# Compiler Construction
# Lent Term 2014
# Lecture 14

- **Slang.3 language (subset of SPL's L3)**
- **VSM.2 : Stack machine with heap**
- **Examples**

**Timothy G. Griffin**
**tgg22@cam.ac.uk**
**Computer Laboratory**
**University of Cambridge**

# The Source : Slang.3 = an enhanced subset of L3

From Semantics of Programming Languages:

Types:

$$T \quad ::= \quad \text{int} \mid \text{bool} \mid \text{unit} \mid T_1 \to T_2 \mid T_1 * T_2 \mid T_1 + T_2 \mid \{lab_1{:}T_1, .., lab_k{:}T_k\} \mid T \ \text{ref}$$

Expressions

$$\begin{array}{rcl} e & ::= & n \mid b \mid e_1 \ op \ e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \\ & & e_1 := e_2 \mid !e \mid \text{ref } e \mid \ell \mid \\ & & \text{skip} \mid e_1; e_2 \mid \\ & & \text{while } e_1 \text{ do } e_2 \mid \\ & & \text{fn } x{:}T \Rightarrow e \mid e_1 \ e_2 \mid x \mid \\ & & \text{let val } x{:}T = e_1 \text{ in } e_2 \text{ end} \mid \\ & & \text{let val rec } x{:}T_1 \to T_2 = (\text{fn } y{:}T_1 \Rightarrow e_1) \text{ in } e_2 \text{ end} \mid \\ & & (e_1, e_2) \mid \#1 \ e \mid \#2 \ e \mid \\ & & \text{inl } e{:}T \mid \text{inr } e{:}T \mid \\ & & \text{case } e \text{ of inl } (x_1{:}T_1) \Rightarrow e_1 \mid \text{inr } (x_2{:}T_2) \Rightarrow e_2 \mid \\ & & \{lab_1 = e_1, .., lab_k = e_k\} \mid \#lab \ e \end{array}$$

Not implemented : records, disjoint union, tuples
(left as exercises for you …)

# Slang.3 Examples

slang3/examples/rem2.slang

```
%
% m = (m div n) * n + rem(m, n)
%
% Like rem1.slang, but this version tests that 0 < n.
%
fun rem (m : int, n : int) : int =
    fun aux(m: int) : int = if (m >= n) then aux(m - n) else m
    in
         if (n >= 1) then aux(m) else 0
    end
in
   let x1 : int = read()
   in let x2 : int = read()
      in
            print(rem(x1, x2))
      end
   end
end
```

# Slang.3 Examples

slang3/examples/counter.slang

```
fun mk_counter (i : int) : unit -> int =
    let c : int ref = ref (i)
    in fun inc () : int =
          let x : int = !c
          in (c := !c + 1; x) end
       in inc end
    end
in
   let v : int  = read()
   in let g : unit -> int = mk_counter (v)
      in
          (print (v);
           print (g ());
           print (g ());
           print (g ());
           print (g ()))
      end
   end
end
```

# Slang.3 Concrete Syntax

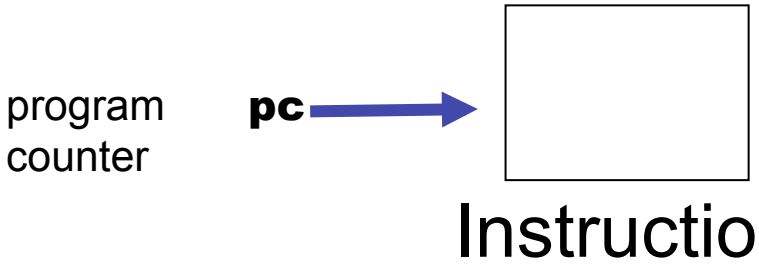
```
E ::=  A
     | while E do E end
     | fn ID : T => E end
     | if E then E else E
     | let ID : T = E in E end
     | fun ID ( PL ) : T = E in E end
     | fun ID () : T = E in E end

A ::= ID | true | false | n | skip | ( E ) | ( S1 )
    | A () | A (EL) | !A | ref A | not A | -A
    | A + A | A - A | A * A | A = A | A >= A | A := A

S1 ::= E ; S2

S2 ::= E ; S1 | E

EL := E, EL | E

PL := ID : T | ID : T, PL

PT : T * T | T * PT

T ::= ( T ) | int | unit | bool | T -> T | (PT) -> T | T ref
```

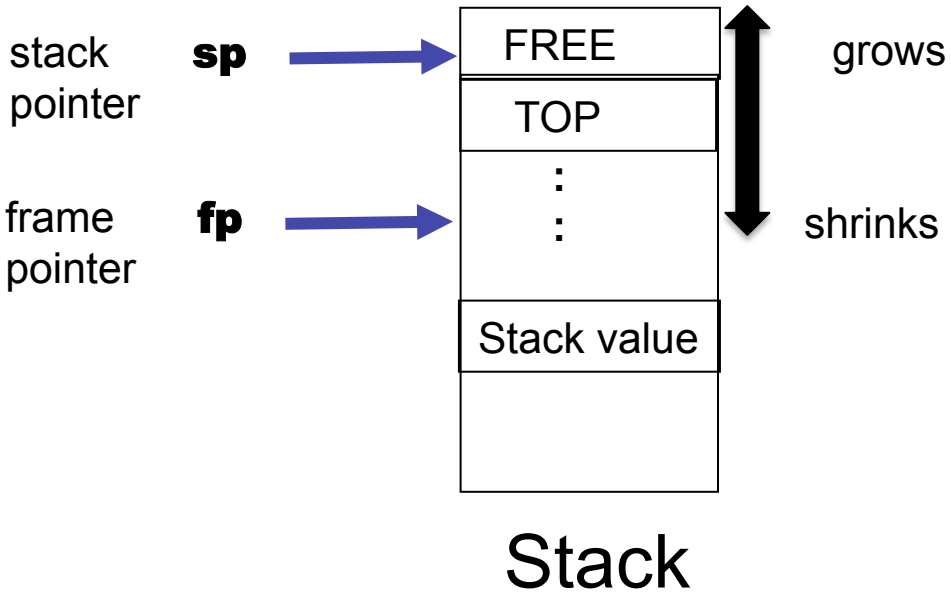
Some modifications:
Add “end” to while and fn.
Drop “val” and “rec”.
Functions have 0 or more arguments.

# Slang.3 Abstract Syntax

```
type var = string
datatype oper = Plus | Mult | Subt | GTEQ | EQ
datatype unary_oper = Neg | Not

datatype type_expr =
         TEint
       | TEref of type_expr
       | TEunit
       | TEbool
       | TEfunc of (type_expr list) * type_expr

datatype expr =
         Skip
       | Integer of int
       | Boolean of bool
       | UnaryOp of unary_oper * expr
       | Op of expr * oper * expr
       | Assign of expr * expr
       | Deref of expr
       | Ref of expr
       | Seq of expr * expr
       | If of expr * expr * expr
       | While of expr * expr
       | Var of var
       | Fn of var * type_expr * expr * (type_expr option)
       | App of expr * expr list
       | Let of var * type_expr * expr * expr
       | Letrec of var * (var * type_expr) list * type_expr * expr * expr
```

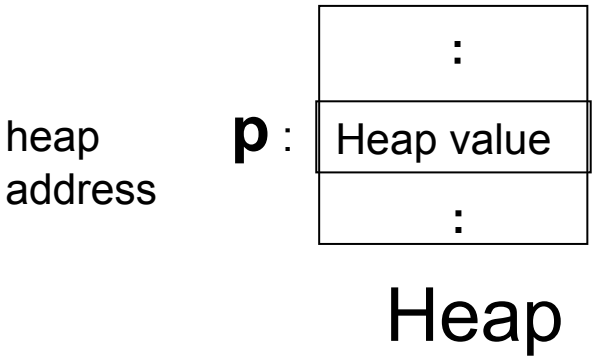
# Target: VSM.2



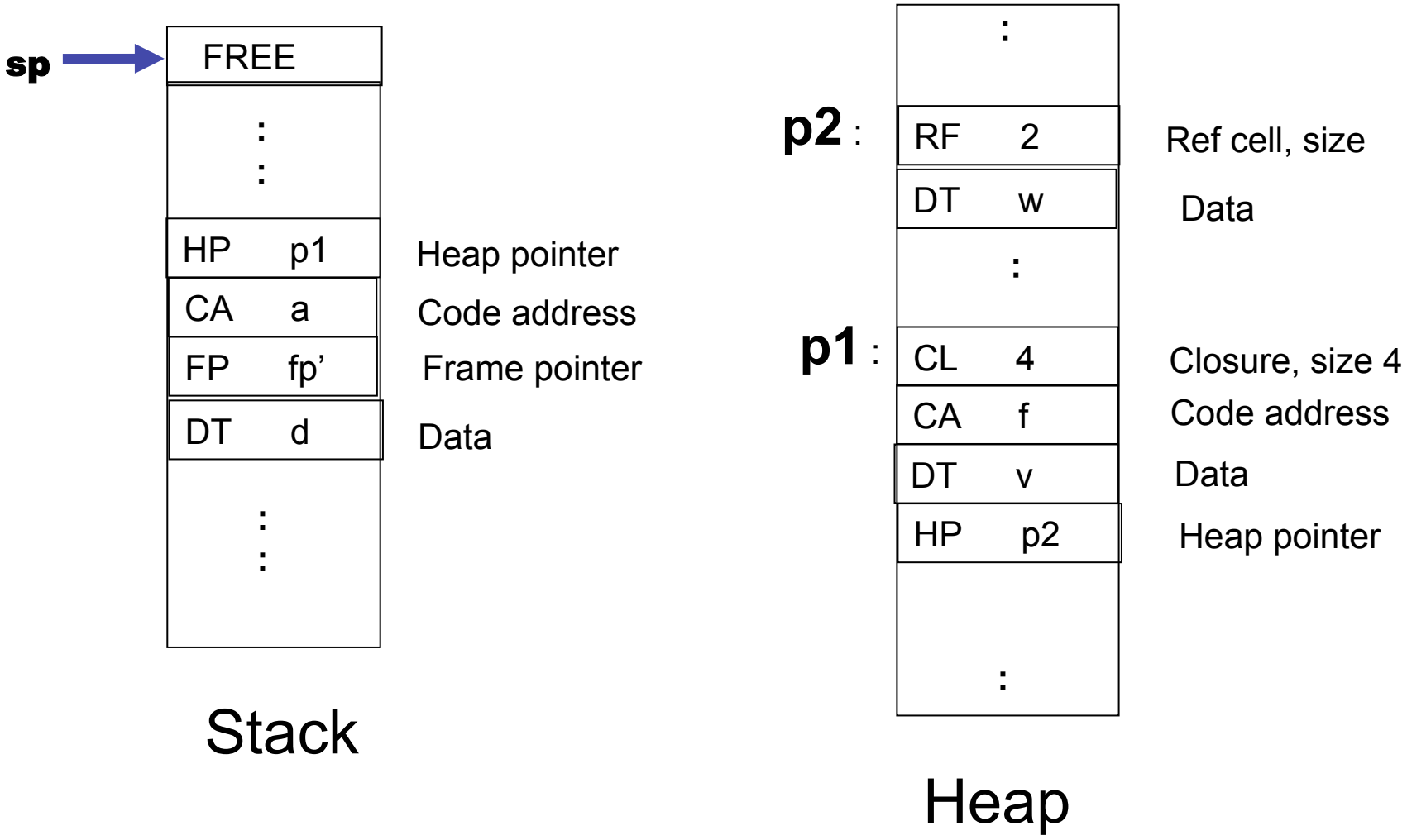




Stack and heap values have the form (TY, value). The type TY indicates if the value v is a pointer into the heap or not.

For clarity we will will add additional types.

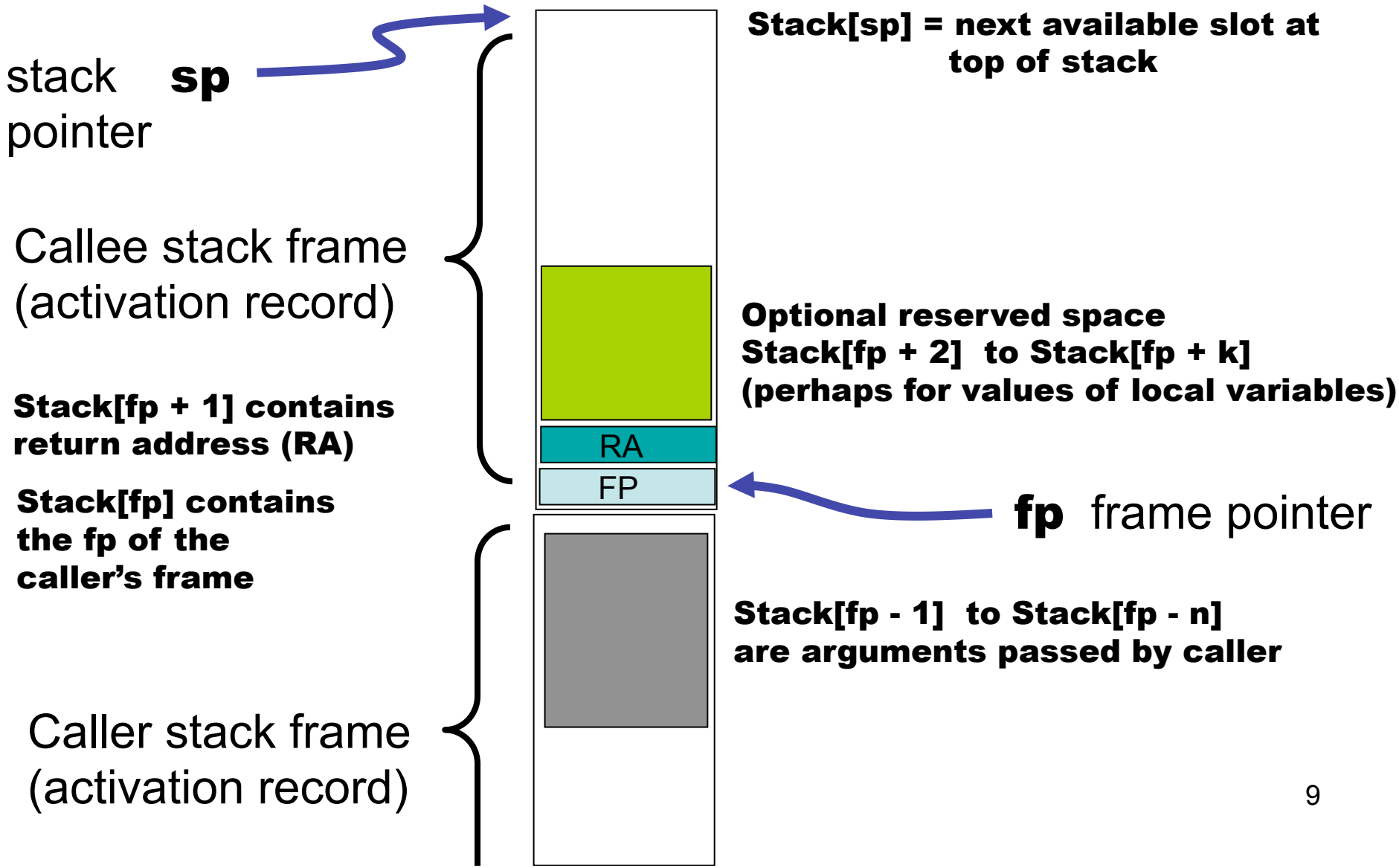


# Stack and Heap Types

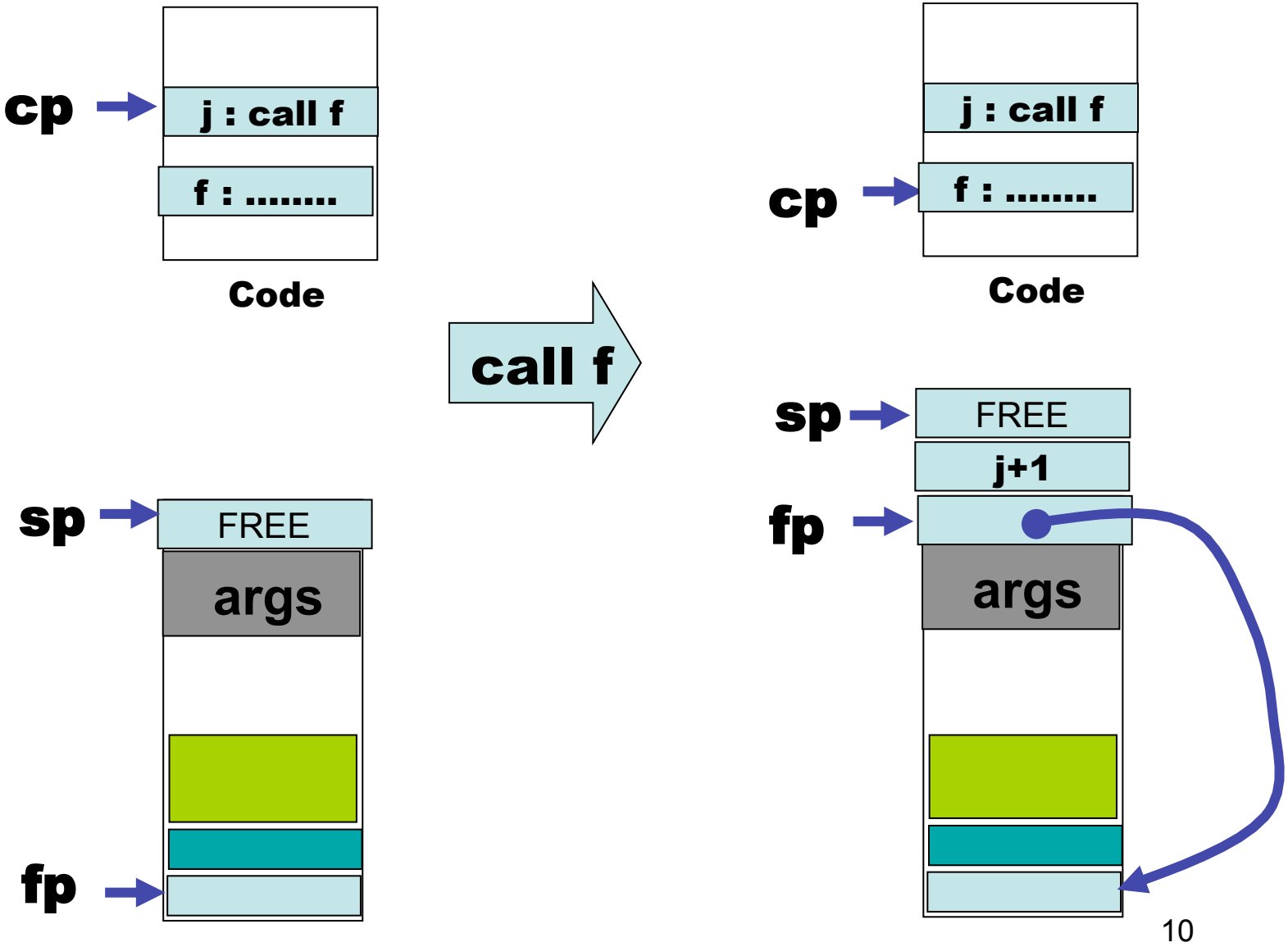



Again: only "one bit" really needed for types (pointer/not pointer).

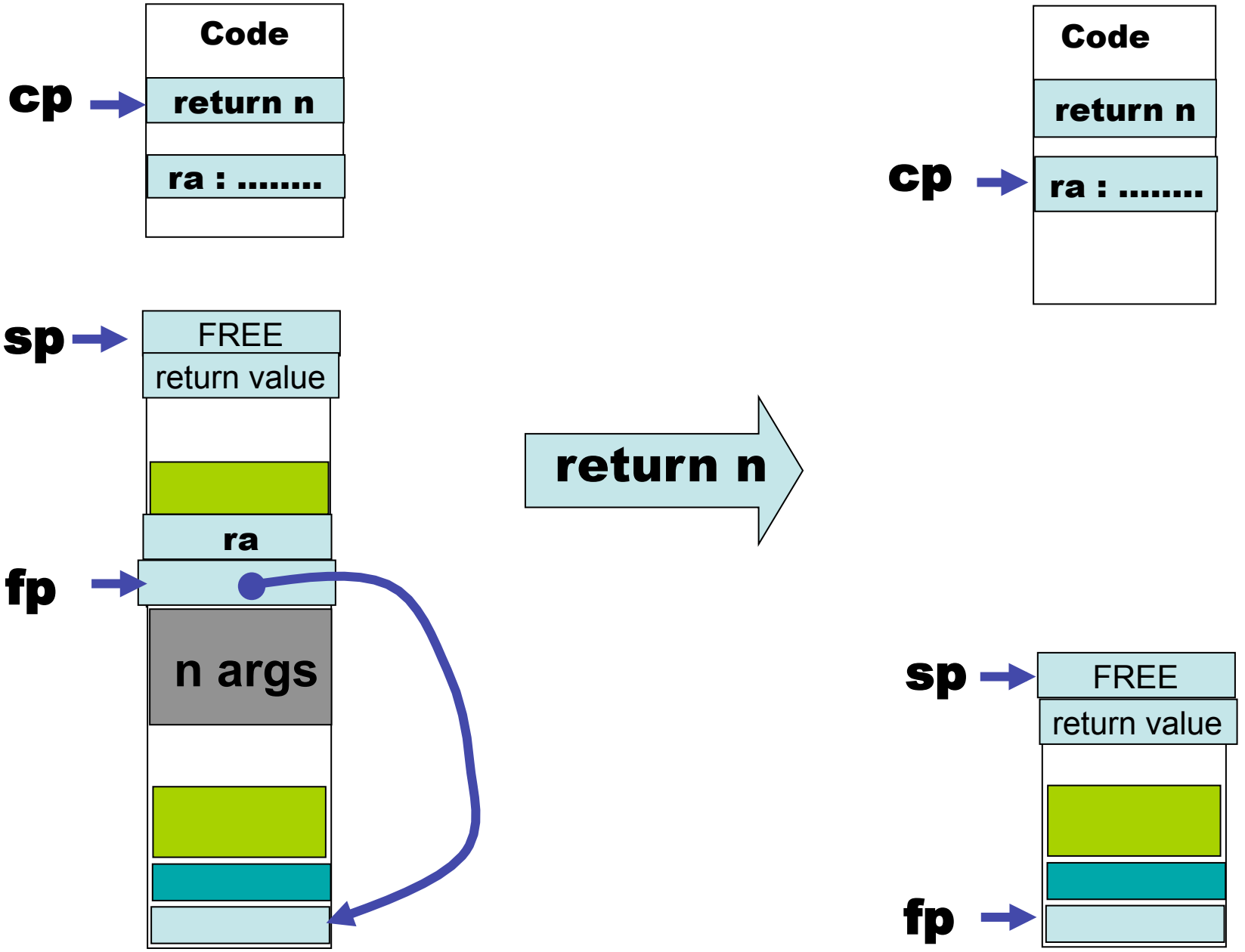
# Recall : VSM Call Stack



# Calling a “known function” f



# return n


Code
cp
return n
ra : .....
sp
FREE
return value
ra
fp
n args
return n
Code
return n
cp
ra : .....
sp
FREE
return value
fp

# Why is “one bit” needed?

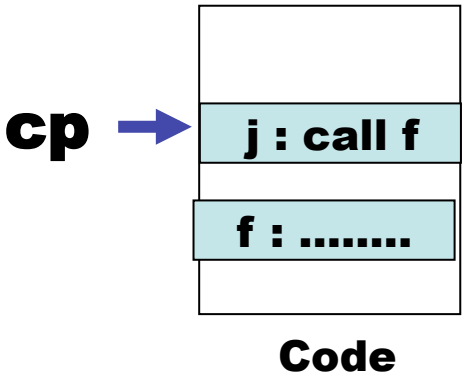
It may be that most functions are not closures, but are “known” at compile time. We don’t want to pay the overhead of “closures on the heap” for all functions/procedures.

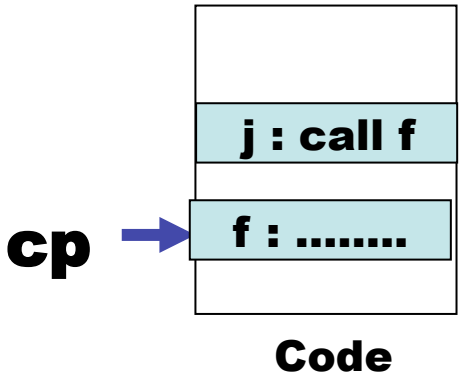
How can we compile the following expression?

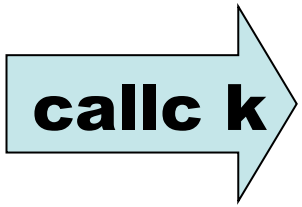
```
e(e1, e2)
```

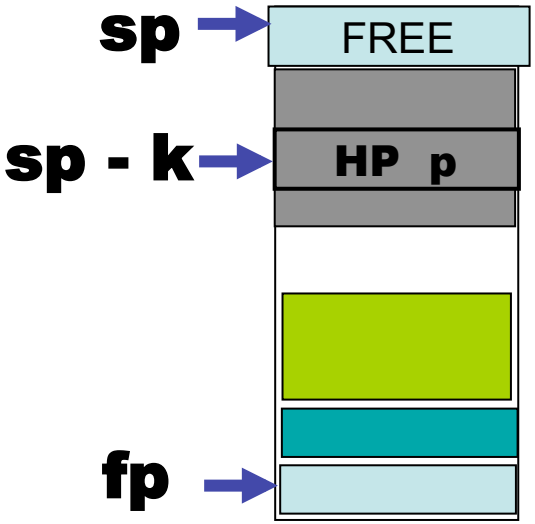
We do not know until run-time the result of evaluating `e` will require a “normal” function call or a call through a closure on the heap.

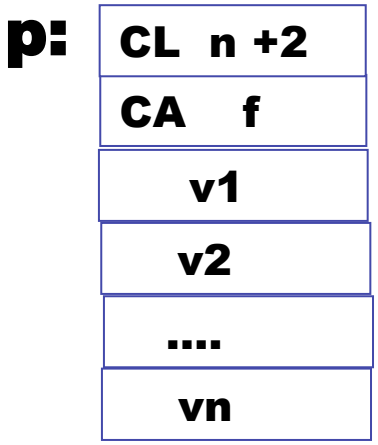
# callc k (CASE 1)

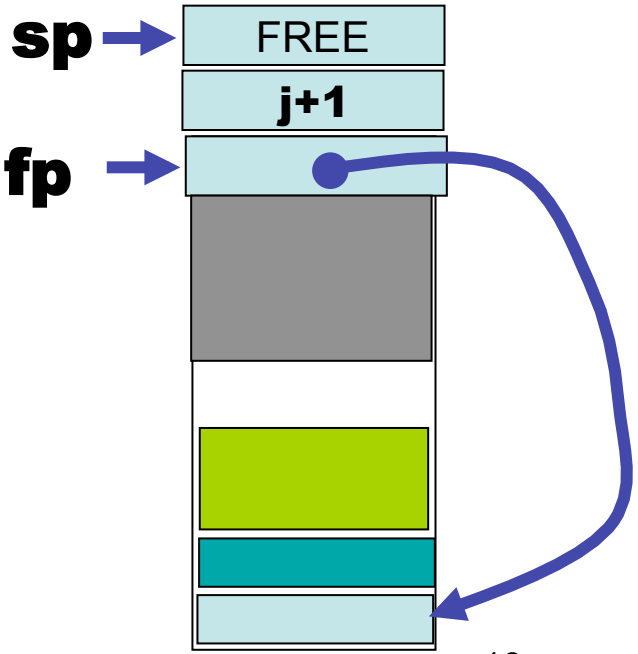


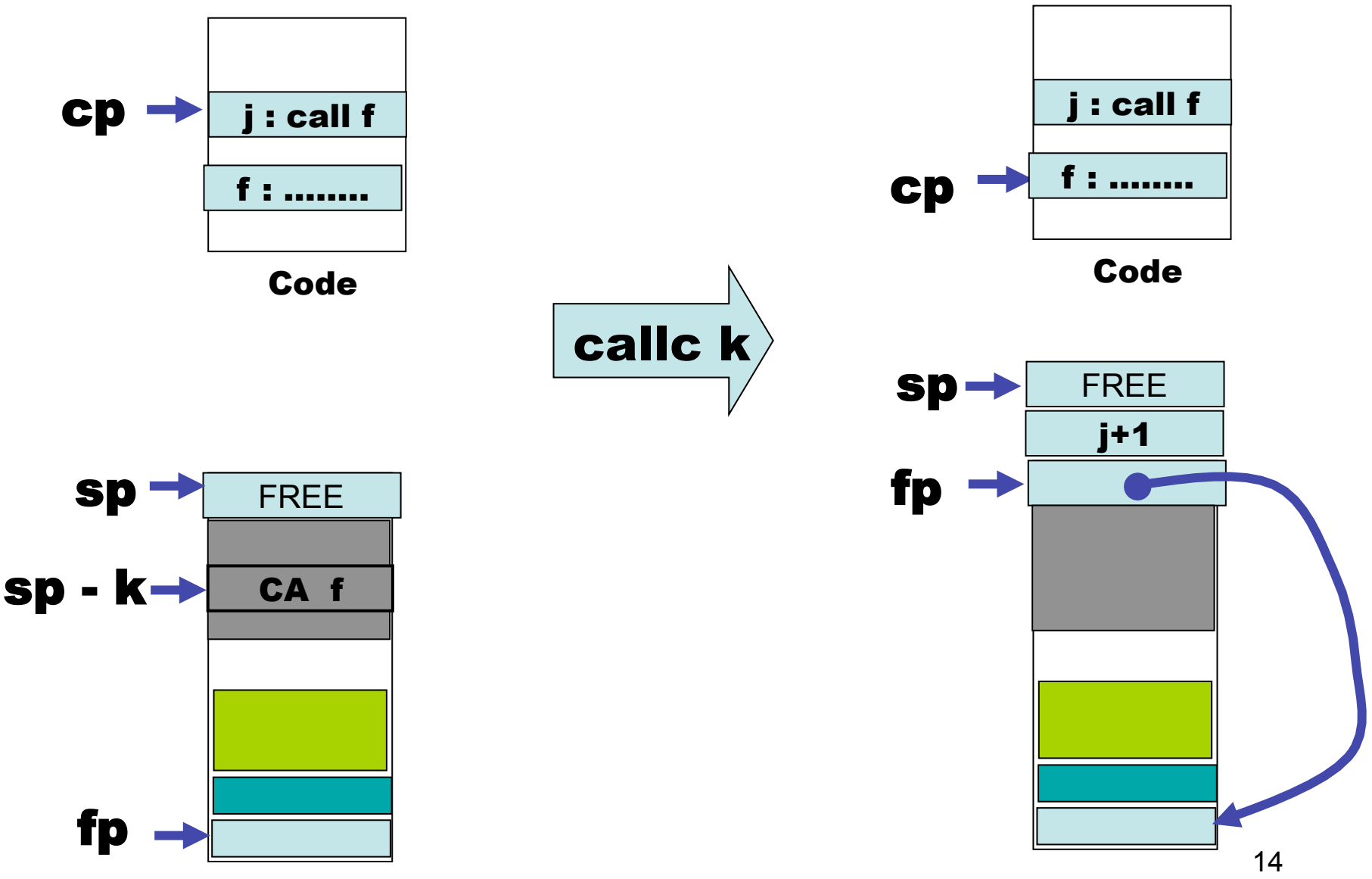



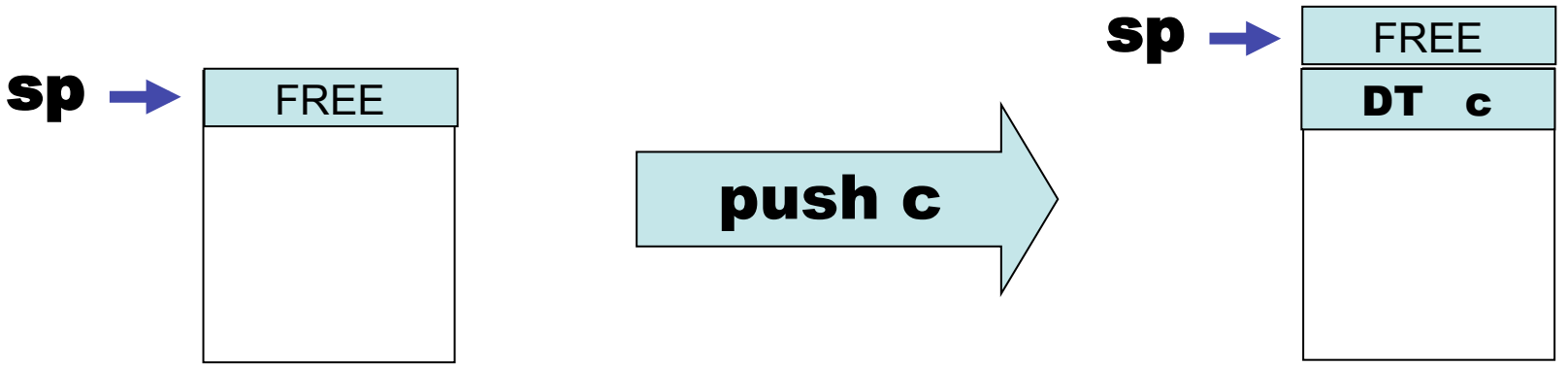



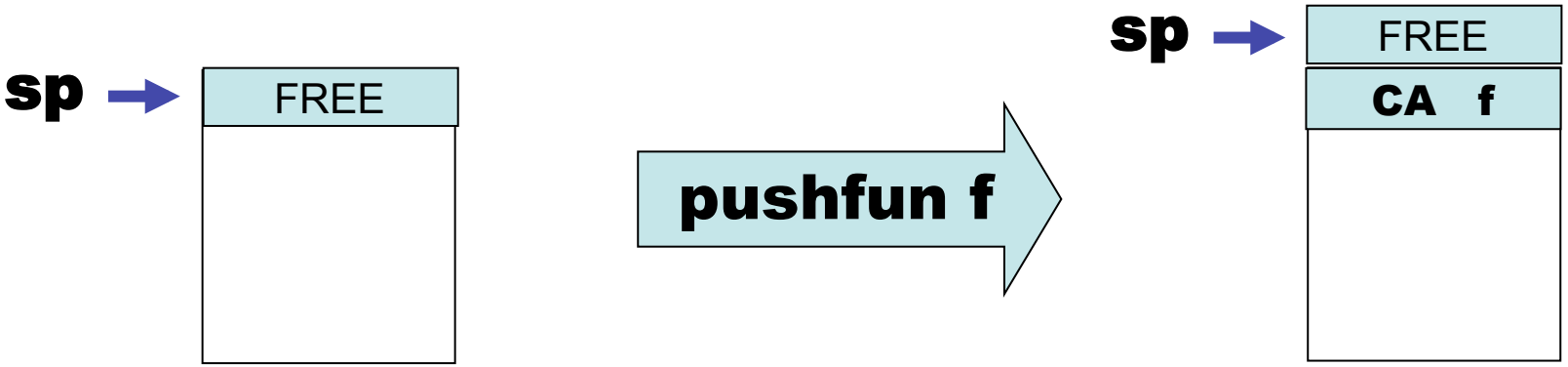



# callc k (CASE 2)

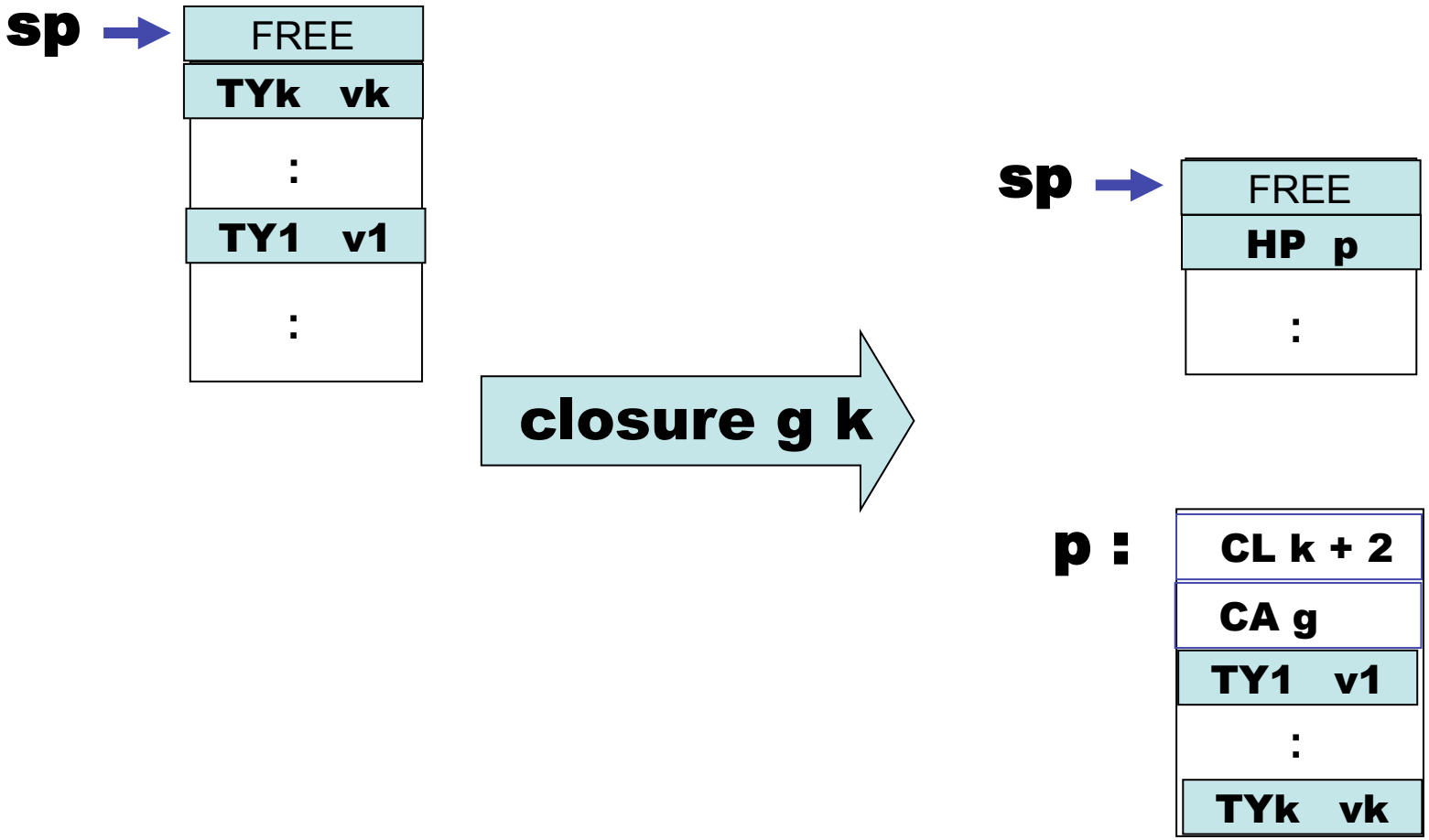


# push c, pushfun f



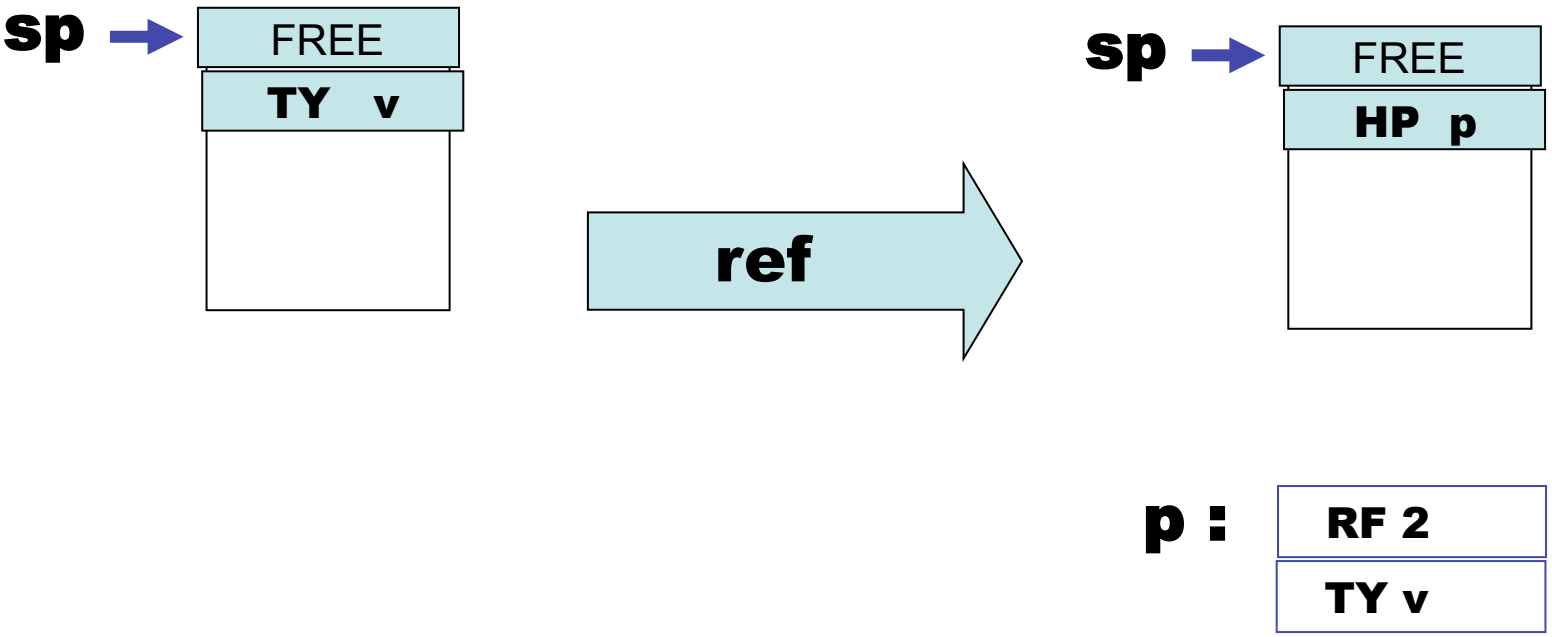
# closure g k




p = first free in heap

p + k +2 = first free in heap

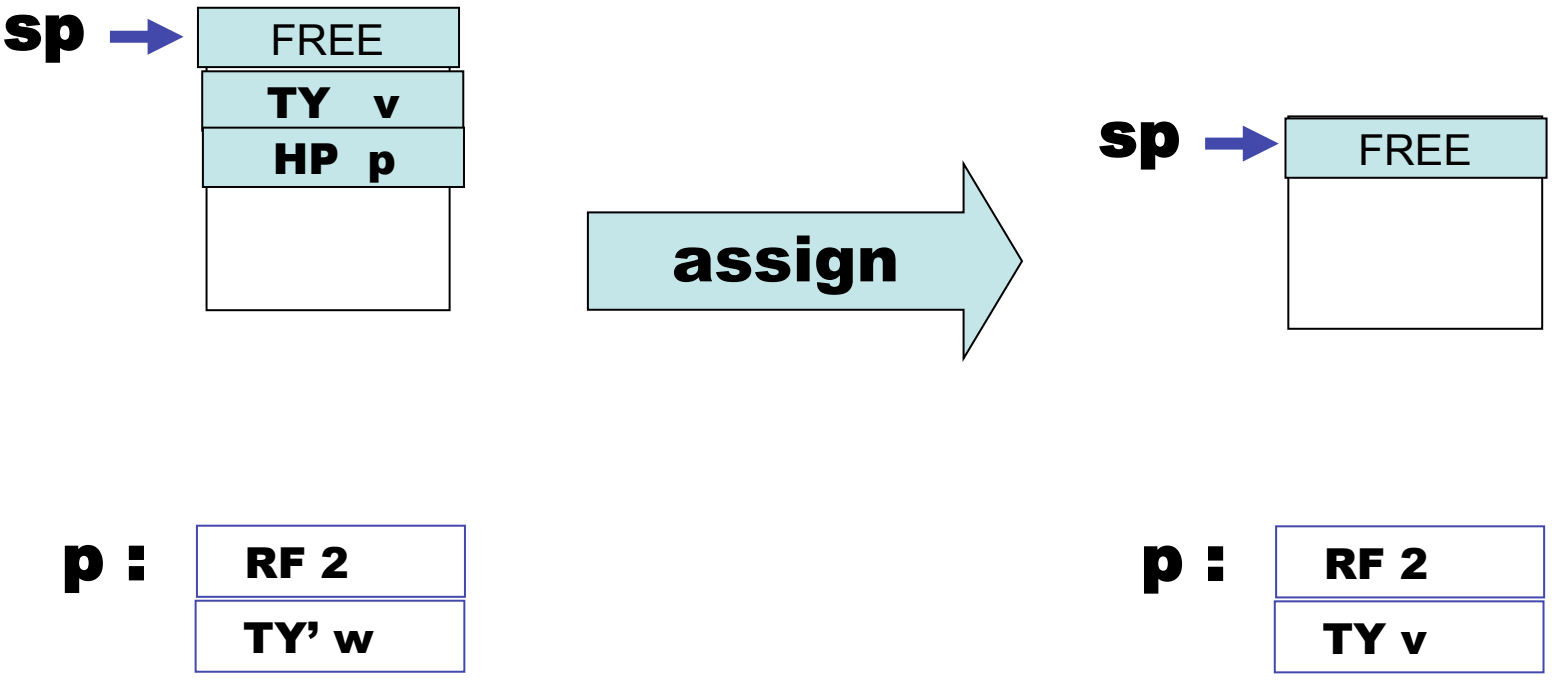
# ref




p = first free in heap

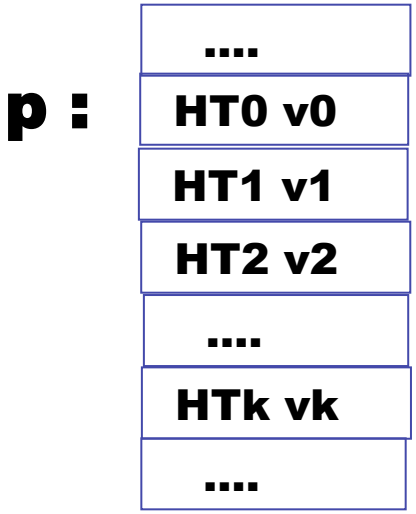
p + 2 = first free in heap

# assign




If the source code is “type correct” then it is probably the case that TY’ = TY.

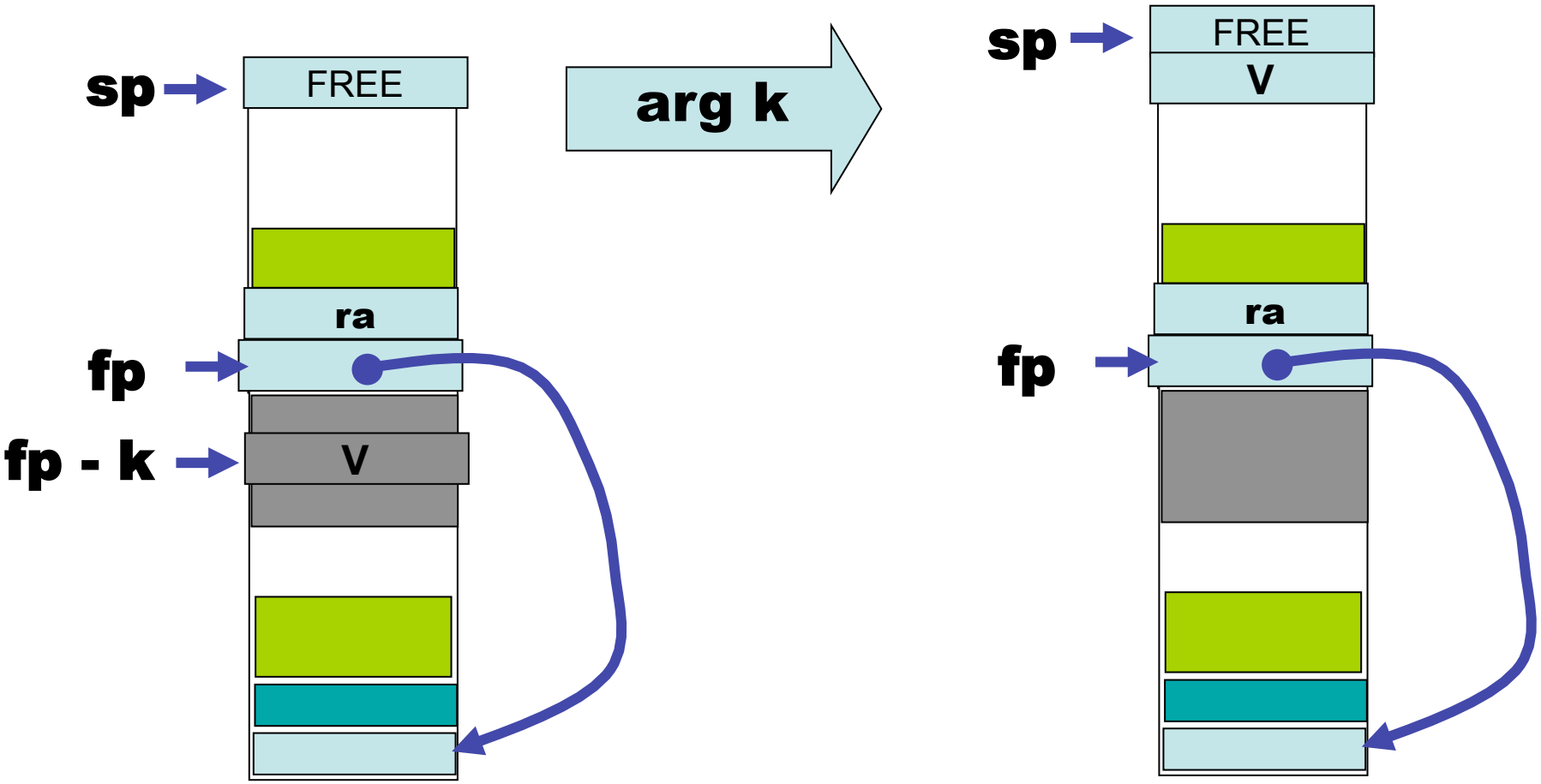
# deref k


In Heap

**Will use the same operation to deref a ref cell and a closure.**

# arg k



# AST_vsm_assembler.sml

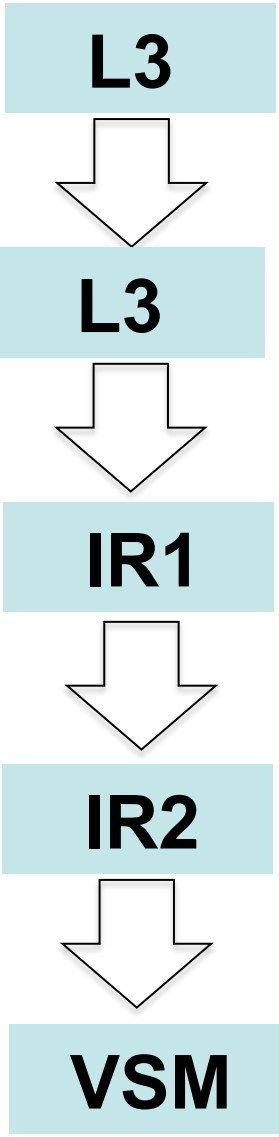
```
type offset = int
type label = string
type constant = int

datatype vsm_operation =
         VSM_Comment of string      (* Comment string                          *)
       | VSM_Label of label         (* symbolic location (code address)        *)
       | VSM_Nop                    (* do nothing                              *)
       | VSM_Push of constant       (* push a simple data value                *)
       | VSM_PushFun of label       (* push a function/procedure address       *)
       | VSM_Arg_Load of offset     (* push value at fp - offset               *)
       | VSM_AllocateLocals of int  (* allocate stack space for local values   *)
       | VSM_Load of offset         (* push local value at fp + offset         *)
       | VSM_Store of offset        (* pop top to fp + offset                  *)
       | VSM_Pop                    (* pop top of stack                        *)
       | VSM_Add                    (* replace top 2 values with sum           *)
       | VSM_Sub                    (* replace top 2 values with difference    *)
       | VSM_Mul                    (* replace top 2 values with product       *)
       | VSM_Hlt                    (* stop the machine, I want to get off     *)
       | VSM_Jmp of label           (* jump to label                           *)
       | VSM_Ifz of label           (* jump to label if top is zero, pop       *)
       | VSM_Ifp of label           (* jump to label if top is positive, pop   *)
       | VSM_Ifn of label           (* jump to label if top is negative, pop   *)
       | VSM_Pri                    (* print stack-top, then pop it            *)
       | VSM_Rdi                    (* read integer from stdin, push it        *)
       | VSM_Assign                 (* assign via pointer to ref cell in heap *)
       | VSM_Deref of int           (* replace top with dereferenced value     *)
       | VSM_Ref                    (* create ref cell on heap                 *)
       | VSM_Call of label          (* call known function                     *)
       | VSM_Call_Closure of int    (* call closure                            *)
       | VSM_Return of int          (* return and clear args off stack         *)
       | VSM_Closure of label * int (* allocate closure on heap, push pointer *)
```

# command-line

- **slang3 -vsm examples/file.slang**
  compile file.slang to VSM.2 to binary object file examples/file.vrmo
- **slang3 examples/file.slang**
  same as above (VSM.2 is the default)
- **slang3 -v examples/file.slang**
  same as above, but with verbose output at each stage of compilation

- **vsm2 examples/file.vsmo**
  run VSM.0 on bytecode file
- **vsm2 -v examples/file.vsmo**
  same as above, but with verbose output
- **vsm2 -s examples/file.vsmo**
  just print the bytecode

# Mind The Gap




Type-checked Abstract Syntax Tree (AST)

Alpha converted (all bound variables made unique)

Closure converted. All functions/procedures at top-level.

High-level control removed (“almost flattened”). Can now target either register-oriented or stack-oriented machine.

Target VSM. Fully flattened.

# Alpha Conversion of rem2.slang

```
fun _1_rem (_2_m:int, _3_n:int ) : int =
   fun _4_aux (_5_m:int ) : int =
      if (_5_m >= _3_n) then _4_aux ((_5_m - _3_n)) else _5_m
    in
      if (_3_n >= 1) then _4_aux (_2_m) else 0
    end
 in
    let _6_x1 : int = read ()
    in let _7_x2 : int = read ()
       in
          print (_1_rem (_6_x1, _7_x2))
       end
    end
 end
```

Later stages of compiler will not have to worry about name clashes.

Variable X is transformed to _n_X for some n.
(The variable name X is preserved to enhance debugging ….).

We could have used *De Bruijn indices …*

# IR1

```
datatype var_kind =
       IR1_ArgVar of var
     | IR1_LetVar of var
     | IR1_EnvVar of var * int

datatype expr =
         IR1_Skip
       | IR1_Var of var_kind
       | IR1_KnownFun of var
       | IR1_Integer of int
       | IR1_Boolean of bool
       | IR1_UnaryOp of unary_oper * expr
       | IR1_Op of expr * oper * expr
       | IR1_Assign of expr * expr
       | IR1_Deref of expr
       | IR1_Ref of expr
       | IR1_Seq of expr * expr
       | IR1_If of expr * expr * expr
       | IR1_While of expr * expr
       | IR1_App of expr * expr list
       | IR1_Closure of var * (var_kind list)
       | IR1_Let of var * expr * expr

datatype ir0_function = IR1_Letrec of var * (var list) * expr

type program = expr * (ir0_function list)
```

# Example of L3 to IR1 translation

```
fun _1_rem (_2_m:int, _3_n:int ) : int =
   fun _4_aux (_5_m:int ) : int =
      if (_5_m >= _3_n) then _4_aux ((_5_m - _3_n)) else _5_m
    in  if (_3_n >= 1) then _4_aux (_2_m) else 0 end
 in let _6_x1 : int = read ()
    in let _7_x2 : int = read ()
       in print (_1_rem (_6_x1, _7_x2)) end
    end
 end
```

rem2.slang

```
let _6_x1 = KNOWN(read) ()
 in let _7_x2 = KNOWN(read) ()
    in KNOWN(print) (KNOWN(_1_rem) (_6_x1, _7_x2)) end
 end

fun _1_rem (_2_m, _3_n ) =
   let _9_CL = CLOSURE(_4_aux, [_3_n])
   in  if (_3_n >= 1) then _9_CL (_2_m) else 0 end;

fun _4_aux (_8_ENV, _5_m ) =
   if (_5_m >= _8_ENV[1])
    then KNOWN(_4_aux) (_8_ENV, (_5_m - _8_ENV[1]))
    else _5_m;
```