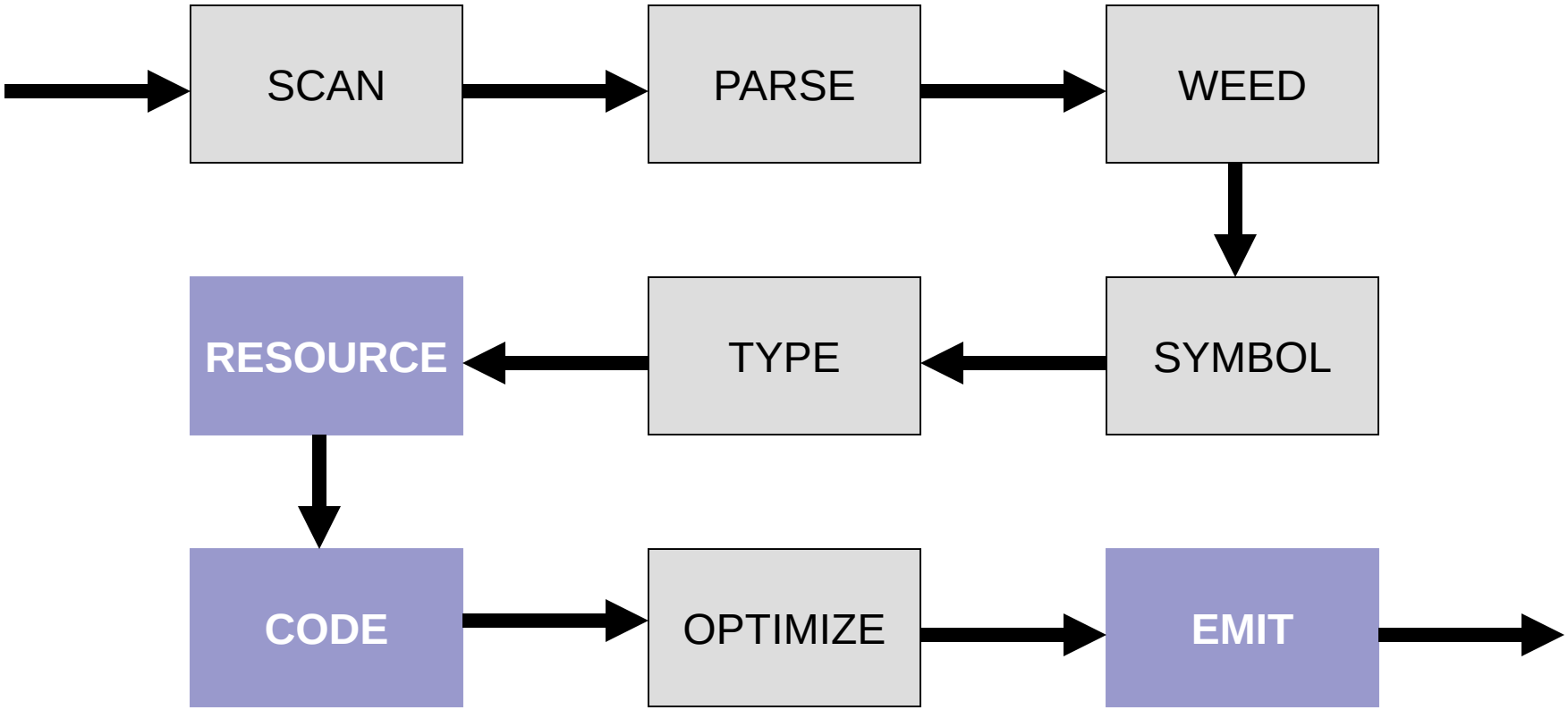
Compiler

Code Generation

Compiler Architecture



Code Generation

- Issues
  - generating an internal representation of machine codes for source code abstractions:
    - statements and expressions
    - scopes and types
    - procedures, parameters, return values
    - control flow constructs etc.
  - runtime-system
  - optimizing the code (ignored for now); and
  - emitting the code to files in binary format.

Code Generation

- Not addressed now ..
  - low level issues
    - register allocation
    - multicore architectures
    - …..
  - functional / logic / domain-specific languages
  - …

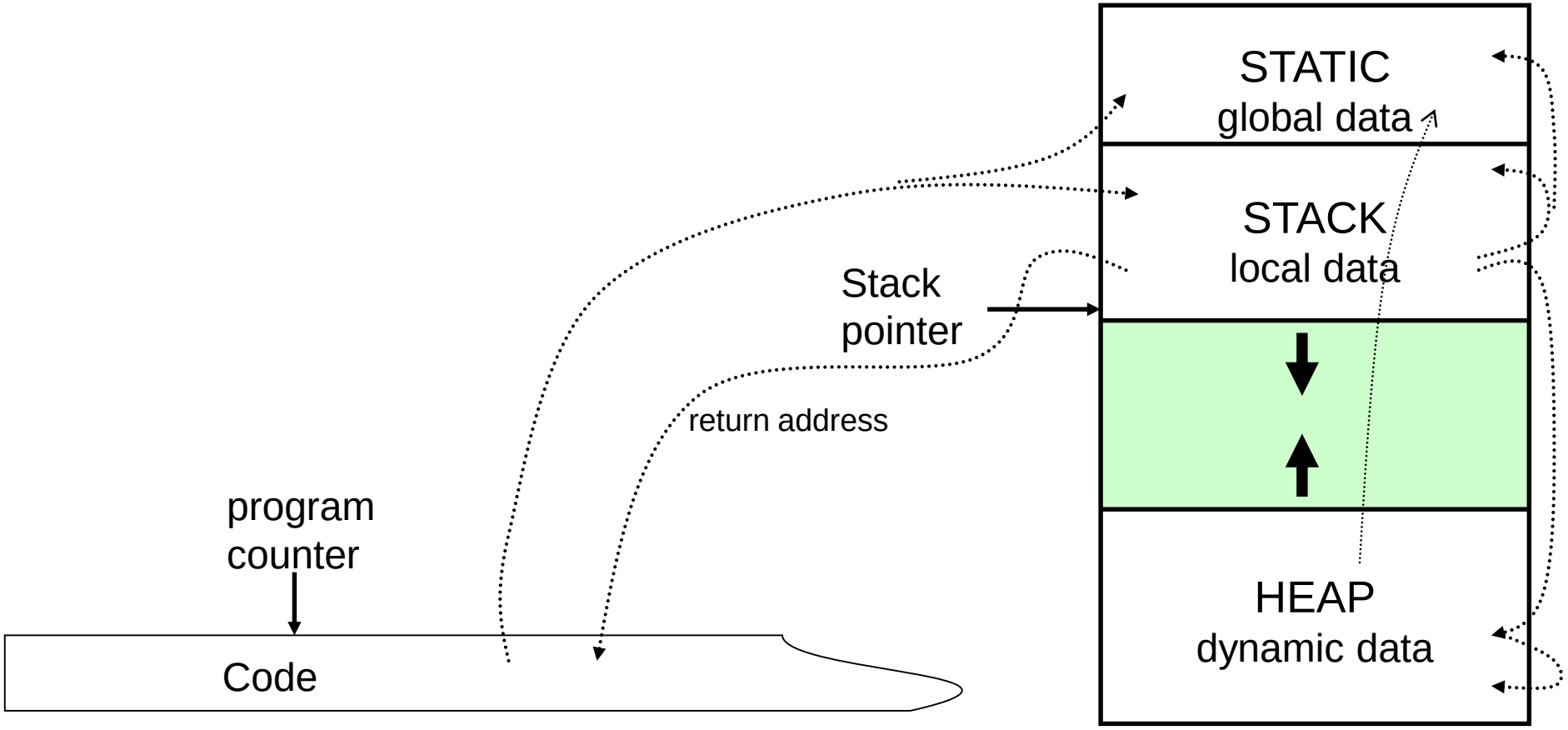
Runtime system

- Compiler builds
  - an environment and assumes the program be executed in this environment
  - a set of procedures that are needed for program execution, e.g., for
    - interaction with the OS
    - data access and memory administration
    - error handling
    - …
  - uses virtual memory addresses

Runtime system

- Managing data accesses
  - Stack: Procedure (method) invocations
  - Heap: Garbage collection
  - File system / IO
- Scheduling
  - Concurrency
  - Non-determinism
- Monitoring
  - Exceptions
  - Debugging

Memory architecture

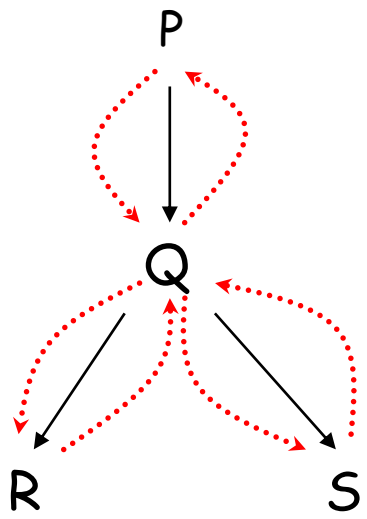


Activation records

Control flows between procedure activations in LIFO (l*ast-in-first-out*) discipline ($\Rightarrow$ stack principle)

```
procedure P(...)
    …
   Q(…);
    …
end;

procedure Q (… );
      …
    R(…);
      …
    S(…);
      …
 end;
```

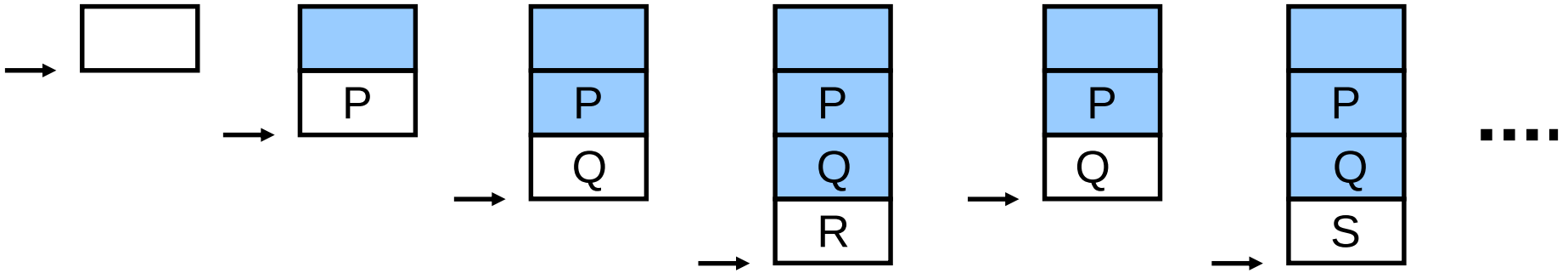


Activation tree

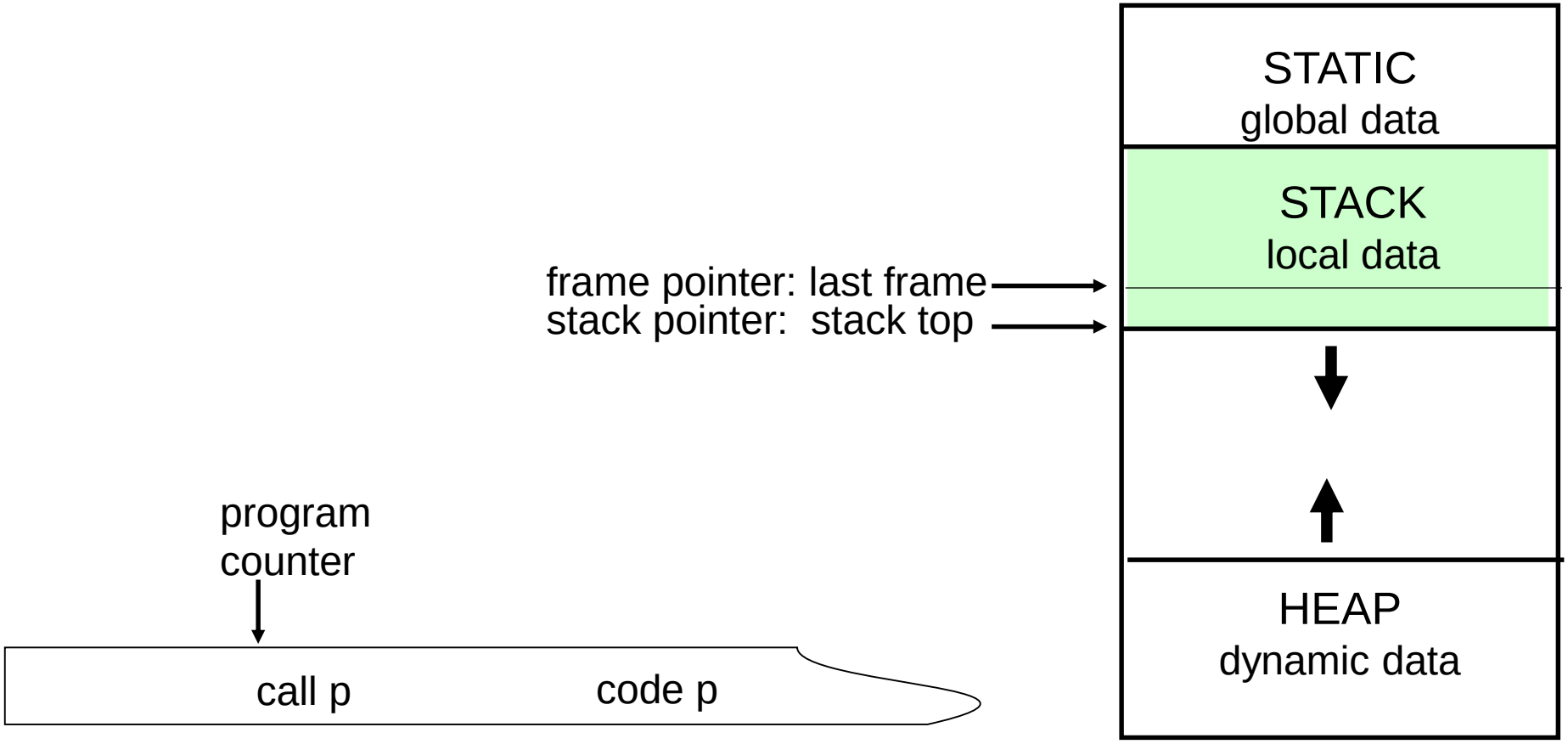
processed by pre-order traversal

Activation records

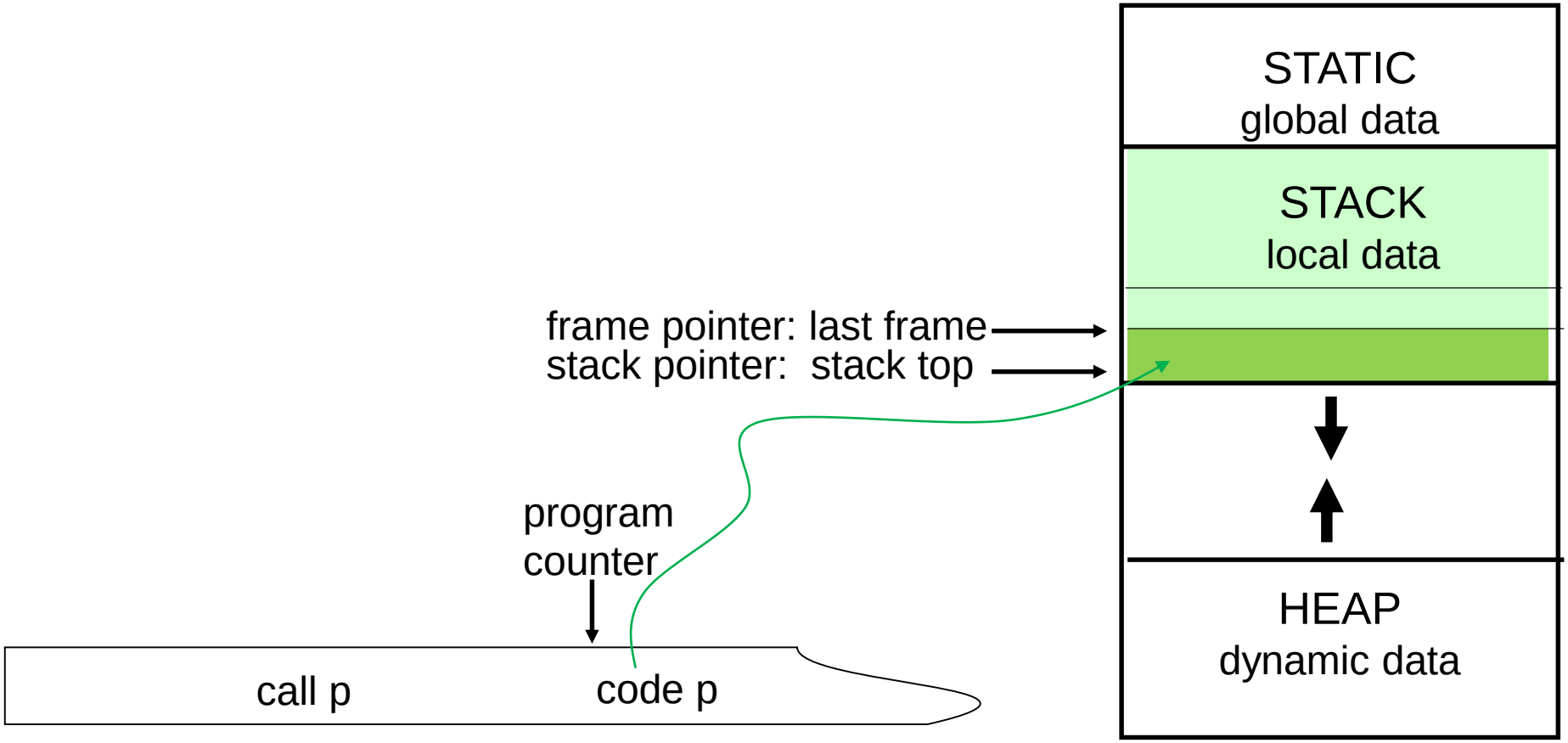
Stack-like storage allocation for activations:



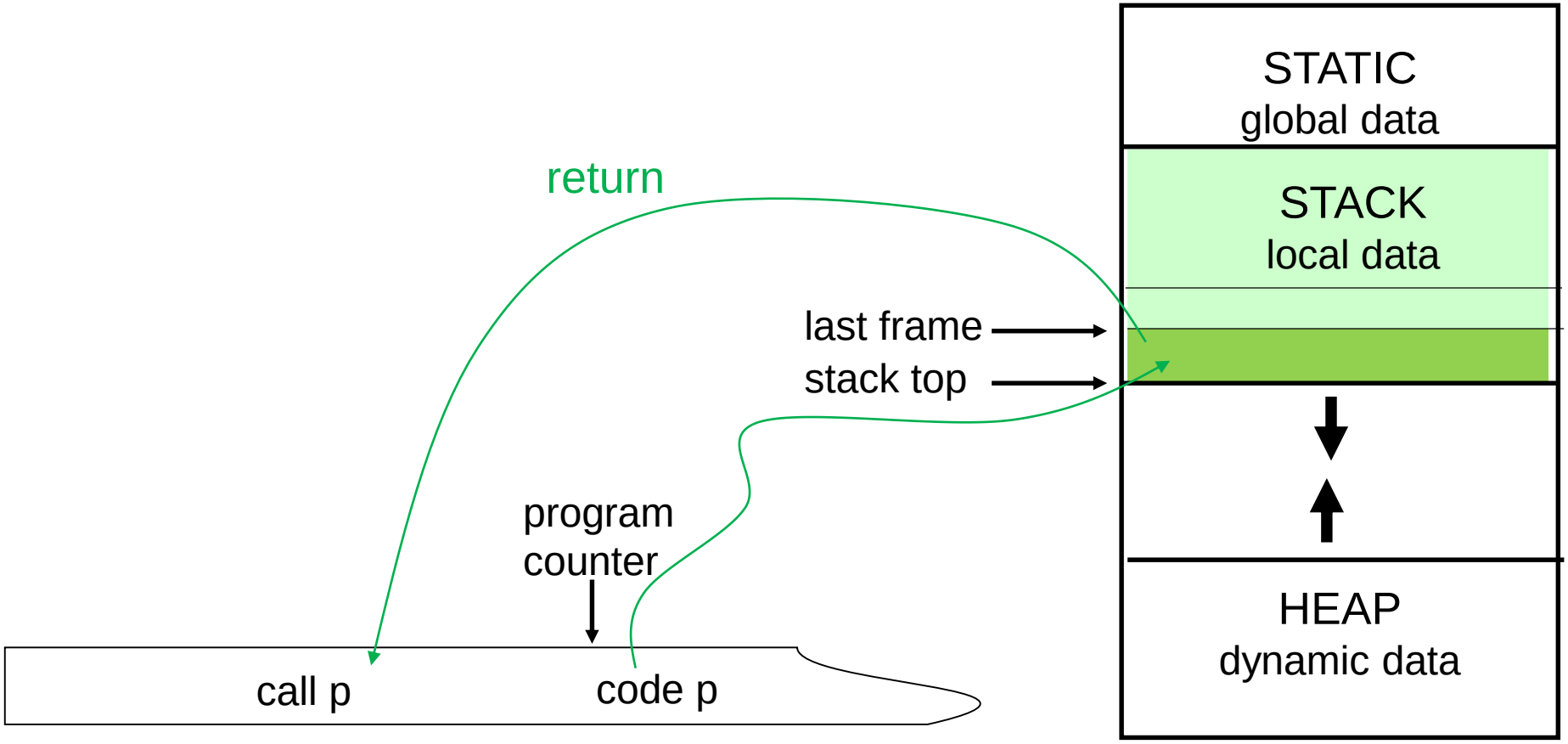
Activation records



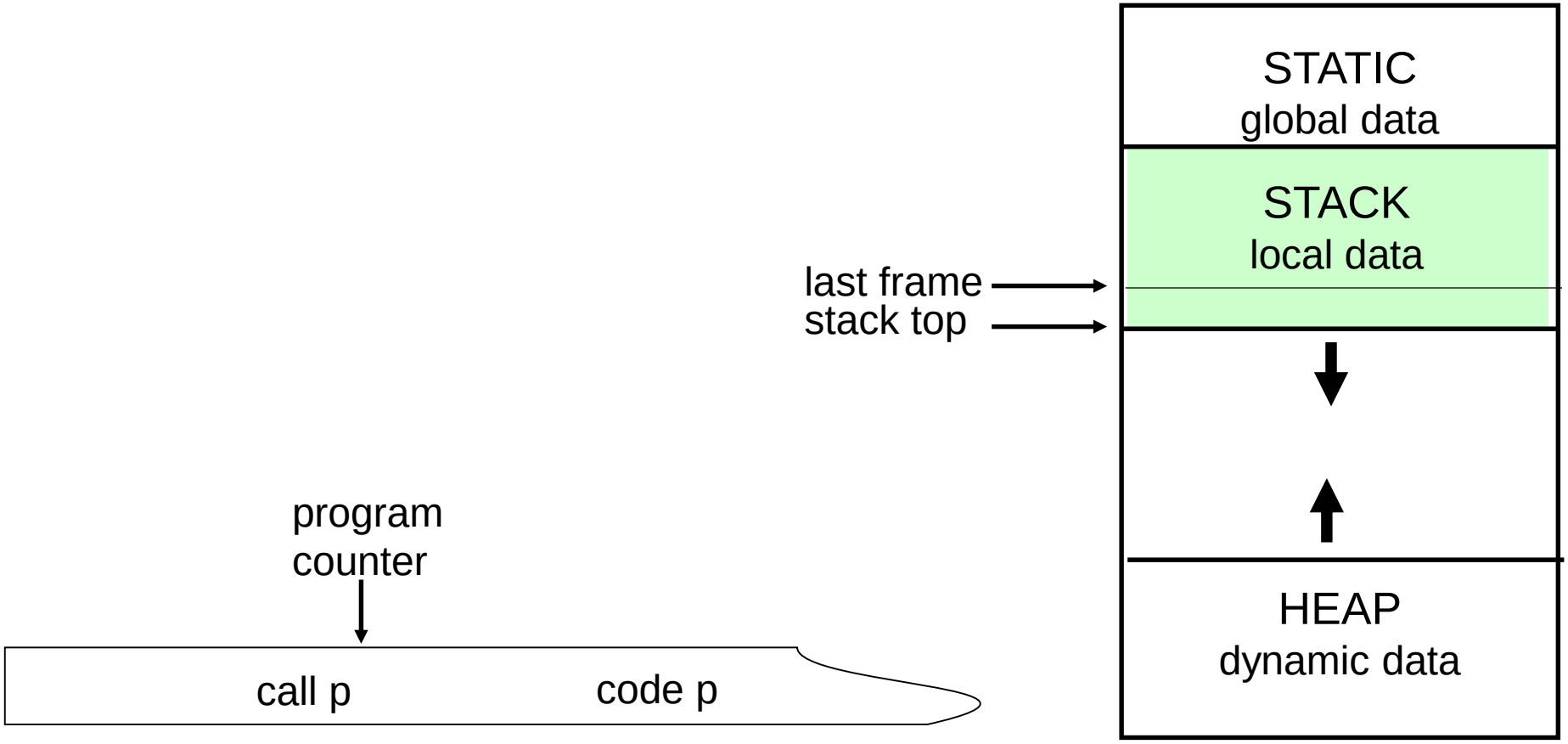
Activation records



Activation records

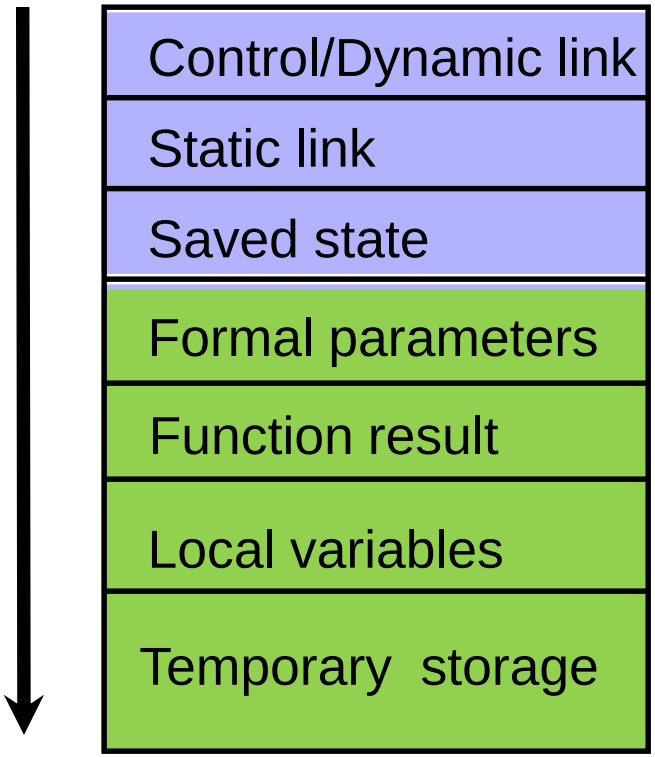


Activation records



Activation records (AR)

Data needed within a procedure invocation is collected in an *activation record* or *stack frame*.



Reference to AR of caller

Ref. to AR of callee‘s static parent

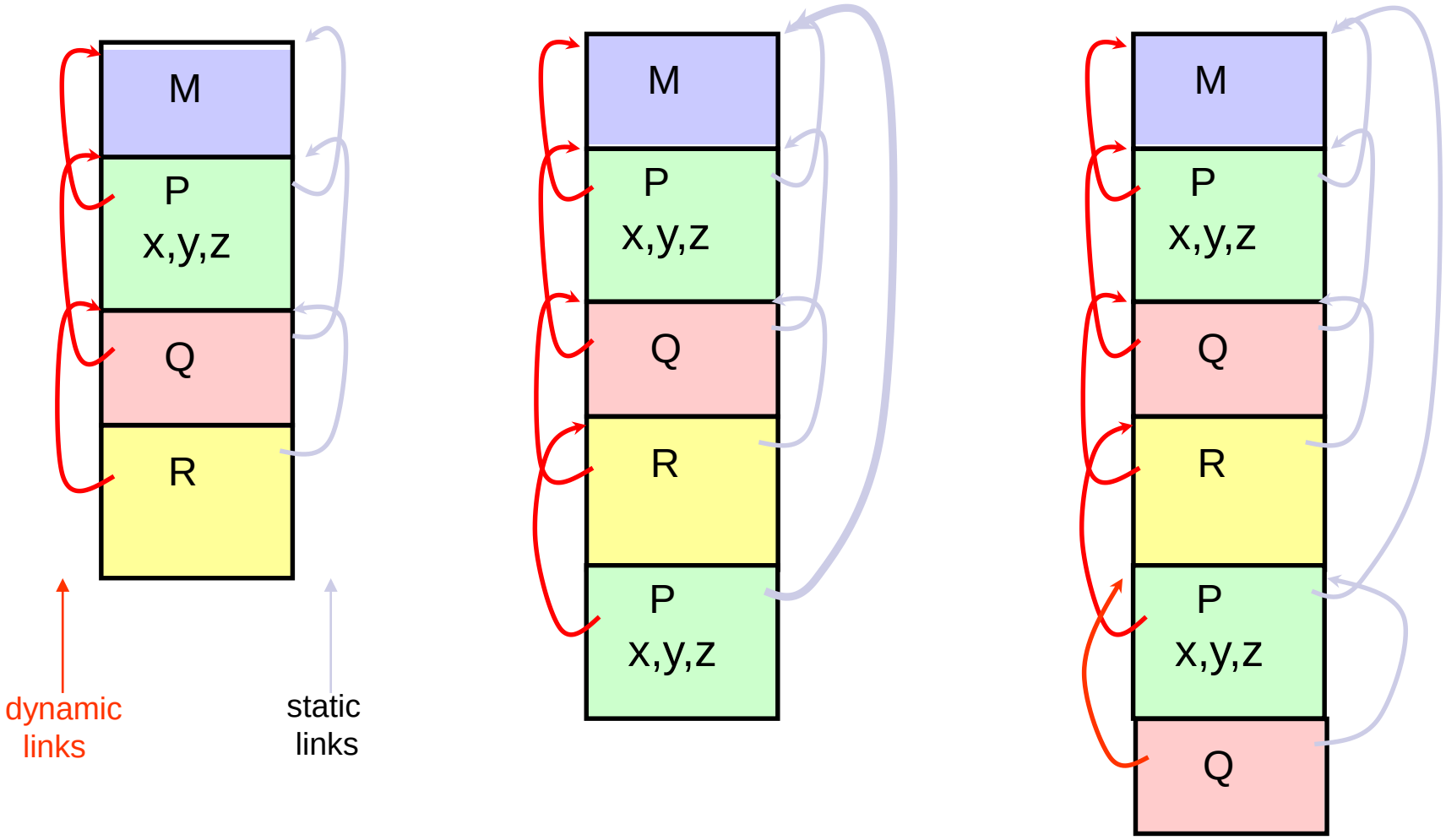
In particular return-address

Activation records

```
program M;
    function P(...): integer;
        var x,y,z: integer;
        function Q(…): integer;
            function R(…): integer;
            begin { R }
                ...; z := P(…); …
            end;  { R }
        begin { Q }
            …; y := R(…);…
        end; { Q }
    begin { P }
        …;   x := Q(… ); …
    end; { P }
begin { M }
    …; P(…); …
end. { M }
```

access of variable `y`
defined in an outer block

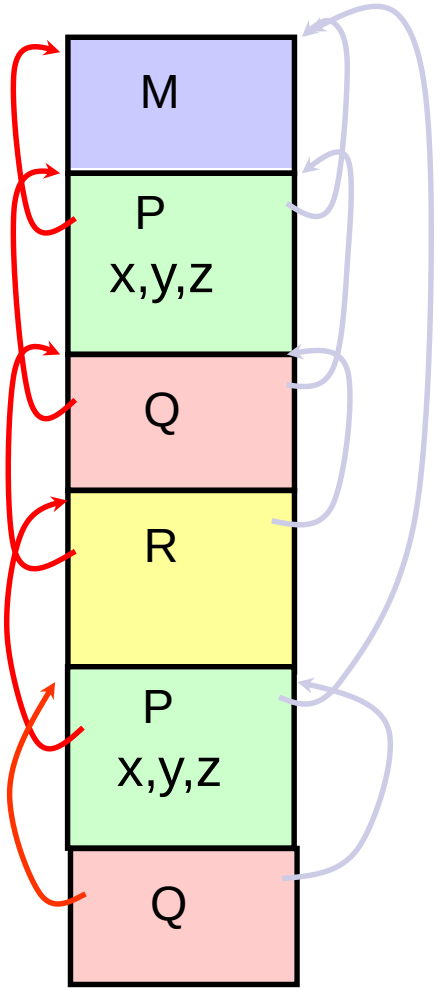
Activation records



Determining static link for new activation record:

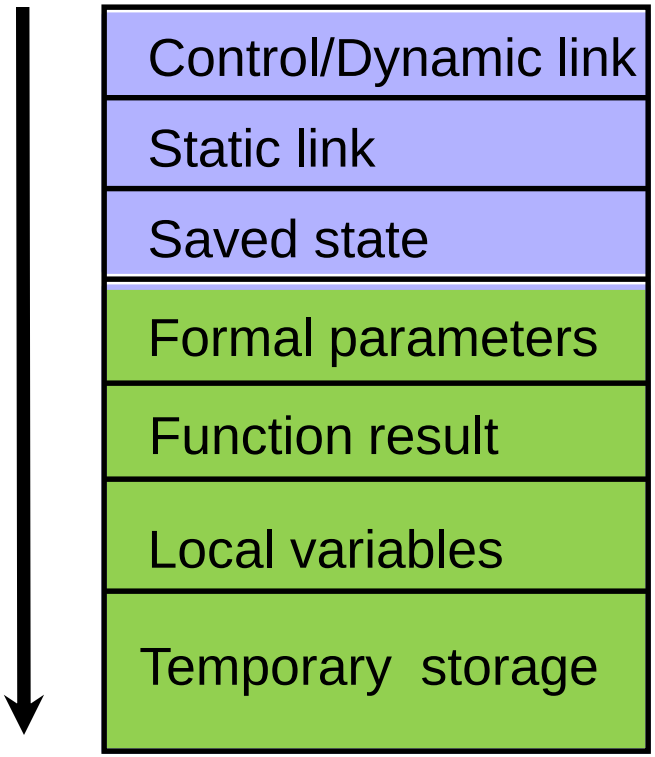
→ If m-n+1 = 0 use dynamic link;
otherwise follow static link chain
of caller for m-n+1 levels
(performed at run time)

m caller's static level (known at compile time)
n callee's static level (known at compile time)



Activation records (AR)

Data needed within a procedure invocation is collected in an *activation record* or *stack frame*.



Reference to AR of caller

Ref. to AR of callee‘s static parent

In particular return-address

An important issue:
Memory space
needed for data?!

Memory layout: elementary data types

- Boolean (sometimes absent or identified with integers, e.g. C, LISP)
- Integer (sometimes divided into several types like in Java's integer-types byte, short, integer, long, (BigInteger))
- Real (floating standard IEEE 745, sometimes divided into several types like Java's float, double)
- Characters
- Enumerations (e.g.: Pascal declaration

type month = (jan, feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec);

with implicit order jan < feb <…<dec ;

booleans and characters can be seen as predefined enumerations)

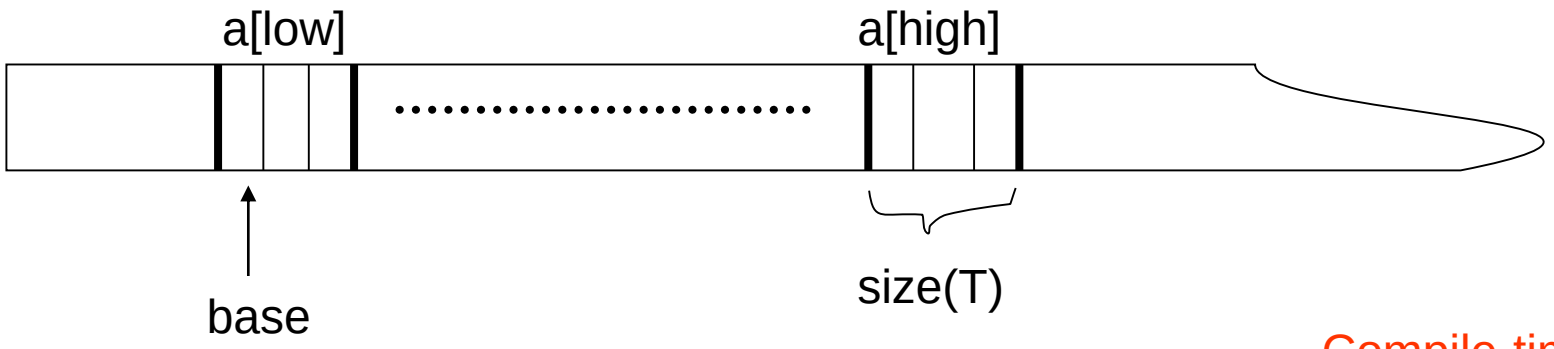
Memory layout: composite data types

- Arrays
- Records
- Sets
- Variant records
- Pointers

Memory layout: arrays

One-dimensional array: a: array[low..high] of T;

Layout in store



Addr(a[i]) = base + (i - low) * size(T)
= base – low * size(T) + i * size(T)
= $base_{red}$ + i * size(T)

Compile-time known (for static arrays)

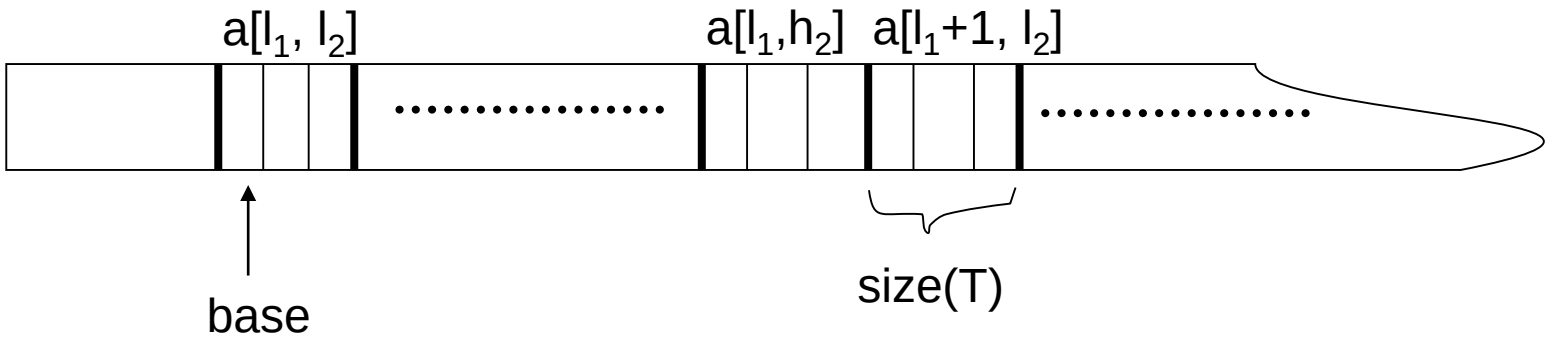
low = 0 in C-oriented languages

Memory layout: arrays

Memory layout of arrays:

Two-dimensional array: a: array[l_1..h_1,l_2..h_2] of T;

Layout in store



Row-major array layout

Memory layout

$$\text{Addr}(a[i, j]) = \text{base} + ((i-l_1) * (\underbrace{h_2-l_2+1}_{d_2}) + (j-l_2)) * \text{size}(T)$$

$$= \text{base} - (l_1 * d_2 + l_2) * \text{size}(T) + (i * d_2 + j) * \text{size}(T)$$

$$= \text{base}_{red} + (i * d_2 + j) * \text{size}(T)$$

Memory layout

n-dimensional array: a: array[$l_1..h_1,\ldots,l_n..h_n$] of T;

Let $d_i = h_i - l_i + 1$

$$\text{Addr}(a[i_1,..,i_n]) = \text{base} + \left(\sum_{j=1}^{n} (i_j - l_j) * \prod_{k=j+1}^{n} d_k\right) * \text{size}(T)$$

$$= \text{base} - \left(\sum_{j=1}^{n} l_j * \underbrace{\prod_{k=j+1}^{n} d_k}_{d^{(j)}}\right) * \text{size}(T) + \left(\sum_{j=1}^{n} i_j * \prod_{k=j+1}^{n} d_k\right) * \text{size}(T)$$

$$= \text{base}_{red} + \left(\sum_{j=1}^{n} i_j * d^{(j)}\right) * \text{size}(T)$$

Memory layout: arrays

Taxonomy of arrays (wrt. to allocation time, place):

- Static arrays:
  - Allocated at compile time
    (Example in C: static int a[10] ;)
  - Not included in PASCAL
  - Rapid address calculation (→ previous slides)

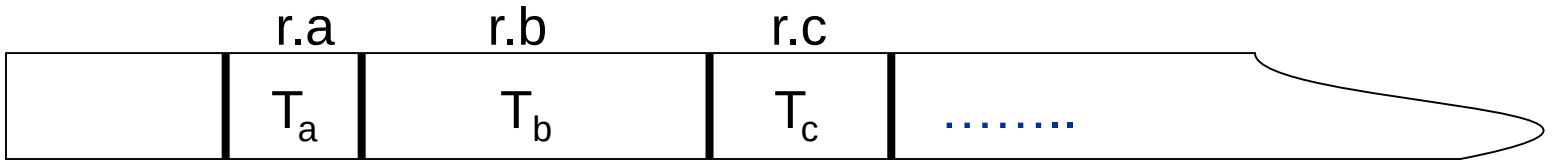
- Arrays with static bounds:
  - Bounds known at compile-time
    (Example in C: int a[10];
    Example in PASCAL: a = array[0..10] of integer;)
  - Stack-allocated at every procedure
  - base unknown at compile time
  - Fast address calculation

Memory layout: arrays

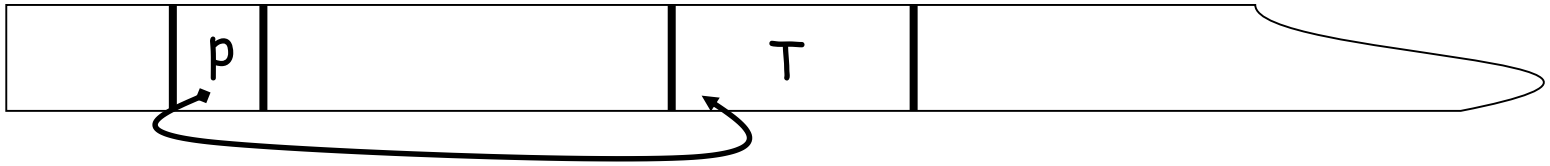
- Dynamic arrays:
  - Bounds known only at run-time
  - Stack-allocated:
    - Size determined once for every procedure activation
      (Example in ALGOL 60: ARRAY A[0:N,0:N];)
    - Technical issue: Storage has to be allocated at the end of a stack frame
    - More flexible than arrays with static bounds, but address calculation gets slower
  - Heap-allocated:
    - Size determined at any allocation point
      (Example JAVA: int[] a;……; a = new int[10];)
    - Very flexible but additional run-time overhead due to dereferencing

Memory layout: records

r: **record**
a: T_a;
b: T_b;
c: T_c
end;



Memory layout: pointers



- Pointers are addresses in data memory (heap, stack)

- Only heap pointers in PASCAL, JAVA.
  Non-null pointers generated only via new-instructions.

- In C, C++ stack as well as heap pointers
  Explicit address-off operator &.
  Pointer arithmetics

Memory layout: pointers

Problems with manual pointer deallocation:

① **Dangling pointers**: Pointers to deallocated heap locations

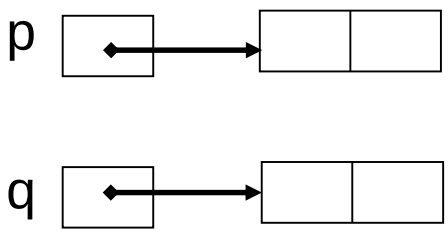
② **Memory leaks**: Unaccessible heap locations

Memory layout: pointers

Problems with manual pointer deallocation:

① **Dangling pointers**: Pointers to deallocated heap locations

② **Memory leaks**: Unaccessible heap locations



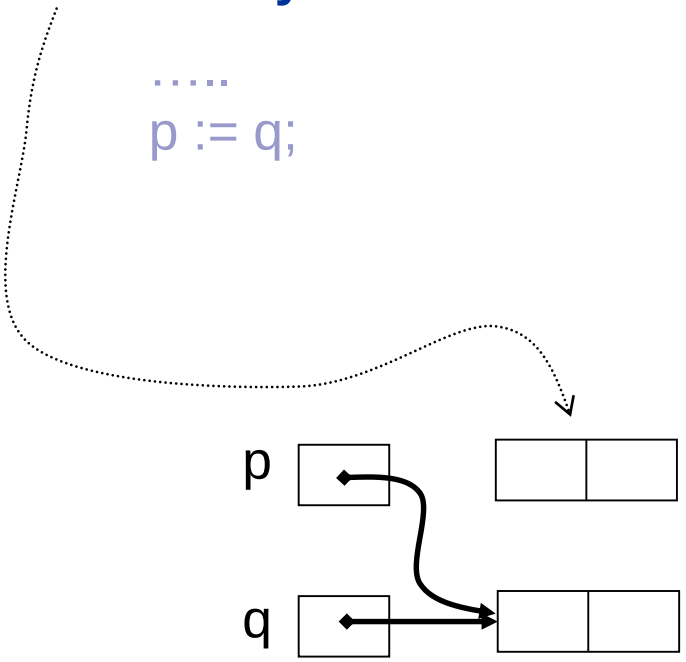
Memory layout: pointers

Problems with manual pointer deallocation:

① **Dangling pointers**: Pointers to deallocated heap locations

② **Memory leaks**: Unaccessible heap locations

```
…..
p := q;
```



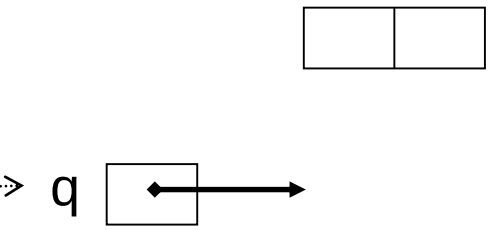
Memory layout: pointers

Problems with manual pointer deallocation:

① **Dangling pointers**: Pointers to deallocated heap locations

② **Memory leaks**: Unaccessible heap locations

```
…..
p := q;
dispose(p);
  ……
```



Memory layout: pointers

- The dangling pointer problem is avoided by using automatic memory deallocation only (*garbage collection*) like in JAVA.

- Memory leaks can also be reliably avoided by advanced garbage collection methods (not reference counting).

However, garbage collected programming languages are not well-suited for real-time sensitive applications.

Memory layout: pointers

C.A.R. Hoare (1973): “Their introduction into high-level languages may be a step backward from which we may never recover.”

Runtime system

- Managing data accesses
  - Stack: Procedure (method) invocations
  - Heap: Garbage collection
  - File system / IO
- Scheduling
  - Concurrency
  - Non-determinism
- Monitoring
  - Exceptions
  - Debugging

Garbage collection

- Nature of object oriented programming: Almost everything is an object (Smalltalk, Java) and thus heap-located.
- Heap management is a central issue
- Heap-allocated objects that are not reachable from stack-located program variables (possibly through chains of pointers) are called garbage

Garbage collection

Reference counting

- Keep track on the number of references to a heap allocated record. Reference count stored within each record.

- Garbage collector emits extra instructions.
  If pointer $p \rightarrow r_1$ is changed towards $p \rightarrow r_2$ perform:

  1) Increment the reference count of r_2

  2) Decrement the reference count of r_1.
     If reference count of r_1 reaches zero, then r_1 is put on the freelist.

- New heap allocated objects use locations from freelist.

Garbage collection

Reference counting

Pros:

- Simple
- Smooth. Suited for real-time sensitive applications

Cons

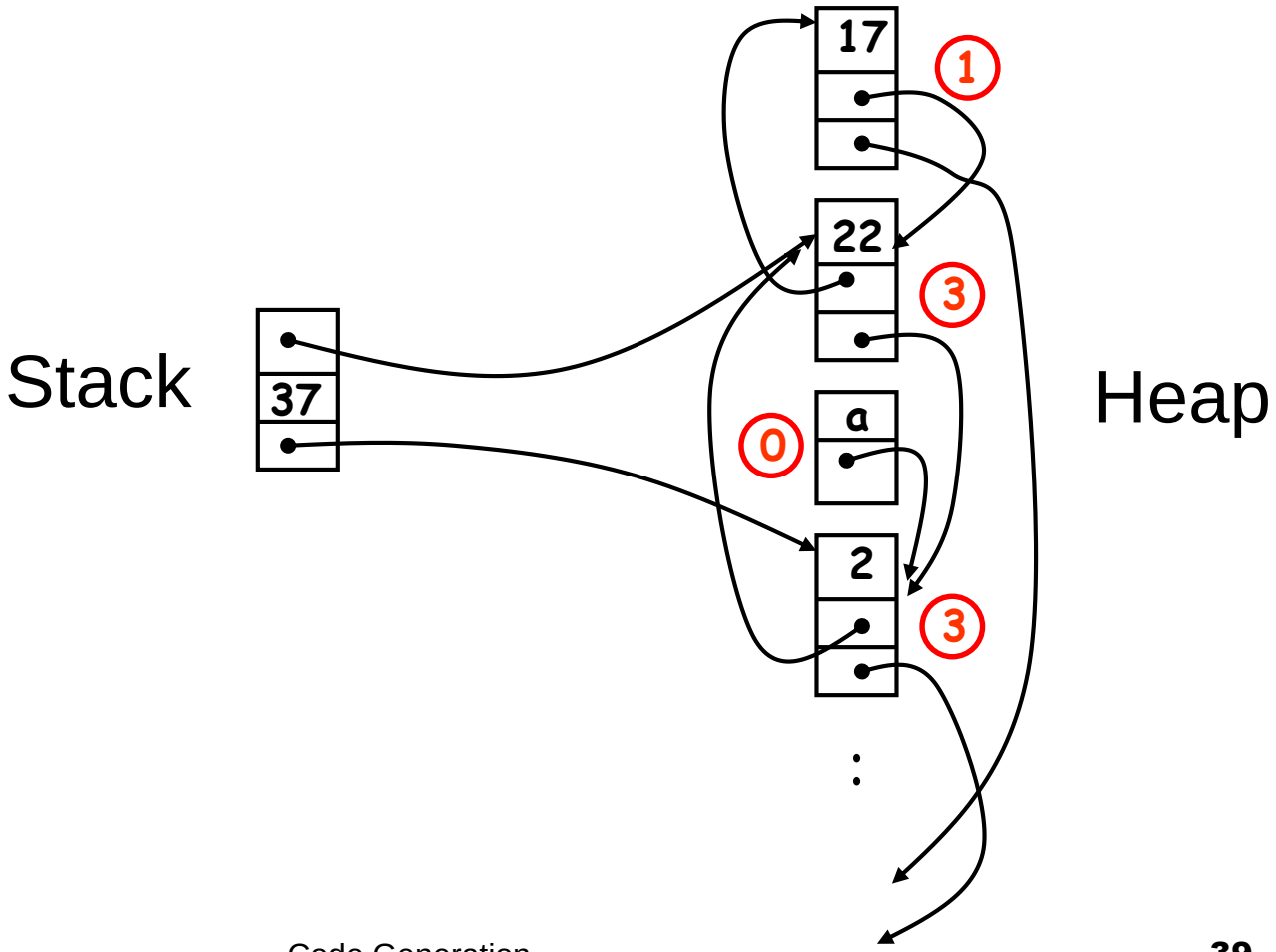
- Cycles of garbage cannot be reclaimed $\rightarrow$ memory leakage
  (see example on the next slide)
- Expensive!

  Even increment/decrement-operations take several machine instructions.

  Naive implementation has to perform updates on every assignment
  (some can be eliminated using static program analysis)

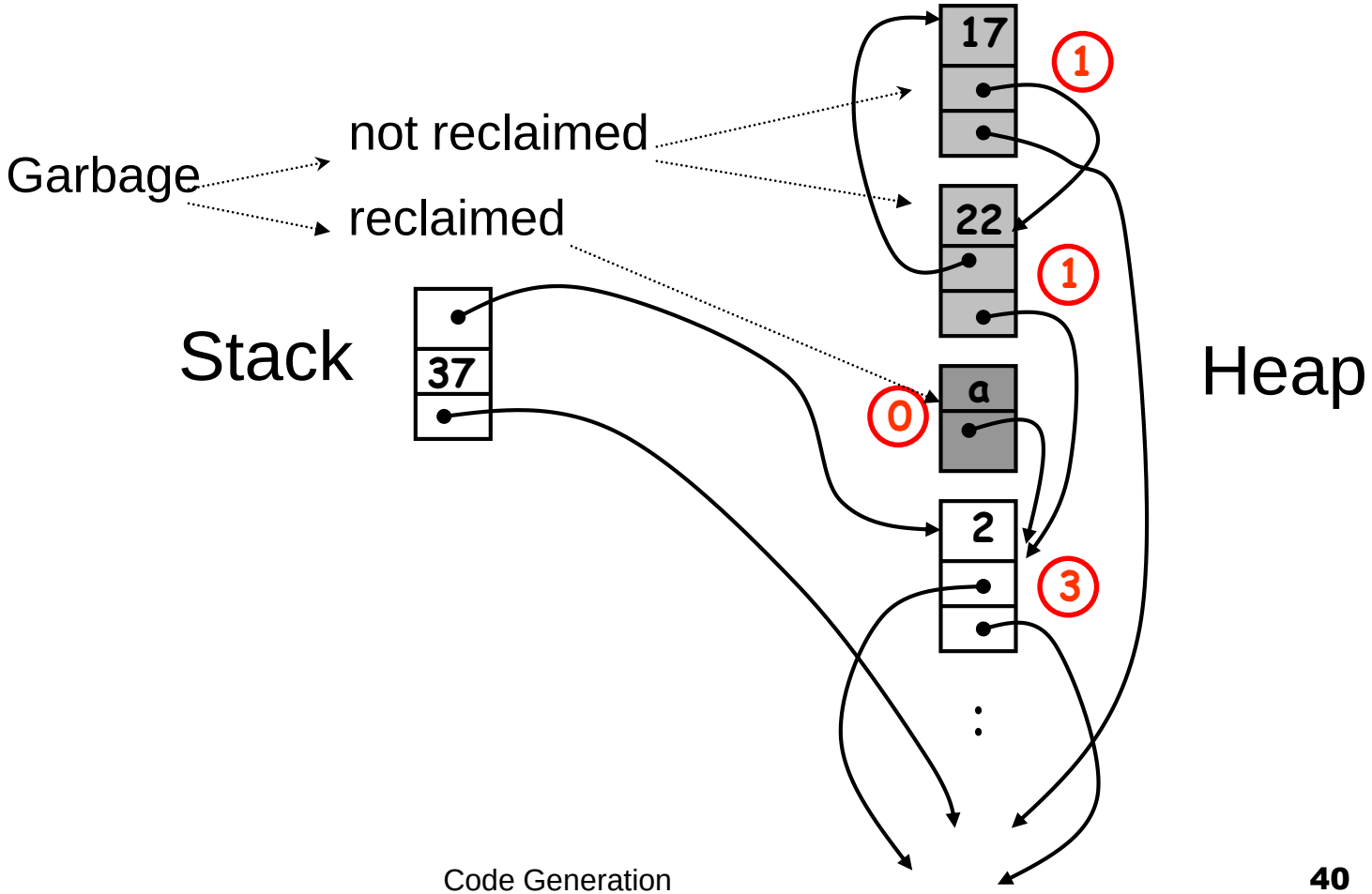
Garbage collection

Reference counting



Garbage collection

Reference counting



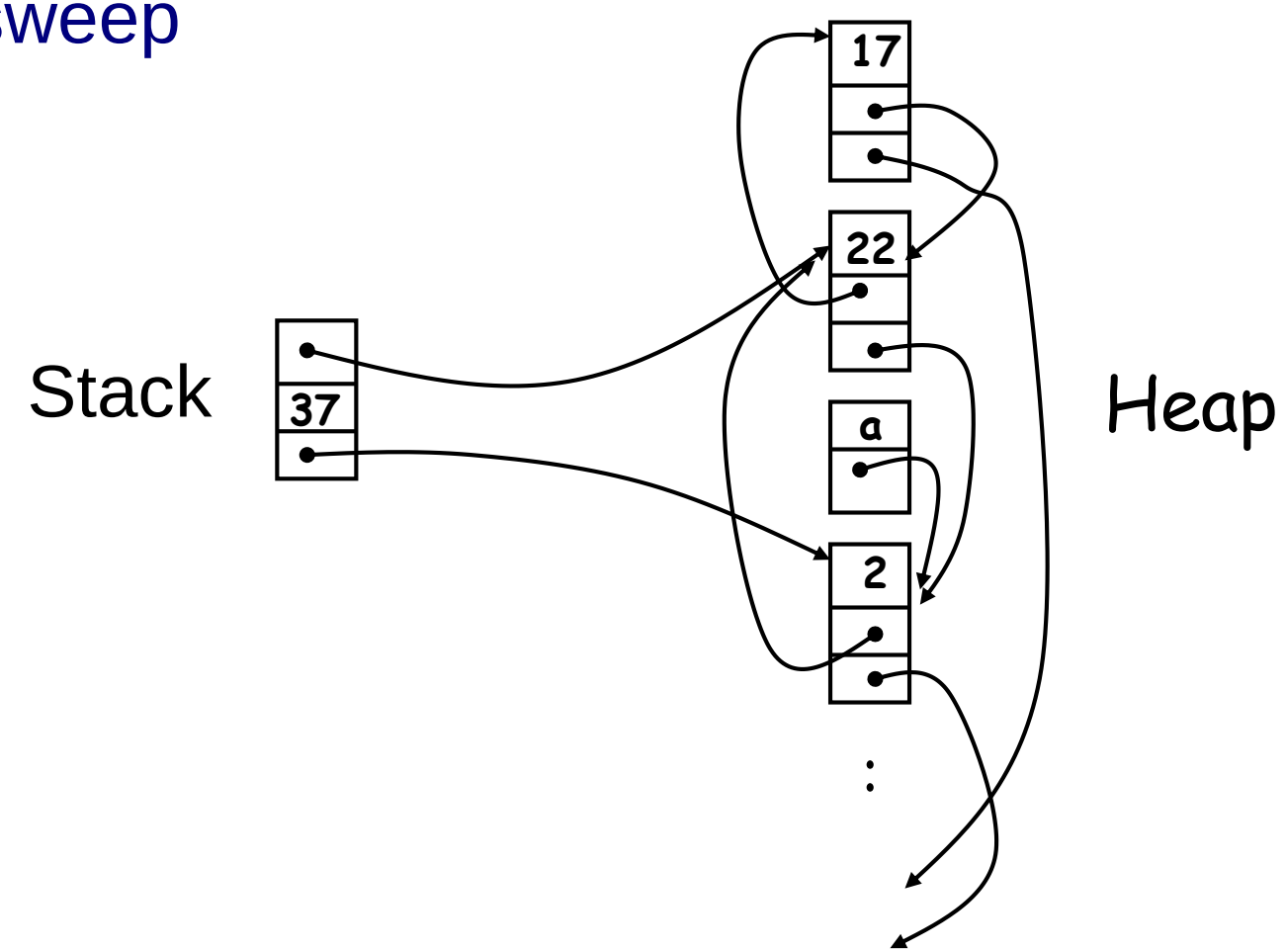
Garbage collection

Mark and sweep

- Program variables and heap-allocated records form a directed graph.
- A graph-traversal algorithm like depth-first search (DFS) is used in order to mark all reachable nodes.
  (→ *mark phase*)
- Nodes that are not marked must be garbage and should be reclaimed (put on freelist). (→ *sweep phase*)
- New heap allocated objects use locations from freelist.

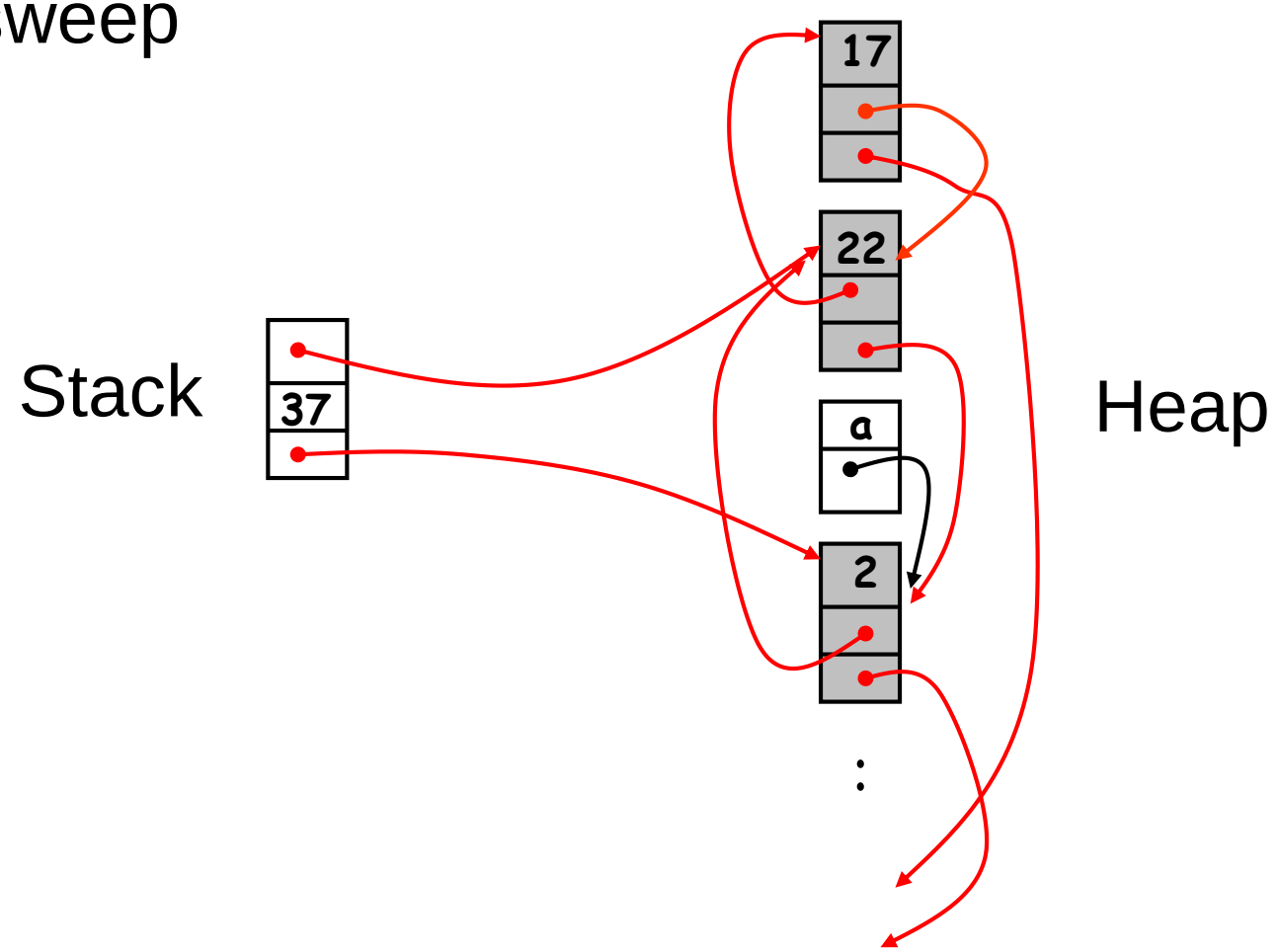
Garbage collection

Mark and sweep

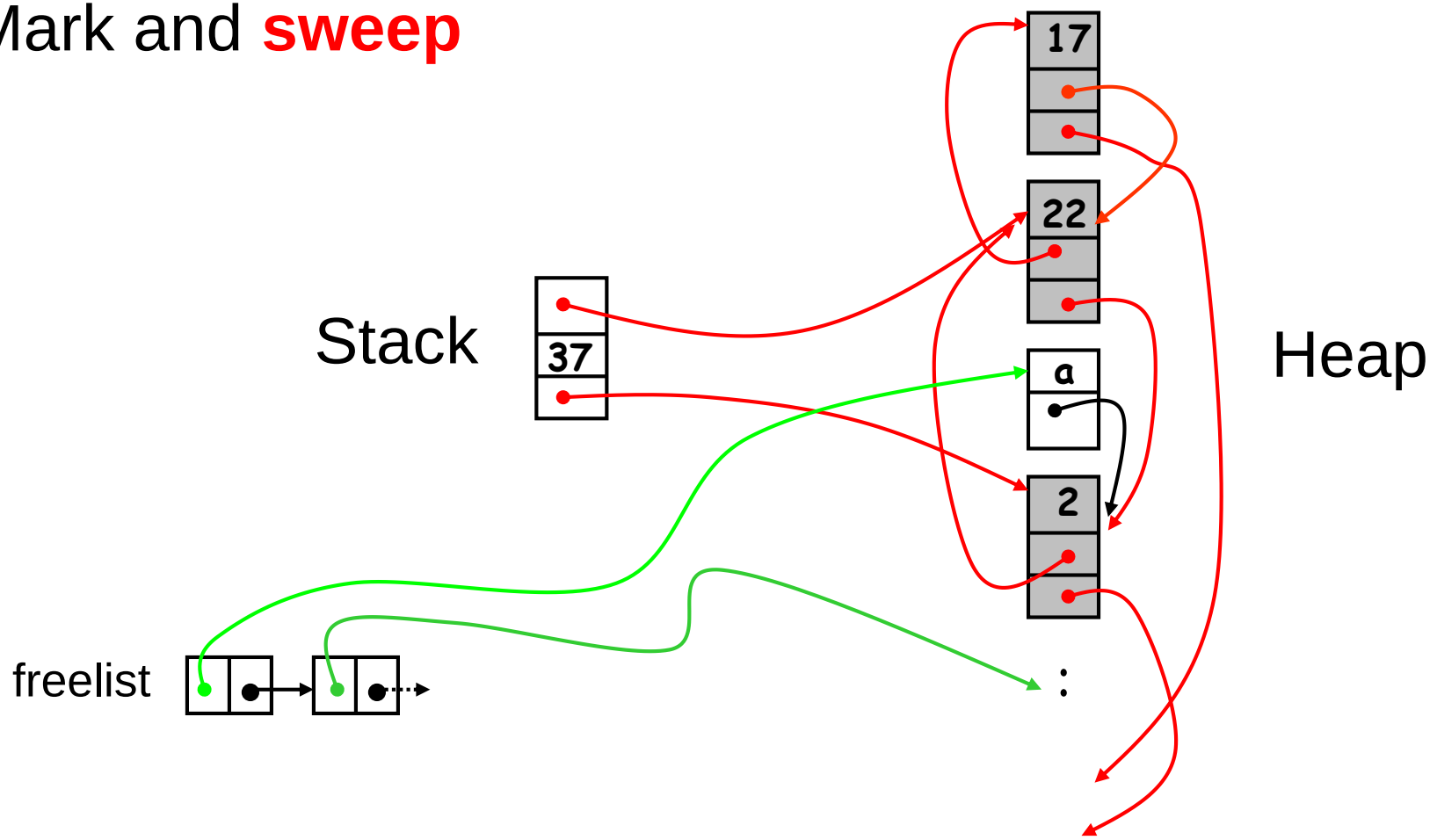


Garbage collection

Mark and sweep



Mark and **sweep**



Garbage collection

Mark and sweep

Pros:

- No memory leakage
- Simple to implement

Cons

- Expensive!

  Large portions of the heap have to be explored (Some improvements exist)

- Not smooth
- Overhead to manage freelist

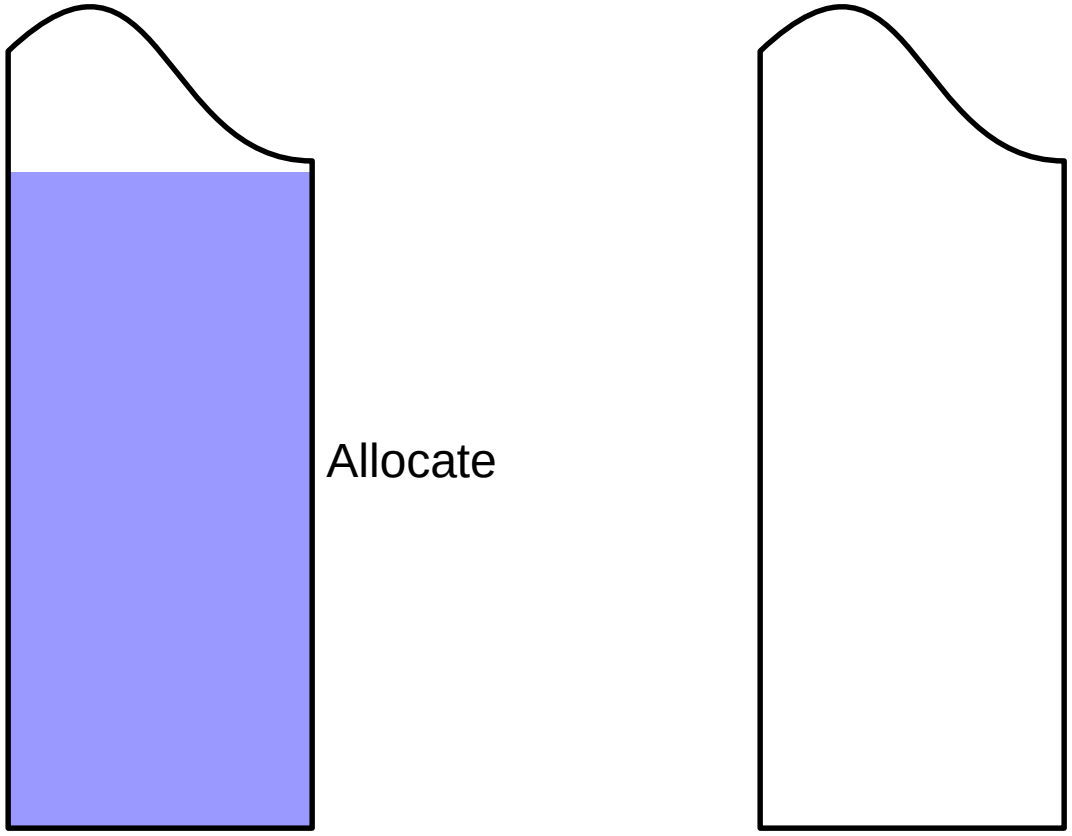
Garbage collection

Copying collection

- Essentially: Mark & sweep + compaction
  - No freelist
  - Two separate heaps (*from-area*, *to-area*) switching roles
- Reachable records of from-area are copied into consecutive front segment of to-area
- Triggered by heap-limits

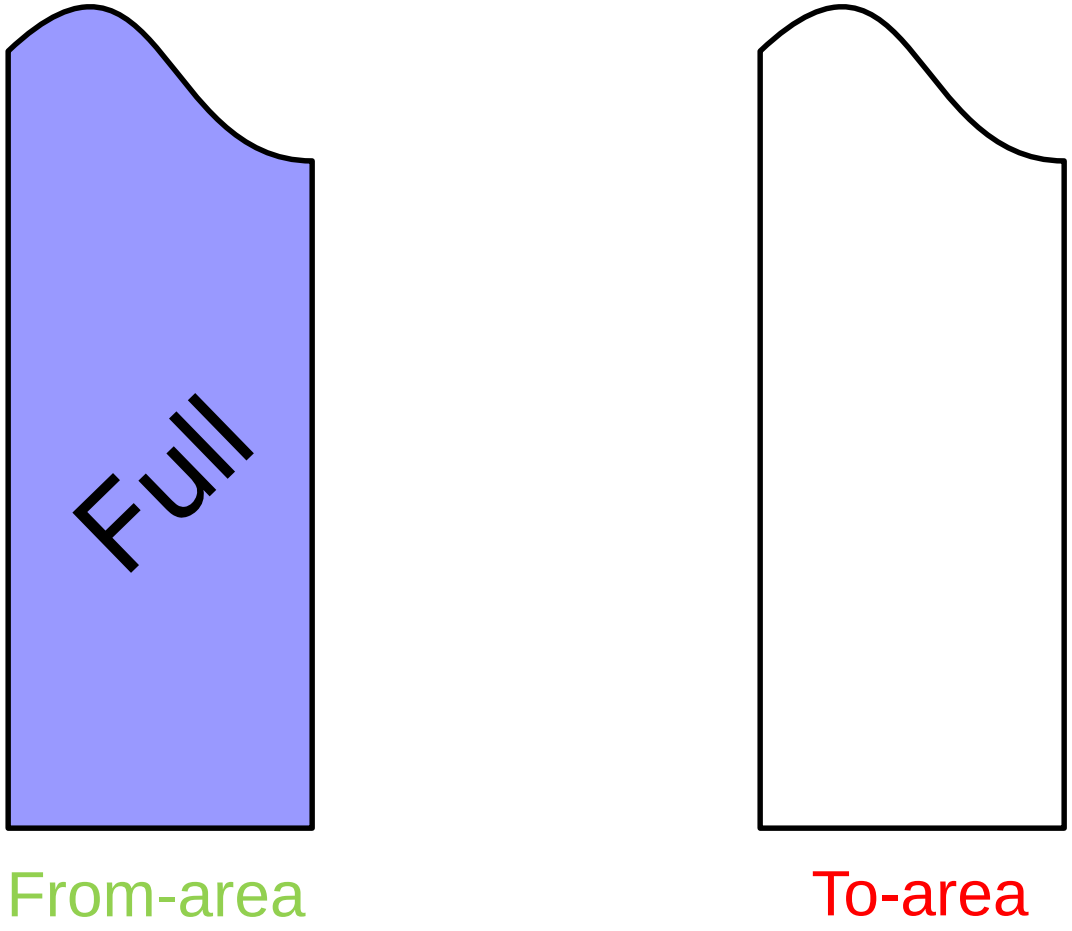
Garbage collection

Copying collection



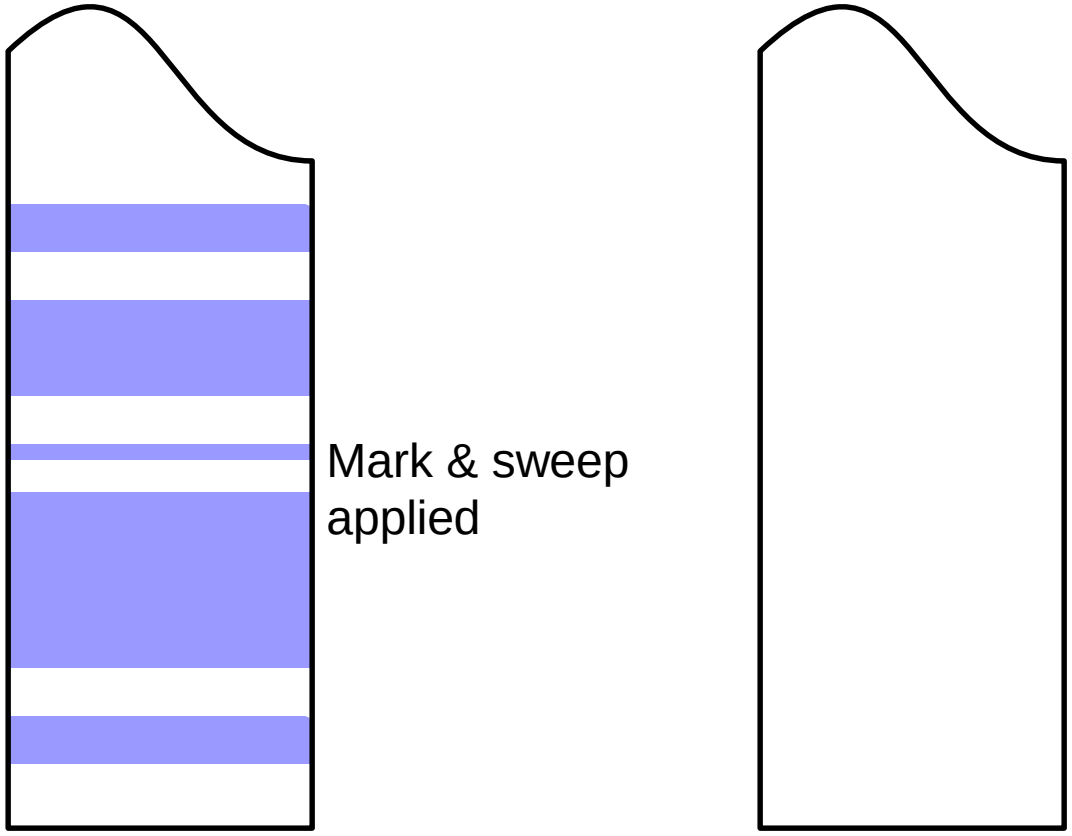
Garbage collection

Copying collection



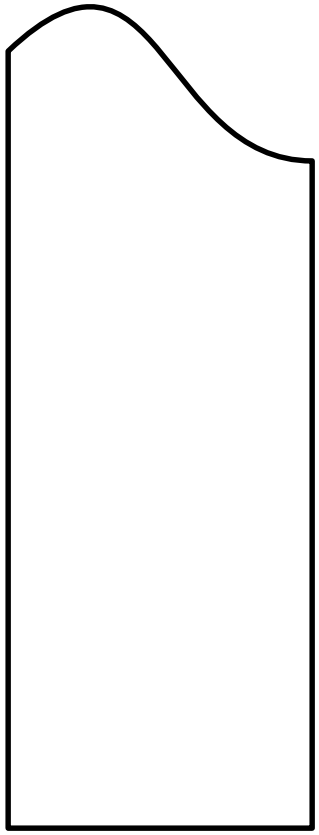
Garbage collection

Copying collection

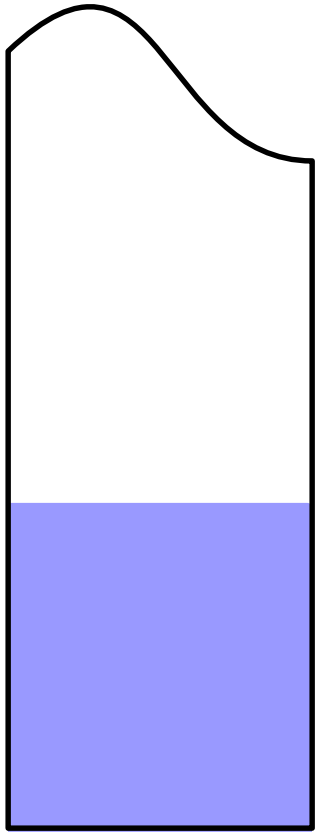


Garbage collection

Copying collection



From-area

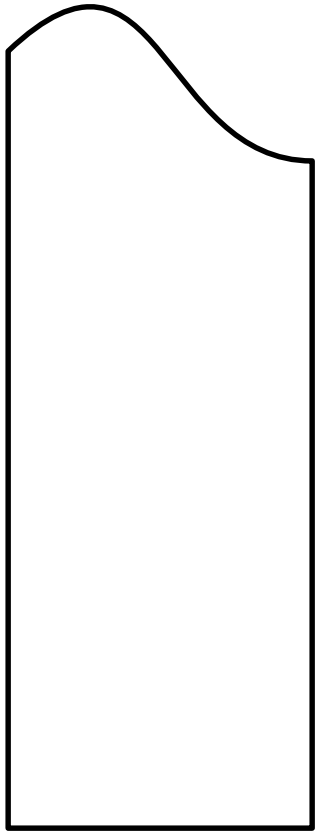


To-area

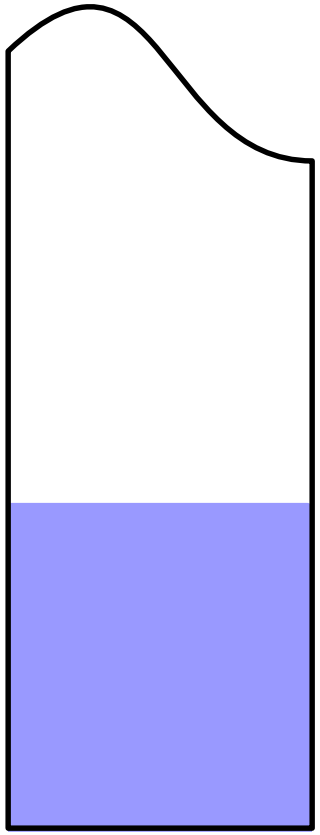
Copying & compactification completed

Garbage collection

Copying collection



To-area

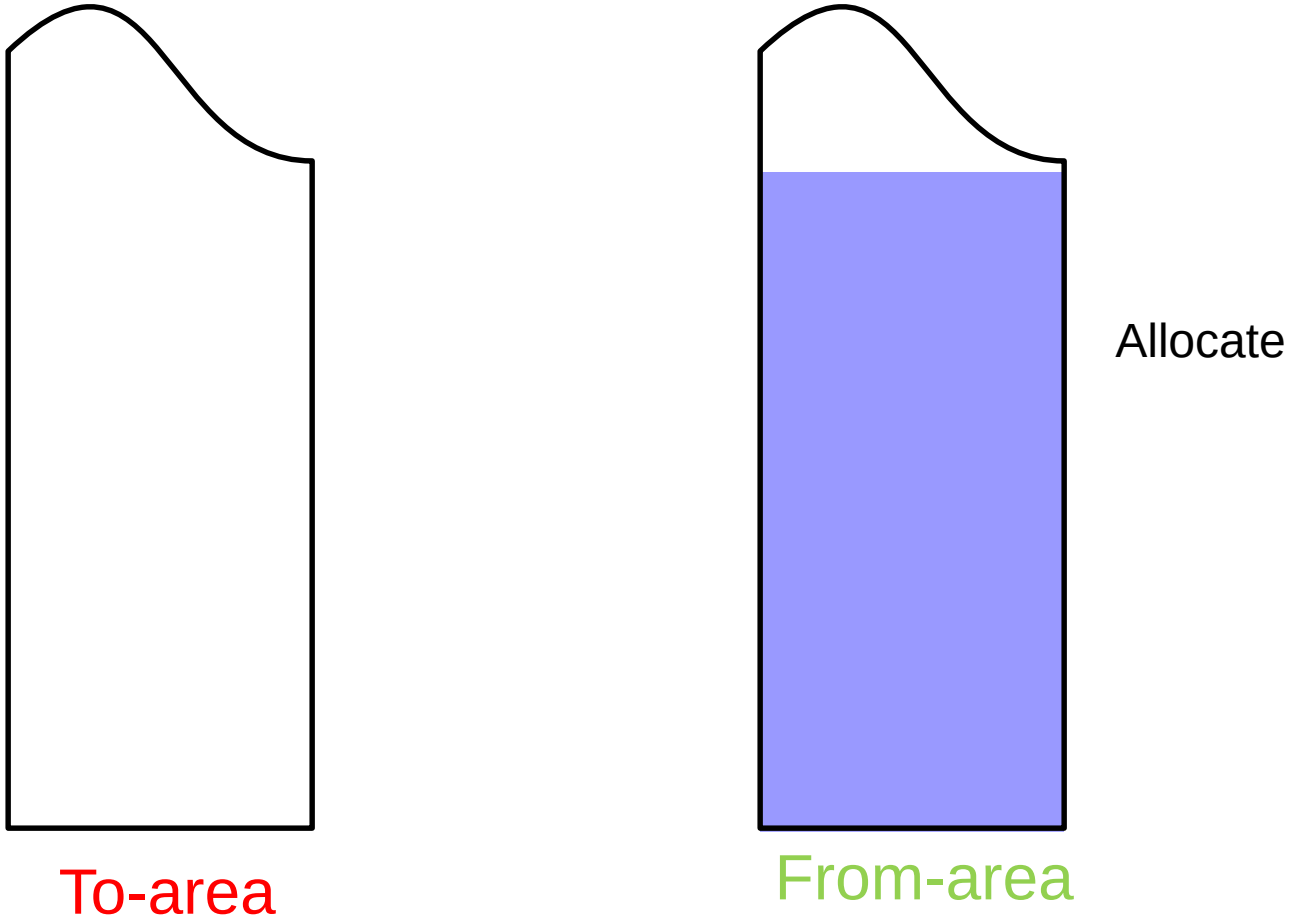


From-area

Copying & compactification completed

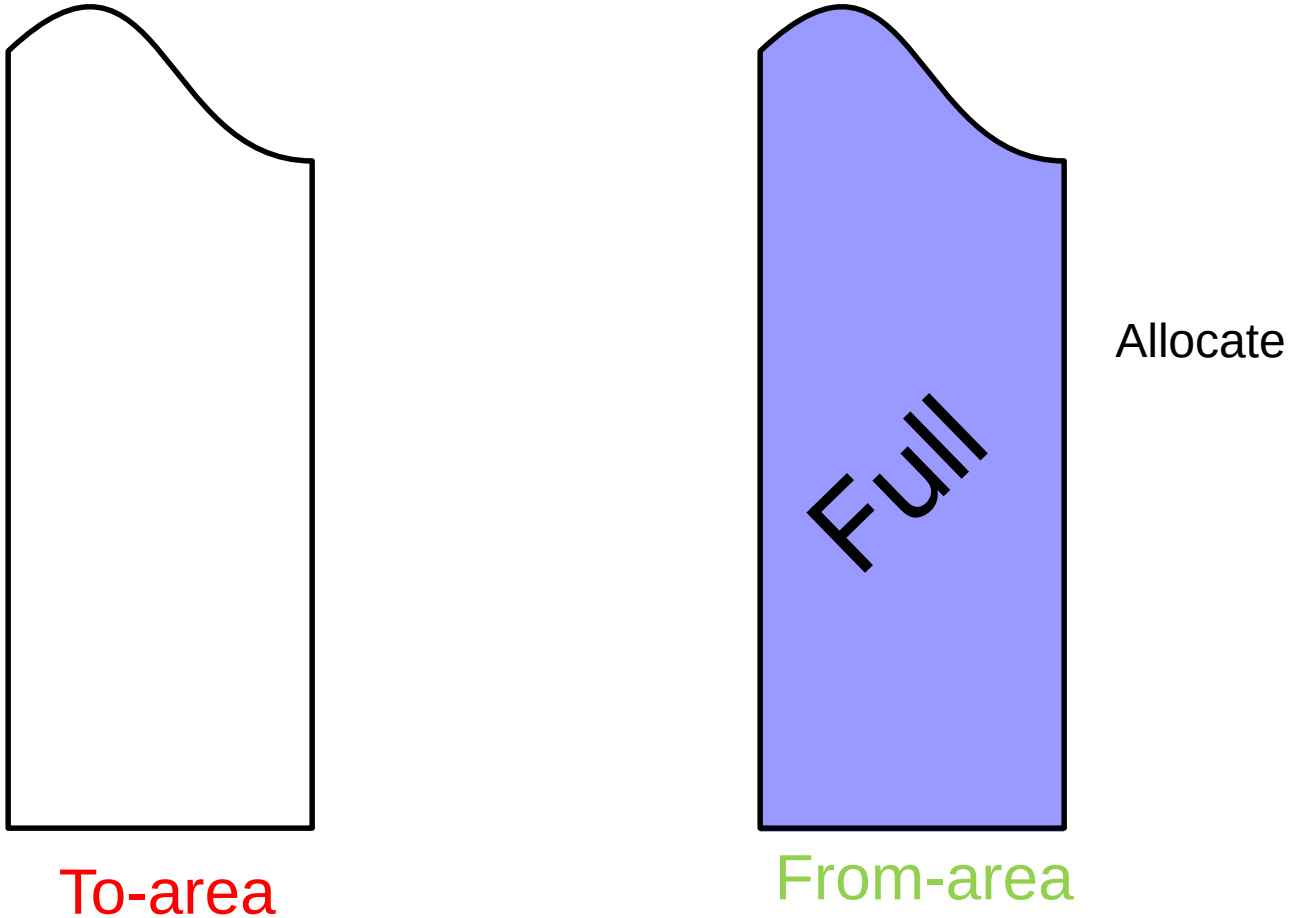
Garbage collection

Copying collection



Garbage collection

Copying collection



Garbage collection

Copying collection

… and so on

Garbage collection

Generational garbage collection

- Basic observation (empirically justified):
  - Newly created objects are likely to die soon.
  - Old objects will most likely survive many more collections.

- Divide the heap into generations G_0, G_1, G_2,…,G_k
  - Often only 2 or 3 generations (nursery,…)

Garbage collection

Generational garbage collection

Idea:

- Each generation is exponentially larger than the one before, e.g.: $G_0 \equiv ½$ MB, $G_1 \equiv 2$ MB , $G_2 \equiv 8$ MB, ….
- Objects are promoted to the next generation, if they survived two or three collections
- G_0 is collected most often, G_0 together with G_1 more rarely, G_0 together with G_1 and G_2 even more rarely, etc.

Code Generation

Idea:

- Use intermediate code
  - Parse tree, AST, Symbol Table
  - Three-Address-Code

- Three-Address-Code
  - can be generated from AST
  - turning the tree representation into a linear sequence of instructions
  - use (at most) three *symbolic* addresses

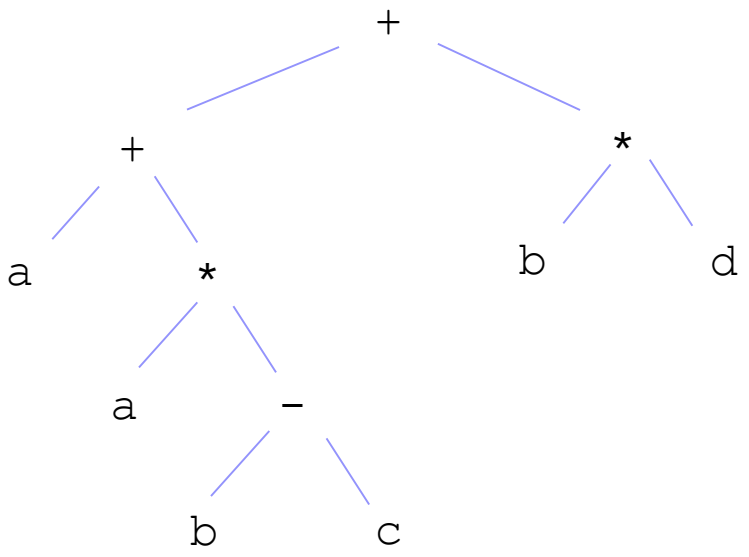
  - Symbolic addresses are mapped to memory addresses or registers later

Code Generation

Three-Address-Code (TAC): Expressions

`a+a*(b-c)+b*d`

AST:



TAC:

$$
\begin{aligned}
t_0 &= b - c \\
t_1 &= a * t_0 \\
t_2 &= a + t_1 \\
t_3 &= b * d \\
t_4 &= t_2 + t_3
\end{aligned}
$$

Code Generation

Three-Address-Code (TAC): Expressions

`a + a * (b - c) + b * d`

TAC:

$t_0 = b - c$
$t_1 = a * t_0$
$t_2 = a + t_1$
$t_3 = b * d$
$t_4 = t_2 + t_3$

generated code, e.g.:

op	dest	src	[src_2]
LD	R0	b	
SUB	R0	R0	c
LD	R1	a	
MUL	R1	R1	R0
LD	R2	a	
ADD	R2	R2	R1
LD	R3	b	
MUL	R3	R3	d
ADD	R4	R2	R3

Full example: assignments

- **Input stream:**

```
cost = (price + tax) * 6
```

- **Token stream:**

```
<ID,1> <=> <(> <ID,2> <+> <ID,3> <)> <*> <NUMBER,4>
```

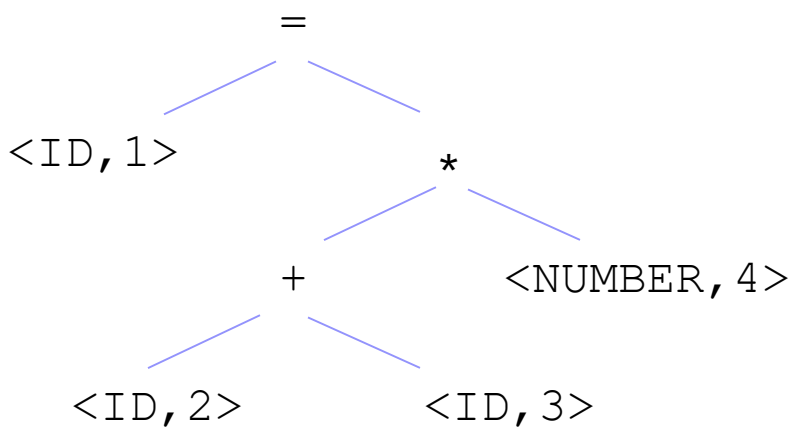
- **Symbol Table:**

1	cost
2	price
3	tax
4	6

Full example: assignments

<ID,1> <=> <(> <ID,2> <+> <ID,3> <)> <*> <NUMBER,4>

- **AST:**



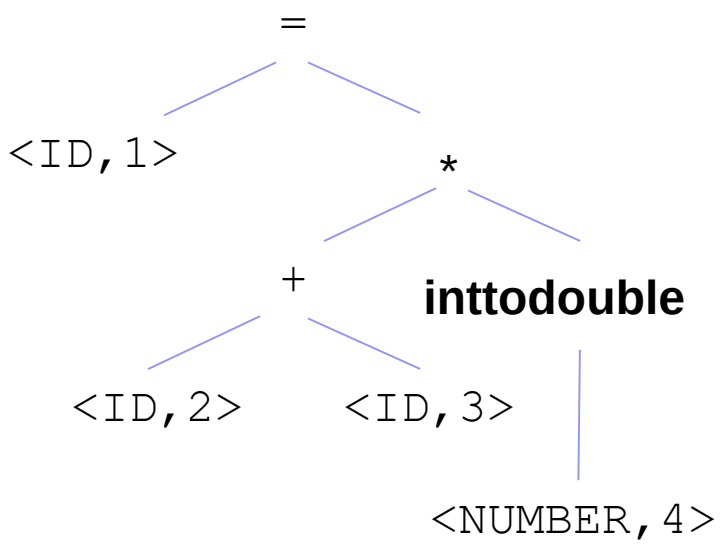
- **Symbol Table:**

1	cost	local	double
2	price	local	double
3	tax	local	double
4	6	constant	int

Full example: assignments

<ID,1> <=> <(> <ID,2> <+> <ID,3> <)> <*> <NUMBER,4>

- **modified AST:**



- **Symbol Table:**

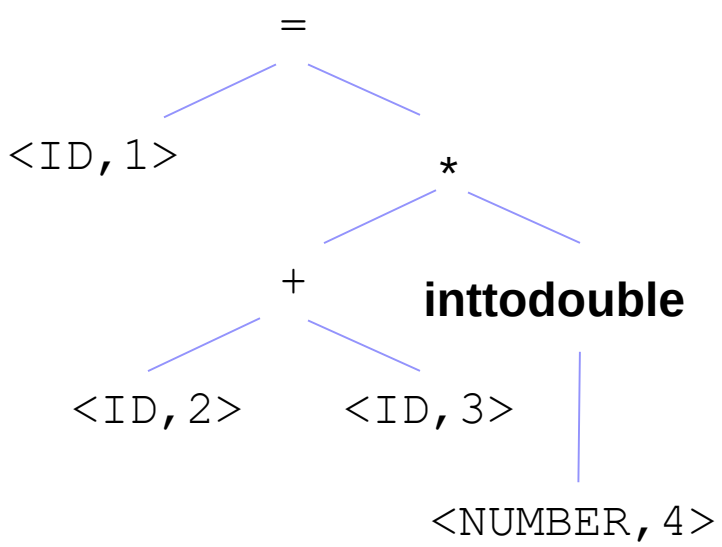
1	cost	local	double
2	price	local	double
3	tax	local	double
4	6	constant	int

after type checking

Full example: assignments

<ID,1> <=> <(> <ID,2> <+> <ID,3> <)> <*> <NUMBER,4>

- **modified AST:**



- **TAC:**

$t_1 = \text{inttodouble}(6)$
$t_2 = id_2 + id_3$
$t_3 = t_2 * t_1$
$id_1 = t_3$

- **optimized TAC:**

$t_1 = id_2 + id_3$
$id_1 = t_1 * 6.0$

Full example: assignments

<ID,1> <=> <(> <ID,2> <+> <ID,3> <)> <*> <NUMBER,4>

- **optimized TAC**

$t_1 = id_2 + id_3$
$id_1 = t_1 * 6.0$

- **generated code, e.g.:**

```
LD  R0 ID2
ADD R0 R0 ID3
MUL R0 R0 #6.0
ST  ID1 R0
```

Code generation: conditionals

if (cond) S_{then} else S_{else}

```
code(cond)              // result in r0
if r0=0 goto lelse
code(Sthen)
goto lexit
lelse:
code(Selse)
lexit:
```

Code generation: loops

```
while (cond) S_body
```

```
l_loop:
code(cond)              // result in r_0
if r_0=0 goto l_exit
code(S_body)
goto l_loop
l_exit:
```

Code generation: switch-case

```
switch (e)
    case c1: S1;
      :
    case cn: Sn;
    default: Sd;
```

Code generation: switch-case

```
code(e)                    // result in r0
goto l_base+r_0            // computed jump
l_base:
   :
goto l_1
   :                       // Jumptable
goto l_n
goto l_default
l_1: code(S_1)
   :
l_n: code(S_n)
l_default: code(S_d)
```

Stack-based code generation: assignments

```
x = 2 * (x - y);
```

Register-based
(TAC)

```
t1 = 2
t2 = x-y
t1 = t1*t2
x  = t1
```

Stack-based
(e.g. Java Byte-Code)

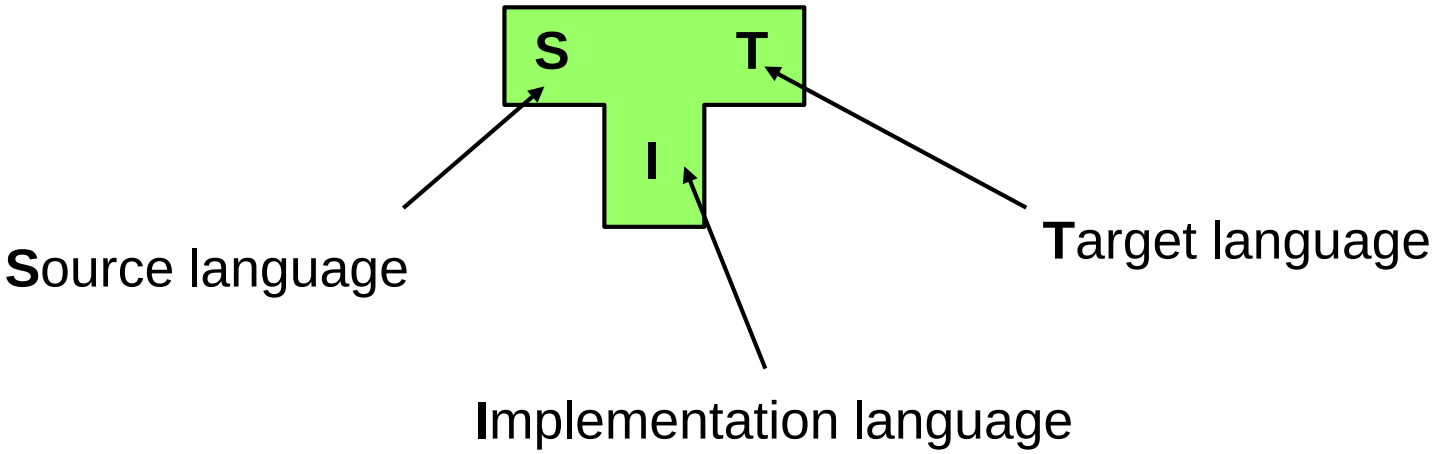
```
ICONST_2
ILOAD 1          // x
ILOAD 2          // y
ISUB
IMUL
ISTORE 1         // Assign
```

Bootstrapping

- Modern compilers translate to target languages for which compilers or interpreters exist.
- Generation of machine code is unnecessary.
- Appropriate composition of existing compilers and/or interpreters with “simple-to-write translators” (if necessary)

T-diagrams

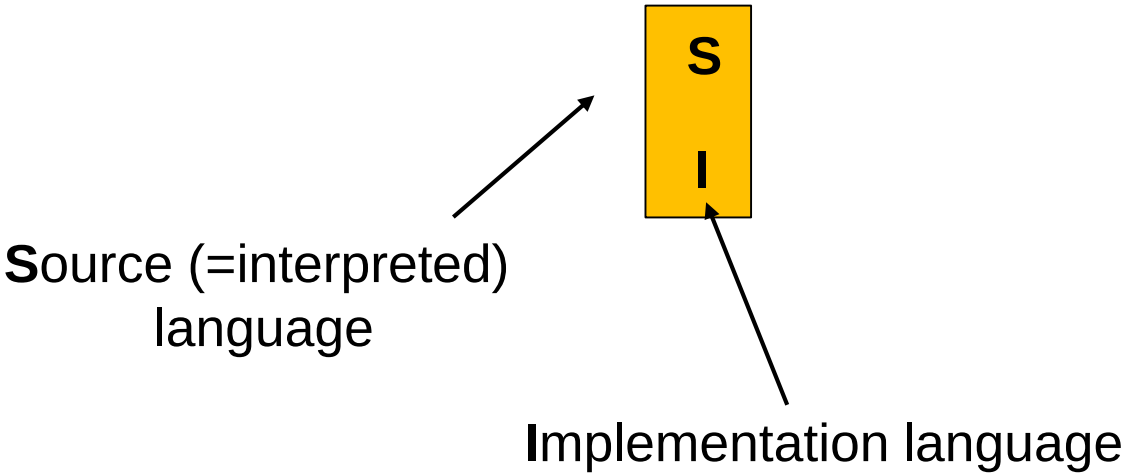
For compilers:



To be read as: Compiler from S to T written in I

T-diagrams

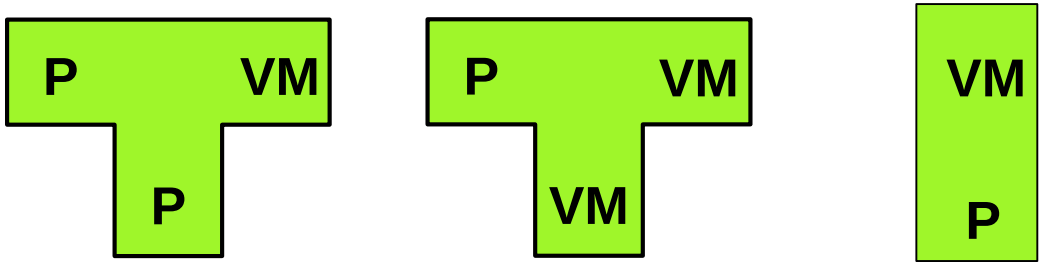
For interpreters:



To be read as: Interpreter for S written in I

Bootstrapping

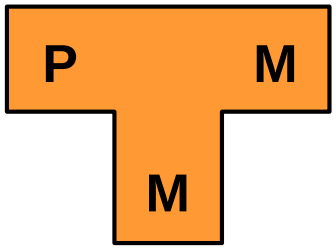
Provided:



ETH Zürich Pascal portable compiler

(P = Pascal, VM = Pascal P-code (virtual machine))

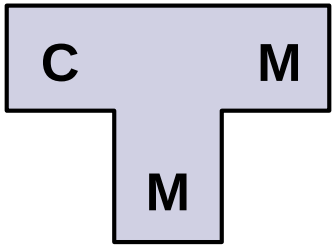
Goal:



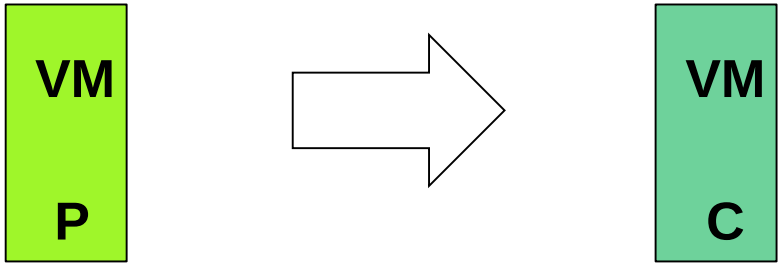
?

Bootstrapping

Available:

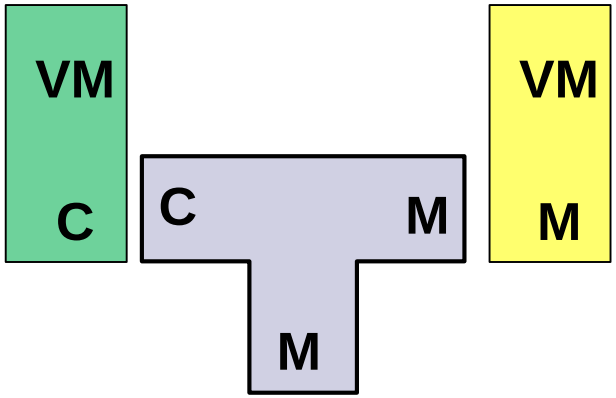


1. Rewrite:



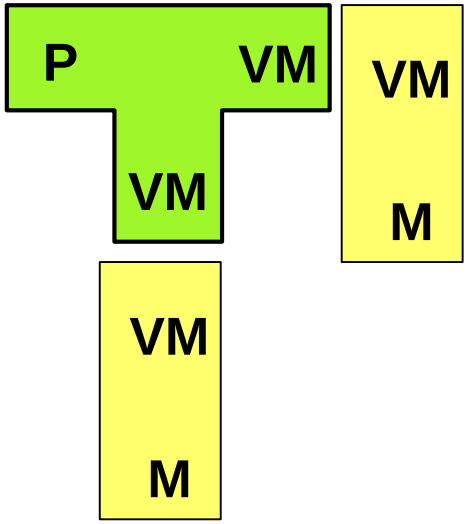
Bootstrapping

2. Compile interpreter:



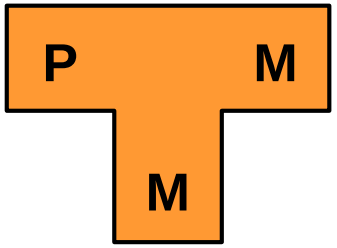
Bootstrapping

3. Yields (interpretive) compiler:



But recall our …

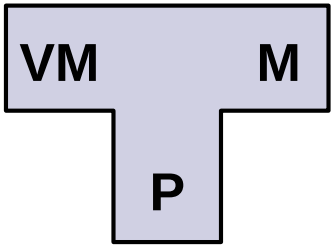
Goal:



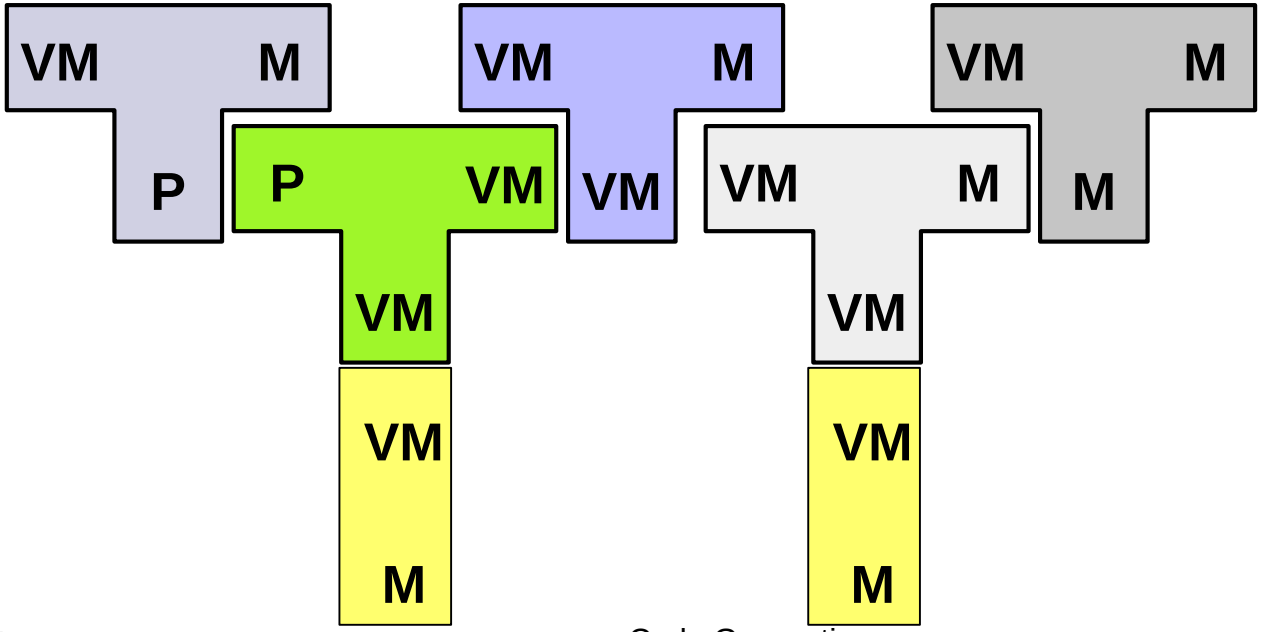
?

Bootstrapping

4. Hand-written backend:

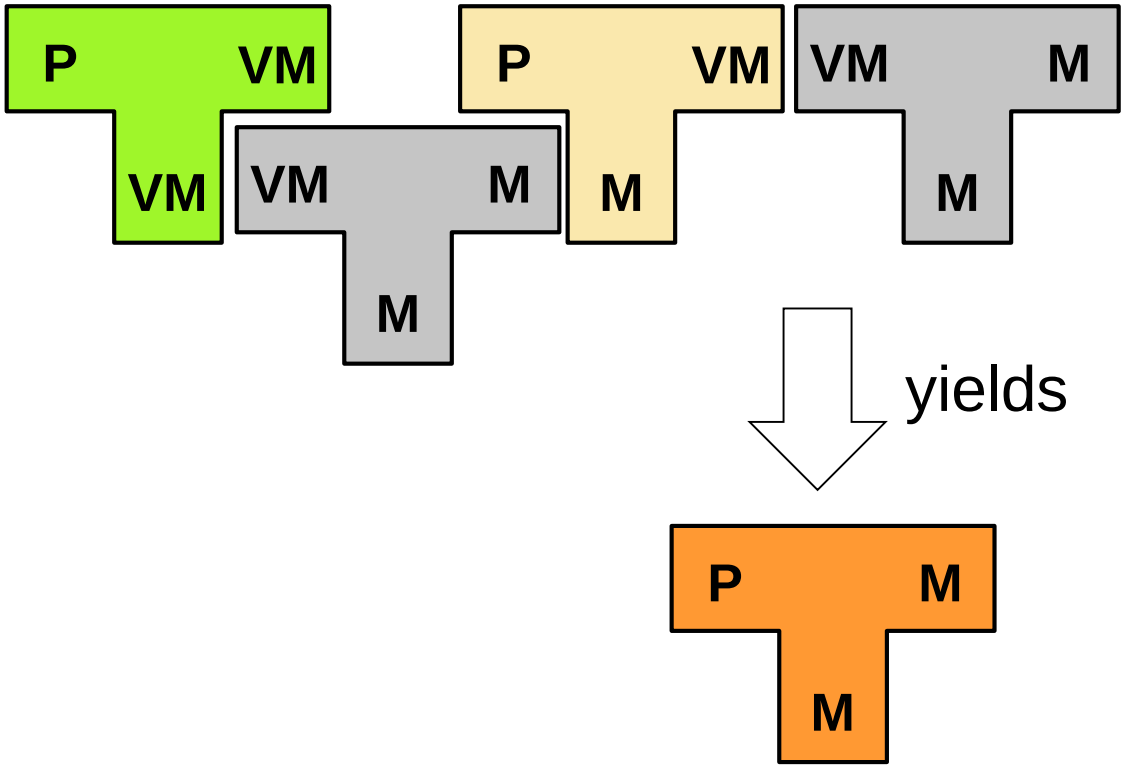


5. Bootstrap backend:



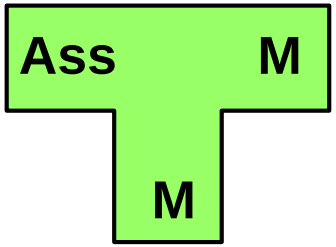
Bootstrapping

6. Bootstrap compiler:

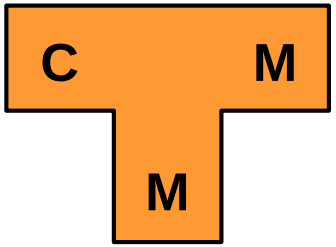


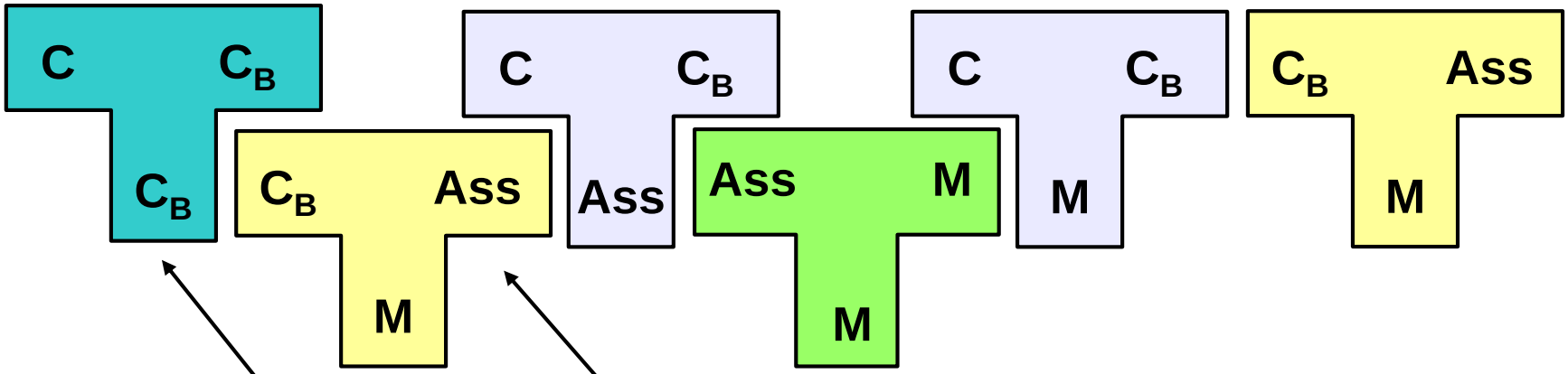
Bootstrapping

Given:



Goal:





Bootstrapping