



Type Systems

Lecture 6 Nov. 24th, 2004

Sebastian Maneth

<http://lampwww.epfl.ch/teaching/typeSystems/2004>



Today ... towards Featherweight JAVA

1. Objects
2. Simple Classes
3. Open Recursion through Self
4. Featherweight Java (FJ)

1. Objects

OBJECT = a data structure, encapsulating some **internal state**, and offering access to it via a collection of **methods**.

```
let x = ref 1 in  
  {get = λ_: Uni t. !x,  
   inc = λ_: Uni t. x := succ(!x)}
```

mutable instance variable

methods

1. Objects

OBJECT = a data structure, encapsulating some **internal state**, and offering access to it via a collection of **methods**.

```
c = let x = ref 1 in
    {get = λ_: Uni t. !x,
     inc = λ_: Uni t. x := succ(!x)}
```

mutable instance variable methods

The type of the record `c` is `{ get : Unit → Nat, inc : Unit → Unit }`

→ `c. inc uni t` evaluates to `uni t : Uni t`

→ `c. get uni t` evaluates to `2 : Nat`

1. Objects

Let `Counter` be the type `{ get : Unit → Nat, inc : Unit → Unit }`

`inc3` = `λc: Counter. (c.inc unit; c.inc unit; c.inc unit)`

Takes an argument of type `Counter` (“a counter object”) and applies three times its `inc` method.

Can `inc3` be applied to the following `ResetCounter` record?

```
let x = ref 1 in
  {get    = λ_: Unit. !x,
   inc    = λ_: Unit. x := succ(!x),
   reset  = λ_: Unit. x := 1}
```



1. Objects

Can you write a function that **generates** and returns a **new counter object**, each time it is called?

1. Objects

Can you write a function that **generates** and returns a **new counter object**, each time it is called?

Sure!

```
c = let x = ref 1 in
      {get = λ_: Uni t. !x,
       inc = λ_: Uni t. x := succ(!x)} : Counter
```

```
newCounter = λ_: Uni t. c : Uni t → Counter
```


1. Objects

Can you write a function that **generates** and returns a **new counter object**, each time it is called?

Sure!

```
c = let x = ref 1 in
      {get = λ_: Unit. !x,
       inc = λ_: Unit. x := succ(!x)} : Counter
```

```
newCounter = λ_: Unit. c : Unit → Counter
```

```
nc = newCounter unit
```

1. Objects

Group all instance variables into a record

```
c = let r = {x=ref 1} in
      {get = λ_: Unit. !(r.x),
       inc = λ_: Unit. r.x := succ(!(r.x))} : Counter
```

```
CounterRep = {x: Ref Nat}
```

The *representation type* of the object.



2. Simple Classes

`newCounter = λ_: Unit. c : Unit → Counter`

`newResetCounter = λ_: Unit. c : Unit → ResetCounter`

`ResetCounter <: Counter`

How can we define a `ResetCounter`, using the definition of `Counter`?

2. Simple Classes

```
let r = {x=ref 1} in
  {get = λ_: Unit. !(r.x),
   inc = λ_: Unit. r.x := succ(!(r.x))} : Counter
```

```
let r = {x=ref 1} in
  {get    = λ_: Unit. !(r.x),
   inc    = λ_: Unit. r.x := succ(!(r.x))
   reset  = λ_: Unit. r.x := 1} : ResetCounter
```

ResetCounter <: Counter

How can we define a ResetCounter, using the definition of Counter?

2. Simple Classes

should NOT be shared!

```
let r = {x=ref 1} in  
  {get = λ_: Uni t. !(r.x),  
   inc = λ_: Uni t. r.x := succ(!(r.x))} : Counter
```

```
let r = {x=ref 1} in  
  {get    = λ_: Uni t. !(r.x),  
   inc    = λ_: Uni t. r.x := succ(!(r.x))  
   reset  = λ_: Uni t. r.x := 1} : ResetCounter
```

Or rather, how to describe their **common functionality**?

2. Simple Classes

should NOT be shared!

```
let r = {x=ref 1} in
```

```
f(x) : Counter
```

```
let r = {x=ref 1} in
```

```
{get    = λ_: Uni t. !(r.x),  
inc    = λ_: Uni t. r.x := succ(!(r.x))  
reset  = λ_: Uni t. r.x := 1} : ResetCounter
```

Or rather, how to describe their **common functionality**?

NOT $f(x)$!

Cannot be in terms of x !

→ Use the object's representation type!

```
CounterRep = {x: Ref Nat}
```

2. Simple Classes

```
let r = {x=ref 1} in counterClass r : Counter
```

```
counterClass =
```

```
λr: CounterRep.
```

```
{get    = λ_: Unit. ! (r.x),  
inc    = λ_: Unit. r.x := succ(! (r.x))} : CounterRep → Counter
```

→ How to define `resetCounterClass` in terms of `counterClass`??

2. Simple Classes

```
let r = {x=ref 1} in counterClass r : Counter
```

```
counterClass =
```

```
λr: CounterRep.
```

```
{get    = λ_: Uni t. !(r.x),  
inc    = λ_: Uni t. x:=succ(!(r.x))} : CounterRep→Counter
```

→ How to define `resetCounterClass` in terms of `counterClass`??

```
resetCounterClass =
```

```
λr: CounterRep.
```

```
{get    = λ_: Uni t. !(r.x),  
inc    = λ_: Uni t. r.x:=succ(!(r.x))  
reset  = λ_: Uni t. r.x:=1} : CounterRep→ResetCounter
```


2. Simple Classes

```
let r = {x=ref 1} in counterClass r : Counter
```

```
counterClass =
```

```
λr: CounterRep.
```

```
{get    = λ_: Unit. ! (r.x),  
inc    = λ_: Unit. x:=succ(! (r.x))} : CounterRep→Counter
```

→ How to define `resetCounterClass` in terms of `counterClass`??

```
resetCounterClass =
```

```
λr: CounterRep. let super = counterClass r in
```

```
{get    = super.get,
```

```
inc    = super.inc,
```

```
reset  = λ_: Unit. r.x:=1} : CounterRep→ResetCounter
```



2. Simple Classes

resetCounterClass =

```
λr: CounterRep. let super = counterClass r in  
  {get    = super.get,  
   inc    = super.inc,  
   reset  = λ_: Unit. r.x := 1} : CounterRep → ResetCounter
```

Can we instantiate `resetCounterClass` with a
different record of instance variables?

E.g. `BackupCounterRep = {x: Ref Nat, b: Ref Nat}`

2. Simple Classes

```
resetCounterClass =  
  λr: CounterRep. let super = counterClass r in  
  {get    = super.get,  
   inc    = super.inc,  
   reset  = λ_: Unit. r.x := 1} : CounterRep → ResetCounter
```

Can we instantiate `resetCounterClass` with a
different record of instance variable??

E.g. `BackupCounterRep = {x: Ref Nat, b: Ref Nat}`

```
backupCounterClass =  
  λr: BackupCounterRep. let super = resetCounterClass r in  
  {get    = super.get,  
   inc    = super.inc,  
   reset  = λ_: Unit. r.x := ! (r.b),  
   backup = λ_: Unit. r.b := ! (r.x)} :  
  BackupCounterRep → BackupCounter
```

2. Simple Classes

What is a Class?

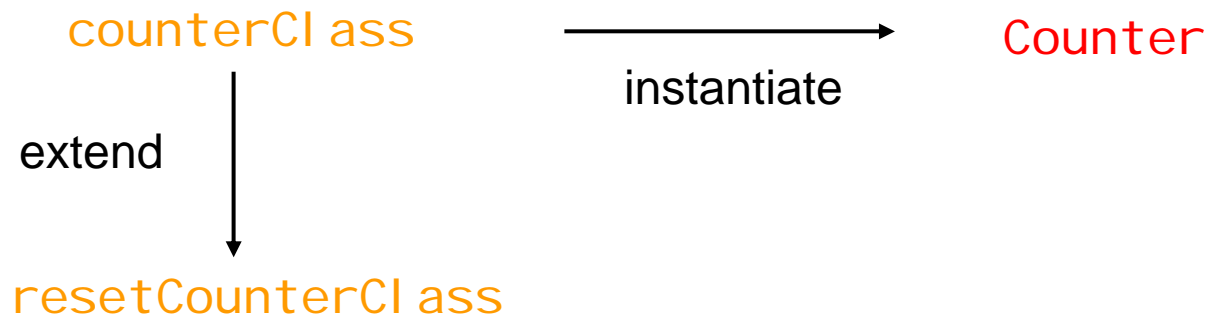
```
let r = {x=ref 1} in counterClass r : Counter
```

Class = collection of methods

obtained from an object by abstracting its methods w.r.t its instance variables.
(type: **CounterRep**→**Counter**)

An object can be obtained from a Class by *instantiating* it.

Only other use of a Class: *extending* (= “subtype”) it



2. Simple Classes

$\text{SetCounter} = \{\text{get}: \text{Unit} \rightarrow \text{Nat}, \text{set}: \text{Nat} \rightarrow \text{Unit}, \text{inc}: \text{Unit} \rightarrow \text{Unit}\}$

$\text{setCounterClass} =$
 $\lambda r: \text{CounterRep}.$

$\{\text{get} = \lambda_: \text{Unit}. ! (r.x),$
 $\text{set} = \lambda i: \text{Nat}. r.x := i,$
 $\text{inc} = \lambda_: \text{Unit}. \text{set}(\text{succ}(\text{get } \text{unit}))\} :$
 $\text{CounterRep} \rightarrow \text{SetCounter}$

not possible!!
 get / set come from SetCounter itself!

Recall the `fix` operator.

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_1}{\Gamma \vdash \text{fix } t_1 : T_1}$$

T_1 need not be a function type!! \rightarrow can be a record type!!

```
ff = λmutrec: {i seven: Nat → Bool , i sodd: Nat → Bool }.
      {i seven = λx: Nat.
          if iszero x then true
          else mutrec.i sodd (pred x),
       i sodd = λx: Nat.
          if isZero x then false
          else mutrec.i seven (pred x)}
```


`(fix ff).i seven 7` \rightarrow ... \rightarrow false

2. Simple Classes

$\text{SetCounter} = \{\text{get}: \text{Unit} \rightarrow \text{Nat}, \text{set}: \text{Nat} \rightarrow \text{Unit}, \text{inc}: \text{Unit} \rightarrow \text{Unit}\}$

$\text{setCounterClass} =$
 $\lambda r: \text{CounterRep}.$

$\{\text{get} = \lambda_: \text{Unit}. ! (r.x),$
 $\text{set} = \lambda i: \text{Nat}. r.x := i,$
 $\text{inc} = \lambda_: \text{Unit}. \text{set}(\text{succ}(\text{get } \text{unit}))\}$: $\text{CounterRep} \rightarrow \text{SetCounter}$



not possible!!
 get / set come from SetCounter itself!

2. Simple Classes

$\text{SetCounter} = \{\text{get}: \text{Unit} \rightarrow \text{Nat}, \text{set}: \text{Nat} \rightarrow \text{Unit}, \text{inc}: \text{Unit} \rightarrow \text{Unit}\}$

$\text{setCounterClass} =$

$\lambda r: \text{CounterRep}. \text{fix}$

$(\lambda \text{self}: \text{SetCounter}.$

$\{\text{get} = \lambda_: \text{Unit}. !(r.x),$

$\text{set} = \lambda i: \text{Nat}. r.x := i,$

$\text{inc} = \lambda_: \text{Unit}. \text{self.set}(\text{succ}(\text{self.get unit}))\}) :$

$\text{CounterRep} \rightarrow \text{SetCounter}$

get / set come from SetCounter itself!

$\text{newSetCounter} = \lambda_: \text{Unit}. \text{let } r = \{x = \text{ref } 1\} \text{ in } \text{setCounterClass } r$

3. Open Recursion through Self

```
setCounterClass =  
  λr: CounterRep. fix  
    (λself: SetCounter.  
      {get    = λ_: Unit. ! (r.x),  
       set    = λi: Nat.  r.x := i,  
       inc    = λ_: Unit. self.set (succ (self.get unit))}) :  
                                     CounterRep → SetCounter
```

```
newSetCounter = λ_: Unit. let r = {x = ref 1} in setCounterClass r
```

3. Open Recursion through Self

```
setCounterClass =  
  λr: CounterRep. fix  
  (λself: SetCounter.  
    {get    = λ_: Unit. ! (r.x),  
      set   = λi: Nat.  r.x := i,  
      inc   = λ_: Unit. self.set (succ (self.get unit))}) :  
    CounterRep → SetCounter
```

```
newSetCounter = λ_: Unit. let r = {x = ref 1} in setCounterClass r
```

3. Open Recursion through Self

```
setCounterClass =  
  λr: CounterRep. fix  
  (λsel f: SetCounter.  
    {get   = λ_: Unit. ! (r.x),  
     set   = λi: Nat.  r.x := i,  
     inc   = λ_: Unit. sel f.set (succ (sel f.get unit))}) :  
    CounterRep → SetCounter  
    CounterRep → SetCounter → SetCounter  
  
newSetCounter = λ_: Unit. let r={x=ref 1} in setCounterClass r  
    fix (setCounterClass r)
```

→ A “sel f-object” has to be supplied at instantiation time!!

3. Open Recursion through Self

```
setCounterClass =  
  λr: CounterRep.  
    (λsel f: SetCounter.  
      {get    = λ_: Unit. !(r.x),  
       set    = λi: Nat.  r.x:=i,  
       inc    = λ_: Unit. sel f.set(succ(sel f.get unit))}) :  
      CounterRep → SetCounter → SetCounter
```

NOW, methods of a *superclass* can call methods of a *subclass*,
even though the subclass does not exist when the superclass is being defined!!

3. Open Recursion through Self

```
setCounterClass =  
  λr: CounterRep.  
    (λself: SetCounter.  
      {get    = λ_: Unit. !(r.x),  
       set    = λi: Nat.  r.x:=i,  
       inc    = λ_: Unit. self.set(succ(self.get unit))}) :  
      CounterRep → SetCounter → SetCounter
```

NOW, methods of a *superclass* can call methods of a *subclass*,
even though the subclass does not exist when the superclass is being defined!!

```
accCounterClass =  
  λr: AccCounterRep.  
    λself: AccCounter. | let super = setCounterClass r self in  
      {get    = super.get,  
       set    = λi: Nat. (r.a:=succ(!(r.a)); super.set i),  
       inc    = super.inc,  
       acc    = λ_: Unit. !(r.a)} : AccCounterRep →  
      AccCounter → AccCounter
```

↓

3. Open Recursion through Self

Nice idea. BUT does NOT work like this!!

```
newAccCounterClass unit
```

```
→ let r = {x=ref 1, a=ref 0} in fix (accCounterClass r)
```

```
→ ... → < does NOT terminate!!! >
```

Why? Because occurrence of `self` is “unprotected”

```
accCounterClass =  
  λr: AccCounterRep.  
    λself: AccCounter. let super = setCounterClass r self in  
      {get = ...
```

Unprotected!
↓

3. Open Recursion through Self

Nice idea. BUT does NOT work like this!!

```
newAccCounterClass unit
```

```
→ let r = {x=ref 1, a=ref 0} in fix (accCounterClass r)
```

```
→ ... → < does NOT terminate!!! >
```

Why? Because occurrence of `self` is “unprotected”

- How to fix this??
1. protect the `self` by λ -abstraction
 2. model semantics of classes differently (using refs)
 3. take objects/classes as **NEW PRIMITIVES**

```
accCounterClass =  
  λr: AccCounterRep.  
    λself: AccCounter. let super = setCounterClass r self in  
    {get = ...
```

Unprotected!
↓



4. Featherweight Java

Take objects/classes as **NEW PRIMITIVES**

In fact, the language FJ consist **only** of these primitives.
“everything is an object!”

→ Almost as pure as the lambda-calculus

(and almost as degenerate as simply typed lambda calculus wo. base types.)



4. Featherweight Java

Take objects/classes as **NEW PRIMITIVES**

```
let r = {x=ref 1} in counterClass r : Counter
```

Class = collection of methods (w.r.t. instance variables)

→ can be extended, or instantiated.

4. Featherweight Java

Take objects/classes as **NEW PRIMITIVES**

```
let r = {x=ref 1} in counterClass r : Counter
```

Class = collection of methods (w.r.t. instance variables)

→ ~~can be extended~~, or instantiated.
must extend another Class

Every class (transitively)
extends the class **Object**

```
Class resetCounter extends Counter {  
    Nat b;  
  
    resetCounter(Nat x, Nat b){  
        super(x), this.b=b;  
    }  
    Nat reset( ){ return new Nat(1); }  
}
```

4. Featherweight Java

Take objects/classes as **NEW PRIMITIVES**

```
let r = {x=ref 1} in counterClass r : Counter
```

Class = collection of methods (w.r.t. instance variables)

→ ~~can be extended~~, or instantiated.
must extend another Class

Every class (transitively)
extends the class **Object**

```
Class resetCounter extends Counter {  
  new fields    Nat b;  
  all instance variables  
  resetCounter(Nat x, Nat b){  
    super(x), this.b=b;  
  } constructor  
  new methods  Nat reset( ){ return new Nat(1); }  
}
```

4. Featherweight Java

Take objects/classes as **NEW PRIMITIVES**

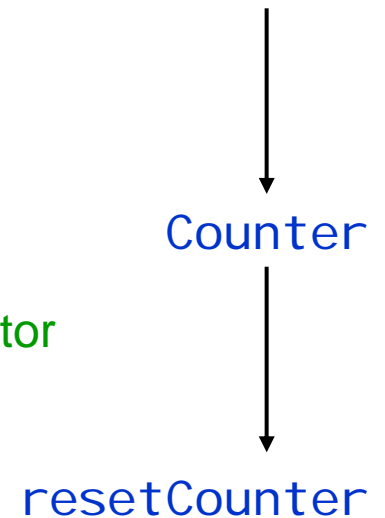
```
let r = {x=ref 1} in counterClass r : Counter
```

Class = collection of methods (w.r.t. instance variables)

→ ~~can be extended~~, or instantiated.
must extend another Class

Every class (transitively)
extends the class **Object**

```
Class resetCounter extends Counter {  
  new fields Nat b;  
    
    
  resetCounter(Nat x, Nat b){  
    super(x), this.b=b;  
  }  
  new methods Nat reset() { return new Nat(1); }  
}
```



4. Featherweight Java: terms

How do we instantiate a class?

`new Counter(t1, ..., tn)` Object Creation (similar to λ -abstraction)

`t.m(t1, ..., tn)` Method Invocation (similar to function application)

`x` Variables (same as in λ -calculus)

`t.f` Field Access (like in records..)

`(sClass) t` Cast (like subsumption)

Object Creations are the only values! (like in pure λ)

E.g. `new A(), new B(), new A(new B(), new C()), ...`



FJ Program = collection of **Class Declarations** plus a **Term** to be evaluated

```
Class A extends Object { A(){super();} }
Class B extends Object { B(){super();} }
Class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snc=snd; }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd); }
}
```

```
new Pair(new A(), new B()).setfst(new B())
```



FJ Program = collection of **Class Declarations** plus a **Term** to be evaluated

```
Class A extends Object { A(){super();} }
Class B extends Object { B(){super();} }
Class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snc=snd; }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd); }
}
```

`new Pair(new A(), new B()).setfst(new B())`

→ `new Pair(new B(), new B())`

FJ Program = collection of **Class Declarations** plus a **Term** to be evaluated

```
Class A extends Object { A(){super();} }
Class B extends Object { B(){super();} }
Class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snc=snd; }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd); }
}
```

new Pair(new A(), new B()).**setfst**(new B())

→ new Pair(new B(), new B())

Method Invocation: $\text{mbody}(\text{setfst}, \text{Pair}) = (\text{newfst}, \text{new Pair}(\text{newfst}, \text{this.snd}))$

$(\text{new C}(v_1, v_2)).\text{setfst}(u_1) \rightarrow [\text{newfst} \rightarrow u_1, \text{this} \rightarrow \text{new C}(v_1, v_2)] t_0$



FJ Program = collection of **Class Declarations** plus a **Term** to be evaluated

```
Class A extends Object { A(){super();} }
Class B extends Object { B(){super();} }
Class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snc=snd; }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd); }
}
```

((Pair) new Pair(new Pair(new A(), new B()), new A()).fst).snd

→

FJ Program = collection of **Class Declarations** plus a **Term** to be evaluated

```
Class A extends Object { A(){super();} }
```

```
Class B extends Object { B(){super();} }
```

```
Class Pair extends Object {
```

```
  Object fst;
```

```
  Object snd;
```

```
  Pair(Object fst, Object snd) {  
    super(); this.fst=fst; this.snd=snd; }
```

```
  Pair setfst(Object newfst) {  
    return new Pair(newfst, this.snd); }
```

```
}
```

((Pair) new Pair(new Pair(new A(), new B()), new A()).fst).snd

→ ((Pair) new Pair(new A(), new B())).snd

Field Selection: **fst** is declared to contain an **Object**

Thus, the return **new Pair(new A(), new B())** is now an Object!

FJ Program = collection of **Class Declarations** plus a **Term** to be evaluated

```
Class A extends Object { A(){super();} }
```

```
Class B extends Object { B(){super();} }
```

```
Class Pair extends Object {
```

```
  Object fst;
```

```
  Object snd;
```

```
  Pair(Object fst, Object snd) {  
    super(); this.fst=fst; this.snc=snd; }
```

```
  Pair setfst(Object newfst) {  
    return new Pair(newfst, this.snd); }
```

```
}
```

((Pair) new Pair(new Pair(new A(), new B()), new A()).fst).snd

→ ((Pair) new Pair(new A(), new B())).snd

Cast is needed, because `mbody(snd, Object)` is not defined!!



FJ Program = collection of **Class Declarations** plus a **Term** to be evaluated

```
Class A extends Object { A(){super();} }
Class B extends Object { B(){super();} }
Class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snc=snd; }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd); }
}
```

((Pair) new Pair(new Pair(new A(), new B()), new A()).fst).snd

→ ((Pair)new Pair(new A(), new B())).snd

→ new Pair(new A(), new B()).snd

→ new B()

FJ Program = collection of **Class Declarations** plus a **Term** to be evaluated

```
Class A extends Object { A(){super();} }
Class B extends Object { B(){super();} }
Class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snc=snd; }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd); }
}
```

((Pair) new Pair(new Pair(new A(), new B()), new A()).fst).snd

→ ((Pair)new Pair(new A(), new B())).snd

→ new Pair(new A(), new B()).snd

→ new B()

Syntactically you canNOT see the type **Object**

4. Featherweight Java

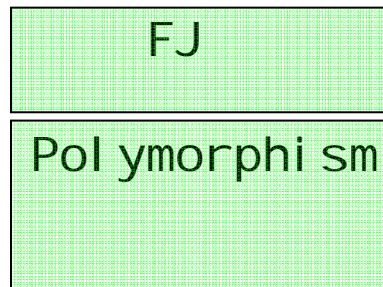
→ Casts only make sense with run-time type look-up.

Today's LAB:

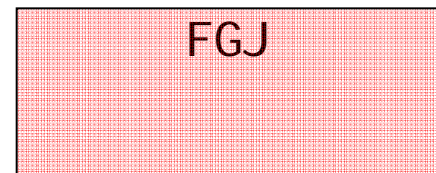
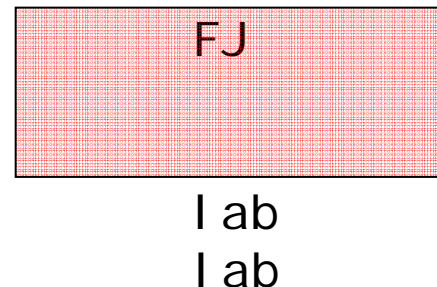
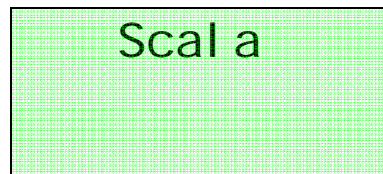
→ start implementing evaluation of FJ terms

→ to do that, simply ignore cases (by evaluating them away)

today
1. 12.
8. 12.
15. 12.
22. 12.



12. 1.
19. 1.
26. 1.
2. 2.



Programming
Assignments
To be turned in!

