Tecniche per risolvere problemi: riduzione a sottoproblemi più semplici

Ricorsione

Tecniche per risolvere problemi: riduzione a sottoproblemi più semplici

Problema: dati tre numeri interi, calcolarne il maggiore

Input: una tripla di numeri interi

Output: un intero

```
max3 (n,m,k):
   se n>m allora se n>k allora n
                 altrimenti k
   altrimenti se m>k allora m
              altrimenti k
```

Alternativa:

- definire l'operazione `max` che riporta il massimo tra due numeri
- il massimo di tre numeri è il massimo tra il primo e il massimo tra gli altri due

```
max(n,m):
   se n>m allora n altrimenti m

max3(n,m,k):
   max(n, max(n,k))
```

In Ocaml

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

let max3(n,m,k) =
  max(n, max(m,k))
```

Nota: ogni "nome" deve essere definito prima di essere usato

Test:

```
# max3 (1,2,3);;
- : int = 3
# max3(1,3,2);;
- : int = 3
# max3(3,2,1);;
- : int = 3
```

Commenti e documentazione di un programma

```
(* max: 'a * 'a -> 'a
   max(n,m) riporta il massimo tra n e m *)
let max(n,m) =
  if n>m then n else m

(* max3: 'a * 'a * 'a -> 'a
   max3(n,m,k) riporta il massimo tra n, m e k *)
let max3(n,m,k) =
  max(n, max(m,k))
```

Le parti di testo comprese tra i caratteri `(*` e `*)` sono **commenti**: non vengono interpretate da OCaml.

Esse costituiscono la **documentazione del programma**

Ogni dichiarazione di funzione in OCaml deve essere documentata includendo:

- la dichiarazione del **tipo** della funzione
- una **specifica dichiarativa** della funzione: descrivere **che cosa** calcola la funzione (non come).

Dichiarazioni locali

Problema : **dati tre numeri interi, n, m e k, calcolare il quoziente e il resto della divisione di $n+m$ per k**

Procedura :

```
procedure (n,m,k):
                    sia somma il valore di n+m;
                    riporta (somma/k, somma mod k)
```

Una procedura della forma:

`sia x il valore di E ; calcolare F`

si implementa in OCaml con la **dichiarazione locale** di `x` in `F`:

`let x = E in F`

(dove `E` e `F` sono espressioni).

Il *tipo* dell'espressione `let x=E in F` è il tipo di `F`.
Il suo *valore* è lo stesso valore che ha l'espressione `F` quando `x` è sostituito da `E`:
in particolare, quindi, il valore di

```
let somma = n+m in (somma/k, somma mod k)
```

è il valore di

```
((n+m)/k, (n+m) mod k)
```

Variabili locali

Nell'espressione

```
let x = E in F
```

`x` è una **variabile locale** tale che :

- `x` ha un valore (quello dell'espressione `E`) soltanto all'interno dell'espressione `F`.
- quando tutta l'espressione `let x = E in F` è stata valutata, `x` non ha più un valore.

```
# let x = 1+2 in x*8;;
- : int = 24
# x;;
Characters 0-1:
  x;;
  ^

Unbound value x
```

ATTENZIONE ! il legame "locale" sovrascrive altri eventuali legami "globali"

```
# let x="pippo";;
val x : string = "pippo"
# let x="pluto" in "ciao "^x;;    (* concatenazione di stringhe *)
- : string = "ciao pluto"
# x;;
- : string = "pippo"
```

Valutazione di una dichiarazione locale

Per valutare un'espressione della forma

```
let x = E in F
```

- viene calcolato il valore di `E`;
- la variabile `x` viene provvisoriamente legata al valore di `E`;
- tenendo conto di questo nuovo legame, viene calcolato il valore di `F`: questo è il valore dell'intera espressione;
- il legame provvisorio di `x` viene sciolto: `x` torna ad avere il valore che aveva prima o nessun valore.

Dichiarazioni locali nidificate

```
# let x=3*8
  in let y=4+1
  in x-y;;
- : int = 19

# let x=3*8
  in let y=x+1                (* il valore di x e' visibile *)
  in (x+y) mod 5;;            (* x+y=24+25 *)
```

Esercizio

Qualè il valore dell'espressione seguente?

```
let x = 3*8 in let x=x+1 in (x+x) mod 5
```

E qual è il valore di `x` dopo la valutazione dell'espressione?

RICORSIONE

A volte i sottoproblemi sono dello stesso tipo di quello principale:

la ricorsione è una tecnica per risolvere problemi complessi riducendoli a problemi più semplici dello stesso tipo.

Esempio: la funzione fattoriale

$$n! = 1 \times 2 \times ... \times (n-1) \times n$$

- C'è un caso "facile" del problema, che sappiamo risolvere subito: $n = 1$.
  In questo caso "base" $n! = 1$
- Se invece $n > 1$, il problema "fattoriale di n" si risolve facilmente se si sa risolvere il problema "fattoriale di $n-1$":

$$n! = n \times (n-1)! \text{ se } n > 1$$

Quindi, scrivendo $fact(n)$ invece di $n!$ si ha:

Caso base: $fact(1) = 1$

Sottoproblema: Volendo calcolare $fact(n)$, per $n > 1$, calcolare $fact(n-1)$

Utilizzazione della soluzione del sottoproblema: Per ottenere $fact(n)$, moltiplicare $fact(n-1)$ per n

Definizioni ricorsive

Definizione ricorsiva del fattoriale:

$$fact(n) = \begin{cases} 1 & \text{se } n = 1 \\ n \times fact(n-1) & \text{altrimenti} \end{cases}$$

Il fattoriale è "definito in termini di se stesso", ma per un caso "più facile".

La definizione si può "tradurre" immediatamente in un

programma OCaml

```
let rec fact n =
   if n=1 then 1
   else n * fact(n-1)
```

La **parola chiave** `rec` è necessaria per indicare che la definizione è ricorsiva

```
# let fact n = if n=0 then 1 else n * fact(n-1);;
Characters 36-40:
  let fact n = if n=0 then 1 else n * fact(n-1);;
                                      ^^^^

Unbound value fact
```

Calcolo mediante riduzioni

Processo di calcolo del valore di `fact 3`:

```
fact 3 = if 3=0 then 1 else 3 * fact(3-1)
       = 3 * fact(3-1)
       = 3 * fact 2
       = 3 * if 2=0 then 1 else 2 * fact(2-1)
       = 3 * (2 * fact 1)
       = 3 * (2 * if 1=0 then 1 else 1 * fact(1-1))
       = 3 * (2 * (1 * fact 0))
       = 3 * (2 * (1 * if 0=0 then 1
                               else 0 * fact(0-1)))
       = 3 * (2 * (1 * 1))
       = 3 * (2 * 1)
       = 3 * 2
       = 6
```

Risolvere problemi mediante ricorsione

Per risolvere un problema ricorsivamente occorre:

1. Identificare i casi semplicissimi (casi di base) che possono essere risolti immediatamente
2. Dato un generico problema complesso, identificare i problemi più semplici di esso la cui soluzione può aiutare a risolvere il problema complesso
3. Assumendo di saper risolvere i problemi più semplici (**ipotesi di lavoro**), determinare come operare sulla soluzione di tali problemi per ottenere la soluzione del problema complesso

per calcolare $F(n)$:

se n e' un caso base, riporta la soluzione per il caso n

altrimenti: risolvi i problemi piu' semplici

$F(n_1), F(n_2), ...F(n_k)$ (* **chiamate ricorsive** *)

combina le soluzioni ottenute e riporta

$combina(F(n_1), F(n_2), ...F(n_k))$

Un processo ricorsivo termina se le chiamate ricorsive si avvicinano ai casi di base: dopo un numero finito di chiamate ricorsive si arriva a casi base.

Il caso del fattoriale

Procedimento:

per calcolare $fact(n)$:

se $n = 1$, riporta la soluzione per il caso 1

altrimenti: risolvi il problema piu' semplice

$fact(n-1)$ (* **chiamata ricorsiva** *)

combina la soluzione ottenuta e riporta

$n \times fact(n-1)$

La chiamata ricorsiva è su $n-1$, che è più vicino al caso base 0, rispetto a n: dopo un numero finito di chiamate ricorsive si arriva al caso base $n = 1$

Esercizio: Cosa succede se $fact(n)$ è chiamata con $n < 1$? Per rispondere, eseguire qualche passaggio di riduzione dell'espressione $fact(0)$. Implementare poi la funzione fattoriale e richiamarla con argomento 0.

Ricorsione sui numeri naturali

$$\mathbb{N} = \{0, 1, 2, 3, ...\}$$

Un procedimento ricorsivo sui numeri naturali (per calcolare il valore di una funzione F sui naturali) ha questa forma:

Caso base ($n = 0$): definire il valore di F per 0

Caso ricorsivo ($n > 0$): si assume, come **ipotesi di lavoro**, di saper calcolare il valore $F(n-1)$; si determina allora come utilizzare questo valore per calcolare $F(n)$.

Somma dei primi n numeri naturali

Problema: dato un numero naturale n, calcolare la somma $0 + 1 + 2 + ... + n$

Input: un numero naturale

Output: un numero naturale

$$0 + 1 + 2 + ... + (n-1) + n = \underbrace{(0 + 1 + 2 + ... + (n-1))}_{caso\ piu'\ semplice} + n$$

Procedimento ricorsivo:

Caso base ($n = 0$): il valore è 0.

Caso ricorsivo ($n > 0$): assumiamo di saper calcolare il valore di

$$0 + 1 + 2 + ... + (n-1)$$

A questo valore è sufficiente aggiungere n per ottenere la somma dei primi n numeri naturali.

```
(* sumto : int -> int *)
(* sumto n = 0 + 1 + 2 + ... + n *)
let rec sumto n =
  if n = 0 then 0
  else n + sumto (n-1);;
```

Errori

La funzione `fact` è definita su $\mathbb{N} - \{0\}$: è una funzione parziale sugli interi.

La funzione `sumto` è definita su $\mathbb{N}$: è una funzione parziale sugli interi.

OCaml non ha un tipo di dati per $\mathbb{N}$ o per $\mathbb{N} - \{0\}$. Il tipo di `sumto` è:

```
sumto:  int -> int
```

Le restrizioni sui tipi non impediscono che `sumto` sia richiamata con argomento negativo.

```
# sumto (-1);;
Stack overflow during evaluation (looping recursion?).
```

Per impedire che si verifichi questo tipo di errore durante la valutazione, possiamo prevedere esplicitamente nel codice il caso "argomento negativo", rendendo la funzione totale sugli interi:

```
let rec fact n =
   if n <= 1 then 1
   else n * fact(n-1)

let rec sumto n =
  if n <= 0 then 0
  else n + sumto (n-1);;
```

Questo però non è sempre possibile, ed abbiamo definito funzioni diverse da quelle volute.

Gestione dei casi eccezionali

OCaml prevede un tipo di dati particolare, quello delle le **ECCEZIONI**:

`exn`

Le eccezioni consentono di scrivere programmi che segnalano un errore: cioè di definire funzioni parziali.

Procedura di definizione di una funzione parziale:

se *caso particolare* allora ERRORE
altrimenti ....

Esiste un insieme di **eccezioni predefinite**: `Match_failure, Division_by_zero,...`
Ma l'insieme dei valori del tipo `exn` può essere esteso, mediante la

dichiarazione di eccezioni:

```
(* dichiarazione di eccezione *)
exception NegativeNumber
```

Nomi delle eccezioni

Il nome di un'eccezione deve iniziare con una lettera maiuscola

Come segnalare un errore

Dopo aver dichiarato un'eccezione, l'eccezione può essere **sollevata**:

```
(* fact: int -> int
   fact n solleva l'eccezione NegativeNumber se n e' minore di 1,
          altrimenti riporta il fattoriale di n *)
let rec fact n =
  if n <= 0 then raise NegativeNumber
  else if n=1 then 1
       else n * fact (n-1)

# fact 3;;
- : int = 6
# fact (-1);;
Exception: NegativeNumber.
```

Come segnalare un errore

```
(* sumto: int -> int
   sumto n solleva l'eccezione NegativeNumber se n e' negativo,
           altrimenti riporta 0+1+2+...+n *)
let rec sumto n =
  if n < 0 then raise NegativeNumber
  else if n = 0 then 0
       else n + sumto (n-1);;

# sumto 4;;
- : int = 10
# sumto (-3);;
Exception: NegativeNumber.
```

Propagazione delle eccezioni

```
# 4 * fact (-1) ;;
Exception: NegativeNumber.
```

Se durante la valutazione un'espressione E viene sollevata un'eccezione, il calcolo del valore di E termina immediatamente (il resto dell'espressione non viene valutato), e viene sollevata l'eccezione.

```
(* loop : 'a -> 'b *)
let rec loop n = loop n;;

# let f=fact(-1) in loop f;;
Exception: NegativeNumber.
```

Catturare un'eccezione

Un'eccezione può essere **catturata** per implementare procedure di questo tipo:

```
calcolare il valore di E
se nel calcolo di tale valore si verifica un errore
     allora .....
```

```
# try 4 * fact(-1)
  with NegativeNumber -> 0;;
- : int = 0
```

Copia di una stringa

Problema: data una stringa S e un intero n, costruire la stringa che contiene n copie di S. Ad esempio, la stringa che contiene 3 copie di pippo è pippopippopippo

Input: una coppia di tipo `string * int`

Output: una stringa

Le stringhe

Costituiscono un tipo di dati semplici

Sono stringhe `pippo, pluto, 12Ev,...`: sequenze finite di caratteri delimitate dalle virgolette.

Alcune operazioni predefinite sulle stringhe:

concatenazione ^

si applica a una coppia di stringhe e restituisce una stringa

Si utilizza in notazione infissa:

```
# "programmazione " ^ "funzionale";;
- : string = "programmazione funzionale"
```

String.length: `string -> int` (lunghezza)

```
# String.length "pippo";;
- : int = 5
```

Copia di una stringa

Procedimento ricorsivo:

Caso base ($n = 0$): si vuole la stringa che contiene 0 copie di S; qualsiasi sia S, il risultato è la "stringa vuota" ‘‘

Caso ricorsivo ($n > 0$): Osserviamo che

$$\underbrace{s\hat{}s\hat{}s.....\hat{}s}_{n\ volte} = s\hat{}(\underbrace{s\hat{}s.....\hat{}s}_{n-1\ volte})$$

Quindi, se sappiamo calcolare la concatenazione della stringa s con se stessa per $n-1$ volte (ipotesi di lavoro), è sufficiente concatenare s a questo valore.

Cosa vogliamo fare nei casi "eccezionali", cioè quando $n < 0$?

Occorre dettagliare la specifica del problema:

Problema: data una stringa S e un intero n, costruire la stringa che contiene n copie di S, se $n > 0$, la stringa vuota altrimenti.

```
(* stringcopy : string * int -> string *)
(* stringcopy(s,n) = concatenazione di s con se stesso n volte *)
let rec stringcopy (s,n) =
    if n <= 0 then ""
    else s^stringcopy(s,n-1);;
```

Esercizio: implementare `sumto` e `stringcopy` e verificarne il comportamento su un insieme di dati di test.

Elevamento a potenza

Problema: dato un numero intero k e un numero naturale n, calcolare il valore di k^n

Input: due numeri interi ***Output:*** un numero intero

$$k^n = \underbrace{k \times k \times k \times ... \times k}_{n \ volte} = k \times (\underbrace{k \times k \times ... \times k}_{n-1 \ volte})$$

Procedimento ricorsivo sull'esponente n:

Caso base ($n = 0$): $n^0 = 1$ per ogni n

Caso ricorsivo ($n > 0$): assumiamo di saper calcolare il valore di k^{n-1} (ipotesi di lavoro).
Allora $k^n = k \times k^{n-1}$.

Casi eccezionali: $n < 0$. In questi casi la funzione è indefinita (Errore).

```
exception NegativeNumber

(* power : int * int -> int *)
(* power(k,n) = potenza n-esima di k *)
let rec power (k,n) =
  if n<0 then raise NegativeNumber
  else if n=0 then 1
  else k * power(k,n-1)
```

Esercizio: implementare `power` e verificarne il comportamento su un insieme di dati di test

Somma dei numeri compresi tra n e m

Problema: dati due numeri interi n e m, calcolare la somma dei numeri compresi tra n e m.

Input: due numeri interi ***Output:*** un numero intero

```
(* sumbetween : int * int -> int *)
(* sumbetween(n,m) = n + (n+1) + ... + (m-1) + m *)
```

$$n + (n+1) + ... + (m-1) + m = n + \underbrace{[(n+1) + ... + (m-1) + m]}_{chiamata\ ricorsiva}$$

In che senso calcolare la somma $(n+1) + ... + (m-1) + m$ è "più semplice che calcolare la somma $n + (n+1) + ... + (m-1) + m$?

Il primo termine della somma aumenta, l'ultimo resta uguale. Esiste una "misura" che diminuisce?

$$\text{se } m - n = k \qquad \text{allora} \qquad m - (n+1) = k - 1$$

La ricorsione è sulla differenza $k = m - n$

Somma dei numeri compresi tra n e m

Caso base ($m - n = 0$): dunque $n = m$. Allora la somma dei numeri compresi tra n e n è uguale a n stesso.

Caso ricorsivo ($m - n > 0$): dunque $m > n$. Poiché $m - (n + 1) < m - n$, la somma $(n + 1) + (n + 2) + ... + m$ è un caso "più semplice" e possiamo assumere di saperla calcolare. Al valore ottenuto è sufficiente aggiungere n.

Casi eccezionali: $m - n < 0$, cioè $n > m$. In questo caso il risultato è 0: poiché non ci sono numeri compresi tra n e m nel caso in cui $n > m$, la loro somma è 0.

```
(* sumbetween : int * int -> int *)
(* sumbetween(n,m) = n + (n+1) + ... + (m-1) + m *)
let rec sumbetween(n,m) =
  if n> m then 0
  else if n = m then n
       else n + sumbetween(n+1,m);;
```

Definizione ricorsiva di predicati

- `E && F` abbrevia `if E then F else false`
- `E or F` abbrevia `if E then true else F`

Verificare se un numero è una potenza di 2

$$2^k = \underbrace{2 \times 2 \times ... \times 2}_{k \ volte} \qquad 2^{k+1} = 2 \times 2^k$$

Quindi:

Procedimento: n è una potenza di 2 se

Caso base: `n = 1` ($n = 2^0$), oppure

Caso ricorsivo: `n` è pari e `n/2` è una potenza di 2

```
(* even: int -> bool
   sottoproblema di determinare se un numero e' pari:
        n e' pari se il resto della divisione n/2 e' 0 *)
let even n = n mod 2 = 0;;

(* power_of_two: int -> bool
   power_of_two n = true se n e' una potenza di 2
        assumendo che n sia positivo *)
let rec power_of_two n =
      n = 1 or (even n && power_of_two (n / 2));;
```

Esempio di valutazione di `power_of_two`

```
power_of_two ==> 6=1 or (even 6 && power_of_two (6 / 2))
          ==> false or (even 6 && power_of_two (6 / 2))
          ==> even 6 && power_of_two (6 / 2)
          ==> (6 mod 2 = 0) && power_of_two (6 / 2)
          ==> (0=0) && power_of_two (6 / 2)
          ==> true && power_of_two (6 / 2)
          ==> power_of_two (6 / 2)
          ==> power_of_two 3
          ==> 3=1 or (even 3 && power_of_two (3 / 2))
          ==> false or (even 3 && power_of_two (3 / 2))
          ==> even 3 && power_of_two (3 / 2)
          ==> (3 mod 2 = 0) && power_of_two (3 / 2)
          ==> (1 = 0) && power_of_two (3 / 2)
          ==> false && power_of_two (3 / 2)
          ==> false
```