Compiler Construction
Lent Term 2016

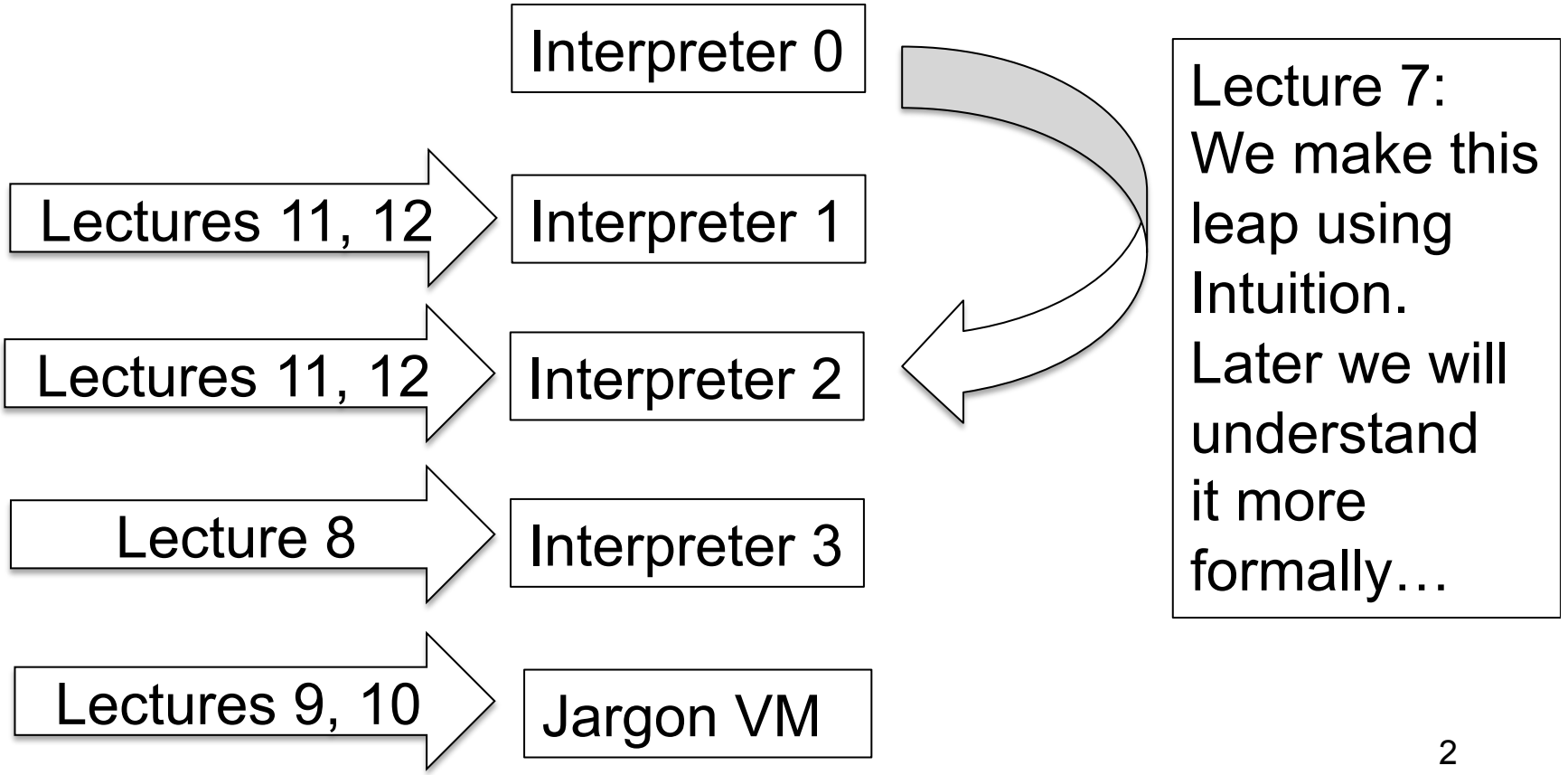
Part II : Lectures 7 – 12 (of 16)

Timothy G. Griffin
tgg22@cam.ac.uk

Computer Laboratory
University of Cambridge

Roadmap

Starting from a direct implementation of Slang/L3 semantics, we will **DERIVE** a Virtual Machine in a step-by-step manner. The correctness of each step is (more or less) easy to check.



LECTURE 7
Interpreter 0, Interpreter 2

1. Interpreter 0 : The high-level "definitional" interpreter
   1. Slang/L3 values represented directly as OCaml values
   2. Recursive interpreter implements a denotational semantics
   3. The interpreter implicitly uses OCaml's runtime stack
2. Interpreter 2: A high-level stack-oriented machine
   1. Makes the Ocaml runtime stack explicit
   2. Complex values pushed onto stacks
   3. One stack for values and environments
   4. One stack for instructions
   5. Heap used only for references
   6. Instructions have tree-like structure

Approaches to Mathematical Semantics

- Axiomatic: Meaning defined through logical specifications of behaviour.
  - Hoare Logic (Part II)
  - Separation Logic
- Operational: Meaning defined in terms of transition relations on states in an abstract machine.
  - Semantics (Part 1B)
- Denotational: Meaning is defined in terms of mathematical objects such as functions.
  - Denotational Semantics (Part II)

A denotational semantics for L3?

N = set of integers **B** = set of booleans **A** = set of addresses

I = set of identifiers Expr = set of L3 expressions

E = set of environments = **A** → **V** **S** = set of stores = **A** → **V**

V = set of value
≈ **A**
+ **N**
+ **B**
+ { () }
+ **V** × **V**
+ (**V** + **V**)
+ (**V** × **S**) → (**V** × **S**)

Set of values **V** solves this "domain equation" (here + means disjoint union).

Solving such equations is where some difficult maths is required …

M = the meaning function

M : (Expr × **E** × **S**) → (**V** × **S**)

Not examinable!!

Our shabby OCaml approximation

A = set of addresses

S = set of stores = **A** → **V**

V = set of value

≈ **A**

+ **N**

+ **B**

+ { () }

+ **V** × **V**

+ (**V** + **V**)

+ (**V** × **S**) → (**V** × **S**)

E = set of environments = **A** → **V**

M = the meaning function

M : (Expr × **E** × **S**) → (**V** × **S**)

```
type address

type store = address -> value

and value =
    | REF of address
    | INT of int
    | BOOL of bool
    | UNIT
    | PAIR of value * value
    | INL of value
    | INR of value
    | FUN of ((value * store)
                    -> (value * store))

type env = Ast.var -> value

val interpret :
    Ast.expr * env * store
                    -> (value * store)
```

Most of the code is obvious!

```
let rec interpret (e, env, store) =
   match e with
   | If(e1, e2, e3) ->
     let (v, store') = interpret(e1, env, store) in
          (match v with
           | BOOL true -> interpret(e2, env, store')
           | BOOL false -> interpret(e3, env, store')
           | v -> complain "runtime error.  Expecting a boolean!")
   | Pair(e1, e2)  ->
     let (v1, store1) = interpret(e1, env, store) in
     let (v2, store2) = interpret(e2, env, store1) in (PAIR(v1, v2), store2)
   | Fst e ->
      (match interpret(e, env, store) with
       | (PAIR (v1, _), store') -> (v1, store')
       | (v, _) -> complain "runtime error.  Expecting a pair!")
   | Snd e  ->
     (match interpret(e, env, store) with
       | (PAIR (_, v2), store') -> (v2, store')
       | (v, _) -> complain "runtime error.  Expecting a pair!")
   | Inl e   -> let (v, store') = interpret(e, env, store) in (INL v, store')
   | Inr e  -> let (v, store') = interpret(e, env, store) in (INR v, store')
   :
   .
```

Tricky bits : Slang functions mapped to OCaml functions!

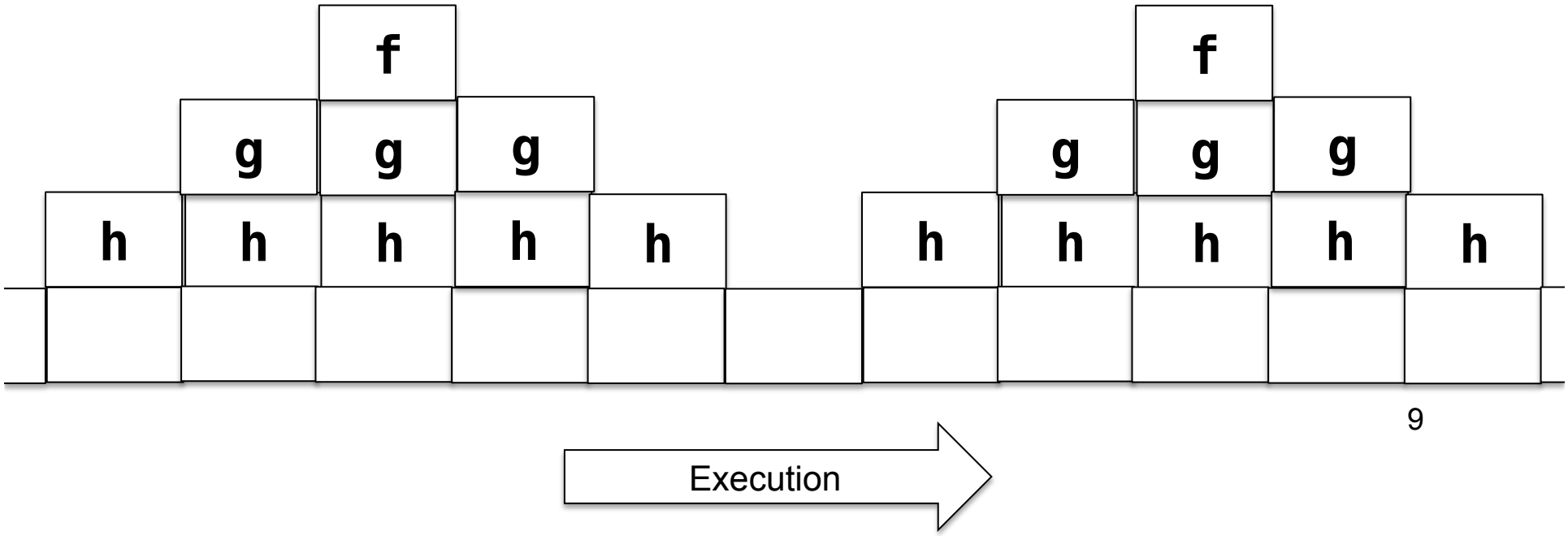
```
let rec interpret (e, env, store) =
    match e with
    :
    :
    | Lambda(x, e)  -> (FUN (fun (v, s) -> interpret(e, update(env, (x, v)), s)), store)
    | App(e1, e2) -> (* I chose to evaluate argument first!  *)
      let (v2, store1) = interpret(e2, env, store) in
      let (v1, store2) =  interpret(e1, env, store1) in
          (match v1 with
          | FUN f -> f (v2, store2)
          | v -> complain "runtime error.  Expecting a function!")
    | LetFun(f, (x, body), e) ->
      let new_env =
           update(env, (f, FUN (fun (v, s) -> interpret(body, update(env, (x, v)), s))))
      in interpret(e, new_env, store)
    | LetRecFun(f, (x, body), e) ->
      let rec new_env g = (* a recursive environment!!! *)
          if g = f then FUN (fun (v, s) -> interpret(body, update(new_env, (x, v)), s))
                   else env g
      in interpret(e, new_env, store)
```

update : env * (var * value) -> env

Typical implementation of function calls

```
let fun f (x) = x + 1
    fun g(y) = f(y+2)+2
    fun h(w) = g(w+1)+3
in
   h(h(17))
end
```

The run-time data structure is the call stack containing an activation record for each function invocation.



interpret **is implicitly using Ocaml's runtime stack**

```
let rec interpret (e, env, store) =
   match e with
   | Integer n          -> (INT n, store)
   | Op(e1, op, e2)   ->
     let (v1, store1) = interpret(e1, env, store) in
     let (v2, store2) = interpret(e2, env, store1) in
         (do_oper(op, v1, v2), store2)
   :
   :
```

- Every invocation of interpret is building an activation record on Ocaml's runtime stack.
- **We will know define interpreter 2 which makes this stack explicit**

Inpterp_2 data types

```
type address

type store = address -> value

and value =
    | REF of address
    | INT of int
    | BOOL of bool
    | UNIT
    | PAIR of value * value
    | INL of value
    | INR of value
    | FUN of ((value * store)
                    -> (value * store))

type env = Ast.var -> value
```

Interp_0

```
type address = int

type value =
    | REF of address
    | INT of int
    | BOOL of bool
    | UNIT
    | PAIR of value * value
    | INL of value
    | INR of value
    | CLOSURE of bool *
                    closure

and closure = code * env
```

Interp_2

```
and instruction =
  | PUSH of value
  | LOOKUP of var
  | UNARY of unary_oper
  | OPER of oper
  | ASSIGN
  | SWAP
  | POP
  | BIND of var
  | FST
  | SND
  | DEREF
  | APPLY
  | MK_PAIR
  | MK_INL
  | MK_INR
  | MK_REF
  | MK_CLOSURE of code
  | MK_REC of var * code
  | TEST of code * code
  | CASE of code * code
  | WHILE of code * code
```

Interp_2.ml : The Abstract Machine

```
and code = instruction list

and binding = var * value

and env = binding list

type env_or_value = EV of env | V of value

type env_value_stack = env_or_value list

type state = code * env_value_stack

val step : state -> state

val driver : state -> value

val compile : expr -> code

val interpret : expr -> value
```

The state is actually comprised of a heap --- a global array of values --- a pair of the form

(code, evn_value_stack)

Interpreter 2: The Abstract Machine

```
type state = code * env_value_stack

val step : state -> state
```

The state transition function.

```
let step = function
(* (code stack,              value/env stack) -> (code stack,  value/env stack) *)
  | ((PUSH v) :: ds,                          evs) -> (ds, (V v) :: evs)
  | (POP :: ds,                          s :: evs) -> (ds, evs)
  | (SWAP :: ds,                s1 :: s2 :: evs) -> (ds, s2 :: s1 :: evs)
  | ((BIND x) :: ds,               (V v) :: evs) -> (ds, EV([(x, v)]) :: evs)
  | ((LOOKUP x) :: ds,                        evs) -> (ds, V(search(evs, x)) :: evs)
  | ((UNARY op) :: ds,             (V v) :: evs) -> (ds, V(do_unary(op, v)) :: evs)
  | ((OPER op) :: ds,   (V v2) :: (V v1) :: evs) -> (ds, V(do_oper(op, v1, v2)) :: evs)
  | (MK_PAIR :: ds,     (V v2) :: (V v1) :: evs) -> (ds, V(PAIR(v1, v2)) :: evs)
  | (FST :: ds,           V(PAIR (v, _)) :: evs) -> (ds, (V v) :: evs)
  | (SND :: ds,           V(PAIR (_, v)) :: evs) -> (ds, (V v) :: evs)
  | (MK_INL :: ds,             (V v) :: evs) -> (ds, V(INL v) :: evs)
  | (MK_INR :: ds,             (V v) :: evs) -> (ds, V(INR v) :: evs)
  | (CASE (c1, _) :: ds,        V(INL v)::evs) -> (c1 @ ds, (V v) :: evs)
  | (CASE (_, c2) :: ds,        V(INR v)::evs) -> (c2 @ ds, (V v) :: evs)
  | ((TEST(c1, c2)) :: ds,  V(BOOL true) :: evs) -> (c1 @ ds, evs)
  | ((TEST(c1, c2)) :: ds, V(BOOL false) :: evs) -> (c2 @ ds, evs)
  | (ASSIGN :: ds,  (V v) :: (V (REF a)) :: evs) -> (heap.(a) <- v; (ds, V(UNIT) :: evs))
  | (DEREF :: ds,           (V (REF a)) :: evs) -> (ds, V(heap.(a)) :: evs)
  | (MK_REF :: ds,             (V v) :: evs) -> let a = allocate () in (heap.(a) <- v;
                                                  (ds, V(REF a) :: evs))
  | ((WHILE(c1, c2)) :: ds,V(BOOL false) :: evs) -> (ds, evs)
  | ((WHILE(c1, c2)) :: ds, V(BOOL true) :: evs) -> (c1 @ [WHILE(c1, c2)] @ ds, evs)
  | ((MK_CLOSURE c) :: ds,                    evs) -> (ds,  V(mk_fun(c, evs_to_env evs)) :: evs)
  | (MK_REC(f, c) :: ds,                      evs) -> (ds,  V(mk_rec(f, c, evs_to_env evs)) :: evs)
  | (APPLY :: ds,  V(CLOSURE (_, (c, env))) :: (V v) :: evs)
                                                  -> (c @ ds, (V v) :: (EV env) :: evs)
  | state -> complain ("step : bad state = " ^ (string_of_state state) ^ "\n")
```

The driver. Correctness

```
(* val driver : state -> value *)
let rec driver state =
    match state with
    | ([], [V v]) -> v
    | _           -> driver (step state)
```

```
val compile : expr -> code
```

The idea: if e passes the frond-end and

Interp_0.interpret e = v

then

driver (compile e, []) = v'

where v' (somehow) represents v.

In other words, evaluating

compile e

should leave the value of e on top of the stack

Implement inter_0 in interp_2

interp_0.ml

```
let rec interpret (e, env, store) =
   match e with
| Pair(e1, e2)  ->
     let (v1, store1) = interpret(e1, env, store) in
     let (v2, store2) = interpret(e2, env, store1) in (PAIR(v1, v2), store2)
   | Fst e ->
      (match interpret(e, env, store) with
      | (PAIR (v1, _), store') -> (v1, store')
      | (v, _) -> complain "runtime error.  Expecting a pair!")
   :
```

interp_2.ml

```
let step = function
 | (MK_PAIR :: ds,  (V v2) :: (V v1) :: evs)  ->  (ds,   V(PAIR(v1, v2)) :: evs)
 | (FST :: ds,          V(PAIR (v, _)) :: evs)  ->  (ds,   (V v) :: evs)
 :

let rec compile = function
 | Pair(e1, e2)   -> (compile e1) @ (compile e2) @ [MK_PAIR]
 | Fst e          -> (compile e) @ [FST]
 :
```

Implement inter_0 in interp_2

interp_0.ml

```
let rec interpret (e, env, store) =
   match e with
    | If(e1, e2, e3) ->
      let (v, store') = interpret(e1, env, store) in
           (match v with
           | BOOL true -> interpret(e2, env, store')
           | BOOL false -> interpret(e3, env, store')
           | v -> complain "runtime error.  Expecting a boolean!")
    :
```

interp_2.ml

```
let step = function
 | ((TEST(c1, c2)) :: ds,  V(BOOL true) :: evs) -> (c1 @ ds, evs)
 | ((TEST(c1, c2)) :: ds, V(BOOL false) :: evs) -> (c2 @ ds, evs)
 :

let rec compile = function
 | If(e1, e2, e3) -> (compile e1) @ [TEST(compile e2, compile e3)]
 :
```

Tricky bits again!

interp_0.ml

```
let rec interpret (e, env, store) =
    match e with
    | Lambda(x, e)  -> (FUN (fun (v, s) -> interpret(e, update(env, (x, v)), s)), store)
    | App(e1, e2) -> (* I chose to evaluate argument first!  *)
      let (v2, store1) = interpret(e2, env, store) in
      let (v1, store2) =  interpret(e1, env, store1) in
          (match v1 with
          | FUN f -> f (v2, store2)
          | v -> complain "runtime error.  Expecting a function!")
    :
```

interp_2.ml

```
let step = function
 | (POP :: ds,                          s :: evs) -> (ds,  evs)
 | (SWAP :: ds,                 s1 :: s2 :: evs) -> (ds,  s2 :: s1 :: evs)
 | ((BIND x) :: ds,                (V v) :: evs) -> (ds,  EV([(x, v)]) :: evs)
 | ((MK_CLOSURE c) :: ds,                evs) -> (ds,   V(mk_fun(c, evs_to_env evs)) :: evs)
 | (APPLY :: ds,  V(CLOSURE (_, (c, env))) :: (V v) :: evs)
                                          -> (c @ ds,  (V v) :: (EV env) :: evs)
let rec compile = function
 | Lambda(x, e)   -> [MK_CLOSURE((BIND x) :: (compile e) @ [SWAP; POP])]
 | App(e1, e2)      -> (compile e2) @ (compile e1) @ [APPLY; SWAP; POP]
 :
```

Example : Compiled code for rev_pair.slang

```
let rev_pair (p : int * int) : int * int  = (snd p, fst p)
in
    rev_pair (21, 17)
end
```

```
MK_CLOSURE([BIND p; LOOKUP p; SND; LOOKUP p; FST; MK_PAIR; SWAP; POP]);
 BIND rev;
 PUSH 21;
 PUSH 17;
 MK_PAIR;
 LOOKUP rev;
 APPLY;
 SWAP;
 POP;
 SWAP;
 POP
```

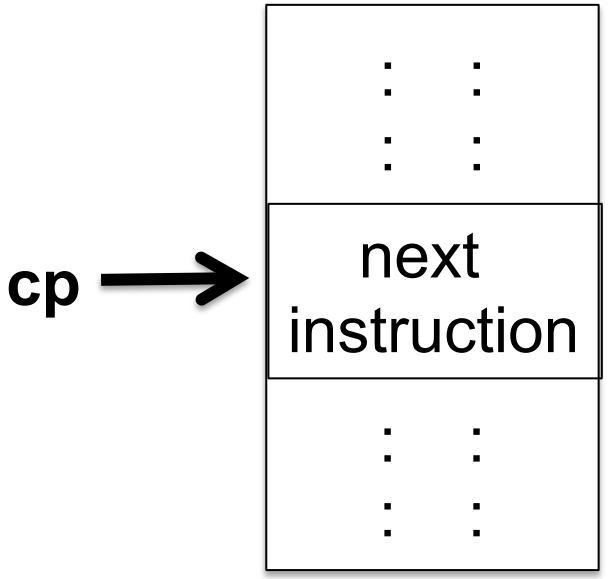
DEMO TIME!!!

LECTURE 8
Derive Interpreter 3

1. “Flatten” code into linear array
2. Add “code pointer” (cp) to machine state
3. New instructions : LABEL, GOTO, RETURN
4. “Compile away” conditionals and while loops

Linearise code

Interpreter 2 copies code
on the code stack.
We want to introduce one
global array of instructions
indexed by a code pointer (**cp**).
At runtime the **cp** points at the
next instruction to be executed.



This will require two new instructions:

LABEL L : Associate label L with this location in the code array

GOTO L : Set the **cp** to the code address associated with L

Compile conditionals, loops

`If(e1, e2, e3)`

code for `e1`
TEST k
code for `e2`
GOTO m
k: code for `e3`
m:

`While(e1, e2)`

m: code for `e1`
TEST k
code for `e2`
GOTO m
k:

If ? = 0 Then 17 else 21 end

interp_2

```
PUSH UNIT;
UNARY READ;
PUSH 0;
OPER EQI;
TEST(
   [PUSH 17],
   [PUSH 17]
)
```

interp_3

```
PUSH UNIT;
UNARY READ;
PUSH 0;
OPER EQI;
TEST L0;
PUSH 17;
GOTO L1;
LABEL L0;
PUSH 21;
LABEL L1;
HALT
```

Symbolic code locations

interp_3 (loaded)

```
0: PUSH UNIT;
1: UNARY READ;
2: PUSH 0;
3: OPER EQI;
4: TEST L0 = 7;
5: PUSH 17;
6: GOTO L1 = 9;
7: LABEL L0;
8: PUSH 21;
9: LABEL L1;
10: HALT
```

Numeric code locations

Implement inter_2 in interp_3

```
let step = function
 | ((TEST(c1, c2)) :: ds,  V(BOOL true) :: evs) -> (c1 @ ds, evs)
 | ((TEST(c1, c2)) :: ds, V(BOOL false) :: evs) -> (c2 @ ds, evs)
 :
```

interp_2.ml

```
let step (cp, evs) =
 match (get_instruction cp, evs) with
 | (TEST (_, Some _),  V(BOOL true) :: evs)  ->  (cp + 1, evs)
 | (TEST (_, Some i),  V(BOOL false) :: evs)  ->  (i,        evs)
 | (LABEL l,                                  evs)  ->  (cp + 1, evs)
 | (GOTO (_, Some i),                         evs)  -> (i,        evs)
 :
```

Interp_3.ml

Code locations are represented as

("L", None) : not yet loaded (assigned numeric address)

("L", Some i) : label "L" has been assigned numeric address i

Tricky bits again!

interp_2.ml

```
let step = function
 | (POP :: ds,                    s :: evs) -> (ds,  evs)
 | (SWAP :: ds,              s1 :: s2 :: evs) -> (ds,  s2 :: s1 :: evs)
 | ((BIND x) :: ds,             (V v) :: evs) -> (ds,  EV([(x, v)]) :: evs)
 | ((MK_CLOSURE c) :: ds,             evs) -> (ds,   V(mk_fun(c, evs_to_env evs)) :: evs)
 | (APPLY :: ds,  V(CLOSURE (_, (c, env))) :: (V v) :: evs)
                                          -> (c @ ds, (V v) :: (EV env) :: evs)
```

interp_3.ml

```
let step (cp, evs) =
 match (get_instruction cp, evs) with
 | (POP,                          s :: evs) -> (cp + 1, evs)
 | (SWAP,                   s1 :: s2 :: evs) -> (cp + 1, s2 :: s1 :: evs)
 | (BIND x,                    (V v) :: evs) -> (cp + 1, EV([(x, v)]) :: evs)
 | (MK_CLOSURE loc,                  evs) -> (cp + 1,
                                              V(CLOSURE(loc, evs_to_env evs)) :: evs)
 | (RETURN,   (V v) :: _ :: (RA i) :: evs)  -> (i, (V v) :: evs)
 | (APPLY,  V(CLOSURE ((_, Some i), env)) :: (V v) :: evs)
                                      ->  (i, (V v) :: (EV env) :: (RA (cp + 1)) :: evs)
```

Note that in interp_2 the body of a closure is consumed from the code stack. But in interp_3 we need to save the return address on the stack (here i is the location of the closure's code).

Tricky bits again!

interp_2.ml

```
let rec compile = function
 | Lambda(x, e)   -> [MK_CLOSURE((BIND x) :: (compile e) @ [SWAP; POP])]
 | App(e1, e2)    -> (compile e2) @ (compile e1) @ [APPLY; SWAP; POP]
 :
```

Interp_3.ml

```
let rec comp = function
 | App(e1, e2)   ->
  let (defs1, c1) = comp e1 in
  let (defs2, c2) = comp e2 in
      (defs1 @ defs2, c2 @ c1 @ [APPLY])
 | Lambda(x, e)   ->
  let (defs, c) = comp e in
  let f = new_label () in
  let def = [LABEL f ; BIND x] @ c @ [SWAP; POP; RETURN] in
     (def @ defs, [MK_CLOSURE((f, None))])
```

Interp_3.ml

```
let compile e =
   let (defs, c) = comp e in
     c              (* body of program *)
    @ [HALT]        (* stop the interpreter *)
    @ defs          (*  function definitions *)
```

Interpreter 3
(very similar to interpreter 2)

```
let step (cp, evs) =
 match (get_instruction cp, evs) with
  | (PUSH v,                                  evs) -> (cp + 1, (V v) :: evs)
  | (POP,                                   s :: evs) -> (cp + 1, evs)
  | (SWAP,                          s1 :: s2 :: evs) -> (cp + 1, s2 :: s1 :: evs)
  | (BIND x,                           (V v) :: evs) -> (cp + 1, EV([(x, v)]) :: evs)
  | (LOOKUP x,                                evs) -> (cp + 1, V(search(evs, x)) :: evs)
  | (UNARY op,                         (V v) :: evs) -> (cp + 1, V(do_unary(op, v)) :: evs)
  | (OPER op,          (V v2) :: (V v1) :: evs) -> (cp + 1, V(do_oper(op, v1, v2)) :: evs)
  | (MK_PAIR,          (V v2) :: (V v1) :: evs) -> (cp + 1, V(PAIR(v1, v2)) :: evs)
  | (FST,                V(PAIR (v, _)) :: evs) -> (cp + 1, (V v) :: evs)
  | (SND,                V(PAIR (_, v)) :: evs) -> (cp + 1, (V v) :: evs)
  | (MK_INL,                           (V v) :: evs) -> (cp + 1, V(INL v) :: evs)
  | (MK_INR,                           (V v) :: evs) -> (cp + 1, V(INR v) :: evs)
  | (CASE (_, Some _),               V(INL v)::evs) -> (cp + 1, (V v) :: evs)
  | (CASE (_, Some i),               V(INR v)::evs) -> (i,       (V v) :: evs)
  | (TEST (_, Some _),    V(BOOL true) :: evs) -> (cp + 1, evs)
  | (TEST (_, Some i),   V(BOOL false) :: evs) -> (i,       evs)
  | (ASSIGN,     (V v) :: (V (REF a)) :: evs) -> (heap.(a) <- v; (cp + 1, V(UNIT) :: evs))
  | (DEREF,                (V (REF a)) :: evs) -> (cp + 1, V(heap.(a)) :: evs)
  | (MK_REF,                           (V v) :: evs) -> let a = new_address () in (heap.(a) <- v;
                                                       (cp + 1, V(REF a) :: evs))
  | (MK_CLOSURE loc,                          evs) -> (cp + 1, V(CLOSURE(loc, evs_to_env evs)) :: evs)
  | (APPLY,  V(CLOSURE ((_, Some i), env)) :: (V v) :: evs)
                                                   -> (i, (V v) :: (EV env) :: (RA (cp + 1)) :: evs)
(* new intructions *)
  | (RETURN,     (V v) :: _ :: (RA i) :: evs) -> (i, (V v) :: evs)
  | (LABEL l,                                 evs) -> (cp + 1, evs)
  | (HALT,                                    evs) -> (cp, evs)
  | (GOTO (_, Some i),                        evs) -> (i, evs)
  | _ -> complain ("step : bad state = " ^ (string_of_state (cp, evs)) ^ "\n")
```

Some observations

- A very clean machine!
- But it still has a **very** inefficient treatment of environments.
- Also, pushing complex values on the stack is not what most virtual machines do. In fact, we are still using OCaml's runtime memory management to manipulate complex values.

Example : Compiled code for rev_pair.slang

```
let rev_pair (p : int * int) : int * int  = (snd p, fst p)
in
    rev_pair (21, 17)
end
```

```
MK_CLOSURE(
   [BIND p; LOOKUP p; SND;
    LOOKUP p; FST; MK_PAIR;
    SWAP; POP]);
BIND rev;
PUSH 21;
PUSH 17;
MK_PAIR;
LOOKUP rev_pair;
APPLY;
SWAP;
POP;
SWAP;
POP
```

Interp_2

```
MK_CLOSURE(rev_pair)
BIND rev
PUSH 21
PUSH 17
MK_PAIR
LOOKUP rev_pair
APPLY
SWAP
POP
HALT
```

Interp_3

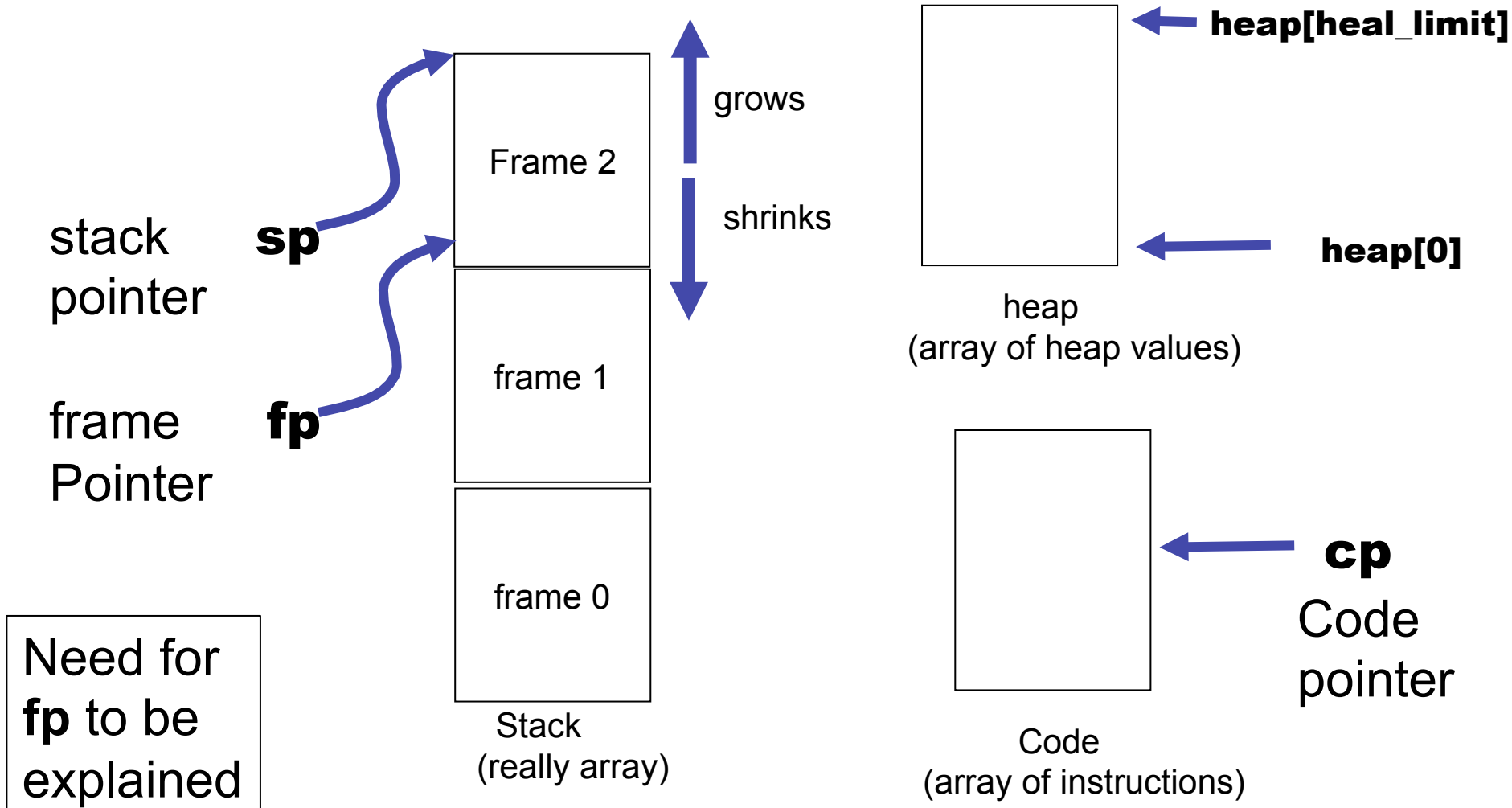
```
LABEL rev_pair
BIND p
LOOKUP p
SND
LOOKUP p
FST
MK_PAIR
SWAP
POP
RETURN
```

DEMO TIME!!!

LECTURES 9, 10
Deriving The Jargon VM
(interpreter 4)

1. **First change**: Introduce an **addressable stack.**
2. Replace variable lookup by a (relative) location on the stack or heap determined at **compile time**.
3. Relative to what? A **frame pointer** (**fp**) pointing into the stack is needed to keep track of the current **activation record.**
4. **Second change**: Optimise the representation of closures so that they contain **only** the values associated with the **free variables** of the closure.
5. **Third change**: Restrict values on stack to be simple (ints, bools, heap addresses, etc). Complex data is moved to the heap, leaving pointer into the heap on the stack.
6. How might things look different in a language without first-class functions? In a language with multiple arguments to function calls?

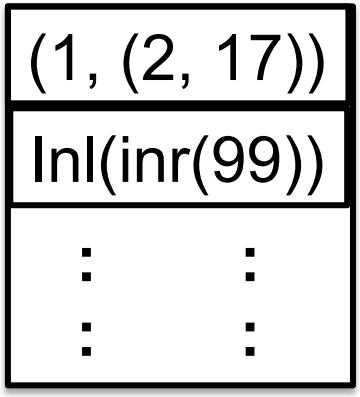
Jargon Virtual Machine



Need for **fp** to be explained soon …

The stack in interpreter 3

A stack
in interpreter 3



"All problems in computer science can be solved by another level of indirection, except of course for the problem of too many indirections."
--- David Wheeler

Stack elements in interpreter 3 are not of fixed size.

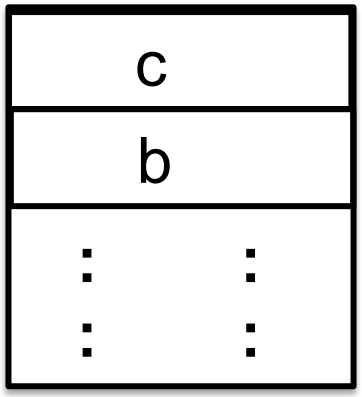
Virtual machines (JVM, etc) typically restrict stack elements to be of a fixed size

We need to shift data from the high-level stack of interpreter 3 to a lower-level stack with fixed size elements.

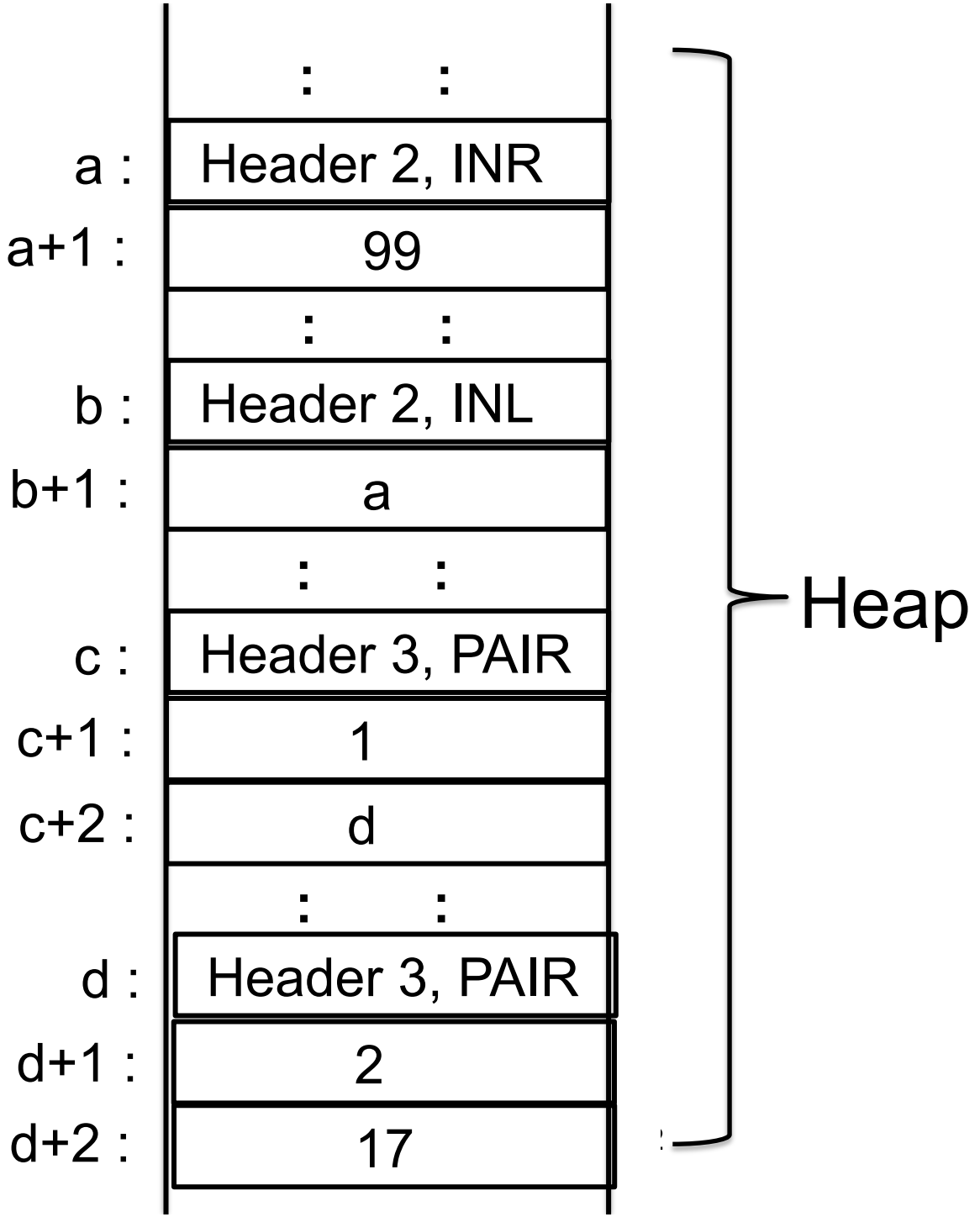
Solution : put the data in the heap. Place pointers to the heap on the stack.

The Jargon VM stack

Stack



Some stack elements represent pointers into the heap



Small change to instructions

interp_3.mli

```
type instruction =
  | PUSH of value
  | LOOKUP of Ast.var
  | UNARY of Ast.unary_oper
  | OPER of Ast.oper
  | ASSIGN
  | SWAP
  | POP
  | BIND of Ast.var
  | FST
  | SND
  | DEREF
  | APPLY
  | RETURN
  | MK_PAIR
  | MK_INL
  | MK_INR
  | MK_REF
  | MK_CLOSURE of location
  | TEST of location
  | CASE of location
  | GOTO of location
  | LABEL of label
  | HALT
```

jargon.mli

```
type instruction =
  | PUSH of stack_item          (* modified *)
  | LOOKUP of value_path        (* modified *)
  | UNARY of Ast.unary_oper
  | OPER of Ast.oper
  | ASSIGN
  | SWAP
  | POP
  (*  | BIND of var            not needed *)
  | FST
  | SND
  | DEREF
  | APPLY
  | RETURN
  | MK_PAIR
  | MK_INL
  | MK_INR
  | MK_REF
  | MK_CLOSURE of location * int    (* modified *)
  | TEST of location
  | CASE of location
  | GOTO of location
  | LABEL of label
  | HALT
```

A word about implementation

Interpreter 3

```
type value = | REF of address | INT of int | BOOL of bool | UNIT
  | PAIR of value * value | INL of value | INR of value | CLOSURE of location * env
type env_or_value = | EV of env | V of value | RA of address
type env_value_stack = env_or_value list
```

Jargon VM

```
type stack_item =
  | STACK_INT of int
  | STACK_BOOL of bool
  | STACK_UNIT
  | STACK_HI of heap_index    (* Heap Index          *)
  | STACK_RA of code_index    (* Return Address      *)
  | STACK_FP of stack_index   (* (saved) Frame Pointer *)
```

```
type heap_type =
   | HT_PAIR
   | HT_INL
   | HT_INR
   | HT_CLOSURE
```

```
type heap_item =
  | HEAP_INT of int
  | HEAP_BOOL of bool
  | HEAP_UNIT
  | HEAP_HI of heap_index              (* Heap  Index                    *)
  | HEAP_CI of code_index              (* Code pointer for closures      *)
  | HEAP_HEADER of int * heap_type (* int is number items in heap block *)
```

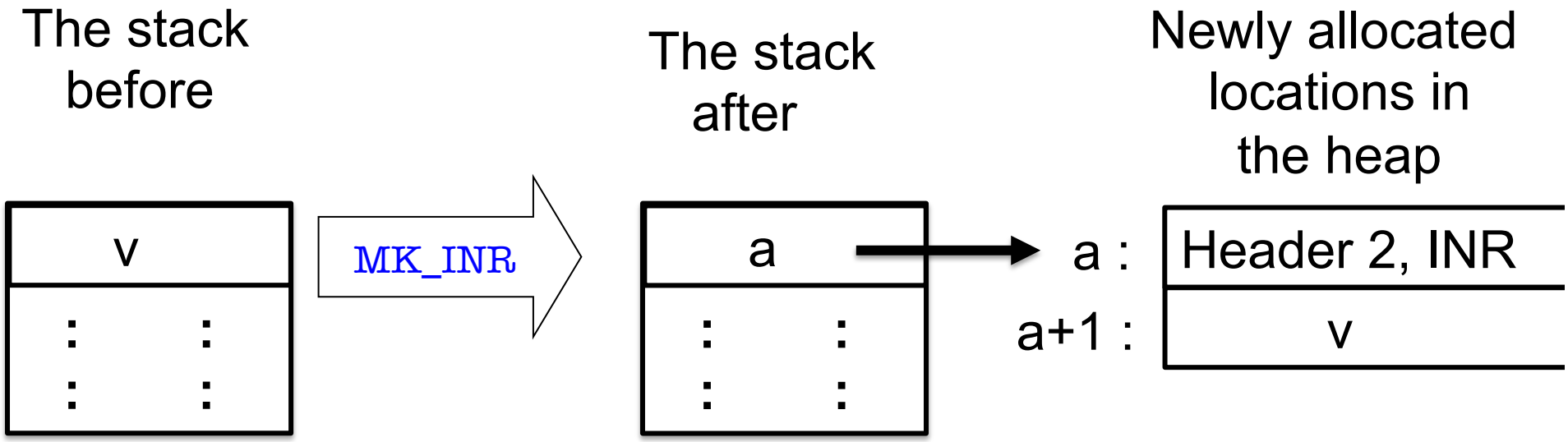
The headers will be essential for garbage collection!

MK_INR (MK_INL is similar)

In interpreter 3

```
(MK_INR,    (V v) :: evs)   ->   (cp + 1, V(INR(v)) :: evs)
```

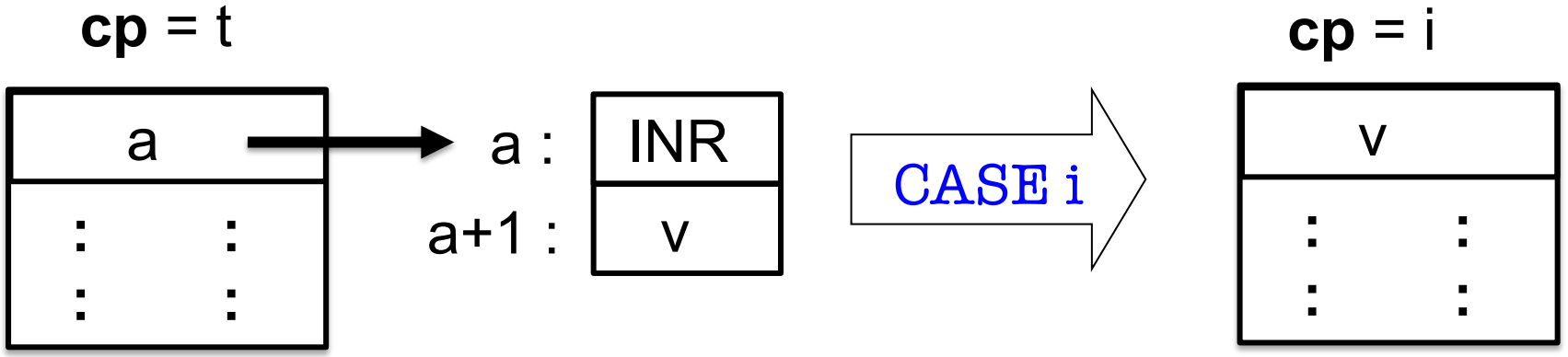
Jargon VM

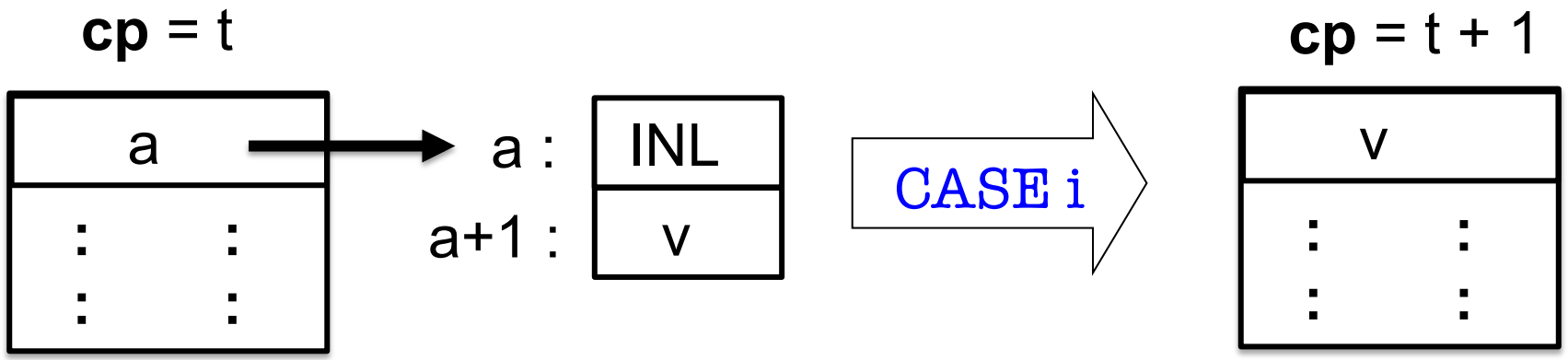


Note: The header types are not really required. We could instead add an extra field here (for example, 0 or 1). However, header types aid in understanding the code and traces of runtime execution.

CASE (TEST is similar)

```
(CASE (_, Some _),  V(INL v)::evs) -> (cp + 1, (V v) :: evs)
(CASE (_, Some i),  V(INR v)::evs) -> (i,       (V v) :: evs)
```



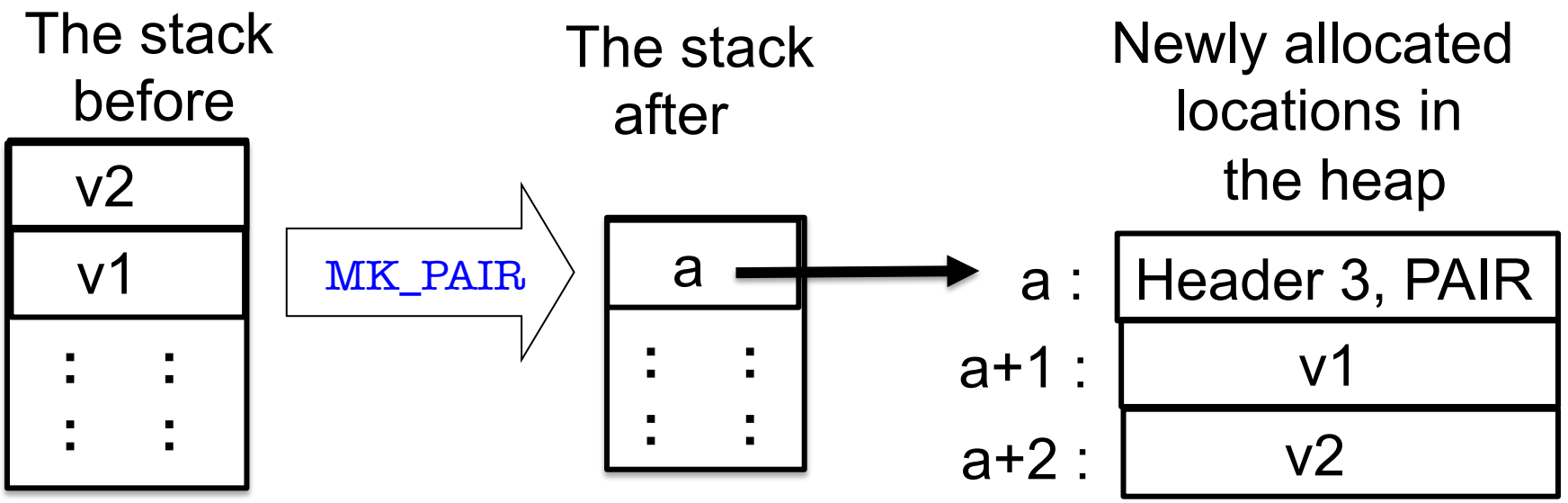


MK_PAIR

In interpreter 3:

```
(MK_PAIR,    (V v2) :: (V v1) :: evs)   ->   (cp + 1, V(PAIR(v1, v2)) :: evs)
```

In Jargon VM:

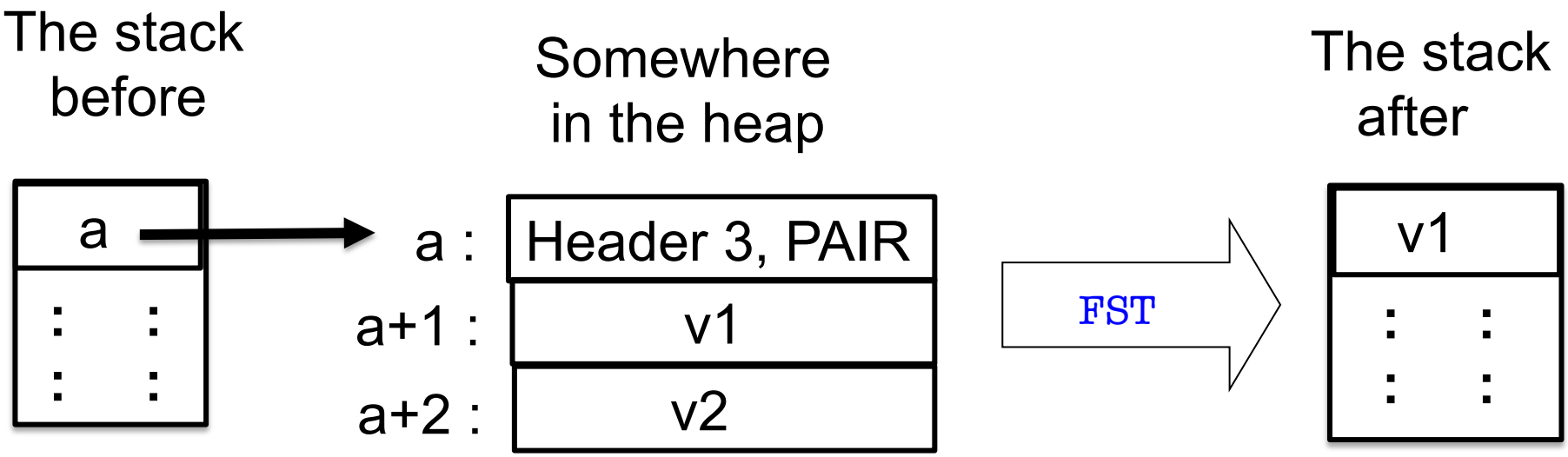


FST (similar for SND)

In interpreter 3:

```
(FST,    V (PAIR(v1, v2)) :: evs)   ->   (cp + 1, v1 :: evs)
```

In Jargon VM:



Note that v1 could be a simple value (int or bool), or aother heap address.

These require more care ...

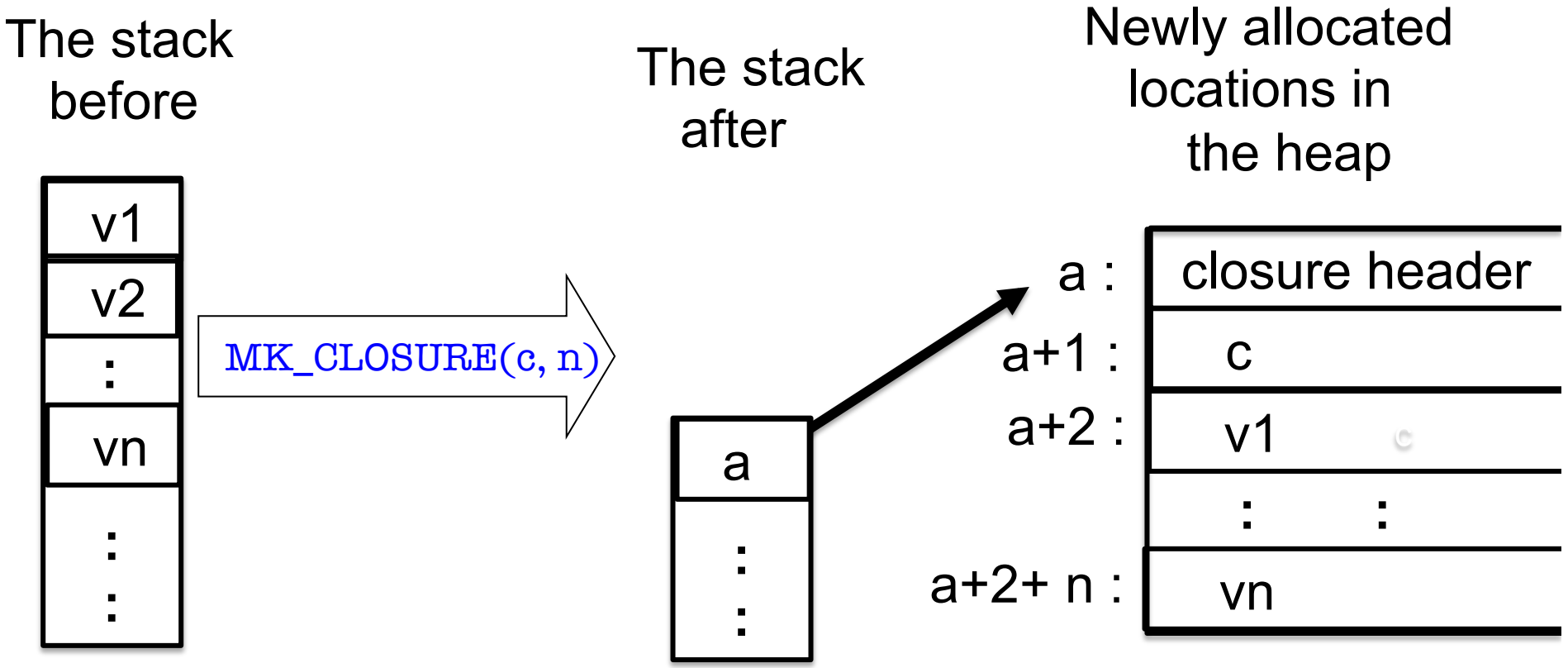
In interpreter 3:

```
let step (cp, evs) =
 match (get_instruction cp, evs) with
| (MK_CLOSURE loc,   evs)
   -> (cp + 1, V(CLOSURE(loc, evs_to_env evs)) :: evs)

| (APPLY,    V(CLOSURE ((_, Some i), env)) :: (V v) :: evs)
   -> (i,  (V v) :: (EV env) :: (RA (cp + 1)) :: evs)

| (RETURN,    (V v) :: _ :: (RA i) :: evs)
   -> (i,  (V v) :: evs)
```

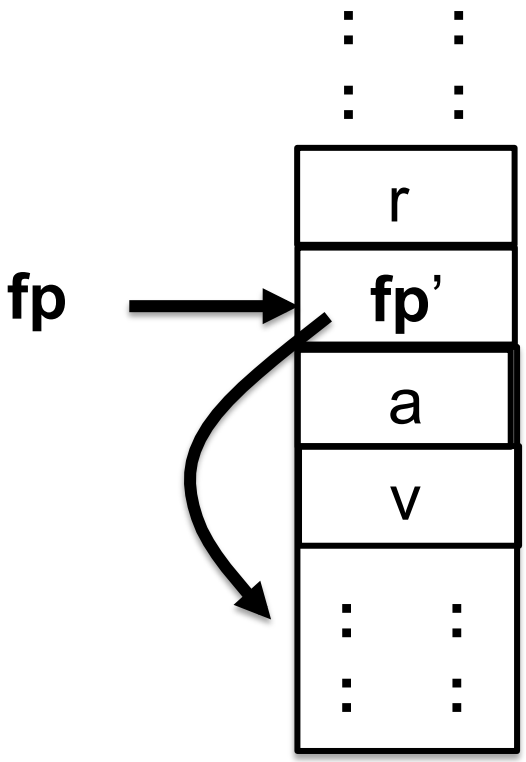
MK_CLOSURE(c, n)

c = code location of start of instructions for closure,
n = number of free variables in the body of closure.

Put values associated with **free variables** on stack, then construct the closure on the heap



A stack frame



Return address
Saved frame pointer
Pointer to closure
Argument value

Stack frame. (Boundary May vary in the literature.)

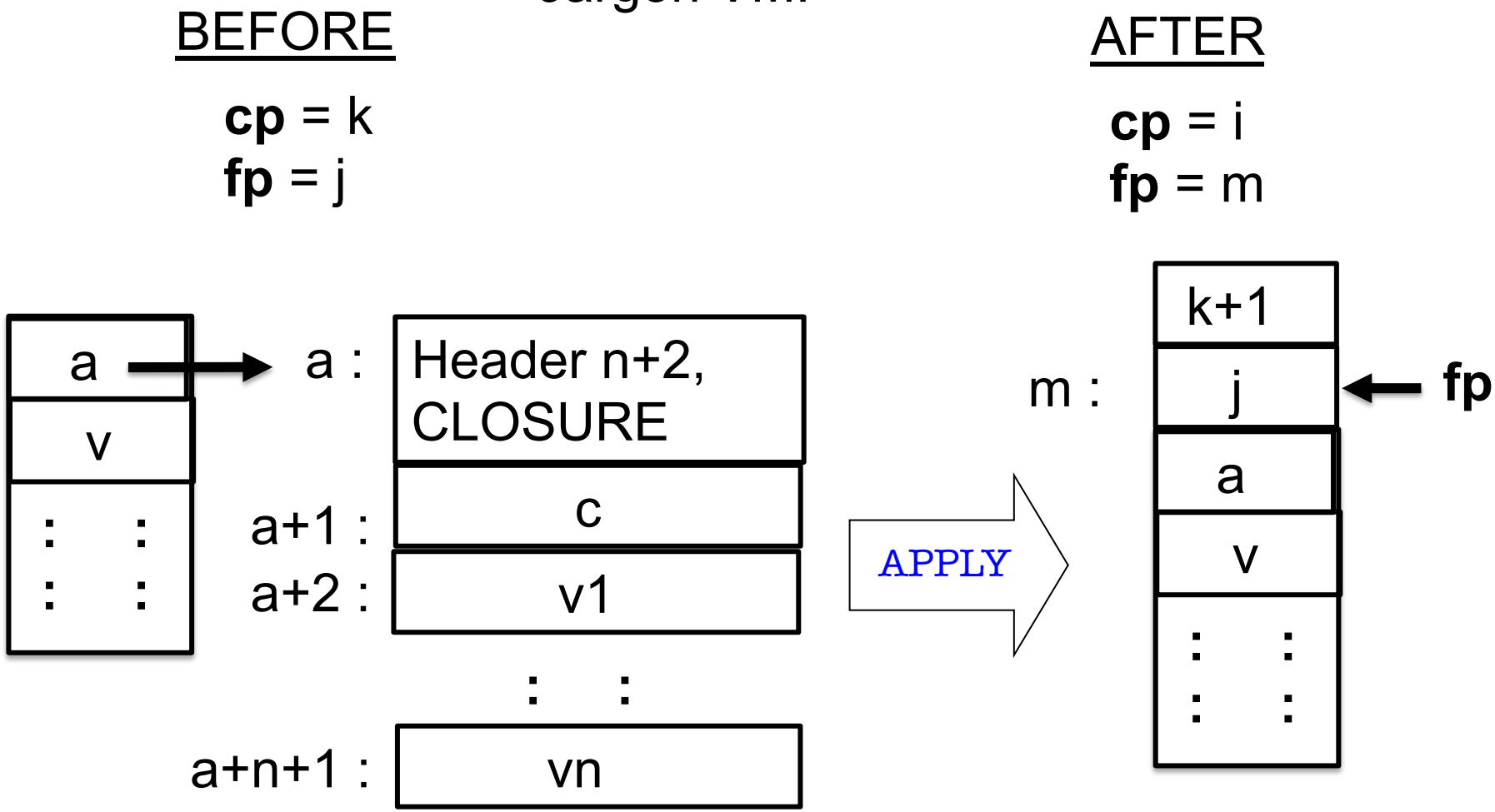
Currently executing code for the closure at heap address “a” after it was applied to argument v.

APPLY

Interpreter 3:

```
(APPLY,   V(CLOSURE ((_, Some i), env)) :: (V v) :: evs)
                                  -> (i,  (V v) :: (EV env) :: (RA (cp + 1)) :: evs)
```

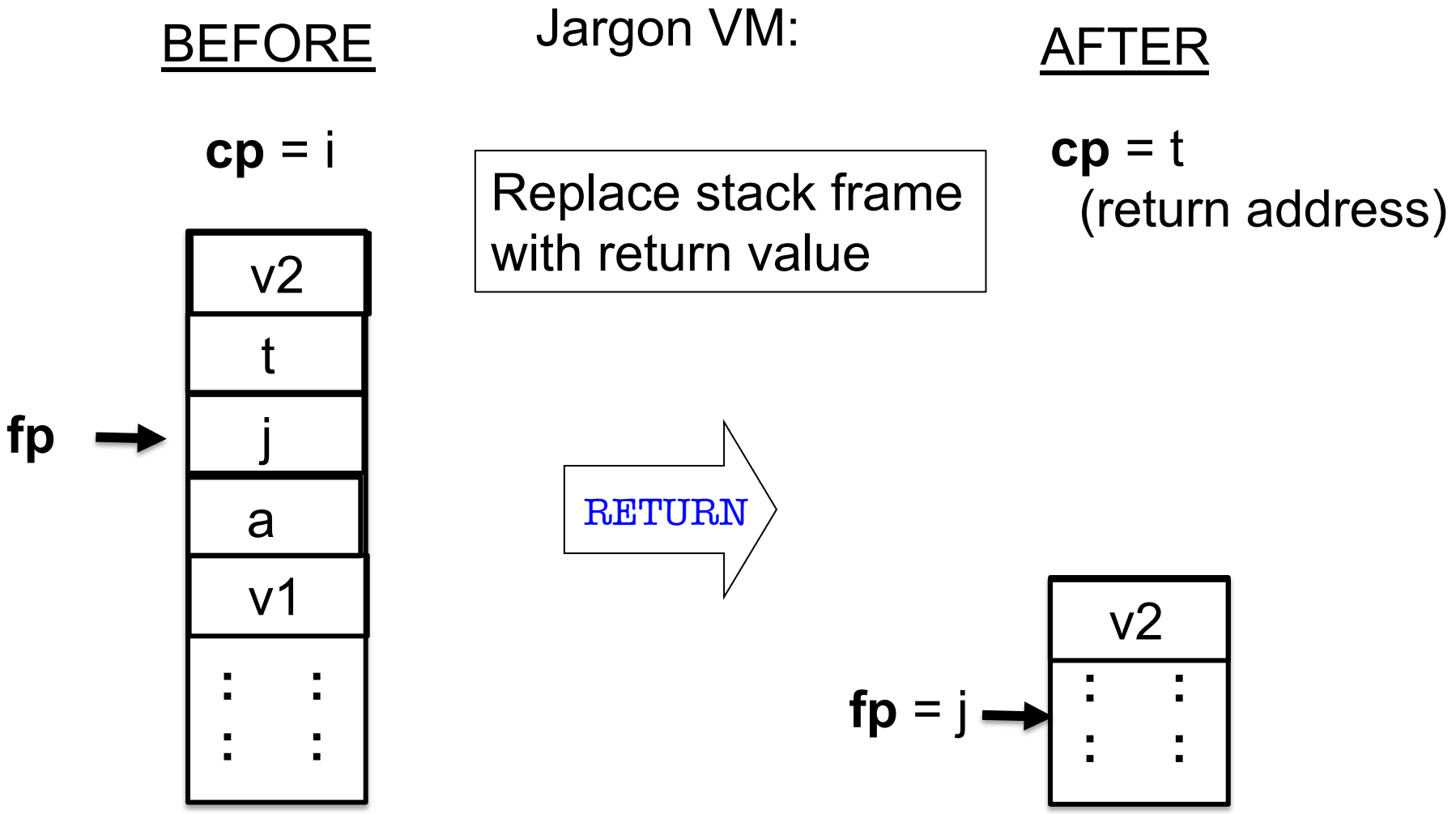
Jargon VM:



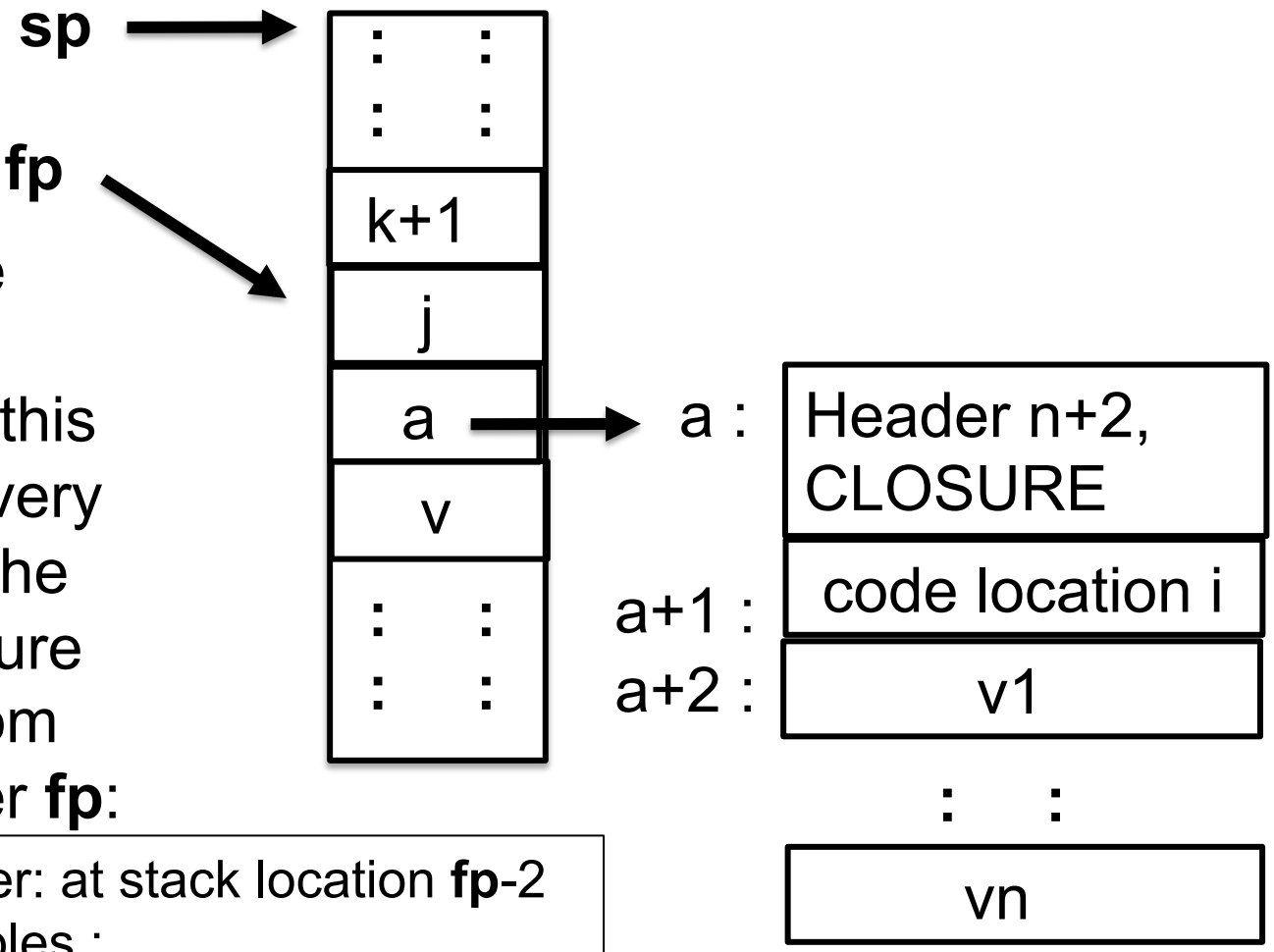
RETURN

Interpreter 3:

```
(RETURN,  (V v) :: _ :: (RA i) :: evs)  ->  (i,  (V v) :: evs)
```



Finding a variable's value at runtime



Suppose we are executing code associated with this closure. Then every free variable in the body of the closure can be found from the frame pointer **fp**:

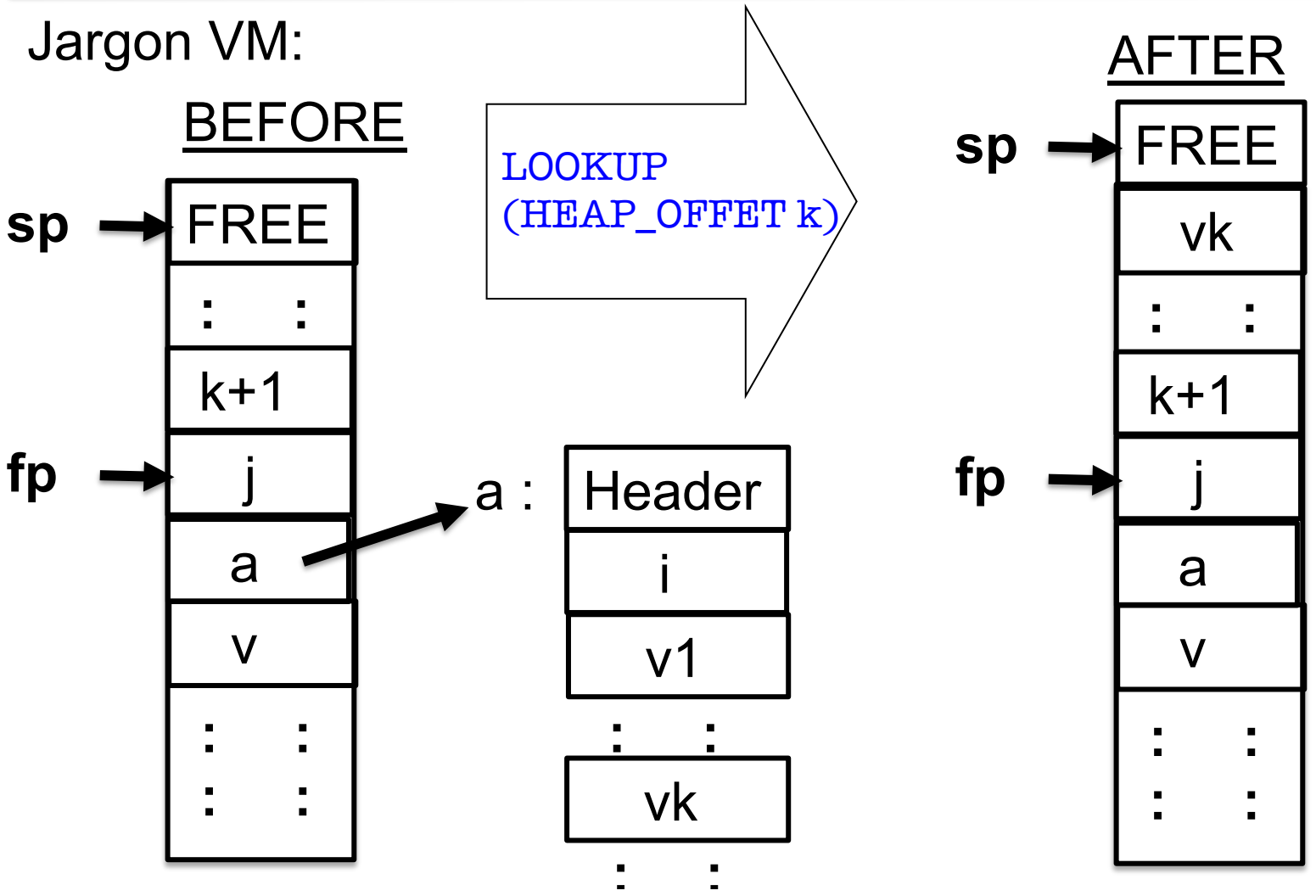
- Formal parameter: at stack location **fp**-2
- Other free variables :
  - Follow heap pointer found at **fp** -1
  - Each free variable can be associated with a fixed offset from this heap address

LOOKUP (HEAP_OFFSET k)

Interpreter 3:

(LOOKUP x, evs) -> (cp + 1, V(search(evs, x)) :: evs)

Jargon VM:



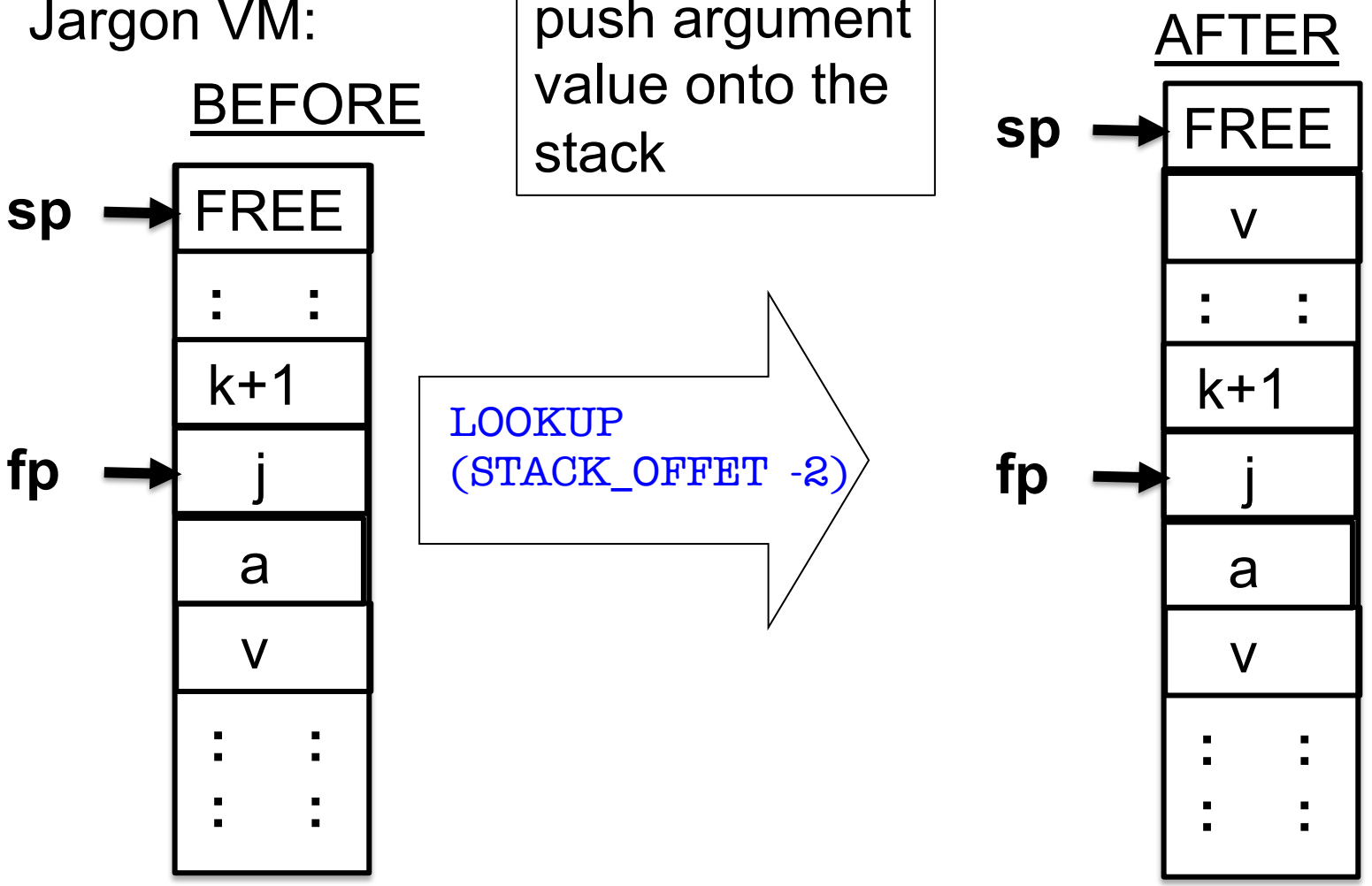
LOOKUP (STACK_OFFSET -2)

Interpreter 3:

```
(LOOKUP x,          evs) -> (cp + 1, V(search(evs, x)) :: evs)
```

Jargon VM:

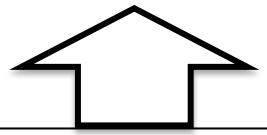
push argument value onto the stack



Oh, one problem

interpreter 3

```
let rec comp = function
 :
 | LetFun(f, (x, e1), e2) ->
              let (defs1, c1) = comp e1 in
              let (defs2, c2) = comp e2 in
              let def = [LABEL f; BIND x] @ c1 @ [SWAP; POP; RETURN] in
                 (def @ defs1 @ defs2,
                  [MK_CLOSURE((f, None)); BIND f] @ c2 @ [SWAP; POP])
 :
```



Problem: Code c2 can be anything --- how are we going to find the closure for f when we need it? It has to be a fixed offset from a frame pointer --- we no longer scan the stack for bindings!

Solution in Jargon VM

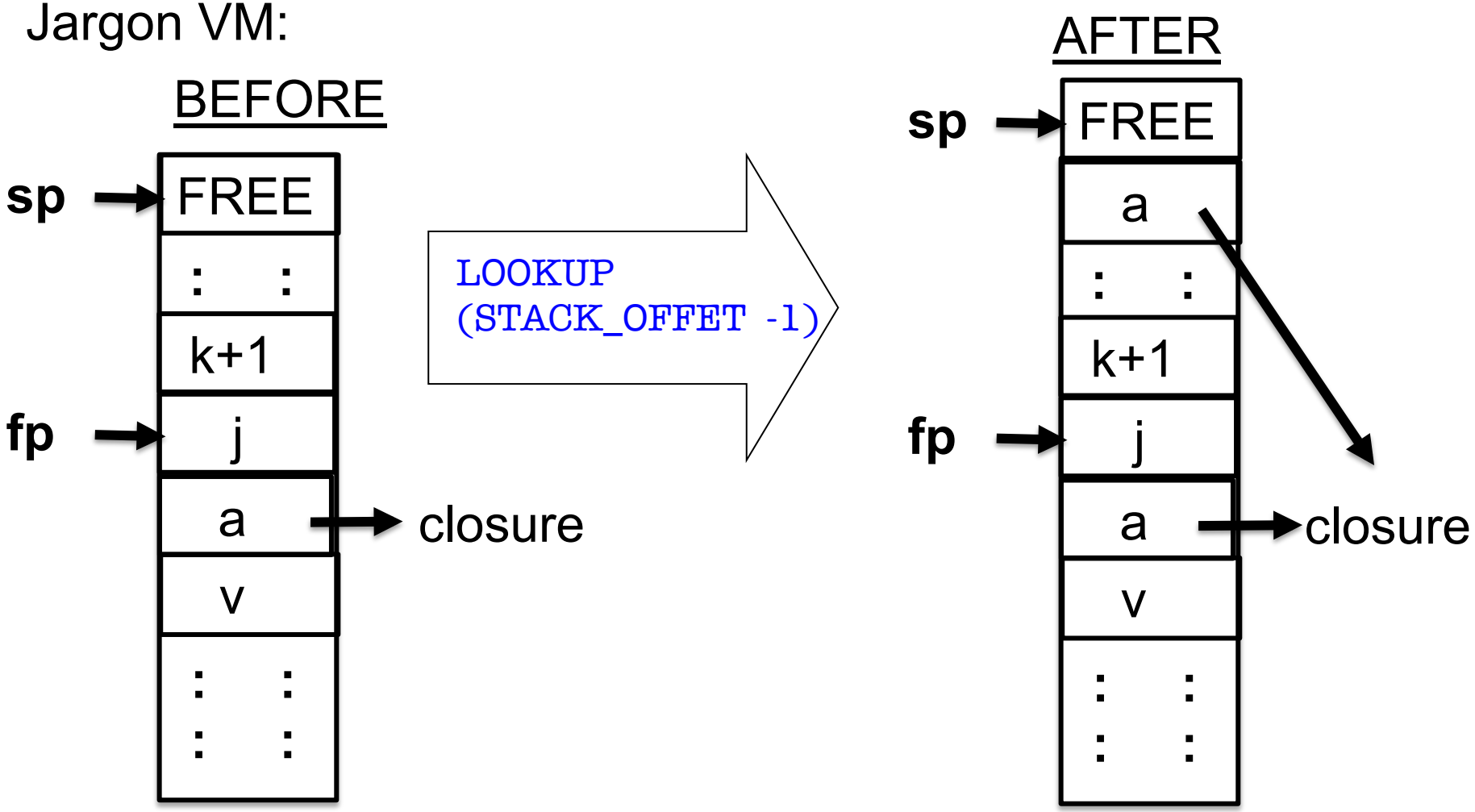
```
let rec comp vmap = function
 :
| LetFun(f, (x, e1), e2) -> comp vmap (App(Lambda(f, e2), Lambda(x, e1)))
:
```

Similar trick for LetRecFun

LOOKUP (STACK_OFFSET -1)

For recursive function calls,
push current closure on to the stack.

Jargon VM:



Example : Compiled code for rev_pair.slang

```
let rev_pair (p : int * int) : int * int  = (snd p, fst p)
in
    rev_pair (21, 17)
end
```

After the front-end, compile treats this as follows.

```
App(
  Lambda(
    "rev_pair",
     App(Var "rev_pair",  Pair (Integer 21, Integer 17))),
  Lambda("p", Pair(Snd (Var "p"), Fst (Var "p"))))
```

Example : Compiled code for rev_pair.slang

```
App(
  Lambda("rev_pair",
            App(Var "rev_pair",  Pair (Integer 21, Integer 17))),
  Lambda("p", Pair(Snd (Var "p"), Fst (Var "p"))))
```

"first lambda"

"second lambda'

```
          MK_CLOSURE(L1, 0)          -- Make closure for second lambda
          MK_CLOSURE(L0, 0)          -- Make closure for first lambda
          APPLY                      -- do application
          HALT                       -- the end!
L0 :      PUSH STACK_INT 21          -- code for first lambda, push 21
          PUSH STACK_INT 17          -- push 17
          MK_PAIR                    -- make the pair on the heap
          LOOKUP STACK_LOCATION -2   -- push closure for second lambda on stack
          APPLY                      -- apply first lambda
          RETURN                     -- return from first lambda
L1 :      LOOKUP STACK_LOCATION -2   -- code for second lambda, push arg on stack
          SND                        -- extract second part of pair
          LOOKUP STACK_LOCATION -2   -- push arg on stack again
          FST                        -- extract first part of pair
          MK_PAIR                    -- construct a new pair
          RETURN                     -- return from second lambda
```

Example : trace of rev_pair.slang execution

```
Installed Code =
0: MK_CLOSURE(L1 = 11, 0)
1: MK_CLOSURE(L0 = 4, 0)
2: APPLY
3: HALT
4: LABEL L0
5: PUSH STACK_INT 21
6: PUSH STACK_INT 17
7: MK_PAIR
8: LOOKUP STACK_LOCATION-2
9: APPLY
10: RETURN
11: LABEL L1
12: LOOKUP STACK_LOCATION-2
13: SND
14: LOOKUP STACK_LOCATION-2
15: FST
16: MK_PAIR
17: RETURN
```

```
========== state 1 ==========
cp = 0 -> MK_CLOSURE(L1 = 11, 0)
fp = 0
Stack =
1: STACK_RA 0
0: STACK_FP 0

========== state 2 ==========
cp = 1 -> MK_CLOSURE(L0 = 4, 0)
fp = 0
Stack =
2: STACK_HI 0
1: STACK_RA 0
0: STACK_FP 0

Heap =
0 -> HEAP_HEADER(2, HT_CLOSURE)
1 -> HEAP_CI 11

.....
```

Example : trace of rev_pair.slang execution

```
========== state 15 ==========
cp = 16 -> MK_PAIR
fp = 8
Stack =
11: STACK_INT 21
10: STACK_INT 17
9: STACK_RA 10
8: STACK_FP 4
7: STACK_HI 0
6: STACK_HI 4
5: STACK_RA 3
4: STACK_FP 0
3: STACK_HI 2
2: STACK_HI 0
1: STACK_RA 0
0: STACK_FP 0

Heap =
0 -> HEAP_HEADER(2, HT_CLOSURE)
1 -> HEAP_CI 11
2 -> HEAP_HEADER(2, HT_CLOSURE)
3 -> HEAP_CI 4
4 -> HEAP_HEADER(3, HT_PAIR)
5 -> HEAP_INT 21
6 -> HEAP_INT 17
```

```
========== state 19 ==========
cp = 3 -> HALT
fp = 0
Stack =
2: STACK_HI 7
1: STACK_RA 0
0: STACK_FP 0

Heap =
0 -> HEAP_HEADER(2, HT_CLOSURE)
1 -> HEAP_CI 11
2 -> HEAP_HEADER(2, HT_CLOSURE)
3 -> HEAP_CI 4
4 -> HEAP_HEADER(3, HT_PAIR)
5 -> HEAP_INT 21
6 -> HEAP_INT 17
7 -> HEAP_HEADER(3, HT_PAIR)
8 -> HEAP_INT 17
9 -> HEAP_INT 21


Jargon VM :
output> (17, 21)
```

Example : closure_add.slang

```
let f(y : int) : int -> int = let g(x :int) : int = y + x  in g end
in let add21 : int -> int  = f(21)
    in let add17 : int -> int  = f(17)
        in add17(3) + add21(10)
        end
   end
end
```

Note : we really do need closures on the heap here — the values 21 and 17 do not exist on the stack at this point in the execution.

After the front-end, this becomes represented as follows.

```
App(Lambda(f, App(Lambda(add21,
                         App(Lambda(add17,
                                    Op(App(Var(add17), Integer(3)),
                                       ADD,
                                       App(Var(add21), Integer(10)))),
                             App(Var(f), Integer(17)))),
                  App(Var(f), Integer(21))))),
    Lambda(y, App(Lambda(g, Var(g)), Lambda(x, Op(Var(y), ADD, Var(x))))))
```

Can we make sense of this?

```
        MK_CLOSURE(L3, 0)
        MK_CLOSURE(L0, 0)
        APPLY
        HALT
L0 :    PUSH STACK_INT 21
        LOOKUP STACK_LOCATION -2
        APPLY
        LOOKUP STACK_LOCATION -2
        MK_CLOSURE(L1, 1)
        APPLY
        RETURN
L1 :    PUSH STACK_INT 17
        LOOKUP HEAP_LOCATION 1
        APPLY
        LOOKUP STACK_LOCATION -2
        MK_CLOSURE(L2, 1)
        APPLY
        RETURN
```

```
L2 :    PUSH STACK_INT 3
        LOOKUP STACK_LOCATION -2
        APPLY
        PUSH STACK_INT 10
        LOOKUP HEAP_LOCATION 1
        APPLY
        OPER ADD
        RETURN
L3 :    LOOKUP STACK_LOCATION -2
        MK_CLOSURE(L5, 1)
        MK_CLOSURE(L4, 0)
        APPLY
        RETURN
L4 :    LOOKUP STACK_LOCATION -2
        RETURN
L5 :    LOOKUP HEAP_LOCATION 1
        LOOKUP STACK_LOCATION -2
        OPER ADD
        RETURN
```

The Gap, illustrated

fib.slang

```
let fib (m :int) : int  =
   if m = 0
   then 1
   else if m = 1
         then 1
         else fib(m - 1) + fib (m - 2)
         end
    end
in fib (?) end
```

slang.byte –c –i4 fib.slang

```
                MK_CLOSURE(fib, 0)
                MK_CLOSURE(L0, 0)
                APPLY
                HALT
L0 :            PUSH STACK_UNIT
                UNARY READ
                LOOKUP STACK_LOCATION -2
                APPLY
                RETURN
fib :           LOOKUP STACK_LOCATION -2
                PUSH STACK_INT 0
                OPER EQI
                TEST L1
                PUSH STACK_INT 1
                GOTO L2
L1 :            LOOKUP STACK_LOCATION -2
                PUSH STACK_INT 1
                OPER EQI
                TEST L3
                PUSH STACK_INT 1
                GOTO L4
L3 :            LOOKUP STACK_LOCATION -2
                PUSH STACK_INT 1
                OPER SUB
                LOOKUP STACK_LOCATION -1
                APPLY
                LOOKUP STACK_LOCATION -2
                PUSH STACK_INT 2
                OPER SUB
                LOOKUP STACK_LOCATION -1
                APPLY
                OPER ADD
L4 :
L2 :            RETURN
```

Jargon VM code

Remarks

1. The semantic GAP between a Slang/L3 program and a low-level translation (say x86/Unix) has been significantly reduced.
2. Implementing the Jargon VM at a lower-level of abstraction (in C?, JVM bytecodes? X86/Unix? …) looks like a relatively easy programming problem.
3. However, using a lower-level implementation (say x86, exploiting fast registers) to generate very efficient code is not so easy. See Part II Optimising Compilers.

Verification of compilers is an active area of research. See CompCert, CakeML, and DeepSpec.

What about languages other than Slang/L3?

- Many textbooks on compilers treat only languages with first-order functions --- that is, functions cannot be passes as an argument or returned as a result. In this case, we can avoid allocating environments on the heap since all values associated with free variables will be somewhere on the stack!
- But how do we find these values? We optimise stack search by following a chain of **static links**. Static links are added to every stack frame and the point to the stack frame of the last invocation of the defining function.
- One other thing: most languages take multiple arguments for a function/procedure call.

Terminology: Caller and Callee

```
fun f (x, y) = e1

…

fun g(w, v) =
    w + f(v, v)
```

For this invocation of the function f, we say that g is the caller while f is the callee

Recursive functions can play both roles at the same time …

Nesting depth

Pseudo-code

```
fun b(z) = e

 fun g(x1) =
   fun h(x2) =
     fun f(x3) = e3(x1, x2, x3, b, g h, f)
     in
        e2(x1, x2, b, g, h, f)
      end
   in
      e1(x1, b, g, h)
  end
...
b(g(17))
...
```

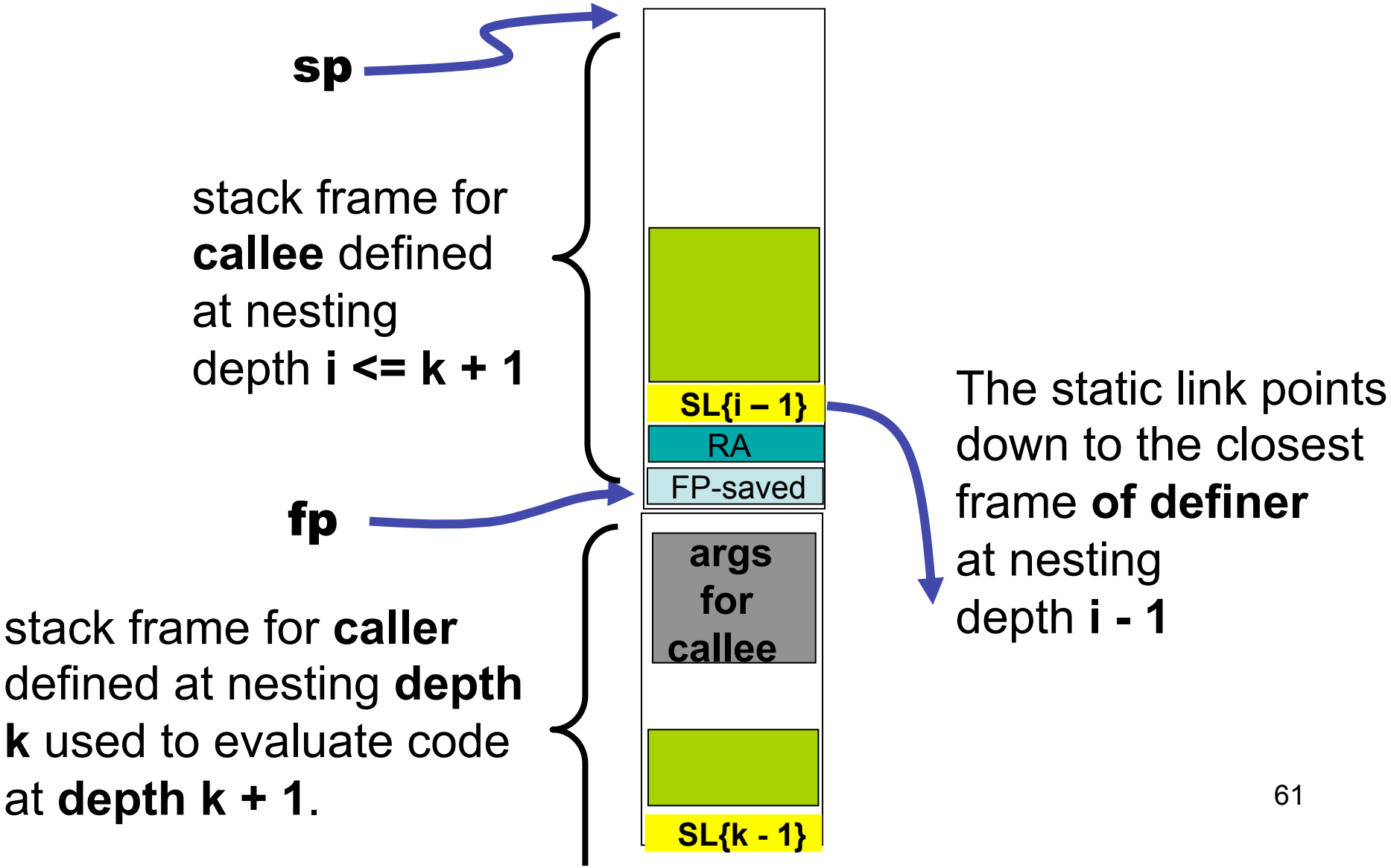
Nesting depth

code in big box is at nesting depth k

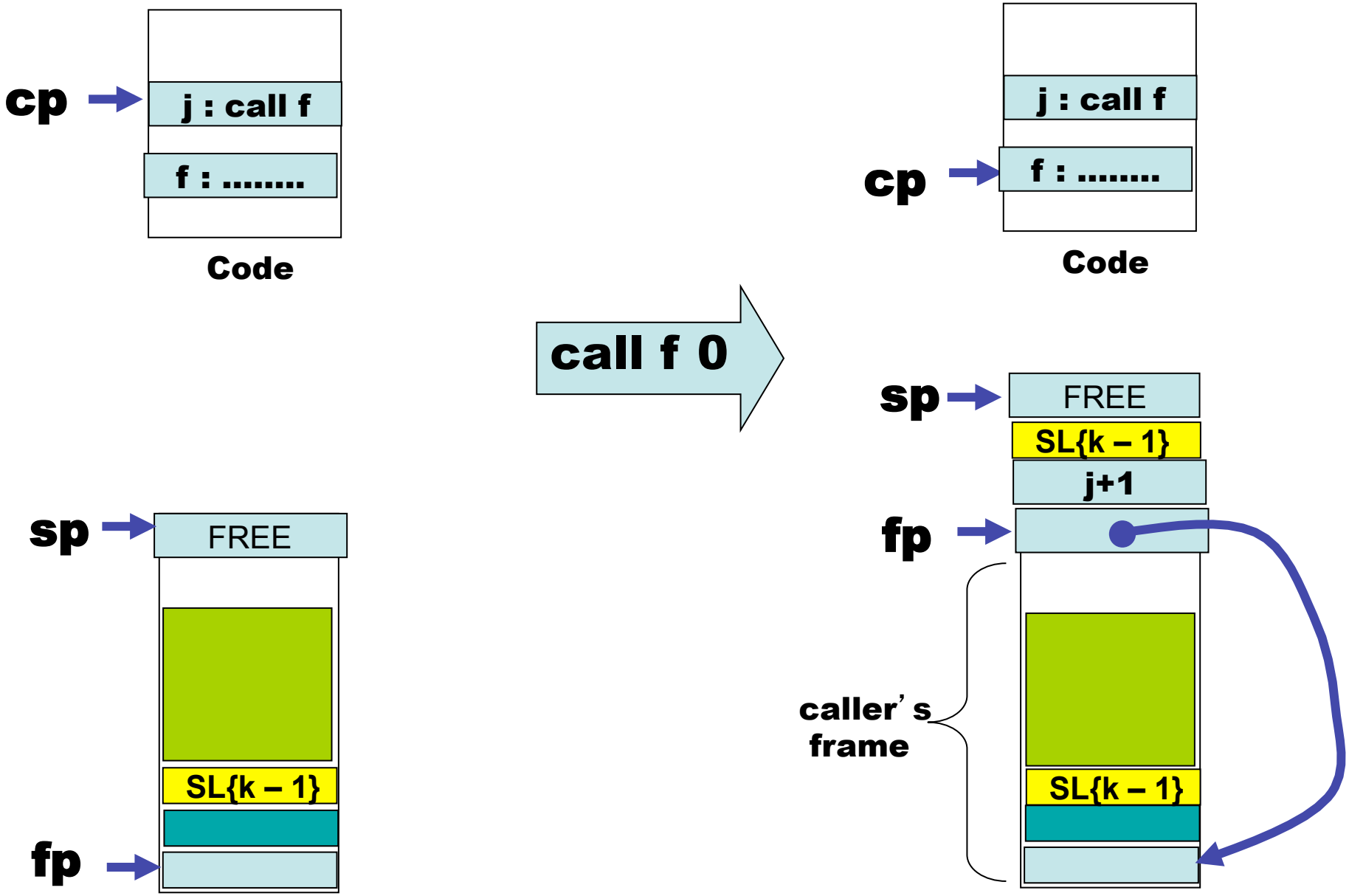
```
fun b(z) = e  nesting depth k + 1

 fun g(x1) =
   fun h(x2) =
     fun f(x3) = e3(x1, x2, x3, b, g h, f)        nesting depth k + 3
     in
        e2(x1, x2, b, g, h, f)
      end                                          nesting depth k + 2
   in
       e1(x1, b, g, h)
  end                                              nesting depth k + 1
...
b(g(17))
...
```


Function g is the **definer** of h. Functions g and b must share a definer defined at depth k-1

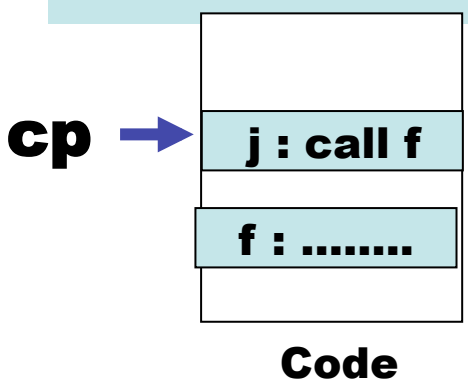
Stack with static links and variable number of arguments

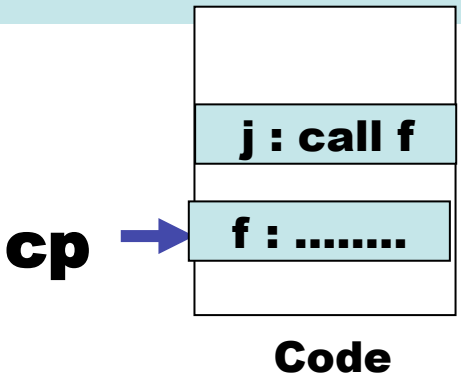


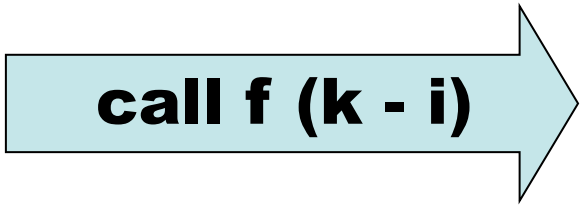
caller and callee at same nesting depth k

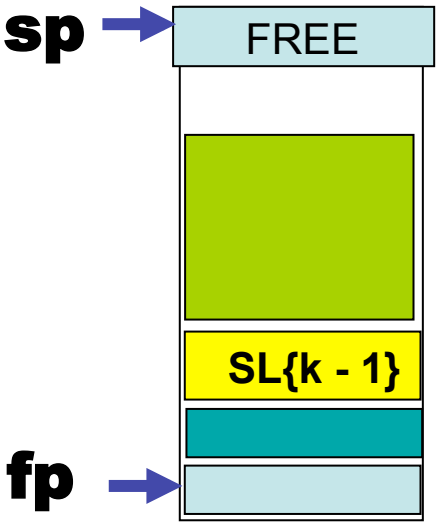


caller at depth k and callee at depth i < k

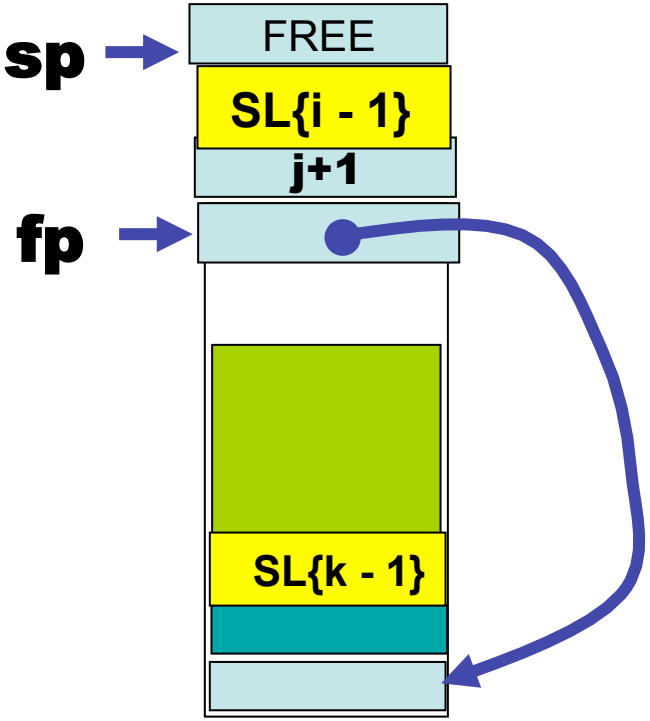




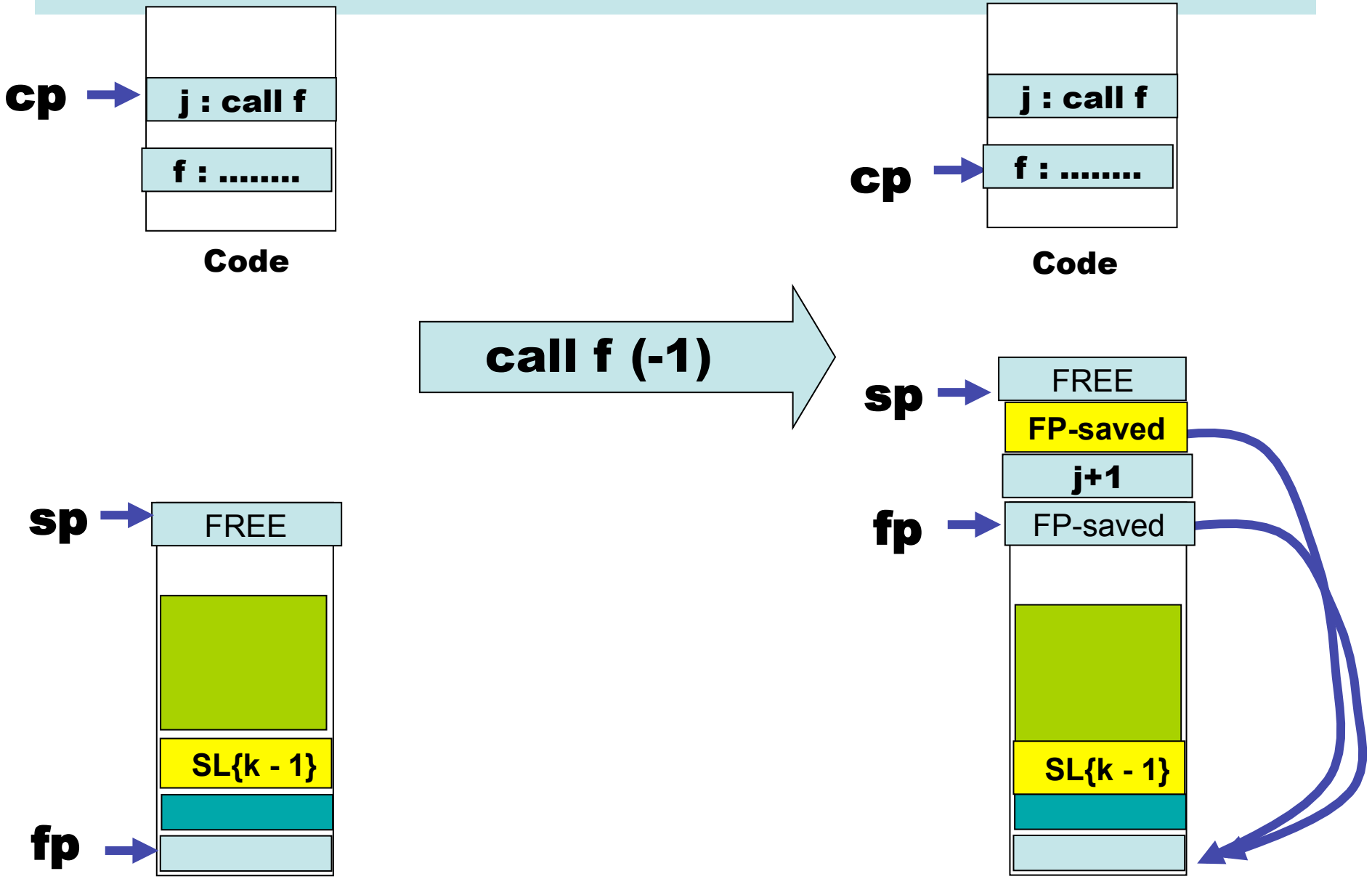




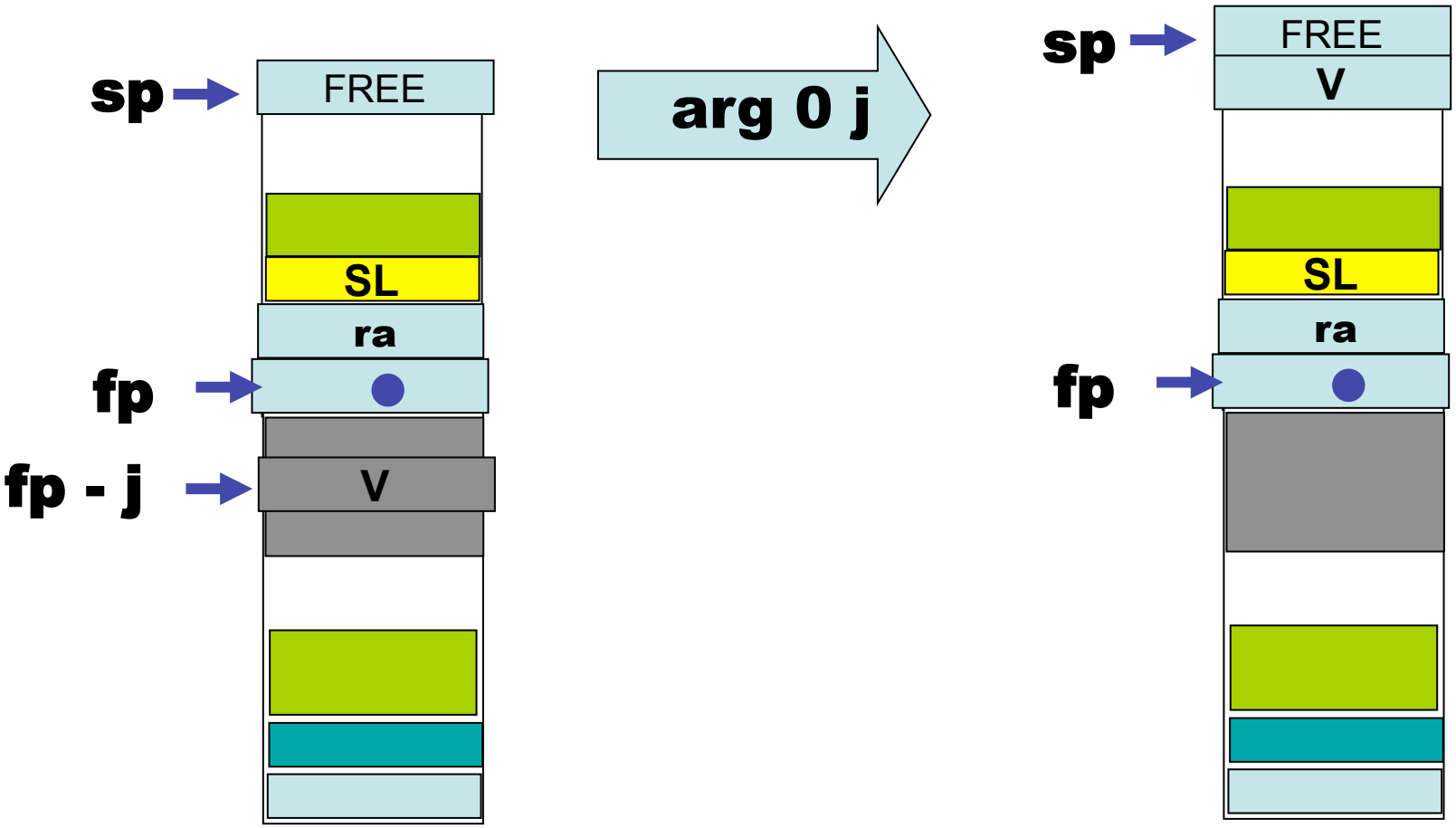

```
p := !(fp + 2);
for c = 1 to k - i
{
   p := !(p + 2);
}
SL{i-1} := p;
```



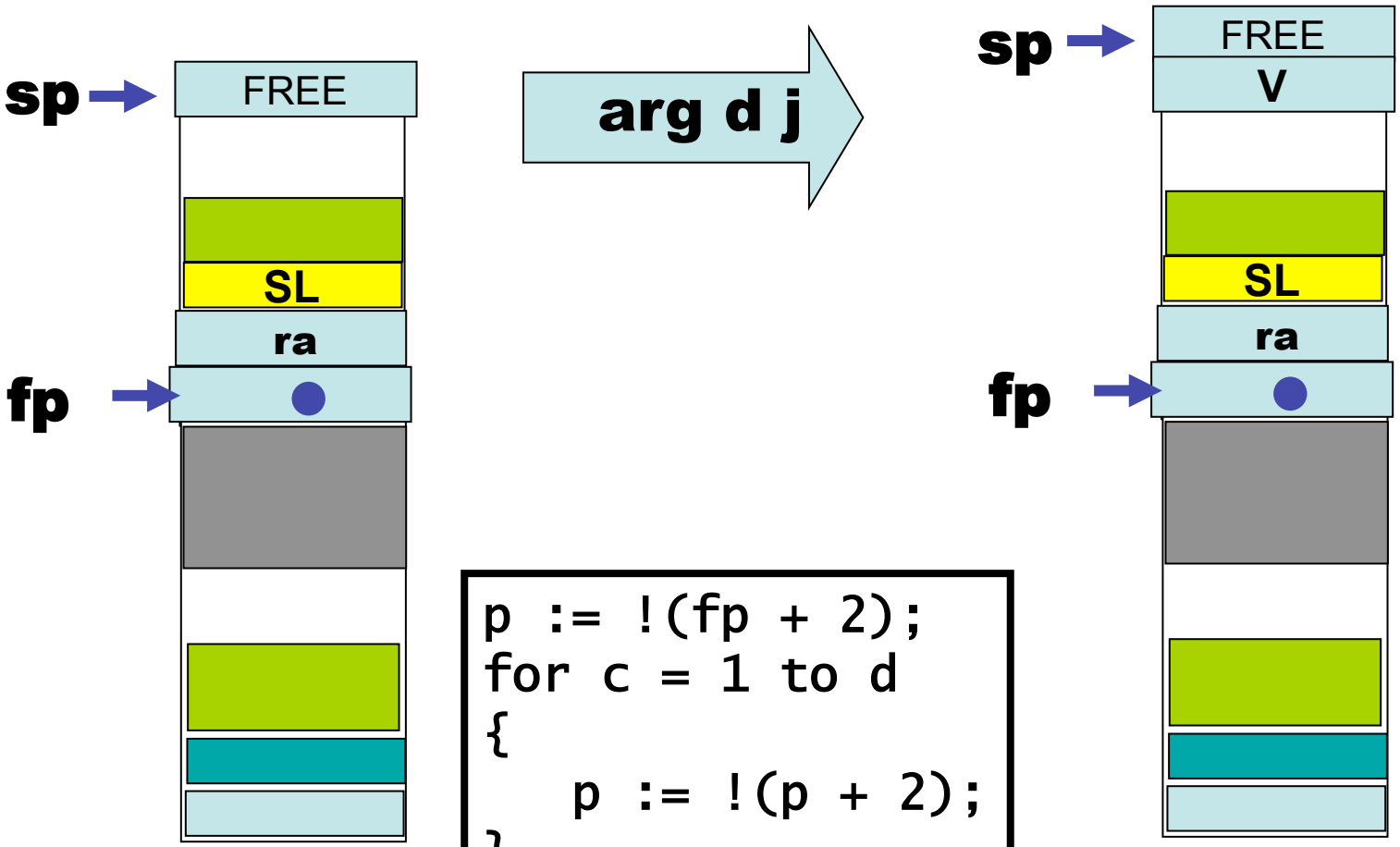
caller at depth k and callee at depth k + 1



Access to argument values at static distance 0



Access to argument values at static distance d, 0 < d




```
p := !(fp + 2);
for c = 1 to d
{
    p := !(p + 2);
}
v := !(p - j);
```

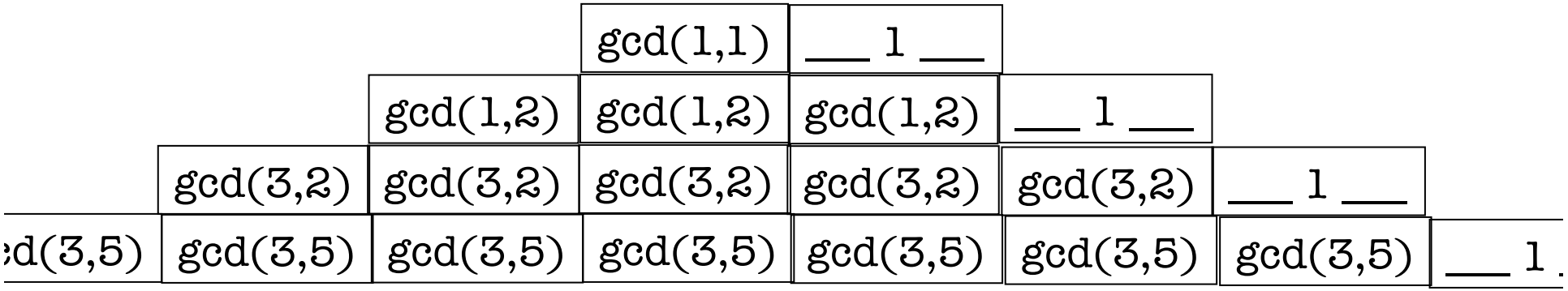
LECTUREs 11, 12
What about Interpreter 1?

- Evaluation using a stack
- Recursion using a stack
- Tail recursion elimination: from recursion to iteration
- Continuation Passing Style (CPS) : transform any recursive function to a tail-recursive function
- "Defunctionalisation" (DFC) : replace higher-order functions with a data structure
- Putting it all together:
  - Derive the Fibonacci Machine
  - Derive the Expression Machine, and "compiler"!
- This provides a roadmap for the interp_0 → interp_1 → interp_2 derivations.

Example of tail-recursion : gcd

```
(* gcd : int * int -> int *)
let rec gcd(m, n) =
   if m = n
   then m
   else if m < n
      then gcd(m,     n - m)
      else  gcd(m - n,     n)
```

Compared to fib, this function uses recursion in a different way. It is **tail-recursive**. If implemented with a stack, then the "call stack" (at least with respect to gcd) will simply grow and then shrink. No "ups and downs" in between.



Tail-recursive code can be replaced by iterative code that does not require a "call stack" (constant space)

gcd_iter : gcd without recursion!

```
(* gcd : int * int -> int *)
let rec gcd(m, n) =
   if m = n
   then m
   else if m < n
        then gcd(m,     n - m)
        else  gcd(m - n,      n)
```

Here we have illustrated tail-recursion elimination as a source-to-source transformation. However, the OCaml compiler will do something similar to a lower-level intermediate representation. Upshot : we will consider all tail-recursive OCaml functions as representing iterative programs.

```
(* gcd_iter : int * int -> int *)
let gcd_iter (m, n) =
   let rm = ref m
   in let rn = ref n
   in let result = ref 0
   in let not_done = ref true
   in let _ =
       while !not_done
        do
            if !rm = !rn
            then (not_done := false;
                    result := !rm)
            else if !rm < !rn
                    then rn := !rn - !rm
                    else rm := !rm - !rn
        done
in !result
```

Familiar examples : fold_left, fold_right

From ocaml-4.01.0/stdlib/list.ml :

```
(* fold_left :  ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

    fold_left f a [b1; ...; bn]] = f (... (f (f a b1) b2) ...) bn
*)
let rec fold_left f a l =
  match l with
  | []          -> a
  | b :: rest -> fold_left f (f a b) rest

(* fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

   fold_right f [a1; ...; an] b = f a1 (f a2 (... (f an b) ...))
 *)
let rec fold_right f l b =
  match l with
  | []         -> b
  | a::rest -> f a (fold_right f rest b)
```

This is tail recursive

This is NOT tail recursive

Question: can we transform any recursive function into a tail recursive function?

The answer is YES!

- **We add an extra argument, called a *continuation*, that represents "the rest of the computation"**
- **This is called the Continuation Passing Style (CPS) transformation.**
- **We will then "defunctionalize" (DFC) these continuations and represent them with a stack.**
- **Finally, we obtain a tail recursive function that carries its own stack as an extra argument!**

We will apply this kind of transformation to the code of interpreter 0 as the first steps towards deriving interpreter 1.

(CPS) transformation of fib

```
(* fib : int -> int *)
let rec fib m =
   if m = 0
   then 1
   else if m = 1
        then 1
        else fib(m - 1) + fib (m - 2)

(* fib_cps : int * (int -> int)  -> int *)
 let rec fib_cps (m,  cnt) =
   if m = 0
   then cnt 1
   else if m = 1
       then cnt 1
       else fib_cps(m -1,  fun a -> fib_cps(m - 2 , fun b  -> cnt (a + b)))
```

A closer look

The rest of the computation after computing “fib(m)”. That is, cnt is a function expecting the result of “fib(m)” as its argument.

```
let rec fib_cps (m, cnt) =
   if m = 0
   then cnt 1
   else if m = 1
        then cnt 1
        else fib_cps(m -1, fun a -> fib_cps(m - 2 , fun b -> cnt (a + b)))
```

The computation waiting for the result of “fib(m-1)”

The computation waiting for the result of “fib(m-2)”

This makes explicit the order of evaluation that is implicit in the original “fib(m-1) + fib(m-2)” :
-- first compute fib(m-1)
-- then compute fib(m-1)
-- then add results together
-- then return

Expressed with “let” rather than “fun”

```
(* fib_cps_v2 : (int -> int) * int -> int *)
let rec fib_cps_v2 (m, cnt) =
   if m = 0
   then cnt 1
   else if m = 1
        then cnt 1
        else let cnt2 a b = cnt (a + b)
             in let cnt1 a = fib_cps_v2(m - 2, cnt2 a)
             in fib_cps_v2(m - 1, cnt1)
```

Some prefer writing CPS forms without explicit funs ….

Use the identity continuation ...

```
(* fib_cps : int * (int -> int)  -> int *)
 let rec fib_cps (m, cnt) =
   if m = 0
   then cnt 1
   else if m = 1
        then cnt 1
        else fib_cps(m -1,  fun a -> fib_cps(m - 2 , fun b  -> cnt (a + b)))


let id (x : int) = x

let fib_1 x = fib_cps(x, id)
```

```
List.map fib_1 [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;

  =  [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
```

Correctness?

For all c : int -> int, for all m, 0 <= m,
we have, c(fib m) = fib_cps(m, c).

Proof: assume c : int -> int. By Induction on m. Base case : m = 0:

fib_cps(0, c) = c(1) = c(fib(0).

NB: This proof pretends that we can treat OCaml functions as ideal mathematical functions, which of course we cannot. OCaml functions might raise exceptions like "stack overflow" or "you burned my toast", and so on. But this is a convenient fiction as long as we remember to be careful.

Induction step: Assume for all n < m, c(fib n) = fib_cps(n, c).
(That is, we need course-of-values induction!)

```
fib_cps(m + 1, c)
= if m + 1 = 1
  then c 1
  else fib_cps((m+1) -1, fun a -> fib_cps((m+1) -2, fun b -> c (a + b)))
= if m + 1 = 1
  then c 1
  else fib_cps(m, fun a -> fib_cps(m-1, fun b -> c (a + b)))
= (by induction)
  if m + 1 = 1
  then c 1
  else (fun a -> fib_cps(m -1, fun b -> c (a + b))) (fib m)
```

Correctness?

```
= if m + 1 = 1
  then c 1
  else fib_cps(m-1, fun b -> c ((fib m) + b))
= (by induction)
  if m + 1 = 1
  then c 1
  else (fun b -> c ((fib m) + b)) (fib (m-1))
 = if m + 1 = 1
  then c 1
  else c ((fib m) + (fib (m-1)))
 = c (if m + 1 = 1
      then 1
      else ((fib m) + (fib (m-1))))
 = c(if m +1 = 1
      then 1
      else fib((m + 1) - 1) + fib ((m + 1) - 2))
 = c (fib(m + 1))
```


QED.

Can with express fib_cps without a functional argument ?

```
(* fib_cps_v2 : (int -> int) * int -> int *)
let rec fib_cps_v2 (m, cnt) =
   if m = 0
   then cnt 1
   else if m = 1
        then cnt 1
        else let cnt2 a b = cnt (a + b)
             in let cnt1 a = fib_cps_v2(m - 2, cnt2 a)
             in fib_cps_v2(m - 1, cnt1)
```

Idea of “defunctonalisation” (DFC): replace id, cnt1 and cnt2 with instances of a new data type:

```
type cnt = ID | CNT1 of int * cnt | CNT2 of int * cnt
```

Now we need an “apply” function of type cnt * int -> int

“Defunctionalised” version of fib_cps

```
(* datatype to represent continuations *)
type cnt = ID | CNT1 of int * cnt | CNT2 of int * cnt

(* apply_cnt : cnt * int -> int *)
let rec apply_cnt = function
  | (ID, a)                  -> a
  | (CNT1 (m, cnt), a) -> fib_cps_dfc(m - 2, CNT2 (a, cnt))
  | (CNT2 (a, cnt), b)   -> apply_cnt (cnt, a + b)

(*  fib_cps_dfc : (cnt * int) -> int *)
and fib_cps_dfc (m, cnt) =
   if m = 0
   then apply_cnt(cnt, 1)
   else if m = 1
         then apply_cnt(cnt, 1)
         else fib_cps_dfc(m -1, CNT1(m, cnt))

(*  fib_2 : int -> int *)
let fib_2 m = fib_cps_dfc(m, ID)
```

Correctness?

Let < c > be of type cnt representing
a continuation c : int -> int constructed by fib_cps.

Then
apply_cnt(< c >, m) = c(m)
and
fib_cps(n, c) = fib_cps_dfc(n, < c >).

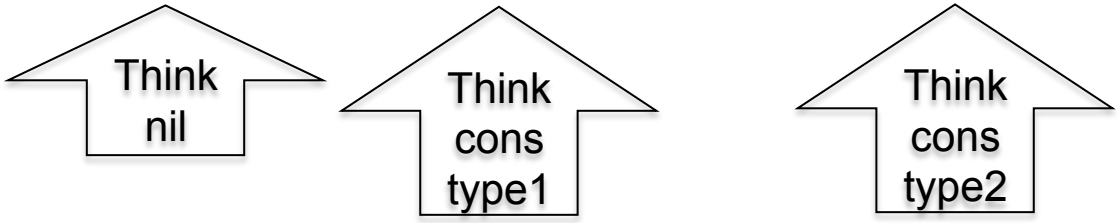
Proof left as an exercise!

Functional continuation c	Representation < c >
fun a -> fib_cps(m - 2 , fun b -> cnt (a + b))	CNT1(m, < cnt >)
fun b -> cnt (a + b)	CNT2(a, < cnt >)
fun x -> x	ID

Eureka! Continuations are just lists (used like a stack)

```
type int_list = NIL | CONS of int * int_list
```

```
type cnt = ID | CNT1 of int * cnt | CNT2 of int * cnt
```



Replace the above continuations with lists! (I've selected more suggestive names for the constructors.)

```
type tag = SUB2 of int | PLUS of int

type tag_list_cnt = tag list
```

The continuation lists are used like a stack!

```
type tag = SUB2 of int | PLUS of int
type tag_list_cnt = tag list

(* apply_tag_list_cnt : tag_list_cnt * int -> int *)
let rec apply_tag_list_cnt = function
  | ([], a)                    -> a
  | ((SUB2 m) :: cnt, a) -> fib_cps_dfc_tags(m - 2, (PLUS a):: cnt)
  | ((PLUS a) :: cnt, b)  -> apply_tag_list_cnt (cnt, a + b)

(* fib_cps_dfc_tags : (tag_list_cnt * int) -> int *)
and fib_cps_dfc_tags (m, cnt) =
   if m = 0
   then apply_tag_list_cnt(cnt, 1)
   else if m = 1
        then apply_tag_list_cnt(cnt, 1)
        else fib_cps_dfc_tags(m - 1, (SUB2 m) :: cnt)

(*  fib_3 : int -> int *)
let fib_3 m = fib_cps_dfc_tags(m, [])
```

Combine Mutually tail-recursive functions into a single function

```
type state_type =
  | SUB1 (* for right-hand-sides starting with fib_   *)
  | APPL  (* for right-hand-sides starting with apply_ *)

type state = (state_type * int * tag_list_cnt) -> int

(* eval : state -> int          A two-state transition function*)
let rec eval = function
  | (SUB1, 0,                cnt) -> eval (APPL, 1,                    cnt)
  | (SUB1, 1,                cnt) -> eval (APPL, 1,                    cnt)
  | (SUB1, m,                cnt) -> eval (SUB1, (m-1), (SUB2 m) :: cnt)
  | (APPL, a, (SUB2 m) :: cnt) -> eval (SUB1, (m-2), (PLUS a) :: cnt)
  | (APPL, b,  (PLUS a) :: cnt) -> eval (APPL, (a+b),                 cnt)
  | (APPL, a,                []) -> a

(*  fib_4 : int -> int *)
let fib_4 m = eval (SUB1, m, [])
```

Eliminate tail recursion to obtain The Fibonacci Machine!

```
(* step : state -> state *)
let step = function
  | (SUB1, 0,                  cnt) -> (APPL, 1,                  cnt)
  | (SUB1, 1,                  cnt) -> (APPL, 1,                  cnt)
  | (SUB1, m,                  cnt) -> (SUB1, (m-1), (SUB2 m) :: cnt)
  | (APPL, a, (SUB2 m) :: cnt) -> (SUB1, (m-2),  (PLUS a) :: cnt)
  | (APPL, b,  (PLUS a) :: cnt) -> (APPL, (a+b),                cnt)
  | _ -> failwith "step : runtime error!"

(* clearly TAIL RECURSIVE! *)
let rec driver state = function
    | (APPL, a, []) -> a
    |  state        -> driver (step state)

(*  fib_5 : int -> int *)
let fib_5 m = driver  (SUB1, m, [])
```

In this version we have simply made the tail-recursive structure very explicit.

Here is a trace of fib_5 6.

```
 1 SUB1 || 6 || []
 2 SUB1 || 5 || [SUB2 6]
 3 SUB1 || 4 || [SUB2 6, SUB2 5]
 4 SUB1 || 3 || [SUB2 6, SUB2 5, SUB2 4]
 5 SUB1 || 2 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3]
 6 SUB1 || 1 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3, SUB2 2]
 7 APPL || 1 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3, SUB2 2]
 8 SUB1 || 0 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3, PLUS 1]
 9 APPL || 1 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3, PLUS 1]
10 APPL || 2 || [SUB2 6, SUB2 5, SUB2 4, SUB2 3]
11 SUB1 || 1 || [SUB2 6, SUB2 5, SUB2 4, PLUS 2]
12 APPL || 1 || [SUB2 6, SUB2 5, SUB2 4, PLUS 2]
13 APPL || 3 || [SUB2 6, SUB2 5, SUB2 4]
14 SUB1 || 2 || [SUB2 6, SUB2 5, PLUS 3]
15 SUB1 || 1 || [SUB2 6, SUB2 5, PLUS 3, SUB2 2]
16 APPL || 1 || [SUB2 6, SUB2 5, PLUS 3, SUB2 2]
17 SUB1 || 0 || [SUB2 6, SUB2 5, PLUS 3, PLUS 1]
18 APPL || 1 || [SUB2 6, SUB2 5, PLUS 3, PLUS 1]
19 APPL || 2 || [SUB2 6, SUB2 5, PLUS 3]
20 APPL || 5 || [SUB2 6, SUB2 5]
21 SUB1 || 3 || [SUB2 6, PLUS 5]
22 SUB1 || 2 || [SUB2 6, PLUS 5, SUB2 3]
23 SUB1 || 1 || [SUB2 6, PLUS 5, SUB2 3, SUB2 2]
24 APPL || 1 || [SUB2 6, PLUS 5, SUB2 3, SUB2 2]
25 SUB1 || 0 || [SUB2 6, PLUS 5, SUB2 3, PLUS 1]
26 APPL || 1 || [SUB2 6, PLUS 5, SUB2 3, PLUS 1]
27 APPL || 2 || [SUB2 6, PLUS 5, SUB2 3]
28 SUB1 || 1 || [SUB2 6, PLUS 5, PLUS 2]
29 APPL || 1 || [SUB2 6, PLUS 5, PLUS 2]
30 APPL || 3 || [SUB2 6, PLUS 5]
31 APPL || 8 || [SUB2 6]
32 SUB1 || 4 || [PLUS 8]
33 SUB1 || 3 || [PLUS 8, SUB2 4]
34 SUB1 || 2 || [PLUS 8, SUB2 4, SUB2 3]
35 SUB1 || 1 || [PLUS 8, SUB2 4, SUB2 3, SUB2 2]
36 APPL || 1 || [PLUS 8, SUB2 4, SUB2 3, SUB2 2]
37 SUB1 || 0 || [PLUS 8, SUB2 4, SUB2 3, PLUS 1]
38 APPL || 1 || [PLUS 8, SUB2 4, SUB2 3, PLUS 1]
39 APPL || 2 || [PLUS 8, SUB2 4, SUB2 3]
40 SUB1 || 1 || [PLUS 8, SUB2 4, PLUS 2]
41 APPL || 1 || [PLUS 8, SUB2 4, PLUS 2]
42 APPL || 3 || [PLUS 8, SUB2 4]
43 SUB1 || 2 || [PLUS 8, PLUS 3]
44 SUB1 || 1 || [PLUS 8, PLUS 3, SUB2 2]
45 APPL || 1 || [PLUS 8, PLUS 3, SUB2 2]
46 SUB1 || 0 || [PLUS 8, PLUS 3, PLUS 1]
47 APPL || 1 || [PLUS 8, PLUS 3, PLUS 1]
48 APPL || 2 || [PLUS 8, PLUS 3]
49 APPL || 5 || [PLUS 8]
50 APPL ||13|| []
```

The OCaml file in basic_transformations/fibonacci_machine.ml contains some code for pretty printing such traces….

Pause to reflect

- What have we accomplished?
- We have taken a recursive function and turned it into an iterative function that does not require "stack space" for its evaluation (in OCaml)
- However, this function now carries its own evaluation stack as an extra argument!
- We have derived this iterative function in a step-by-step manner where each tiny step is easily proved correct.
- Wow!

That was fun! Let's do it again!

```
type expr =
   | INT of int
   | PLUS of expr * expr
   | SUBT of expr * expr
   | MULT of expr * expr
```

This time we will derive a stack-machine AND a "compiler" that translates expressions into a list of instructions for the machine.

```
(* eval : expr -> int
   a simple recusive evaluator for expressions *)
let rec eval = function
   | INT a              -> a
   | PLUS(e1, e2)   -> (eval e1) + (eval e2)
   | SUBT(e1, e2)   -> (eval e1) - (eval e2)
   | MULT(e1, e2)   -> (eval e1) * (eval e2)
```

Here we go again : CPS

```
type cnt_2  = int -> int

type state_2 = expr * cnt_2

(* eval_aux_2 : state_2 -> int *)
let rec eval_aux_2 (e, cnt) =
  match e with
   | INT a       -> cnt a
   | PLUS(e1, e2) ->
     eval_aux_2(e1, fun v1 -> eval_aux_2(e2, fun v2 -> cnt(v1 + v2)))
   | SUBT(e1, e2) ->
     eval_aux_2(e1, fun v1 -> eval_aux_2(e2, fun v2 -> cnt(v1 - v2)))
   | MULT(e1, e2) ->
     eval_aux_2(e1, fun v1 -> eval_aux_2(e2, fun v2 -> cnt(v1 * v2)))

 (* id_cnt : cnt_2 *)
let id_cnt (x : int) = x

(*  eval_2 : expr -> int *)
let eval_2 e = eval_aux_2(e, id_cnt)
```

Defunctionalise!

```
type cnt_3 =
  | ID
  | OUTER_PLUS of expr * cnt_3
  | OUTER_SUBT of expr * cnt_3
  | OUTER_MULT of expr * cnt_3
  | INNER_PLUS of int * cnt_3
  | INNER_SUBT of int * cnt_3
  | INNER_MULT of int * cnt_3

type state_3 = expr * cnt_3

(* apply_3 : cnt_3 * int -> int *)
let rec apply_3 = function
   | (ID,                  v)                    -> v
   | (OUTER_PLUS(e2, cnt), v1) -> eval_aux_3(e2, INNER_PLUS(v1, cnt))
   | (OUTER_SUBT(e2, cnt), v1) -> eval_aux_3(e2, INNER_SUBT(v1, cnt))
   | (OUTER_MULT(e2, cnt), v1) -> eval_aux_3(e2, INNER_MULT(v1, cnt)
   | (INNER_PLUS(v1, cnt), v2) -> apply_3(cnt, v1 + v2)
   | (INNER_SUBT(v1, cnt), v2) -> apply_3(cnt, v1 - v2)
   | (INNER_MULT(v1, cnt), v2) -> apply_3(cnt, v1 * v2)
```

Defunctionalise!

```
(* eval_aux_2 : state_3 -> int *)
and eval_aux_3 (e, cnt) =
  match e with
   | INT a       -> apply_3(cnt, a)
   | PLUS(e1, e2) -> eval_aux_3(e1, OUTER_PLUS(e2, cnt))
   | SUBT(e1, e2) -> eval_aux_3(e1, OUTER_SUBT(e2, cnt))
   | MULT(e1, e2) -> eval_aux_3(e1, OUTER_MULT(e2, cnt))

(* eval_3 : expr -> int *)
let eval_3 e = eval_aux_3(e, ID)
```

Eureka! Again we have a stack!

```
type tag =
  | O_PLUS of expr
  | I_PLUS of int
  | O_SUBT of expr
  | I_SUBT of int
  | O_MULT of expr
  | I_MULT of int

type cnt_4 = tag list
type state_4 = expr * cnt_4

(* apply_4 : cnt_4 * int -> int *)
let rec apply_4 = function
   | ([],          v)                -> v
   | ((O_PLUS e2) :: cnt, v1) -> eval_aux_4(e2, (I_PLUS v1) :: cnt)
   | ((O_SUBT e2) :: cnt, v1) -> eval_aux_4(e2, (I_SUBT v1) :: cnt)
   | ((O_MULT e2) :: cnt, v1) -> eval_aux_4(e2, (I_MULT v1) :: cnt)
   | ((I_PLUS v1) :: cnt, v2) -> apply_4(cnt, v1 + v2)
   | ((I_SUBT v1) :: cnt, v2) -> apply_4(cnt, v1 - v2)
   | ((I_MULT v1) :: cnt, v2) -> apply_4(cnt, v1 * v2)
```

Eureka! Again we have a stack!

```
(* eval_aux_4 : state_4 -> int *)
and eval_aux_4 (e, cnt) =
  match e with
   | INT a             -> apply_4(cnt, a)
   | PLUS(e1, e2) -> eval_aux_4(e1, O_PLUS(e2) :: cnt)
   | SUBT(e1, e2) -> eval_aux_4(e1, O_SUBT(e2) :: cnt)
   | MULT(e1, e2) -> eval_aux_4(e1, O_MULT(e2) :: cnt)

(* eval_4 : expr -> int *)
let eval_4 e = eval_aux_4(e, [])
```

Eureka! Can combine apply_4 and eval_aux_4

```
type acc =
  | A_INT of int
  | A_EXP of expr

type cnt_5 = cnt_4

type state_5 = cnt_5 * acc

val : step : state_5 -> state_5

val driver : state_5 -> int

val eval_5 : expr -> int
```

Type of an “accumulator” that contains either an int or an expression.

The driver will be clearly tail-recursive …

Rewrite to use driver, accumulator

```
let step_5 = function
  | (cnt,                A_EXP (INT a)) -> (cnt, A_INT a)
  | (cnt,    A_EXP (PLUS(e1, e2))) -> (O_PLUS(e2) :: cnt, A_EXP e1)
  | (cnt,    A_EXP (SUBT(e1, e2))) -> (O_SUBT(e2) :: cnt, A_EXP e1)
  | (cnt,    A_EXP (MULT(e1, e2))) -> (O_MULT(e2) :: cnt, A_EXP e1)
  | ((O_PLUS e2) :: cnt,  A_INT v1) -> ((I_PLUS v1) :: cnt, A_EXP e2)
  | ((O_SUBT e2) :: cnt,  A_INT v1) -> ((I_SUBT v1) :: cnt, A_EXP e2)
  | ((O_MULT e2) :: cnt, A_INT v1) -> ((I_MULT v1) :: cnt, A_EXP e2)
  | ((I_PLUS v1) :: cnt,   A_INT v2) -> (cnt, A_INT (v1 + v2))
  | ((I_SUBT v1) :: cnt,   A_INT v2) -> (cnt, A_INT (v1 - v2))
  | ((I_MULT v1) :: cnt,   A_INT v2) -> (cnt, A_INT (v1 * v2))
  | ([],                     A_INT v) -> ([], A_INT v)

let rec driver_5 = function
   | ([], A_INT v) -> v
   | state         -> driver_5 (step_5 state)

let eval_5 e = driver_5([], A_EXP e)
```

Eureka! There are really two independent stacks here --- one for "expressions" and one for values

```
type directive =
  | E of expr
  | DO_PLUS
  | DO_SUBT
  | DO_MULT

type directive_stack = directive list

type value_stack = int list

type state_6 = directive_stack * value_stack

val step_6 : state_6 -> state_6

val driver_6 : state_6 -> int

val exp_6 : expr -> int
```

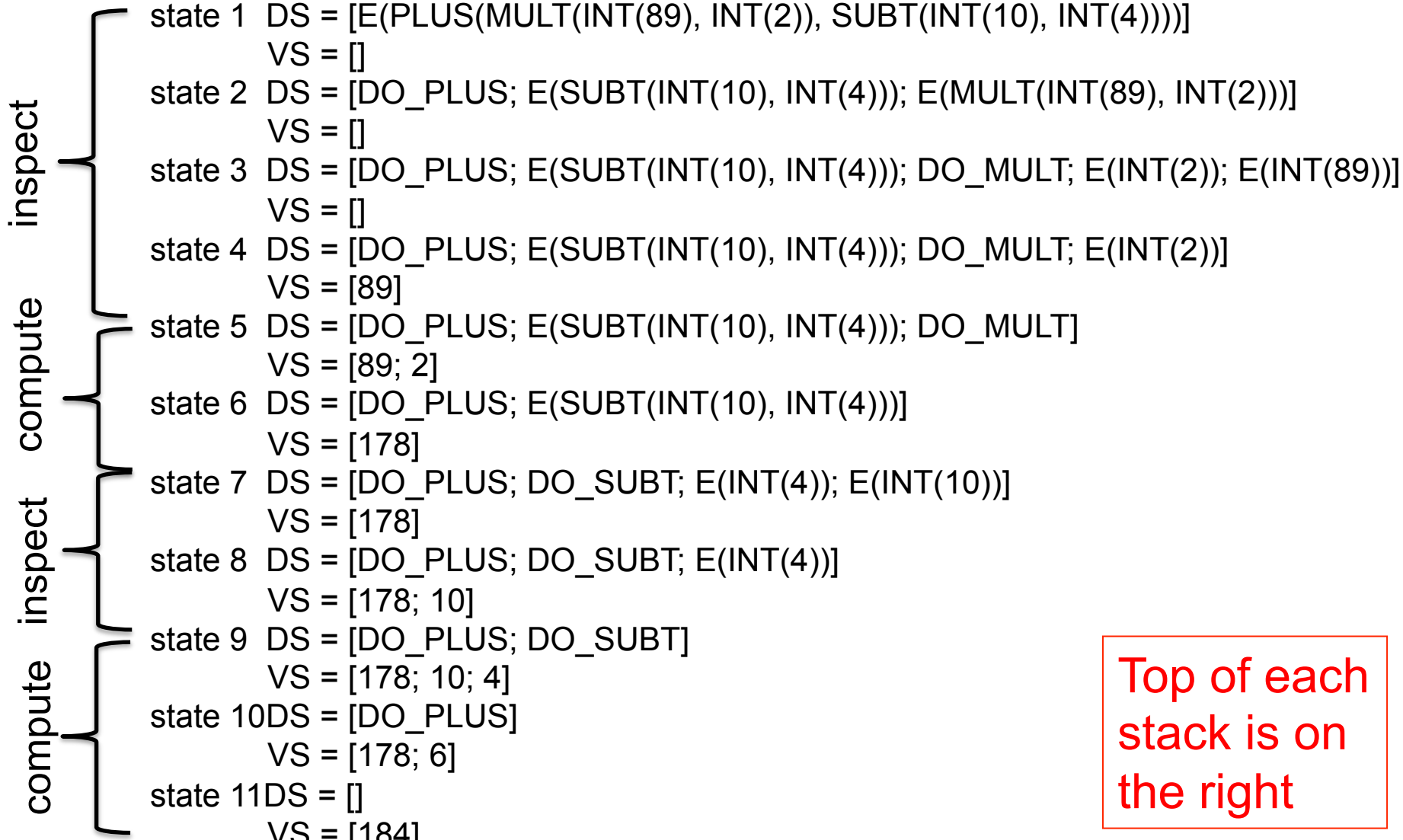
The state is now two stacks!

Split into two stacks

```
let step_6 = function
    | (E(INT v) :: ds,              vs) -> (ds, v :: vs)
    | (E(PLUS(e1, e2)) :: ds,    vs) -> ((E e1) :: (E e2) :: DO_PLUS :: ds, vs)
    | (E(SUBT(e1, e2)) :: ds,    vs) -> ((E e1) :: (E e2) :: DO_SUBT :: ds, vs)
    | (E(MULT(e1, e2)) :: ds,    vs) -> ((E e1) :: (E e2) :: DO_MULT :: ds, vs)

    | (DO_PLUS :: ds, v2 :: v1 :: vs) -> (ds, (v1 + v2) :: vs)
    | (DO_SUBT :: ds, v2 :: v1 :: vs) -> (ds, (v1 - v2) :: vs)
    | (DO_MULT :: ds, v2 :: v1 :: vs) -> (ds, (v1 * v2) :: vs)
    | _ -> failwith "eval : runtime error!"

let rec driver_6 = function
    | ([], [v]) -> v
    | state     -> driver_6 (step_6 state)

let eval_6 e = driver_6 ([E e], [])
```

An eval_6 trace

```
e = PLUS(MULT(INT 89, INT 2), SUBT(INT 10, INT 4))
```



Top of each stack is on the right

Key insight

This evaluator is interleaving two distinct computations:

(1) decomposition of the input expression into sub-expressions
(2) the computation of +, -, and *.

Idea: why not do the decomposition BEFORE the computation?

Key insight: An interpreter can (usually) be **refactored** into a translation (compilation!) followed by a lower-level interpreter.

Interpret_higher (e) = interpret_lower(compile(e))

Note : this can occur at many levels of abstraction: think of machine code being interpreted in micro-code …

Refactor --- compile!

```
(* low-level instructions *)
type instr =
  | Ipush of int
  | Iplus
  | Isubt
  | Imult

type code = instr list

type state_7 = code * value_stack

(* compile : expr -> code *)
let rec compile = function
   | INT a             -> [Ipush a]
   | PLUS(e1, e2)  -> (compile e1) @ (compile e2) @ [Iplus]
   | SUBT(e1, e2)  -> (compile e1) @ (compile e2) @ [Isubt]
   | MULT(e1, e2)  -> (compile e1) @ (compile e2) @ [Imult]
```

> Never put off till run-time what you can do at compile-time.
>
> -- David Gries

Evaluate compiled code.

```
(* step_7 : state_7 -> state_7 *)
let step_7 = function
   | (Ipush v :: is,        vs) ->  (is, v :: vs)
   | (Iplus :: is, v2::v1::vs) -> (is, (v1 + v2) :: vs)
   | (Isubt :: is, v2::v1::vs) -> (is, (v1 - v2) :: vs)
   | (Imult :: is, v2::v1::vs) -> (is, (v1  * v2) :: vs)
   | _ -> failwith "eval : runtime error!"

let rec driver_7 = function
    | ([], [v]) -> v
    | _ -> driver_7 (step_7 state)

let eval_7 e = driver_7  (compile e, []) l
```

An eval_7 trace

inspect

```
compile (PLUS(MULT(INT 89, INT 2), SUBT(INT 10, INT 4)))
  = [push 89; push 2; mult; push 10; push 4; subt; plus]
```

compute

```
state 1   IS = [add; sub; push 4; push 10; mul; push 2; push 89]
          VS = []
state 2   IS = [add; sub; push 4; push 10; mul; push 2]
          VS = [89]
state 3   IS = [add; sub; push 4; push 10; mul]
          VS = [89; 2]
state 4   IS = [add; sub; push 4; push 10]
          VS = [178]
state 5   IS = [add; sub; push 4]
          VS = [178; 10]
state 6   IS = [add; sub]
          VS = [178; 10; 4]
state 7   IS = [add]
          VS = [178; 6]
state 8   IS = []
          VS = [184]
```

Top of each stack is on the right

Interp_0.ml → interp_1.ml → interp_2.ml

The derivation from eval to compile+eval_7 can be used as a guide to a derivation from Interpreter 0 to interpreter 2.

1. Apply CPS to the code of Interpreter 0
2. Defunctionalise
3. Arrive at interpreter 1, which has a single continuation stack containing expressions, values and environments
4. Spit this stack into two stacks : one for instructions and the other for values and environments
5. Refactor into compiler + lower-level interpreter
6. Arrive at interpreter 2.

Taking stock

Starting from a direct implementation of Slang/L3 semantics, we have **DERIVED** a Virtual Machine in a step-by-step manner. The correctness of aach step is (more or less) easy to check.