Compiler Construction

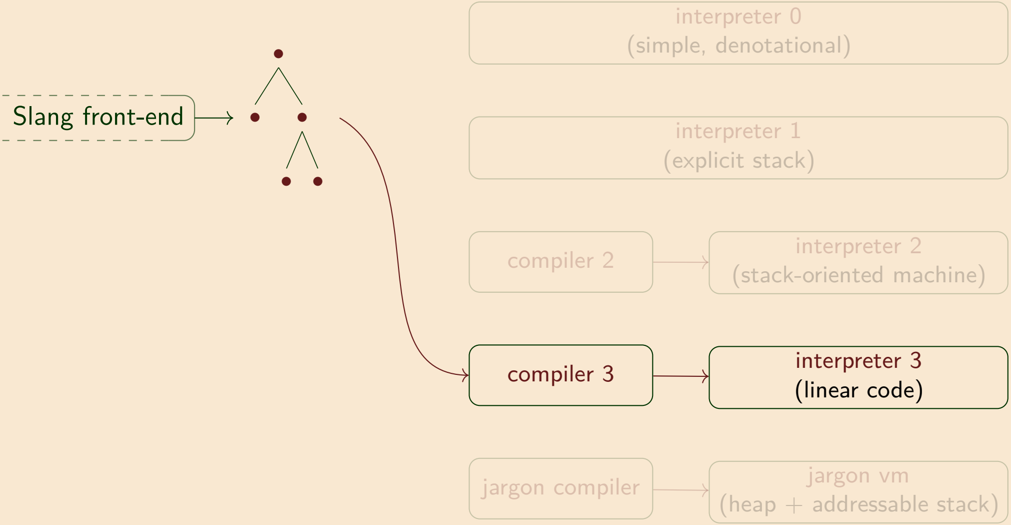
Lecture 11: The Jargon VM

Jeremy Yallop

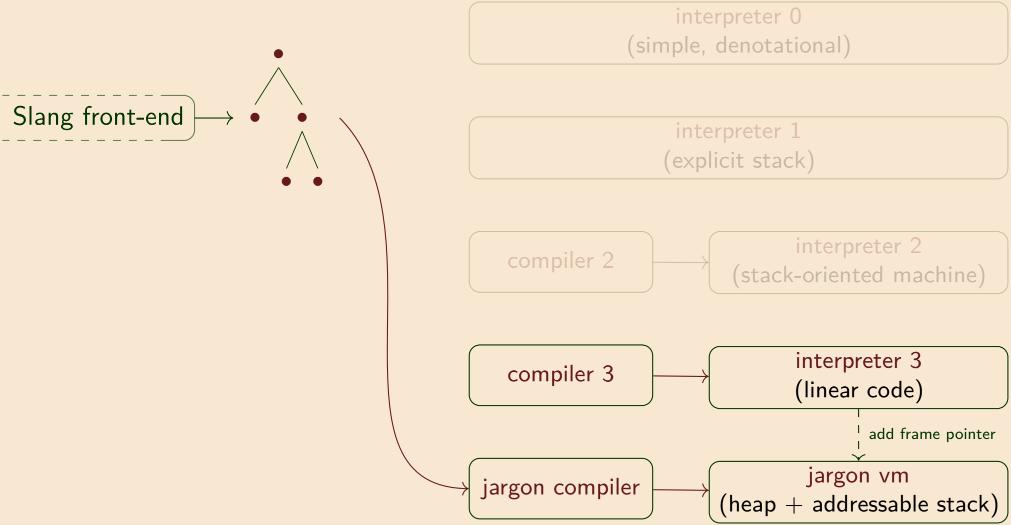
jeremy.yallop@cl.cam.ac.uk

Lent 2023

Reminder: the derivation



Reminder: the derivation



Slang front-end

interpreter 0
(simple, denotational)

interpreter 1
(explicit stack)

compiler 2

interpreter 2
(stack-oriented machine)

compiler 3

interpreter 3
(linear code)

add frame pointer

jargon compiler

jargon vm
(heap + addressable stack)

Jargon VM

Deriving the Jargon VM (interpreter 4)

Three changes to interpreter 3:

Addressable stack

Replace variable lookup by a static offset from a **frame pointer** or closure

Closure representation

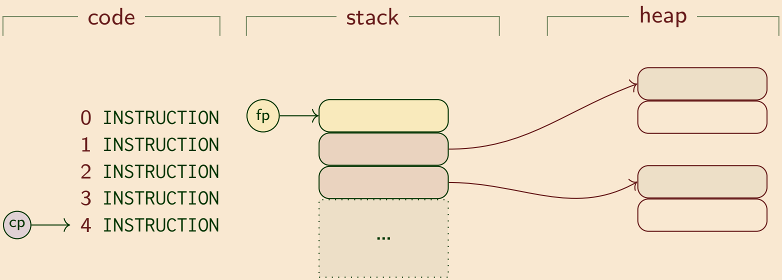
Optimise the **representation of closures** to contain only a code pointer and values for the free variables of the closure

Simple stack values

Restrict values on stack to be simple (ints, bools, heap addresses, etc).
Move complex data to the **heap**

(How might things look different in a language without first-class functions? In a language with multiple arguments to function calls?)

Jargon Virtual Machine



code

stack

heap

```
0 INSTRUCTION
1 INSTRUCTION
2 INSTRUCTION
3 INSTRUCTION
4 INSTRUCTION
```

cp

fp

...

cp code pointer (to next instruction)

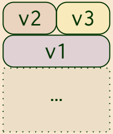
fp frame pointer (to current activation frame)

not shown: stack pointer, heap limit

The stack in interpreter 3

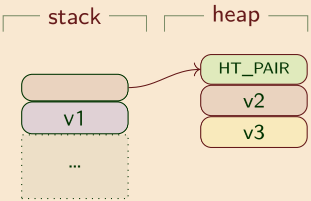
> *All problems in computer science can be solved by another level of indirection, except of course for the problem of too many indirections.*
>
> — *David Wheeler*

Problem: Interpreter 3 stack elements not fixed size



Virtual machines (JVM, etc) typically restrict stack elements to have fixed size

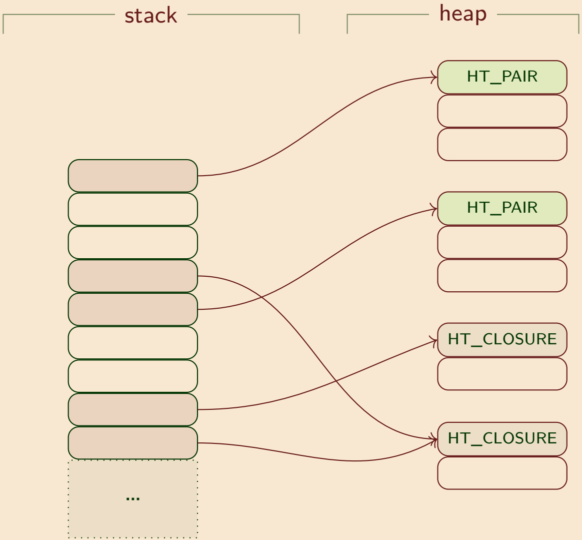
Solution: put the data in the **heap**



Place pointers to heap on stack

Jargon VM: the stack and the heap

Some stack elements represent pointers into the heap:



Instructions

Small changes to instructions

Interpreter 3

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

Jargon VM

```
type instruction =
| PUSH of stack_item                        (* ! *)
| LOOKUP of value_path                      (* ! *)
| UNARY of Ast.unary_oper
| OPER of Ast.oper
| SWAP
| POP
                                            (* ! *)
| FST
| SND
| APPLY
| RETURN
| MK_PAIR
| MK_CLOSURE of location * int              (* ! *)
| TEST of location
| GOTO of location
| LABEL of label
| HALT
...
and value_path =
  | STACK_LOCATION of offset
  | HEAP_LOCATION of offset
```

Value and stack data types

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
```

The interpreter 3 stack contains structured values

Jargon VM

```
type stack_item = STACK_INT of int
                | STACK_BOOL of bool
                | STACK_UNIT
                | STACK_HI of heap_index
                | STACK_RA of code_index
                | STACK_FP of stack_index
```

The Jargon VM stack contains integers, heap addresses, code addresses, and stack addresses

Heap data types

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
```

The interpreter 3 stack contains structured values.

Jargon VM

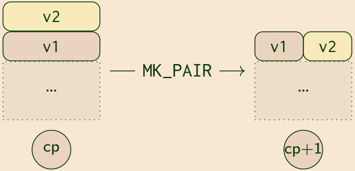
```
type heap_type =  HT_PAIR
                | HT_INL
                | HT_INR
                | HT_CLOSURE

type heap_item =  HEAP_INT of int
                | HEAP_BOOL of bool
                | HEAP_UNIT
                | HEAP_HI of heap_index
                | HEAP_CI of code_index
                | HEAP_HEADER of int
                                  * heap_type
```

The Jargon VM heap contains integers, addresses, and code addresses

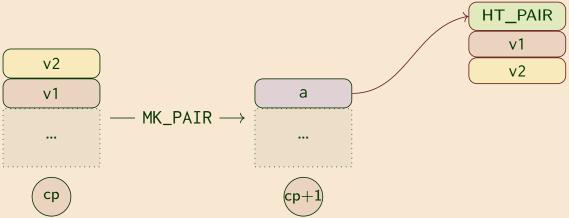
(The headers will be essential for garbage collection)

MK_PAIR

In interpreter 3:



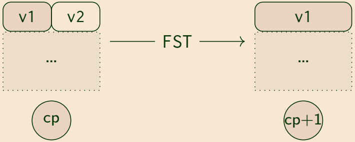
In Jargon VM:



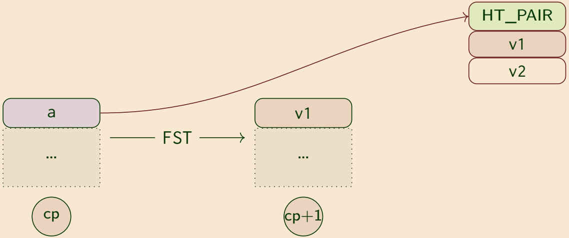
The pair is freshly allocated on the heap

FST (similar for SND)

In interpreter 3:



In Jargon VM:

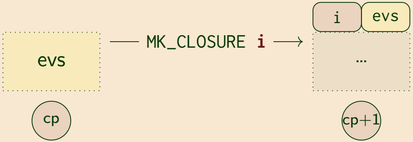


Note: v1 might be a simple value (`int` or `bool`) or another heap address

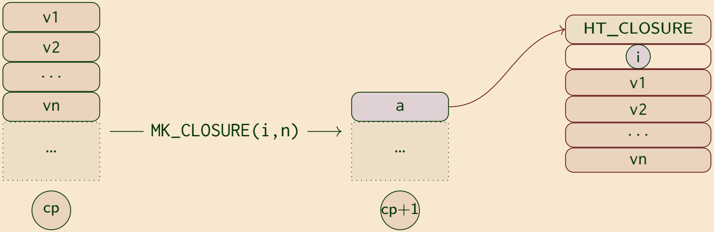
Functions

MK_CLOSURE (c,n)

In interpreter 3:



In Jargon VM:

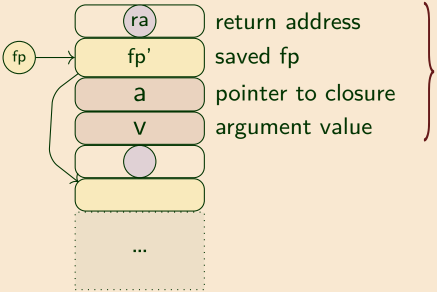


i = code location of start of instructions for closure

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 fp

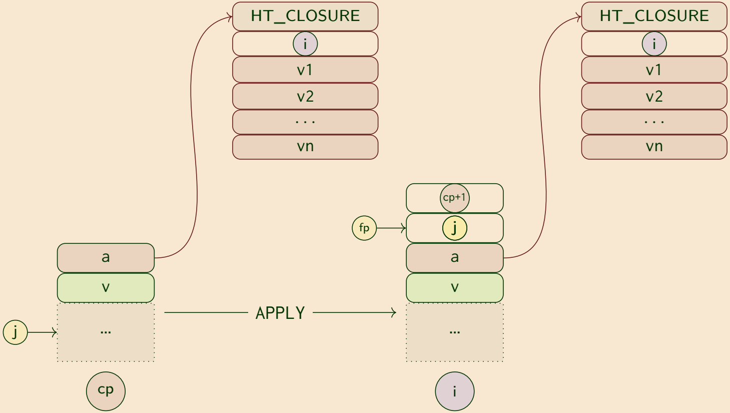
pointer to closure

argument value

Stack frame
(Boundary may vary in the literature)

Executing code for closure at heap address a after it was applied to argument v.

APPLY

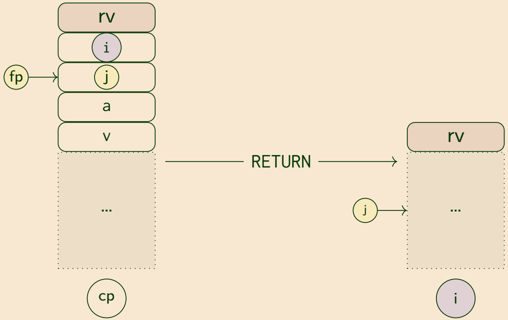


Push stack frame

Save return address (code pointer)

Save and update frame pointer

RETURN



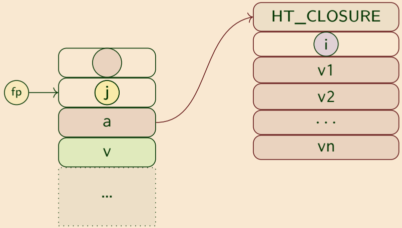
Discard stack frame

Update code pointer to return address

Restore frame pointer

Variables

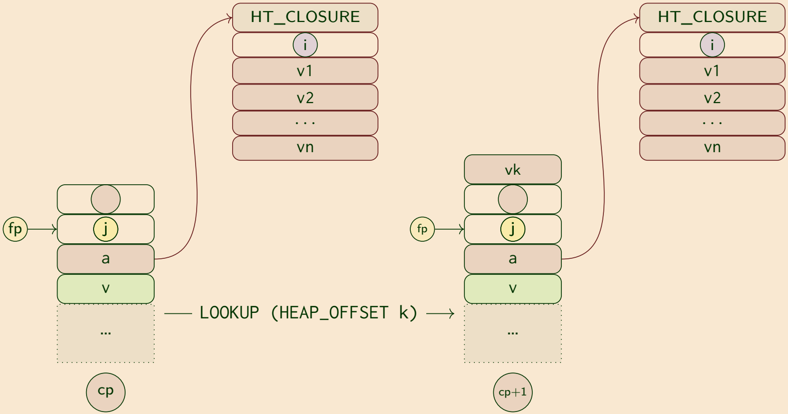
Finding a variable’s value at runtime



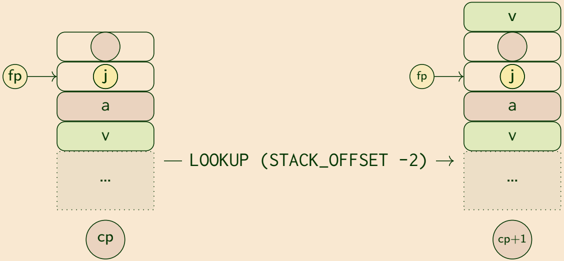
Suppose we are executing code associated with this closure.

Then:
The **argument** can be found at a fixed offset from fp
The **free variables** in the closure body can be found at fixed offsets from the closure a

LOOKUP (HEAP_OFFSET k)



LOOKUP (STACK_OFFSET -2)



Example

Example: compiling rev_pair.slang (front end)

Jargon VM

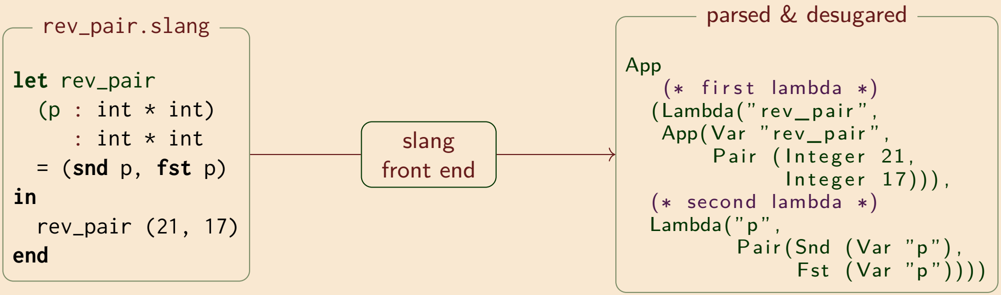
Instructions

Functions

Variables

Example
● ○ ○

The Gap



rev_pair.slang

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

slang front end

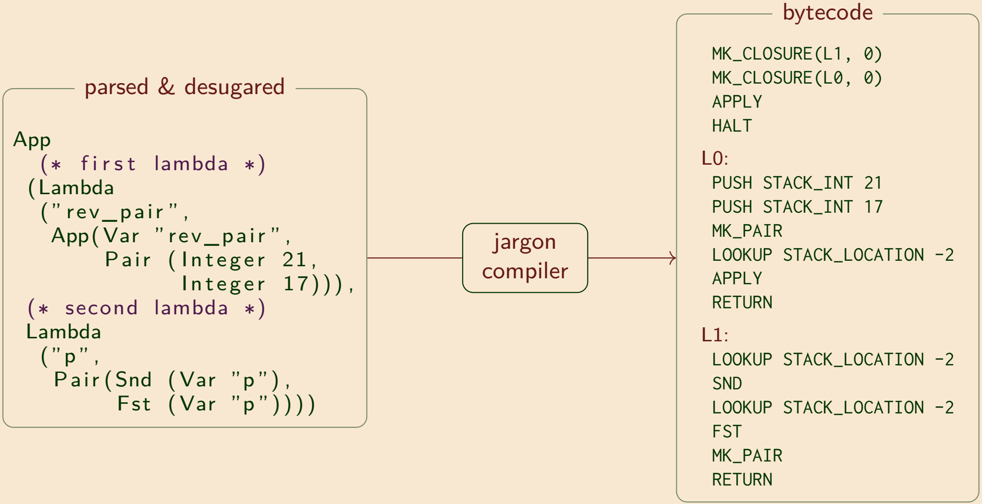
parsed & desugared

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

i.e.:

```
(fun rev_pair =>
    rev_pair (21, 17))
(fun p => (snd p, fst p))
```

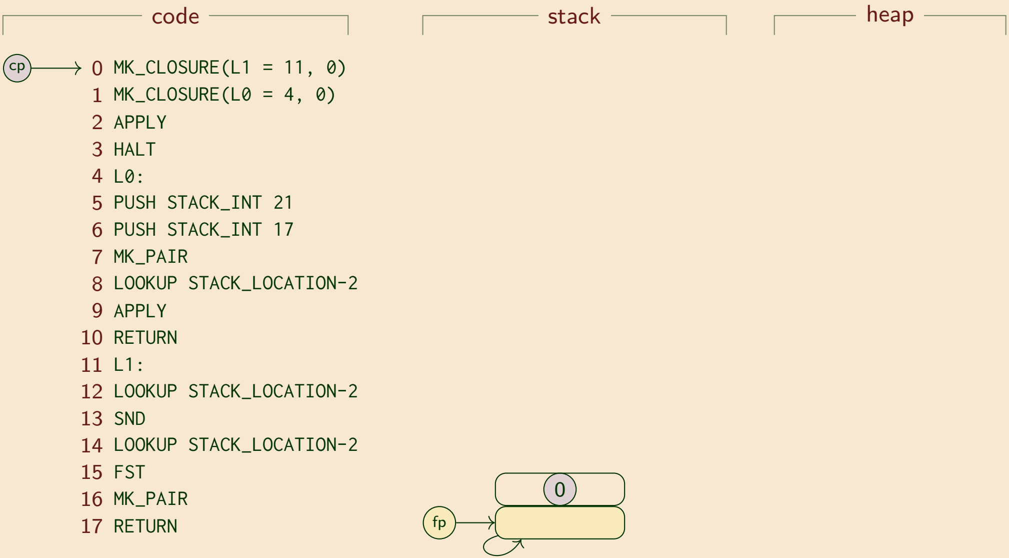
Example: compiling rev_pair.slang (back end)



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

```
 MK_CLOSURE(L1, 0)
 MK_CLOSURE(L0, 0)
 APPLY
 HALT
L0:
 PUSH STACK_INT 21
 PUSH STACK_INT 17
 MK_PAIR
 LOOKUP STACK_LOCATION -2
 APPLY
 RETURN
L1:
 LOOKUP STACK_LOCATION -2
 SND
 LOOKUP STACK_LOCATION -2
 FST
 MK_PAIR
 RETURN
```

Example: executing rev_pair.slang



code

cp

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

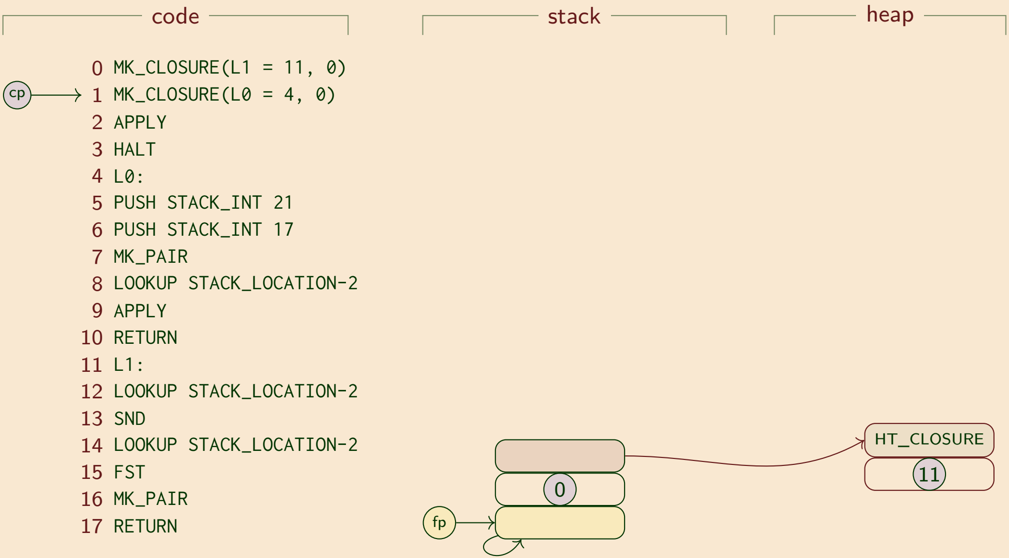
stack

0

fp

heap

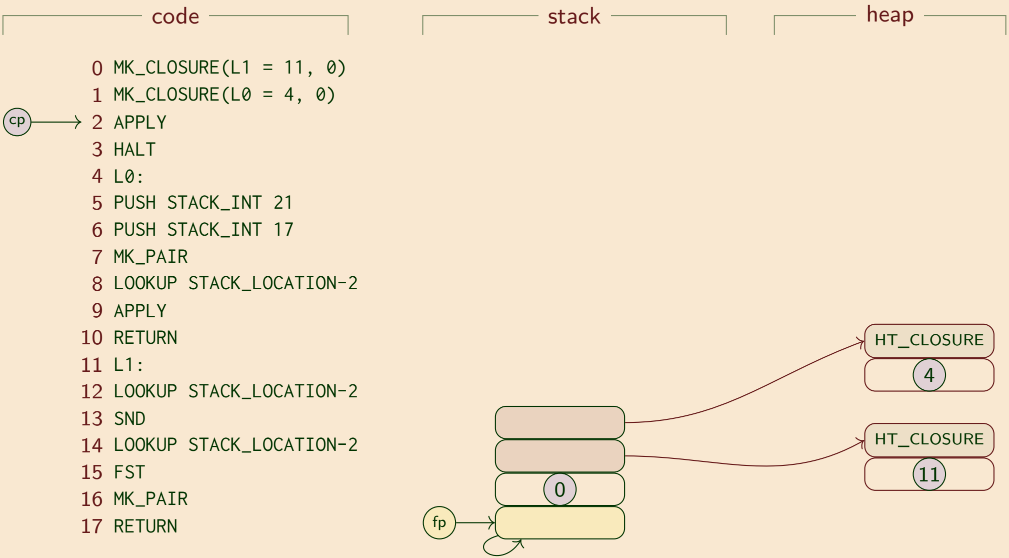
Example: executing rev_pair.slang



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

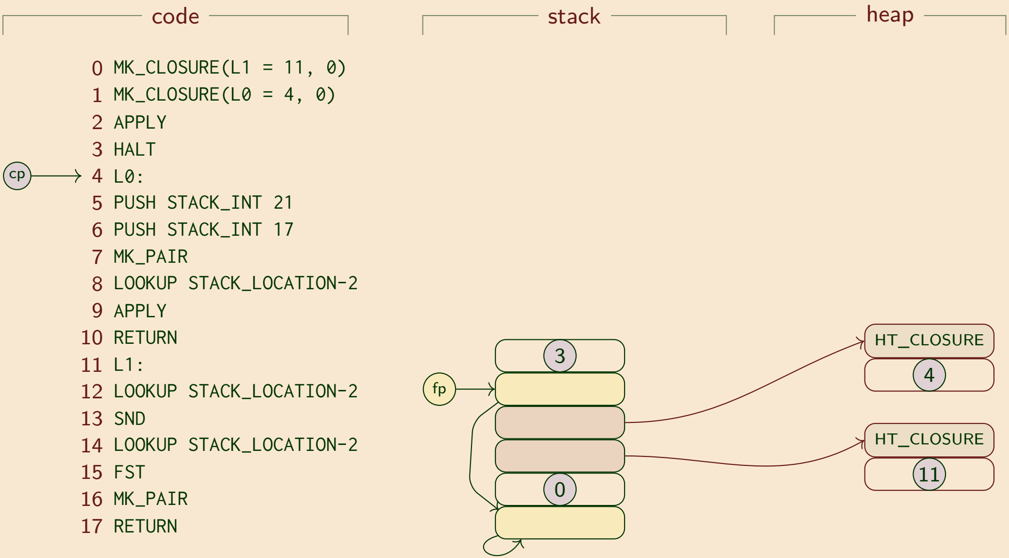
Jargon VM
Instructions
Functions
Variables
Example
The Gap

Example: executing rev_pair.slang



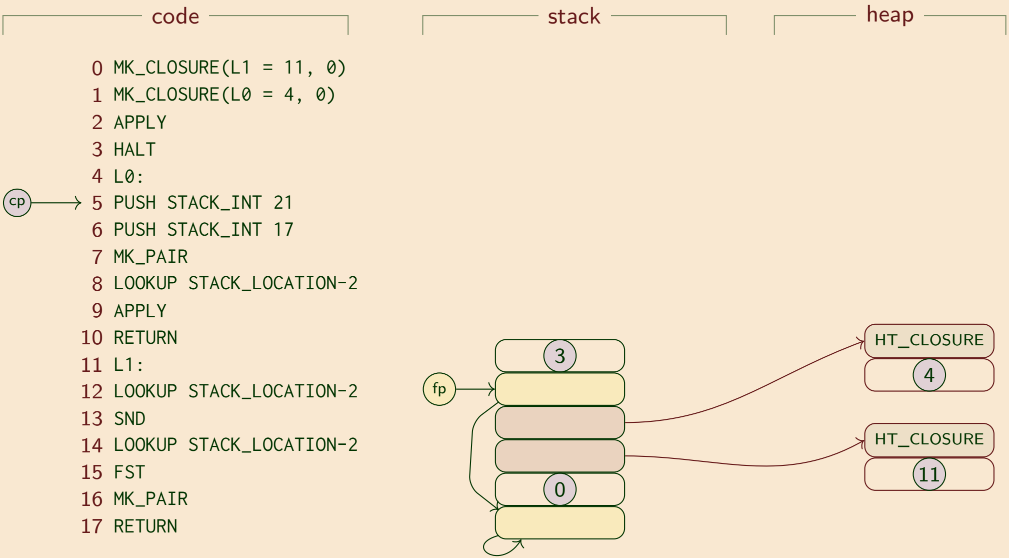
```
 0 MK_CLOSURE(L1 = 11, 0)
 1 MK_CLOSURE(L0 = 4, 0)
 2 APPLY
 3 HALT
 4 L0:
 5 PUSH STACK_INT 21
 6 PUSH STACK_INT 17
 7 MK_PAIR
 8 LOOKUP STACK_LOCATION-2
 9 APPLY
10 RETURN
11 L1:
12 LOOKUP STACK_LOCATION-2
13 SND
14 LOOKUP STACK_LOCATION-2
15 FST
16 MK_PAIR
17 RETURN
```

Example: executing rev_pair.slang



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

Example: executing rev_pair.slang



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

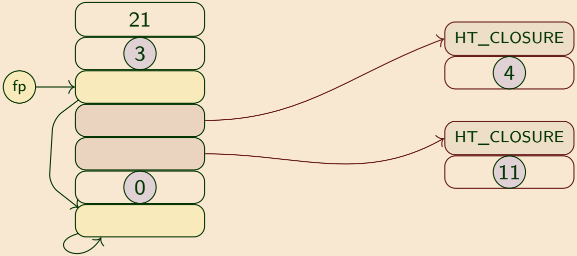
Example: executing rev_pair.slang

code

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

stack

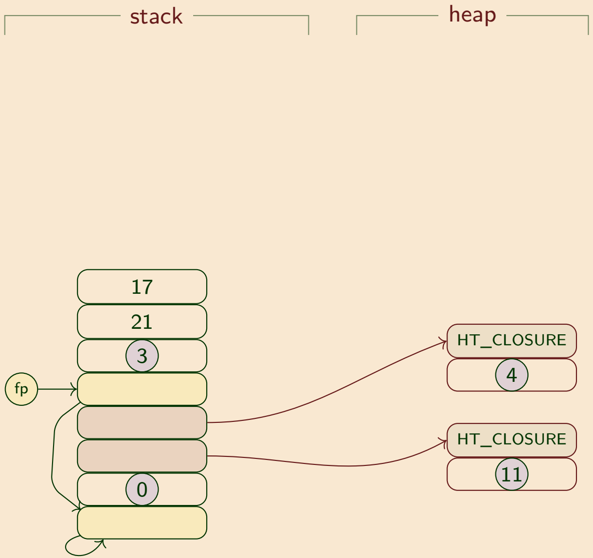
heap



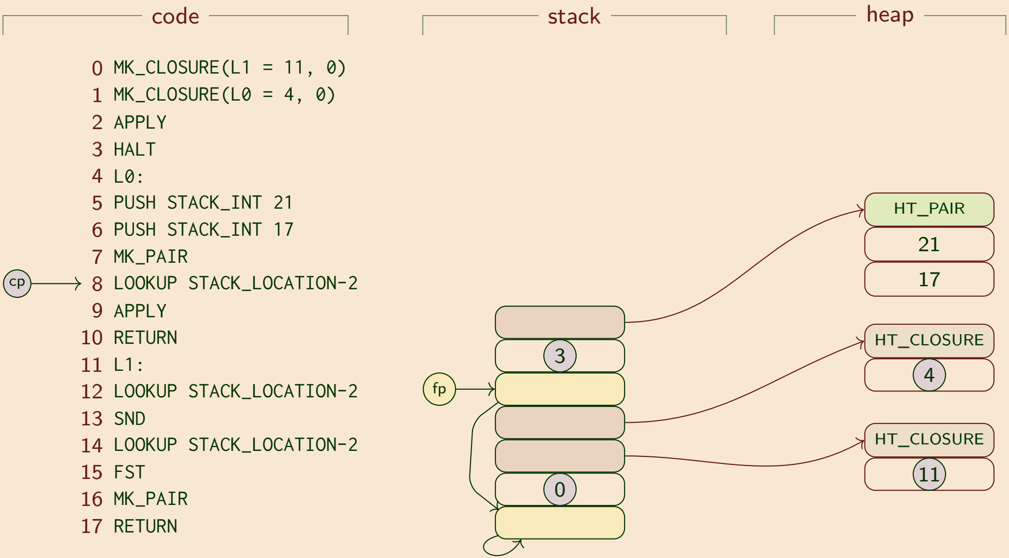
Example: executing rev_pair.slang

code

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



Example: executing rev_pair.slang

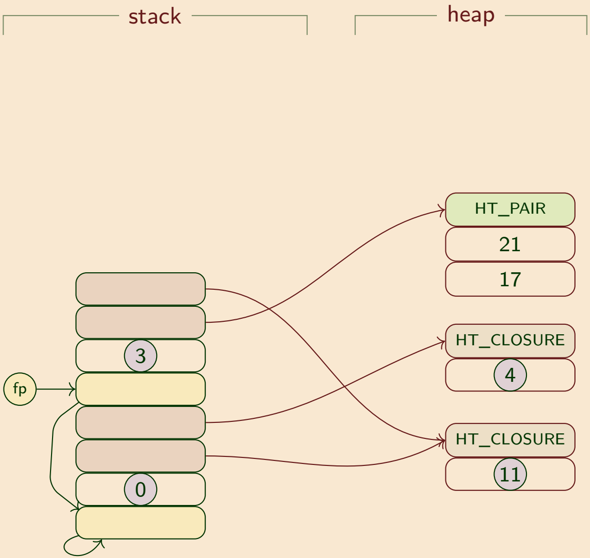


```
 0 MK_CLOSURE(L1 = 11, 0)
 1 MK_CLOSURE(L0 = 4, 0)
 2 APPLY
 3 HALT
 4 L0:
 5 PUSH STACK_INT 21
 6 PUSH STACK_INT 17
 7 MK_PAIR
 8 LOOKUP STACK_LOCATION-2
 9 APPLY
10 RETURN
11 L1:
12 LOOKUP STACK_LOCATION-2
13 SND
14 LOOKUP STACK_LOCATION-2
15 FST
16 MK_PAIR
17 RETURN
```

Example: executing rev_pair.slang

code

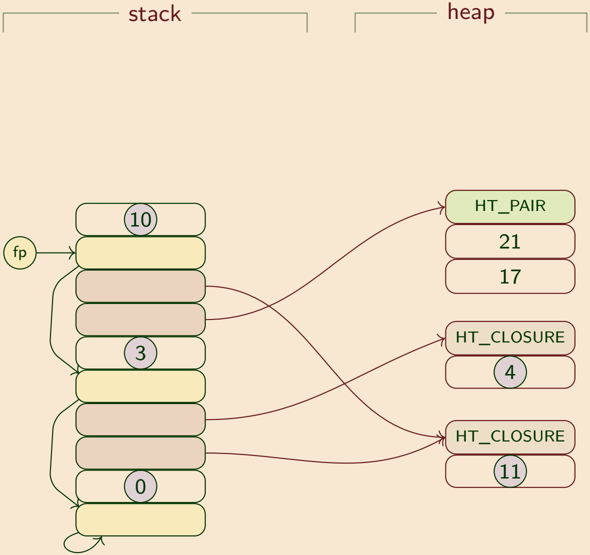
```
         0 MK_CLOSURE(L1 = 11, 0)
         1 MK_CLOSURE(L0 = 4, 0)
         2 APPLY
         3 HALT
         4 L0:
         5 PUSH STACK_INT 21
         6 PUSH STACK_INT 17
         7 MK_PAIR
         8 LOOKUP STACK_LOCATION-2
cp ───→  9 APPLY
        10 RETURN
        11 L1:
        12 LOOKUP STACK_LOCATION-2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



Example: executing rev_pair.slang

code

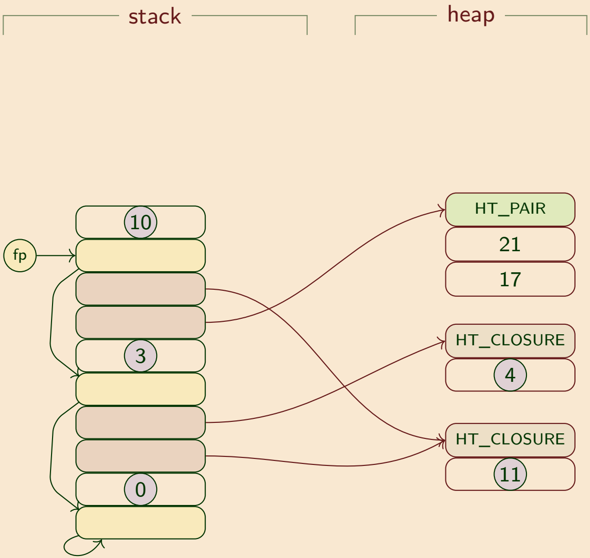
```
         0 MK_CLOSURE(L1 = 11, 0)
         1 MK_CLOSURE(L0 = 4, 0)
         2 APPLY
         3 HALT
         4 L0:
         5 PUSH STACK_INT 21
         6 PUSH STACK_INT 17
         7 MK_PAIR
         8 LOOKUP STACK_LOCATION-2
         9 APPLY
        10 RETURN
cp →    11 L1:
        12 LOOKUP STACK_LOCATION-2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



Example: executing rev_pair.slang

code

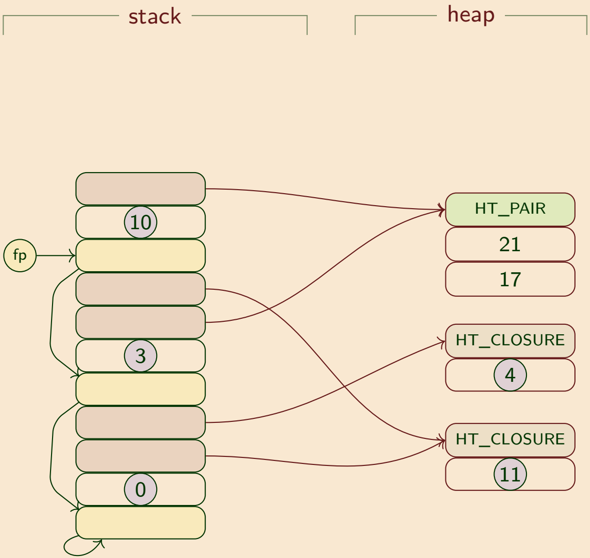
```
    0 MK_CLOSURE(L1 = 11, 0)
    1 MK_CLOSURE(L0 = 4, 0)
    2 APPLY
    3 HALT
    4 L0:
    5 PUSH STACK_INT 21
    6 PUSH STACK_INT 17
    7 MK_PAIR
    8 LOOKUP STACK_LOCATION-2
    9 APPLY
   10 RETURN
   11 L1:
cp → 12 LOOKUP STACK_LOCATION-2
   13 SND
   14 LOOKUP STACK_LOCATION-2
   15 FST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



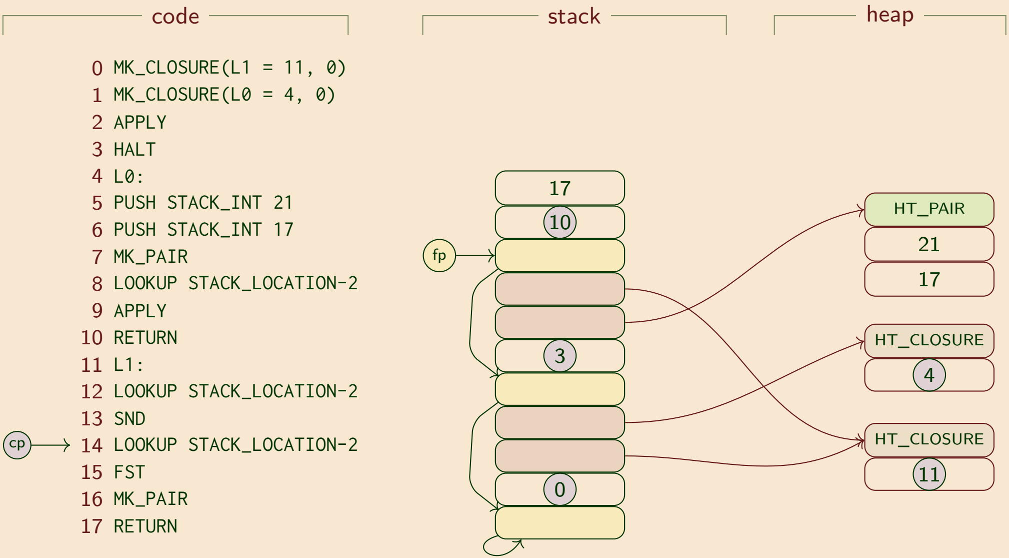
Example: executing rev_pair.slang

code

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



Example: executing rev_pair.slang



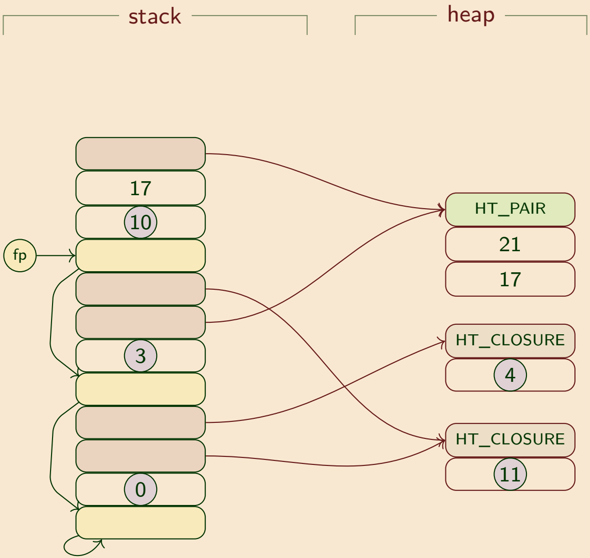
```
 0 MK_CLOSURE(L1 = 11, 0)
 1 MK_CLOSURE(L0 = 4, 0)
 2 APPLY
 3 HALT
 4 L0:
 5 PUSH STACK_INT 21
 6 PUSH STACK_INT 17
 7 MK_PAIR
 8 LOOKUP STACK_LOCATION-2
 9 APPLY
10 RETURN
11 L1:
12 LOOKUP STACK_LOCATION-2
13 SND
14 LOOKUP STACK_LOCATION-2
15 FST
16 MK_PAIR
17 RETURN
```

Example: executing rev_pair.slang

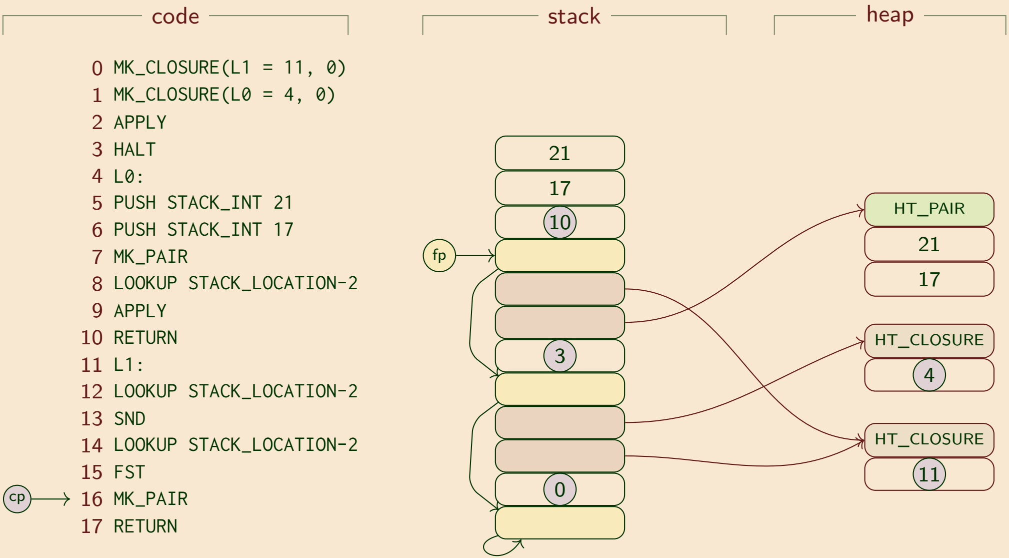
code

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

cp → 15



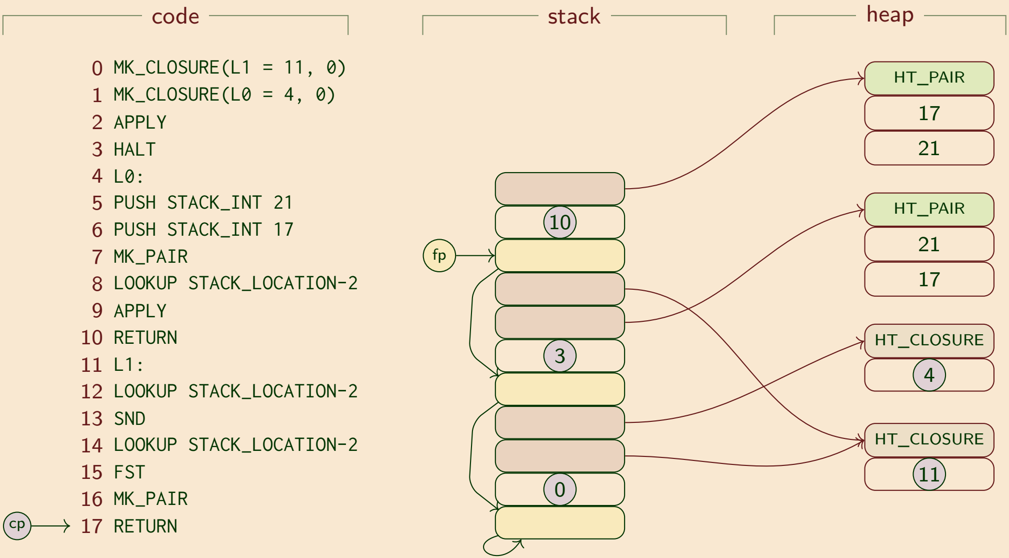
Example: executing rev_pair.slang



code

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

Example: executing rev_pair.slang

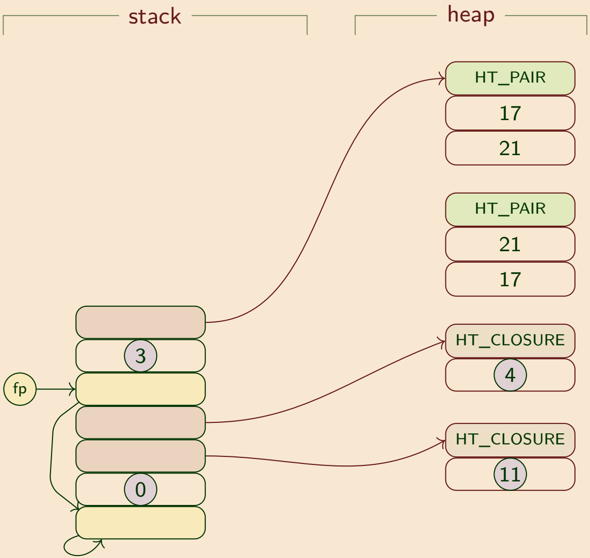


```
 0 MK_CLOSURE(L1 = 11, 0)
 1 MK_CLOSURE(L0 = 4, 0)
 2 APPLY
 3 HALT
 4 L0:
 5 PUSH STACK_INT 21
 6 PUSH STACK_INT 17
 7 MK_PAIR
 8 LOOKUP STACK_LOCATION-2
 9 APPLY
10 RETURN
11 L1:
12 LOOKUP STACK_LOCATION-2
13 SND
14 LOOKUP STACK_LOCATION-2
15 FST
16 MK_PAIR
17 RETURN
```

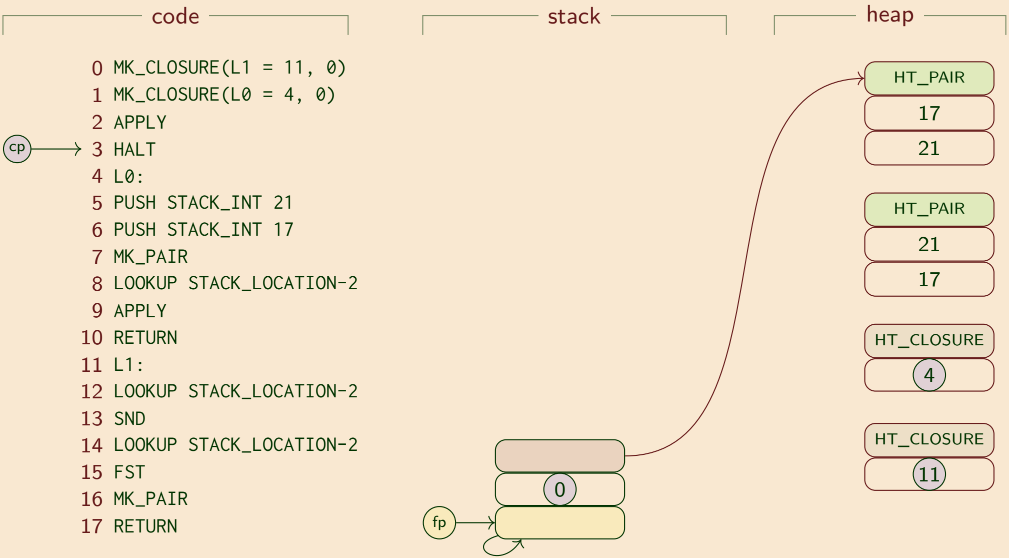
Example: executing rev_pair.slang

code

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



Example: executing rev_pair.slang



code

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

The Gap, revisited

The Gap: Slang to Jargon VM

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

Transform the evaluator:

CPS + defunctionalize, make stack explicit (Lecture 8)

split stacks
refactor: compiler + low-level interpeter (Lecture 9)

linearise + cp + LABEL/GOTO
compile away conditionals and loops (Lecture 10)

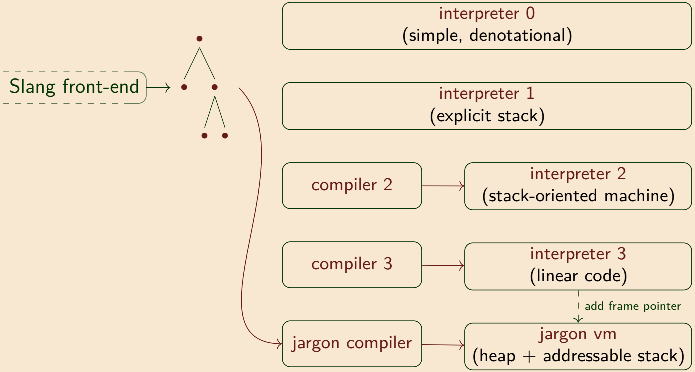
make stack addressable + add fp
optimize closure representation (Lecture 11)
move complex data to the heap

```
MK_CLOSURE(L1 = 11, 0)
MK_CLOSURE(L0 = 4, 0)
APPLY
HALT
LABEL L0
PUSH STACK_INT 21
PUSH STACK_INT 17
MK_PAIR
LOOKUP STACK_LOCATION-2
APPLY
RETURN
LABEL L1
LOOKUP STACK_LOCATION-2
SND
LOOKUP STACK_LOCATION-2
FST
MK_PAIR
RETURN
```

The derivation

Starting from a direct implementation of Slang/L3 semantics, we **derived** a virtual machine in a step-by-step manner.

The correctness of each step is easy to check.



Slang front-end

interpreter 0
(simple, denotational)

interpreter 1
(explicit stack)

compiler 2

interpreter 2
(stack-oriented machine)

compiler 3

interpreter 3
(linear code)

add frame pointer

jargon compiler

jargon vm
(heap + addressable stack)

Remarks

The semantic GAP between a Slang/L3 program and a low-level translation (say x86/Unix) has been significantly reduced.

Implementing the Jargon VM at a lower-level of abstraction (in C?, JVM bytecodes? X86/Unix? ...) now a relatively easy programming problem.

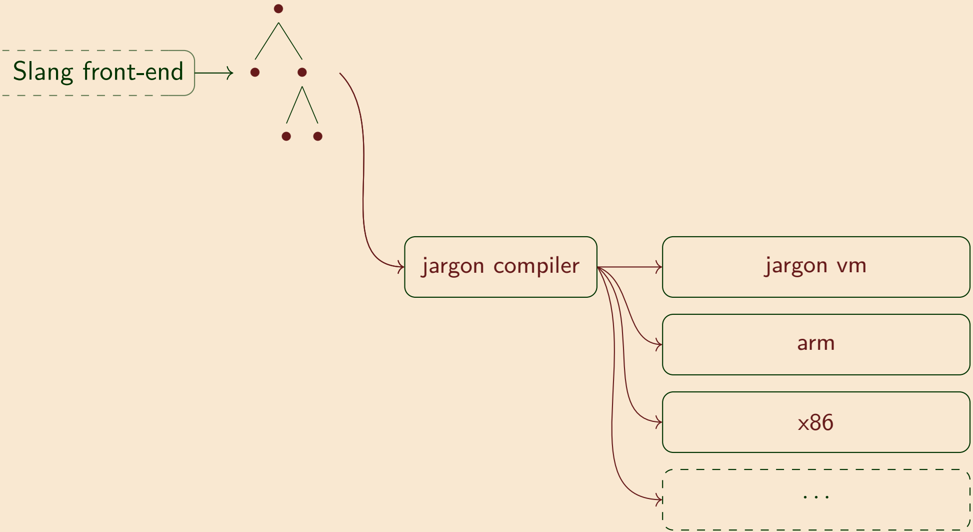
However, using a lower-level implementation (say x86, exploiting fast registers) to generate very efficient code is not so easy. (See Part II Optimising Compilers).

New possibility: Jargon bytecode interpreter in C

```
...
...
void vsm_execute_instruction(vsm_state *state, bytecode instruction) {
  opcode code = instruction.code;
  argument arg1 = instruction.arg1;
  switch (code) {
    case PUSH: { state→stack[state→sp++] = arg1; state→pc++; break; }
    case GOTO: { state→pc = arg1; break; }
    case STACK_LOOKUP: {
      state→stack[state→sp++] =
      state→stack[state→fp + arg1];
      state→pc++; break;! }
    ...
  }
} ...
```

Idea: Generate compact bytecode for each Jargon instruction.
Compiler writes bytecode to a file
Implement an interpreter in C or C++ for the bytecode
Execution much faster than `jargon.ml`
Alternatively: generate assembly code from Jargon instructions

Backend could target multiple platforms



Slang front-end

jargon compiler

jargon vm

arm

x86

. . .

Next time: miscellany