

# Linguaggi di Programmazione I – Lezione 17

Proff. Piero Bonatti e Marco Faella

<mailto://pab@unina.it>

<mailto://marfaella@gmail.com>

6 maggio 2025



**Paradigma funzionale**

**ML**

**Dichiarazioni e scoping in ML**

**Tipi strutturati in ML**

**Patterns e matching**

**Liste**

**Currying**

**Funzioni di ordine superiore**

**Polimorfismo parametrico**

**Encapsulation e interfacce**

**Eccezioni e integrazione con type checking**

**Esempio: un semplice compilatore**



## Paradigma funzionale

Pensare funzionale

Introduzione

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

# Paradigma funzionale



# L'essenza del paradigma funzionale

[Paradigma funzionale](#)

[Pensare funzionale](#)

[Introduzione](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Programmare in stile funzionale puro significa usare *solo espressioni e funzioni*, eventualmente ricorsive



# L'essenza del paradigma funzionale

[Paradigma funzionale](#)

[Pensare funzionale](#)

[Introduzione](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Programmare in stile funzionale puro significa usare *solo espressioni e funzioni*, eventualmente ricorsive
- Non vi sono assegnamenti, non c'è una memoria che cambia
  - ◆ perché gli environment mappano gli identificatori direttamente sul loro valore (immutabile) invece di una locazione di memoria (il cui contenuto può cambiare)



# L'essenza del paradigma funzionale

[Paradigma funzionale](#)

[Pensare funzionale](#)

[Introduzione](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Programmare in stile funzionale puro significa usare *solo espressioni e funzioni*, eventualmente ricorsive
- Non vi sono assegnamenti, non c'è una memoria che cambia
  - ◆ perché gli environment mappano gli identificatori direttamente sul loro valore (immutabile) invece di una locazione di memoria (il cui contenuto può cambiare)
- Quindi, senza assegnamenti, non ci possono essere cicli while/for



# L'essenza del paradigma funzionale

Paradigma funzionale

Pensare funzionale

Introduzione

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Programmare in stile funzionale puro significa usare *solo espressioni e funzioni*, eventualmente ricorsive
- Non vi sono assegnamenti, non c'è una memoria che cambia
  - ◆ perché gli environment mappano gli identificatori direttamente sul loro valore (immutabile) invece di una locazione di memoria (il cui contenuto può cambiare)
- Quindi, senza assegnamenti, non ci possono essere cicli while/for
- Conseguenze sulla programmazione:
  - ◆ ricorsione al posto dei cicli
  - ◆ modifiche all'ambiente anziché alla memoria:
  - ◆ creazione identificatori con stesso nome mediante chiamate ricorsive
  - ◆ creazione di nuovi identificatori con lo stesso nome che mascherano la versione precedente (come nello scoping statico)



# Introduzione

- In questo corso illustreremo brevemente i linguaggi funzionali:
  - ◆ ML (Meta Language, nella versione *Standard ML of New Jersey*, [www.smlnj.org](http://www.smlnj.org))
  - ◆ Altri linguaggi funzionali: OCaml, F#; Lisp, Scheme, Haskell, Scala
- I linguaggi OO più diffusi stanno acquistando caratteristiche funzionali (prossime slide)
- Testi di riferimento
  - ◆ Capitolo 13 di: Gabbrielli e Martini, *Linguaggi di Programmazione* (2a ed.)
  - ◆ Riccardo Pucella, *Notes on Programming Standard ML of New Jersey*.  
<https://www.cs.cornell.edu/riccardo/prog-smlnj/notes-011001.pdf>





# Costrutti funzionali in Java

[Paradigma funzionale](#)

[Pensare funzionale](#)

[Introduzione](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Lambda espressioni (funzioni anonime) [Java 8]
- Collezioni funzionali (*stream*) [Java 8]
- Type inference per metodi parametrici [Java 5]
- Type inference per variabili locali (keyword `var`) [Java 10]
- Pattern matching su `instanceof` [Java 16]
- Pattern matching su `switch` [Java 21]
- Classi immutabili (record) [Java 16]



# Costrutti funzionali in C++

[Paradigma funzionale](#)

[Pensare funzionale](#)

[Introduzione](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Lambda espressioni (funzioni anonime) [C++11]
- Type inference (keyword auto) [C++11]



Paradigma funzionale

**ML**

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in  
ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione  
con type checking

Esempio: un semplice  
compilatore

**ML**



# Il sistema di tipi

Paradigma funzionale

ML

**Il sistema di tipi**

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML è *fortemente e staticamente* tipato
  - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione



## Il sistema di tipi

Paradigma funzionale

ML

**Il sistema di tipi**

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML è *fortemente e staticamente* tipato
  - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione
- Ma non richiede di dichiarare il tipo degli identificatori
  - ◆ spesso lo capisce da solo (*type inference*)



## Il sistema di tipi

Paradigma funzionale

ML

**Il sistema di tipi**

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML è *fortemente e staticamente* tipato
  - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione
- Ma non richiede di dichiarare il tipo degli identificatori
  - ◆ spesso lo capisce da solo (*type inference*)
- Usa sia *structural equivalence* sia *name equivalence*



## Il sistema di tipi

Paradigma funzionale

ML

**Il sistema di tipi**

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML è *fortemente e staticamente* tipato
  - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione
- Ma non richiede di dichiarare il tipo degli identificatori
  - ◆ spesso lo capisce da solo (*type inference*)
- Usa sia *structural equivalence* sia *name equivalence*
- Permette di definire *tipi ricorsivi* (liste, alberi, ...)



## Il sistema di tipi

Paradigma funzionale

ML

**Il sistema di tipi**

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML è *fortemente e staticamente* tipato
  - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione
- Ma non richiede di dichiarare il tipo degli identificatori
  - ◆ spesso lo capisce da solo (*type inference*)
- Usa sia *structural equivalence* sia *name equivalence*
- Permette di definire *tipi ricorsivi* (liste, alberi, ...)
- Supporta *polimorfismo parametrico* (come i template)





## Il sistema di tipi

Paradigma funzionale

ML

**Il sistema di tipi**

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML è *fortemente e staticamente* tipato
  - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione
- Ma non richiede di dichiarare il tipo degli identificatori
  - ◆ spesso lo capisce da solo (*type inference*)
- Usa sia *structural equivalence* sia *name equivalence*
- Permette di definire *tipi ricorsivi* (liste, alberi, ...)
- Supporta *polimorfismo parametrico* (come i template)
- Supporta *encapsulation* (tipi di dato astratti) ma non è un linguaggio a oggetti
  - ◆ mancano la gerarchia di tipi e – di conseguenza – l'ereditarietà



## Il sistema di tipi

[Paradigma funzionale](#)

[ML](#)

[Il sistema di tipi](#)

[Implementazioni](#)

[I tipi primitivi](#)

[Usare l'interprete](#)

[Ancora tipi primitivi](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- ML è *fortemente e staticamente* tipato
  - ◆ Il controllo dei tipi avviene interamente a tempo di compilazione
- Ma non richiede di dichiarare il tipo degli identificatori
  - ◆ spesso lo capisce da solo (*type inference*)
- Usa sia *structural equivalence* sia *name equivalence*
- Permette di definire *tipi ricorsivi* (liste, alberi, ...)
- Supporta *polimorfismo parametrico* (come i template)
- Supporta *encapsulation* (tipi di dato astratti) ma non è un linguaggio a oggetti
  - ◆ mancano la gerarchia di tipi e – di conseguenza – l'ereditarietà
- Il linguaggio OCaml supporta anche gerarchie di tipi ed ereditarietà (è object-oriented)



# Implementazioni di ML

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

ML può essere usato in 2 modi:

1. Interagendo con l'interprete (ad es. **Standard ML of New Jersey**)

- inserendo definizioni ed espressioni una per una
- l'interprete risponde ad ogni passo
- si possono caricare programmi da file digitando nella shell dell'interprete il comando

```
use "nome del file";
```

questo rende utilizzabili le dichiarazioni contenute nel file



# Implementazioni di ML

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

ML può essere usato in 2 modi:

1. Interagendo con l'interprete (ad es. **Standard ML of New Jersey**)

- inserendo definizioni ed espressioni una per una
- l'interprete risponde ad ogni passo
- si possono caricare programmi da file digitando nella shell dell'interprete il comando

```
use "nome del file";
```

questo rende utilizzabili le dichiarazioni contenute nel file

2. Compilando un programma in codice oggetto direttamente eseguibile

- ad es. mediante il compilatore **mlton** per standard ML

```
mlton "nome del file.sml"
```

questo comando produce un file eseguibile con lo stesso nome (ma senza l'estensione .sml)



# I tipi primitivi (I)

[Paradigma funzionale](#)

[ML](#)

[Il sistema di tipi](#)

[Implementazioni](#)

[I tipi primitivi](#)

[Usare l'interprete](#)

[Ancora tipi primitivi](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Detti anche *base types*

## ■ **int**: gli interi

0, 1, ~1, 2, ~2, ...

0xff, ~0x32, ...

Notare che si usa ~ invece del segno meno (-). Alcuni operatori:

+, -, \*, **div**, mod, =, <, ...



# I tipi primitivi (I)

[Paradigma funzionale](#)

[ML](#)

[Il sistema di tipi](#)

[Implementazioni](#)

[I tipi primitivi](#)

[Usare l'interprete](#)

[Ancora tipi primitivi](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Detti anche *base types*

## ■ **int**: gli interi

0, 1, ~1, 2, ~2, ...      0xff, ~0x32, ...

Notare che si usa ~ invece del segno meno (-). Alcuni operatori:

+, -, \*, **div**, mod, =, <, ...

## ■ **real**

3.14, 2.0, 0.1E6, ...

Alcuni operatori su **real**:

+, -, \*, /, <, ...



# I tipi primitivi (II)

[Paradigma funzionale](#)

[ML](#)

[Il sistema di tipi](#)

[Implementazioni](#)

[I tipi primitivi](#)

[Usare l'interprete](#)

[Ancora tipi primitivi](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ string

```
"abc", "123", ...
```

Alcuni operatori su string:

```
^, size, =, <, ...
```



# I tipi primitivi (II)

[Paradigma funzionale](#)

[ML](#)

[Il sistema di tipi](#)

[Implementazioni](#)

[I tipi primitivi](#)

[Usare l'interprete](#)

[Ancora tipi primitivi](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ string

```
"abc", "123", ...
```

Alcuni operatori su string:

```
^, size, =, <, ...
```

## ■ char

```
#"a", #"\\n", #"\\163", ...
```

Alcuni operatori su char:

```
ord, chr, =, <, ...
```





# I tipi primitivi (II)

[Paradigma funzionale](#)

[ML](#)

[Il sistema di tipi](#)

[Implementazioni](#)

[I tipi primitivi](#)

[Usare l'interprete](#)

[Ancora tipi primitivi](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ **string**

```
"abc", "123", ...
```

Alcuni operatori su `string`:

```
^, size, =, <, ...
```

## ■ **char**

```
#"a", #"\\n", #"\\163", ...
```

Alcuni operatori su `char`:

```
ord, chr, =, <, ...
```

## ■ **bool**

```
true, false
```

Alcuni operatori su `bool`:

```
not, andalso, orelse, =
```



# Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

## ■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
-
```



# Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

**Usare l'interprete**

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

## ■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
- 3;
```



# Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

**Usare l'interprete**

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

## ■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
- 3;
val it = 3 : int
```



# Interazione con l'interprete

[Paradigma funzionale](#)

[ML](#)

[Il sistema di tipi](#)

[Implementazioni](#)

[I tipi primitivi](#)

[Usare l'interprete](#)

[Ancora tipi primitivi](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
- 3;
val it = 3 : int

- "Hallo_" ^ "world";
val it = "Hallo_world" : string
```



# Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

**Usare l'interprete**

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

## ■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
- 3;
val it = 3 : int

- "Hallo_" ^ "world";
val it = "Hallo_world" : string

- ord #"a"; ord #"b";
val it = 97 : int
val it = 98 : int
```



# Interazione con l'interprete

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

**Usare l'interprete**

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

## ■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
- 3;
val it = 3 : int

- "Hallo_" ^ "world";
val it = "Hallo_world" : string

- ord #"a"; ord #"b";
val it = 97 : int
val it = 98 : int

- 3 + 2.2;
Error: operator and operand don't agree [overload conflict]
```



# Interazione con l'interprete

[Paradigma funzionale](#)

[ML](#)

[Il sistema di tipi](#)

[Implementazioni](#)

[I tipi primitivi](#)

[Usare l'interprete](#)

[Ancora tipi primitivi](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ Esempi di interazioni con l'interprete:

```
$ sml
Standard ML of New Jersey v110.79
- 3;
val it = 3 : int

- "Hallo_" ^ "world";
val it = "Hallo_world" : string

- ord #"a"; ord #"b";
val it = 97 : int
val it = 98 : int

- 3 + 2.2;
Error: operator and operand don't agree [overload conflict]

- real(3) + 2.2;
val it = 5.2 : real
```

## ■ 'it' si riferisce all'espressione data; calcola sia *valore* che *tipo*





## I tipi primitivi (III)

[Paradigma funzionale](#)

[ML](#)

[Il sistema di tipi](#)

[Implementazioni](#)

[I tipi primitivi](#)

[Usare l'interprete](#)

[Ancora tipi primitivi](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- *Nessuna conversione automatica tra tipi numerici!* Usare `real:int->real` e *basis library*

```
- val r = 3.0 + 2;  
Error: operator and operand don't agree
```

```
- val r = 3.0 + (real 2);  
val r = 5.0 : real
```

```
- val i = 1 + r;  
Error: operator and operand don't agree
```

```
- val i = 1 + (Real.round r);  
val i = 6 : int
```

- C'è una *basis library* per ogni tipo primitivo (`Int`, `Real`...) con funzioni per conversioni, parsing, e altre utilità



## I tipi primitivi (IV)

[Paradigma funzionale](#)

[ML](#)

[Il sistema di tipi](#)

[Implementazioni](#)

[I tipi primitivi](#)

[Usare l'interprete](#)

[Ancora tipi primitivi](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

### ■ *Real non supporta l'uguaglianza!* Usare `Real.==`

```
- val x = 1.0; val y = 2.0;
val x = 1.0 : real
val y = 2.0 : real
- x = y;
Error:  operator and operand don't agree [equality type required]

- Real.==(x,y);
val it = false : bool
```

# I tipi primitivi (IV)

Paradigma funzionale

ML

Il sistema di tipi

Implementazioni

I tipi primitivi

Usare l'interprete

Ancora tipi primitivi

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- *Real non supporta l'uguaglianza!* Usare `Real.==`

```
- val x = 1.0; val y = 2.0;
val x = 1.0 : real
val y = 2.0 : real
- x = y;
Error: operator and operand don't agree [equality type required]

- Real.==(x,y);
val it = false : bool
```

- Questo perchè lo standard IEEE prevede valori che risultano da operazioni non definite, denominati **NaN** (not a number)

- ◆ Un NaN non è uguale a nessun altro numero, nemmeno a sé stesso

```
- val e = Math.sqrt(~2.0);
val e = nan : real

- Real.==(e,e);
val it = false : bool
```

- Nota: anche in Java `Double.NaN` risulta diverso da se stesso



Paradigma funzionale

ML

**Dichiarazioni e scoping in ML**

Funzioni

Annotazioni di tipo

Una funzione ricorsiva

Altre dichiarazioni di  
identificatori

Scoping

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione  
con type checking

Esempio: un semplice  
compilatore

# Dichiarazioni e scoping in ML



# Funzioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)

[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Ci sono diversi modi di definire e chiamare funzioni in ML. Iniziamo con i più tradizionali

```
- fun quadr x = x * x;  
val quadr = fn : int -> int
```



# Funzioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)  
[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Ci sono diversi modi di definire e chiamare funzioni in ML. Iniziamo con i più tradizionali

```
- fun quadr x = x * x;  
val quadr = fn : int -> int
```

```
- quadr(3);  
val it = 9 : int
```

# Funzioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)  
[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Ci sono diversi modi di definire e chiamare funzioni in ML. Iniziamo con i più tradizionali

```
- fun quadr x = x * x;  
val quadr = fn : int -> int
```

```
- quadr(3);  
val it = 9 : int
```

```
- quadr 3;                                (* in questo caso le parentesi sono opzionali *)  
val it = 9 : int
```



# Funzioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)

[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Ci sono diversi modi di definire e chiamare funzioni in ML. Iniziamo con i più tradizionali

```
- fun quadr x = x * x;  
val quadr = fn : int -> int
```

```
- quadr 3;  
val it = 9 : int
```

```
- quadr 3;                                     (* in questo caso le parentesi sono opzionali *)  
val it = 9 : int
```

- Con **fun** si *dichiara* la funzione

- ◆ fun aggiunge all'ambiente l'identificatore quadr
- ◆ e lo associa alla funzione da interi a interi
- ◆ Nota: in caso di ambiguità tra int e real, la type inference sceglie int



- Ci sono diversi modi di definire e chiamare funzioni in ML. Iniziamo con i più tradizionali

```
- fun quadr x = x * x;  
val quadr = fn : int -> int
```

```
- quadr(3);  
val it = 9 : int
```

```
- quadr 3;                                     (* in questo caso le parentesi sono opzionali *)  
val it = 9 : int
```

- Con **fun** si *dichiara* la funzione

- ◆ fun aggiunge all'ambiente l'identificatore quadr
- ◆ e lo associa alla funzione da interi a interi
- ◆ Nota: in caso di ambiguità tra int e real, la type inference sceglie int

- Si può vedere cosa è associato a quadr *senza chiamare la funzione*

```
- quadr;                                       (* nome della funzione senza argomenti *)  
val it = fn : int -> int
```

Mostra solo il tipo (il valore è stato trasformato in bytecode)



# Annotazioni di tipo

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)

[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

É possibile indicare esplicitamente i tipi:

```
- fun quadr (x :int) : int = x * x;  
val quadr = fn : int -> int
```

In assenza di annotazioni di tipo, scatta la *type inference*



# Una funzione ricorsiva

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)

[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
- fun fatt x = if x=0 then 1 else x*fatt(x-1);  
val fatt = fn : int -> int
```



# Una funzione ricorsiva

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)

[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
- fun fatt x = if x=0 then 1 else x*fatt(x-1);  
val fatt = fn : int -> int  
  
- fatt(3);  
val it = 6 : int
```



# Una funzione ricorsiva

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)

[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
- fun fatt x = if x=0 then 1 else x*fatt(x-1);  
val fatt = fn : int -> int
```

```
- fatt(3);  
val it = 6 : int
```

```
- fatt 3;                                (* in questo caso le parentesi sono opzionali *)  
val it = 6 : int
```

- In ML il costrutto `if-then-else` denota un'espressione
- In Java/C/C++/etc., denota uno *statement* (non ha un valore)
- Quindi, `if-then-else` in ML è simile all'operatore ternario `?:` in Java/C/C++



# Altre dichiarazioni di identificatori

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)

[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Con **val** si aggiunge un nuovo identificatore all'ambiente e gli si associa un valore

```
- val x = 2+2;  
val x = 4 : int
```

```
- x+2;  
val it = 6 : int
```



# Altre dichiarazioni di identificatori

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)

[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Con **val** si aggiunge un nuovo identificatore all'ambiente e gli si associa un valore

```
- val x = 2+2;  
val x = 4 : int  
  
- x+2;  
val it = 6 : int
```

- Grammatica delle dichiarazioni viste sinora

```
<declaration> ::=  
    val <id name> = <expression> |  
    fun <func name> <argument>* = <expression>
```

Vedremo più avanti che `val` è più generale di `fun`



# Scoping

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)  
[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ L'equivalente dei blocchi in ML è

```
let      <dichiarazioni>
in      <espressione>  (* le dichiarazioni valgono solo qui *)
end
```



[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)  
[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ L'equivalente dei blocchi in ML è

```
let      <dichiarazioni>
in      <espressione>  (* le dichiarazioni valgono solo qui *)
end
```

Lo scoping è **statico**. Esempi:

```
- let  val x=2  in  3*x  end;
val it = 6 : int

- x;
Error: unbound variable or constructor: x

- let  val x=2  in
      let  val x=3 in  (* questa def. maschera la precedente *)
        3*x
      end
    end;
val it = 9 : int
```



# Scoping (II)

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)

[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ Ambiente non locale delle funzioni

```
- val x=0;  
val x=0 : int
```



# Scoping (II)

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)

[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ Ambiente non locale delle funzioni

```
- val x=0;
val x=0 : int

- let val x=1 in
      let fun f(y) = x+y in          (* x è non locale *)
        f(0)
      end
    end;
val it = 1 : int
```



## Scoping (II)

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)

[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

### ■ Ambiente non locale delle funzioni

```
- val x=0;
val x=0 : int

- let val x=1 in
      let fun f(y) = x+y in      (* x è non locale *)
        f(0)
      end
    end;
val it = 1 : int
```

### ■ Forma equivalente più concisa

```
let
  val x=1
  fun f(y) = x+y
in
  f(0)
end;
```

dopo “let” possiamo mettere quante dichiarazioni vogliamo



# Scoping (III)

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Funzioni](#)

[Annotazioni di tipo](#)

[Una funzione ricorsiva](#)

[Altre dichiarazioni di identificatori](#)

[Scoping](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## Definizioni ausiliarie

### ■ locali ad altre definizioni

```
local
    <dichiarazioni>
in
    <dichiarazione>  (* le dichiarazioni sopra valgono solo qui *)
end
```

Simile a `let` ma dopo `in` c'è una **dichiarazione** invece di una espressione da valutare



Paradigma funzionale

ML

Dichiarazioni e scoping in ML

**Tipi strutturati in ML**

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

## Tipi strutturati in ML



# Prodotti cartesiani

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

**Prodotti cartesiani**

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Si possono definire  $n$ -uple semplicemente mettendo i valori tra parentesi
- Il prodotto cartesiano viene indicato con ‘\*’
- Si estrae l’ $i$ -esimo elemento da una  $n$ -upla con l’operatore prefisso  $\#i$



# Prodotti cartesiani

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Si possono definire  $n$ -uple semplicemente mettendo i valori tra parentesi
- Il prodotto cartesiano viene indicato con ‘\*’
- Si estrae l’ $i$ -esimo elemento da una  $n$ -upla con l’operatore prefisso  $\#i$

```
- (1+1, "A");  
val it = (2,"A") : int * string
```





# Prodotti cartesiani

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Si possono definire  $n$ -uple semplicemente mettendo i valori tra parentesi
- Il prodotto cartesiano viene indicato con ‘\*’
- Si estrae l' $i$ -esimo elemento da una  $n$ -upla con l'operatore prefisso  $\#i$

```
- (1+1, "A");  
val it = (2,"A") : int * string  
  
- val x = (1,"A",3.5);  
val x = (1,"A",3.5) : int * string * real
```



# Prodotti cartesiani

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Si possono definire  $n$ -uple semplicemente mettendo i valori tra parentesi
- Il prodotto cartesiano viene indicato con ‘\*’
- Si estrae l’ $i$ -esimo elemento da una  $n$ -upla con l’operatore prefisso  $\#i$

```
- (1+1, "A");  
val it = (2,"A") : int * string  
  
- val x = (1,"A",3.5);  
val x = (1,"A",3.5) : int * string * real  
  
- #1(x);  
val it = 1 : int
```



# Prodotti cartesiani

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Si possono definire  $n$ -uple semplicemente mettendo i valori tra parentesi
- Il prodotto cartesiano viene indicato con ‘\*’
- Si estrae l' $i$ -esimo elemento da una  $n$ -upla con l'operatore prefisso  $\#i$

```
- (1+1, "A");  
val it = (2,"A") : int * string  
  
- val x = (1,"A",3.5);  
val x = (1,"A",3.5) : int * string * real  
  
- #1(x);  
val it = 1 : int  
  
- #2(x);  
val it = "A" : string
```



# Prodotti cartesiani

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)  
[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Si possono definire  $n$ -uple semplicemente mettendo i valori tra parentesi
- Il prodotto cartesiano viene indicato con ‘\*’
- Si estrae l' $i$ -esimo elemento da una  $n$ -upla con l'operatore prefisso  $\#i$

```
- (1+1, "A");  
val it = (2,"A") : int * string  
  
- val x = (1,"A",3.5);  
val x = (1,"A",3.5) : int * string * real  
  
- #1(x);  
val it = 1 : int  
  
- #2(x);  
val it = "A" : string  
  
- #3(x);  
val it = 3.5 : real
```



# Record

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Insiemi di espressioni `<nome>=<valore>`. Notate come viene rappresentato il tipo

```
- val r = {nome="Mario",nato=1998};  
val r = {nato=1998, nome="Mario"} : {nato:int, nome:string}
```



# Record

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

**[Record](#)**

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Insiemi di espressioni `<nome>=<valore>`. Notate come viene rappresentato il tipo

```
- val r = {nome="Mario",nato=1998};  
val r = {nato=1998, nome="Mario"} : {nato:int, nome:string}
```

- Il valore associato al nome  $N$  si estrae con  $\#N$

```
- #nome(r);  
val it = "Mario" : string  
  
- #nato(r);  
val it = 1998 : int
```



# Record

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

**Record**

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Insiemi di espressioni  $\langle \text{nome} \rangle = \langle \text{valore} \rangle$ . Notate come viene rappresentato il tipo

```
- val r = {nome="Mario", nato=1998};  
val r = {nato=1998, nome="Mario"} : {nato:int, nome:string}
```

- Il valore associato al nome  $N$  si estrae con  $\#N$

```
- #nome(r);  
val it = "Mario" : string  
  
- #nato(r);  
val it = 1998 : int
```

- L'ordine delle coppie non conta

```
- {nome="Mario", nato=1998} = {nato=1998, nome="Mario"};  
val it = true : bool
```



# Dichiarazioni di tipo

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

**Dichiarazioni di tipo**

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML permette di definire nuovi tipi similmente ai typedef del C

```
- type coord = real * real;  
type coord = real * real
```





# Dichiarazioni di tipo

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML permette di definire nuovi tipi similmente ai typedef del C

```
- type coord = real * real;  
type coord = real * real
```

- Il compilatore va aiutato a stabilire il tipo

```
- val x = (3.0,4.0);                                (* senza aiutino *)  
val x = (3.0,4.0) : real * real  
  
- val x:coord = (3.0,4.0);                          (* con aiutino *)  
val x = (3.0,4.0) : coord
```

- I tipi coord e (real \* real) sono compatibili tra loro (*structural equivalence*)



# Dichiarazioni di tipo

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML permette di definire nuovi tipi similmente ai typedef del C

```
- type coord = real * real;  
type coord = real * real
```

- Il compilatore va aiutato a stabilire il tipo

```
- val x = (3.0,4.0);                                (* senza aiutino *)  
val x = (3.0,4.0) : real * real  
  
- val x:coord = (3.0,4.0);                           (* con aiutino *)  
val x = (3.0,4.0) : coord
```

- I tipi coord e (real \* real) sono compatibili tra loro (*structural equivalence*)
  - ◆ posso passare una espressione di tipo coord a un parametro di tipo (real \* real) e viceversa

# Dichiarazioni di tipo

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML permette di definire nuovi tipi similmente ai typedef del C

```
- type coord = real * real;  
type coord = real * real
```

- Il compilatore va aiutato a stabilire il tipo

```
- val x = (3.0,4.0);                                (* senza aiutino *)  
val x = (3.0,4.0) : real * real  
  
- val x:coord = (3.0,4.0);                          (* con aiutino *)  
val x = (3.0,4.0) : coord
```

- I tipi coord e (real \* real) sono compatibili tra loro (*structural equivalence*)
  - ◆ posso passare una espressione di tipo coord a un parametro di tipo (real \* real) e viceversa
  - ◆ similmente coord è compatibile con ogni altro tipo definito come (real \* real), come ad esempio

```
type coppia = real * real;
```



# Datatypes e costruttori

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Con **datatype** si può fare di più che dare un nome a un tipo ML
  - ◆ si possono definire **costruttori** per creare *data objects*

```
- datatype color = red | green | blue;  
datatype color = blue | green | red
```



# Datatypes e costruttori

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Con **datatype** si può fare di più che dare un nome a un tipo ML
  - ◆ si possono definire **costruttori** per creare *data objects*

```
- datatype color = red | green | blue;  
datatype color = blue | green | red  
  
- val c = red;  
val c = red : color
```

red, green, blue sono costruttori. Definiscono i possibili valori del tipo color



# Datatypes e costruttori

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Con **datatype** si può fare di più che dare un nome a un tipo ML
  - ◆ si possono definire **costruttori** per creare *data objects*

```
- datatype color = red | green | blue;  
datatype color = blue | green | red  
  
- val c = red;  
val c = red : color
```

red, green, blue sono costruttori. Definiscono i possibili valori del tipo color

- Notare la somiglianza con le enum del C. Solo apparente...



# Differenza tra datatypes e enumerazioni C

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

**Datatypes e costruttori**

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- C non prende sul serio le enumerazioni: non sono altro che int

```
enum color { red, green, blue };  
printf("%d%d%d",red,green,blue);    (* stampa 012 *)
```



# Differenza tra datatypes e enumerazioni C

- C non prende sul serio le enumerazioni: non sono altro che int

```
enum color { red, green, blue };  
printf("%d%d%d",red,green,blue);      (* stampa 012 *)  
  
enum color c = red;  
if( c == 0 ) then ... else ...;      (* esegue il then *)
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)





# Differenza tra datatypes e enumerazioni C

- C non prende sul serio le enumerazioni: non sono altro che int

```
enum color { red, green, blue };  
printf("%d%d%d",red,green,blue);      (* stampa 012 *)  
  
enum color c = red;  
if( c == 0 ) then ... else ...;      (* esegue il then *)  
  
c = 10;                               (* nessun errore!!! *)
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

# Differenza tra datatypes e enumerazioni C

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- C non prende sul serio le enumerazioni: non sono altro che int

```
enum color { red, green, blue };  
printf("%d%d%d",red,green,blue);      (* stampa 012 *)  
  
enum color c = red;  
if( c == 0 ) then ... else ...;      (* esegue il then *)  
  
c = 10;                               (* nessun errore!!! *)
```

- Invece i datatypes di ML definiscono tipi genuinamente nuovi: nessuna corrispondenza con gli int

```
- val c:color = 10;  
Error: pattern and expression in val dec don't agree  
  
- c = 0;                               (* c è uguale a 0? *)  
Error: operator and operand don't agree
```

red, green, blue sono oggetti completamente nuovi



# Differenza tra datatypes e enumerazioni C

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

Datatypes e costruttori

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- C non prende sul serio le enumerazioni: non sono altro che int

```
enum color { red, green, blue };  
printf("%d%d%d",red,green,blue);      (* stampa 012 *)  
  
enum color c = red;  
if( c == 0 ) then ... else ...;      (* esegue il then *)  
  
c = 10;                               (* nessun errore!!! *)
```

- Invece i datatypes di ML definiscono tipi genuinamente nuovi: nessuna corrispondenza con gli int

```
- val c:color = 10;  
Error: pattern and expression in val dec don't agree  
  
- c = 0;                               (* c è uguale a 0? *)  
Error: operator and operand don't agree
```

red, green, blue sono oggetti completamente nuovi

- Ogni tipo definito con datatype è incompatibile con *tutti gli altri tipi* (name equivalence)



# Costruttori con argomenti

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

**Datatypes e costruttori**

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Un esempio: definire una lista concatenata di interi

- Se ne può dare una *definizione ricorsiva*: una lista di interi è
  - ◆ la lista vuota (caso base)
  - ◆ un nodo che contiene un intero e una lista di interi (caso induttivo)



# Costruttori con argomenti

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

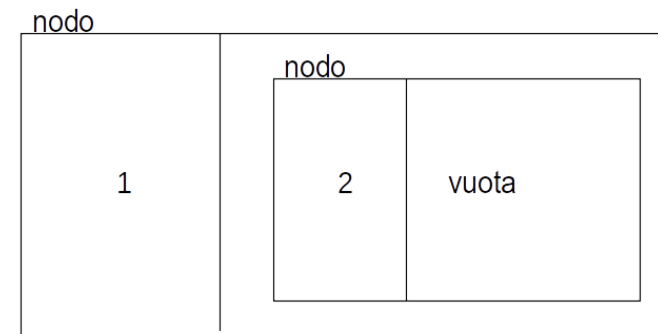
[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Un esempio: definire una lista concatenata di interi

- Se ne può dare una *definizione ricorsiva*: una lista di interi è
  - ◆ la lista vuota (caso base)
  - ◆ un nodo che contiene un intero e una lista di interi (caso induttivo)

Rappresentazione della lista [1,2]





# Costruttori con argomenti

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

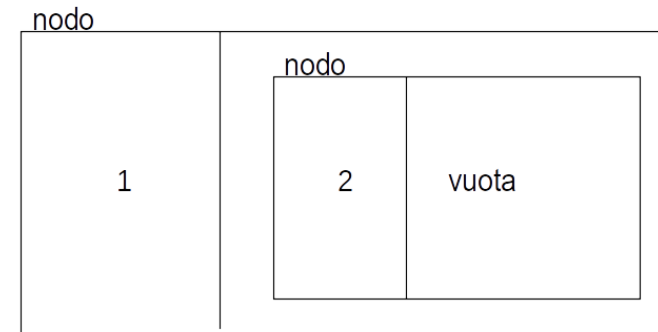
[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Un esempio: definire una lista concatenata di interi

- Se ne può dare una *definizione ricorsiva*: una lista di interi è
  - ◆ la lista vuota (caso base)
  - ◆ un nodo che contiene un intero e una lista di interi (caso induttivo)

Rappresentazione della lista [1,2]



- Quindi servono 2 costruttori: per la lista vuota e per i nodi

```
- datatype listaInt = vuota | nodo of int * listaInt;  
datatype listaInt = nodo of int * listaInt | vuota
```



# Costruttori con argomenti

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

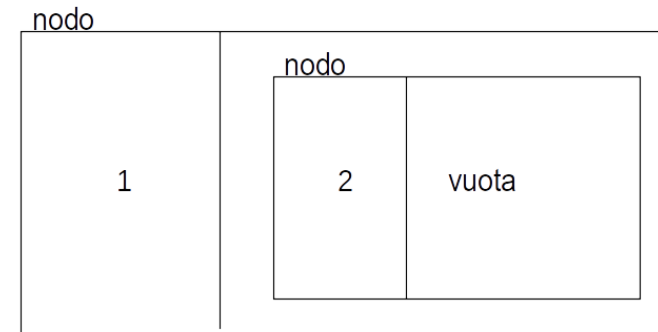
[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Un esempio: definire una lista concatenata di interi

- Se ne può dare una *definizione ricorsiva*: una lista di interi è
  - ◆ la lista vuota (caso base)
  - ◆ un nodo che contiene un intero e una lista di interi (caso induttivo)

Rappresentazione della lista [1,2]



- Quindi servono 2 costruttori: per la lista vuota e per i nodi

```
- datatype listaInt = vuota | nodo of int * listaInt;
datatype listaInt = nodo of int * listaInt | vuota

- val L = nodo(1, nodo(2, vuota));
val L = nodo (1,nodo (2,vuota)) : listaInt
```



## Costruttori con argomenti (II)

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Prodotti cartesiani

Record

Dichiarazioni di tipo

**Datatypes e costruttori**

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Altro esempio: albero binario con nodi etichettati da interi

- Definizione ricorsiva: un albero simile è
  - ◆ un albero vuoto, oppure
  - ◆ un nodo che contiene un intero e due alberi dello stesso tipo





## Costruttori con argomenti (II)

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Altro esempio: albero binario con nodi etichettati da interi

- Definizione ricorsiva: un albero simile è
  - ◆ un albero vuoto, oppure
  - ◆ un nodo che contiene un intero e due alberi dello stesso tipo

```
datatype albero = vuoto | nodoAlb of int * albero * albero
```



## Costruttori con argomenti (II)

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Altro esempio: albero binario con nodi etichettati da interi

- Definizione ricorsiva: un albero simile è
  - ◆ un albero vuoto, oppure
  - ◆ un nodo che contiene un intero e due alberi dello stesso tipo

```
datatype albero = vuoto | nodoAlb of int * albero * albero
```

In questo esempio costruiamo un albero con radice 1, figlio sinistro 2 (che è una foglia), mentre il figlio destro manca.

```
nodoAlb (1,nodoAlb (2,vuoto,vuoto),vuoto)
```



# Tipi mutuamente ricorsivi

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Definire un tipo per un albero binario in cui i nodi di profondità dispari sono etichettati da interi e i nodi di profondità pari da stringhe (supponiamo che la radice abbia profondità 1)

Primo tentativo:

```
datatype albero = vuoto
| nodop of string * albero * albero
| nodod of      int * albero * albero;
```



# Tipi mutuamente ricorsivi

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Definire un tipo per un albero binario in cui i nodi di profondità dispari sono etichettati da interi e i nodi di profondità pari da stringhe (supponiamo che la radice abbia profondità 1)

Primo tentativo:

```
datatype albero = vuoto
| nodop of string * albero * albero
| nodod of      int * albero * albero;
```

Così un nodo dispari può avere figli dispari!



# Tipi mutuamente ricorsivi

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Definire un tipo per un albero binario in cui i nodi di profondità dispari sono etichettati da interi e i nodi di profondità pari da stringhe (supponiamo che la radice abbia profondità 1)

Secondo tentativo:

```
datatype nodopari =  
    nodop of string * nododispari * nododispari | vuotop;  
  
datatype nododispari =  
    nodod of int * nodopari * nodopari | vuotod;  
  
type albero = nododispari;
```



# Tipi mutuamente ricorsivi

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Definire un tipo per un albero binario in cui i nodi di profondità dispari sono etichettati da interi e i nodi di profondità pari da stringhe (supponiamo che la radice abbia profondità 1)

Secondo tentativo:

```
datatype nodopari =  
    nodop of string * nododispari * nododispari | vuotoop;  
Error: unbound type constructor: nododispari  
  
datatype nododispari =  
    nodod of int * nodopari * nodopari | vuotod;  
  
type albero = nododispari;
```



# Tipi mutuamente ricorsivi

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Prodotti cartesiani](#)

[Record](#)

[Dichiarazioni di tipo](#)

[Datatypes e costruttori](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Definire un tipo per un albero binario in cui i nodi di profondità dispari sono etichettati da interi e i nodi di profondità pari da stringhe (supponiamo che la radice abbia profondità 1)

Terzo (e ultimo) tentativo:

```
datatype nodopari =  
    nodop of string * nododispari * nododispari | vuotop  
and nododispari =  
    nodod of int * nodopari * nodopari | vuotod;  
  
type albero = nododispari;
```

“and” si usa anche per definire *funzioni mutuamente ricorsive*

```
fun <funzione1> and <funzione2> and ... ;
```



[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Patterns](#)

[Pattern](#)

[Def. per casi](#)

[Esercizio](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

# Patterns e matching





# Utilizzo dei costruttori con argomenti

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Patterns](#)

[Pattern](#)

[Def. per casi](#)

[Esercizio](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Per scandire una lista abbiamo innanzitutto bisogno di controllare se è vuota

```
- val L = nodo(1, nodo(2, vuota));  
val L = nodo (1,nodo (2,vuota)) : listaInt  
  
- L = vuota;  
val it = false : bool
```



# Utilizzo dei costruttori con argomenti

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Pattern

Def. per casi

Esercizio

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Per scandire una lista abbiamo innanzitutto bisogno di controllare se è vuota

```
- val L = nodo(1, nodo(2, vuota));  
val L = nodo (1,nodo (2,vuota)) : listaInt  
  
- L = vuota;  
val it = false : bool
```

- Se non è vuota potrebbe servirci il primo elemento. Si estrae con *pattern matching*

```
- val nodo(p,_) = L;    (* assegna a p il 1° elemento di L *)  
val p = 1 : int         (* "_" è una wildcard *)
```

La parte in rosso è chiamata *pattern*



# Utilizzo dei costruttori con argomenti

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Pattern

Def. per casi

Esercizio

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Per scandire una lista abbiamo innanzitutto bisogno di controllare se è vuota

```
- val L = nodo(1, nodo(2, vuota));  
val L = nodo (1,nodo (2,vuota)) : listaInt  
  
- L = vuota;  
val it = false : bool
```

- Se non è vuota potrebbe servirci il primo elemento. Si estrae con *pattern matching*

```
- val nodo(p,_) = L;    (* assegna a p il 1° elemento di L *)  
val p = 1 : int         (* "_" è una wildcard *)
```

La parte in rosso è chiamata *pattern*

- Per ottenere il resto della lista

```
- val nodo(_,r) = L;    (* assegna a r il resto *)  
val r = nodo (2,vuota) : listaInt
```



# Funzione che conta gli elementi della lista

- Ovviamente deve essere *ricorsiva* (niente cicli!)

```
- fun conta x =  
    if x = vuota then 0  
    else  
        let val nodo(_, r) = x in  
            conta(r) + 1  
        end;  
val conta = fn : listaInt -> int  
  
- conta L;  
val it = 2 : int
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Patterns](#)

[Pattern](#)

[Def. per casi](#)

[Esercizio](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



# Funzione che conta gli elementi della lista

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Pattern

Def. per casi

Esercizio

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Ovviamente deve essere *ricorsiva* (niente cicli!)

```
- fun conta x =  
    if x = vuota then 0  
    else  
        let val nodo(_, r) = x in  
            conta(r) + 1  
        end;  
val conta = fn : listaInt -> int  
  
- conta L;  
val it = 2 : int
```

- Notare come il compilatore ha *inferito* il tipo della funzione

1. x viene confrontato con “vuota”, che è di tipo listaInt  $\Rightarrow$  anche x è di tipo listaInt  $\Rightarrow$  l’input di “conta” è un listaInt
2. il “then” restituisce 0, che è un intero; quindi l’output di “conta” è un intero



# Funzione che conta gli elementi della lista

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Pattern

Def. per casi

Esercizio

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Ovviamente deve essere *ricorsiva* (niente cicli!)

```
- fun conta x =  
    if x = vuota then 0  
    else  
        let val nodo(_, r) = x in  
            conta(r) + 1  
        end;  
val conta = fn : listaInt -> int  
  
- conta L;  
val it = 2 : int
```

- Inoltre il compilatore controlla che anche il resto della funzione sia compatibile con questi tipi

1.  $r$  corrisponde al 2° argomento del nodo, che è di tipo `listaInt`  $\Rightarrow$  è corretto passarlo a `conta` che restituisce un intero
2. quindi anche l'`else` restituisce un intero, e tutto torna



# Pattern

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Patterns](#)

**[Pattern](#)**

[Def. per casi](#)

[Esercizio](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ I pattern possono coinvolgere datatype, tuple, record

```
- val a = { nome="Mario", nascita=2002 };  
val a = {nascita=2002,nome="Mario"} : {nascita:int, nome:string}  
- val { nome=_, nascita=anno } = a;  
val anno = 2002 : int
```



# Pattern

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Patterns](#)

**[Pattern](#)**

[Def. per casi](#)

[Esercizio](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ I pattern possono coinvolgere datatype, tuple, record

```
- val a = { nome="Mario", nascita=2002 };  
val a = {nascita=2002,nome="Mario"} : {nascita:int, nome:string}  
- val { nome=_, nascita=anno } = a;  
val anno = 2002 : int
```

## ■ Un pattern può contenere più variabili

```
- val nodo(p, r) = L;  
val p = 1 : int  
val r = nodo (2,vuota) : listaInt
```

## ■ Cioè dichiara due identificatori (p e r) in un colpo solo





# Pattern

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

**Pattern**

Def. per casi

Esercizio

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

## ■ I pattern possono coinvolgere datatype, tuple, record

```
- val a = { nome="Mario", nascita=2002 };  
val a = {nascita=2002,nome="Mario"} : {nascita:int, nome:string}  
- val { nome=_, nascita=anno } = a;  
val anno = 2002 : int
```

## ■ Un pattern può contenere più variabili

```
- val nodo(p, r) = L;  
val p = 1 : int  
val r = nodo (2,vuota) : listaInt
```

## ■ Cioè dichiara due identificatori (p e r) in un colpo solo

## ■ Un pattern non può contenere la stessa variabile due volte

```
- val (x,x) = (2,2);  
stdIn:1.2-1.19 Error: duplicate variable in pattern(s): x
```



# Definire funzioni per casi

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Pattern

**Def. per casi**

Esercizio

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Si può *definire una funzione per casi*

```
- fun conta(vuota)      = 0
  | conta(nodo(_, r)) = conta(r) + 1;
```

mettendo direttamente i pattern al posto dei parametri formali

- L'ordine dei casi è rilevante
- Notare l'eleganza e la concisione

# Esercizio

Scrivere una funzione che confronta due alberi binari con valori interi e restituisce vero se sono identici

```
fun alberi_uguali empty empty = true
  | alberi_uguali (btnode(v1,l1,r1)) (btnode(v2,l2,r2)) =
    v1=v2 andalso
    alberi_uguali l1 l2 andalso
    alberi_uguali r1 r2
  | alberi_uguali _ _ = false;
```

Errori comuni:

```
fun alberi_uguali empty empty = true
  | alberi_uguali (btnode(v,l1,r1)) (btnode(v,l2,r2)) =
    alberi_uguali l1 l2 andalso alberi_uguali2 r1 r2
  | ...
```

Error: duplicate variable in pattern(s): v

```
fun alberi_uguali empty empty = true
  | alberi_uguali (btnode(v1,l1,r1)) (btnode(v2,l2,r2)) =
    v1=v2 andalso
    alberi_uguali l1 l2 andalso
    alberi_uguali2 r1 r2;
```

Warning: match nonexhaustive

(empty,empty) => ...

(btnode (v1,l1,r1),btnode (v2,l2,r2)) => ...

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Patterns

Pattern

Def. per casi

Esercizio

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



# Costrutto case-of

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Patterns](#)

[Pattern](#)

[Def. per casi](#)

[Esercizio](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- L'idea è la stessa della definizione per casi delle funzioni

```
- case L of vuota => true  
          | nodo(_, _) => false;
```

```
val it = false : bool
```

- Simile al costrutto *switch* di C/Java
- L'ordine dei casi è rilevante
- Il costrutto case-of crea un'espressione



# Esercizi

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Patterns](#)

[Pattern](#)

[Def. per casi](#)

[Esercizio](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

1. Scrivere in ML una funzione `size` che restituisce il *numero di nodi* in un albero binario come quello visto in precedenza
2. Scrivere in ML una funzione che, dato un albero binario *A* come quello visto in precedenza, e un intero *N*, restituisce l'etichetta del nodo che si raggiunge in *N* passi visitando l'albero in preordine
3. Simile all'esercizio 2, ma visitando l'albero in postordine (difficile)



[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Le liste in ML](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

# Liste



# Le liste in ML

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

**Le liste in ML**

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le liste sono tra le strutture dati più usate in programmazione funzionale
  - ◆ in qualche misura sostituiscono i vettori (concetto essenzialmente imperativo)



# Le liste in ML

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Le liste in ML](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Le liste sono tra le strutture dati più usate in programmazione funzionale
  - ◆ in qualche misura sostituiscono i vettori (concetto essenzialmente imperativo)
- Perciò ML le fornisce built-in con i costruttori **nil** e **::**

```
nil                (* lista vuota *)  
1 :: 2 :: 3 :: nil (* lista che contiene 1, 2, 3 *)
```





# Le liste in ML

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

**Le liste in ML**

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le liste sono tra le strutture dati più usate in programmazione funzionale
  - ◆ in qualche misura sostituiscono i vettori (concetto essenzialmente imperativo)
- Perciò ML le fornisce built-in con i costruttori **nil** e **::**

```
nil                (* lista vuota *)
1 :: 2 :: 3 :: nil  (* lista che contiene 1, 2, 3 *)

(* formato equivalente basato su parentesi quadre *)
[]
[1,2,3]
```



# Le liste in ML

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le liste sono tra le strutture dati più usate in programmazione funzionale
  - ◆ in qualche misura sostituiscono i vettori (concetto essenzialmente imperativo)
- Perciò ML le fornisce built-in con i costruttori **nil** e **::**

```
nil                (* lista vuota *)
1 :: 2 :: 3 :: nil  (* lista che contiene 1, 2, 3 *)

(* formato equivalente basato su parentesi quadre *)
[]
[1,2,3]

(* sono veramente equivalenti *)
- [] = nil;
val it = true : bool

- 1::2::3::nil = [1,2,3];
val it = true : bool
```



# Principali operatori sulle liste in ML

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Le liste in ML](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La funzione `length` restituisce la lunghezza di una lista



# Principali operatori sulle liste in ML

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Le liste in ML](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La funzione **length** restituisce la lunghezza di una lista
- La funzione **null** restituisce true se la lista è vuota



# Principali operatori sulle liste in ML

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Le liste in ML](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La funzione **length** restituisce la lunghezza di una lista
- La funzione **null** restituisce true se la lista è vuota
- Le funzioni **hd** (*head*) e **tl** (*tail*) restituiscono il primo elemento e il resto della lista, rispettivamente



# Principali operatori sulle liste in ML

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Le liste in ML](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La funzione `length` restituisce la lunghezza di una lista
- La funzione `null` restituisce true se la lista è vuota
- Le funzioni `hd` (*head*) e `tl` (*tail*) restituiscono il primo elemento e il resto della lista, rispettivamente
- L'operatore infisso `@` restituisce la concatenazione di due liste

```
- val L = [1,2,3];  
val L = [1,2,3] : int list  
  
- hd L;  
val it = 1 : int  
  
- tl L;  
val it = [2,3] : int list  
  
- val M = [1,2] @ [3,4];  
val M = [1,2,3,4] : int list
```



# Funzioni aggiuntive su liste

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Le liste in ML](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Altre funzioni si trovano nella *struttura* `List`, ad esempio
  - ◆ `List.nth(L,i)` restituisce l'*i*-esimo elemento di `L` (partendo da 0)



## Funzioni aggiuntive su liste

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

**Le liste in ML**

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Altre funzioni si trovano nella *struttura* `List`, ad esempio
  - ◆ `List.nth(L,i)` restituisce l'*i*-esimo elemento di `L` (partendo da 0)
  - ◆ `List.last(L)` restituisce l'ultimo elemento di `L`
- vedere <http://sml-family.org/Basis/list.html>
- In ML, una *struttura* è simile a un modulo o un package





# Esercizi

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Le liste in ML

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Scrivere in ML:

1. le funzioni standard su liste riportate nelle slide precedenti: `hd`, `tl`, `List.nth`, `List.last`;
2. una funzione `member` che dati una lista `L` e un intero `N`, restituisce `true` se e solo se `N` appartiene ad `L`.
3. una funzione `append` che, dati una lista `L` e un intero `N`, accoda `N` in fondo alla lista (inserimento in coda);
4. una funzione `concat` che concatena due liste (come l'operatore `@`);
5. una funzione `reverse` che inverte l'ordine degli elementi di una lista (suggerimento: usare un parametro aggiuntivo che serve a costruire progressivamente la lista invertita).



[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

# Currying



# Funzioni con più argomenti: esistono?

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- In realtà in ML ogni funzione ha un solo argomento

```
fun f (x,y) = ...      (* l'argomento è una (singola) coppia *)
```

```
fun f x y = ...        (* l'argomento è x ! *)
```

Nel secondo caso,  $f$  è una funzione che *restituisce una funzione* che prende  $y$  e calcola l'espressione dopo '='



# Funzioni con più argomenti: esistono?

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- In realtà in ML ogni funzione ha un solo argomento

```
fun f (x,y) = ...      (* l'argomento è una (singola) coppia *)  
  
fun f x y = ...        (* l'argomento è x ! *)
```

Nel secondo caso,  $f$  è una funzione che *restituisce una funzione* che prende  $y$  e calcola l'espressione dopo '='

- Esempio semplice

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int
```

Il tipo  $\text{int} \rightarrow \text{int} \rightarrow \text{int}$  va inteso come  $\text{int} \rightarrow (\text{int} \rightarrow \text{int})$



# Currying

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La trasformazione da n-uple (come  $f(x,y)$ ) a funzioni che restituiscono funzioni (come  $f' \ x \ y$ ) si chiama *currying* (dal nome di Haskell Curry)
- L'utilizzo è ovviamente diverso

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int
```



# Currying

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La trasformazione da n-uple (come  $f(x,y)$ ) a funzioni che restituiscono funzioni (come  $f' \ x \ y$ ) si chiama *currying* (dal nome di Haskell Curry)
- L'utilizzo è ovviamente diverso

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int  
  
- f (3,2);  
val it = 5 : int
```

# Currying

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La trasformazione da n-uple (come  $f(x,y)$ ) a funzioni che restituiscono funzioni (come  $f' x y$ ) si chiama *currying* (dal nome di Haskell Curry)
- L'utilizzo è ovviamente diverso

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int  
  
- f (3,2);  
val it = 5 : int  
  
- f' 3 2; (* viene interpretato come (f'3)(2) *)  
val it = 5 : int
```



# Currying

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La trasformazione da n-uple (come  $f(x,y)$ ) a funzioni che restituiscono funzioni (come  $f' x y$ ) si chiama *currying* (dal nome di Haskell Curry)
- L'utilizzo è ovviamente diverso

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int  
  
- f (3,2);  
val it = 5 : int  
  
- f' 3 2;                                (* viene interpretato come (f'3)(2) *)  
val it = 5 : int  
  
- f 3 2;  
Error: operator and operand don't agree
```





# Currying

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La trasformazione da n-uple (come  $f(x,y)$ ) a funzioni che restituiscono funzioni (come  $f' x y$ ) si chiama *currying* (dal nome di Haskell Curry)
- L'utilizzo è ovviamente diverso

```
- fun f (x,y) = x+y;  
val f = fn : int * int -> int  
  
- fun f' x y = x+y;  
val f' = fn : int -> int -> int  
  
- f (3,2);  
val it = 5 : int  
  
- f' 3 2;                                (* viene interpretato come (f'3)(2) *)  
val it = 5 : int  
  
- f 3 2;  
Error: operator and operand don't agree  
  
- f' (3,2);  
Error: operator and operand don't agree
```



# Applicazione parziale

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- E' possibile invocare una funzione di più argomenti con un numero inferiore di parametri
- Si ottiene una funzione che:
  - ◆ ricorda i parametri già passati
  - ◆ e accetta i restanti argomenti
- Questo meccanismo si chiama *partial application*

```
- fun somma x y = x+y;  
val somma = fn : int -> int -> int
```



# Applicazione parziale

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- E' possibile invocare una funzione di più argomenti con un numero inferiore di parametri
- Si ottiene una funzione che:
  - ◆ ricorda i parametri già passati
  - ◆ e accetta i restanti argomenti
- Questo meccanismo si chiama *partial application*

```
- fun somma x y = x+y;  
val somma = fn : int -> int -> int
```

```
- val g = somma 3;  
val g = fn : int -> int
```



# Applicazione parziale

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- E' possibile invocare una funzione di più argomenti con un numero inferiore di parametri
- Si ottiene una funzione che:
  - ◆ ricorda i parametri già passati
  - ◆ e accetta i restanti argomenti
- Questo meccanismo si chiama *partial application*

```
- fun somma x y = x+y;  
val somma = fn : int -> int -> int
```

```
- val g = somma 3;  
val g = fn : int -> int
```

```
- g 1;  
val it = 4 : int
```



Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

**Funzioni di ordine superiore**

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

## Funzioni di ordine superiore



## Invece dei cicli: Funzioni di ordine superiore

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La maggior parte delle funzioni ricorsive che operano su liste, alberi e simili hanno la stessa struttura
- Cambia solo l'operazione che si applica ai nodi



## Invece dei cicli: Funzioni di ordine superiore

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La maggior parte delle funzioni ricorsive che operano su liste, alberi e simili hanno la stessa struttura
- Cambia solo l'operazione che si applica ai nodi
- Quindi basta scrivere una volta per tutte la funzione che scandisce la struttura dati (che fa la funzione del ciclo)
- e passargli la funzione da applicare ai nodi



## Invece dei cicli: Funzioni di ordine superiore

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La maggior parte delle funzioni ricorsive che operano su liste, alberi e simili hanno la stessa struttura
- Cambia solo l'operazione che si applica ai nodi
- Quindi basta scrivere una volta per tutte la funzione che scandisce la struttura dati (che fa la funzione del ciclo)
- e passargli la funzione da applicare ai nodi
  - ◆ Le funzioni che hanno altre funzioni come parametri sono dette di *ordine superiore*





# Esempio

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

**Esempio**

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Una funzione  $f$  che esegue due volte una funzione  $g$  su un valore  $x$ :

```
- fun f g x = g(g(x));  
val f = fn : ('a -> 'a) -> 'a -> 'a
```

Equivalente in C (solo per interi):

```
int f( int(*g)(int), int x) {  
    return g(g(x));  
}
```



## Invece dei cicli: Funzioni di ordine superiore

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

**Esempio**

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ Le tipologie di funzioni/ciclo più comuni sono tre:

- ◆ filtrare: *filter*
- ◆ trasformare: *map*
- ◆ aggregare: *reduce/fold*

Nel seguito mostriamo le loro versioni per le liste



# Filter

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La funzione `filter` prende una funzione booleana `f` e una lista `L`
- seleziona gli elementi di `L` per cui `f` è vera

```
fun filter f [] = []  
  | filter f (x::y) = if f(x) then x::(filter f y)  
                      else filter f y
```



# Filter

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La funzione `filter` prende una funzione booleana `f` e una lista `L`
- seleziona gli elementi di `L` per cui `f` è vera

```
fun filter f [] = []  
  | filter f (x::y) = if f(x) then x::(filter f y)  
                      else filter f y  
  
(* esempio:  seleziona gli elementi negativi da una lista *)  
- let fun neg x = x<0  
    in filter neg [0,~1,3,~2] end;  
val it = [~1,~2] : int list
```



# Filter

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La funzione `filter` prende una funzione booleana `f` e una lista `L`
- seleziona gli elementi di `L` per cui `f` è vera

```
fun filter f [] = []  
  | filter f (x::y) = if f(x) then x::(filter f y)  
                      else filter f y
```

(\* esempio: seleziona gli elementi negativi da una lista \*)

```
- let fun neg x = x < 0  
    in filter neg [0, ~1, 3, ~2] end;  
val it = [~1, ~2] : int list
```

(\* esempio: seleziona gli elementi positivi da una lista \*)

```
- let fun pos x = x > 0  
    in filter pos [0, ~1, 3, ~2] end;  
val it = [3] : int list
```

Nella libreria standard: `List.filter`



# Map

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La funzione `map` prende una funzione `f` e una lista `L`
- applica `f` a tutti gli elementi della lista, ottenendo una nuova lista

```
fun map f [] = []  
  | map f (x::y) = (f x)::(map f y)
```



# Map

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La funzione map prende una funzione f e una lista L
- applica f a tutti gli elementi della lista, ottenendo una nuova lista

```
fun map f [] = []  
  | map f (x::y) = (f x)::(map f y)  
  
(* esempio: conversione in lista di reali *)  
- map real [1,2,3];  
val it = [1.0,2.0,3.0] : real list
```



# Map

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La funzione map prende una funzione f e una lista L
- applica f a tutti gli elementi della lista, ottenendo una nuova lista

```
fun map f [] = []  
  | map f (x::y) = (f x)::(map f y)  
  
(* esempio: conversione in lista di reali *)  
- map real [1,2,3];  
val it = [1.0,2.0,3.0] : real list  
  
(* esempio: conversione in lista di stringhe *)  
- map Int.toString [1,2,3];  
val it = ["1","2","3"] : string list
```

Nella libreria standard: List.map





# Reduce/fold

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La funzione reduce serve per calcolare *aggregati* di una lista
  - ◆ min, max, somma, prodotto, media...



## Reduce/fold

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

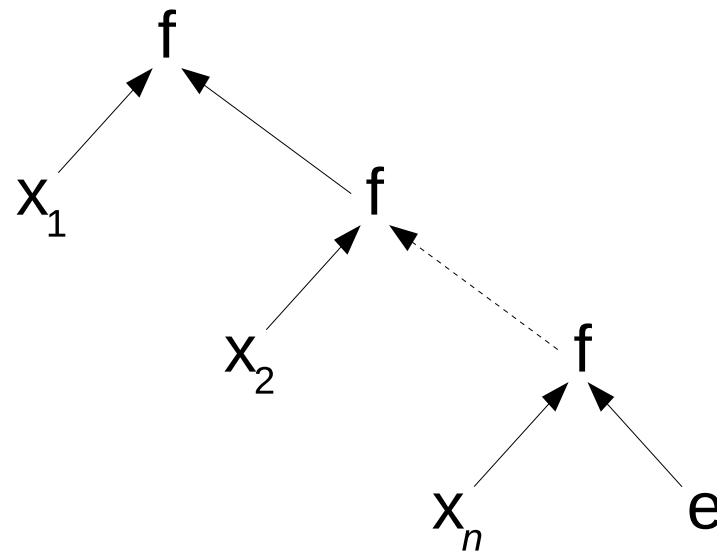
[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La funzione reduce serve per calcolare *aggregati* di una lista
  - ◆ min, max, somma, prodotto, media...
- Prende in input una funzione a 2 argomenti  $f$ , un valore finale  $e$  e una lista  $L$  ed effettua questo calcolo:

$$\text{reduce } f \text{ e } [x_1, x_2, \dots, x_n] = f(x_1, f(x_2, \dots f(x_n, e) \dots))$$





## Reduce/fold

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

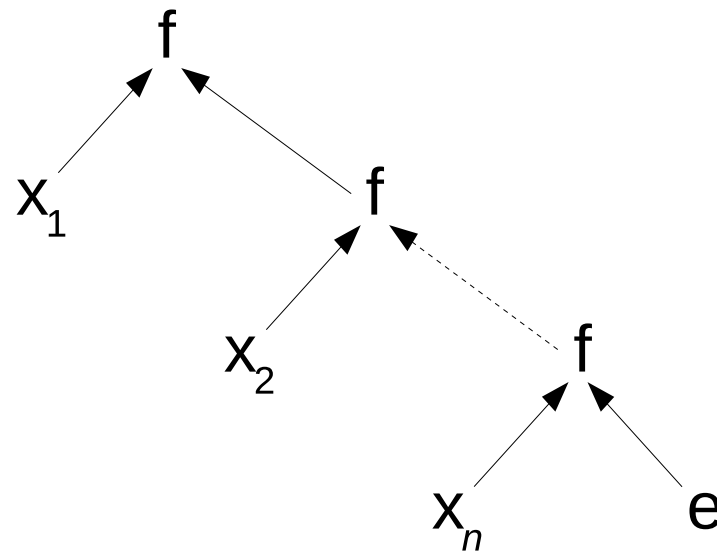
[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La funzione reduce serve per calcolare *aggregati* di una lista
  - ◆ min, max, somma, prodotto, media...
- Prende in input una funzione a 2 argomenti  $f$ , un valore finale  $e$  e una lista  $L$  ed effettua questo calcolo:

$$\text{reduce } f \text{ e } [x_1, x_2, \dots, x_n] = f(x_1, f(x_2, \dots f(x_n, e) \dots))$$





# Reduce

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ Definizione di reduce ed esempi

```
fun reduce f e [] = e
  | reduce f e (x :: y) = f (x, reduce f e y)
```

# Reduce

## ■ Definizione di reduce ed esempi

```
fun reduce f e [] = e
  | reduce f e (x :: y) = f (x, reduce f e y)
```

```
(* esempio: somma (sbagliato, + è infisso...) *)
- reduce + 0 [1,2,3];
Error: ...
```

```
(* esempio: somma (corretto) *)
- reduce (op +) 0 [1,2,3];
val it = 6 : int
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

# Reduce

## ■ Definizione di reduce ed esempi

```
fun reduce f e [] = e
  | reduce f e (x :: y) = f (x, reduce f e y)

(* esempio: somma (sbagliato, + è infisso...) *)
- reduce + 0 [1,2,3];
Error: ...

(* esempio: somma (corretto) *)
- reduce (op +) 0 [1,2,3];
val it = 6 : int

(* esempio: media di  $[x_1, \dots, x_n] = x_1/n + \dots + x_n/n$  *)
- let
    fun f L (elem, accum) = elem / real(length L) + accum
    val lista = [1.0, 2.0, 3.0]
  in
    reduce (f lista) 0.0 lista
  end;
val it = 2.0 : real
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



# Reduce

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore  
Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione  
con type checking

Esempio: un semplice  
compilatore

- Simile alla funzione standard `List.foldr`

- `List.foldr f e l` restituisce:  $f(x_1, f(x_2, \dots f(x_{n-1}, f(x_n, e))))$

- Su lista vuota, restituisce  $e$

- É più generale di reduce, perché  $f$  può accettare due tipi diversi:  $f : 'a * 'b \rightarrow 'b$

- L'aggregato può avere un tipo ( $'b$ ) diverso dagli elementi della lista ( $'a$ )

- Tipo di `List.foldr`:  $fn : ('a * 'b \rightarrow 'b) \rightarrow 'b \rightarrow 'a \text{ list} \rightarrow 'b$



# Esempio di foldr

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
- fun f (n:int, s:string) = Int.toString(n) ^ ";" ^ s;  
val f = fn : int * string -> string
```

```
- f (323, "aaa");  
val it = "323;aaa" : string
```

```
- List.foldr f "fine" [23,3434,44,0,12];  
val it = "23;3434;44;0;12;fine" : string
```





## In altri linguaggi

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

### In Java 8+ (semplificato):

```
interface java.util.stream.Stream<T> {  
    ...  
    Stream<T> filter(Predicate<T> predicate)  
    <R> Stream<R> map(Function<T,R> mapper)  
    T reduce(T identity, BinaryOperator<T> accumulator)  
}
```

### In Python:

```
filter(funzione lista)  
map(funzione, lista)  
reduce(funzione, lista [, inizializzatore])
```



# Esercizi su funzioni di ordine superiore

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Implementare in ML le seguenti funzioni, sfruttando al meglio filter, map e reduce

1. `pari`: data una lista di interi `L`, restituisce la sottolista che contiene solo i numeri pari in `L`
2. `membro`: dati una lista di interi e un intero, restituisce *true* se e solo se l'intero compare nella lista
3. `concat`: data una lista di stringhe, restituisce la loro concatenazione
4. `max`: data una lista di interi, restituisce il massimo



# Funzioni anonime

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Quando utilizziamo funzioni di ordine superiori come filter, map e reduce, può far comodo passargli funzioni semplici, specificate lì per lì senza dover dare loro un nome (né usare un blocco *let*)
- Queste funzioni anonime si specificano con la keyword **fn**

Sintassi:

```
<expr> ::= fn <argomento> => <expr>
```

Esempi:

```
- fn x => x+1;
val it = fn : int -> int

(* per sommare uno a tutti gli elementi di una lista *)
- map (fn x => x+1) [1,2,3];
val it = [2,3,4] : int list

- filter (fn x => (x mod 2) = 0) [10,11,12,13];
val it = [10,12] : int list

- fn x => fn y => x+y;
val it = fn : int -> int -> int
```



## In altri linguaggi funzionali

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter, Map, Reduce

**Funzioni anonime**

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

In Lisp e Scheme l'equivalente di `fn` è la keyword *lambda*

- storicamente deriva dal *lambda calcolo*, un modello di calcolo matematico basato su funzioni di ordine superiore a cui tutti i linguaggi funzionali si sono ispirati
- nel lambda calcolo l'operatore  $\lambda$  è l'analogo di `fn`
  - ◆ come in  $\lambda x. x + 1$



## In altri linguaggi imperativi

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

Una funzione anonima che accetta una stringa e restituisce vero se è più lunga di 10 caratteri

In Java:

```
s -> s.length() > 10;
```

In C++:

```
[](string s) { return s.length() > 10; };
```

In Python:

```
lambda s: len(s) > 10
```



# Esercizi su funzioni anonime

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

1. Spiegare perchè una funzione anonima non può essere ricorsiva
2. Svolgere nuovamente gli esercizi sulle funzioni di ordine superiore, sfruttando al meglio le *funzioni anonime*



## Val vs. fun

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Esempio

Filter, Map, Reduce

Funzioni anonime

Val vs. fun

Currying

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Adesso possiamo vedere in che senso `val` è più generale di `fun`. Le seguenti definizioni sono equivalenti:

```
- fun f x = x + 1;  
val f = fn : int -> int  
  
(* equivalente al precedente *)  
- val f = fn x => x+1;  
val f = fn : int -> int
```

In altre parole, `fun` è *zucchero sintattico*, cioè una utile abbreviazione per qualcosa che si potrebbe fare in altro modo (con `val`)



## Ulteriori dettagli su currying

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Adesso possiamo mostrare più esplicitamente la natura del currying. Riprendiamo l'esempio

```
- fun f' x y = x+y;
```

- In effetti  $f'$  può essere definita equivalentemente come

```
fun f' x = fn y => x+y
```

```
val f' = fn : int -> int -> int
```





## Ulteriori dettagli su currying

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Adesso possiamo mostrare più esplicitamente la natura del currying. Riprendiamo l'esempio

```
- fun f' x y = x+y;
```

- In effetti  $f'$  può essere definita equivalentemente come

```
fun f' x = fn y => x+y
```

```
val f' = fn : int -> int -> int
```

- L'esempio della chiamata di  $f'$  con un solo parametro può essere spiegata così:

```
- val g = f' 3;      (* come fosse val g = fn y => 3+y *)  
val g = fn : int -> int
```

```
- g 1;  
val it = 4 : int
```



## Ulteriori dettagli su currying

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- È anche possibile definire funzioni che effettuano il currying e la sua trasformazione inversa per una data funzione del tipo giusto

```
(* f deve accettare una coppia (x,y) *)  
val curry_2args = fn f => fn x => fn y => f (x, y)
```



## Ulteriori dettagli su currying

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- È anche possibile definire funzioni che effettuano il currying e la sua trasformazione inversa per una data funzione del tipo giusto

```
(* f deve accettare una coppia (x,y) *)  
val curry_2args = fn f => fn x => fn y => f (x, y)  
  
(* f deve essere del tipo f x y *)  
val uncurry_2args = fn f => fn (x, y) => f x y
```



## Ulteriori dettagli su currying

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- È anche possibile definire funzioni che effettuano il currying e la sua trasformazione inversa per una data funzione del tipo giusto

```
(* f deve accettare una coppia (x,y) *)  
val curry_2args = fn f => fn x => fn y => f (x, y)  
  
(* f deve essere del tipo f x y *)  
val uncurry_2args = fn f => fn (x, y) => f x y
```

- Esempi di utilizzo

```
fun f (x,y) = x+y;  
  
val f' = curry_2args f; (* equivalente a f' x y = x+y *)
```



## Ulteriori dettagli su currying

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Esempio](#)

[Filter, Map, Reduce](#)

[Funzioni anonime](#)

[Val vs. fun](#)

[Currying](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- È anche possibile definire funzioni che effettuano il currying e la sua trasformazione inversa per una data funzione del tipo giusto

```
(* f deve accettare una coppia (x,y) *)  
val curry_2args = fn f => fn x => fn y => f (x, y)  
  
(* f deve essere del tipo f x y *)  
val uncurry_2args = fn f => fn (x, y) => f x y
```

- Esempi di utilizzo

```
fun f (x,y) = x+y;  
  
val f' = curry_2args f; (* equivalente a f' x y = x+y *)
```

oppure

```
fun f' x y = x+y;  
  
val f = uncurry_2args f'; (* equivalente a f(x,y) = x+y *)
```



[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Tipi parametrici](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

# Polimorfismo parametrico



# Tipi parametrici

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Tipi parametrici](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- ML supporta l'analogo dei *template* di C++ e dei *generics* di Java, ovvero i *tipi parametrici*



# Tipi parametrici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

**Tipi parametrici**

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML supporta l'analogo dei *template* di C++ e dei *generics* di Java, ovvero i *tipi parametrici*
- In realtà li stiamo usando da quando usiamo le liste:
  - ◆ il costruttore `::` può essere applicato a qualunque tipo

```
1 :: nil    oppure    "abc" :: nil    ...
```





# Tipi parametrici

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

**Tipi parametrici**

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- ML supporta l'analogo dei *template* di C++ e dei *generics* di Java, ovvero i *tipi parametrici*
- In realtà li stiamo usando da quando usiamo le liste:
  - ◆ il costruttore `::` può essere applicato a qualunque tipo

```
1 :: nil    oppure    "abc" :: nil    ...
```

```
- length;  
val it = fn : 'a list -> int
```

La funzione *length* prende una lista di elementi il cui tipo `'a` non è specificato

- ◆ cioè accetta liste con qualsiasi contenuto
- ◆ in effetti non ha bisogno di saperne il tipo: deve solo contare in nodi



# Tipi parametrici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Tipi parametrici

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML supporta l'analogo dei *template* di C++ e dei *generics* di Java, ovvero i *tipi parametrici*
- In realtà li stiamo usando da quando usiamo le liste:
  - ◆ il costruttore `::` può essere applicato a qualunque tipo

```
1 :: nil    oppure    "abc" :: nil    ...
```

```
- length;  
val it = fn : 'a list -> int
```

La funzione *length* prende una lista di elementi il cui tipo `'a` non è specificato

- ◆ cioè accetta liste con qualsiasi contenuto
- ◆ in effetti non ha bisogno di saperne il tipo: deve solo contare in nodi
- Anche la funzione *map* è parametrica:

```
- map;  
val it = fn : ('a -> 'b) -> 'a list -> 'b list
```



# Tipi parametrici

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Tipi parametrici](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ Come definire tipi parametrici (come le liste )

```
(* generalizzazione delle nostre liste *)
```

```
- datatype 'a lista = vuota | nodo of ('a * 'a lista);  
datatype 'a lista = nodo of 'a * 'a lista | vuota
```



# Tipi parametrici

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Tipi parametrici](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## ■ Come definire tipi parametrici (come le liste )

```
(* generalizzazione delle nostre liste *)
```

```
- datatype 'a lista = vuota | nodo of ('a * 'a lista);  
datatype 'a lista = nodo of 'a * 'a lista | vuota
```

```
(* alberi binari con etichette parametriche *)
```

```
- datatype 'a bt = emptybt | btnode of ('a * 'a bt * 'a bt);  
datatype 'a bt = btnode of 'a * 'a bt * 'a bt | emptybt
```



# Tipi parametrici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Tipi parametrici

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

## ■ Come definire tipi parametrici (come le liste )

```
(* generalizzazione delle nostre liste *)
```

```
- datatype 'a lista = vuota | nodo of ('a * 'a lista);  
datatype 'a lista = nodo of 'a * 'a lista | vuota
```

```
(* alberi binari con etichette parametriche *)
```

```
- datatype 'a bt = emptybt | btnode of ('a * 'a bt * 'a bt);  
datatype 'a bt = btnode of 'a * 'a bt * 'a bt | emptybt
```

## ■ Per le funzioni (quando il compilatore non ha bisogno di aiuto per stabilirne il tipo) non dobbiamo fare niente di speciale

### ◆ ci pensa la *type inference*

```
- fun conta(vuota) = 0  
    | conta (nodo(x,l)) = conta(l) + 1;  
val conta = fn : 'a lista -> int
```



# Tipi parametrici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

**Tipi parametrici**

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Come si può vedere dagli esempi precedenti ML usa l'apice prima del nome per indicare che quella è una *variabile di tipo*
  - ◆ ad esempio 'a o 'b o 'c ...



# Tipi parametrici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

**Tipi parametrici**

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Come si può vedere dagli esempi precedenti ML usa l'apice prima del nome per indicare che quella è una *variabile di tipo*
  - ◆ ad esempio `'a` o `'b` o `'c` ...
- Quando si vuole che il tipo supporti l'uguaglianza, allora si mette un doppio apice
  - ◆ ad esempio `''a` o `''b` o `''c` ...



## Tipi parametrici

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Tipi parametrici

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Come si può vedere dagli esempi precedenti ML usa l'apice prima del nome per indicare che quella è una *variabile di tipo*
  - ◆ ad esempio 'a o 'b o 'c ...
- Quando si vuole che il tipo supporti l'uguaglianza, allora si mette un doppio apice
  - ◆ ad esempio ''a o ''b o ''c ...
- Ogni tanto la type inference se ne accorge da sola

```
- fun diag (x,y) = x=y;  
val diag = fn : ''a * ''a -> bool  
  
- diag (1.0, 1.0);  
Error: operator and operand don't agree [equality type required]  
operator domain: ''Z * ''Z  
operand:          real * real
```

(ricordarsi che i reali non supportano l'uguaglianza...)





## Esercizi su polimorfismo parametrico

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Tipi parametrici](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

1. Definire un tipo per le triple che hanno un intero come primo e terzo componente, e qualsiasi tipo al centro
2. Per un programma di gestione di un magazzino, definire un tipo di record chiamato *prodotto* che contenga un valore arbitrario, accompagnato da un prezzo e da una numerosità intera
3. Definire un datatype per un albero binario *non vuoto* in cui i nodi interni contengono valori di un certo tipo, e le foglie contengono un valore di un altro tipo
4. Ripetere l'esercizio precedente, ammettendo anche alberi vuoti
5. Definire un datatype per un albero binario in cui i nodi di profondità pari contengono valori di un certo tipo, e i nodi di profondità dispari valori di un altro tipo (assumendo che la profondità della radice sia 1)



[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

**[Encapsulation e interfacce](#)**

[Signatures](#)

[Structures](#)

[Incapsulamento](#)

[Functors](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

# Encapsulation e interfacce



# Signatures = Interfacce

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

**Signatures**

[Structures](#)

[Incapsulamento](#)

[Functors](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati



# Signatures = Interfacce

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

**[Signatures](#)**

[Structures](#)

[Incapsulamento](#)

[Functors](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati
- Esempio: STACK

```
signature STACK =  
sig  
  type 'a stack  
  val empty: 'a stack  
  val push: ('a * 'a stack) -> 'a stack  
  val pop: 'a stack -> ('a * 'a stack)  
end;
```



# Signatures = Interfacce

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

**[Signatures](#)**

[Structures](#)

[Incapsulamento](#)

[Functors](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati
- Esempio: STACK

```
signature STACK =  
sig  
  type 'a stack  
  val empty: 'a stack  
  val push: ('a * 'a stack) -> 'a stack  
  val pop: 'a stack -> ('a * 'a stack)  
end;
```

- dichiara un tipo parametrico 'a stack senza dire com'è definito



# Signatures = Interfacce

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

**[Signatures](#)**

[Structures](#)

[Incapsulamento](#)

[Functors](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati
- Esempio: STACK

```
signature STACK =  
sig  
  type 'a stack  
  val empty: 'a stack  
  val push: ('a * 'a stack) -> 'a stack  
  val pop: 'a stack -> ('a * 'a stack)  
end;
```

- dichiara un tipo parametrico 'a stack senza dire com'è definito
- una funzione empty per costruire uno stack vuoto



# Signatures = Interfacce

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

**[Signatures](#)**

[Structures](#)

[Incapsulamento](#)

[Functors](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati
- Esempio: STACK

```
signature STACK =  
sig  
  type 'a stack  
  val empty: 'a stack  
  val push: ('a * 'a stack) -> 'a stack  
  val pop: 'a stack -> ('a * 'a stack)  
end;
```

- dichiara un tipo parametrico 'a stack senza dire com'è definito
- una funzione empty per costruire uno stack vuoto
- una funzione push per inserire un elemento nello stack



# Signatures = Interfacce

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

**Signatures**

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati
- Esempio: STACK

```
signature STACK =  
sig  
  type 'a stack  
  val empty: 'a stack  
  val push: ('a * 'a stack) -> 'a stack  
  val pop: 'a stack -> ('a * 'a stack) ●  
end;
```

- dichiara un tipo parametrico 'a stack senza dire com'è definito
- una funzione empty per costruire uno stack vuoto
- una funzione push per inserire un elemento nello stack
- una funzione pop per estrarre la testa dallo stack
- ...senza dire come sono implementate (ovviamente)...





# Signatures = Interfacce

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

**Signatures**

Structures

Incapsulamento

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Le *signature* sono il costrutto ML per definire interfacce (nel senso di Java)
- Definiscono tipi e funzioni senza specificare come sono implementati
- Esempio: STACK

```
signature STACK =  
sig  
  type 'a stack  
  val empty: 'a stack  
  val push: ('a * 'a stack) -> 'a stack  
  val pop: 'a stack -> ('a * 'a stack)  
end;
```

- dichiara un tipo parametrico 'a stack senza dire com'è definito
- una funzione empty per costruire uno stack vuoto
- una funzione push per inserire un elemento nello stack
- una funzione pop per estrarre la testa dallo stack
- ...senza dire come sono implementate (ovviamente)...
- *per assegnare un tipo a una espressione usare :*



# Structures = Implementazioni delle signature

- Le structure, come le classi, definiscono *tipi di dato astratti*
- Esempio di implementazione di STACK mediante una lista

```
structure Stack :> STACK =  
  struct  
    type 'a stack = 'a list;  
    val empty = [];  
    fun push (x,s) = x :: s;  
    fun pop (x::s) = (x,s);  
  end;
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Signatures](#)

**Structures**

[Incapsulamento](#)

[Functors](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



# Structures = Implementazioni delle signature

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Signatures](#)

**Structures**

[Incapsulamento](#)

[Functors](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Le structure, come le classi, definiscono *tipi di dato astratti*
- Esempio di implementazione di STACK mediante una lista

```
structure Stack :> STACK =  
struct  
  type 'a stack = 'a list;  
  val empty = [];  
  fun push (x,s) = x :: s;  
  fun pop (x::s) = (x,s);  
end;
```

- Con l'espressione `Stack :> STACK` diciamo diverse cose:
  1. Stack deve implementare tutti gli identificatori dichiarati in STACK



# Structures = Implementazioni delle signature

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Signatures](#)

**Structures**

[Incapsulamento](#)

[Functors](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Le structure, come le classi, definiscono *tipi di dato astratti*
- Esempio di implementazione di STACK mediante una lista

```
structure Stack :> STACK =  
struct  
  type 'a stack = 'a list;  
  val empty = [];  
  fun push (x,s) = x :: s;  
  fun pop (x::s) = (x,s);  
end;
```

- Con l'espressione `Stack :> STACK` diciamo diverse cose:
  1. Stack deve implementare tutti gli identificatori dichiarati in STACK
  2. I tipi di dato dichiarati in Stack possono essere utilizzati *solo* con le operazioni dichiarate in STACK
    - ◆ ogni altra funzione definita nella structure non è accessibile da fuori



# Structures = Implementazioni delle signature

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Signatures](#)

**Structures**

[Incapsulamento](#)

[Functors](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Le structure, come le classi, definiscono *tipi di dato astratti*
- Esempio di implementazione di STACK mediante una lista

```
structure Stack :> STACK =  
struct  
  type 'a stack = 'a list;  
  val empty = [];  
  fun push (x,s) = x :: s;  
  fun pop (x::s) = (x,s);  
end;
```

- Con l'espressione `Stack :> STACK` diciamo diverse cose:
  1. Stack deve implementare tutti gli identificatori dichiarati in STACK
  2. I tipi di dato dichiarati in Stack possono essere utilizzati *solo* con le operazioni dichiarate in STACK
    - ◆ ogni altra funzione definita nella structure non è accessibile da fuori
    - ◆ così si ottiene l'*encapsulation*
    - ◆ e si definiscono *tipi di dato astratti*



# Incapsulamento dell'implementazione dei tipi

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Signatures](#)

[Structures](#)

**[Incapsulamento](#)**

[Functors](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Anche se in Stack il tipo stack è implementato con list...

```
type 'a stack = 'a list;
```

- ... è compatibile con list ...



# Incapsulamento dell'implementazione dei tipi

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

**Incapsulamento**

Functors

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Anche se in Stack il tipo stack è implementato con list...

```
type 'a stack = 'a list;
```

- ... è compatibile con list ...

```
- length [];  
val it = 0 : int  
  
- length Stack.empty;  
stdIn:39.1-39.19 Error: operator and operand don't agree  
      operator domain: 'Z list  
      operand:         'Y Stack.stack
```

- ... perchè la structure Stack non mette a disposizione alcuna funzione length sul tipo stack e ne nasconde l'implementazione



# Functors = structure parametriche

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Signatures

Structures

Incapsulamento

**Functors**

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Alcune delle componenti di una structure possono essere variabili ed essere specificate come dei parametri
- Si usa una keyword diversa: `functor`. Analogo dei template





# Functors = structure parametriche

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Signatures](#)

[Structures](#)

[Incapsulamento](#)

**[Functors](#)**

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Alcune delle componenti di una structure possono essere variabili ed essere specificate come dei parametri
- Si usa una keyword diversa: `functor`. Analogo dei template
- Ecco un esempio di definizione di immagini parametrica rispetto alla codifica del colore
  - ◆ supponiamo di avere due structure RGB e CMYK per gli omonimi modelli di colore
  - ◆ e che entrambe implementino la signature `COLOR`
  - ◆ la struttura parametrica si può dichiarare così:



# Functors = structure parametriche

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Signatures](#)

[Structures](#)

[Incapsulamento](#)

**[Functors](#)**

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Alcune delle componenti di una structure possono essere variabili ed essere specificate come dei parametri
- Si usa una keyword diversa: `functor`. Analogo dei template
- Ecco un esempio di definizione di immagini parametrica rispetto alla codifica del colore
  - ◆ supponiamo di avere due structure RGB e CMYK per gli omonimi modelli di colore
  - ◆ e che entrambe implementino la signature COLOR
  - ◆ la struttura parametrica si può dichiarare così:

```
functor Image( X : COLOR ) =  
  struct  
    (* qui si può usare X come un tipo *)  
    (* con le operazioni definite da COLOR *)  
  end  
  
  (* col functor si possono generare diversi tipi di dato *)  
  structure Image_RGB = Image( RGB );  
  structure Image_CMYK = Image( CMYK );
```



[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

## Eccezioni e integrazione con type checking



# Eccezioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```



# Eccezioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col type checking

```
- fun pop(x::s) = (x,s);  
Warning: match nonexhaustive  
      x :: s => ...
```

# Eccezioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col type checking

```
- fun pop(x::s) = (x,s);  
Warning: match nonexhaustive  
      x :: s => ...  
  
- pop [];  
uncaught exception Match [nonexhaustive match failure]
```

Ecco come funziona:



# Eccezioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col type checking

```
- fun pop(x::s) = (x,s);  
Warning: match nonexhaustive  
      x :: s => ...  
  
- pop [];  
uncaught exception Match [nonexhaustive match failure]
```

Ecco come funziona:

1. la type inference capisce che l'input di pop è una lista



# Eccezioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col type checking

```
- fun pop(x::s) = (x,s);  
Warning: match nonexhaustive  
      x :: s => ...  
  
- pop [];  
uncaught exception Match [nonexhaustive match failure]
```

Ecco come funziona:

1. la type inference capisce che l'input di pop è una lista
2. il datatype lista ha due costruttori: `::` e `[]`





# Eccezioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col type checking

```
- fun pop(x::s) = (x,s);  
Warning: match nonexhaustive  
      x :: s => ...  
  
- pop [];  
uncaught exception Match [nonexhaustive match failure]
```

Ecco come funziona:

1. la type inference capisce che l'input di pop è una lista
2. il datatype lista ha due costruttori: `::` e `[]`
3. la definizione per casi di pop ha un caso solo per `::`
4. da cui il warning



# Eccezioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Come Java, anche ML ha le sue eccezioni predefinite...

```
- 3 div 0;  
uncaught exception Div [divide by zero]
```

- In ML la gestione delle eccezioni è integrata col type checking

```
- fun pop(x::s) = (x,s);  
Warning: match nonexhaustive  
      x :: s => ...  
  
- pop [];  
uncaught exception Match [nonexhaustive match failure]
```

Ecco come funziona:

1. la type inference capisce che l'input di pop è una lista
2. il datatype lista ha due costruttori: `::` e `[]`
3. la definizione per casi di pop ha un caso solo per `::`
4. da cui il warning
5. il compilatore inserisce automaticamente una eccezione `Match` nei casi mancanti



# Dichiarazione e generazione eccezioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Il programmatore può definire le proprie eccezioni:

```
exception EmptyStack;      (* dichiara una nuova eccezione *)

fun pop(x::s) = (x,s)
  | pop [] = raise EmptyStack;  (* come il throw di Java *)
```



# Dichiarazione e generazione eccezioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Il programmatore può definire le proprie eccezioni:

```
exception EmptyStack;          (* dichiara una nuova eccezione *)

fun pop(x::s) = (x,s)
  | pop [] = raise EmptyStack;  (* come il throw di Java *)
```

Il risultato in caso di errore è più esplicativo dell'eccezione "automatica" Match

```
- pop [];  
uncaught exception EmptyStack
```



# Dichiarazione e generazione eccezioni

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Il programmatore può definire le proprie eccezioni:

```
exception EmptyStack;          (* dichiara una nuova eccezione *)

fun pop(x::s) = (x,s)
  | pop [] = raise EmptyStack;  (* come il throw di Java *)
```

Il risultato in caso di errore è più esplicativo dell'eccezione "automatica" Match

```
- pop [];  
uncaught exception EmptyStack
```

- Le eccezioni possono essere catturate e gestite con handle:

```
pop x  
handle EmptyStack =>  
  ( print "messaggio di errore specializzato";  
    raise EmptyStack );
```



## Il costrutto `handle`

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- può essere messo dopo qualunque espressione che può generare una eccezione

```
(3 div x) handle ...
```



## Il costrutto `handle`

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- può essere messo dopo qualunque espressione che può generare una eccezione

```
(3 div x) handle ...
```

- un singolo `handle` può gestire diverse eccezioni

```
<expr> ::= <expr> handle  
          <exception 1> => ...  
          | <exception 2> => ...  
          | ...
```



## Il costrutto `handle`

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- può essere messo dopo qualunque espressione che può generare una eccezione

```
(3 div x) handle ...
```

- un singolo `handle` può gestire diverse eccezioni

```
<expr> ::= <expr> handle  
          <exception 1> => ...  
          | <exception 2> => ...  
          | ...
```

*Ogni ordine è buono: perchè?*





## Il costrutto `handle`

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- può essere messo dopo qualunque espressione che può generare una eccezione

```
(3 div x) handle ...
```

- un singolo `handle` può gestire diverse eccezioni

```
<expr> ::= <expr> handle
          <exception 1> => ...
          | <exception 2> => ...
          | ...
```

*Ogni ordine è buono: perchè?*

- Due modi di usarlo in ML funzionale puro:

1. fare qualcosa come stampare un messaggio e *rilanciare un'eccezione*



## Il costrutto `handle`

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- può essere messo dopo qualunque espressione che può generare una eccezione

```
(3 div x) handle ...
```

- un singolo `handle` può gestire diverse eccezioni

```
<expr> ::= <expr> handle
          <exception 1> => ...
          | <exception 2> => ...
          | ...
```

*Ogni ordine è buono: perchè?*

- Due modi di usarlo in ML funzionale puro:

1. fare qualcosa come stampare un messaggio e *rilanciare un'eccezione*
2. “aggiustare” l'errore restituendo un valore *dello stesso tipo* dell'espressione che ha sollevato l'eccezione

Sono le uniche opzioni che passano il type checking senza errori



## Il costruito handle

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

### ■ Altro esempio (da non seguire acriticamente ...)

```
fun pos x (y::z) =  
  if (x=y) then 1 else 1 + pos x z;
```

Questa funzione restituisce la posizione di x nella lista, ma se non trova x solleva una eccezione (manca un caso terminale per [])



## Il costrutto handle

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

### ■ Altro esempio (da non seguire acriticamente ...)

```
fun pos x (y::z) =  
  if (x=y) then 1 else 1 + pos x z;
```

Questa funzione restituisce la posizione di x nella lista, ma se non trova x solleva una eccezione (manca un caso terminale per [])

### ■ Si può usare handle per modificare pos per restituire -1 quando non trova x nella lista:

```
fun pos2 x y = (pos x y) handle Match => ~1;
```



## Il costruito handle

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

### ■ Altro esempio (da non seguire acriticamente ...)

```
fun pos x (y::z) =  
  if (x=y) then 1 else 1 + pos x z;
```

Questa funzione restituisce la posizione di x nella lista, ma se non trova x solleva una eccezione (manca un caso terminale per [])

### ■ Si può usare handle per modificare pos per restituire -1 quando non trova x nella lista:

```
fun pos2 x y = (pos x y) handle Match => ~1;
```

### ■ Esempio didattico un po' artificiale: si potrebbe obiettare che pos è realizzata male...



# Eccezioni con parametri

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Si possono aggiungere dettagli sull'errore che si è verificato aggiungendo parametri alle eccezioni
- Esempio di eccezione con parametri:

```
exception SyntaxError of string
```



# Eccezioni con parametri

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Si possono aggiungere dettagli sull'errore che si è verificato aggiungendo parametri alle eccezioni
- Esempio di eccezione con parametri:

```
exception SyntaxError of string
```

- Questa eccezione può essere lanciata in diversi modi...

```
raise SyntaxError "Identifier_ expected"
```

```
raise SyntaxError "Integer_ expected"
```



# Eccezioni con parametri

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Si possono aggiungere dettagli sull'errore che si è verificato aggiungendo parametri alle eccezioni

- Esempio di eccezione con parametri:

```
exception SyntaxError of string
```

- Questa eccezione può essere lanciata in diversi modi...

```
raise SyntaxError "Identifier␣expected"
```

```
raise SyntaxError "Integer␣expected"
```

- ... e il parametro “letto” col pattern matching

```
... handle SyntaxError x => ... (* qui si può usare x *)
```





[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

**Esempio: un semplice compilatore**

## Esempio: compilazione di espressioni



# Elaborazione di simboli in ML

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML – come gli altri linguaggi dichiarativi (= non imperativi) – è particolarmente adatto alla *manipolazione di simboli*
- La *realizzazione di compilatori* è un esempio di questo tipo di problema
  - ◆ bisogna elaborare espressioni e comandi di un linguaggio di programmazione (codice sorgente) e tradurli in un altro linguaggio (codice oggetto o intermedio)



# Elaborazione di simboli in ML

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- ML – come gli altri linguaggi dichiarativi (= non imperativi) – è particolarmente adatto alla *manipolazione di simboli*
- La *realizzazione di compilatori* è un esempio di questo tipo di problema
  - ◆ bisogna elaborare espressioni e comandi di un linguaggio di programmazione (codice sorgente) e tradurli in un altro linguaggio (codice oggetto o intermedio)
- Ne approfittiamo per dare un'idea parziale di alcune strutture dati interne al compilatore e dei procedimenti di generazione e ottimizzazione del codice



## Elaborazione di simboli in ML

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- ML – come gli altri linguaggi dichiarativi (= non imperativi) – è particolarmente adatto alla *manipolazione di simboli*
- La *realizzazione di compilatori* è un esempio di questo tipo di problema
  - ◆ bisogna elaborare espressioni e comandi di un linguaggio di programmazione (codice sorgente) e tradurli in un altro linguaggio (codice oggetto o intermedio)
- Ne approfittiamo per dare un'idea parziale di alcune strutture dati interne al compilatore e dei procedimenti di generazione e ottimizzazione del codice
- L'esempio che segue realizza un compilatore per un linguaggio molto semplificato che supporta solo semplici espressioni su numeri interi. Il codice oggetto deve calcolare l'espressione data.



# Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

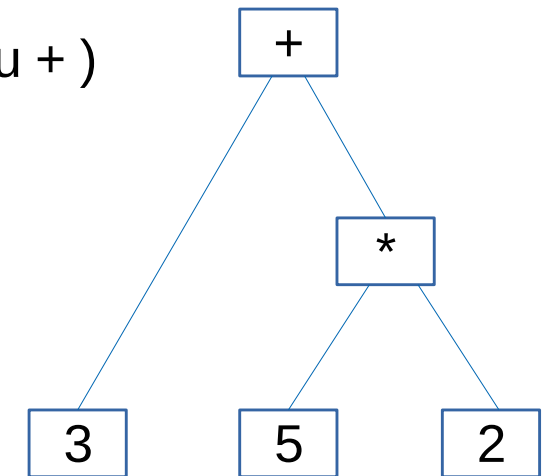
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione:  $3 + 5 * 2$

parser  
→

Albero sintattico  
( \* ha precedenza su + )





# Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

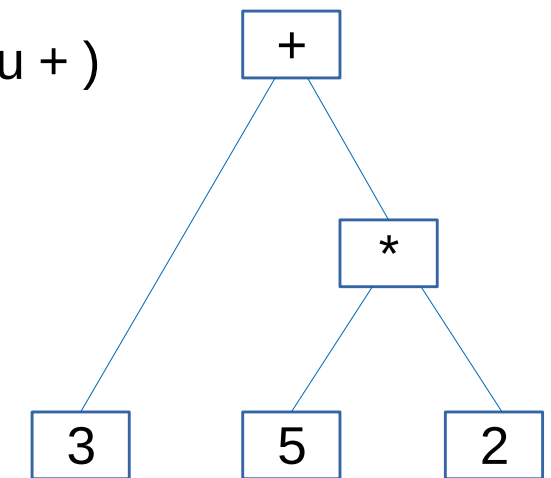
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione:  $3 + 5 * 2$  

parser

Albero sintattico  
( \* ha precedenza su + )



Procedimento di calcolo:

- salvare i risultati intermedi (come  $5*2$ ) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
  - se sono su una foglia la metto sullo stack
  - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Esempio:

- ordine posticipato di visita:  $3\ 5\ 2\ *\ +$



# Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

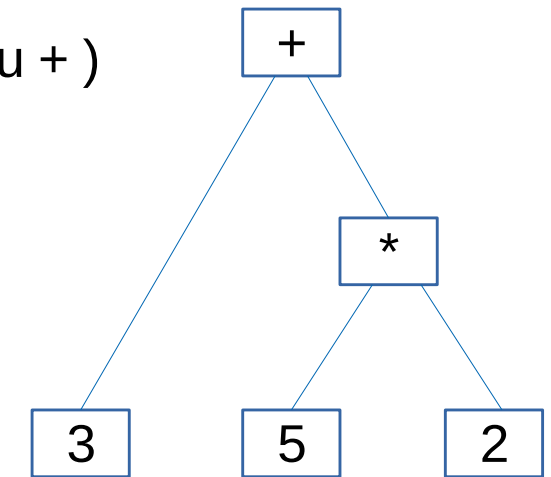
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione:  $3 + 5 * 2$  

parser

Albero sintattico  
( \* ha precedenza su + )



Procedimento di calcolo:

- salvare i risultati intermedi (come  $5*2$ ) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
  - se sono su una foglia la metto sullo stack
  - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Stack

3

Esempio:

- ordine posticipato di visita:  $3\ 5\ 2\ *\ +$



# Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying


Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

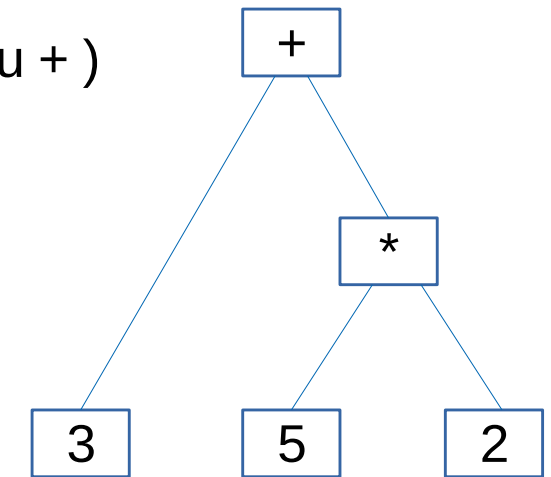
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione:  $3 + 5 * 2$  

parser

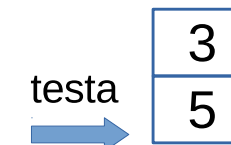
Albero sintattico  
( \* ha precedenza su + )



Procedimento di calcolo:

- salvare i risultati intermedi (come  $5*2$ ) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
  - se sono su una foglia la metto sullo stack
  - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Stack



Esempio:

- ordine posticipato di visita:  $3\ 5\ 2\ *\ +$



# Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying


Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

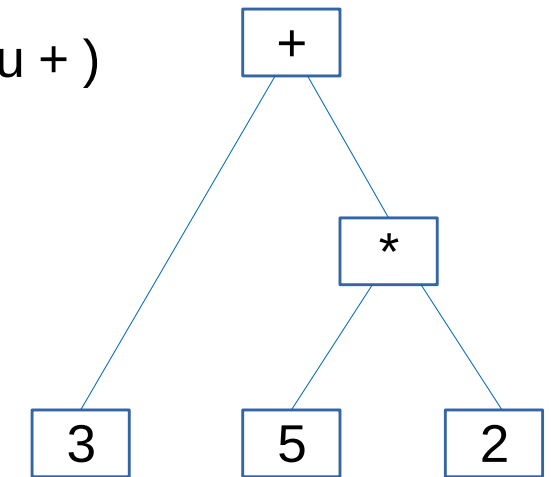
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione:  $3 + 5 * 2$  

parser

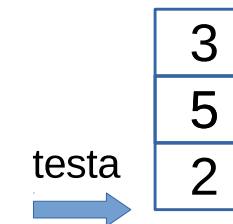
Albero sintattico  
( \* ha precedenza su + )



Procedimento di calcolo:

- salvare i risultati intermedi (come  $5*2$ ) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
  - se sono su una foglia la metto sullo stack
  - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Stack



Esempio:

- ordine posticipato di visita:  $3\ 5\ 2\ *\ +$

# Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

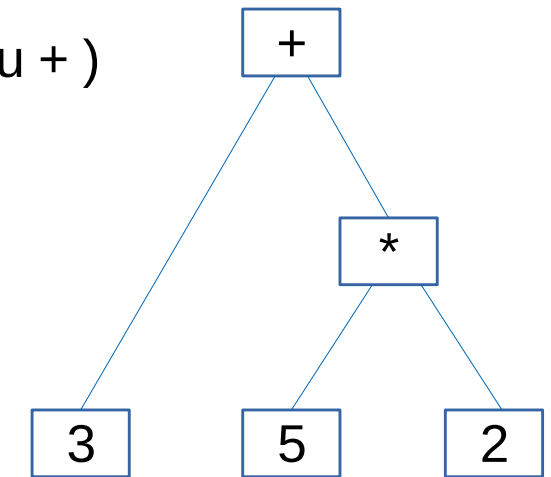
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione:  $3 + 5 * 2$  

parser

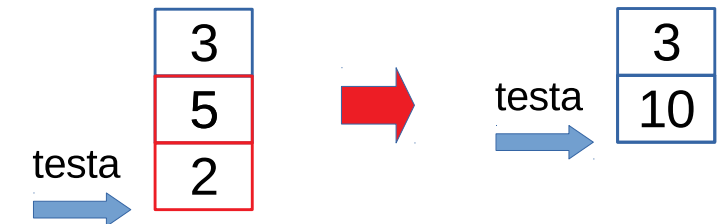
Albero sintattico  
( \* ha precedenza su + )



Procedimento di calcolo:

- salvare i risultati intermedi (come  $5*2$ ) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
  - se sono su una foglia la metto sullo stack
  - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Stack



Esempio:

- ordine posticipato di visita:  $3\ 5\ 2\ *\ +$

# Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying


Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

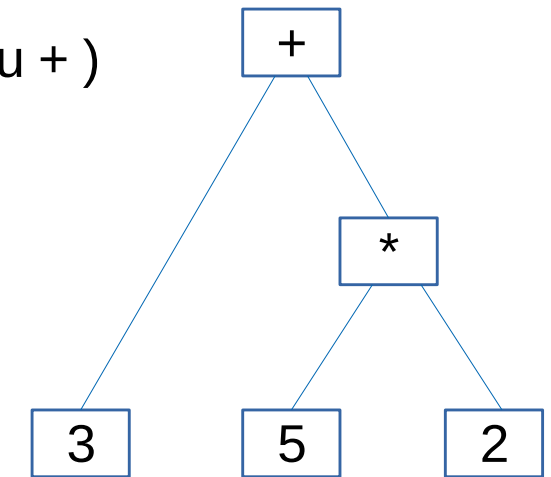
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione:  $3 + 5 * 2$  

parser

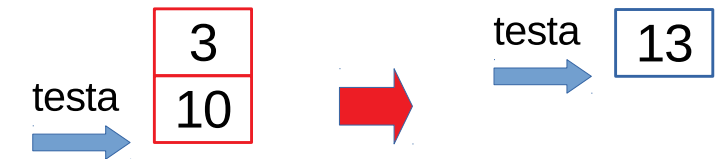
Albero sintattico  
( \* ha precedenza su + )



Procedimento di calcolo:

- salvare i risultati intermedi (come  $5*2$ ) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
  - se sono su una foglia la metto sullo stack
  - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Stack



Esempio:

- ordine posticipato di visita:  $3\ 5\ 2\ *\ +$

# Procedimento di valutazione delle espressioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

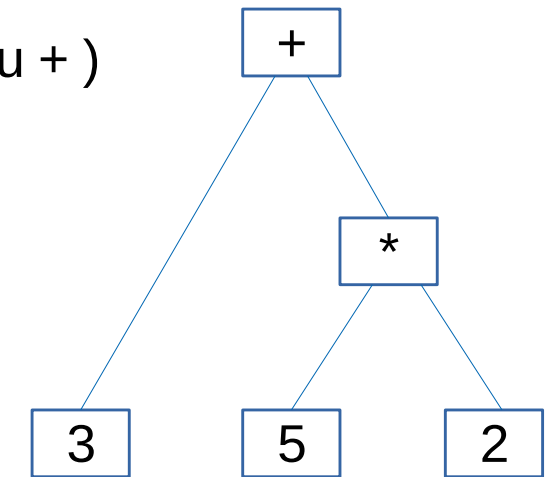
Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

Espressione:  $3 + 5 * 2$  

parser

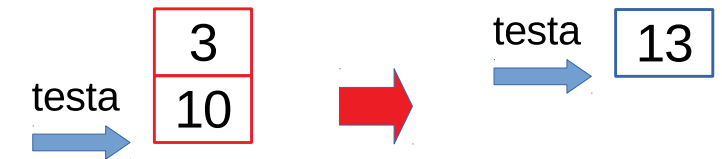
Albero sintattico  
( \* ha precedenza su + )



Procedimento di calcolo:

- salvare i risultati intermedi (come  $5*2$ ) su un piccolo stack
- prima di applicare un operatore bisogna aver calcolato i suoi figli
- visitando l'albero in ordine *posticipato* si ottiene l'ordine giusto di valutazione
  - se sono su una foglia la metto sullo stack
  - se sono su un nodo operazione, i primi due elementi dello stack sono i suoi operandi

Stack



Esempio:

- ordine posticipato di visita:  $3\ 5\ 2\ *\ +$

*Il compilatore, a partire dall'albero sintattico, deve generare un codice che implementa questo procedimento*



# Definizione degli alberi sintattici

- Introduciamo un costruttore per ogni operazione supportata dal linguaggio sorgente
- ogni costruttore corrisponde a un tipo di nodo dell'albero sintattico

```
datatype syntree = co of int    (* le costanti *)  
                  | plus of syntree * syntree  
                  | minus of syntree * syntree  
                  | times of syntree * syntree  
                  | divide of syntree * syntree  
                  | modulus of syntree * syntree
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



# Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



# Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC    of int * int    (* LOADC i c => Ri := c *)
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



# Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC   of int * int    (* LOADC i c  => Ri := c          *)
  | LOADI   of int * int    (* LOADI i j  => Ri := mem(Rj)       *)
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)





# Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC   of int * int    (* LOADC i c  => Ri := c          *)
  | LOADI   of int * int    (* LOADI i j  => Ri := mem(Rj)      *)
  | STOREI  of int * int    (* STOREI i j => mem(Rj) := Ri      *)
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



# Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC  of int * int    (* LOADC i c  => Ri := c          *)
  | LOADI  of int * int    (* LOADI i j  => Ri := mem(Rj)      *)
  | STOREI of int * int    (* STOREI i j  => mem(Rj) := Ri    *)
  | INCR   of int          (* INCR i      => Ri := Ri + 1     *)
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



# Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC  of int * int    (* LOADC i c  => Ri := c          *)
  | LOADI  of int * int    (* LOADI i j  => Ri := mem(Rj)       *)
  | STOREI of int * int    (* STOREI i j  => mem(Rj) := Ri      *)
  | INCR   of int          (* INCR i      => Ri := Ri + 1       *)
  | DECR   of int          (* DECR i      => Ri := Ri - 1       *)
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



# Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC of int * int      (* LOADC i c => Ri := c *)
  | LOADI of int * int      (* LOADI i j => Ri := mem(Rj) *)
  | STOREI of int * int      (* STOREI i j => mem(Rj) := Ri *)
  | INCR of int              (* INCR i => Ri := Ri + 1 *)
  | DECR of int              (* DECR i => Ri := Ri - 1 *)
  | SUM of int * int         (* SUM i j => Ri := Ri + Rj *)
```

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)



# Definizione del linguaggio target

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC of int * int      (* LOADC i c => Ri := c *)
  | LOADI of int * int      (* LOADI i j => Ri := mem(Rj) *)
  | STOREI of int * int     (* STOREI i j => mem(Rj) := Ri *)
  | INCR of int             (* INCR i => Ri := Ri + 1 *)
  | DECR of int             (* DECR i => Ri := Ri - 1 *)
  | SUM of int * int        (* SUM i j => Ri := Ri + Rj *)
  | SUB of int * int        (* SUB i j => Ri := Ri - Rj *)
  | MUL of int * int        (* MUL i j => Ri := Ri * Rj *)
  | DIV of int * int        (* DIV i j => Ri := Ri / Rj *)
  | MOD of int * int        (* MOD i j => Ri := Ri mod Rj *)
```

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



# Definizione del linguaggio target

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Ovvero le operazioni della macchina astratta che eseguirà il codice oggetto sorgente
- qui ci ispiriamo alle istruzioni di hardware classico

```
datatype instruction
  = LOADC of int * int      (* LOADC i c => Ri := c *)
  | LOADI of int * int      (* LOADI i j => Ri := mem(Rj) *)
  | STOREI of int * int      (* STOREI i j => mem(Rj) := Ri *)
  | INCR of int              (* INCR i => Ri := Ri + 1 *)
  | DECR of int              (* DECR i => Ri := Ri - 1 *)
  | SUM of int * int         (* SUM i j => Ri := Ri + Rj *)
  | SUB of int * int         (* SUB i j => Ri := Ri - Rj *)
  | MUL of int * int         (* MUL i j => Ri := Ri * Rj *)
  | DIV of int * int         (* DIV i j => Ri := Ri / Rj *)
  | MOD of int * int         (* MOD i j => Ri := Ri mod Rj *)
  | HALT
```

Obiettivo: terminare il programma con il risultato in cima allo stack.



## Generazione del codice – funzione codegen

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione  
con type checking

Esempio: un semplice  
compilatore

- Il codice oggetto utilizza tre registri:
  - ◆ R1 come puntatore alla testa dello stack
  - ◆ R2,R3 per calcolare le singole operazioni



## Generazione del codice – funzione codegen

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Il codice oggetto utilizza tre registri:
  - ◆ R1 come puntatore alla testa dello stack
  - ◆ R2,R3 per calcolare le singole operazioni
- La traduzione vera e propria è effettuata da una funzione ausiliaria `translate` che prende in input:
  - ◆ un albero sintattico `tree`
  - ◆ una *continuazione*, ovvero il codice da eseguire *dopo* avere eseguito le operazioni contenute in `tree`





## Generazione del codice – funzione codegen

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- Il codice oggetto utilizza tre registri:
  - ◆ R1 come puntatore alla testa dello stack
  - ◆ R2,R3 per calcolare le singole operazioni
- La traduzione vera e propria è effettuata da una funzione ausiliaria `translate` che prende in input:
  - ◆ un albero sintattico `tree`
  - ◆ una *continuazione*, ovvero il codice da eseguire *dopo* avere eseguito le operazioni contenute in `tree`
- Pertanto la prima chiamata a `translate` gli passerà
  - ◆ l'albero sintattico dell'intera espressione da compilare
  - ◆ la lista di istruzioni `[HALT]`



# Generazione del codice

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
fun codegen tree =
```



# Generazione del codice

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

```
fun codegen tree =  
let  
  (* definizione della funzione translate *)  
  (* R1:  stack pointer; R2:  accumulator *)
```



# Generazione del codice

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
fun codegen tree =  
let  
  (* definizione della funzione translate *)  
  (* R1:  stack pointer; R2:  accumulator *)  
  fun translate (co x) cont =  
    LOADC(2,x) :: INCR(1) :: STOREI(2,1) :: cont
```



# Generazione del codice

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
fun codegen tree =  
let  
  (* definizione della funzione translate *)  
  (* R1:  stack pointer; R2:  accumulator *)  
  fun translate (co x) cont =  
    LOADC(2,x) :: INCR(1) :: STOREI(2,1) :: cont  
  | translate (plus(t1,t2)) cont =  
    translate t1 (  
      translate t2 (  
        LOADI(2,1) :: DECR(1) :: LOADI(3,1) :: SUM(2,3) :: STOREI(2,1) ::  
        cont))  
end
```



# Generazione del codice

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
fun codegen tree =  
let  
  (* definizione della funzione translate *)  
  (* R1:  stack pointer; R2:  accumulator *)  
  fun translate (co x) cont =  
    LOADC(2,x) :: INCR(1) :: STOREI(2,1) :: cont  
  | translate (plus(t1,t2)) cont =  
    translate t1 (  
      translate t2 (  
        LOADI(2,1)::DECR(1)::LOADI(3,1)::SUM(2,3)::STOREI(2,1)::  
        cont))  
  | translate (times(t1,t2)) cont = simile ma con MUL al posto di SUM  
  | translate (minus(t1,t2)) cont = simile ma  
    con SUB(3,2) al posto di SUM(2,3)  
  | translate (divide(t1,t2)) cont = simile ma con DIV al posto di SUB  
  | translate (modulus(t1,t2)) cont = simile ma con MOD al posto di SUB
```



# Generazione del codice

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
fun codegen tree =
let
  (* definizione della funzione translate *)
  (* R1: stack pointer; R2: accumulator *)
  fun translate (co x) cont =
    LOADC(2,x) :: INCR(1) :: STOREI(2,1) :: cont
  | translate (plus(t1,t2)) cont =
    translate t1 (
      translate t2 (
        LOADI(2,1)::DECR(1)::LOADI(3,1)::SUM(2,3)::STOREI(2,1)::
        cont))
  | translate (times(t1,t2)) cont = simile ma con MUL al posto di SUM
  | translate (minus(t1,t2)) cont = simile ma
                                   con SUB(3,2) al posto di SUM(2,3)
  | translate (divide(t1,t2)) cont = simile ma con DIV al posto di SUB
  | translate (modulus(t1,t2)) cont = simile ma con MOD al posto di SUB
in
  translate tree [HALT]
end
```



# Ottimizzazione del codice

- La generazione meccanica introduce operazioni inutili.

```
LOADC 2 3      (* codice generato per 3+5*2 *)
INCR 1
STOREI 2 1
—
LOADC 2 5
INCR 1
STOREI 2 1
—
LOADC 2 2
INCR 1
STOREI 2 1
—
LOADI 2 1
DECR 1
LOADI 3 1
MUL 2 3
STOREI 2 1
—
LOADI 2 1
DECR 1
LOADI 3 1
SUM 2 3
STOREI 2 1
—
HALT
```





## Ottimizzazione del codice – funzione `optimize`

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

- La funzione `optimize` elimina le più comuni operazioni ridondanti
- Itera una funzione ausiliaria `opt1` che esegue un singolo passo di ottimizzazione
- Questo può attivare ulteriori semplificazioni, quindi `optimize` itera `opt1` finchè il codice non può essere ulteriormente ridotto



# Ottimizzazione del codice – funzione optimize

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
local
(* definizione singolo passo di ottimizzazione *)
```



# Ottimizzazione del codice – funzione optimize

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
local
  (* definizione singolo passo di ottimizzazione *)
  fun opt1 [] = []
```



# Ottimizzazione del codice – funzione optimize

[Paradigma funzionale](#)

[ML](#)

[Dichiarazioni e scoping in ML](#)

[Tipi strutturati in ML](#)

[Patterns e matching](#)

[Liste](#)

[Currying](#)

[Funzioni di ordine superiore](#)

[Polimorfismo parametrico](#)

[Encapsulation e interfacce](#)

[Eccezioni e integrazione con type checking](#)

[Esempio: un semplice compilatore](#)

```
local
  (* definizione singolo passo di ottimizzazione *)
  fun opt1 [] = []
  | opt1 (INCR(x)::DECR(y)::cont) =
    let val cont' = opt1 cont in
    if x=y
      then cont'
      else INCR(x)::DECR(y)::cont'
    end
```



# Ottimizzazione del codice – funzione optimize

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

```
local
  (* definizione singolo passo di ottimizzazione *)
  fun opt1 [] = []
  | opt1 (INCR(x)::DECR(y)::cont) =
    let val cont' = opt1 cont in
    if x=y
      then cont'
      else INCR(x)::DECR(y)::cont'
    end
  | opt1 (STOREI(x,y)::LOADI(x1,y1)::cont) =
    let val cont' = opt1 cont in
    if x=x1 andalso y=y1
      then STOREI(x,y)::cont'
      else STOREI(x,y)::LOADI(x1,y1)::cont'
    end
```



# Ottimizzazione del codice – funzione optimize

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

```
local
  (* definizione singolo passo di ottimizzazione *)
  fun opt1 [] = []
  | opt1 (INCR(x)::DECR(y)::cont) =
    let val cont' = opt1 cont in
    if x=y
      then cont'
      else INCR(x)::DECR(y)::cont'
    end
  | opt1 (STOREI(x,y)::LOADI(x1,y1)::cont) =
    let val cont' = opt1 cont in
    if x=x1 andalso y=y1
      then STOREI(x,y)::cont'
      else STOREI(x,y)::LOADI(x1,y1)::cont'
    end
  | opt1 (STOREI(x,1)::DECR(1)::cont) = DECR(1)::(opt1 cont)
```



# Ottimizzazione del codice – funzione optimize

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

```
local
  (* definizione singolo passo di ottimizzazione *)
  fun opt1 [] = []
  | opt1 (INCR(x)::DECR(y)::cont) =
    let val cont' = opt1 cont in
    if x=y
      then cont'
      else INCR(x)::DECR(y)::cont'
    end
  | opt1 (STOREI(x,y)::LOADI(x1,y1)::cont) =
    let val cont' = opt1 cont in
    if x=x1 andalso y=y1
      then STOREI(x,y)::cont'
      else STOREI(x,y)::LOADI(x1,y1)::cont'
    end
  | opt1 (STOREI(x,1)::DECR(1)::cont) = DECR(1)::(opt1 cont)
  | opt1 (c :: cont) = c :: (opt1 cont)
```



# Ottimizzazione del codice – funzione optimize

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

```
local
  (* definizione singolo passo di ottimizzazione *)
  fun opt1 [] = []
  | opt1 (INCR(x)::DECR(y)::cont) =
    let val cont' = opt1 cont in
    if x=y
      then cont'
      else INCR(x)::DECR(y)::cont'
    end
  | opt1 (STOREI(x,y)::LOADI(x1,y1)::cont) =
    let val cont' = opt1 cont in
    if x=x1 andalso y=y1
      then STOREI(x,y)::cont'
      else STOREI(x,y)::LOADI(x1,y1)::cont'
    end
  | opt1 (STOREI(x,1)::DECR(1)::cont) = DECR(1)::(opt1 cont)
  | opt1 (c :: cont) = c :: (opt1 cont)
in
  fun optimize code =
    let val code' = opt1 code in      (* fa 1 passo di ottimizz.*)
    if length(code') = length(code) (* se nessun progresso *)
      then code'                      (* termina *)
      else optimize code'             (* altrimenti riprova *)
    end
end
```



Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Applichiamo l'ottimizzazione alla solita espressione  $3+5*2$  A sinistra il codice non ottimizzato

```
LOADC 2 3
INCR 1
STOREI 2 1
LOADC 2 5
INCR 1
STOREI 2 1
LOADC 2 2
INCR 1
STOREI 2 1
LOADI 2 1
DECR 1
LOADI 3 1
MUL 2 3
STOREI 2 1
LOADI 2 1
DECR 1
LOADI 3 1
SUM 2 3
STOREI 2 1
HALT
```

```
LOADC 2 3
INCR 1
STOREI 2 1
LOADC 2 5
INCR 1
STOREI 2 1
LOADC 2 2
INCR 1
STOREI 2 1
DECR 1
LOADI 3 1
MUL 2 3
STOREI 2 1
DECR 1
LOADI 3 1
SUM 2 3
STOREI 2 1
HALT
```



# Efficacia dell'ottimizzazione

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- Ecco il risultato dell'ottimizzazione
- A sinistra il codice non ottimizzato, a destra quello ottimizzato

```
LOADC 2 3
INCR 1
STOREI 2 1
LOADC 2 5
INCR 1
STOREI 2 1
LOADC 2 2
INCR 1
STOREI 2 1
LOADI 2 1
DECR 1
LOADI 3 1
MUL 2 3
STOREI 2 1
LOADI 2 1
DECR 1
LOADI 3 1
SUM 2 3
STOREI 2 1
HALT
```

```
LOADC 2 3
INCR 1
STOREI 2 1
LOADC 2 5
INCR 1
STOREI 2 1
LOADC 2 2
LOADI 3 1
MUL 2 3
DECR 1
LOADI 3 1
SUM 2 3
STOREI 2 1
HALT
```

Guadagno: 30% di istruzioni in meno



# Combinare le fasi con composizione di funzioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

■ L'operatore  $\circ$  denota la composizione di funzioni

◆  $(f \circ g)(x) = f(g(x))$



# Combinare le fasi con composizione di funzioni

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- L'operatore  $\circ$  denota la composizione di funzioni

$$\blacklozenge (f \circ g)(x) = f(g(x))$$

- Con la composizione è facile definire l'intero processo di compilazione assemblando le diverse fasi

```
- val compile = optimize o codegen o parse;
```

```
val it = fn : string -> instruction list
```



## Conclusioni sull'esempio di compilazione

- La combinazione di costruttori, pattern e definizione per casi rende le trasformazioni del codice sorgente e del codice oggetto particolarmente chiare

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore



## Conclusioni sull'esempio di compilazione

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La combinazione di costruttori, pattern e definizione per casi rende le trasformazioni del codice sorgente e del codice oggetto particolarmente chiare
  - ◆ in Java, che pure è un ottimo linguaggio, ogni nodo dell'albero sintattico sarebbe un oggetto e “leggere” la struttura di pezzi di albero non sarebbe immediato



## Conclusioni sull'esempio di compilazione

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La combinazione di costruttori, pattern e definizione per casi rende le trasformazioni del codice sorgente e del codice oggetto particolarmente chiare
  - ◆ in Java, che pure è un ottimo linguaggio, ogni nodo dell'albero sintattico sarebbe un oggetto e “leggere” la struttura di pezzi di albero non sarebbe immediato
- Inoltre la type inference ci permette di omettere il tipo degli identificatori, producendo un codice più snello
  - ◆ come fosse uno scripting language debolmente tipato
  - ◆ ma senza sacrificare il controllo di tipi forte



## Conclusioni sull'esempio di compilazione

Paradigma funzionale

ML

Dichiarazioni e scoping in ML

Tipi strutturati in ML

Patterns e matching

Liste

Currying

Funzioni di ordine superiore

Polimorfismo parametrico

Encapsulation e interfacce

Eccezioni e integrazione con type checking

Esempio: un semplice compilatore

- La combinazione di costruttori, pattern e definizione per casi rende le trasformazioni del codice sorgente e del codice oggetto particolarmente chiare
  - ◆ in Java, che pure è un ottimo linguaggio, ogni nodo dell'albero sintattico sarebbe un oggetto e “leggere” la struttura di pezzi di albero non sarebbe immediato
- Inoltre la type inference ci permette di omettere il tipo degli identificatori, producendo un codice più snello
  - ◆ come fosse uno scripting language debolmente tipato
  - ◆ ma senza sacrificare il controllo di tipi forte
- Per queste ragioni linguaggi come ML vengono utilizzati per la prototipizzazione rapida di compilatori e interpreti