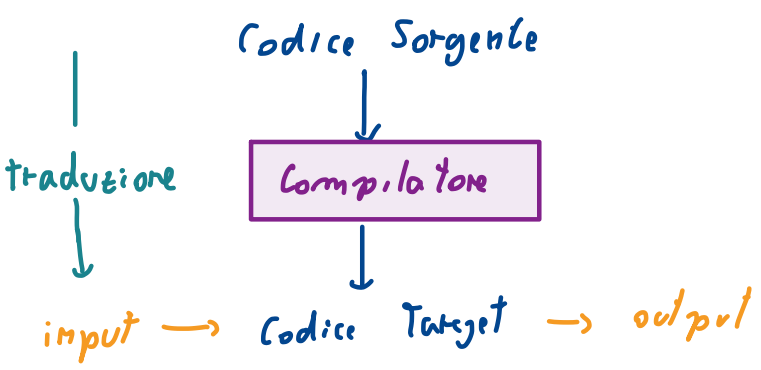
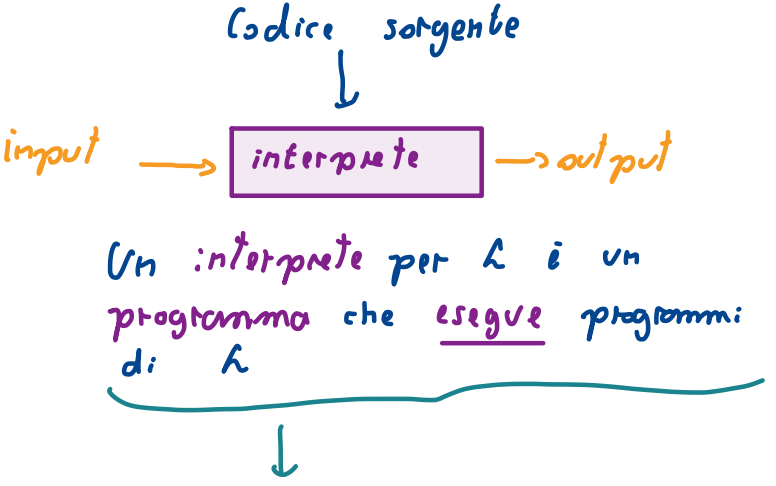
Interpreti e Compilatori

Obbiettivo: Capire le basi dell'implementazione di: OCaml





Un compilatore per $\mathcal{L}$ è un programma che traduce programmi di $\mathcal{L}$ } → codice come dato

$t \Rightarrow v$

$t \to t'$

$$\left\{ \frac{t \Rightarrow \text{fun } x \to t' \quad s \Rightarrow v \quad t'[v/x] \Rightarrow w}{ts \Rightarrow w} \right.$$

INTERPRETI

- "facili" da implementare ☺
- utili per capire come funziona un linguaggio ☺
  (→ definitional interpreter)
- poco efficienti ☹
- analisi estensionale (→ type checking) ☺/☹

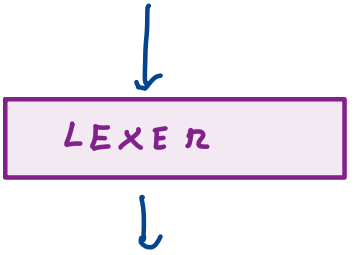
COMPILATORI

- "difficili" (almeno un intero corso dedicato ai compilatori) ☹
- semanticamente oscuri ☹
- efficienti ☺
- analisi e ottimizzazione (intensionale) ☺

Fasi di Compliazione e Interpretazione

① Analisi lessicale

`if x = 0 then 1 else fact (x - 1)` ⇝ stringa di: caratteri

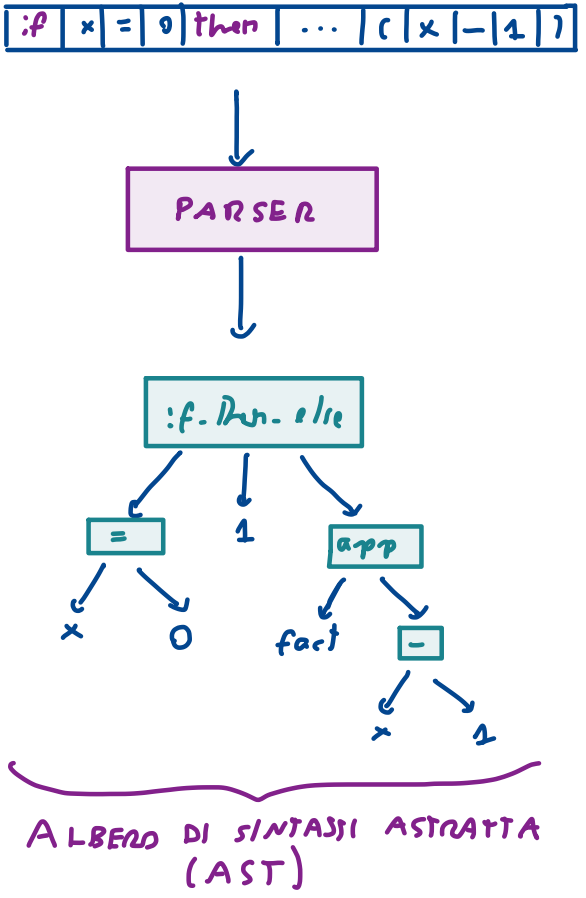


⇝ stream / lista di: token

Token = minima unità simbolica semanticamente sensata del linguaggio

⇝ parole

② Analisi Sintattica



AST catturano la struttura sintattica di una espressione

- dicono quali sono gli operatori
- quali e quanti: i loro argomenti

LA struttura è gerarchica e strutturata ~~ no parentesi

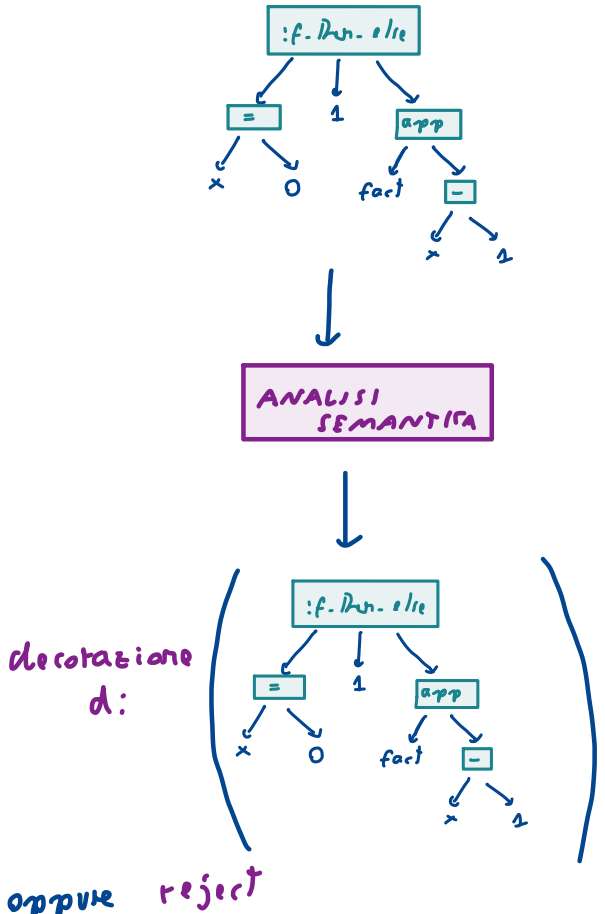
NB Sintassi è un tipo di dato

⇒ ricorsione

⇒ pattern matching

⋮

③ Analisi Semantica



Analisi semantica determina se il programma, visto come oggetto statico, è semanticamente sensato

- Creazione tabelle simboliche
  ident ↦ tipo
- Decorazione albero con tipi
- type inference e type checking
- Esaustività pattern matching
- ⋮

5 + "ciao"
↓

④ Interprete → esegue AST

Compilatore → traduce AST ... → codice macchina

Linguaggi funzionali ottimi per interpretazione e compilazione

- Base simbolica
- AST come tipo algebrico
- esecuzione / traduzione ricorsiva tramite pattern matching

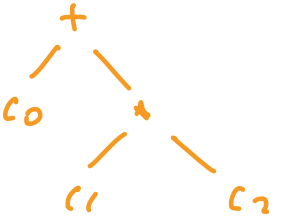
dinamica

3 + 2 → 5

3 + giallo

la casa è cresciuta

WARM-UP: Un interprete per aritmetica
(~> calcolatrice)



(m ∈ ℕ)

① Linguaggio

$e ::= c_m \mid e + e \mid e * e$ } SINTASSI ASTRATTA

BNF

$\Sigma = \{ c_m, +, *, (,), - \} \quad [m \in \mathbb{N}]$

alfabeto

$\Upsilon = \{ c_m, +, *, \ldots, END \}$

tokens

} SINTASSI CONCRETA

In ocaml

string list $\xrightarrow{\text{parse}}$ exp

string list: Σ^*

exp: tipo delle espressioni: AST

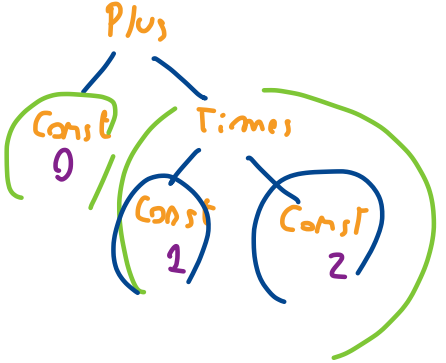
Una espressione è

- un intero
- oppure un operatore $(+, *)$ applicato a due espressioni

```
type exp =
    | Const of int
    | Plus of exp * exp
    | Times of exp * exp
```

Const n "c_n"

$c_0 + (c_1 * c_2)$



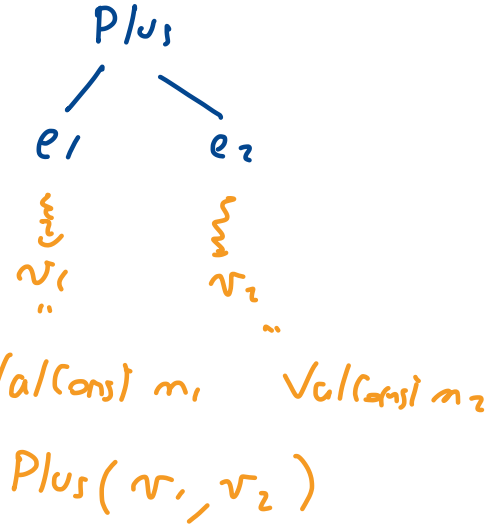
Plus (Const 0, Times (Const 1, Const 2))

prePy-printing : exp → string

Sintassi ✓

(Semantica Statica)

Semantica Dinamica



eval : exp → val

```
type val =
    | Valconst of int
```

```
let eval e = match e with
    | Const n → Valconst n
    | Plus (e1, e2) → let v1 = eval e1
                          v2 = eval e2
                      in
                          match v1 with
                            | Valconst n1 →
```

```
type exp =
    | Const of int
    | Plus of exp * exp
    | Times of exp * exp
```

] base / constant

] expression articulate

$C(e_1, \ldots e_n) \rightsquigarrow C(v_1, \ldots, v_n)$

redex

```
let rec eval e =
  match e with
    | Const n -> Valconst n
    | Plus (e1, e2) -> let v1 = eval e1
                           v2 = eval e2
                       in
                       match v1 with
                         | Valconst n1 -> match v2 with
                                            | Valconst n2 ->
                                                Valconst (n1 + n2)

    | Times (e1, e2) -> ....
```

$v_1 + v_2$

$(+) : int \to int \to int$

$v_1, v_2 : val$

plus_val : val -> val -> val

```
let plus_val (v1 : val) (v2 : val) : val =
    match v1 with
      | Valconst n1 -> match v2 with
                         | Valconst n2 -> Valconst (n1+n2)

let times_val (v1 : val) (v2 : val) : val =
                                  ⋮
```

```
let rec eval e =
    match e with
      | Const n -> Valconst n
      | Plus (e1, e2) -> let v1 = eval e1
                             v2 = eval e2
                         in plus_val v1 v2
      | Times (e1, e2) -> ....
```

```
type binop =
    | Plus
    | Times

type exp =
    | Const of int
    | App of binop * exp * exp

let primitive_op (op: binop) (v1: val) (v2: val) : val =
    match v1 with
        | Valconst n1 -> match v2 with
                | Valconst n2 -> match op with
                        | Plus -> Valconst (n1 + n2)
                        | Timer -> Valconst (n1 * n2)
```

```
let rec eval e = match e with
    | Const n -> Valconst n
    | App(op, e1, e2) -> let v1 = eval e1
                         v2 = eval e2
                         in primite_op op v1 v2
```

$e \longrightarrow e'$ $\quad e \Rightarrow v$

$$\frac{}{c_n \Rightarrow c_n} \qquad \frac{e_1 \Rightarrow c_{n_1} \quad e_2 \Rightarrow c_{n_2}}{e_1 + e_2 \Rightarrow c_{n_1 + n_2}}$$