Linguaggi per descrivere algoritmi

Un algoritmo è costituito da un insieme finito di istruzioni **non ambigue** ed **eseguibili**.

I linguaggi per descrivere algoritmi sono **linguaggi formali**

- Pseudocodice
- Linguaggi di programmazione: linguaggi per la specifica di algoritmi, eseguibli da un elaboratore

Un algoritmo "descritto" in un linguaggio di programmazione (o **implementato**) può essere sottoposto a test: verifica della sua correttezza mediante l'**esecuzione** del programma su un insieme di dati di test.

Un linguaggio di programmazione è un linguaggio artificiale con

- una **sintassi** rigida, che definisce l'insieme delle espressioni "ben formate" del linguaggio
- una **semantica** che definisce il significato di tutte le espressioni ben formate.

Linguaggi di programmazione

Un linguaggio di programmazione fornisce degli strumenti per descrivere algoritmi:

- un insieme di oggetti di base per rappresentare informazioni (**tipi di dati** predefiniti)
- un insieme di operazioni di base su tali oggetti (**operazioni primitive**)
- modalità per definire nuovi oggetti e nuove operazioni

Ogni linguaggio di programmazione si basa su un

modello di calcolo

un'idea astratta di come vengono eseguiti i calcoli

Objective Caml

Linguaggio della famiglia ML sviluppato e distribuito dall'INRIA (Francia) dal 1984

Disponibile al sito

`http://caml.inria.fr/`

nelle versioni per Linux, Windows NT, 95 e 98, e MacOS 7 e 8.

Modello di calcolo sottostante a OCaml:

PROGRAMMAZIONE FUNZIONALE

Un programma è una funzione

- I costrutti di base sono **ESPRESSIONI**
- Le espressioni sono costruite a partire da espressioni semplici (**COSTANTI**) mediante **APPLICAZIONE di operazioni**
- Si calcola riducendo un'espressione a un'altra più semplice, fino a ottenere un VALORE (un'espressione non ulteriormente semplificabile)

$$(6+3) \times (8-2) \Rightarrow 9 \times (8-2) \Rightarrow 9 \times 6 \Rightarrow 54$$

VALUTAZIONE

OCaml è un linguaggio di alto livello

```
let confronta (s1,s2) =
      s1=s2
```

L'operazione di confronto tra stringhe per verificare se sono uguali è predefinita, non serve definire un'operazione che confronti carattere per carattere

Ocaml consente di lavorare in

Modalità interattiva

Ciclo: *LETTURA, VALUTAZIONE, STAMPA*

```
        Objective Caml version 3.06

#
```

Il "cancelletto" è il *prompt* di Caml.

```
# 3*8;;
- : int = 24
```

- LETTURA: viene letta un'espressione (`3*8`), terminata da `;;` e dalla pressione del tasto di ritorno carrello (ENTER)
- VALUTAZIONE: viene calcolato il valore dell'espressione
- STAMPA: viene stampato il valore dell'espressione, specificando di che **tipo** è:

  `- :  int` il valore è di tipo `int` (intero)

  $= 24$ il valore è 24

```
# <ESPRESSIONE> ;;
- : <TIPO> = <VALORE>
```

OGNI ESPRESSIONE DEL LINGUAGGIO HA UN VALORE E UN TIPO

Quando si immette un'espressione al *prompt* di OCaml (seguita da `;;` e `<ENTER>`), OCaml ne calcola e stampa il valore e il tipo.

```
# (5/2, 5 mod 2);;
- : int * int = (2, 1)
# if not(3>0) then "pippo" else "pluto";;
- : string = "pluto"
```

`/`, `mod`, `not` sono **FUNZIONI PRIMITIVE** del linguaggio

`int * int` è il tipo delle coppie di interi

`if...then...else...` è un'espressione condizionale;
`if`, `then`, `else` sono **PAROLE CHIAVE**

`"pippo"`, `"pluto"` sono stringhe

L’ambiente di valutazione

La valutazione di un’espressione avviene in un

AMBIENTE

che contiene il “significato” (**VALORE**) di un insieme di parole (**VARIABILI**).

Quando si avvia l’interprete OCaml, le espressioni vengono valutate nell’ambiente predefinito, che contiene il “significato” delle operazioni primitive del linguaggio.

```
# succ;;
- : int -> int = <fun>
# succ 4;;
- : int = 5
# abs;;
- : int -> int = <fun>
# abs (-3);;
- : int = 3
# fst;;
- : 'a * 'b -> 'a = <fun>
# fst (3,"pippo");;
- : int = 3
```

L'ambiente può essere esteso mediante

DICHIARAZIONI

```
# let three = 3;;
val three : int = 3
# three * 8;;
- : int = 24
```

- **Una dichiarazione dà un nome a un valore**
- I nomi dei valori si chiamano VARIABILI

```
let <VARIABILE> = <ESPRESSIONE>;;
```

```
# let base = 3;;
val base : int = 3
# let altezza = 4;;
val altezza : int = 4
# let area = base * altezza;;
val area : int = 12
```

Mediante il linguaggio di programmazione è possibile “insegnare” al computer ad eseguire nuove operazioni.

DICHIARAZIONE DI FUNZIONI

```
let <NOME-FUNZIONE> = function <PARAMETRO-FORMALE> -> <ESPRESSIONE>;;
```

```
# let area_quadrato = function n -> n * n;;
val area_quadrato : int -> int = <fun>
```

`area_quadrato` è una funzione da interi a interi. Applicata a un intero `n`, riporta l'area del quadrato con lato di lunghezza `n`

ATTENZIONE: il valore di una funzione non è stampabile. OCaml informa soltanto che si tratta di una funzione (`<fun>`)

ML ha dedotto il tipo della funzione:

- Se `n` viene moltiplicato per sé stesso mediante la moltiplicazione tra interi (`*`), deve essere `n: int`.
  Quindi il dominio di `area_quadrato` è `int`.
- Il valore della funzione, `n*n`, è di tipo `int`.
  Quindi il codominio di `area_quadrato` è `int`.

Dichiarazione di funzioni: sintassi alternativa

La funzione `area_quadrato` si può definire anche così:

```
# let area_quadrato n = n * n;;
val area_quadrato : int -> int = <fun>
```

```
let <NOME-FUNZIONE> <PARAMETRO-FORMALE> = <ESPRESSIONE>
```

Definizione di una funzione che calcola il doppio di un intero:

```
# let double x = 2 * x;;
val double : int -> int = <fun>
```

ML ha dedotto il tipo della funzione:

- Se `x` viene moltiplicato per 2 (con la moltiplicazione tra interi), deve essere `x: int`.
  Quindi il dominio di `double` è `int`.
- Il valore della funzione, `x*2`, è di tipo `int`.
  Quindi il codominio di `double` è `int`.

Applicazione di funzioni

Abbiamo "insegnato" a OCaml a calcolare l'area di un quadrato ed il doppio di un intero.

Quando una funzione è stata dichiarata, si può utilizzare il suo nome nelle espressioni, in particolare per applicarle a un argomento:

```
# area_quadrato 5;;
- : int = 25
# double 3;;
- : int = 6
```

Tipi

L'argomento (parametro attuale) della funzione deve essere del tipo appropriato:

```
# area_quadrato "pippo";;
Characters 14-21:
  area_quadrato "pippo";;
                ^^^^^^^
This expression has type string but is here used with type int
```

NELLE ESPRESSIONI:

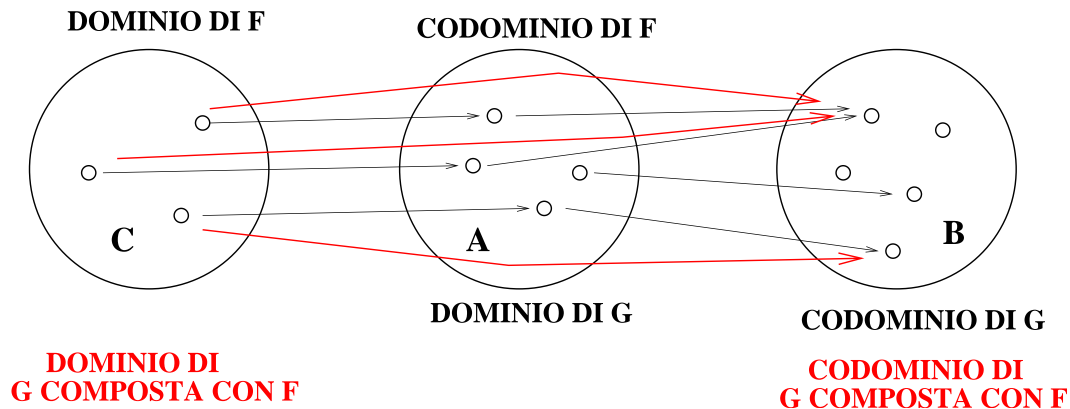
Il **tipo** degli argomenti deve essere consistente con quello richiesto dalle funzioni, senza eccezioni.

Composizione di funzioni

Se

$$F : C \to A \qquad \text{e} \qquad G : A \to B$$

allora si può applicare G a un valore prodotto da F:



Se $x : C$, allora $G(F(x)) : B$

La funzione che, applicata a un elemento $x \in C$ riporta il valore $G(F(x))$

$$function\ x \to G(F(x))$$

si chiama **composizione di G con F**:

$$G \circ F : C \to B$$

Esempio

Poichè `area_quadrato: int -> int` e `double: int -> int`, possiamo comporre `double` con `area_quadrato`:

```
# double (area_quadrato 3);;
- : int = 18
```

L'espressione `double (area_quadrato 3)` è corretta:

- `area_quadrato int -> int`
- `area_quadrato 3: int`
- `double: int -> int`, quindi `double` si può applicare a `area_quadrato 3`

La composizione di `double` con `area_quadrato` è la funzione:

```
# function n -> double (area_quadrato n);;
- : int -> int = <fun>
```

Questa funzione, applicata a un intero n calcola il doppio dell'area del quadrato di lato n

Possiamo darle un nome ed utilizzare tale nome:

```
# let doppia_area n = double (area_quadrato n);;
val doppia_area : int -> int = <fun>
# doppia_area 3;;
- : int = 18
```

Associatività dell'applicazione

ATTENZIONE: l'applicazione di funzioni è associativa a sinistra.

In mancanza di parentesi `double area_quadrato 3` viene interpretato come `(double area_quadrato) 3`, che non è un'espressione corretta:

- `area_quadrato int -> int`
- `double: int -> int`, quindi `double` non si può applicare a `area_quadrato`
- se anche `double area_quadrato` fosse corretto, il suo tipo sarebbe `int`, che non è il tipo di una funzione, quindi non si può applicare a 3

```
# double area_quadrato 3;;
Characters 0-6:
  double area_quadrato 3;;
  ^^^^^^
This function is applied to too many arguments
```

Un linguaggio di programmazione è un linguaggio formale, con una sintassi rigida che determina quali sono le espressioni corrette del linguaggio

Inferenza dei tipi

Quando si immette un'espressione o una dichiarazione:

- OCaml ne controlla la correttezza: un'espressione è corretta se è possibile determinarne il tipo.
- Ne determina il tipo, mediante un processo di **inferenza dei tipi**.
- Se si tratta di un'espressione, ne calcola il valore e lo stampa. Il valore viene calcolato nell'ambiente di valutazione attuale.
- Se si tratta di una dichiarazione, estende l'ambiente di valutazione creando un nuovo "legame" per la variabile dichiarata.

I numeri interi

Un linguaggio di programmazione fornisce:

- un insieme di oggetti di base per rappresentare informazioni: **tipi semplici**
- un insieme di operazioni di base su tali oggetti

Tra i tipi semplici di OCaml: `int`

Il tipo `int` è l'insieme dei numeri interi, denotati da sequenze di caratteri numerici eventualmente precedute dal segno: `0, 1, +4, -35, ...`

Operazioni sugli interi:

`+, -, *, /`: somma, sottrazione, prodotto e divisione intera

```
# 15 / 4;;
- : int = 3
```

`succ, pred`: successore e predecessore, di tipo `int -> int`

```
# succ 27;;
- : int = 28
# pred 0;;
- : int = -1
```

`mod`: resto della divisione (modulo):

```
# 15 mod 4;;
- : int = 3
```

Operatori di confronto e booleani

Operatori di confronto: `=, <, >, <=, >=`

```
# 12 > 5;;
- : bool = true
# 15 mod 4 = 0;;
- : bool = false
```

`true` e `false` sono valori di tipo `bool`

Sono gli unici valori di tipo `bool`

`15 mod 4 = 0` è un'**espressione di tipo** `bool`

Ci sono infinite espressioni di tipo `bool` (come ci sono infinite espressioni di tipo `int`), ma soltanto due valori di tipo `bool`

Operazioni booleane:

`not` : negazione

`&&` : congiunzione

`or` : disgiunzione

Espressioni condizionali

`if E then F else G`

è un'espressione condizionale se:

- `E` è di tipo `bool`
- `F` e `G` hanno lo stesso tipo

Le espressioni hanno sempre un tipo e un valore:

- Il tipo di `if E then F else G` è il tipo di `F` e `G`
- Il suo valore è:
  - il valore di `F` se `E` ha valore `true`
  - il valore di `G` se `E` ha valore `false`

Nel valutare un'espressione `if E then F else G`:

- se `E` è `true`, `G` non viene valutata
- se `E` è `false`, `F` non viene valutata

Esempio: valutazione di un'espressione condizionale

```
# 4 + (if 1 < 0 then 3 * 8 else 5 / 2);;
- : int = 6

4 + (if 1 < 0 then 3 * 8 else 5 / 2)
   ===> 4 + (if false then 3 * 8 else 5 / 2)
   ===> 4 + (5 / 2)
   ===> 4 + 2
   ===> 6
```

Il problema “massimo tra due numeri”

Problema: Dati due numeri interi n e m, calcolarne il massimo

Input: Una coppia di numeri interi (n, m) (di tipo `int * int`)

Output: Il massimo tra n e m (di tipo `int`)

Algoritmo `max: int * int -> int`

```
max (n,m):
   confrontare n e m,
   se n>m riportare n
   altrimenti riportare m
```

Programma:

```
let max(n,m) =
  if n>m then n else m
```

Verifica:

```
# max(4,6);;
- : int = 6
# max(6,4);;
- : int = 6
# max(6,6);;
- : int = 6
```

Il problema "ordinamento di una coppia di numeri"

Problema: Data una coppia di numeri interi (n,m), riportare la coppia ordinata:

$$(min(n,m), max(n,m))$$

Input: Una coppia di numeri interi (n,m) (di tipo `int * int`)

Output: La coppia costituita dagli stessi elmenti, ma ordinata (di tipo `int * int`)

Algoritmo `sort: int * int -> int * int`

```
sort (n,m):
   confrontare n e m,
   se n<m riportare (n,m)
   altrimenti riportare (m,n)
```

Programma:

```
let sort (n,m) =
  if n<m then (n,m)
  else (m,n)
```

Verifica:

```
# sort (1,2);;
- : int * int = (1, 2)
# sort (2,1);;
- : int * int = (1, 2)
# sort (2,2);;
- : int * int = (2, 2)
```

Polimorfismo

```
# let max(n,m) =
  if n>m then n else m;;
  val max : 'a * 'a -> 'a = <fun>
```

OCaml utilizza espressioni della forma `'a, 'b, 'c` per le variabili di tipo (α, β, γ).

Quindi per OCaml, `max` è una funzione polimorfa:

$$max : \alpha \times \alpha \to \alpha$$

```
# max (true, false);;
- : bool = true
```

Ogni coppia di valori che si possa confrontare con $=$, si può anche confrontare con gli operatori `<, >, <=, >=`.

Il tipo dell'argomento di `max` deve essere un'istanza dello schema $\alpha \times \alpha$: i due elementi della coppia devono essere dello stesso tipo

```
# max (2,true);;
Characters 5-11:
  max (2,true);;
      ^^^^^^^
This expression has type int * bool but is here used with type int * int
```

I tipi con uguaglianza

L'uguaglianza e gli operatori di confronto sono definiti su tutti i tipi di dati, tranne che sulle funzioni.

```
# true <> false;;
- : bool = true

# 3*8 <= 30;;
- : bool = true

# (4,true) <= (10,false);;
- : bool = true
```

```
# false < true;;
- : bool = true

# "abc" > "ABC";;
- : bool = true
```

```
# let double x = x*2;;
val double : int -> int = <fun>
# let treble x = x*3;;
val treble : int -> int = <fun>
# double = treble;;
Exception: Invalid_argument "equal: functional value"
```

Strutture di dati: coppie

- Coppie ordinate: `(E,F)`

```
# (5,8);;
- : int * int = (5, 8)
# ("pippo",7);;
- : string * int = ("pippo", 7)
# (true,80);;
- : bool * int = (true, 80)
```

- $t_1 \times t_2$ è il tipo delle coppie ordinate il cui primo elemento è di tipo t_1 ed il secondo di tipo t_2.

$$t_1 \times t_2 \text{ è il prodotto cartesiano di } t_1 \text{ e } t_2$$

- Attenzione: $\times$ è un **costruttore di tipo** (un'operazione su tipi): applicato a due tipi t_1 e t_2 costruisce il tipo delle coppie il cui primo elemento è di tipo t_1 ed il secondo di tipo t_2

Strutture di dati: tuple

- Triple, quadruple o "tuple" di dimensione qualsiasi:

```
# (true,5*4,"venti");;
- : bool * int * string = (true, 20, "venti")
#  ((if 3<5 then "true" else "false"), 10.3, 'K', int_of_string "50");;
- : string * float * char * int = ("true", 10.3, 'K', 50)
```

- Una tupla può essere elemento di una tupla:

```
# (true,("pippo",98),4.0);;
- : bool * (string * int) * float = (true, ("pippo", 98), 4)
```

- Attenzione: `*` non è associativo: ad esempio

  il tipo `bool * (int * string)` è diverso dal tipo `(bool * int) * string`

- Una funzione può essere un elemento di una tupla:

```
# (double,(true && not false, 6*5));;
- : (int -> int) * (bool * int) = (<fun>, (true, 30))
```

Costruttori e Selettori di un tipo di dati

Ogni tipo di dati è caratterizzato da un insieme di

COSTRUTTORI
(costanti + operazioni che "costruiscono"
valori di quel tipo)

e un insieme di

SELETTORI
(operazioni che "selezionano" componenti
da un valore del tipo)

Nel caso di tipi semplici (**`int, float, bool, string, char, unit`**), abbiamo soltanto costruttori:

i costruttori di un tipo di dati semplici sono tutti i valori del tipo

Costruttori e Selettori del tipo coppia

- L'insieme di parentesi e virgola `( , )` è il **costruttore** per il tipo delle coppie
- Per selezionare i componenti: `fst, snd`

```
# let t =  (true,("pippo",98));;
val t : bool * (string * int) = true, ("pippo", 98)
# fst t;;
- : bool = true
# snd t;;
- : string * int = "pippo", 98
# snd (snd t);;
- : int = 98
```

- `fst, snd` sono funzioni **polimorfe**:

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# snd;;
- : 'a * 'b -> 'b = <fun>
```

OCaml utilizza espressioni della forma `'a, 'b, 'c` per le variabili di tipo (α, β, γ).

Funzioni a più argomenti

```
# let area_triangolo (base,altezza) =
            (base * altezza)/2;;
val area_triangolo : int * int -> int = <fun>
```

• Qual è il tipo di `area_triangolo`?

• Per OCaml le funzioni hanno sempre un unico argomento; eventualmente una coppia o una tupla.

```
let quorem (n,m) =
    (n/m, n mod m);;

quorem : int * int -> int * int

# quorem (17,5);;
- : int * int = (3, 2)
```

POLIMORFISMO

```
# let first (x,y) = x;;
val first : 'a * 'b -> 'a = <fun>

# let zero x = 0;;
val zero : 'a -> int = <fun>

# let sort (x,y) = if x<y then (x,y)
                   else (y,x);;
val sort : 'a * 'a -> 'a * 'a = <fun>
```

`fst, zero, sort` sono funzioni **polimorfe**: hanno più tipi.

Il processo di **INFERENZA DEI TIPI** ha lasciato alcuni tipi completamente non ristretti: `'a` e `'b` possono essere *qualsiasi*:

```
let first (x,y) = x
```

`first(x,y)` $\Rightarrow$ se `x:` α e `y:` β,
dominio di `first`: $\alpha \times \beta$
`= x` $\Rightarrow$ codominio di `first`: α

```
let sort (x,y) = if x<y then (x,y) else (y,x)
```

`sort(x,y)` $\Rightarrow$ se `x:` α e `y:` β,
dominio di `sort`: $\alpha \times \beta$
= if `x < y` ... $\Rightarrow$ $\alpha = \beta$
`(x,y):` $\alpha \times \alpha$
`(y,x):` $\alpha \times \alpha$
`... then (x,y) ...` $\Rightarrow$ codominio di `sort`: $\alpha \times \alpha$
`else (y,x)` $\Rightarrow$ OK: il valore della funzione nel caso `then` e nel caso `else` è lo stesso

Schemi di tipo e istanze

- `'a * 'b -> 'a` è uno **schema di tipo**: indica un insieme infinito di tipi, tutti quelli della forma

  `T1 * T2 -> T1`

  Ogni tipo che si ottiene sostituendo `'a` con un tipo e `'b` con un tipo è un'**istanza** di `'a * 'b -> 'a`

  ```
  int * bool -> int
  int * int -> int
  (int * bool) * (int -> bool) -> (int * bool)
  ('a * 'b) * 'c -> ('a * 'b)
  ```

- `'a -> int` è uno **schema di tipo**: indica un insieme infinito di tipi, tutti quelli della forma

  `T -> int`

  Istanze di `'a -> int`:

  ```
  int * bool -> int
  'a * 'b -> int
  ('a -> 'b) -> int
  ```

- `'a * 'a -> 'a * 'a` è uno **schema di tipo**: indica un insieme infinito di tipi, tutti quelli della forma

  ```
  T * T -> T * T
  ```

  Istanze di `'a * 'a -> 'a * 'a`

  ```
  int * int -> int * int
  bool * bool -> bool * bool
  ```

All'interno di un'espressione, il tipo di un oggetto polimorfo viene istanziato:

- Il tipo dell'occorrenza di `first` nell'espressione

  ```
  first (8,true)
  ```

  è `int * bool -> int`

- Il tipo dell'occorrenza di `first` nell'espressione

  ```
  first (double,"pippo")
  ```

  è `(int -> int) * string -> (int -> int)`