SAPIENZA
UNIVERSITÀ DI ROMA

Implementazione di un dialetto di LISP

Candidato
Scandone Alessandro
1700455

Relatore
Piperno Adolfo

Cosa è LISP

- *“Recursive Functions of Symbolic Expressions and Their Computation by Machine”*, McCarthy, 1960
- *“s-expression”* come sintassi
- numerosi dialetti

```
(defun fibonacci (n)
  (cond
    ((eq? 0 n) 0)
    ((eq? 1 n) 1)
    (otherwise (+
                (fibonacci (- n 1))
                (fibonacci (- n 2))))))
```

Dialetto realizzato

- funzionale
  - ricorsione come unico costrutto di looping
  - bindings immutabili
  - strutture dati persistenti
- dinamicamente tipato ed interpretato
- concorrenza ad actor model
- implementato nel linguaggio scala
  - garbage collected
  - funzionale ed ad oggetti

```
(defun fibonacci (n)
  (cond
    ((eq? 0 n) 0)
    ((eq? 1 n) 1)
    (otherwise (+
                (fibonacci (- n 1))
                (fibonacci (- n 2))))))
```

Architettura interprete

sorgente

parsing ↓

AST

compilazione ↓

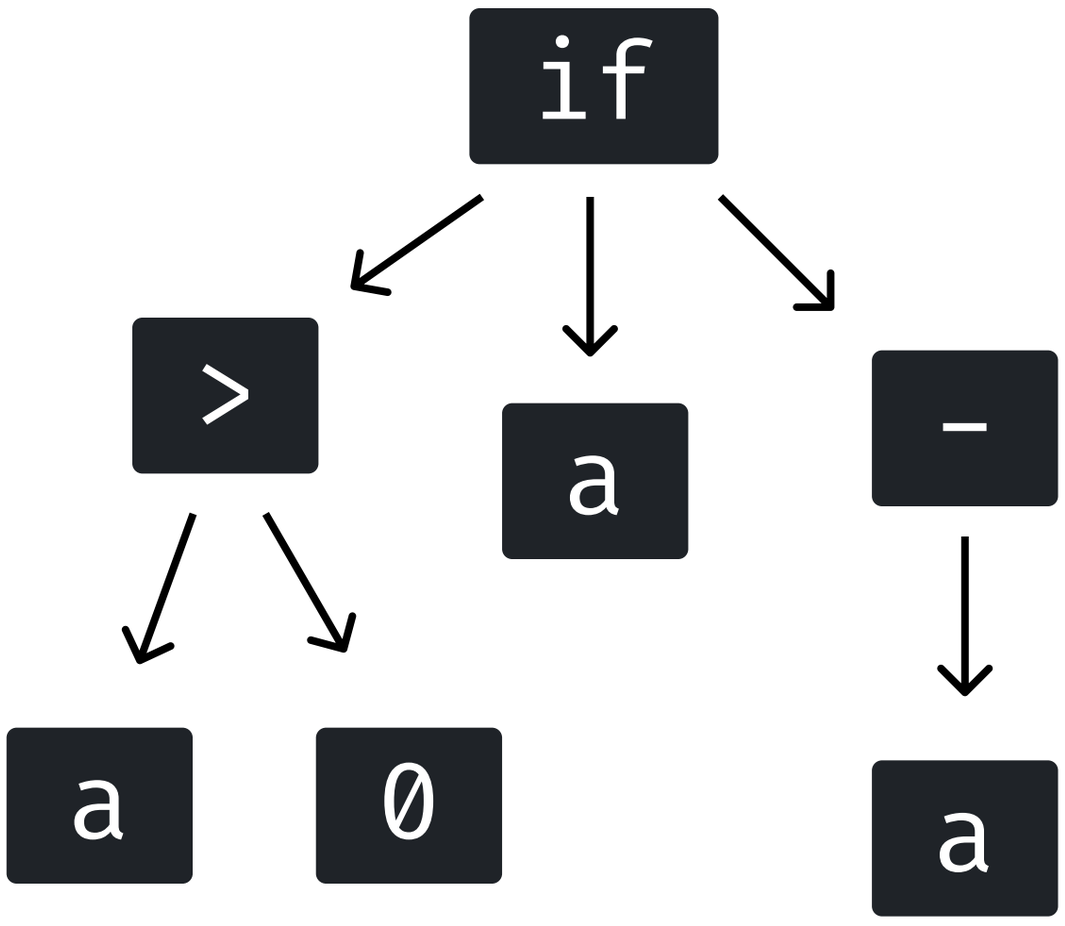
opcodes

virtual machine ↓

output

Architettura interprete

AST



s-expression

```
(if (> a 0)
    a
    (- a))
```

Architettura interprete

sorgente

parsing ↓

AST

compilazione ↓

opcodes

virtual machine ↓

output

Architettura interprete

sorgente

read ↓

sexpr

compilazione ↓

opcodes

virtual machine ↓

output

Architettura interprete

sorgente

read ↓

sexpr

compilazione ↓

opcodes

virtual machine ↓

output

Architettura interprete

sorgente

read ↓

sexpr

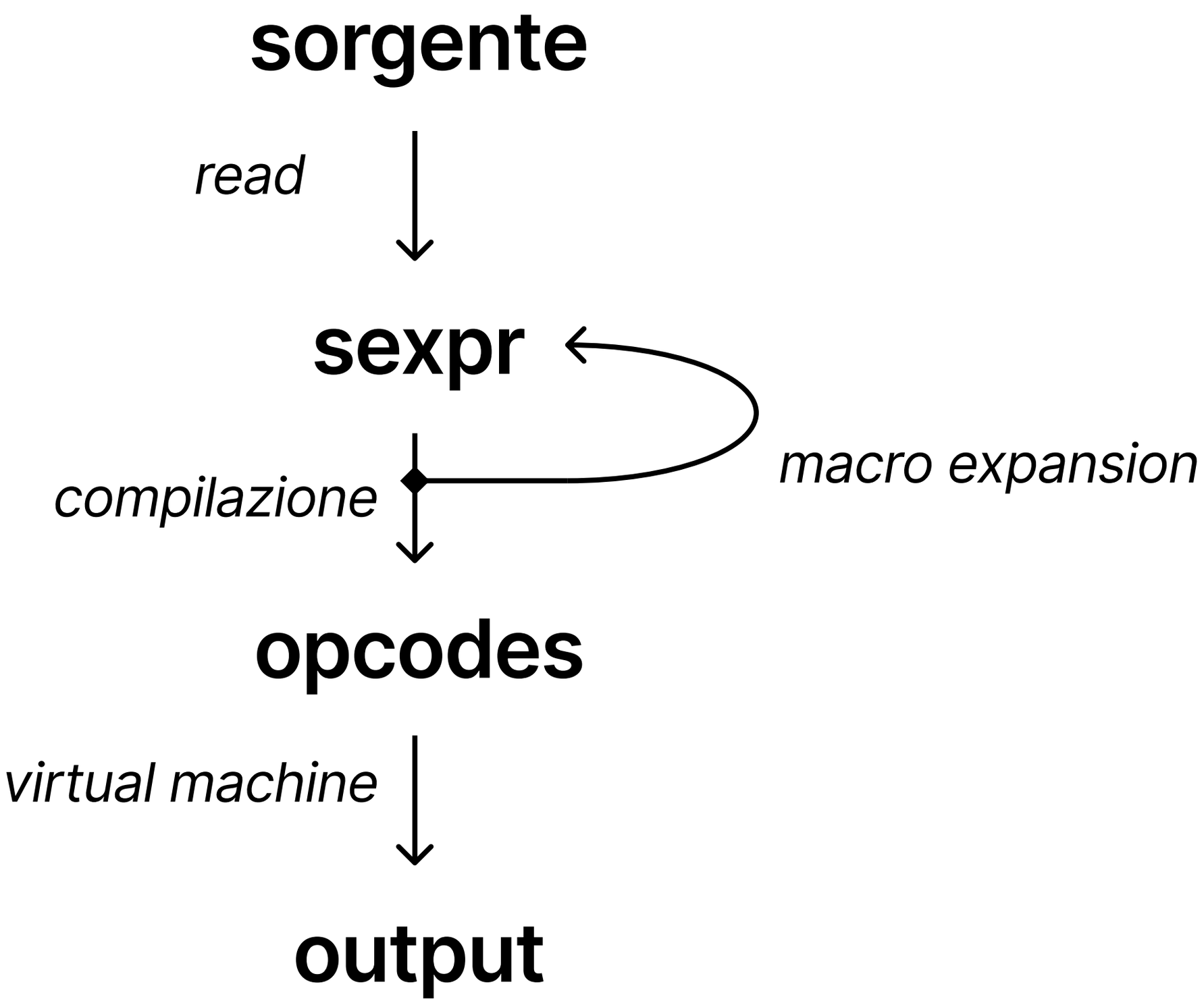
compilazione ↓

opcodes

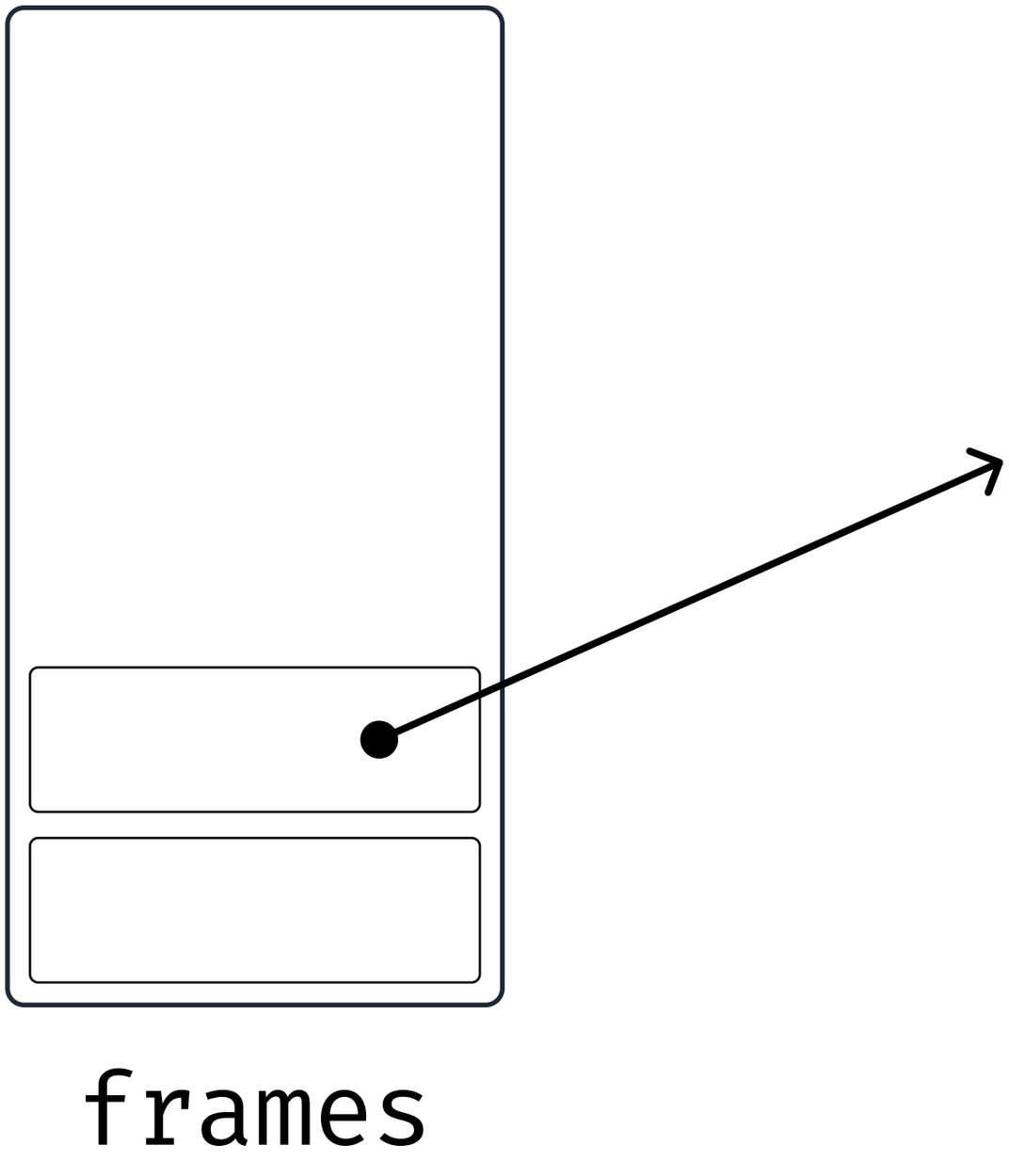
virtual machine ↓

output

Architettura interprete



Architettura interprete / VM (frames)



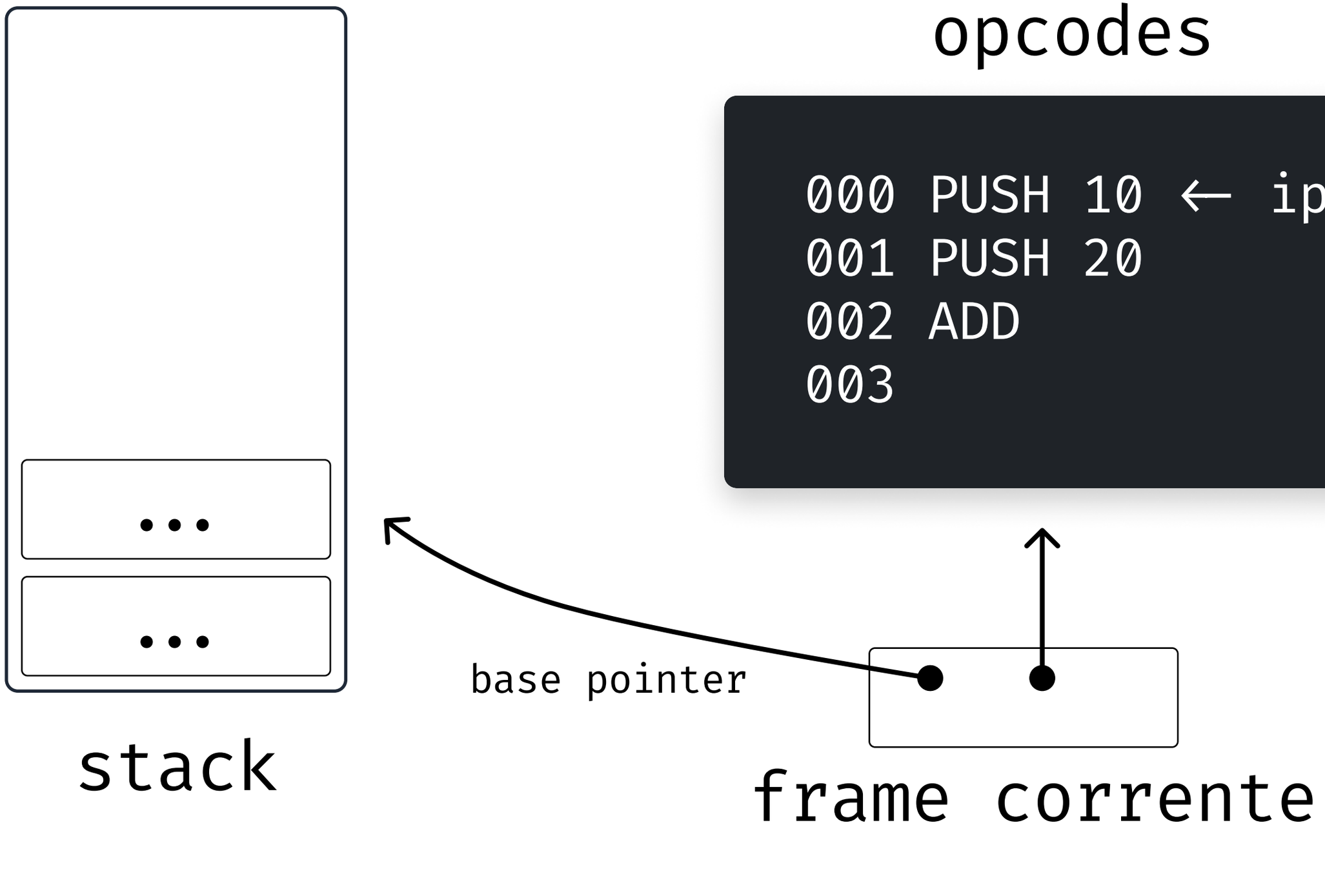
opcodes

```
000 PUSH 10 ← ip
001 PUSH 20
002 ADD
003
```

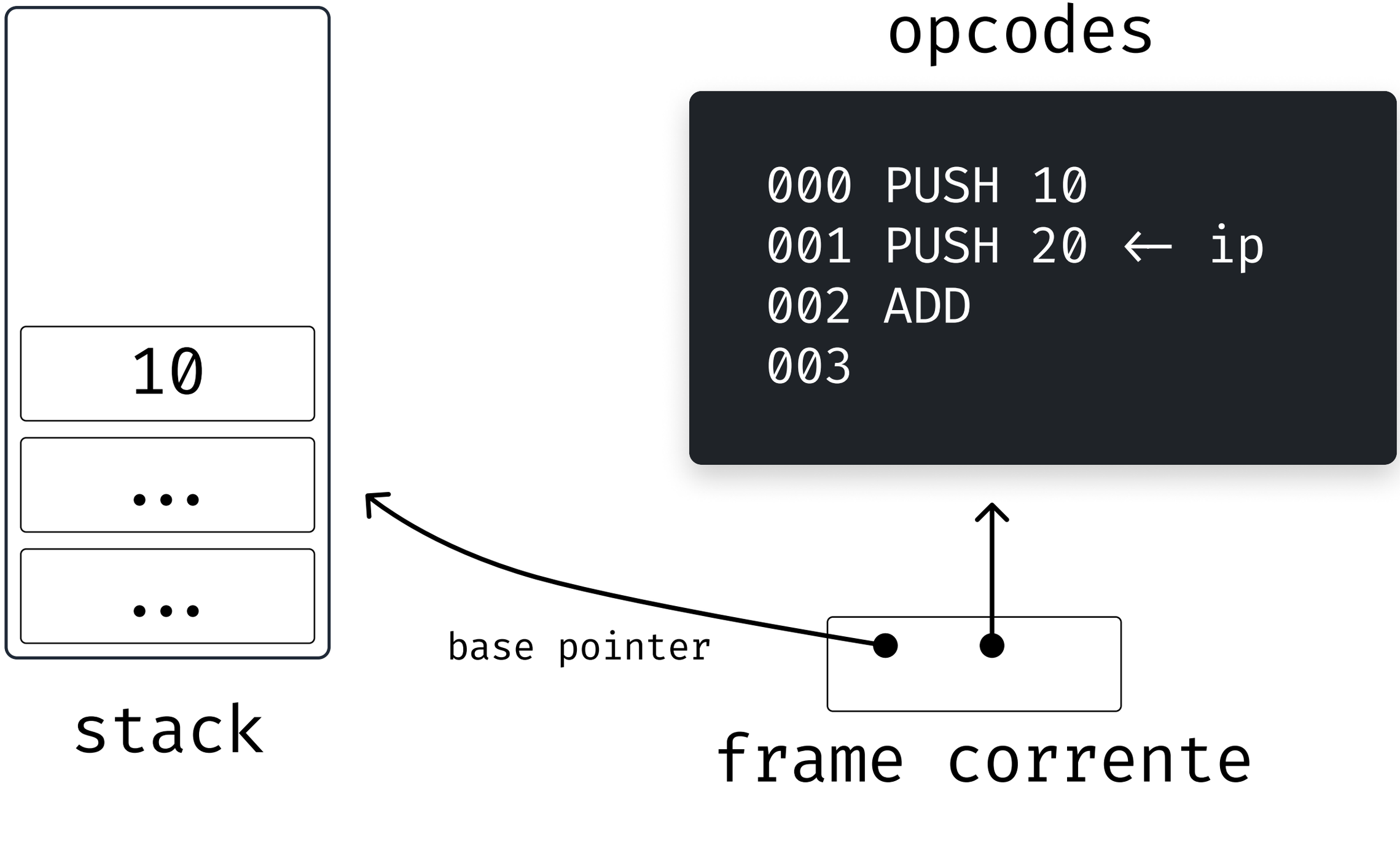
Architettura interprete / VM (stack)

opcodes

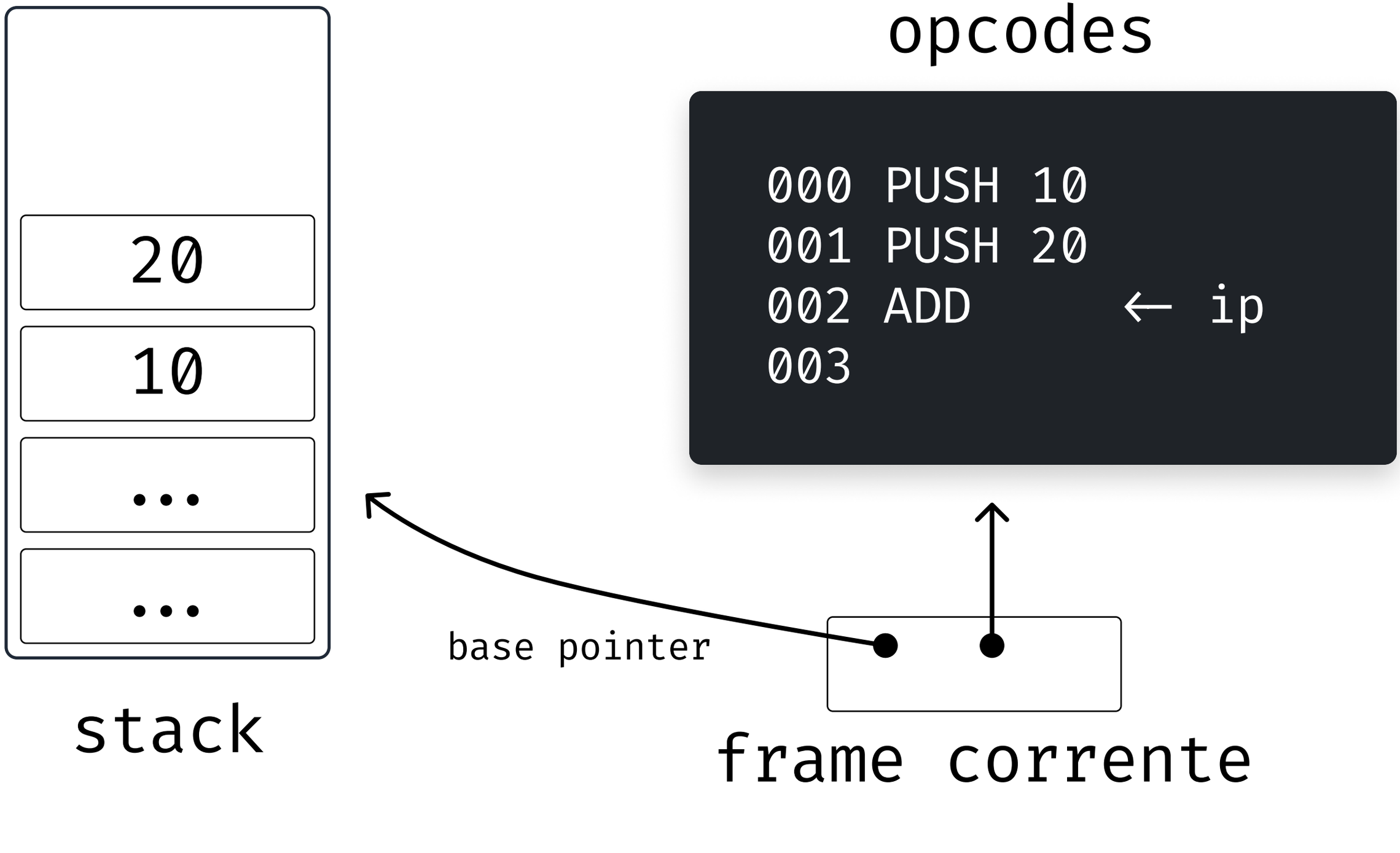
```
000 PUSH 10 ← ip
001 PUSH 20
002 ADD
003
```



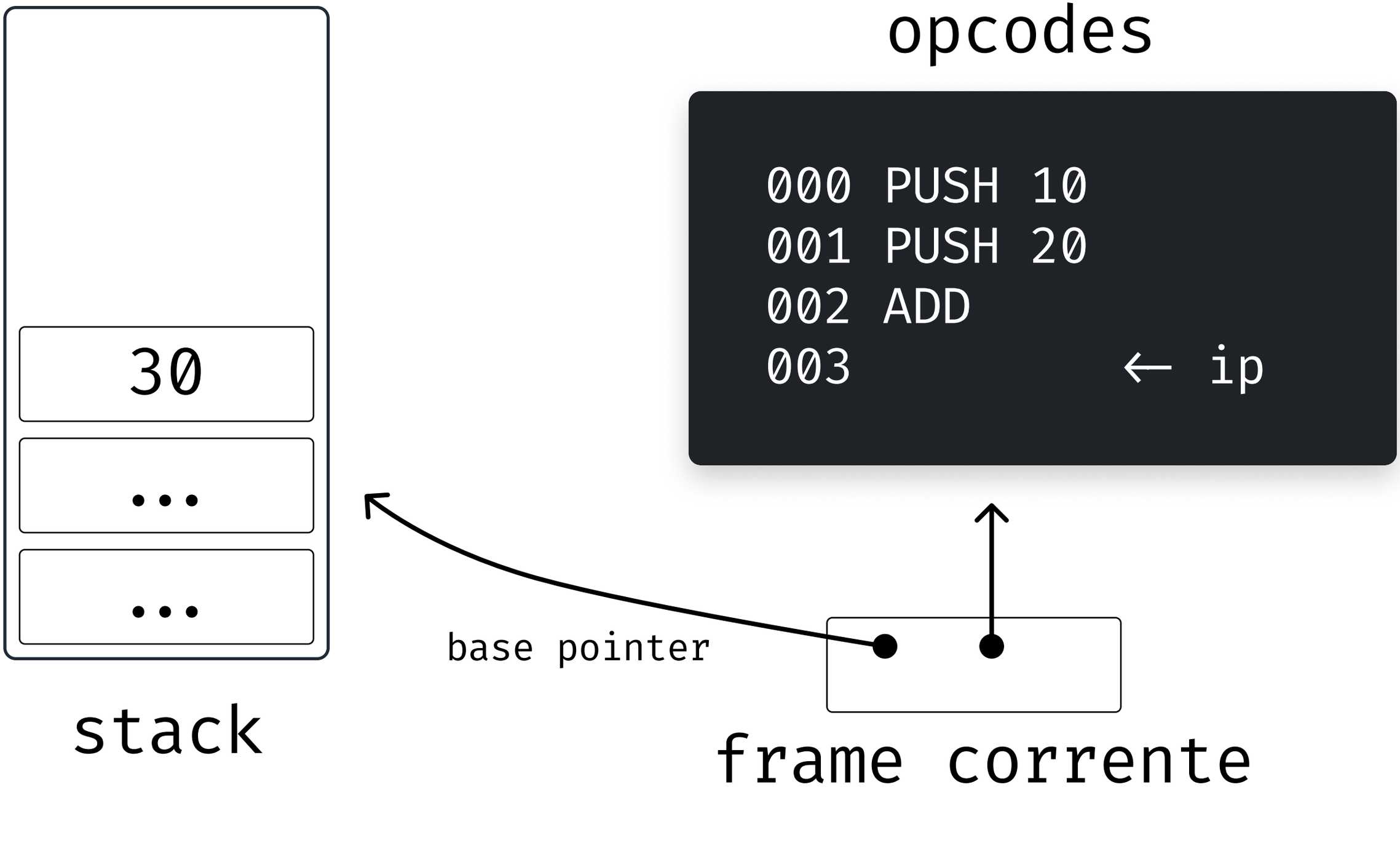
Architettura interprete / VM (stack)



Architettura interprete / VM (stack)



Architettura interprete / VM (stack)



Sintassi

```
;; atomi:
42 ;; numeri
"abc" ;; stringhe
abc ;; simboli
+
x?

;; liste
() ;; lista vuota (anche detta nil)
(abc "abc" (a c) +)
```

Semantica / Costanti

- valutate come sè stesse
- numeri, stringhe, nil, simboli **`true`** e **`false`**

```
42 ; ⇒ 42

"abc" ; ⇒ "abc"

; nil è un alias per ()
() ; ⇒ nil

true ; ⇒ true

false ; ⇒ false
```

Semantica / Costanti (opcodes)

- valutate come sè stesse
- numeri, stringhe, nil, simboli **`true`** e **`false`**

```
42
```

```
PUSH  42
```

Semantica / Forms

- interpretati come applicazione di funzione
- `(f a ... z)` notazione prefissa per `f(a, ..., z)`
- semantica strict

```
(+ 1 2) ; ⇒ 3
```

Semantica / Forms (opcode)

- interpretati come applicazione di funzione
- `(f a ... z)` notazione prefissa per `f(a, ..., z)`
- semantica strict

```
(+ 1 2) ; ⇒ 3
```

```
PUSH 1
PUSH 2
ADD
```

Semantica / Special Forms / `def`

- definisce bindings globali
- variabili immutabili: ri-definizioni successive fanno shadowing

```
(def x 42)

x ; ⇒ 42
```

Semantica / Special Forms / `def` (opcodes)

- definisce bindings globali
- variabili immutabili: ri-definizioni successive fanno shadowing
- variabili identificate da interi

```
(def x 42)

x ; ⇒ 42
```

```
PUSH 42
SET_GLOBAL 0

POP
GET_GLOBAL
```

Semantica / Special Forms / do

- valuta espressioni una ad una
- restituisce il risultato dell'ultima
- utile per side-effects (es. `print`)
- esegue opcode **POP** dopo ogni espressione

```
(do
  (log "hello")
  (log "world")
  (+ 1 2)) ; ⇒ 3
```

Semantica / Special Forms / `if`

- **`false`** e **`nil`** considerati *falsy*
- i valori restanti *truthy*

```
(def my-num 42)

(if (eq? my-num 42)
  "my-num equals 42"
  "my-num does not equal 42") ; ⇒ "my-num equals 42"
```

Semantica / Special Forms / `if` (opcodes)

- **`false`** e **`nil`** considerati *falsy*
- i valori restanti *truthy*

```
(if (eq? my-num 42)
  "a"
  "b")
```

```
0 - GET_GLOBAL 0
1 - PUSH 42
2 - EQ
# if not stack.pop():
#     instruction_pointer := 6
3 - JUMP_IF_NOT 6
4 - PUSH "a"
# instruction_pointer := 7
5 - JUMP 7
6 - PUSH "b"
7 -
```

Semantica / Special Forms / `lambda`

- definisce funzioni anonime
- *first-class citizen*

```clojure
(def max
  (lambda (x y)
    (if (> x y)
      x
      y))))

(max 10 20) ; ⇒ 20
```

Semantica / Special Forms / `quote`

- restituisce espressioni non valutate, come sintassi
- costrutto fondamentale per metaprogrammazione

```
(quote (+ 1 2)) ; ⇒ (+ 1 2)

;; zucchero sintattico
'(+ 1 2)
```

Semantica / Special Forms / `quote`

significante

La Sapienza

significato

Università a Roma

Semantica / Special Forms / `quote`

significante

La Sapienza

significato

Università a Roma

significante

“La Sapienza”

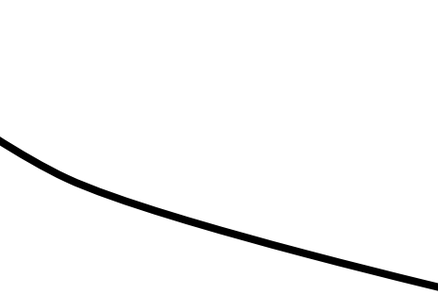
significato

Parola di 10 lettere

Semantica / Special Forms / `quote`

significante

La Sapienza



significato

Università a Roma

significante

“La Sapienza”

significato

Parola di 10 lettere

Semantica (riepilogo)

- costanti
- applicazione di funzione
- 5 special forms: **`def, if, do, lambda, quote`**

Metaprogrammazione

macro expansion

```lisp
(defun max (x y)
  (if (> x y)
    x
    y))
```

```lisp
(def max
  (lambda (x y)
    (if (> x y)
      x
      y))))
```

Metaprogrammazione

```lisp
(defmacro defun (name params body)
  (list 'def name
    (list 'lambda params body)))
```

macro expansion

```lisp
(defun max (x y)
  (if (> x y)
    x
    y))
```

```lisp
(def max
  (lambda (x y)
    (if (> x y)
      x
      y)))
```

Metaprogrammazione

```lisp
(defmacro defun (name params body)
  `(def (lambda ,params ,body)))
```

macro expansion

```lisp
(defun max (x y)
  (if (> x y)
    x
    y))
```

```lisp
(def max
  (lambda (x y)
    (if (> x y)
      x
      y)))
```

Metaprogrammazione

macro expansion

```lisp
(cond
  ((> 0 n) "positive")
  ((< 0 n) "negative")
  (otherwise "zero"))
```

```lisp
(if (> 0 n)
    "positive"
    (if (< 0 n)
        "negative"
        (if otherwise
            "zero"
            nil)))
```

Metaprogrammazione

macro expansion

```
(let1 (n (expensive-op 42))
  (* n n))
```

```
((lambda (n) (* n n))
  (expensive-op 42))
```

Metaprogrammazione

macro expansion

```
(and
  (positive? n)
  (even? n))
```

```
(let1 (G__42 (is-positive? n))
  (if G__42
    (even? n)
    G__42))
```

Conclusione

- core minimale, ma estremamente flessibile
- procedimento assiomatico

```
        (second nested)
        (list 'list (list 'backquote nested))))
    (list 'list (list 'quote nested))))

(defmacro backquote (body)
  (if (list? body)
    (if (eq? (first body) 'unquote)
      (second body)
      (cons 'concat (map body backquote-helper)))
    (list 'quote body)))

(defmacro let1 (binding &rest body)
    `((lambda (,(first binding)) ,@body)
      ,(second binding)))

(defmacro let (pairs body)
  (if (nil? pairs)
    body
    `(let1 (,@(first pairs))
       (let (,@(rest pairs)) ,body))))

(defmacro and (&rest clauses)
  (if (nil? clauses)
    true
    (let1 (s (gensym))
      `(let1 (,s ,(first clauses))
```