Operational semantics

- Reduction semantics
- Abstract machine semantics.

Reduction semantics

For functional programs,
with no computational effects,
program execution can be
modelled by just evaluation.

β-reduction (function expansion)

$$(\lambda x.\ M)\ N \underset{\beta}{\rightarrow} M[N/x].$$

δ- reductions (primitive functions)

if true M N $\rightarrow$ M

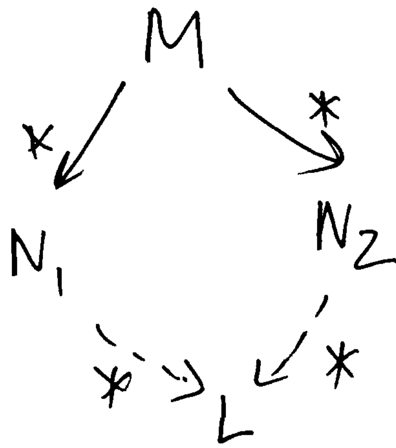
if false M N $\rightarrow$ N

car (cons M N) $\rightarrow$ M

$\vdots$

Church-Rosser property

A reduction semantics is Church-Rosser or confluent if



If M can ~~red~~ reduce to distinct N_1, N_2 then N_1 and N_2 reduce to a common term.

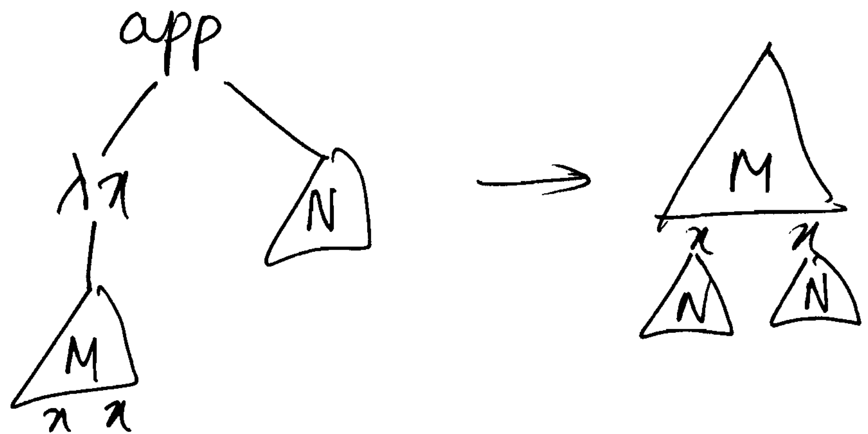
Consequence:

The results are independent of the reduction strategy.

(The choice of subterms reduced)

However, the reduction strategy makes a difference for termination.

Outermost reduction



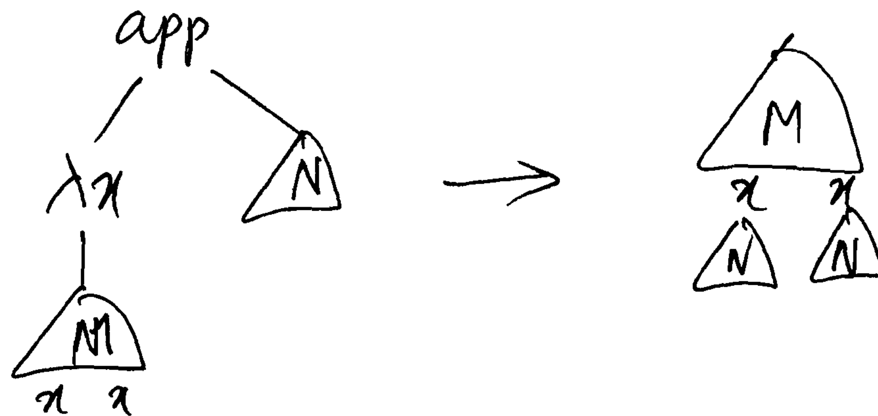
The arguments are not evaluated before calling the functions.

Pro – If the argument value is not needed, we save computation.

Con – If the argument ~~val~~ is used in multiple places, we may duplicate computation.

Call-by-name programming languages use outermost reduction. They allow the use of <u>infinite data structures</u>.

Innermost reduction

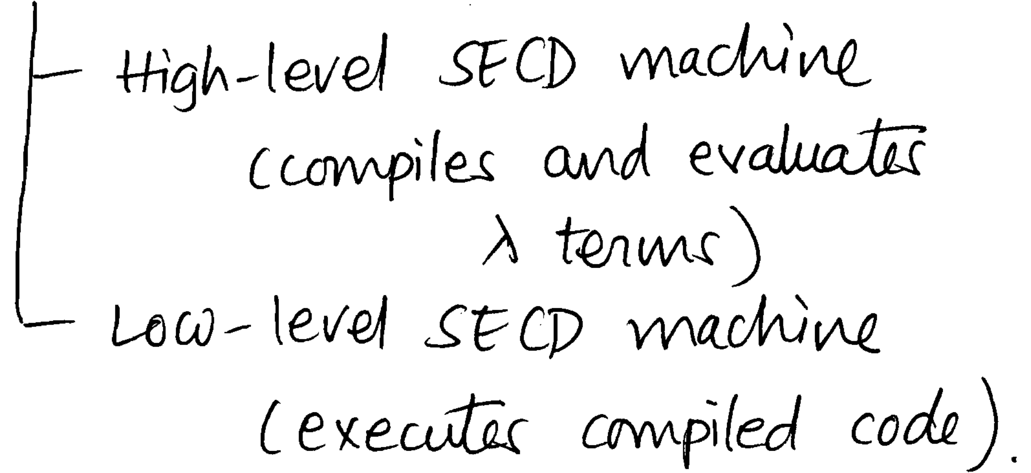


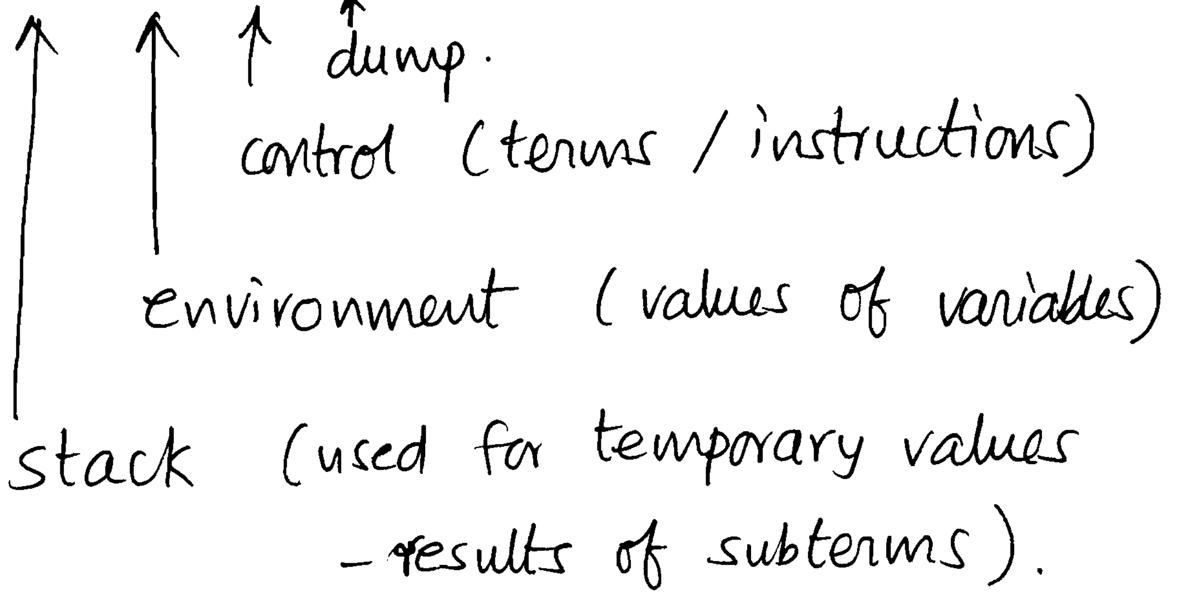
Both M and N should be in ~~normal fo~~ normal form.

This is the mechanism used in call-by-value programming languages.

Infinite data structures are not available in call-by-value languages.

Abstract Machine Semantics





Initial configuration:

([ ], [ ], [M], [ ]).

↑ term to be evaluated.

Example rules: (constant)

$$(S, E, [c \mid C], D) \rightarrow ([S \mid c], E, C, D).$$

(variable)

$$(S, E, [x \mid C], D) \rightarrow ([S \mid lookup(E, x)], E, C, D)$$

$$(S, E, [(M\ N) \mid C], D) \rightarrow (S, E, [M, N, app \mid C], D)$$

translates M N to a sequence of instructions.

Low-level SECD machine

Does not do any translation.

A compiler is used to translate terms to instructions ahead of time.

Closures

The SECD machine does not do variable substitution. (unlike β-reduction).

The values of variables are stored in the environment, and looked up when needed.

$$(S, E, [\lambda x.M \mid C], D).$$

($\lambda x.M$: abstraction term)

- The values of the free variables of M are currently in E.
- when the function is called, we need to use this environment for the values of free variables.

→ $([S \mid cl(\lambda x.M, E)], E, C, D)$.

($cl(\lambda x.M, E)$ ↑ closure that packages a function term and environment)

Using the closure:

$$([cl(\lambda x.M, E_0), N \mid S], E, [app \mid C], D).$$

(annotations: $cl(\lambda x.M, E_0)$ ↑ function closure; N ↑ argument.)

- We should "call" the function, but change the environment to E_0;
- The current environment is saved on the dump.

$$\to (S, upd(E_0, x, N), [M, ret \mid C], [D \mid E]).$$

(annotations: $upd(E_0, x, N)$: new environment for the function call; E ↑ saved environment.)

The return instruction restores the old environment.

Compilation issues

(Handout 7, Abelson & Sussman).

The environment is structured as a sequence of <u>frames</u>, where each frame contains the variable bindings from a single call.

$$E = F_0 \leftarrow F_1 \leftarrow F_2 \leftarrow \cdots \leftarrow F_n .$$

F_0: the "global" frame

$F_1 \ldots F_n$: each frame comes from a function call.

Example:

let $d = \lambda(n,k).\ =(\mathrm{mod}(n,k), 0)$

in **let** $e = \lambda(k).\ d(k,2)$

in $e(3)$

$F_0 = \{\}$

~~$F_1 = \{d \mapsto \lambda(n,k). \cdots, F$~~

$F_1 = \{d \mapsto cl(\lambda(n,k), \cdots, F_0)\}$

— from calling the let.

$F_2 = \{e \mapsto cl(\lambda(k).\ d(k,2),\ F_0 \leftarrow F_1\}$

— from calling the let.

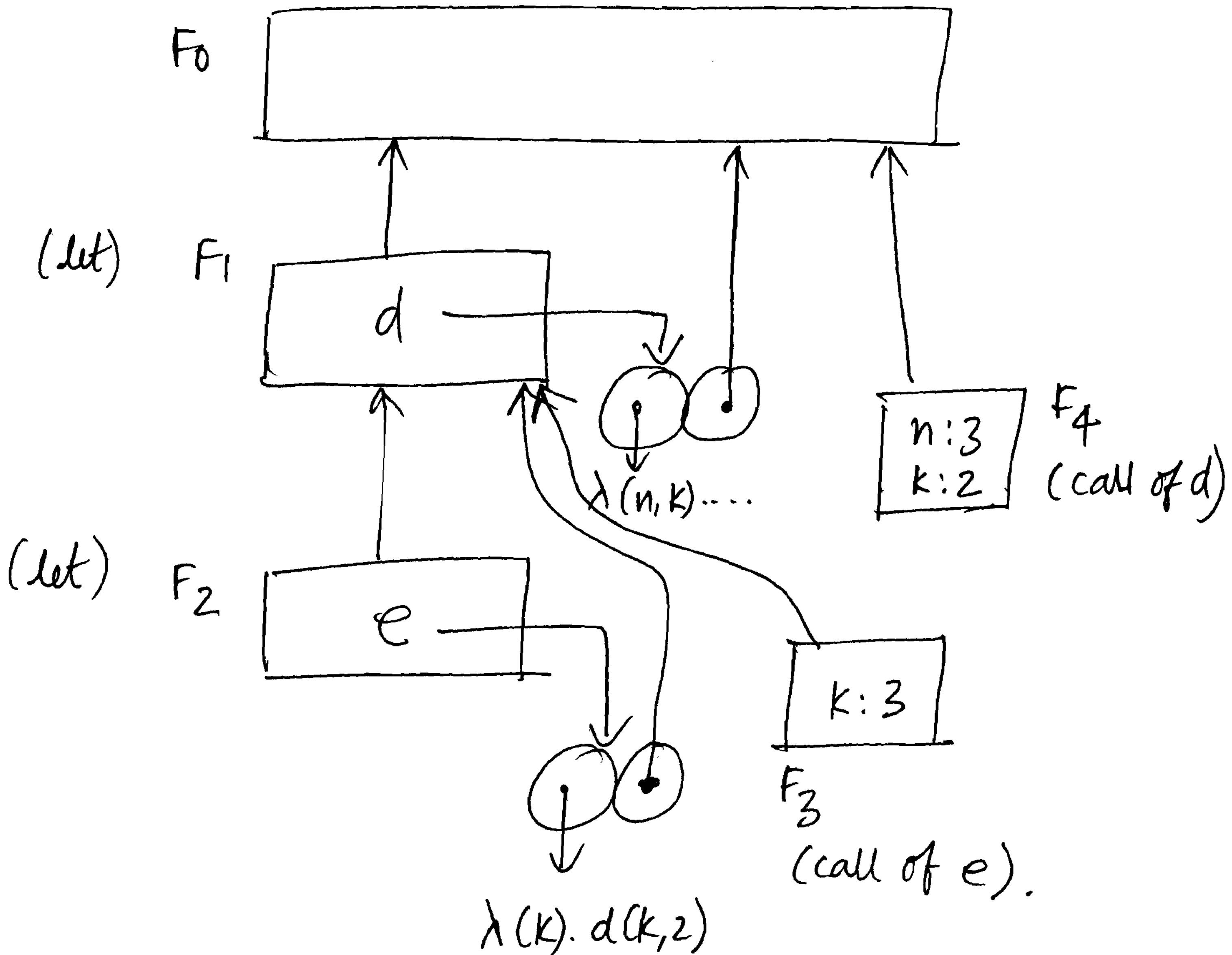
$F_3 = \{k \mapsto 3\}$ ~~$F_0 \leftarrow F_1 \leftarrow F_2\}$~~

— from calling e.

$F_4 = \{n \mapsto 3,\ k \mapsto 2\}$

— from calling d.

Diagrammatic form for environments



Assignments & Commands

A call-by-value λ calculus can be extended with "computational effects"

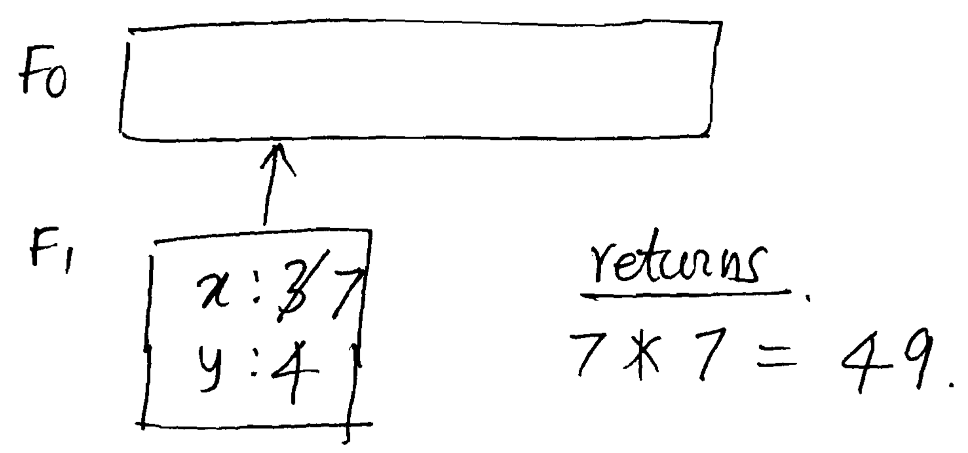
Scheme-style assignments

$$M ::= c \mid x \mid \lambda x.\, M' \mid M_1\, M_2 \mid$$
$$\underbrace{x := M'}_{\text{assignment}} \mid \underbrace{M_1 \,;\, M_2}_{\text{sequencing}}.$$

Example

<u>let</u> $p = \lambda(x,y).\; x := x+y \;;\; x * x$

<u>in</u> $p\,(3,4)$

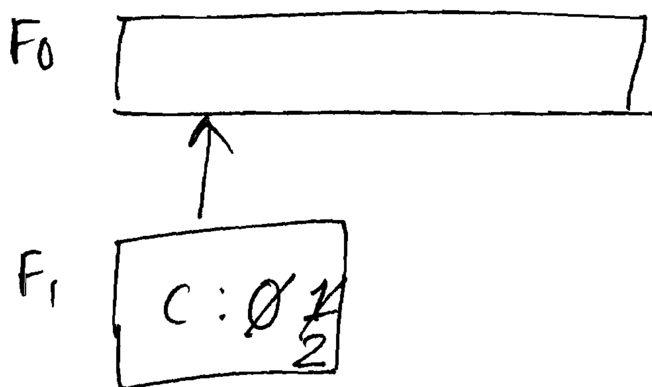


Objects

Function closures that refer to state variables.

Example

$$\text{counter} \equiv \underline{\text{let}}\ c = 0$$
$$\underline{\text{in}}\ \lambda().\ (c := c+1;\ c).$$



counter() $\rightsquigarrow$ $\underline{1}$

but also changes c to 1.

counter() $\rightsquigarrow$ 2

and changes c to 2.

Records

(also called "structs")

- <u>struct</u> $\{ x_1 = M_1 ; \ldots ; x_n = M_n \}$.

  builds a record with fields $x_1, \ldots, x_n$.

- $M.x$

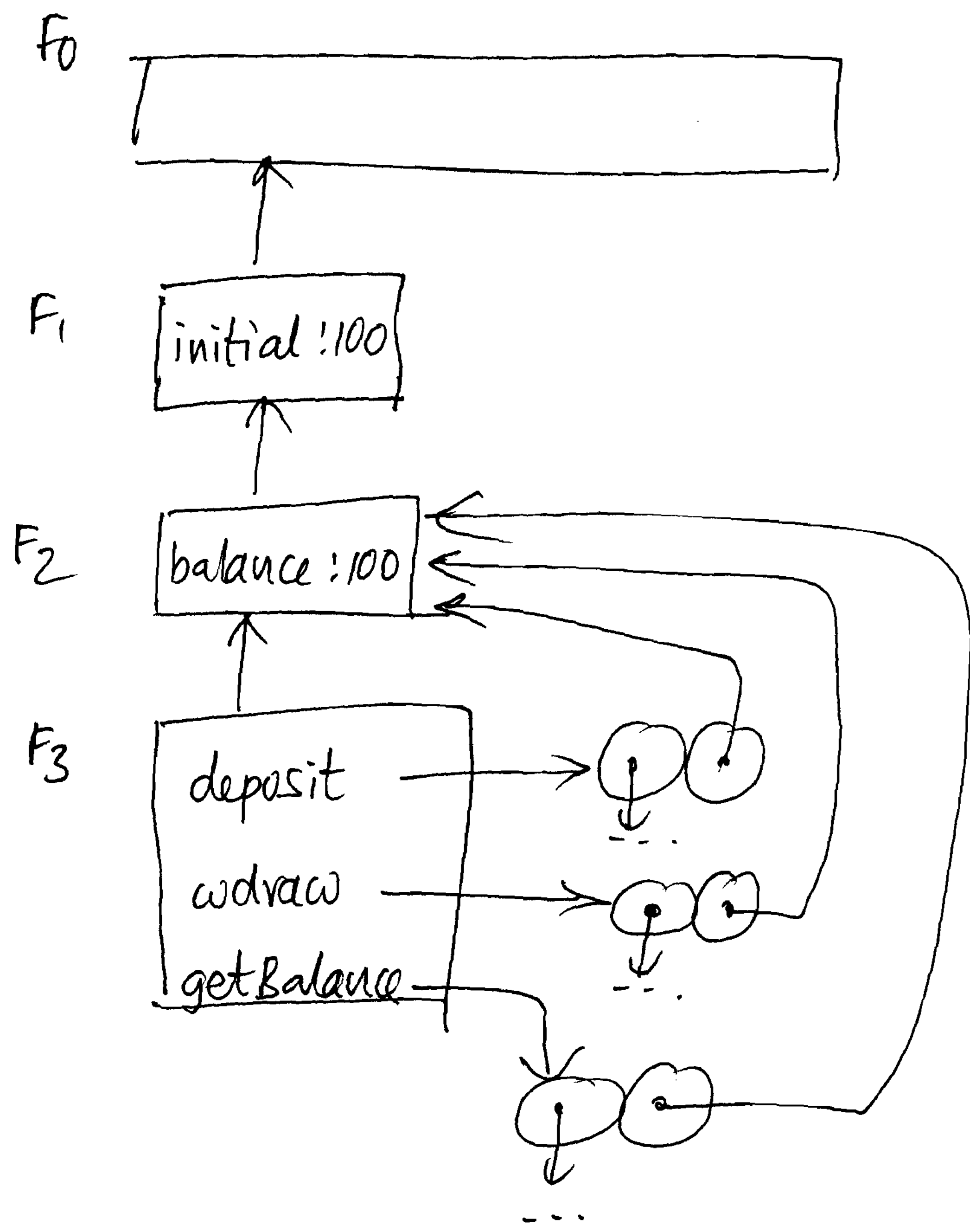
  selects the field x of M.

We can use records to build objects with multiple methods.

Example

```
newaccount =
    λ(initial).
        let balance = initial
        in struct {
            deposit = λ(amount).
                        balance := balance +
                                   amount;
            wdraw = λ(amount).
                        .... ;
            getBalance = λ(). balance
        }.
```

Environment structure



Every call to newaccount produces a new instance of three frames (F_1, F_2, F_3) so that each bank account has its own local variable for balance.

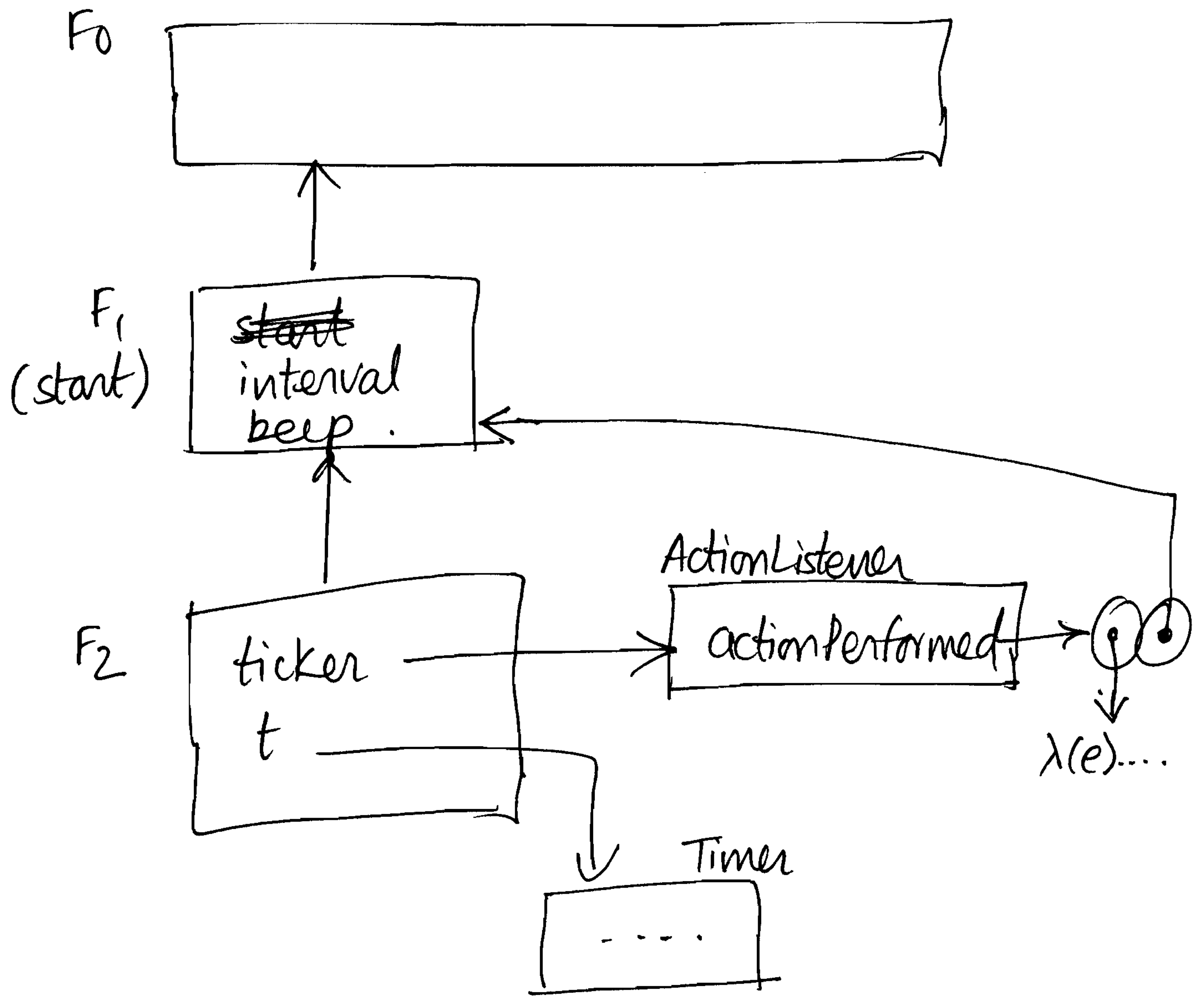
Java classes

A Java class is like a record, ~~whose fields~~ whose fields include
- all the static methods, and
- the new method.

Inner classes give rise to cascaded environment frames.

Example:

```
static void start (int interval, boolean beep) {
    ActionListener ticker =
        new ActionListener () {
            public void actionPerformed (
                              ActionEvent e) {
                   ...
            }
        }
    Timer t = new Timer (interval, ticker);
    t.start () ;
}
```


F0
F1
(start)
~~start~~
interval
beep.
ActionListener
actionPerformed
F2
ticker
t
λ(e)....
Timer
....