Espressioni aritmetiche

Consideriamo espressioni costruite a partire da variabili e costanti intere mediante applicazione delle operazioni di somma, sottrazione, prodotto e divisione (intera).

Ad esempio:

$$(3 + (5 \times x))/(9 - y)$$

L'insieme delle espressioni aritmetiche si può **definire induttivamente**:

(i) se n è un numero intero, allora n è un'espressione;

(ii) se x è una variabile, allora x è un'espressione;

(iii) se E_1 e E_2 sono espressioni, allora anche $(E_1 + E_2)$, $(E_1 - E_2)$, $(E_1 \times E_2)$ e (E_1/E_2) sono espressioni;

(iv) nient'altro è un'espressione.

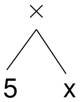
Osservazioni

1. L'insieme così definito contiene un numero infinito di “oggetti di base”.
2. Ci sono quattro “costruttori funzionali” (somma, sottrazione, moltiplicazione, divisione): quattro modi diversi di costruire oggetti complessi a partire da oggetti più semplici.
3. I costruttori funzionali si applicano a **due** oggetti più semplici per costruire un oggetto più complesso.

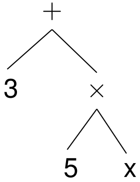
Strutture ad albero

Poiché i costruttori funzionali si applicano a più di un oggetto la struttura di un'espressione aritmetica non è lineare (come quella di una lista), ma è una struttura ad albero.

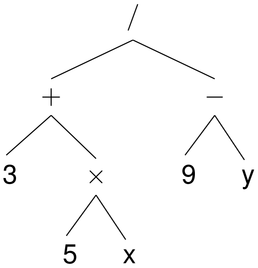
$5 \times x$



$(3 + (5 \times x))$



$(3 + (5 \times x))/(9 - y)$



Rappresentazione delle espressioni in OCaml

Possiamo definire un tipo di dati induttivo, ricalcando la definizione induttiva delle espressioni:

```
type expr =
    Int of int
  | Var of string
  | Sum of expr * expr
  | Diff of expr * expr
  | Mult of expr * expr
  | Div of expr * expr
```

espressione	*rappresentazione*
4	Int 4
$4 \times x$	Mult (Int 4, Var "x")
$7 - 3$	Diff(Int 7,Int 3)
$3 + (y \times 2)$	Sum (Int 3, Mult (Var "y", Int 2))

Problema: valutazione di un'espressione in un ambiente

Rappresentazione di ambienti mediante liste associative:

```
type ambiente = (string * int) list
```

`[("x",5);("z",0);("y",4)]` rappresenta l'ambiente

x	5
z	0
y	4

eval: ambiente -> expr -> int

eval env e = valore dell'espressione **e** nell'ambiente **env**. Errore se qualche variabile in **e** non ha un valore associato in **env**.

Vai al codice

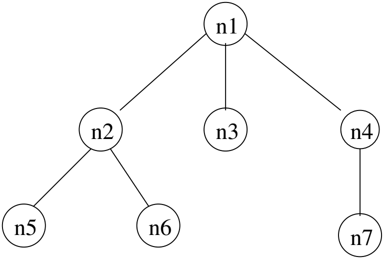
Alberi

Un albero è un insieme di oggetti, chiamati **nodi**, su cui è definita una relazione binaria $G(n, m)$ – che leggiamo "n è **genitore** di m" – tale che:

1. esiste un unico nodo, chiamato **radice**, che non ha genitori;
2. ogni nodo diverso dalla radice ha uno ed unico genitore;
3. per ogni nodo n diverso dalla radice esiste un **cammino** dalla radice a n (l'albero è *connesso*):
   esistono nodi $n_1, .., n_k$ ($k \geq 1$) tali che n_1 è la radice dell'albero, $n_k = n$ e, per ogni $i = 1, ..., k-1$, n_i è genitore di n_{i+1}.

Se n è il genitore di m, allora m è un **figlio** di n.

Un **albero binario** è un albero in cui ogni nodo ha al massimo due figli.



Un po' di terminologia

cammino: sequenza di nodi $n_1, ..., n_k, n_{k+1}$ tale che, per $i = 1, ..., k$, n_i è il genitore di n_{i+1}.

lunghezza del cammino: k (numero degli archi)

antenato e discendente: se esiste un cammino da n a m, allora n è un antenato di m e m un discendente di n

fratelli: nodi che hanno lo stesso genitore

sottoalbero: insieme costituito da un nodo n e tutti i suoi discendenti; n è la radice del sottoalbero

foglia: nodo senza figli

nodo interno: nodo con uno o più figli

profondità di un nodo: lunghezza del cammino dalla radice al nodo

altezza di un nodo: lunghezza del cammino più lungo che va dal nodo a una foglia

altezza dell'albero: altezza della sua radice = profondità massima di un nodo nell'albero

dimensione di un albero: numero dei nodi

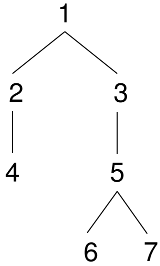
Alberi binari

Definizione degli alberi binari per induzione strutturale

1. Un nodo n è un albero binario (una foglia), con radice n.
2. Se T_0 è un albero binario con radice n_0 e n è un nuovo nodo, allora la struttura che si ottiene aggiungendo n come genitore di n_0 è un albero binario con radice n.
3. Se T_0 e T_1 sono alberi binari con radici n_0 e n_1, rispettivamente, e n è un nuovo nodo, allora la struttura che si ottiene aggiungendo n come genitore di n_0 e di n_1 è un albero binario con radice n.
4. Nient'altro è un albero binario.

Rappresentazione di alberi binari

```
type 'a tree =
    Leaf of 'a
  | One of 'a * 'a tree
  | Two of 'a * 'a tree * 'a tree
```




```
Two(1, One(2, Leaf 4),
       One(3, Two(5, Leaf 6, Leaf 7)))
```

è un **int tree**

Il tipo di un albero binario dipende dal tipo dei nodi:

- **Leaf, One, Two sono costruttori polimorfi**
  Leaf: 'a -> 'a tree
  One: 'a * 'a tree -> 'a tree
  Two: 'a * 'a tree * 'a tree -> 'a tree
- **tree è un costruttore di tipi**

Calcolo della dimensione di un albero binario

Definizione induttiva del tipo ⇒ definizione ricorsiva di funzioni sul tipo

```
type 'a tree =
    Leaf of 'a
  | One of 'a * 'a tree
  | Two of 'a * 'a tree * 'a tree

(* size : 'a tree -> int *)
(* size t = numero di nodi in t *)
let rec size = function
    Leaf _ ->
  | One(_,t) ->

  | Two(_,t1,t2) ->
```

Calcolo della dimensione di un albero binario

Definizione induttiva del tipo ⇒ definizione ricorsiva di funzioni sul tipo

```
type 'a tree =
    Leaf of 'a
  | One of 'a * 'a tree
  | Two of 'a * 'a tree * 'a tree

(* size : 'a tree -> int *)
(* size t = numero di nodi in t *)
let rec size = function
    Leaf _ -> 1
  | One(_,t) ->
        (* ipotesi: si sa calcolare size t *)
  | Two(_,t1,t2) ->
```

Calcolo della dimensione di un albero binario

Definizione induttiva del tipo ⇒ definizione ricorsiva di funzioni sul tipo

```
type 'a tree =
    Leaf of 'a
  | One of 'a * 'a tree
  | Two of 'a * 'a tree * 'a tree

(* size : 'a tree -> int *)
(* size t = numero di nodi in t *)
let rec size = function
    Leaf _ -> 1
  | One(_,t) -> 1 + size t
        (* ipotesi: si sa calcolare size t *)
  | Two(_,t1,t2) ->
        (* ipotesi: si sa calcolare size t1 e size t2 *)
```

Calcolo della dimensione di un albero binario

Definizione induttiva del tipo ⇒ definizione ricorsiva di funzioni sul tipo

```
type 'a tree =
    Leaf of 'a
  | One of 'a * 'a tree
  | Two of 'a * 'a tree * 'a tree

(* size : 'a tree -> int *)
(* size t = numero di nodi in t *)
let rec size = function
    Leaf _ -> 1
  | One(_,t) -> 1 + size t
        (* ipotesi: si sa calcolare size t *)
  | Two(_,t1,t2) -> 1 + size t1 + size t2
        (* ipotesi: si sa calcolare size t1 e size t2 *)
```

Rappresentazione alternativa degli alberi binari

Per rendere le definizioni più compatte, è utile includere l'insieme vuoto tra gli alberi binari: **l'albero "vuoto"** (**Empty**).

In questo modo, ogni albero diverso da **Empty** ha esattamente due sottoalberi: se è una foglia, i sottoalberi sono vuoti, se è un nodo con un solo figlio, uno dei due sottoalberi è vuoto.

Definizione induttiva

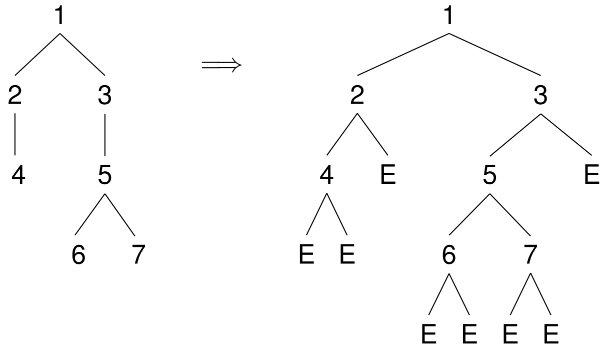
1. **Empty** è un albero binario
2. Se **t1** e **t2** sono alberi binari e **n** è un nuovo nodo, allora **Tr(n,t1,t2)** è un albero binario
3. Nient'altro è un albero binario

Costruttori: **Empty** : **'a tree**
Tr : **'a * 'a tree * 'a tree -> 'a tree**

Dichiarazione di tipo:

```
type 'a tree = Empty
             | Tr of 'a * 'a tree * 'a tree
```

Esempio




```
Tr(1, Tr(2, Tr(4,Empty,Empty), Empty),
      Tr(3, Tr(5, Tr(6,Empty,Empty), Tr(7,Empty,Empty)),
            Empty))
```

Alcune funzioni di base: predicati e selettori

is_empty: **'a tree -> bool**

```
let is_empty = function
    Empty -> true
  |  _ -> false
```

root: **'a tree -> 'a**

```
exception EmptyTree

let root = function
    Empty -> raise EmptyTree
  | Tr(x,_,_) -> x
```

Alcune funzioni di base: predicati e selettori

is_empty: **'a tree -> bool**

```
let is_empty = function
    Empty -> true
  |  _ -> false
```

root: **'a tree -> 'a**

```
exception EmptyTree

let root = function
    Empty -> raise EmptyTree
  | Tr(x,_,_) -> x
```

is_leaf: **'a tree -> bool**

```
let is_leaf = function
    Tr(_,Empty,Empty) ->
                        true
  | _ -> false
```

leaf: **'a -> 'a tree** (utility)

```
let leaf x =
    Tr(x,Empty,Empty)
```

Alcune funzioni di base: predicati e selettori

is_empty: **'a tree -> bool**

```
let is_empty = function
    Empty -> true
  |  _ -> false
```

root: **'a tree -> 'a**

```
exception EmptyTree

let root = function
    Empty -> raise EmptyTree
  | Tr(x,_,_) -> x
```

is_leaf: **'a tree -> bool**

```
let is_leaf = function
    Tr(_,Empty,Empty) ->
                          true
  | _ -> false
```

leaf: **'a -> 'a tree** (utility)

```
let leaf x =
    Tr(x,Empty,Empty)
```

left e **right**: **'a tree -> 'a tree**

```
let left = function
    Empty ->
         raise EmptyTree
  | Tr(_,t,_) -> t
```

```
let right = function
    Empty ->
         raise EmptyTree
  | Tr(_,_,t) -> t
```

Dimensione di un albero binario

size : **'a tree -> int**

Usando il pattern matching:

```
let rec size = function
    Empty -> 0
  | Tr(_,t1,t2) -> 1 + size t1 + size t2;;
```

Usando predicati e selettori:

```
let rec size t =
  if is_empty t then 0
  else 1 + size (left t) + size (right t);;
```

Vai al codice

Visita di un albero con risultato parziale

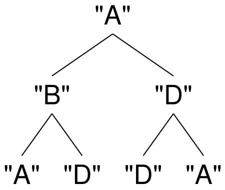
Problema: contare quante volte occorre in un albero ciascuna etichetta.

count : **'a tree -> ('a * int) list**

count t = lista di coppie contenente, per ogni etichetta **x** di qualche nodo dell'albero, una (unica) coppia **(x,n)**, dove **n** è il numero di nodi etichettati da **x**.

Algoritmo 1:

- **count Empty = [ ]** (e questo è facile)
- **count (Tr(x,left,right))**: per ipotesi della ricorsione sappiamo calcolare **count left** e **count right**.



count left=[("B",1);("A",1);("D",1)],
count right=[("D",2);("A",1)]

vanno "fuse" in **[("B",1);("A",2);("D",3)]**,

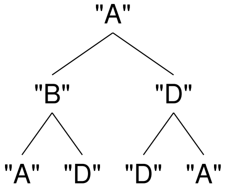
e poi si deve "aggiungere" la radice, ottenendo **[("B",1);("A",3);("D",3)]**

Algoritmo alternativo

Algoritmo 2: visitare l'albero costruendo via via la lista di coppie. Durante la visita si dispone dunque di un risultato parziale: la lista che rappresenta il risultato del procedimento per i nodi già visitati.

Visitare un nodo **a** significa scandire il risultato parziale, aggiungendo 1 al secondo elemento della coppia **(a,n)**, se una tale coppia esiste, o altrimenti, se non esiste, aggiungendo al risultato parziale la coppia **(a,1)**.

Risultato parziale **result**, inizializzato a **[ ]**
Visita in preordine:

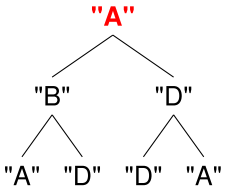


result=[ ]

Algoritmo 2: visitare l'albero costruendo via via la lista di coppie. Durante la visita si dispone dunque di un risultato parziale: la lista che rappresenta il risultato del procedimento per i nodi già visitati.

Visitare un nodo **a** significa scandire il risultato parziale, aggiungendo 1 al secondo elemento della coppia **(a,n)**, se una tale coppia esiste, o altrimenti, se non esiste, aggiungendo al risultato parziale la coppia **(a,1)**.

Risultato parziale **result**, inizializzato a **[ ]**
Visita in preordine:



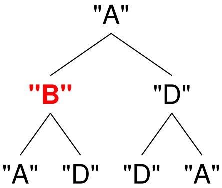
result=[ ]
result=[("A",1)]

Algoritmo alternativo

Algoritmo 2: visitare l'albero costruendo via via la lista di coppie. Durante la visita si dispone dunque di un risultato parziale: la lista che rappresenta il risultato del procedimento per i nodi già visitati.

Visitare un nodo **a** significa scandire il risultato parziale, aggiungendo 1 al secondo elemento della coppia **(a,n)**, se una tale coppia esiste, o altrimenti, se non esiste, aggiungendo al risultato parziale la coppia **(a,1)**.

Risultato parziale **result**, inizializzato a **[ ]**
Visita in preordine:



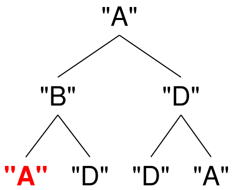
result=[ ]
result=[("A",1)]
result=[("A",1);("B",1)]

Algoritmo alternativo

Algoritmo 2: visitare l'albero costruendo via via la lista di coppie. Durante la visita si dispone dunque di un risultato parziale: la lista che rappresenta il risultato del procedimento per i nodi già visitati.

Visitare un nodo **a** significa scandire il risultato parziale, aggiungendo 1 al secondo elemento della coppia **(a,n)**, se una tale coppia esiste, o altrimenti, se non esiste, aggiungendo al risultato parziale la coppia **(a,1)**.

Risultato parziale **result**, inizializzato a **[ ]**
Visita in preordine:



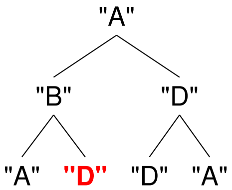
result=[ ]
result=[("A",1)]
result=[("A",1);("B",1)]
result=[("A",2);("B",1)]

Algoritmo alternativo

Algoritmo 2: visitare l'albero costruendo via via la lista di coppie. Durante la visita si dispone dunque di un risultato parziale: la lista che rappresenta il risultato del procedimento per i nodi già visitati.

Visitare un nodo **a** significa scandire il risultato parziale, aggiungendo 1 al secondo elemento della coppia **(a,n)**, se una tale coppia esiste, o altrimenti, se non esiste, aggiungendo al risultato parziale la coppia **(a,1)**.

Risultato parziale **result**, inizializzato a **[ ]**
Visita in preordine:



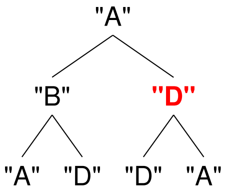
result=[ ]
result=[("A",1)]
result=[("A",1);("B",1)]
result=[("A",2);("B",1)]
result=[("A",2);("B",1);("D",1)]

Algoritmo alternativo

Algoritmo 2: visitare l'albero costruendo via via la lista di coppie. Durante la visita si dispone dunque di un risultato parziale: la lista che rappresenta il risultato del procedimento per i nodi già visitati.

Visitare un nodo **a** significa scandire il risultato parziale, aggiungendo 1 al secondo elemento della coppia **(a,n)**, se una tale coppia esiste, o altrimenti, se non esiste, aggiungendo al risultato parziale la coppia **(a,1)**.

Risultato parziale **result**, inizializzato a **[ ]**
Visita in preordine:



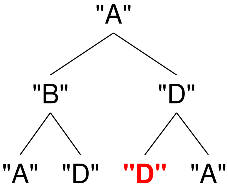
result=[ ]
result=[("A",1)]
result=[("A",1);("B",1)]
result=[("A",2);("B",1)]
result=[("A",2);("B",1);("D",1)]
result=[("A",2);("B",1);("D",2)]

Algoritmo alternativo

Algoritmo 2: visitare l'albero costruendo via via la lista di coppie. Durante la visita si dispone dunque di un risultato parziale: la lista che rappresenta il risultato del procedimento per i nodi già visitati.

Visitare un nodo **a** significa scandire il risultato parziale, aggiungendo 1 al secondo elemento della coppia **(a,n)**, se una tale coppia esiste, o altrimenti, se non esiste, aggiungendo al risultato parziale la coppia **(a,1)**.

Risultato parziale **result**, inizializzato a **[ ]**
Visita in preordine:



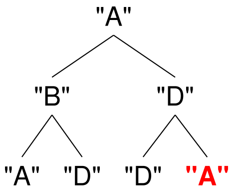
result=[ ]
result=[("A",1)]
result=[("A",1);("B",1)]
result=[("A",2);("B",1)]
result=[("A",2);("B",1);("D",1)]
result=[("A",2);("B",1);("D",2)]
result=[("A",2);("B",1);("D",3)]

Algoritmo alternativo

Algoritmo 2: visitare l'albero costruendo via via la lista di coppie. Durante la visita si dispone dunque di un risultato parziale: la lista che rappresenta il risultato del procedimento per i nodi già visitati.

Visitare un nodo **a** significa scandire il risultato parziale, aggiungendo 1 al secondo elemento della coppia **(a,n)**, se una tale coppia esiste, o altrimenti, se non esiste, aggiungendo al risultato parziale la coppia **(a,1)**.

Risultato parziale **result**, inizializzato a **[ ]**
Visita in preordine:



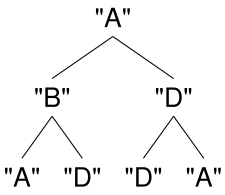
result=[ ]
result=[("A",1)]
result=[("A",1);("B",1)]
result=[("A",2);("B",1)]
result=[("A",2);("B",1);("D",1)]
result=[("A",2);("B",1);("D",2)]
result=[("A",2);("B",1);("D",3)]
result=[("A",3);("B",1);("D",2)]

Algoritmo alternativo

Algoritmo 2: visitare l'albero costruendo via via la lista di coppie. Durante la visita si dispone dunque di un risultato parziale: la lista che rappresenta il risultato del procedimento per i nodi già visitati.

Visitare un nodo **a** significa scandire il risultato parziale, aggiungendo 1 al secondo elemento della coppia **(a,n)**, se una tale coppia esiste, o altrimenti, se non esiste, aggiungendo al risultato parziale la coppia **(a,1)**.

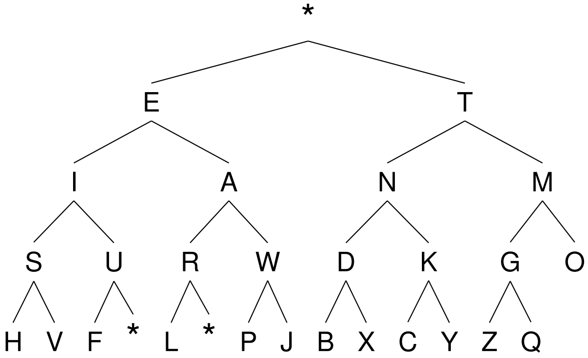
Risultato parziale **result**, inizializzato a **[ ]**
Visita in preordine:



Vai al codice

result=[ ]
result=[("A",1)]
result=[("A",1);("B",1)]
result=[("A",2);("B",1)]
result=[("A",2);("B",1);("D",1)]
result=[("A",2);("B",1);("D",2)]
result=[("A",2);("B",1);("D",3)]
result=[("A",3);("B",1);("D",2)]

L'albero del codice morse (caratteri alfabetici)



Vai al codice