High-level Language VMs

CS 562: Virtual Machines

Kyle Hale

Reading: S&N Ch. 5

Compiled programs tied to ISA

- …and also to an operating system
- to run on another ISA, we (at minimum) have to recompile
- If we *also* want to run on another OS, we have to *port* the program

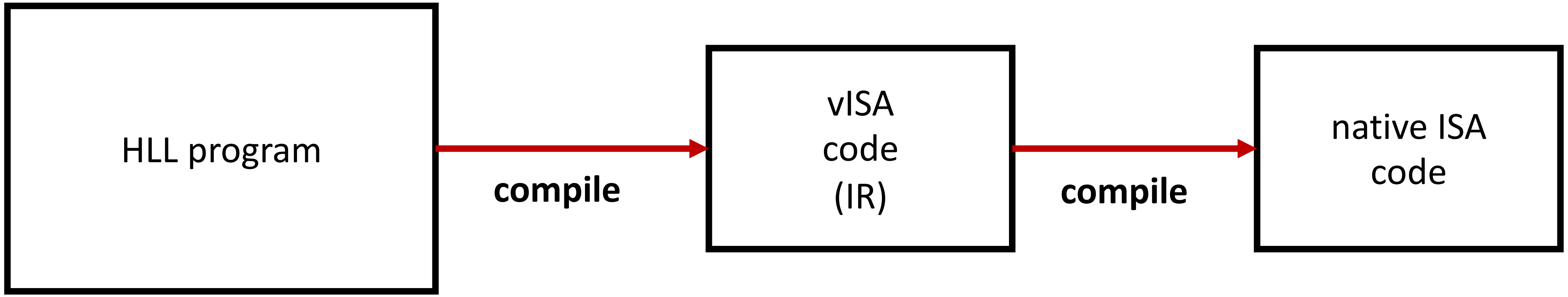
Why not just use a process VM?

- We *can*, but we'd have to do it many times! one for each guest<-> host pair (NxM problem)
- ABI mismatch is hard to deal with (still have to consider different Oses, even if same ISA)
- Performance is elusive

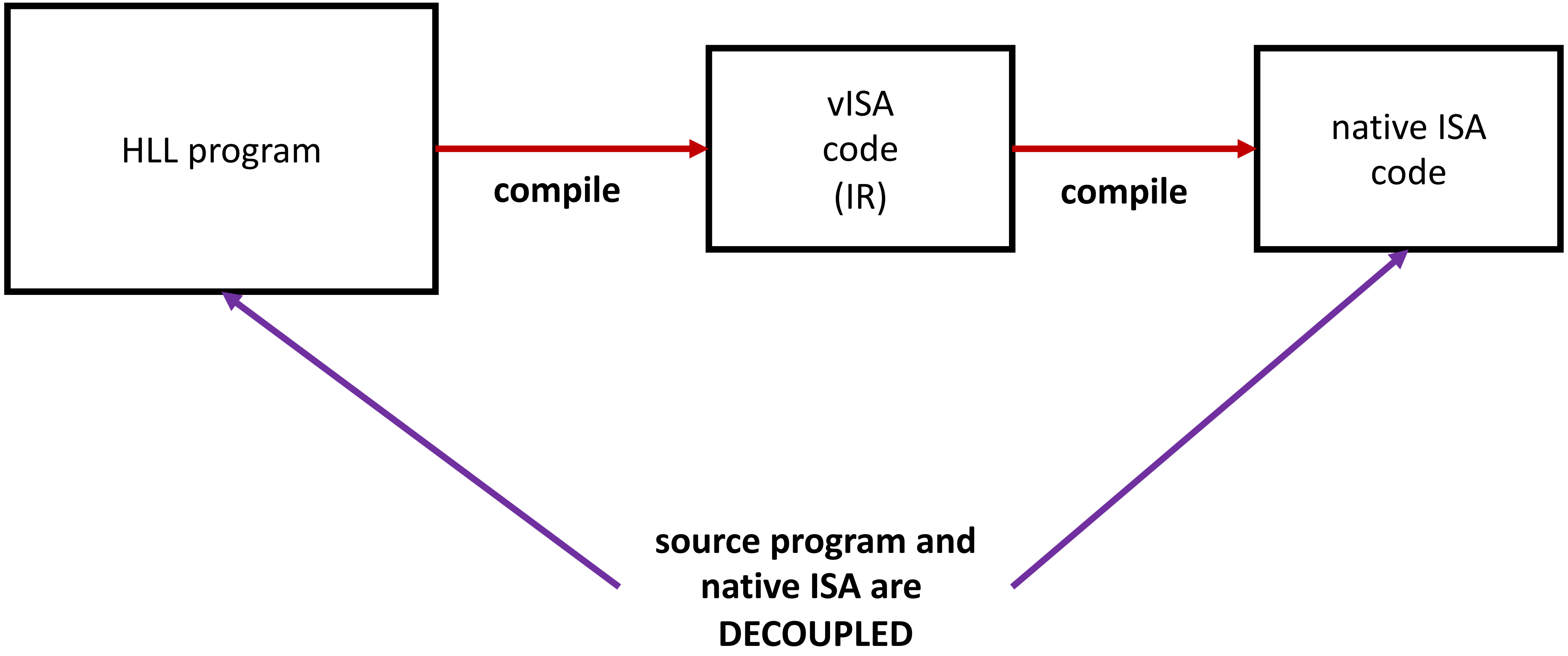
Design a **portable** ISA

- Goal: ISA should be easy to compile to *other* ISAs
  - minimize amount of machine state
  - instruction set should be simple
- Decouple the ISA from any *real* hardware (and associated quirks!)
- What about I/O?
  - Tame complexity of syscalls: these have quicks both in hardware and OS
  - **Instead**: I/O is handled by **system libraries**

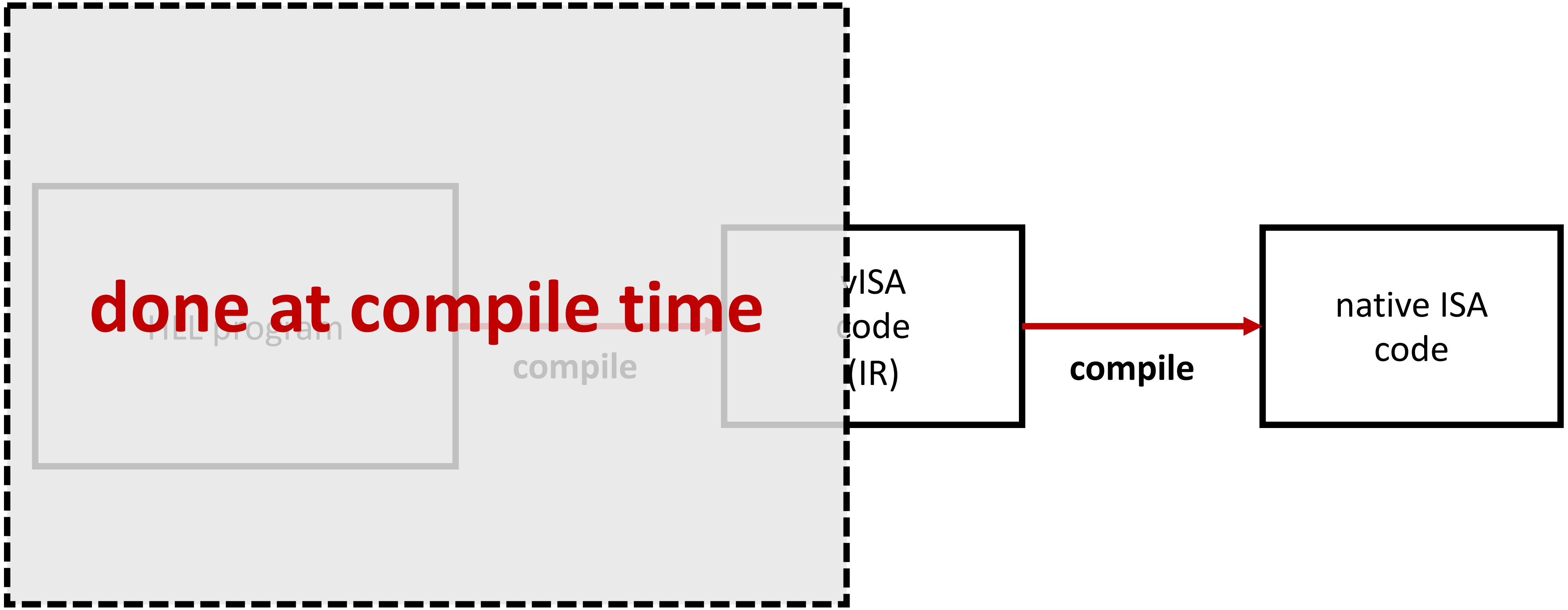
Basic flow for HLL VMs (two steps)



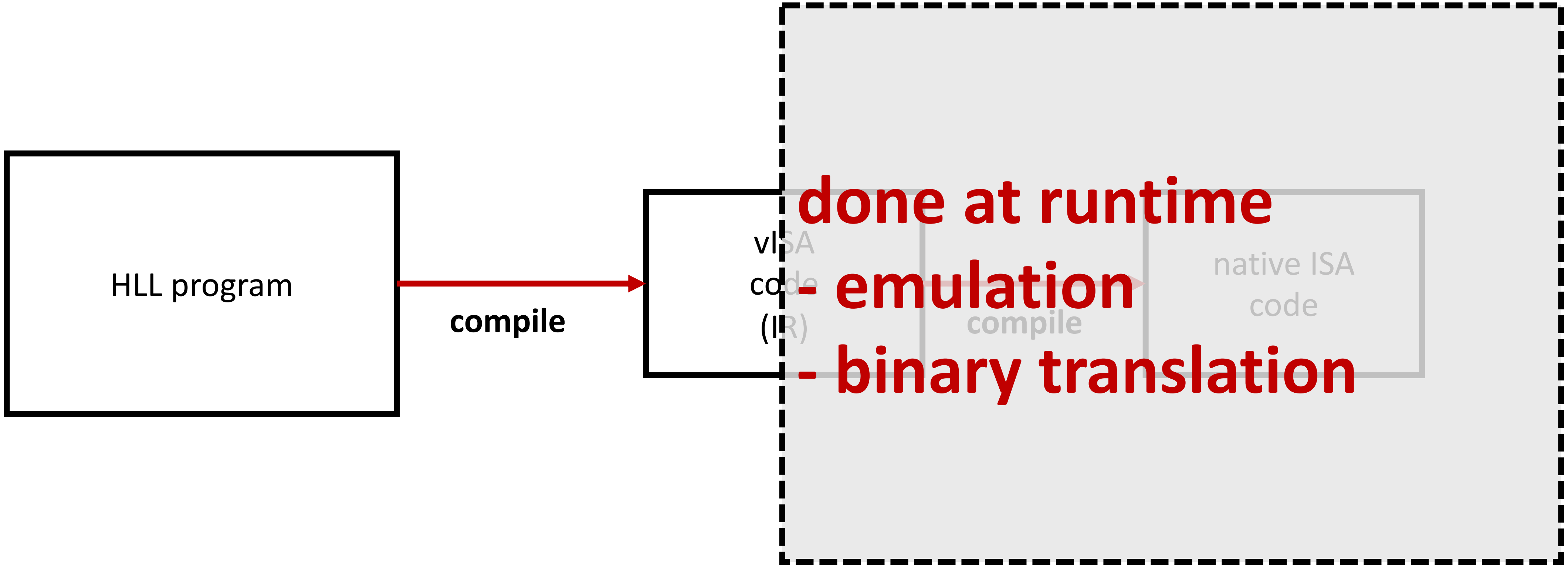
Basic flow for HLL VMs (two steps)



Basic flow for HLL VMs (two steps)



Basic flow for HLL VMs (two steps)



vISA is an *intermediate representation* (IR)

- The **big difference** from the typical compiler pipeline (front-end/backend split): backend is a runtime component!

Brief note on IR history

- First language to use an IR was *basic compiled PL* (BCPL) - ~1967
- The IR was called “object code” (or O-code)

Algol 60 (~1960)

- First *block-structured* language
- First with nested function defs with lexical scope
- First language with a formal definition (led to BNF grammar)
- I/O not part of lang.

```
BEGIN
COMMENT
/////////////////////////////////////////////////////////////
// Name: Peter M. Maurer
// Program: Sieve of Eratosthenes
// Due: Never
// Language: Algol 60
/////////////////////////////////////////////////////////////
;
    COMMENT define the sieve data structure ;
    INTEGER ARRAY candidates[0:1000];
    INTEGER i,j,k;
    COMMENT 1000 to protect against strict evaluation of AND ;
    FOR i := 0 STEP 1 UNTIL 1000 DO
    BEGIN
        COMMENT everything is potentially prime until proven otherwise ;
        candidates[i] := 1;
    END;
    COMMENT Neither 1 nor 0 is prime, so flag them off ;
    candidates[0] := 0;
    candidates[1] := 0;
    COMMENT start the sieve with the integer 0 ;
    i := 0;
    FOR i := i WHILE i<1000 DO
    BEGIN
        COMMENT advance to the next un-crossed out number. ;
        COMMENT this number must be a prime ;
        FOR i := i WHILE i<1000 AND candidates[i] = 0 DO
        BEGIN
            i := i+1;
        END;
        COMMENT insure against running off the end of the data structure ;
        IF i<1000 THEN
        BEGIN
            COMMENT cross out all multiples of the prime, starting with 2*p.;
            j := 2;
            k := i*i;
```

BCPL (~1967)

- Originally developed for writing compilers
- inherited from CPL, but much simpler
- One data type! (bit pattern)
- Can use pointers

```
GET "libhdr"

MANIFEST $(
modulus = #x10001  // 2**16 + 1

N       = upb + 1    // N is a power of 2
MSB     = N>>1
LSB     = 1
$)

STATIC   $( v=0; w=0  $)

LET start() = VALOF
$( v := getvec(upb)
   w := getvec(upb)

   FOR i = 0 TO upb DO v!i := i
   pr(v, 15)
// prints  -- Original data
//     0     1     2     3     4     5     6     7
//     8     9    10    11    12    13    14    15

   w!0 := 1
   FOR i = 1 TO upb DO w!i := mul(w!(i-1), omega)  // roots of unity
   FOR i = 1 TO upb IF w!i=1 DO writef("omega****%n = 1*n", i)
   UNLESS mul(w!upb, omega)=1 DO writef("Bad omega*n")
   fftn(v)
   pr(v, 15)
// prints   -- Transformed data
// 65017 26645 38448 37467 30114 19936 15550 42679
// 39624 42461 43051 65322 18552 37123 60445 26804

   w!0 := 1
   FOR i = 1 TO upb DO w!i := ovr(w!(i-1), omega) // inverse roots of unity
   FOR i = 1 TO upb IF w!i=1 DO writef("omega****-%n = 1*n", i)
   UNLESS ovr(w!upb, omega)=1 DO writef("Bad omega*n")
   fftn(v)
   FOR i = 0 TO upb DO v!i := ovr(v!i, N)
   pr(v, 15)
```

B - ~1969 - Ken Thompson and Dennis Ritchie)

```
main( ) {
     auto a, b, c, sum;
     a = 1;
     b = 2;
     c = 3;
     sum = a+b+c;
     putnumb(sum);
 }
```

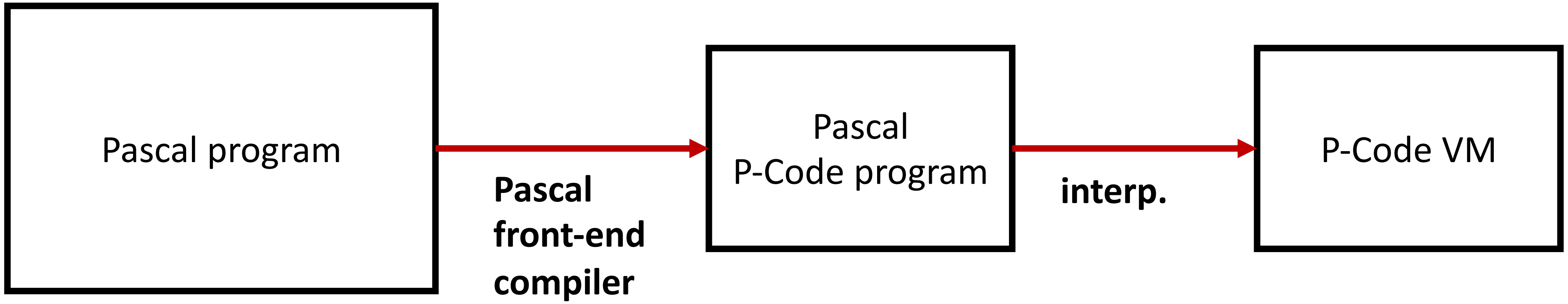
no types!

Our HLL VM assumptions for now

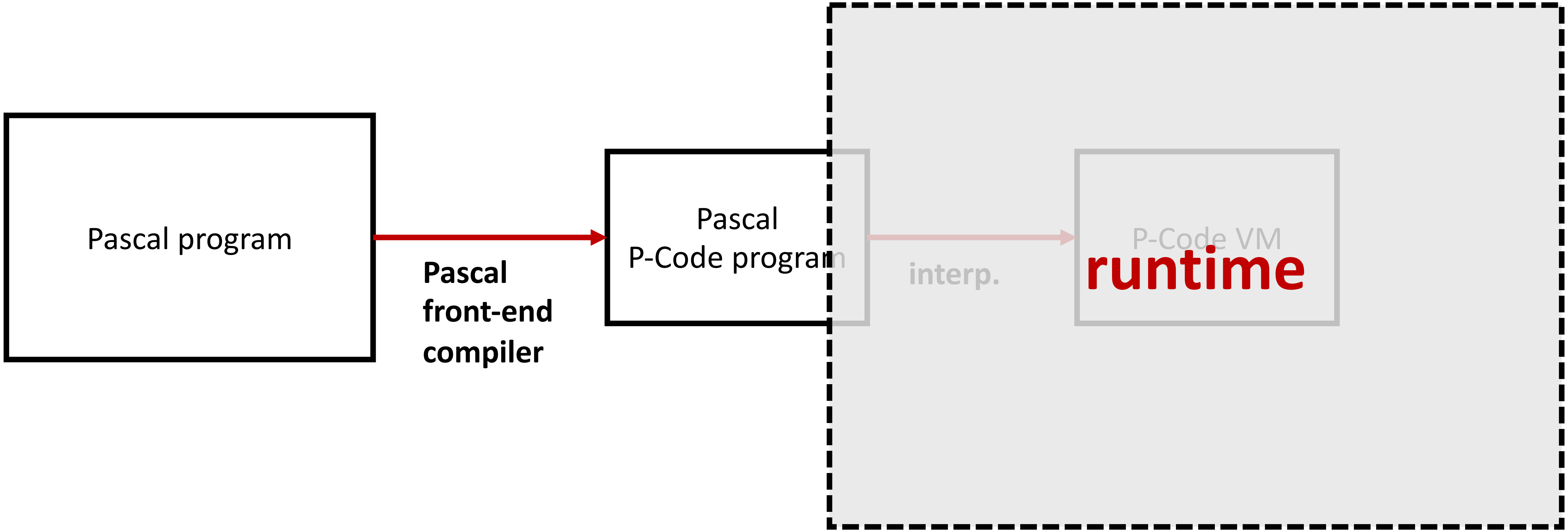
- Run as process VM (user level)
- Instructions execute on a virtual processor (which implements a **vISA**)
- Protection ignored for now

Example: Pascal’s P-Code VM

- Pascal developed in late 60s
- VM implementation came in 1975, making Pascal more popular
- Pascal **heavily** influenced the design of Java
  - Unlike Java, no object-orientation, no networked applications, no garbage collection, etc.
  - similar portability goals though!



Pascal program
Pascal
front-end
compiler
Pascal
P-Code program
interp.
P-Code VM

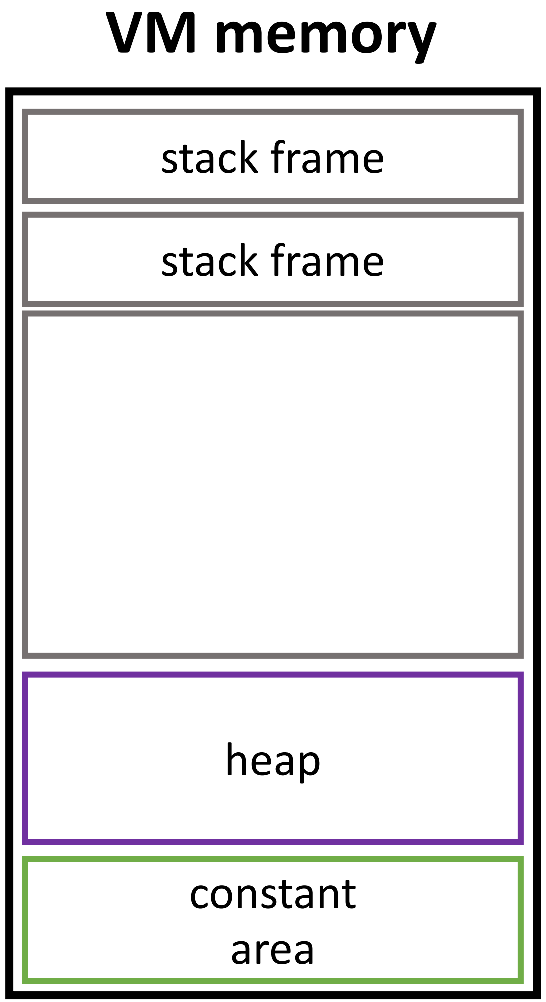


Pascal program
Pascal
front-end
compiler
Pascal
P-Code program
interp.
P-Code VM
runtime

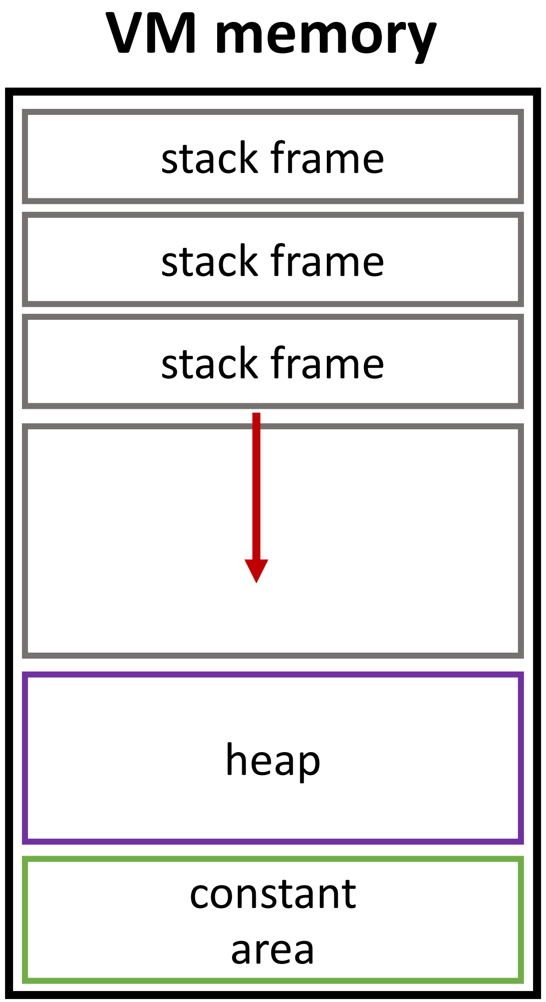
Pascal P-Code VM has two major parts…

- Instruction emulator (interpreter)
- Standard library routines
  - These implement I/O using host OS routines
  - **Implemented as native code!**

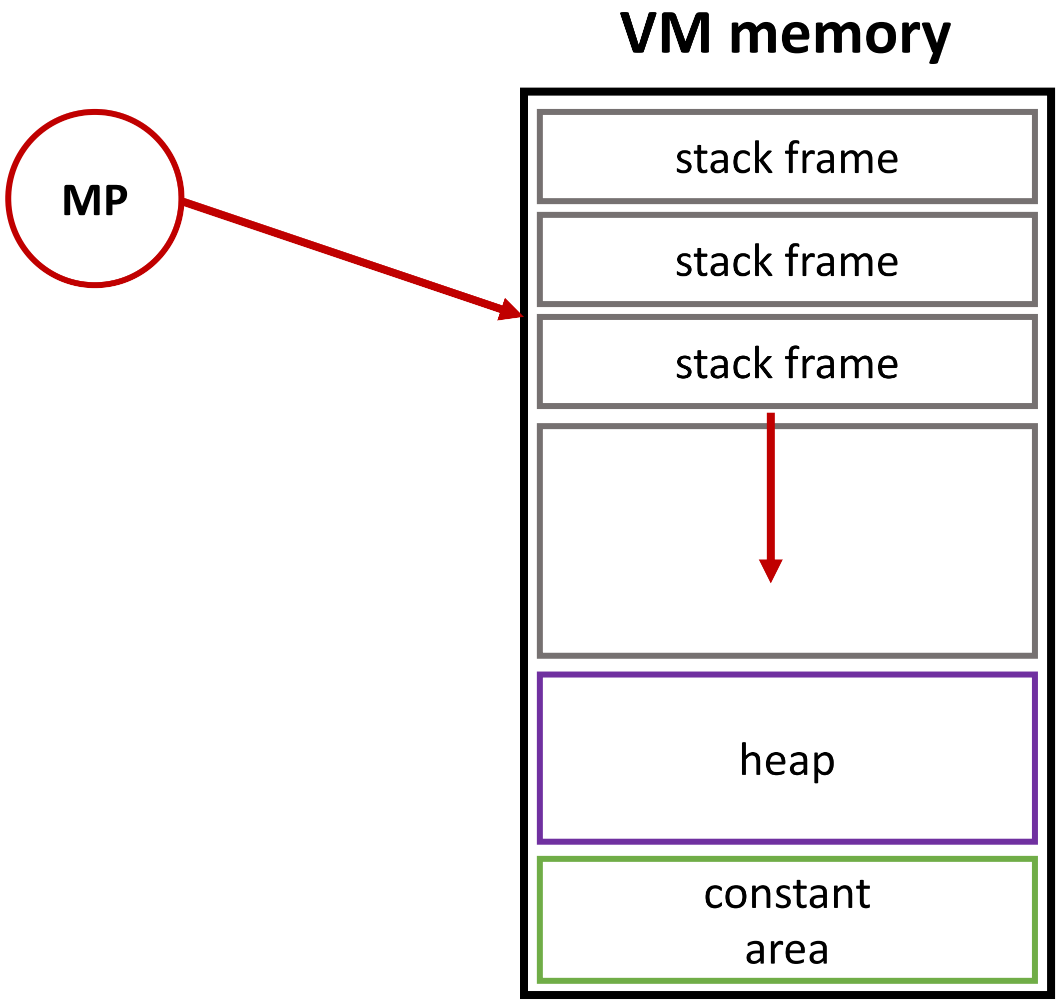
P-Code VM memory layout is similar…but different



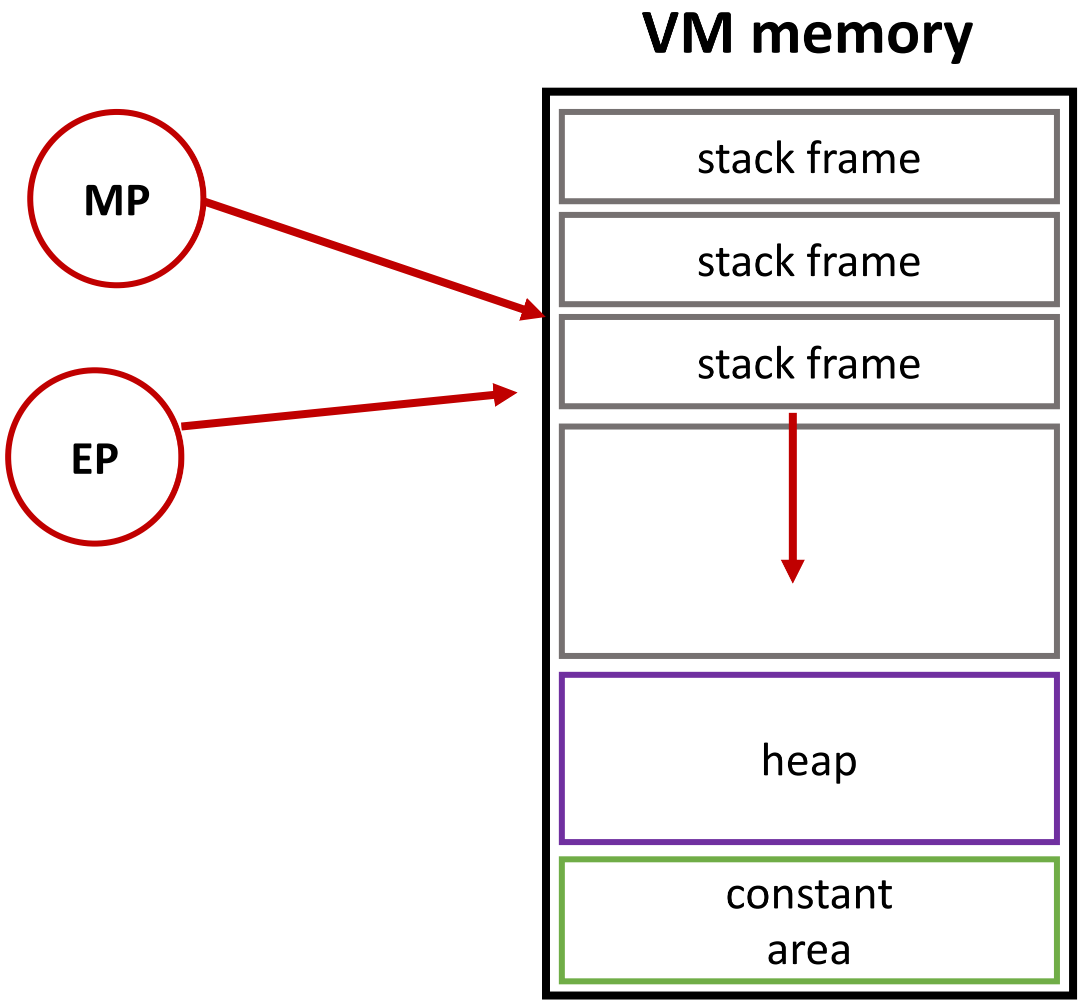
stack grows down



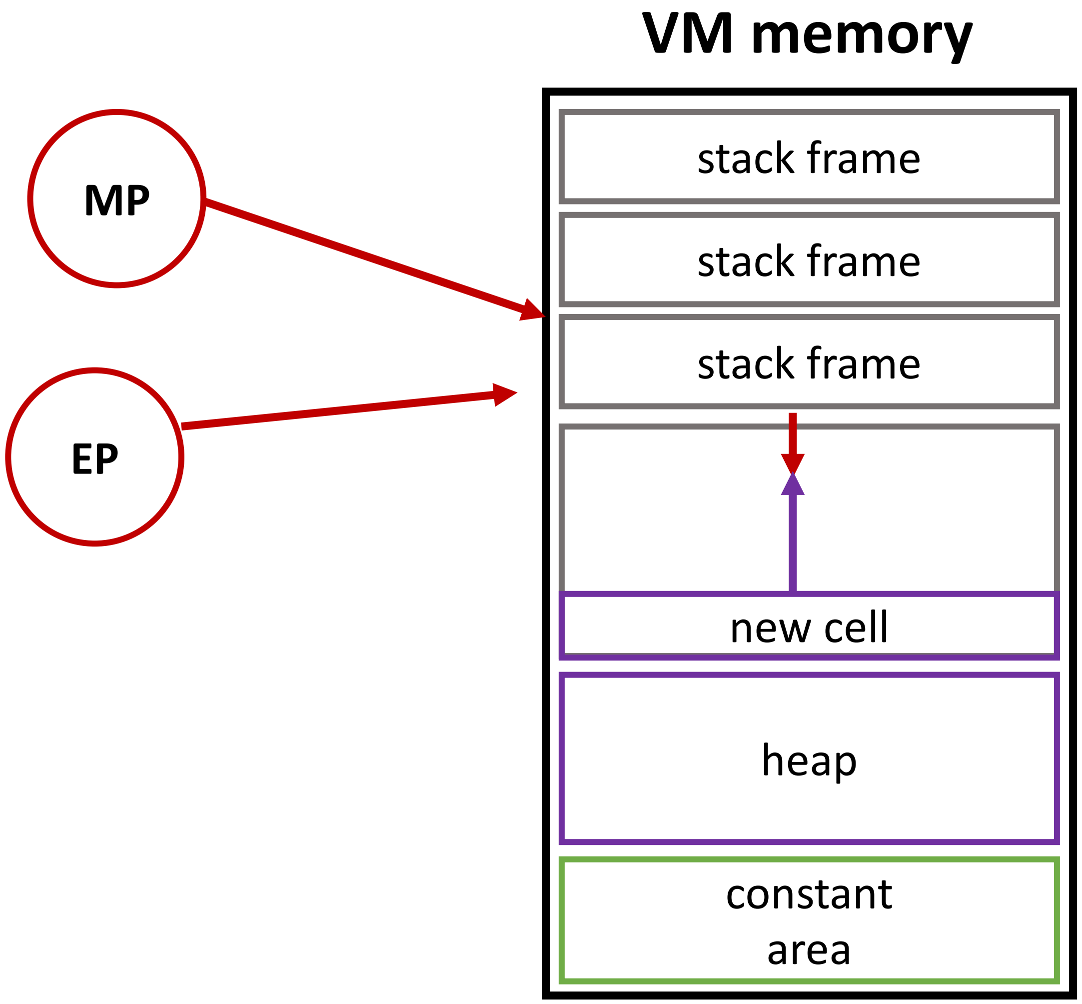
Mark Pointer is base of current frame



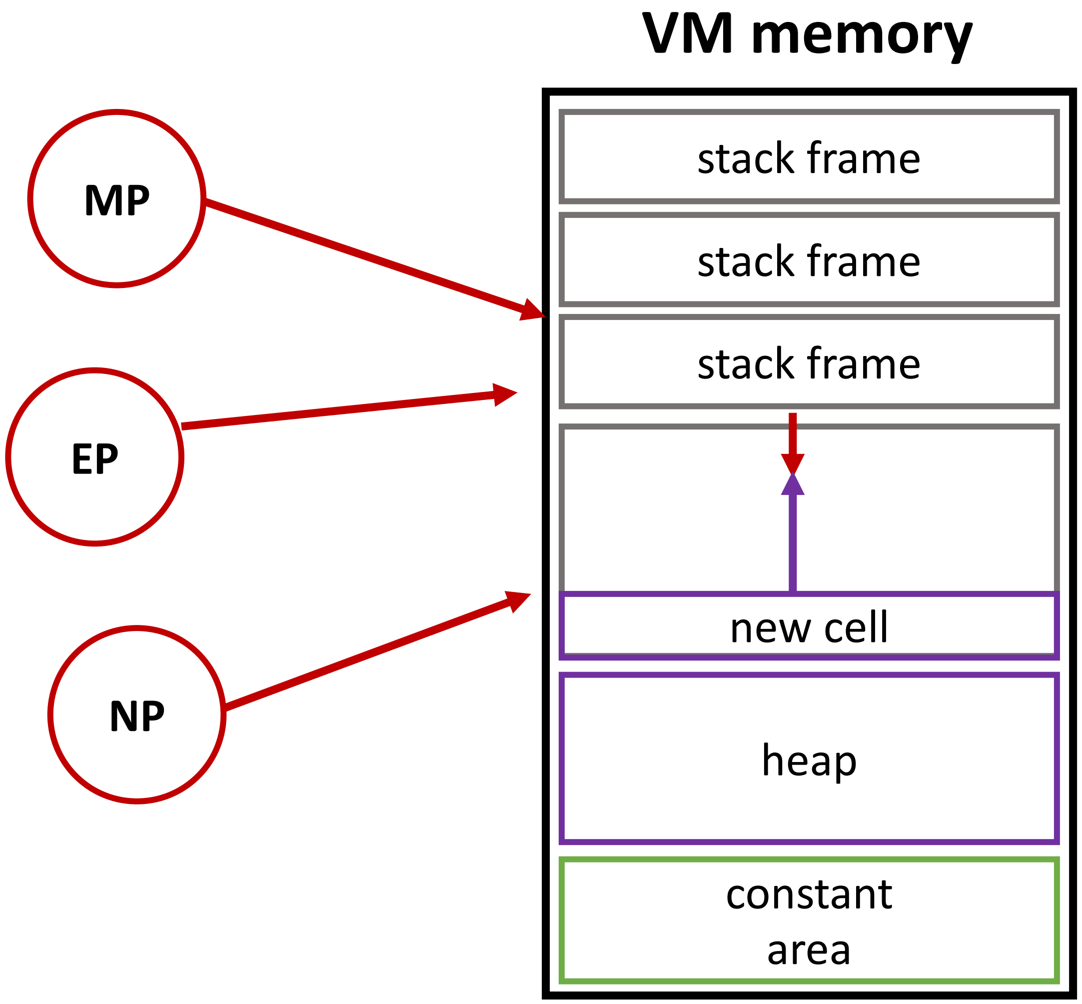
Extreme Pointer is end of current frame



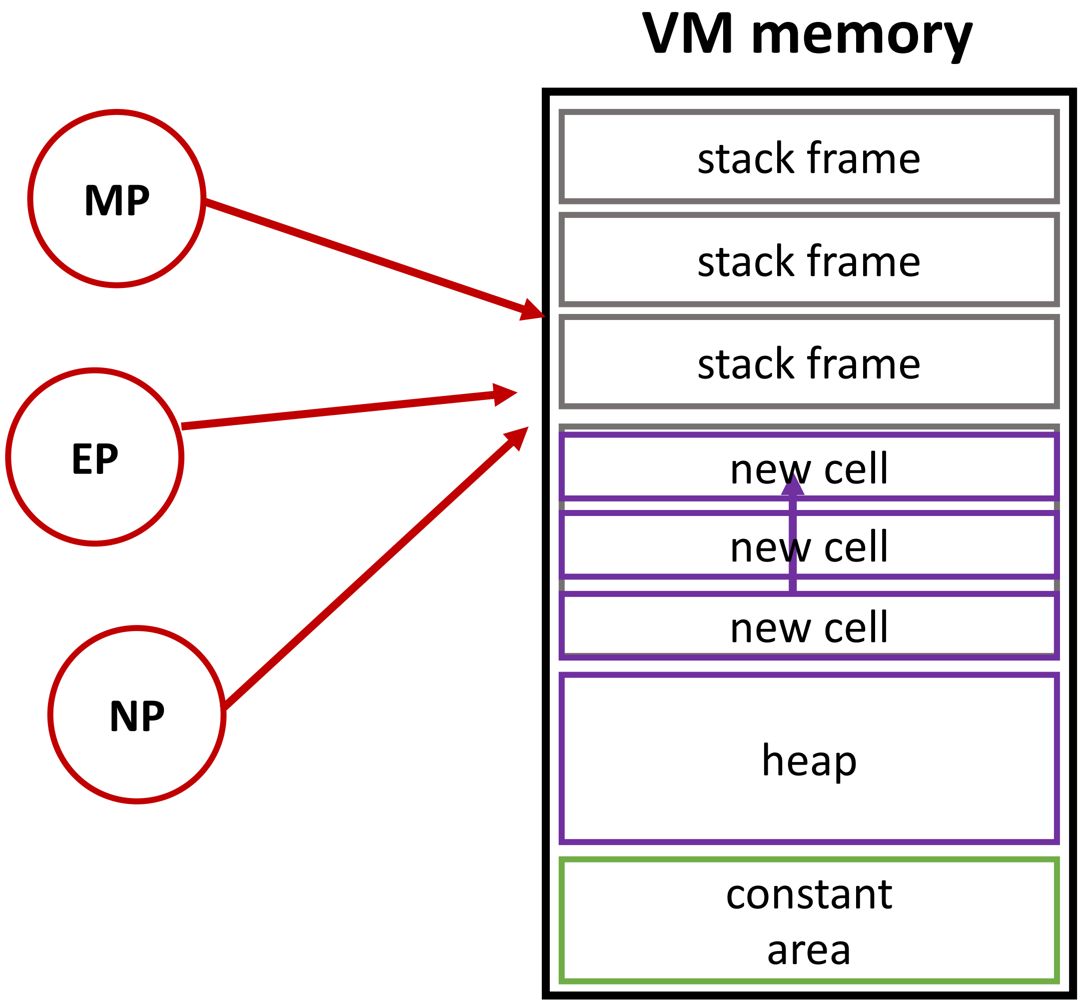
Heap grows up



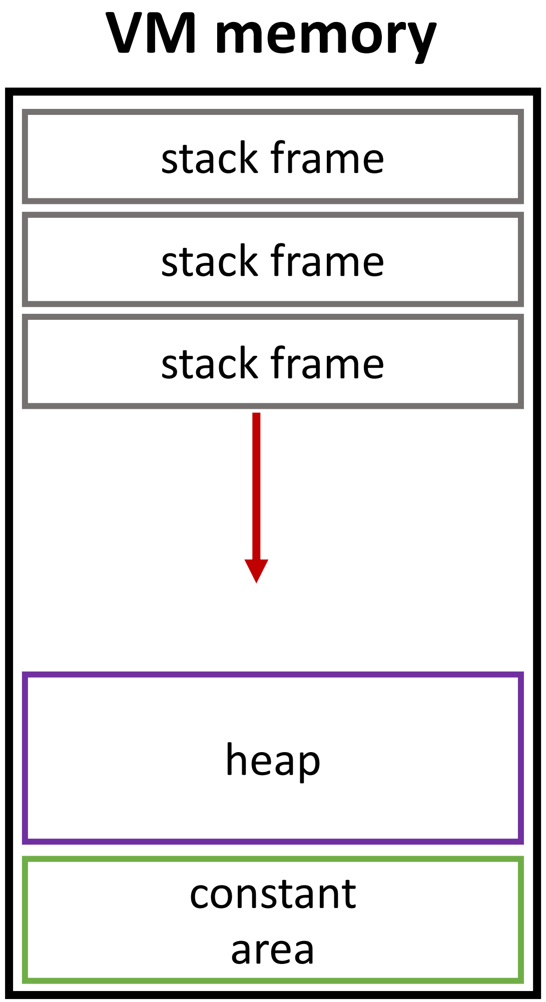
New Pointer indicates next free heap mem.



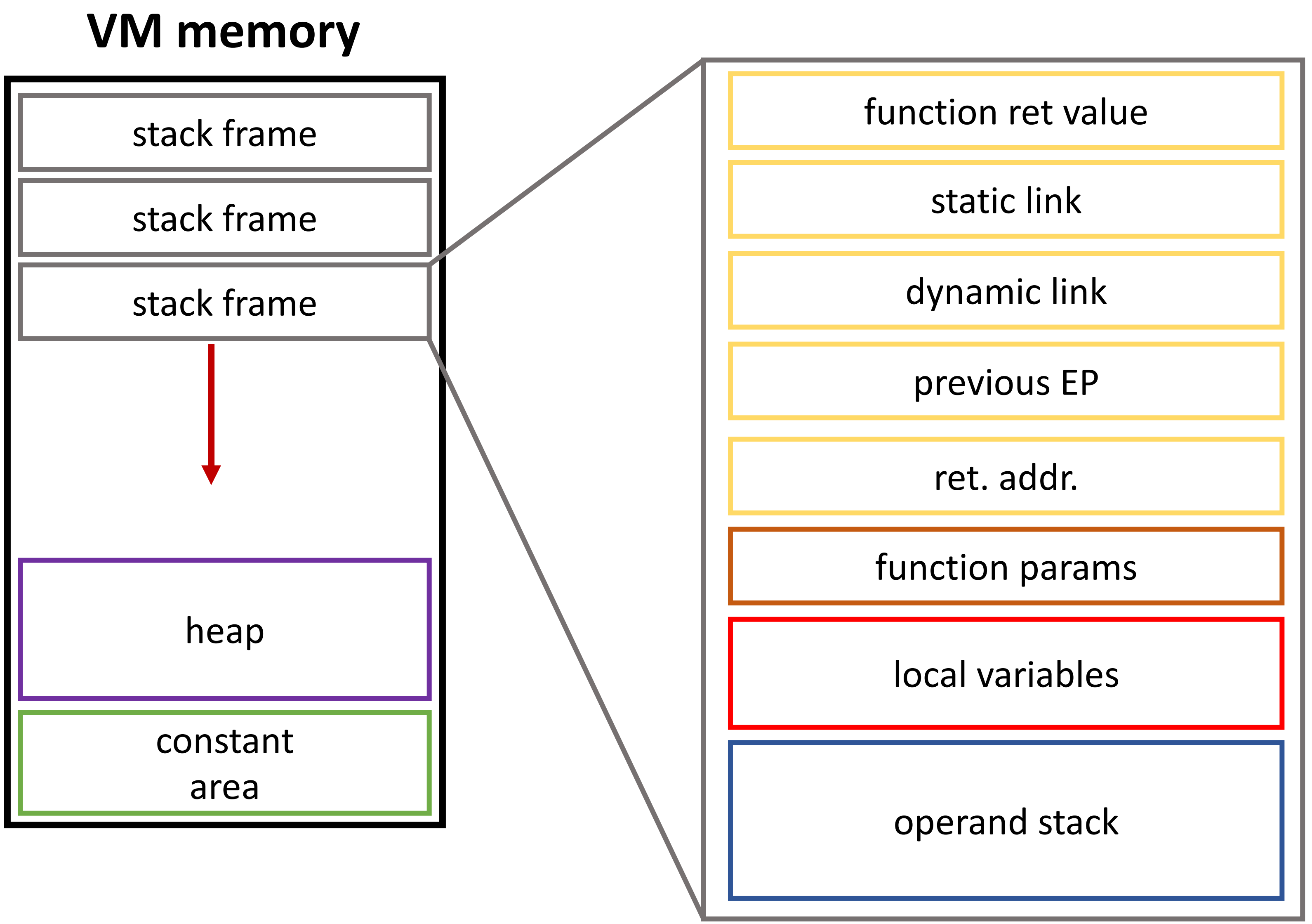
It's up to programmers to free memory! this will cause a runtime error…



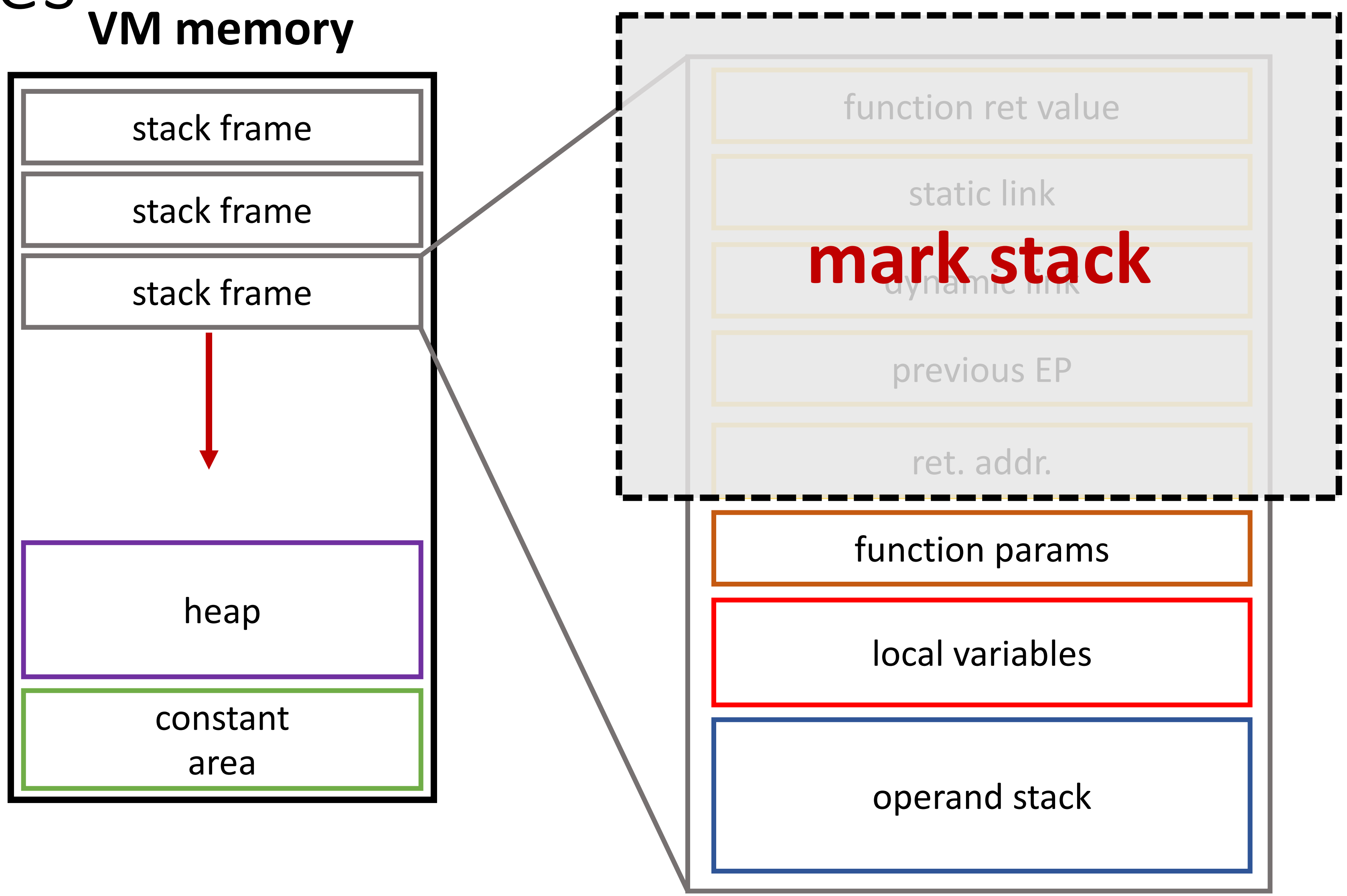
new function → new stack frame



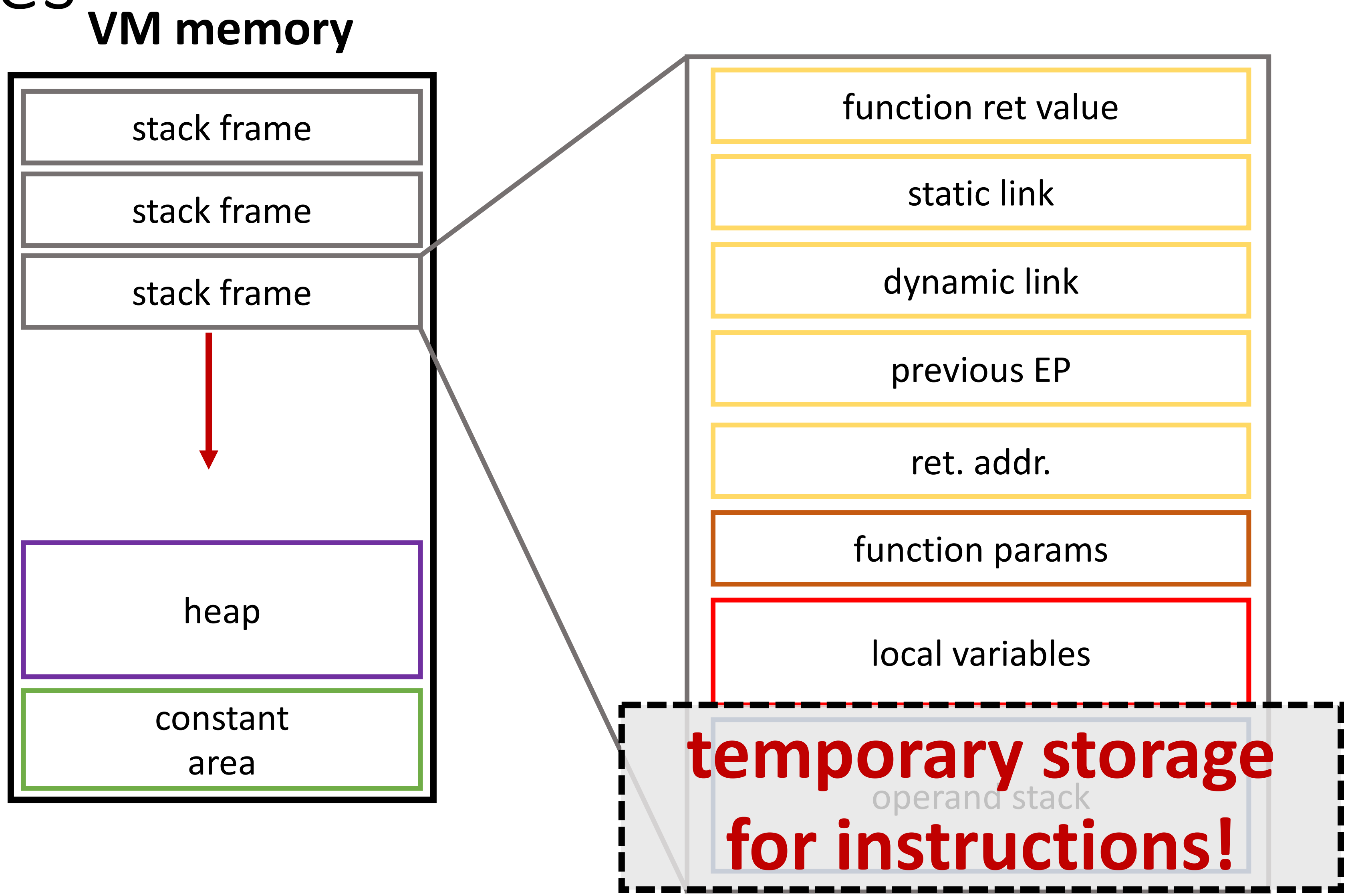
new function → new stack frame



mark stack handles ret vals, links between frames



mark stack handles ret vals, links between frames

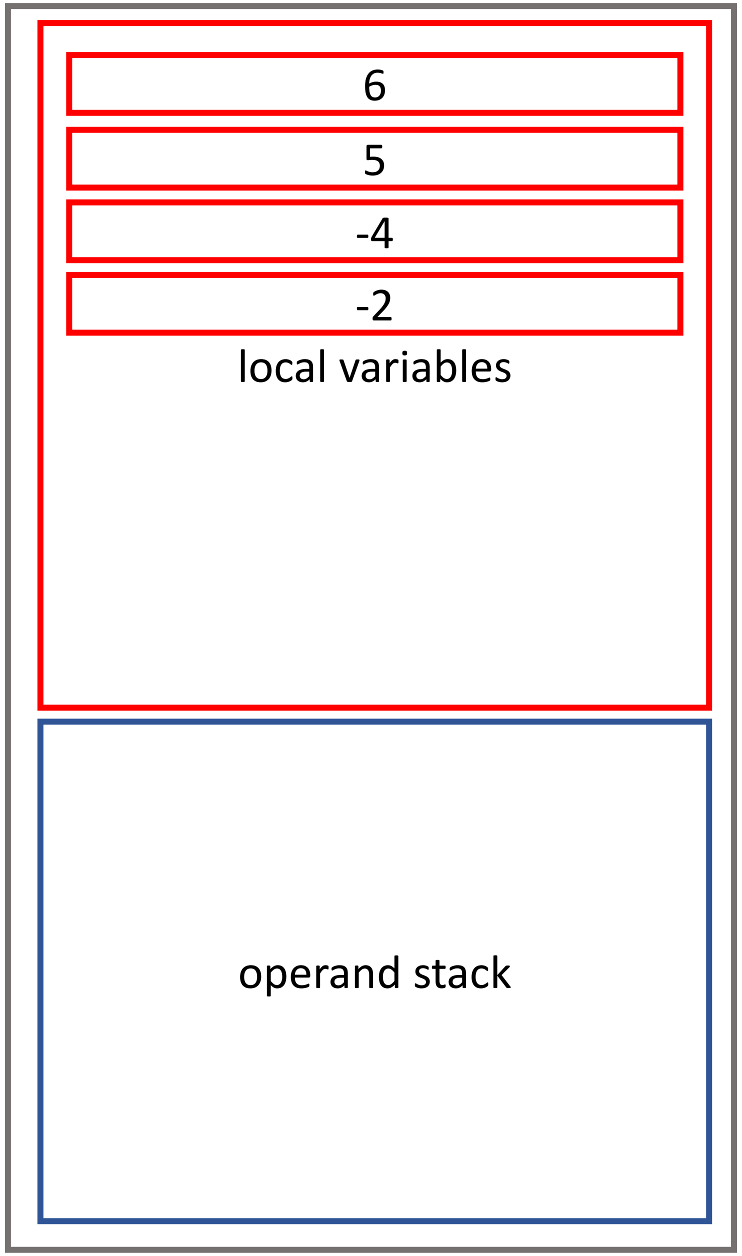


Example P-Code

```
lodi  0  3 // load local var. from cur. frame (nset 0 depth),
           // offset 3 from top of mark stack.
ldci  1    // push constant 1 onto op stack
addi       // add top two items on op stack (implicit pop), push result
stri  0  3 // put result back into local variable location
```

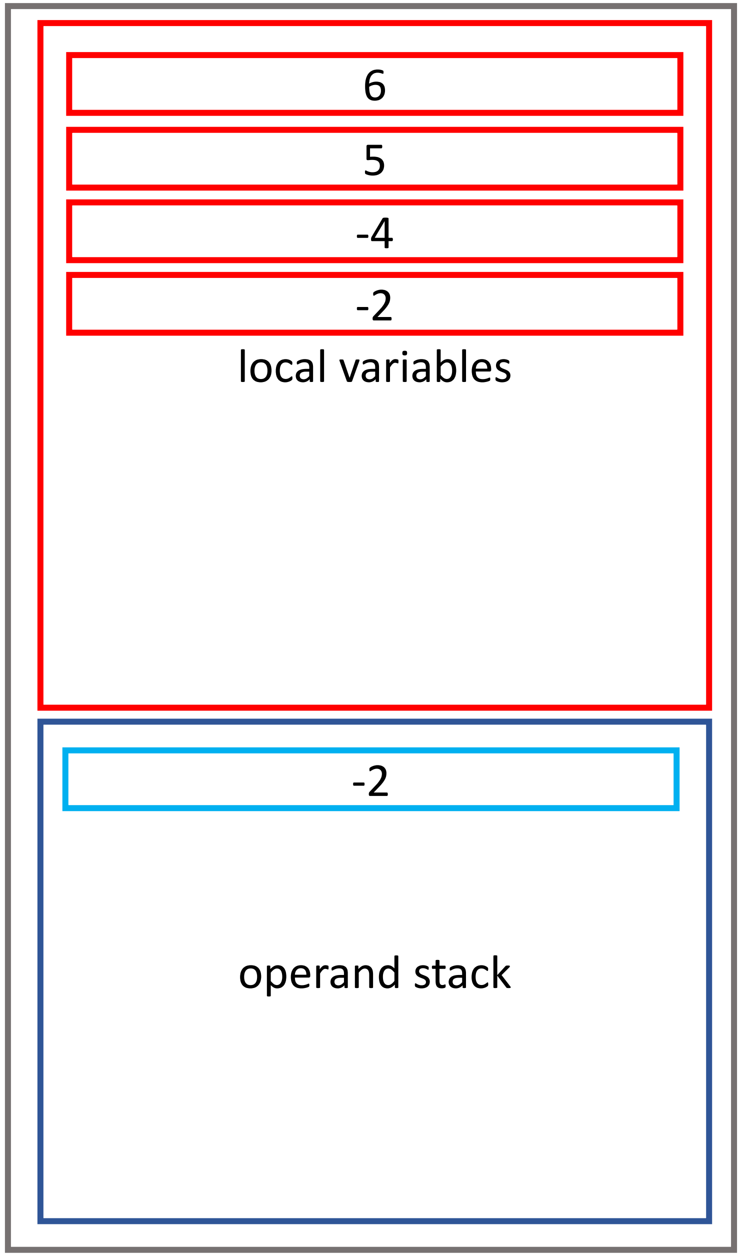
Example P-Code

```
lodi  0  3
ldci  1
addi
stri  0  3
```



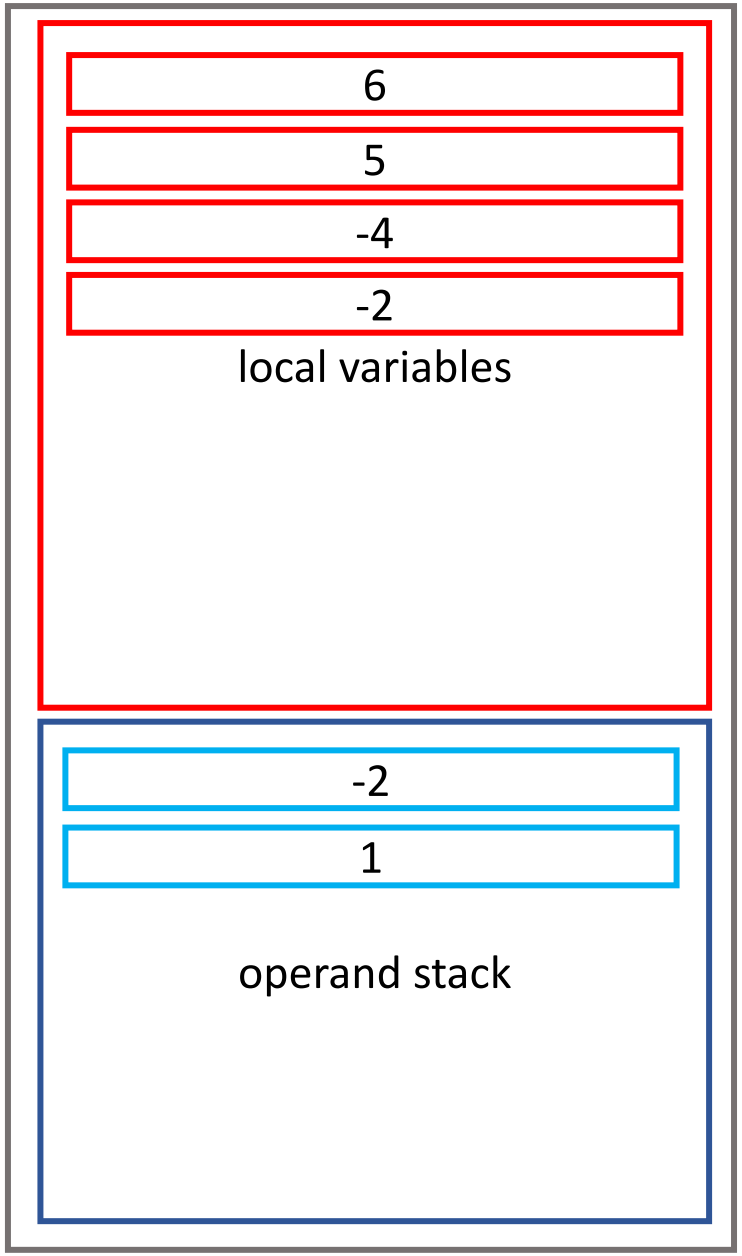
Example P-Code

```
lodi  0  3
ldci  1
addi
stri  0  3
```



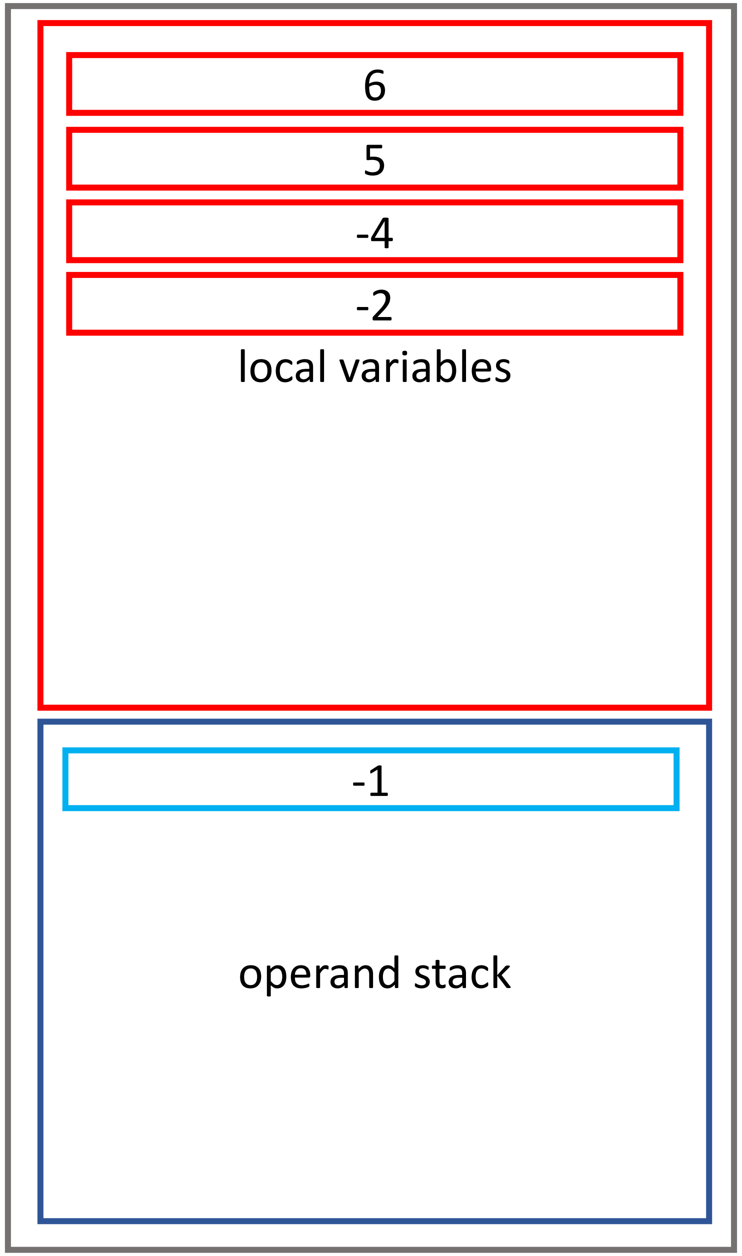
Example P-Code

```
lodi  0  3
ldci  1
addi
stri  0  3
```



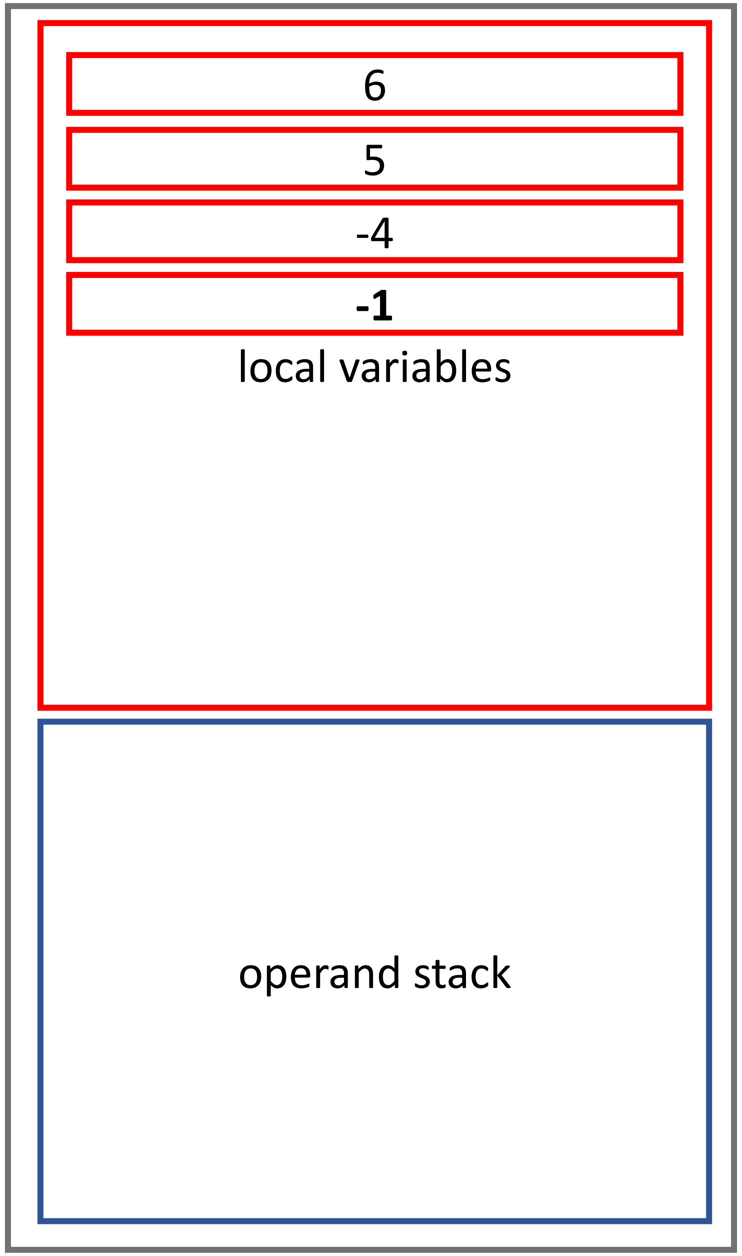
Example P-Code

```
lodi  0  3
ldci  1
addi
stri  0  3
```



Example P-Code

```
lodi  0  3
ldci  1
addi
stri  0  3
```



Important parts of Pascal P-Code

- Stack machine simplifies writing host VM
  - also creates smaller binaries
- *cells* can be sized based on implementation
  - *good for ISAs with different word sizes*
- no memory addresses! programs cannot use them
- Interface to OS is via stdlibs
  - to be generic, I/O libs must be designed for "weakest" host OS interface → lowest common denominator problem
  - Tradeoff: platform independence vs. power of I/O and system interface!

Modern HLL VMs have to handle…

- **Security and protection**: run programs from network/internet (untrusted sources)
- **Robustness**: support for PL abstractions (e.g., objects), strong type checking, garbage collection (automatic mem. mgmt.)
- **Networking**: have to use network efficiently due to bandwidth constraints → on-demand loading and linking, denser instr. encodings
- **Performance!**