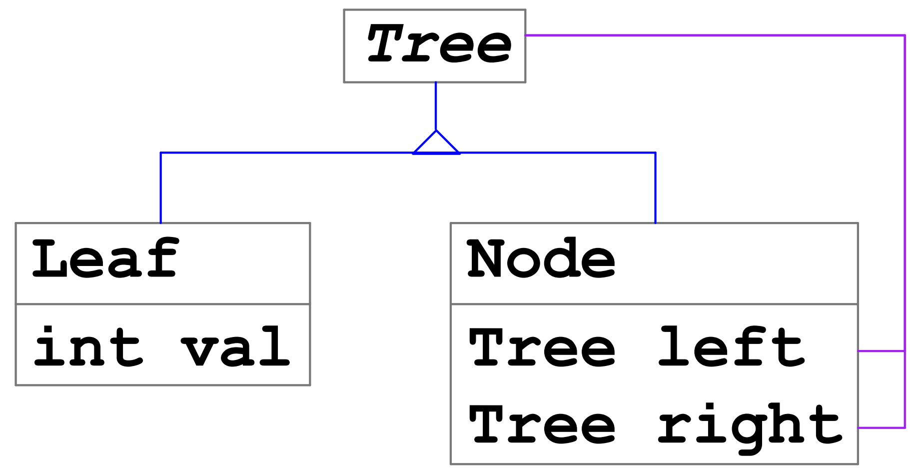
From FP to OOP

```
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:

posn3d
1
2
3

CAE Datatypes

```
(define-type CAE
  [num (n : number)]
  [str (s : string)]
  [add (lhs : CAE) (rhs : CAE)]
  [sub (lhs : CAE) (rhs : CAE)]
  [if0 (test-expr : CAE)
       (then-expr : CAE)
       (else-expr : CAE)]
  [arg]
  [this]
  [new (class : symbol)
       (args : (listof CAE))]
  [get (obj-expr : CAE)
       (field-name : symbol)]
  [dsend (obj-expr : CAE)
         (method-name : symbol)
         (arg-expr : CAE)]
  [ssend (obj-expr : CAE)
         (class-name : symbol)
         (method-name : symbol)
         (arg-expr : CAE)])
```

CAE Datatypes

```
(define-type CDecl
  [class (name : symbol)
    (fields : (listof Field))
    (methods : (listof Method))])

(define-type Field
  [field (name : symbol)])

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

(define-type CAE-Value
  [numV (n : number)]
  [strV (s : string)]
  [objV (class : CDecl)
        (field-values : (listof CAE-Value))])
```

CAE Interpreter

```
(define interp : (CAE (listof CDecl) CAE-Value CAE-Value -> CAE-Value)
  (lambda (a-cae cdecls this-val arg-val)
    (local [(define (recur expr)
              (interp expr cdecls this-val arg-val))]
      (type-case CAE a-cae
        ...
        [num (n) (numV n)]
        [str (s) (strV s)]
        [add (l r) (num+ (recur l) (recur r))]
        [sub (l r) (num- (recur l) (recur r))]
        [if0 (test-expr then-expr else-expr)
             (if (numzero? (recur test-expr))
                 (recur then-expr)
                 (recur else-expr))]
        [this () this-val]
        [arg () arg-val]
        ...))))
```

CAE Interpreter

```
(define interp : (CAE (listof CDecl) CAE-Value CAE-Value -> CAE-Value)
  (lambda (a-cae cdecls this-val arg-val)
    (local [(define (recur expr)
              (interp expr cdecls this-val arg-val))]
      (type-case CAE a-cae
        ...
        [new (class-name field-exprs)
             (local [(define cdecl (find-class class-name cdecls))
                     (define vals (map recur field-exprs))]
               (objV cdecl vals))]
        ...))))
```

CAE Interpreter

```
(define interp : (CAE (listof CDecl) CAE-Value CAE-Value -> CAE-Value)
  (lambda (a-cae cdecls this-val arg-val)
    (local [(define (recur expr)
              (interp expr cdecls this-val arg-val))]
      (type-case CAE a-cae
        ...
        [get (obj-expr field-name)
             (type-case CAE-Value (recur obj-expr)
               [objV (cdecl field-vals)
                     (type-case CDecl cdecl
                       [class (name fields methods)
                         (get-field field-name fields field-vals)])]
               [else (error 'interp "not an object")])]
        ...))))
```

CAE Interpreter

```
(define interp : (CAE (listof CDecl) CAE-Value CAE-Value -> CAE-Value)
  (lambda (a-cae cdecls this-val arg-val)
    (local [(define (recur expr)
              (interp expr cdecls this-val arg-val))]
      (type-case CAE a-cae
        ...
        [dsend (obj-expr method-name arg-expr)
               (local [(define obj (recur obj-expr))
                       (define arg-val (recur arg-expr))]
                 (type-case CAE-Value obj
                   [objV (cdecl field-vals)
                         (type-case CDecl cdecl
                           [class (name fields methods)
                             (type-case Method
                                 (find-method method-name methods)
                               [method (name body-expr)
                                       (interp body-expr
                                               cdecls
                                               obj
                                               arg-val)])])]
                   [else (error 'interp "not an object")]))]
        ...))))
```

CAE Interpreter

```
(define interp : (CAE (listof CDecl) CAE-Value CAE-Value -> CAE-Value)
  (lambda (a-cae cdecls this-val arg-val)
    (local [(define (recur expr)
              (interp expr cdecls this-val arg-val))]
      (type-case CAE a-cae
        ...
        [ssend (obj-expr class-name method-name arg-expr)
               (local [(define obj (recur obj-expr))
                       (define arg-val (recur arg-expr))]
                 (type-case CDecl (find-class class-name cdecls)
                   [class (name fields methods)
                     (type-case Method
                         (find-method method-name methods)
                       [method (name body-expr)
                               (interp body-expr
                                       cdecls
                                       obj
                                       arg-val)])])))]
        ...))))
```

CAE Helpers

```
(define (find what name-of)
  (lambda (name vals)
    (cond
     [(empty? vals)
      (error 'find (string-append
                    (string-append
                     "cannot find "
                     what)
                    (string-append
                     " "
                     (to-string name))))]
     [else (if (equal? name (name-of (first vals)))
               (first vals)
               ((find what name-of) name (rest vals)))])))

(define find-class
  (find "class" (lambda (c)
                  (type-case CDecl c
                    [class (name fields methods) name]))))
```

CAE Helpers

```
(define find-method
  (find "method" (lambda (m)
                   (type-case Method m
                     [method (name body-expr) name]))))

(define (get-field name fields vals)
  (local [(define-values (n v)
            ((find "field"
                   (lambda (n+v)
                     (local [(define-values (n v) n+v)]
                       n)))
             name
             (map2 (lambda (f v)
                     (type-case Field f
                       [field (name) (values name v)]))
                   fields
                   vals)))]
    v))
```