Definizioni ricorsive

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

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) ;;
```

Definizioni ricorsive

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

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
```

Ricorsione

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.

Apriamo il manuale di OCaml

Nel Modulo Pervasives
http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.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 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.

```
# String.sub "0123456" 2 3;;
- : string = "234"
```

- **val length : string -> int**
  Return the length (number of characters) of the given string.

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

Casi di errore

```
# int_of_string "123pippo";;
```

Casi di errore

```
# int_of_string "123pippo";;
Exception: Failure "int_of_string".

# String.sub "0123456" 2 30;;
```

Casi di errore

```
# int_of_string "123pippo";;
Exception: Failure "int_of_string".

# String.sub "0123456" 2 30;;
Exception: Invalid_argument "String.sub".
```

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.

Sottoproblema: determinare se un carattere è numerico.

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

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.

Sottoproblema: determinare se un carattere è numerico.

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

Nel Modulo String:
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.
Raise Invalid_argument if n not a valid index in s.

Sottoproblema primo_non_numerico (II)

Corpo del ciclo: **loop: string -> int -> int**

loop s i = posizione del primo carattere non numerico in s,
a partire dalla posizione i.

```
let rec loop s i =
  if not (numeric s.[i]) then i
  else loop s (i+1)
```

primo_non_numerico s = loop s 0

```
let primo_non_numerico s = loop s 0
```

(inizializza i a 0 prima di entrare nel ciclo)

Dichiarazioni locali

La funzione **loop** serve soltanto come funzione di supporto per **primo_non_numerico**: inutile definirla globalmente

```
let primo_non_numerico s =
  let rec loop s i =
    if not (numeric s.[i]) then i
    else loop s (i+1)
  in loop s 0
```

La funzione locale “vede” il parametro s della funzione principale, che in **loop** non cambia mai.

```
let primo_non_numerico s =
  (* loop: int -> int *)
  let rec loop 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.lengts 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.

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
```

Valutazione di espressioni “let”

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?

Valutazione di espressioni “let”

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 $\Longleftrightarrow$ **(function x -> F) E**

Forma generale delle dichiarazioni locali

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 *)
(* 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)

(* fraction : int * int -> int * int *)
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 *)
(* 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)

(* fraction : int * int -> int * int *)
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

Funzioni in forma currificata

```
(* gcd:
     int -> int -> int *)
let rec gcd a b =
  if b=0 then (abs a)
  else gcd b (a mod b)
```

```
(* gcd2:
     int * int -> int *)
let rec gcd2 (a,b) =
  if b=0 then (abs a)
  else gcd2 (b,a mod b)
```

gcd è la **forma currificata** di **gcd2**: calcola gli stessi valori, ma “consuma un argomento alla volta”

Funzioni in forma currificata

```
(* gcd:
     int -> int -> int *)
let rec gcd a b =
  if b=0 then (abs a)
  else gcd b (a mod b)
```

```
(* gcd2:
     int * int -> int *)
let rec gcd2 (a,b) =
  if b=0 then (abs a)
  else gcd2 (b,a mod b)
```

```
(* times:
     int -> int -> int *)
let  times m n = m * n
```

```
(* mult:
     int * int -> int *)
let mult (m,n) = m * n
```

gcd è la **forma currificata** di **gcd2**: calcola gli stessi valori, ma "consuma un argomento alla volta"

times è la **forma currificata** di **mult**: calcola gli stessi valori, ma "consuma un argomento alla volta"

times 5: int -> int è un'**applicazione parziale** di **times**.

sum (times 5) 1 10 $= \sum_{k=1}^{10}(5 \times k)$

Forma currificata di una funzione su tuple

In generale, f_c è la forma currificata di f se

$$\begin{aligned} f: &\quad t_1 \times \ldots \times t_n \to t \\ f_c: &\quad t_1 \to (t_2 \to \ldots \to (t_n \to t)\ldots) \end{aligned}$$

e per ogni $a_1, \ldots, a_n$: $f(a_1, \ldots, a_n) = (((f_c a_1)\, a_2) \ldots a_n)$

Le parentesi possono essere omesse

- sia nel tipo di f_c (si associa a destra),
- sia nell'applicazione di f_c (si associa a sinistra).

Espressioni per denotare funzioni

1. variabili (**times**)
2. astrazioni funzionali (**function x -> function y -> x*y**, o anche **fun x y -> x*y**)
3. espressioni funzionali (**comp double (times 5)**)

Altri esempi

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

# let lessthan x y = y < x;;
val lessthan : 'a -> 'a -> bool = <fun>
```

lessthan 0 è un predicato: essere minore di 0

```
# let greaterthan x y = y > x;;
val greaterthan : 'a -> 'a -> bool = <fun>

# let equal x y = x=y;;
val equal : 'a -> 'a -> bool = <fun>
```

equal 8 è un predicato: essere uguale a 8

Operazioni predefinite in Ocaml

Molte operazioni predefinite in Ocaml sono in forma currificata

```
# max;;
- : 'a -> 'a -> 'a = <fun>
```

Le operazioni infisse predefinite sono in forma currificata:

```
# ( * );;
- : int -> int -> int = <fun>

# (=);;
- : 'a -> 'a -> bool = <fun>
```

Definizione di operatori infissi

```
# let (@@) f g x = f(g x);;
val @@ : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# double @@ treble;;
- : int -> int = <fun>

# @@;;
Characters 0-2:
Syntax error

# (@@);;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

(* f: int -> int *)
let f = (times 2) @@ ((+) 100)
```

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,...**
Ma l’insieme dei valori del tipo **exn** può essere esteso, mediante la **dichiarazione di eccezioni**:

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

I **nomi delle eccezioni** iniziano tutti 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' negativo,
          altrimenti riporta il fattoriale di n *)
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
```

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.

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 =
  (* aux: int -> int *)
  let rec aux i =
      if not (numeric str.[i]) then i
      else aux (i+1)
  in try aux 0
     with Invalid_argument "index out of bounds"
                                -> 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 Failure "int_of_string" -> raise BadInt
```

Uso “sporco” delle eccezioni

```
(* conta_digits: string -> int *)
(* conta_digits s = numero di caratteri numerici in s *)
```

Uso “sporco” delle eccezioni

```
(* 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 dalla 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
```

Uso “sporco” delle eccezioni

```
(* 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 dalla 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

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 di pattern nelle dichiarazioni di valore

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

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 (n,op,m) = split_string "23+7";;
val n : int = 23
val op : char = '+'
val m : int = 7
```

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.

Restrizione sui pattern

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

(x,x) NON è un pattern

(con un'eccezione, la "*variabile muta*")

Pattern matching: confronto con un pattern

Un valore **V** è **conforme a un pattern P** se è possibile sostuire 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 (n,op,m) = split_string "23+7";;
val n : int = 23
val op : char = '+'
val m : int = 7
```

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)

Pattern	*Espressione*	*Pattern Matching*
"pippo"	"pippo"	Successo
"pippo"	"pluto"	Fallimento
"pippo"	0	ERRORE
x	"pippo"	x $\Rightarrow$ "pippo"
x	"pi"^"ppo"	x $\Rightarrow$ "pippo"
x	3*8	x $\Rightarrow$ 24
("pippo", true)	("pippo", true)	Successo
("pippo",true)	("pippo", 3<0)	Fallimento
("pippo", x)	("pippo", 3<0)	x $\Rightarrow$ false

Pattern matching: esempi (II)

Pattern	*Espressione*	*Pattern Matching*
x	(2.0, 3<0)	x $\Rightarrow$ (2.0,false)
(n,m)	(2.0, 3<0)	n $\Rightarrow$ 2.0 m $\Rightarrow$ false
(n,m,k)	(2.0, 3<0)	ERRORE
(n,m,k)	(2.0, 3*8, 3>0)	n $\Rightarrow$ 2.0 m $\Rightarrow$ 24 k $\Rightarrow$ true
(n,(m,k))	(2.0, 3*8, 3>0)	ERRORE
(n,(m,k))	(2.0, (3*8, 3>0))	n $\Rightarrow$2.0 m $\Rightarrow$ 24 k $\Rightarrow$ true

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

Uso di pattern nelle dichiarazioni di funzioni

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

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

L'**unico** parametro di **fraction** è indicato dal pattern **(n,d)**

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

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

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 **fraction** così:

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

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 P_1 **->** E_1
| P_2 **->** E_2
...
| P_n **->** E_n

- I pattern $P_1, ..., P_n$ devono essere tutti dello stesso tipo T_1.
- Le espressioni $E_1, ..., E_n$ devono essere tutte dello stesso tipo T_2.
- Il tipo dell'espressione `function ...` è: $T_1 \rightarrow T_2$.

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 $Pattern_1$ **->** E_1
| $Pattern_2$ **->** E_2
...
| $Pattern_n$ **->** E_n

Per valutare **F(Expr)** (cioè il valore di una funzione **F** applicata ad una 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 valutato il *corpo* della funzione E_i.
   - Il valore di E_i viene riportato come valore di **F(Expr)**.
   - I legami provvisori vengono sciolti.

La variabile muta : esempi di pattern matching

Pattern	***Espressione***	***Pattern Matching***
`_`	`"pippo"`	`Successo`
`("pippo", _)`	`("pippo", 3<0)`	`Successo`
`(n,_,m)`	`(2.0, 3*8, 3>0)`	`n=2.0` `m=true`
`(_,_)`	`(2.0, 3*8, 3>0)`	`ERRORE`
`(_,_)`	`(2.0, (3*8, 3>0))`	`Successo`

Esempi

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

let primo_non_numerico str =
  let rec aux i = ...
  in try aux 0
     with _ -> raise BadOperation

let split_string s =
  let i = primo_non_numerico s
  in try ....
     with _ -> raise BadInt

let conta_digits s =
  let rec loop i = try ....
                   with _ -> 0
  in loop 0
```

Espressioni **match**

match E **with**
P_1 **->** E_1
| P_2 **->** E_2
| **...**
| P_k **->** E_k

- I pattern $P_1, ..., P_n$ devono essere tutti dello stesso tipo, e dello stesso tipo di E.
- Le espressioni $E_1, ..., E_n$ devono essere tutte dello stesso tipo T.
- Tipo dell'espressione **match ...**: T.

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
```

“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
```

Valutazione di espressioni **match**

match F with $Pattern_1$ **->** E_1
| $Pattern_2$ **->** E_2
...
| $Pattern_n$ **->** E_n

Per valutare l'espressione:

1. Viene valutata l'espressione **F**
2. Il valore ottenuto viene confrontato con $Pattern_1$, $Pattern_2$, ..., nell'ordine; se il confronto dà sempre esito negativo, la valutazione riporta un errore (**Match_failure**).
   Altrimenti:
   - Sia $Pattern_i$ il primo pattern con cui il valore di **F** si confronta positivamente: si aggiungono provvisoriamente i nuovi legami determinati dal pattern matching.
   - Con questi nuovi legami viene valutata l'espressione E_i.
   - Il valore di E_i viene riportato come valore di **Expr**.
   - Vengono sciolti i legami provvisori.

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).
```

Regole di calcolo per la valutazione di espressioni

MODELLO DI CALCOLO:
CALCOLARE = RIDURRE

```
# let square x = x * x;;
val square : int -> int = <fun>
     square (3*2) ==> square 6 ==> 6 * 6 ==> 36
     square (3*2) ==> (3*2)*(3*2) ==> 6*(3*2)
                  ==> 6*6 ==> 36
```

Regole di calcolo:

- **CALL BY VALUE**: calcolare il valore dell'argomento prima di applicare una funzione
- **CALL BY NAME**: applicare la funzione prima di aver calcolato il valore dell'argomento

Regola di calcolo di ML: call by value
Eccezioni: espressioni condizionali, operatori booleani

Necessità di espressioni “lazy”

```
let rec fact n = if n<=0 then 1
                 else n * fact(n-1)
```

Call by value:

```
fact 1 ==> if 1 <= 0 then 1 else 1 * fact(1-1)
       ==> if false then 1 else 1 * fact 0
       ==> if false then 1
           else 1 * (if 0<=0 then 1 else 0 * fact(0-1))
       ==> if false then 1
           else 1 * (if true then 1 else 0 * fact(-1))
       ==> if false then 1
           else 1 * (if true then 1
                     else 0 * (if -1<=0 then 1
                               else fact(-1-1)))
       ==> if false then 1
           else 1 * (if true then 1
                     else 0 * (if true then 1
                               else fact(-2)))
       ..................
```