

Uso di pattern nelle dichiarazioni di valore

Quando si scrive una dichiarazione della forma `let x = <ESPRESSIONE>`, la variabile `x` è un (caso particolare di) **pattern**.

La forma generale delle dichiarazioni di valore è:

```
let <PATTERN> = <ESPRESSIONE>
```

```
# let (x,y) = (3*8, (10<100 or false));;
val x : int = 24
val y : bool = true
# let (quoziente,resto) = (10/3, 10 mod 3);;
val quoziente : int = 3
val resto : int = 1

(* max_min_q : int -> int -> int * int * int *)
let max_min_q x y =
    (max x y, min x y, (max x y)/(min x y));;
# let (m,n,q) = max_min_q 5 12;;
val m : int = 12
val n : int = 5
val q : int = 2
```

Cos'è un pattern?

- Un **pattern** è una espressione costituita mediante **variabili** e **costruttori di tipo**.
- Per i tipi introdotti fin qui, i costruttori sono tutti e solo:
 - i **valori** dei tipi **int**, **float**, **bool**, **char**, **string**, **unit**
 - i **costruttori** di tuple (coppie, triple, ...), cioè parentesi e virgole
- Esempi:
 - **x** è un pattern, in quanto variabile;
 - **"pippo"** è un pattern, in quanto valore, quindi costruttore del tipo **string**;
 - **2.0** è un pattern, in quanto valore, quindi costruttore del tipo **float**;
 - **(x,y)** è un pattern, in quanto espressione ottenuta a partire da variabili e costruttori del tipo coppia;
 - **(0,x)** è un pattern, in quanto espressione ottenuta a partire da variabili, costruttori del tipo **int** e costruttori del tipo coppia;
 - **(x,true,y)** è un pattern, in quanto espressione ottenuta a partire da variabili, costruttori del tipo **bool** e costruttori del tipo tripla;
 - **x+y** NON è un pattern in quanto “+” non è un costruttore;
 - **-n** NON è un pattern in quanto “-” non è un costruttore.

Pattern matching: confronto con un pattern

Un valore **V** è **conforme a un pattern P** se è possibile sostituire le variabili in **P** con sottoespressioni di **V** in modo tale da ottenere **V** stesso

Pattern matching: confronto di un'espressione **E** con un pattern **P**:

- il confronto ha successo se il valore **V** di **E** è conforme al pattern **P**
- in caso di successo, viene stabilito come sostituire le variabili del pattern **P** in modo da ottenere **V**

```
# let (m,n,q) = max_min_q 5 12;;  
val m : int = 12  
val n : int = 5  
val q : int = 2
```

In una dichiarazione **let <PATTERN> = <ESPRESSIONE>**, se il confronto del valore di **<ESPRESSIONE>** con il **<PATTERN>** ha successo, l'ambiente viene esteso aggiungendo i legami delle variabili che risultano dal pattern matching

Pattern matching: esempi (I)

| <i>Pattern</i> | <i>Espressione</i> | <i>Pattern Matching</i> |
|-----------------|--------------------|--------------------------------|
| "pippo" | "pippo" | Successo |
| "pippo" | "pluto" | Fallimento |
| "pippo" | 0 | ERRORE |
| x | "pippo" | $x \Rightarrow \text{"pippo"}$ |
| x | "pi" ^ "ppo" | $x \Rightarrow \text{"pippo"}$ |
| x | 3*8 | $x \Rightarrow 24$ |
| ("pippo", true) | ("pippo", true) | Successo |
| ("pippo", true) | ("pippo", 3<0) | Fallimento |
| ("pippo", x) | ("pippo", 3<0) | $x \Rightarrow \text{false}$ |

Pattern matching: esempi (II)

| <i>Pattern</i> | <i>Espressione</i> | <i>Pattern Matching</i> |
|------------------|-----------------------------|--|
| x | (2.0, 3<0) | x ⇒ (2.0,false) |
| (n,m) | (2.0, 3<0) | n ⇒ 2.0 m ⇒ false |
| (n,m,k) | (2.0, 3<0) | ERRORE |
| (n,m,k) | (2.0, 3*8, 3>0) | n ⇒ 2.0 m ⇒ 24 k ⇒ true |
| (n,(m,k)) | (2.0, 3*8, 3>0) | ERRORE |
| (n,(m,k)) | (2.0, (3*8, 3>0)) | n ⇒ 2.0 m ⇒ 24 k ⇒ true |

Nota : Il pattern matching di un pattern costituito da una sola variabile con qualunque espressione, di qualunque tipo, ha sempre esito positivo.

Restrizione sui pattern e la variabile “muta”

In un pattern **non possono però esserci occorrenze multiple di una stessa variabile:**

(x,x) NON è un pattern

Restrizione sui pattern e la variabile "muta"

In un pattern **non possono però esserci occorrenze multiple di una stessa variabile**:

(x,x) NON è un pattern

Eccezione a questa regola: la **variabile muta**

| <i>Pattern</i> | <i>Espressione</i> | <i>Pattern Matching</i> |
|---------------------------|-----------------------------------|-------------------------|
| <code>_</code> | <code>"pippo"</code> | Successo |
| <code>("pippo", _)</code> | <code>("pippo", 3<0)</code> | Successo |
| <code>(n, _, m)</code> | <code>(2.0, 3*8, 3>0)</code> | n=2.0 m=true |
| <code>(_, _)</code> | <code>(2.0, 3*8, 3>0)</code> | ERRORE |
| <code>(_, _)</code> | <code>(2.0, (3*8, 3>0))</code> | Successo |

Uso di pattern nelle dichiarazioni di funzioni

Come nelle dichiarazioni di valore, anche nelle dichiarazioni di funzione i parametri sono pattern:

```
(* sort : 'a * 'a -> 'a * 'a *)
(* sort (x,y) = coppia con x e y ordinati *)
let sort (x,y) =
  if x<y then (x,y)
  else (y,x)
```

L'unico parametro di **sort** è indicato dal pattern **(x,y)**

L'uso dei pattern permette di evitare l'uso di **selettori**:

```
let sort pair =
  if fst pair < snd pair
  then (fst pair, snd pair)
  else (snd pair, fst pair)
```


Uso di pattern nelle espressioni funzionali

Analogamente, nelle espressioni funzionali della forma

function x -> <ESPRESSIONE>

la variabile **x** è un caso particolare di pattern.

Più in generale, un'espressione funzionale ha la forma:

function <PATTERN> -> <ESPRESSIONE>

Ad esempio, possiamo definire **sort** così:

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

Chiamata di funzioni definite mediante l'uso di pattern

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

```
# sort (3*8,100/2);;
```

```
sort (x,y)
```

| | |
|------|----------------------|
| y | 50 |
| x | 24 |
| sort | function(x,y) -> ... |
| ... | ... |

```
- : int * int = (24, 50)
```

| | |
|------|----------------------|
| sort | function(x,y) => ... |
| ... | ... |

Forma generale delle espressioni **function**

```
function P1 -> E1  
          | P2 -> E2  
          ...  
          | Pn -> En
```

- I pattern P_1, \dots, P_n devono essere tutti dello stesso tipo T_1 (o avere uno stesso sottotipo più generale T_1).
- Le espressioni E_1, \dots, E_n devono essere tutte dello stesso tipo T_2 (o avere tutte uno stesso sottotipo più generale T_2).
- Il tipo dell'espressione `function ...` è: $T_1 \rightarrow T_2$.

N.B. In espressioni di questa forma non si può utilizzare la parola chiave **fun**.

Esempio

```
let rec fact = function
  0 -> 1
  | n -> n * fact (n-1)
```

Si possono anche usare “pattern multipli”:

```
let rec fact = function
  0 | 1 -> 1
  | n -> n * fact (n-1)
```

Valutazione dell'applicazione di funzioni definite mediante espressioni **function** generali

```
let f = function Pattern1 -> E1  
      | Pattern2 -> E2  
      ...  
      | Patternn -> En
```

Per valutare **f (Expr)** (il valore di **f** applicata all'espressione **Expr**):

- 1 Viene calcolato il valore **V** dell'argomento **Expr**.
- 2 Il valore **V** viene confrontato con *Pattern*₁, *Pattern*₂, ..., nell'ordine:
Se il confronto dà sempre esito negativo, la valutazione riporta un errore.
Altrimenti:
 - Sia *Pattern*_{*i*} il primo pattern con cui il confronto di **V** ha successo; si aggiungono provvisoriamente i nuovi legami determinati dal pattern matching.
 - Con questi nuovi legami viene valutata la corrispondente espressione *E*_{*i*}.
 - Il valore di *E*_{*i*} viene riportato come valore di **f(Expr)**.
 - I legami provvisori vengono sciolti.

Esempi

```
(* gcd : int -> int -> int *)
(* gcd a b = massimo comun divisore di a e b *)
(* assumendo che almeno uno tra a e b sia diverso da 0 *)
let rec gcd a b =
  if b=0 then (abs a)
  else gcd b (a mod b)
```

Ma i due casi si possono distinguere mediante pattern:

```
let rec gcd a = function
  0 -> abs a
| b -> gcd b (a mod b)
```

Uso della variabile muta per indicare “in tutti gli altri casi...”

```
(* xor: bool * bool -> bool *)
let xor = function
  (true,false) | (false,true) -> true
| _ -> false
```

Pattern matching esplicito: espressioni **match**

match E with

$$\begin{array}{l} P_1 \rightarrow E_1 \\ | \\ P_2 \rightarrow E_2 \\ | \\ \dots \\ | \\ P_k \rightarrow E_k \end{array}$$

- I pattern P_1, \dots, P_n devono essere tutti dello stesso tipo, e dello stesso tipo di E (o avere un sottotipo in comune).
- Le espressioni E_1, \dots, E_n devono essere tutte dello stesso tipo T (o avere un sottotipo più generale in comune T).
- Tipo dell'espressione $\text{match } \dots : T$.
- Valutazione: se nessun pattern è conforme al valore V di E : *Match Failure*. Altrimenti, se P_i è il primo pattern conforme al valore V di E , l'ambiente viene provvisoriamente esteso con i nuovi legami determinati dal pattern matching, in questo ambiente viene valutato il valore V_i di E_i , i nuovi legami vengono sciolti, e V_i viene riportato come valore dell'espressione *match*.

Esempio

```
let rec fact n =  
  match n with  
  | 0 -> 1  
  | _ -> n * fact (n-1)
```

Cosa succede con questa definizione?

```
let rec fact n = function  
  | 0 -> 1  
  | n -> n * fact (n-1)
```


Esempio

```
let rec fact n =  
  match n with  
    0 -> 1  
  | _ -> n * fact(n-1)
```

Cosa succede con questa definizione?

```
let rec fact n = function  
  0 -> 1  
  | n -> n * fact(n-1)
```

Characters 50-59:

```
| n -> n * fact(n-1);;  
          ^^^^^^^^
```

Error: This expression has type int -> int
but an expression was expected of type int

Dichiarazioni locali: espressioni “let”

Dichiarazione locale di una variabile

let $x = E$ **in** F

Le **espressioni** hanno sempre un tipo e un valore.

Il tipo e il valore di **let** $x = E$ **in** F sono quelli dell’espressione che si ottiene da F sostituendo ovunque x con E .

x è una **variabile locale**:

- 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
```

let $x = E$ in F

- viene calcolato il valore v di E ;
- la variabile x viene provvisoriamente legata a v ;
- 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.

Cosa vi ricorda?

let $x = E$ in F

- viene calcolato il valore v di E ;
- la variabile x viene provvisoriamente legata a v ;
- 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.

Cosa vi ricorda?

$\text{let } x = E \text{ in } F \iff (\text{function } x \rightarrow F) E$

Esempio: **let [rec] f <parametri> = <corpo> in E**

DICHIARAZIONE-LET in ESPRESSIONE

- È un'espressione
- Il suo valore è il valore che ha **ESPRESSIONE** nell'ambiente che si ottiene estendendo l'ambiente attuale mediante **DICHIARAZIONE-LET**
- Per valutare

DICHIARAZIONE-LET in ESPRESSIONE

in un ambiente \mathcal{A} :

- 1 viene "valutata" **DICHIARAZIONE-LET** in \mathcal{A} (l'ambiente \mathcal{A} viene esteso)
- 2 viene valutata **ESPRESSIONE** nel nuovo ambiente
- 3 viene ripristinato l'ambiente \mathcal{A}

Dichiarazioni locali per evitare di calcolare più volte il valore di una stessa espressione

Esempio: dati tre interi n , m e k , determinare il quoziente e il resto di $n+m$ diviso k

```
(* esempio: int * int * int -> int * int *)  
let esempio (n,m,k) = ((n+m)/k, (n+m) mod k)
```

Il valore di $n+m$ viene calcolato 2 volte

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

O, equivalentemente:

```
let esempio(n,m,k) =  
  (function somma -> (somma/k, somma mod k)) (n+m)
```

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

Esempio: riduzione di una frazione ai minimi termini

Rappresentiamo una frazione mediante una coppia di interi.

```
(* gcd : int -> int -> int *)  
let rec gcd a = function  
  0 -> abs a  
  | b -> gcd b (a mod b)  
  
(* fraction : int * int -> int * int *)  
let fraction (n,d) = (n/gcd n d, d/gcd n d)
```

Esempio: riduzione di una frazione ai minimi termini

Rappresentiamo una frazione mediante una coppia di interi.

```
(* gcd : int -> int -> int *)  
let rec gcd a = function  
  0 -> abs a  
  | b -> gcd b (a mod b)  
  
(* fraction : int * int -> int * int *)  
let fraction (n,d) = (n/gcd n d, d/gcd n d)  
  
let fraction (n,d) =  
  let com = gcd n d  
  in (n/com, d/com)
```

Ha senso rendere gcd locale a fraction?

Esempio: riduzione di una frazione ai minimi termini

Rappresentiamo una frazione mediante una coppia di interi.

```
(* gcd : int -> int -> int *)
let rec gcd a = function
  0 -> abs a
  | b -> gcd b (a mod b)

(* fraction : int * int -> int * int *)
let fraction (n,d) = (n/gcd n d, d/gcd n d)

let fraction (n,d) =
  let com = gcd n d
  in (n/com, d/com)
```

Ha senso rendere gcd locale a fraction?

Motivi per dichiarare localmente una funzione:

- non ha significato autonomo
- consente di “risparmiare” parametri

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 “eccezionale” allora ERRORE
altrimenti

Esiste un insieme di **eccezioni predefinite**: **Match_failure**,
Division_by_zero,...

```
# int_of_string "123pippo";;  
Exception: Failure "int_of_string".  
# String.sub "0123456" 2 30;;  
Exception: Invalid_argument "String.sub".
```

I **nomi delle eccezioni** iniziano tutti con **una lettera maiuscola**

Come segnalare un errore

L'insieme dei valori del tipo **exn** può essere esteso, mediante la **dichiarazione di eccezioni**:

exception NegativeNumber

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

```
(* fact: int -> int
   fact n = fattoriale di n, se n non e' negativo,
           altrimenti solleva NegativeNumber *)
let rec fact n =
  if n < 0 then raise NegativeNumber
              (* viene "sollevata" l'eccezione *)
  else if n=0 then 1
       else n * fact (n-1)

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

Propagazione delle eccezioni

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

Se durante la valutazione di 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 x = loop x;;  
  
# 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
```

```
let rec gcd a b =
  try gcd b (a mod b)
  with Division_by_zero -> abs a (* b=0 *)
```

Attenzione:

Se **E** è di tipo **exn**:

- 1 **E** può essere il valore di qualunque funzione
- 2 **E** può essere argomento di qualunque funzione

Le eccezioni sono “eccezioni” alla tipizzazione forte.

Un esempio: conta_digits

conta_digits: string -> int

conta_digits s = numero di caratteri numerici in s

Un esempio: conta_digits

conta_digits: string -> int

conta_digits s = numero di caratteri numerici in s

Algoritmo iterativo:

```
max_index <- (length s) - 1; i <- 0; n <- 0
while i <= max_index:
  if carattere in posizione i e' numerico then n <- n+1;
  i <- i+1;
return n
```

Algoritmo ricorsivo:

Un esempio: conta_digits

conta_digits: string -> int

conta_digits s = numero di caratteri numerici in s

Algoritmo iterativo:

```
max_index <- (length s) - 1; i <- 0; n <- 0
while i <= max_index:
  if carattere in posizione i e' numerico then n <- n+1;
  i <- i+1;
return n
```

Algoritmo ricorsivo:

```
max_index <- (length s) - 1;
def: loop i = (* numero di caratteri numerici a partire
              da quello in posizione i *)
  if i > max_index then 0
  else if carattere in posizione i e' numerico
        then 1 + loop (i+1)
        else loop (i+1)
main: loop 0
```


Apriamo il manuale di OCaml

Nel Modulo Stdlib

<http://caml.inria.fr/pub/docs/manual-ocaml/libref/Stdlib.html>

- **val int_of_string : string -> int**

Convert the given string to an integer.

```
# int_of_string "123";;  
- : int = 123
```

Nel **Modulo String**

<http://caml.inria.fr/pub/docs/manual-ocaml/libref/String.html>

- **val get : string -> int -> char**

String.get s n returns the character at index n in string s.

You can also write **s.[n]** instead of **String.get s n**.

Raises **Invalid_argument** if n not a valid index in s.

- **val length : string -> int**

Return the length (number of characters) of the given string.

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

Un esempio: conta_digits

Sottoproblema: determinare se un carattere è numerico

```
(* numeric : char -> bool *)  
let numeric c =
```

Un esempio: conta_digits

Sottoproblema: determinare se un carattere è numerico

```
(* numeric : char -> bool *)
```

```
let numeric c =
```

```
  c >= '0' && c <= '9'
```

```
(* conta_digits: string -> int
```

```
  conta_digits s = numero di caratteri numerici in s *)
```

Un esempio: conta_digits

Sottoproblema: determinare se un carattere è numerico

```
(* numeric : char -> bool *)
let numeric c =
  c >= '0' && c <= '9'

(* conta_digits: string -> int
   conta_digits s = numero di caratteri numerici in s *)
let conta_digits s =
  let max_index = (String.length s) - 1 in
  (* loop: int -> int *)
  (* loop i = numero di caratteri numerici in s
     a partire da quello in posizione i *)
  let rec loop i =
    if i > max_index then 0
    else if numeric s.[i] then 1 + loop (i+1)
       else loop (i+1)
  in loop 0
```

La dichiarazione locale di loop consente di “risparmiargli” il parametro s

Uso “sporco” delle eccezioni

loop i = prova a estrarre dalla stringa il carattere *c* in posizione *i*;
se esiste: se numeric *c*, allora $1 + \text{loop } (i+1)$
altrimenti $\text{loop } i$
altrimenti vuol dire che $i > \text{String.length } s$, quindi riporta 0

Uso “sporco” delle eccezioni

loop i = prova a estrarre dalla stringa il carattere c in posizione i;
se esiste: se numeric c, allora 1 + loop (i+1)
altrimenti loop i
altrimenti vuol dire che $i > \text{String.length } s$, quindi riporta 0

```
let conta_digits s =  
  let rec loop i =  
    try if numeric s.[i] then 1 + loop (i+1)  
      else loop (i+1)  
    with Invalid_argument "index out of bounds" -> 0  
  in loop 0
```

Uso “sporco” delle eccezioni

loop i = prova a estrarre dalla stringa il carattere c in posizione i;
se esiste: se numeric c, allora 1 + loop (i+1)
altrimenti loop i
altrimenti vuol dire che i > String.length s, quindi riporta 0

```
let conta_digits s =  
  let rec loop i =  
    try if numeric s.[i] then 1 + loop (i+1)  
      else loop (i+1)  
    with _ -> 0  
  in loop 0
```

$$\begin{aligned}n! &= 1 \times 2 \times \dots \times n-1 \times n \\ &= (n-1)! \times n = n \times (n-1)!\end{aligned}$$

Caso “base”:

$$0! = 1$$

```
(* fact: int -> int *)  
let rec fact n =  
  if n=0 then 1  
  else n * fact (n-1)
```

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

rec è una parola chiave:

```
# let fact_errore n =  
  if n=0 then 1  
  else n * fact_errore (n-1) ;;
```


$$\begin{aligned}n! &= 1 \times 2 \times \dots \times n-1 \times n \\ &= (n-1)! \times n = n \times (n-1)!\end{aligned}$$

Caso “base”:

$$0! = 1$$

```
(* fact: int -> int *)
let rec fact n =
  if n=0 then 1
  else n * fact (n-1)
```

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

rec è una parola chiave:

```
# let fact_errore n =
  if n=0 then 1
  else n * fact_errore (n-1) ;;
Error: Unbound value fact_errore
```

Nei linguaggi funzionali “puri” non esistono costrutti di controllo per la realizzazione di cicli quali (**for**, **while**, **repeat**), ma il principale meccanismo di controllo è la ricorsione.

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

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 dello stesso tipo 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 dei problemi più semplici per ottenere la soluzione del problema complesso

Problema: valutazione di un'espressione aritmetica rappresentata da una stringa

Obiettivo: scrivere un programma che, data una stringa che rappresenta un'operazione aritmetica semplice tra interi non negativi, ne riporti il valore numerico.

Esempio: il valore riportato per la stringa "34+12" è l'intero 46. Le operazioni consentite sono somma, differenza, prodotto, divisione (intera).

Tipo e specifica dichiarativa della funzione principale

evaluate: string -> int

evaluate s = valore numerico dell'espressione rappresentata dalla stringa s.
Errore se s non rappresenta un'espressione aritmetica semplice

Progetto: riduzione a sottoproblemi

Identificazione di un sottoproblema utile:

split_string : string -> int * char * int

split_string s = (n,op,m)

dove n = primo operando,

op = carattere che rappresenta l'operazione,

m = secondo operando.

Per risolvere il problema split_string:

- cercare il primo carattere non numerico:

primo_non_numerico: string -> int

primo_non_numerico s = posizione del primo carattere non numerico
nella stringa s.

Errore se non esiste

- estrarre una parte di una stringa:

substring : string -> int -> int -> string

substring s j k = sottostringa di s che va dalla posizione j alla posizione k.

Sottoproblema primo_non_numerico (I)

primo_non_numerico: string -> int

primo_non_numerico s = posizione del primo carattere non numerico
nella stringa s. Errore se non esiste

Algoritmo (iterativo): iniziando con l'indice $i=0$, si incrementa i fino a che il carattere in posizione i è numerico. Quando il carattere in posizione i non è numerico, riportare il valore di i .

```
let primo_non_numerico s =  
  let rec loop i =  
    (* loop: int -> int *)  
    (* loop i = indice del primo carattere non numerico  
       in s, a partire da quello in posizione i *)  
    if not (numeric s.[i]) then i  
    else loop (i+1)  
  in loop 0
```

Soluzione del problema “substring”

substring : string -> int -> int -> string

substring s j k = sottostringa di s che va dalla posizione j alla posizione k.

Utilizziamo **String.sub : string -> int -> int -> string**

String.sub s start len returns a fresh string of length len, containing the substring of s that starts at position start and has length len.

La lunghezza della sottostringa riportata da **substring s j k** è **(k-j)+1**, quindi:

```
let substring s j k =  
  String.sub s j ((k-j)+1)
```

Soluzione del problema “split_string”

split_string : string -> int * char * int

split_string s = (n,op,m), dove, se **i** è la posizione del primo carattere non numerico di **s**:

- **n** è l'intero rappresentato dalla sottostringa di **s** che va dal primo carattere fino a quello in posizione **i-1**
- **op** è il carattere in posizione **i**
- **m** è l'intero rappresentato dalla sottostringa di **s** che va dal carattere in posizione **i+1** fino alla fine della stringa

Usiamo:

- **int_of_string** : string -> int per la conversione di stringhe in interi
- **String.length**: string -> int: **String.lengths s** = numero di caratteri di **s**

```
let split_string s =  
  let i = primo_non_numerico s  
  in (int_of_string (substring s 0 (i-1)),  
      s.[i],  
      int_of_string (substring s (i+1)  
                      ((String.length s)-1)))
```

Soluzione del problema principale “evaluate”

Occorre considerare diversi casi a seconda del carattere che rappresenta l'operazione (che deve essere una delle 4 operazioni)

```
(* evaluate: string -> int *)
let evaluate s =
  let (n,op,m) = split_string s
  in if op='+' then n+m
     else if op='-' then n-m
        else if op='*' then n*m
           else if op='/' then n/m
              else ???
```

let (n,op,m) = split_string s: sia **n** il primo elemento della tripla riportata come valore da **split_string s**, **op** il secondo e **m** il terzo.

Soluzione del problema principale “evaluate”

Occorre considerare diversi casi a seconda del carattere che rappresenta l'operazione (che deve essere una delle 4 operazioni)

exception **BadOperation**

```
(* evaluate: string -> int *)
let evaluate s =
  let (n,op,m) = split_string s
  in if op='+' then n+m
     else if op='-' then n-m
        else if op='*' then n*m
           else if op='/' then n/m
              else raise BadOperation
```

let (n,op,m) = split_string s: sia **n** il primo elemento della tripla riportata come valore da **split_string s**, **op** il secondo e **m** il terzo.

Uso di eccezioni diverse per identificare il tipo di errore

```
(* primo_non_numerico: string -> int *)  
let primo_non_numerico s =  
  let rec loop i =  
    if not (numeric s.[i]) then i  
    else loop (i+1)  
  in loop 0  
  
# primo_non_numerico "1234";;  
Exception: Invalid_argument "index out of bounds".
```

Uso di eccezioni diverse per identificare il tipo di errore

```
(* primo_non_numerico: string -> int *)
let primo_non_numerico s =
  let rec loop i =
    if not (numeric s.[i]) then i
    else loop (i+1)
  in loop 0

# primo_non_numerico "1234";;
Exception: Invalid_argument "index out of bounds".

let primo_non_numerico str =
  let rec loop i =
    if not (numeric str.[i]) then i
    else aux (i+1)
  in try loop 0
     with _ -> raise BadOperation
```

Uso di eccezioni diverse (II)

```
(* split_string : string -> int * char * int *)
let split_string s =
  let i = primo_non_numerico s
  in (int_of_string (substring s 0 (i-1)), s.[i],
      int_of_string(substring s (i+1) ((String.length s)-1)))

# split_string "43+abc";;
Exception: Failure "int_of_string".
```

Uso di eccezioni diverse (II)

```
(* split_string : string -> int * char * int *)
let split_string s =
  let i = primo_non_numerico s
  in (int_of_string (substring s 0 (i-1)), s.[i],
      int_of_string(substring s (i+1) ((String.length s)-1)))

# split_string "43+abc";;
Exception: Failure "int_of_string".
```

exception **BadInt**

```
(* split_string : string -> int * char * int *)
let split_string s =
  let i = primo_non_numerico s
  in try (int_of_string (substring s 0 (i-1)), s.[i],
          int_of_string (substring s (i+1)
                          ((String.length s)-1)))
      with _ -> raise BadInt
```

“evaluate” con un’espressione match

I diversi casi considerati da evaluate si distinguono a seconda della “forma” del carattere che rappresenta l’operatore

```
let evaluate s =  
  let (n,op,m) = split_string s  
  in match op with  
    | '+' -> n+m  
    | '-' -> n-m  
    | '*' -> n*m  
    | '/' -> n/m  
    | _ -> raise BadOperation
```

Pattern non esaustivi

```
# let evaluate s = let (n,op,m) = split_string s
                    in match op with
                        '+' -> n+m
                      | '-' -> n-m
                      | '*' -> n*m
                      | '/' -> n/m;;
```

Characters 54-127:

```
.....match op with
    '+' -> n+m
  | '-' -> n-m
  | '*' -> n*m
  | '/' -> n/m..
```

Warning 8: this pattern-matching is not exhaustive.

Here is an example of a value that is not matched: 'a'

```
val evaluate : string -> int = <fun>
```

```
# evaluate "35=22";;
```

Exception: Match_failure ("//toplevel//", 16, 5).