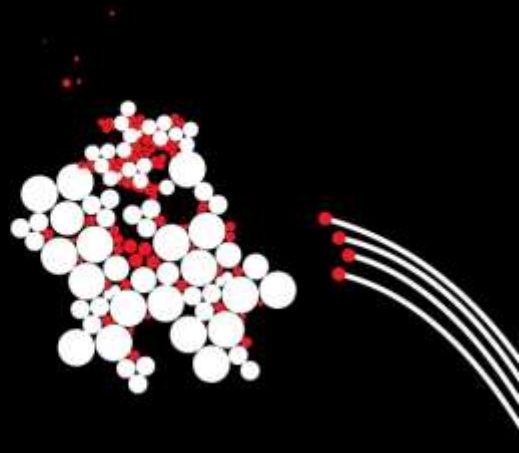
UNIVERSITY OF TWENTE.

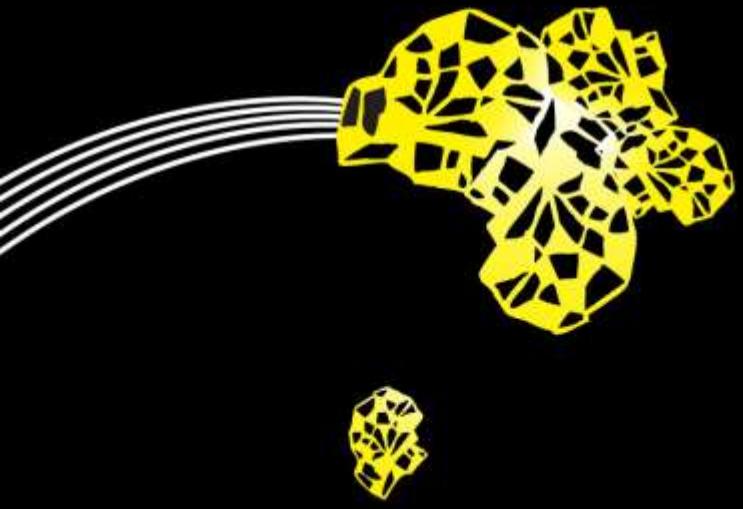


COMPILER CONSTRUCTION

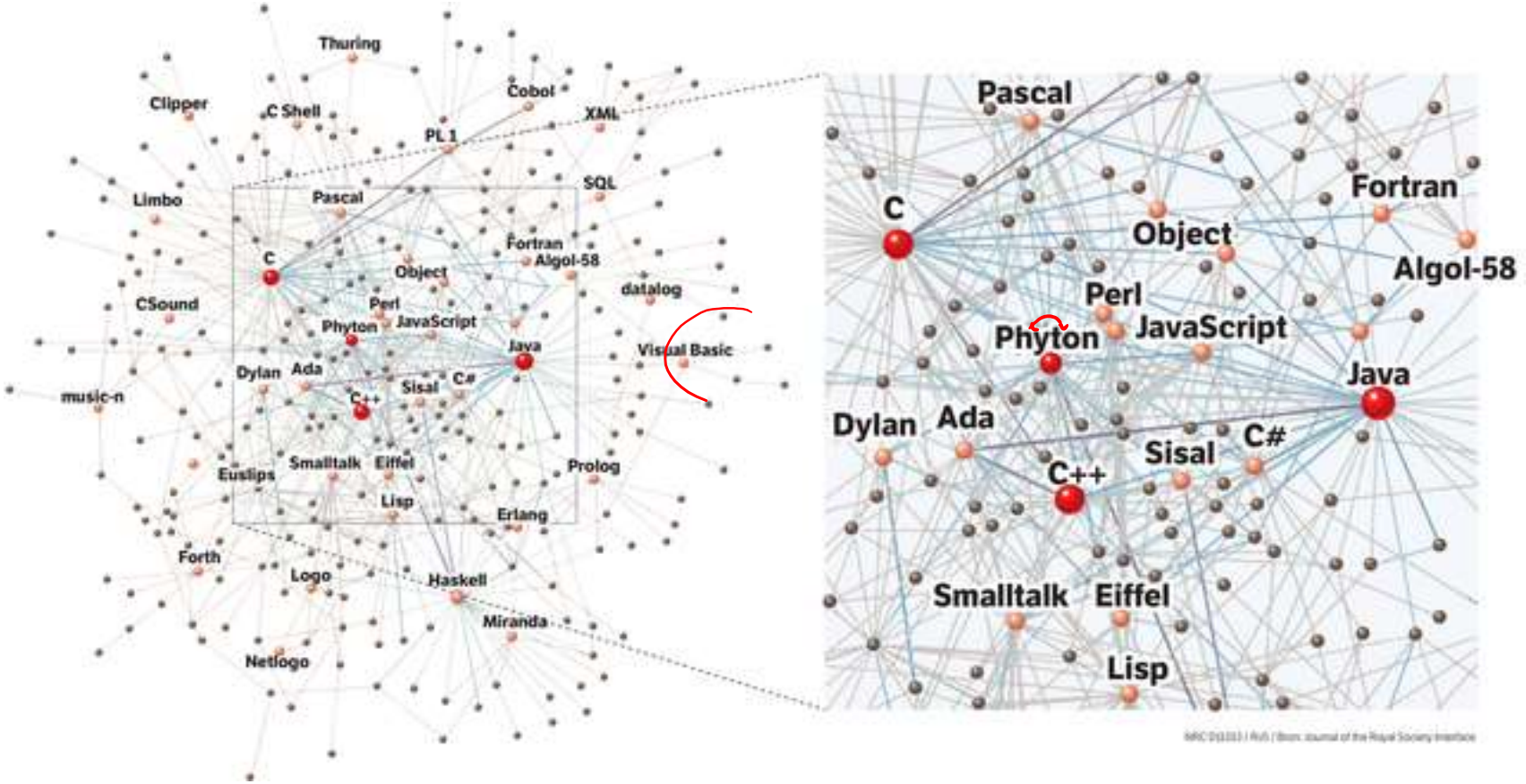
CC 5-2: THE PROCEDURE ABSTRACTION

24 MAY 2019

MODULE 8: PROGRAMMING PARADIGMS

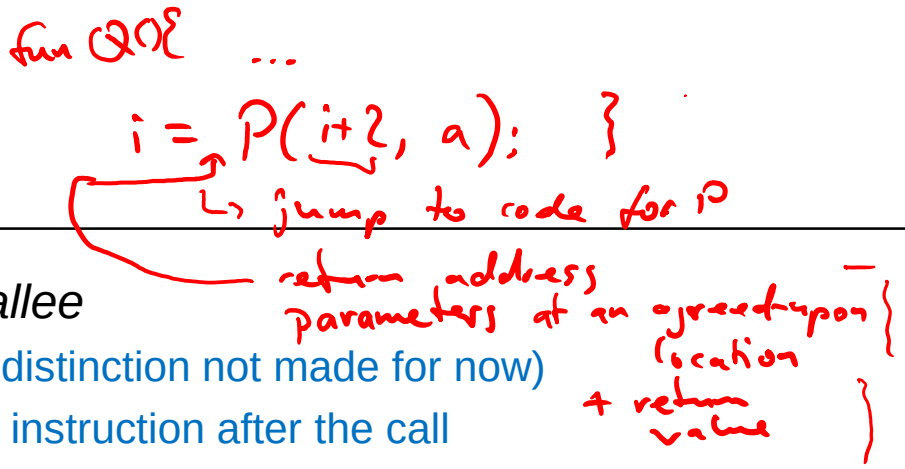


ALGOL-LIKE LANGUAGES (ALLS)



Punctuated equilibrium in the large-scale evolution of programming languages, Sergi Valverde, Ricard V. Solé, J. R. Soc. Interface 2015 12 20150249
Picture lifted from NRC Handelsblad

PROCEDURE CALL IN ALLS

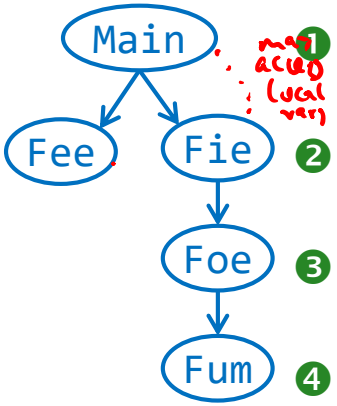


- Transfer of control from *caller* to *callee*
  - Callee is a procedure (or function – distinction not made for now)
  - After finishing, control returns to the instruction after the call
  - Callee may make further calls (recursive or not)
  - Procedures may be lexically nested and sometimes used as parameters
- Identity of callee is known and fixed
  - Determined by instruction number/memory address
  - Fixed *call graph*
    - Shows potential caller/callee relationships during program execution
- What compile-time structures are needed to make this work?
  - Symbol table for procedures: stores their signatures
  - Symbol table for local variables: static coordinates (offset and scope depth) ← r_arp
- What run-time structures are needed to make this work?
  - Activation record per call, typically kept on stack (a.k.a. stack frame)
  - Contains parameter, local variables, return value, return address, registers, …

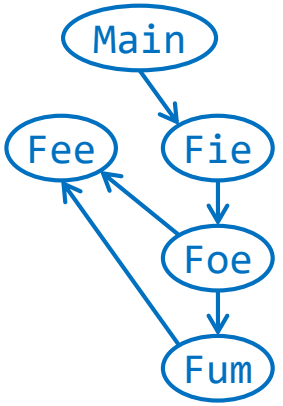
COMPILE-TIME STRUCTURES: STATIC COORDINATES

```
program Main;
❶ var x, y, z: integer;
  procedure Fee;
  ❷ var x: integer;
    begin { body Fee } end;
  procedure Fie;
  ❷ var y: real;
    procedure Foe;
    ❸ var z: real;
      procedure Fum;
      ❹ var y: real;
        begin { body Fum }
          Fee;
        end;
      begin { body Foe }
        Fee; Fum;
      end;
    begin { body Fie }
      Foe;
    end;
  begin { main }
    Fie;
  end.
```

Scope graph



Call graph



Static variable coordinate (depth, offset)

depth: level in scope graph (start at 1)

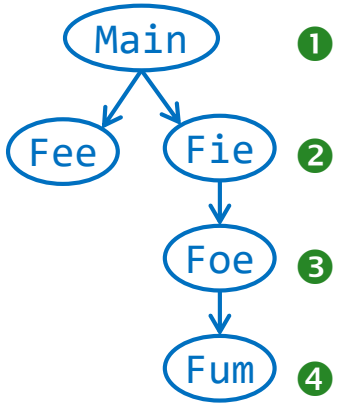
offset: distance to base address

Scope	x	y	z
Main	(1, 0)	(1, 4)	(1, 8)
Fee	(2, 0)	(1, 4)	(1, 8)
Fie	(1, 8)	(2, 0)	(1, 8)
Foe	...		
Fum			

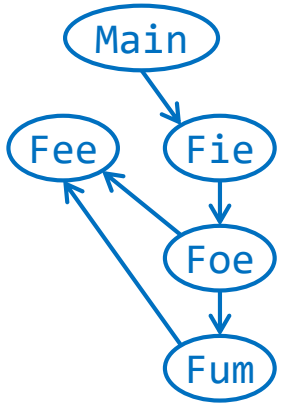
COMPILE-TIME STRUCTURES: STATIC COORDINATES

```
program Main;
❶ var x, y, z: integer;
  procedure Fee;
  ❷ var x: integer;
    begin { body Fee } end;
  procedure Fie;
  ❷ var y: real;
    procedure Foe;
    ❸ var z: real;
      procedure Fum;
      ❹ var y: real;
        begin { body Fum }
          Fee;
        end;
      begin { body Foe }
        Fee; Fum;
      end;
    begin { body Fie }
      Foe;
    end;
  begin { main }
    Fie;
  end.
```

Scope graph



Call graph



Static variable coordinate (depth, offset)

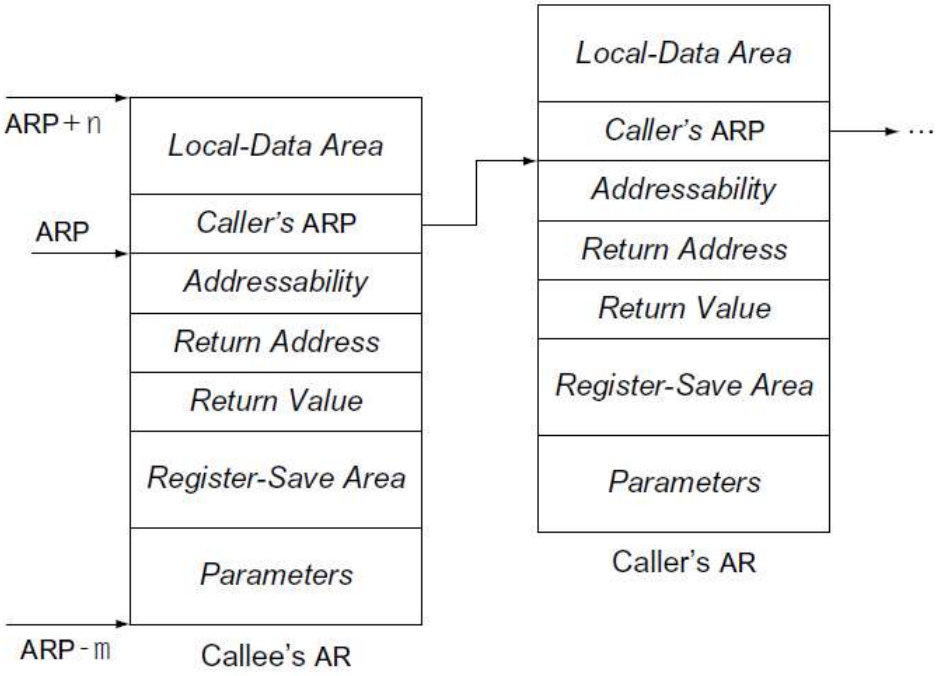
depth: level in scope graph (start at 1)

offset: distance to base address

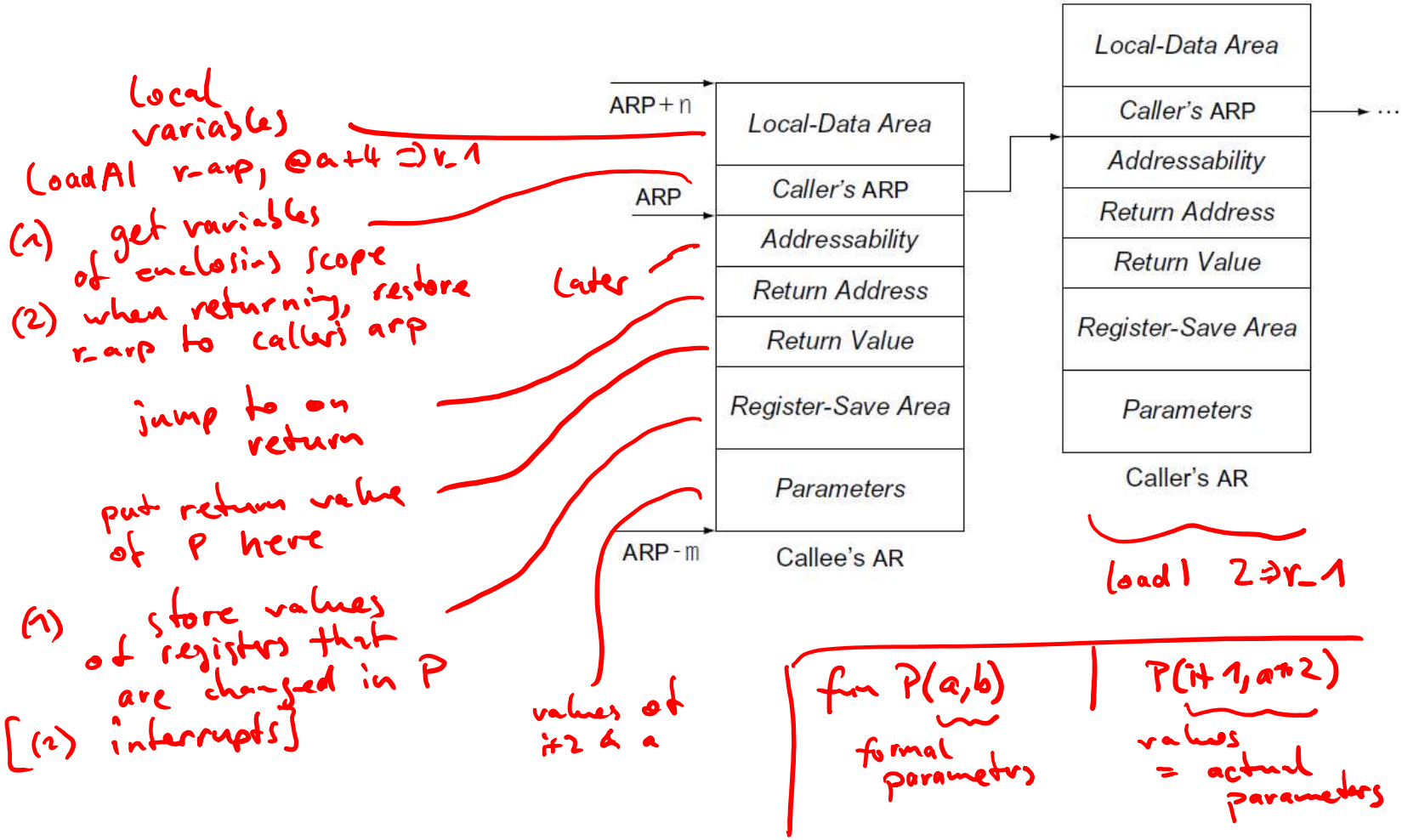
Scope	x	y	z
Main	(1,0)	(1,4)	(1,8)
Fee	(2,0)	(1,4)	(1,8)
Fie	(1,0)	(2,0)	(1,8)
Foe	(1,0)	(2,0)	(3,0)
Fum	(1,0)	(4,0)	(3,0)

RUN-TIME STRUCTURES: ACTIVATION RECORDS

- At every point during execution every active procedure has an *activation record*
  - Stored in memory
  - **A**ctivation **R**ecord **P**ointer: identifies memory address

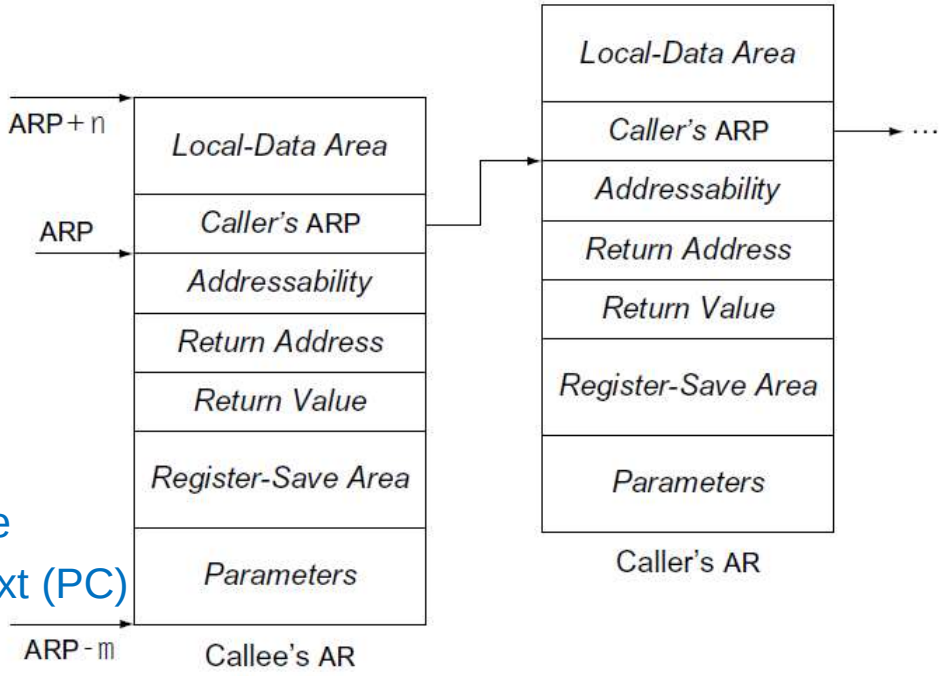


RUN-TIME STRUCTURES: ACTIVATION RECORDS



RUN-TIME STRUCTURES: ACTIVATION RECORDS

- At every point during execution every active procedure has an *activation record*
  - Stored in memory
  - **A**ctivation **R**ecord **P**ointer: identifies memory address
- Ingredients:
  - *Local-Data Area*: local variables
  - *Caller's ARP*: to restore context
  - *Addressability*: Next lexical scope
  - *Return Address*: to restore context (PC)
  - *Return Value*: for functions
  - *Register-Save Area*: to free registers for reuse by callee
  - *Parameters*: speaks for itself



ACTIVATION RECORD EXAMPLE

```
1  private static int g;

2  public static void main(String[] args) {
3     int x = 23;
4     g = 15;
5     int y = 3 + call(x, 2);
6     System.out.println("x+y = " + (x+y));
7  }

8  private static int call(int a, int b) {
9     a = 2 * a;
10    int c = a * b;
11    return g + c;
12 }
```

Memory content at line 5 (before the call)

Global memory | 15 | g

Registers: r_1 ← 3

main's AR (line 5):

arp →

Local data	23	x
Caller's ARP	n/a	
Return address	exit	
Parameters	(pointer)	args

call's AR (line 12):

Local data	92	c
Caller's ARP		arp
Return address	"6"	
Return value	107	
Register saves	3	r_1
Parameters	46	a
	2	b

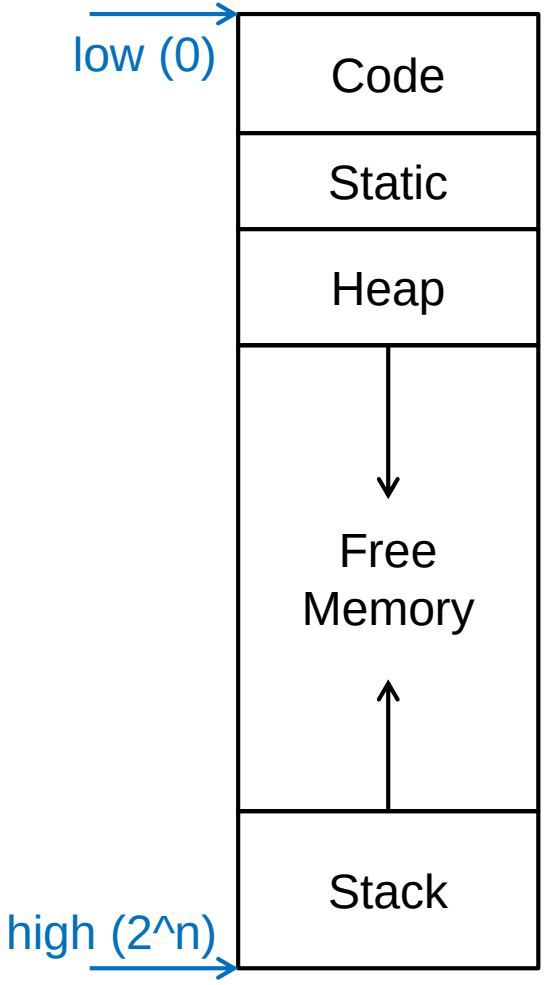
(no return value, addressability, register saves)

SAVING REGISTERS

- Caller does not know what callee will do
  - Callee needs to have freedom to use registers
  - Before callee starts execution, old register values saved in AR
  - This is the price of register-to-register memory model!
- Who should save?
  - Caller: knows which registers it still needs later (after the call)
    - Not needed anymore? Don't save old value
  - Callee: knows which registers it will need
    - Do not plan to use? Don't save old value
- Strategy may depend on situation

↳ part of "calling convention"

MEMORY LAYOUT



- Code memory: fixed-sized
  - In this course we assume separate memory area
  - Avoids problems, e.g. accidentally changing code
- Static: fixed-sized
  - Global variables
  - Information needed for lookup, e.g. display (later)
  - Anything else that can be statically computed
- Heap: variable-sized
  - Objects
  - Data structures of unpredictable/dynamic size
  - Sometimes: allocation records (closures)
  - No order restrictions
- Stack: variable-sized
  - Temporary values
  - Sometimes: allocation records (stack-based)
  - Strictly last-in-first-out

ALLOCATING ACTIVATION RECORDS

- Typical: stack-based allocation
  - Callee exits before caller
  - Accounts for dynamic nature of calls
  - Accounts for recursion
- Sometimes: heap-based (dynamic) allocation
  - If callee can outlive caller
  - For instance, in functional languages: *closures*
- Optimisation: static allocation
  - If callee does not do further calls — "leaf procedure"
  - Can be decided at compile time