Programmazione funzionale, una semplice introduzione.

Massimo Maria Ghisalberti - pragmas.org

2015-04-02

Indice

1 Premessa **2**

2 La programmazione informatica **2**
2.1 I Linguaggi di programmazione . . . 2
2.2 OOP vs FP . . . 3

3 La programmazione funzionale **4**
3.1 Le funzioni di prima classe . . . 5
3.1.1 Brevità su OCaml. . . . 6
3.2 Variabili come *valori* e funzioni come *valori* . . . 7
3.2.1 Sui tipi di dati in OCaml . . . 8
3.3 Funzioni come parametri. . . . 8
3.4 Il *currying* . . . 10
3.5 Funzioni parzialmente applicate. . . . 13
3.6 Funzioni ricorsive . . . 15
3.7 Funzioni anonime . . . 16
3.7.1 Lambda Calculus . . . 17
3.8 Organizzare le funzioni. . . . 17
3.9 Pattern matching . . . 19
3.9.1 Altri esempi di pattern matching . . . 20
3.9.2 Considerazioni sulla velocità . . . 22
3.9.3 Tipi varianti (*variants*) . . . 22
3.10 Tail Call Optimization . . . 23
3.11 *Loop* e cicli - Map e Fold . . . 24
3.12 Lazy and Strict . . . 26
3.12.1 La divisione per zero . . . 27
3.12.2 Tipi *Boxed* e *Unboxed* . . . 29
3.13 Memoization . . . 30
3.14 S-Expressions . . . 38
3.15 Velocità di esecuzione . . . 43
3.16 Conclusioni . . . 52

4 Appendici **53**
4.1 Paradigma di programmazione . . . 53
4.1.1 Linguaggi di programmazione . . . 53
4.1.2 Paradigmi più comuni . . . 53
4.1.2.1 Programmazione procedurale . . . 54
4.1.2.2 Programmazione strutturata . . . 54
4.1.2.3 Programmazione modulare . . . 54
4.1.2.4 Programmazione funzionale . . . 55
4.1.2.5 Programmazione object oriented . . . 55

Massimo Ghisalberti - tel.: +39 3939018290 – email: massimo.ghisalberti@gmail.com

4.1.3 Paradigmi meno comuni 56
4.1.4 Multiparadigma 56
4.1.5 Il Futuro 56
4.1.6 Lista linguaggi di programmazione 56

1 Premessa

Cosa è la programmazione dei computer?

Da wikipedia[1]:

La programmazione, in informatica, è l'insieme delle attività e tecniche che una o più persone specializzate, programmatori o sviluppatori (developer), svolgono per creare un programma, ossia un software da far eseguire ad un computer, scrivendo il relativo codice sorgente in un certo linguaggio di programmazione.

Questa la definizione in lingua italiana, quella in inglese è più articolata[2]:

La programmazione informatica è un processo che conduce da una formulazione originale di un problema computazionale a programmi eseguibili per un elaboratore elettronico. La programmazione prevede attività quali l'analisi, lo sviluppare la comprensione del problema, la generazione di algoritmi nonché la verifica dei requisiti degli stessi tra cui la loro correttezza, il consumo di risorse e la loro implementazione in un linguaggio di programmazione. Lo scopo della programmazione è quello di trovare una sequenza di istruzioni che automatizzi lo svolgimento di un compito specifico o risolva un determinato problema.

La definizione in italiano è presa come è, mentre quella in inglese è *ovviamente* tradotta ma una cosa si capisce al volo: per gli italiani la *programmazione* è roba da *persone specializzate* dotate di *capacità tecniche* mentre per lo scrittore in lingua inglese è un *processo cognitivo*.

È interessante come diverse culture affrontino il problema in modo diverso.

Quindi lasciamo perdere e facciamoci una passeggiata che il sole è alto ed è iniziata la primavera. Chi di voi è un *tecnico* o una *persona specializzata*? Io non lo sono.

Insomma *programmatori* si nasce o si diventa?

Una cosa giusto per finire la premessa: chiunque può imparare a disegnare od a suonare uno strumento anche senza diventare poi un artista. A leggere e scrivere, anche se a martellate per alcuni, impariamo tutti.

2 La programmazione informatica

Come detto giustamente dallo scrittore in inglese della premessa, la *programmazione informatica* è un *processo* ovvero un susseguirsi di attività *cognitive* ed *implementative*. Insomma è come l'*arte concettuale*, idea e realizzazione della sua rappresentazione. Vedetela in questo modo: ognuno di noi pensa, solo che ad alcuni manca la *disciplina del pensare*.

Programmare è un *modo di pensare* è un *approccio al problema* e non solo una serie di *capacità tecniche* (*skill*), è principalmente un *processo cognitivo*. Certo, qualcosa di *tecnico* va conosciuto, ma quello si impara.

La programmazione informatica non è un'arte oscura ed arcana che rasenta la magia, ma solo una delle tante manifestazioni che fanno dell'uomo quello che è o che dovrebbe essere: *un animale razionale dotato di immensa curiosità*.

2.1 I Linguaggi di programmazione

Di linguaggi di programmazione ne esistono a bizzeffe, come potete controllare nella lista e non completa, dei paradigmi di programmazione[4.1] in appendice. La cosa interessante di tutti questi è che rispecchiano il modo di pensare delle persone che li hanno ideati, la loro *forma mentis*.

Ogni linguaggio cerca, a suo modo, di risolvere al meglio alcune problematiche computazionali seguendo degli schemi. Questi possono essere più o meno *flessibili* ma nessuno è perfetto, alcuni sembrano più intuitivi di altri o più familiari ma non sempre a *colpo d'occhio* si è nel giusto.

[1] http://it.wikipedia.org/wiki/Programmazione_%28informatica%29
[2] http://en.wikipedia.org/wiki/Computer_programming

Questo è il caso della *programmazione funzionale* che a prima vista pare una *cosa aliena* ed oltremodo complessa. Si è più propensi all'**OOP** (Object Oriented Programming) visto che ci hanno spiegato che con questo modello possiamo imitare gli *oggetti reali* (non vi vorrei tediare con le gerarchie padre-figlio di albero -> albero di pere o veicolo -> carretto -> automobile). All'OOP poi, la maggior parte dei programmatori associa la *programmazione imperativa* che piace a tutti. A chi non piace comandare? Ci fa sentire dei piccoli *dio del computer*: io ordino tu esegui pedissequamente: fai qui e fai là, se questo fai quello, loop!

I programmatori poi sono gente strana, mettono su discussioni interminabili, spesso sul nulla, peggio dei filosofi più noiosi. Si continua a *girare il dito nella piaga* per affermare che un editor di testo è migliore di un altro (io appartengo alla Church of Emacs, ovviamente), interminabili discussioni sul colore della evidenziazione della sintassi o sul è *meglio questo di quest'altro*.

Si perdono sui *massimi sistemi* della programmazione ma spesso non sono in grado di scrivere tre righe per un semplice FizzBuzz[3].

Una delle *guerre di religione* che va per la maggiore (almeno da qualche anno) è quella tra *OOP* e *FP (functional programming)*. Ognuno accusa l'altro dei peggiori crimini: OOP è responsabile di tutti i difetti del software odierno mentre FP è indecentemente complicata e con la *puzza al naso* oltre ad essere lenta in esecuzione. Si scontrano *oggetti* contro *funzioni* in duelli all'ultimo sangue ed all'arma bianca.

Ad onor del vero, gli sviluppatori FP un po' di *puzza al naso* ce l'hanno davvero in parecchi e capita di trovare codice scritto più per essere *ganzi* che per necessità. Molti linguaggi funzionali permettono di essere parecchio stringati e quindi di comporre codice decisamente oscuro, almeno a prima vista, anche per chi quel linguaggio lo conosce.

Cominciamo a levarci *la puzza al naso*.

2.2 OOP vs FP

Come ho detto sopra molti programmatori considerano OOP ed FP mutualmente esclusivi, immaginate due diverse fazioni che vivono nei loro castelli fortificati che si danno dell'*esaltato* o dello *sfigato*. Insomma una banda di tacchini schiamazzanti.

Molti di questi non hanno neanche capito di cosa parlano e si bombardano a colpi di parentesi (graffe per gli uni e tonde per gli altri).

FP, come dice il nome, ha le *funzioni* ed ogni programmatore ferrato vi dirà per prima cosa che le sue *funzioni* sono *first-class object* ed è cosa realmente vera, dimenticando o non sapendo che anche in **Smalltalk** (che è un linguaggio orientato agli oggetti e non funzionale) lo sono.

OOP ha gli *oggetti* e gli *oggetti* sono composti di dati e metodi (funzioni nel gergo OOP), ma gli oggetti non sono *strutture dati*, le usano semplicemente. Gli *oggetti* sono solo dei delimitatori di ambiente (*namespace*) e delle *borse di funzioni*. Le strutture contengono i dati mentre gli oggetti li usano (almeno dovrebbe essere così). In ogni caso qualunque programma è dati e funzioni. (Algoritmi+Strutture Dati=Programmi - Niklaus Wirth, 1987).

I metodi sono diversi dalle funzioni? C'è differenza estrema nello scrivere una chiamata a funzione con:

`obj.fun()` o `fun(obj)` o `fun obj` o `(fun obj)`

Se è solo una questione di sintassi è meglio andare a coltivare le cipolle.

La differenza reale è nella *disciplina* che le metodologie impongono, il resto è solo preferenza personale. Le due discipline non si sovrappongono e niente vieta di usarle insieme quando serve.

La forza di FP è che *non ha stati mutevoli*, le variabili non esistono e non ha assegnamenti. Ogni cosa è costante e prevedibile, matematicamente corretta e vedremo in seguito cosa significa.
Quindi non si possono fare cose come salvare un file? Certo, ma FP lo rende leggermente *complicato* e *responsabile*, ci obbliga a *renderci conto* di quello che stiamo facendo: *la legge non ammette ignoranza*.

La non mutabilità di stato rende pressoché impossibile che le variabili vengano sovrascritte o alterate da codice esecutivo concorrente rendendo un programma abbastanza *sicuro* in caso di più thread o core multipli.

OOP ha la sua forza nei *puntatori a funzione*.
OOP è nato dalla crisi della *programmazione procedurale* nel risolvere il problema del *polimorfismo* e cioè della

[3] FizzBuzz è un gioco per bambini di gruppo per allenarsi alla divisione, viene usato anche per discriminare i programmatori ai colloqui di lavoro (pare che statisticamente se ne eliminino parecchi). Si tratta di scrivere un programma che stampa i numeri da 1 a 100, ma per i numeri multipli di 3 deve scrivere *Fizz* e per quelli multipli di 5 *Buzz*, mentre per quelli multipli di 3 e 5 contemporaneamente *FizzBuzz*.

invocazione della stessa funzione per tipi di dati diversi o con diversi argomenti. OOP fa questo ed a parte la *fuffa* degli *oggetti del mondo reale* o del *programmare vicino a come si pensa*, incapsula la complessità dei *puntatori a funzione*.
Chi ha programmato in C saprà di cosa si parla e dell'incubo terribile che sia gestirli al meglio.

Il *polimorfismo* ha un grosso beneficio: la possibilità di poter sostituire una funzionalità con un'altra. Questo ha permesso di sviluppare architetture del software a *plug in* o *moduli inseribili*. È la possibilità di disaccoppiare le dipendenze tra il codice sorgente ed il codice esecutivo.

Sono incompatibili? Non direi e ne è la prova come in questi ultimi anni si siano moltiplicati i linguaggi multi-paradigma, cioè quelli che sono in grado di usare più approcci alle soluzioni dei possibili problemi.

Nessun paradigma è migliore di un'altro, ma solo più adatto al caso specifico.

Oggi con l'affermazione del *calcolo parallelo* e dove anche i nostri telefoni sono equipaggiati con CPU *multicore*, la possibilità di avere linguaggi che non mutano il proprio stato è un grosso vantaggio, come vedremo in seguito.

3 La programmazione funzionale

Si chiama *funzionale* perché il suo concetto di base risiede nella *funzione*, ma nel senso *matematico* del termine. Le funzioni dei linguaggi procedurali o imperativi sono realmente delle *procedure* piuttosto che delle *funzioni* e svolgono in questo senso delle pure direttive di calcolo. La *funzione matematica* invece non esegue calcoli ma si limita a *mappare valori* e viene definita infatti, come una: *trasformazione* o *mappa* od *operatore*; essa applica un argomento di un certo valore dall'interno di un *dominio* verso un'altro *dominio*, detto *codominio*:

$$y = f(x) \tag{1}$$

La funziona applica un insieme X detto **dominio** ad un insieme Y detto **codominio** tramite la relazione:

$$f : X \to Y \tag{2}$$

quindi ad ogni elemento dell'insieme X associa uno ed un solo elemento dell'insieme Y.
Il valore x preso nel dominio X viene detto *argomento* o *valore della variabile indipendente*, mentre y diventerà il *valore della variabile dipendente* compreso nel codominio Y.

Complicato? Facciamola semplice. Quante volte diciamo che una cosa è in funzione di un'altra? Cosa vogliamo intendere? La cosa che mi viene per prima in mente è la relazione tra causa ed effetto: se succedesse questo allora quest'altro…

La *funzione* è solo un modo per mappare o trasportare valori ed un'altra sua caratteristica è che ha un solo argomento e restituisce un solo valore.

$$y = f(x, z) \tag{3}$$

Questa non è diversa da quella di sopra, l'argomento adesso è una *coppia di valori* ma è sempre un solo argomento. I linguaggi funzionali cercano di mimare per quanto possibile questo *meccanismo formale*, che anche se può sembrare rigido all'atto pratico porta notevoli benefici.

Non tutti i linguaggi funzionali sono uguali ma hanno dei principi comuni. Sono divisi grossolanamente in *puri* ed *impuri* in base al fatto se permettano o meno degli *effetti collaterali* (side effects), cioè la possibilità di alterare il loro stato di esecuzione. Idealmente un linguaggio *puro* dovrebbe essere preferibile ma i nostri programmi, molto spesso, hanno bisogno di alterare il loro stato di esecuzione (salvare un file per esempio) e pur non essendo impossibile farlo per un linguaggio *puro* (come Haskell) la cosa potrebbe diventare parecchio *cerimoniosa*.

Credo che per attrarre persone verso la **FP** ci sia bisogno di meno *purezza*, specialmente per chi è abituato a fare il lavoro sporco con la **OOP** od i linguaggi procedurali ed imperativi.

Per i prossimi esempi userò in particolare **OCaml** [4] (versione 4.02.1 al momento) ed eventualmente citerò **F#** che fa parte della stessa famiglia e ne implementa un sottoinsieme da *buon cugino*, con qualche puntata in **Scala** o altri. **Haskell** lo lascio nella sua torre d'avorio per adesso.

[4] http://ocaml.org

3.1 Le funzioni di prima classe

Ora sappiamo che le funzioni del nostro linguaggio sono in relazione diretta con il concetto matematico di funzione. Quindi:

$$addone(x) = x + 1 \tag{4}$$

è una funzione che dato un valore x nel dominio dei numeri interi restituisce un nuovo valore che è pari alla somma di $x + 1$.

Se lo scriviamo in Ocaml:

```
let addone x = x + 1
```

mentre in F#

```
let addone x = x + 1
```

è perfettamente uguale. Due identiche sintassi per due linguaggi diversi? In realtà non sono proprio diversi: F# (*Fsharp*) è un sottoinsieme di OCaml sviluppato nei *Microsoft Labs* e poi approdato come linguaggio *mainstream* nella linea *Visual Studio* (l'ambiente di sviluppo di Microsoft). Oggi l'F# è un linguaggio *Open Source* e multi piattaforma in conseguenza del progetto **Mono**[5] (una implementazione del *.NET Framework* su piattaforme diverse da Microsoft Windows[tm]).

Per correttezza va indicato che Mono è un progetto potenzialmente problematico a livello di licenza[6] e la sua adozione, specialmente su Linux, è fonte di discussioni.

Questi altri esempi in diversi linguaggi funzionali:

Scala

```
def addone(x : Int) = x + 1
```

Clojure (un dialetto Lisp)

```
(def addone (fn [x] (+ x 1)))
```

Haskell (GHCi)

```
let addone x = x + 1
```

Tornando alla nostra *funzione matematica*, la sua caratteristica è che restituirà sempre lo stesso risultato in uscita per un dato valore in entrata e di conseguenza non avrà *effetti collaterali*.

Questo fatto ci permette di poter seguire e predire meglio il flusso dei dati. Nella programmazione imperativa si immagina che le funzioni *facciano qualcosa* e siano inerentemente delle *procedure* (in Pascal, un linguaggio imperativo, per esempio esiste una distinzione tra funzione e procedura), ci si aspetta che abbiano dei *side effects*.

La funzione matematica è invece un semplice *mapping*, un mappare un dominio in un altro correlato.

```
#include <stdio.h>

void main() {

  int add_one(int input) {
    switch (input) {
    case 0: return 1;
    case 1: return 2;
    case 2: return 3;
    // ... altri casi a seguire
    }
  }

  printf("%d\n", add_one(2));
}
```

[5] http://www.mono-project.com/

[6] http://www.fsf.org/news/dont-depend-on-mono http://www.fsf.org/news/2009-07-mscp-mono

Questo piccolo programma in **C** illustra come potrebbe essere un semplice *mapping*. La funzione è uno `switch` che mappa un valore in un altro valore: nel caso `input` sia 0 allora ritorna 1, se 1 ritorna due e così via. Non c'è calcolo ed solo una tabella di ispezione (*lookup*).
L'input e l'output sono logicamente due cose differenti e valutare la funzione non può in nessun modo portare ad effetti indesiderati su di loro.

Se torniamo alle espressioni matematiche:

$$a = 10 \tag{5}$$

$$b = a + 1 \tag{6}$$

non ci aspettiamo che a abbia un valore diverso da 10 e che b sia diversa da 11.

La funzione matematica è *relazione* piuttosto che *computazione*, perché matematicamente i *domini* esistono indipendentemente da tutto.

Questo tipo di funzioni vengono dette *pure* ed hanno delle caratteristiche interessanti:

1. Le si possono valutare in qualsiasi ordine.
2. La funzione può essere valutata una volta soltanto e mantenere solo il risultato (*memoization*).
3. La funzione può essere valutata a tempo di compilazione (inferenza di tipo o valore).
4. La funzione può essere valutata solo quando se ne ha bisogno (*lazy*) ed essere sicuri che si avrà quel risultato.
5. Possiamo fa calcolare quella funzione su tutti i processori che abbiamo senza preoccuparci di sovrascriverci i dati.

Ci sono senza dubbio, molti vantaggi.

A prima vista però ci sono anche degli svantaggi notevoli: un solo valore in entrata ed un solo valore in uscita, anche la *immutabilità* potrebbe creare problemi. In teoria tutto questo è *molto bello* ma se voglio avere funzioni che prendono più di un parametro? A volte c'è bisogno anche di alterare lo stato o di assegnare variabili. Vedremo poi come *ovviare* a queste problematiche e volgerle a nostro pro.

Questo testo non vuole essere un corso di programmazione, quanto una modesta introduzione ai concetti che pervadono il paradigma funzionale utilizzando dei piccoli esempi in un linguaggio che reputo estremamente potente ed espressivo, ma abbastanza *pragmatico* ed *impuro*, da impedire a chi cerca di avvicinarsi di *perdersi nei concetti* ed abbandonare l'impresa.

3.1.1 Brevità su OCaml.

OCaml[4] è un linguaggio che ha sulle spalle molti anni di sviluppo ed è focalizzato nella *espressività*, *sicurezza* e *velocità di esecuzione*. Nasce come discendente di una famiglia di linguaggi detti **ML** (*meta-language*). Nella sua implementazione è un diretto derivato del CAML (*Categorical Abstract Machine Language*) con una estensione per la programmazione in stile OOP (O'Caml o Object Caml o OCaml) ed è un software Open Source creato nel 1996 all'INRIA (*Institut national de recherche en informatique et en automatique*).
Le sue caratteristiche sono: un sistema di tipi statico e ad *inferenza* (il tipo dei valori è valutato dal compilatore analizzando il codice), polimorfismo parametrico, *algebraic data types*, *pattern matching*, *tail recursion*, funtori (moduli parametrizzati), *closure* lessicali, gestione delle eccezioni, *garbage collector* incrementale, sistema OOP con ereditarietà multipla, classi virtuali e parametriche, sistema strutturale dei tipi degli oggetti (o *property-based type system*, la compatibilità di tipo è valutata in base alla *firma* o *signature* dei metodi e delle proprietà piuttosto che dalla ereditarietà dichiarata).

Il compilatore è in grado di generare sia codice nativo altamente ottimizzato (con velocità paragonabili al C/C++) che *bytecode* per un runtime più generale. Sono in sviluppo altri target di compilazione come per il *javascript* che è completo e stabile per la produzione o per la JVM (*Java Virtual Machine*), sufficientemente stabile.

Massimo Ghisalberti - tel.: +39 3939018290 – email: massimo.ghisalberti@gmail.com

Ha a disposizione un **REPL** (*Read eval and print loop*) per test rapidi o valutazioni del codice. Un REPL per OCaml è disponibile online: http://try.ocamlpro.com/. Un possibile difetto è il non supporto diretto per il *symmetrical multiprocessing* a causa del suo sistema di *garbage collector*. Ci sono comunque molte librerie (Async[7] e LWT[8] le più importanti) che permettono la programmazione parallela e sono allo studio dei runtime ottimizzati.

È stato sviluppato un *package manager* particolarmente sofisticato per la gestione delle librerie aggiuntive, chiamato OPAM[9].

3.2 Variabili come *valori* e funzioni come *valori*

Riprendendo il semplice esempio di prima: `let x = x + 1`, abbiamo stabilito che è una funzione che accetta un valore da un dominio di input e che usa il nome `x` per rappresentarlo. Questo meccanismo viene chiamato *binding* e cioè *collegamento* e quindi il nome `x` è *collegato* al valore di input. È importante capire come non ci sia assegnamento, `x` non è una variabile ma solo un fattore mnemonico che rappresenta direttamente un valore, non potrà più cambiare una volta collegato al valore.

Per questo motivo non esistono *variabili* ma solo *valori*. Insomma pensatela come una costante di un linguaggio imperativo per farla più semplice.

Se estendiamo il concetto dei *valori* si può notare come anche il nome della funzione è un *valore* in sé. Il nome della funzione `add_one` è collegato ad una espressione che equivale a `x -> x + 1` ed ha come firma (*type signature*):

```
val add_one : int -> int = <fun>
```

Questo è il tipo della funzione `add_one`, un valore (*val*) collegato ad una funzione che ha un valore in entrata nel dominio degli `int` e restituisce un valore in un codominio di `int`. È puro *valore*, un concetto che va ripetuto *ad nauseam* perché è uno dei fondamenti (forse il principale) della programmazione funzionale.

Se scriviamo:

```
let plus_one = add_one;;
(* signature: val plus_one : int -> int = <fun> *)
```

possiamo vedere come `plus_one` sia un nuovo nome collegato alla stessa funzione a cui era collegato `add_one`. Per semplificare, errando, possiamo dire che in un certo senso è come se il nome della funzione fosse un *puntatore a funzione* in C/C++ senza i problemi che questo comporterebbe.

```
plus_one 5;;
(* result: - : int = 6 *)
```

Per controllare molti degli esempi potete usare uno dei REPL (UTop[10]) disponibili per OCaml.

Per avere dei valori semplici possiamo scrivere:

```
let num = 5;;
(* signature: val num : int = 5 *)
```

ed avremo `num` che è collegato al valore 5, senza la freccia che indica il rapporto tra `input` ed `output` come nelle funzioni. Abbiamo definito una *costante* o *valore*.

Un semplice *mapping* su più valori può essere attuato in questa modo:

```
let (a,b) = 1,2;;
(* signature: val a : int = 1 *)
(* signature: val b : int = 2 *)

a;;
(* result: - : int = 1 *)

b;;
(* result: - : int = 2 *)
```

[7] http://janestreet.github.io/
[8] http://ocsigen.org/lwt/
[9] http://opam.ocaml.org/
[10] https://github.com/diml/utop

I valori semplici e le funzioni sono talmente uniti da poter essere *quasi* intercambiabili, sono ambedue valori collegati a nomi mnemonici tramite la parola chiave `let` (permettere).

Se traduciamo l'espressione `let x = 5` in *permetti che x sia collegata al valore 5* forse diventa tutto più chiaro.

La sottile differenza tra un valore ed una funzione è che la funzione è un valore intorno ad un dominio di input ed uno di output, ma per il resto è identica ed utilizzabile nella stessa maniera: una funzione può essere input per altre funzioni.

Il dominio di input deve essere sempre applicato all'argomento della funzione, per esempio se vogliamo definire una funzione costante:

```
let num = fun () -> 5;;
(* signature: val num : unit -> int = <fun> *)
```

o in maniera equivalente:

```
let num() = 5;;
(* signature: val num : unit -> int = <fun> *)
```

vediamo come il dominio di input sia questa volta chiamato `unit` e ciò comporta che la funzione deve essere risolta fornendo `unit` come argomento.

Invocare semplicemente:

```
num;;
(* result: - : unit -> int = <fun> *)
```

abbiamo come risultato la funzione stessa. Dobbiamo fornire l'argomento:

```
num ();;
(* result: - : int = 5 *)
```

`unit` è un tipo speciale e corrisponde a `()` e possiamo vederlo come il `void` dei linguaggi della famiglia **C** (C/C++/C#...). Al contrario di `void` però ha un valore ed un significato ben preciso: esso fa parte di un dominio di input o di uno di output (è un tipo reale ed un valore reale) mentre un `void` come valore di ritorno di una funzione **C** fa si che qualunque esso sia venga ignorato (di fatto in **C** è una indicazione per il compilatore che vogliamo trattare quella funzione come una *procedure* in Pascal, un routine con effetti collaterali mentre se usato come argomento di input indica esplicitamente l'assenza di parametri).

3.2.1 Sui tipi di dati in OCaml

OCaml è un linguaggio *strettamente tipizzato* e come molti linguaggi funzionali (e non) ha un meccanismo di *inferenza* per determinare il tipo del dominio di appartenenza del valore. Come appartenente alla famiglia **ML** utilizza un meccanismo detto di **Hindley–Milner** [11] (ML è stato sviluppato da Robin Milner[12] presso l'Università di Edimburgo alla fine degli anni '70).

Per indicare esplicitamente il tipo si può scrivere in questo modo:

```
let add_one (x : int) : int = x + 1;;
(* signature: val add_one : int -> int = <fun> *)
```

Per specificare il tipo si usano i `:` e per il parametro servono le parentesi. Il sistema però è particolarmente efficiente e non richiede quasi mai delle annotazioni esplicite, i casi sono abbastanza rari.

3.3 Funzioni come parametri.

Abbiamo detto più volte che le funzioni sono valori e che possono essere utilizzati e passati come valori ad altre funzioni. Questo meccanismo è alla base della programmazione funzionale che sostanzialmente si basa sul creare piccole funzioni da collegare le une alle altre per svolgere un compito: il *chaining*. Come anelli di una catena le funzioni si passano valori in un flusso controllato.

[11] https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system
[12] http://en.wikipedia.org/wiki/Robin_Milner

Le funzioni che prendono funzioni come parametri sono dette **higher-order function** o **functor** (funtori). Un *funtore* in matematica è un tipo di *mappa* tra *categorie* (nella teoria delle categorie) che ne conserva le caratteristiche strutturali. In informatica è generalmente un *oggetto che può essere usato come una funzione*, Smalltalk[13] fu uno dei primi linguaggi ad implementare i *funtori*. Il loro utilizzo è molto diffuso specialmente nel sistema di *callback* o *delegati* della *programmazione ad eventi* nelle interfacce grafiche. Molti linguaggi hanno costrutti più o meno semplici per poteri utilizzare e definire.

Nella programmazione funzionale è più o meno la norma utilizzarli e nel senso più matematico del termine. Un *funtore* è una funzione che accoglie una funzione come valore e la applica ad un tipo restituendo una relazione:

```
List.fold_left (+) 0 [1;2;3;4];;
(* result: - : int = 10 *)
```

Questo esempio usa la funzione `fold_left` per sommare tutti gli elementi di una lista. La funzione prende tre parametri: una funzione da applicare (in questo caso +) un accumulatore (inizializzato a zero) ed una lista (`[1;2;3;4]`). Inizierà applicando `(+)` all'accumulatore ed al primo elemento della lista, poi passerà al secondo e così via.

```
List.map (fun e -> "ciao " ^ e) ["max"; "ele"; "tom"];;
(* result: - : bytes list = ["ciao max"; "ciao ele"; "ciao tom"] *)
```

La funzione `map` un *caposaldo* delle funzioni che iterano sulle liste. In questo caso si è usata una *funzione anonima* utilizzando la parola chiave `fun`, si potrebbe anche scrivere definendo prima una funzione che concatena le stringhe:

```
let saluta s = "ciao " ^ s;;
(* signature: val saluta : bytes -> bytes = <fun> *)

List.map saluta ["max"; "ele"; "tom"];;
(* result: - : bytes list = ["ciao max"; "ciao ele"; "ciao tom"] *)
```

Le *funzioni anonime* sono molto utili quando servono piccole funzioni localmente da non giustificare *binding* di visibilità maggiore.

Definiamo una funzione che accetta un tipo funzionale:

```
(* definiamo una funzione che aggiunge uno ad un numero *)
let plus_one x = x + 1;;
(* signature: val plus_one : int -> int = <fun> *)

(* definiamo il funtore *)
let mult_by_fun_value fn x = fn x * x;;
(* signature: val mult_by_fun_value : (int -> int) -> int -> int = <fun> *)

mult_by_fun_value plus_one 3;;
(* result: - : int = 12 *)

(* usando una funzione annonima *)
mult_by_fun (fun x -> x + 1) 3;;
(* result: - : int = 12 *)
```

Credo che basti a questo punto lasciare andare la fantasia sulle possibili implicazioni.

Non è finita però. Una funzione essendo un *tipo primitivo* è restituibile da una funzione. Funzioni che prendono funzioni come parametri e restituiscono funzioni.

Definiamo una funzione che serva da generatore di somma:

```
let adder num_to_add = (+) num_to_add;;
(* signature: val adder : int -> int -> int = <fun> *)
```

La funzione `adder` prende un `int` come parametro e restituisce una funzione che lo mappa in una coppia di `int`.

Una cosa che si nota è `(+)`, questo è un semplice operatore di somma come in tutti i linguaggi o nell'aritmetica elementare. In molti linguaggi è semplicemente una *funzione speciale* e può essere invocato funzionalmente. In OCaml si racchiude tra parentesi ed è in posizione *prefissa* come qualunque invocazione di funzione. Qui si entra anche in un'altra particolarità però: le *funzioni parzialmente applicate* che vedremo in seguito.

[13] http://it.wikipedia.org/wiki/Smalltalk

```
(+);;
(* result: - : int -> int -> int = <fun> *)
(* possible signature: val + : int -> int -> int = <fun>  *)

(+) 1 2;;
(* result: - : int = 3 *)
```

La funzione `adder` quindi restituisce una funzione *parzialmente applicata* e la possiamo usare in questo modo:

```
let add_one = adder 1;;
(* signature: val add_one : int -> int = <fun> *)

let add_ten = adder 10;;
(* signature: val add_ten : int -> int = <fun> *)

add_one 5;;
(* result: - : int = 6 *)

utop # add_ten 2;;
(* result: - : int = 12 *)
```

Ho detto prima che OCaml ha un supporto per la *inferenza di tipo* molto sofisticata e che quindi non sempre serve indicare quale sia il tipo di un valore. In questo caso il meccanismo di inferenza capisce cosa sia `fn`, è abbastanza chiaro che sia una funzione che prende un `int` e restituisce un `int`:

```
let eval_func_with_four_then_add_five fn = fn 4 + 5;;
(* signature: val eval_func_with_four_then_add_five : (int -> int) -> int = <fun> *)
```

Il problema potrebbe sorgere qui:

```
let eval_func_with_four fn = fn 4;;
(* signature: val eval_func_with_four : (int -> 'a) -> 'a = <fun> *)
```

Leggendo la *signature* del tipo si capisce che `fn` sia una funzione che pende un `int` ma cosa ritorna? Cosa è `` `a ``?

Questo è il modo con cui OCaml permette il polimorfismo di tipo, `` `a `` significa qualunque tipo. In questo caso però vorremmo essere un po' più espliciti e non ci serve che la funzione sia polimorfa, deve prendere un `int` e ritornare un `int`:

```
let eval_func_with_for_four (fn:int -> int) = fn 4;;
(* signature: val eval_func_with_for_four : (int -> int) -> int = <fun> *)
```

Usando la stessa annotazione di tipo dei valori indichiamo al compilatore cosa vogliamo che sia.

Qui prendiamo un `int` e ritorniamo un tipo `string`:

```
let eval_with_four_and_print (fn:int -> string) = print_string (fn 4);;
(* signature: val eval_with_four_and_print : (int -> bytes) -> unit = <fun> *)

let string_of_num x = string_of_int x;;
(* signature: val string_of_num : int -> bytes = <fun> *)

eval_with_four_and_print string_of_num;;
(* result: 4- : unit = () *)

eval_with_four_and_print (fun x -> string_of_int x);;
(* result: 4- : unit = () *)
```

3.4 Il *currying*

Fino ad adesso si è detto che le funzioni sono come le funzioni matematiche: prendono un valore in input da un dominio e restituiscono un valore in output in un altro dominio. Nella vita reale però, servono spesso più parametri in input e quindi?

Molti linguaggi funzionali usano un metodo detto **currying**. Il nome viene da un matematico statunitense Haskell Curry[14] che riprendendo i lavori di Moses Schönfinkel[15] sulla *logica combinatoria* ne formulò la base teorica (un altro matematico che ne influenzò le idee fu Gottlob Frege[16]). I due linguaggi di programmazione *Haskell* e

[14] http://en.wikipedia.org/wiki/Haskell_Curry
[15] http://en.wikipedia.org/wiki/Moses_Sch%C3%B6nfinkel
[16] http://en.wikipedia.org/wiki/Gottlob_Frege

Curry sono intestati a suo nome.

Il meccanismo non è complicato ed è simile alla soluzione su carta di una funzione con parametri multipli:

$$f(a,b) = b/a \tag{7}$$

Se vogliamo valutare $f(4,8)$ cominciamo a sostituire a con 4 ottendo $f(4,b)$ e cioè una funzione $g(b)$ che è definita come:

$$g(b) = f(4,b) = b/4 \tag{8}$$

Sostituendo 8 a b il gioco è fatto:

$$g(8) = f(4,8) = 8/4 = 2 \tag{9}$$

Insomma visto che le funzioni sono valori, una funzione con parametri multipli può essere ridotta ad un equivalente numero di funzioni con un solo parametro che applicano in sequenza il risultato della funzione precedente.

```
let multi_params a b c d e f = a + b + c + d + e + f;;
(* signature: val multi_params : int -> int -> int -> int -> int -> int -> int = <fun> *)
```

Abbiamo una funzione che in cascata somma valori decomponendosi in più funzioni con un input soltanto. Come lo calcolereste sulla carta questo: `2 + 2 + 2 + 2 + 2`

$2 + 2 \rightarrow 4$
$4 + 2 \rightarrow 6$
$6 + 2 \rightarrow 8$
$8 + 2 \rightarrow 10$

Credo sia chiaro a questo punto.

Non tutti i linguaggi funzionali sono basati su questo meccanismo, lo consentono alcuni ma non ne sono basati. OCaml e Haskell si, ogni funzione con parametri multipli è decomposta tramite *currying*.

Non va confuso il *currying* con le *funzioni parzialmente applicate* che abbiamo citato prima. Sono due meccanismi diversi che coesistono, la differenza è che una *funzione parzialmente applicata* ritorna sempre un valore (sia esso puro o funzionale) mentre una funzione *curried* propone sempre una nuova funzione nella catena di *currying*.

Una *funzione parzialmente applicata* si riferisce solo al processo di formalizzare il valore di un certo numero di parametri fornendo una funzione con un numero di argomenti (*arity*) minore.

Guardiamo questo esempio:

```
let sum1 (a,b) = a + b;;
(* signature: val sum1 : int * int -> int = <fun> *)

let sum2 a b = a + b;;
(* signature: val sum2 : int -> int -> int = <fun> *)
```

Le due funzioni `sum1` e `sum2` sono definite in modo diverso. La prima, `sum1`, è definita in una maniera più convenzionale per chi proviene da linguaggi imperativi; ho usato le parentesi. Se la invoco:

```
sum1 1 2;;
(* Error: This function has type int * int -> int It is applied to too many arguments; maybe you forgot a `;'. *)

sum1 (1,2);;
(* result: - : int = 3 *)
```

nel primo caso, chiamandola come abbiamo fatto fino ad adesso, senza parentesi, il compilatore si lamenterà segnalando un errore. Nel secondo caso e con le parentesi eseguirà il compito.

Se guardiamo la *signature* del tipo notiamo che è `int * int -> int` e cioè una *tupla* (*ennupla* o *n-upla*: elenco ordinato di valori), quindi praticamente abbiamo detto che la nostra funzione `sum1` non ha *arity* pari a due ma è soltanto a uno. Ha cioè, come argomento una coppia di valori e non più valori. Il *currying* è preservato.

L'errore dice questo: *It is applied to too many arguments*; per questo motivo sono richieste le parentesi.

Si possono usare i due approcci nelle definizioni, ma deve esserne compresa bene la differenza.

Si potrebbe pensare che il *currying* sia un procedura lenta e che impegni troppo calcolo; questo non è il caso di OCaml dove certi meccanismi sono particolarmente ottimizzati e le continue chiamate a funzione non hanno, per esempio, problemi di allocazione nello *stack* come nei linguaggi imperativi.

Ci sono alcune problematiche nelle funzioni *curried*, la prima è che se si forniscono meno parametri di quelli dichiarati non si hanno errori ma si otterrà una funzione parzialmente applicata in un contesto dove si aspetta un semplice valore:

```
let add x y = x + y ;;
(* signature: val add : int -> int -> int = <fun> *)

add 1;;
(* result: - : int -> int = <fun> *)
```

Un altro problema sono le funzioni che non accettano parametri. Le funzioni devono avere sempre un parametro in `input` quindi come abbiamo già incontrato dobbiamo dichiararla:

```
let one_plus_one () = 1 + 1;;
(* signature: val one_plus_one : unit -> int = <fun> *)
```

altrimenti non dichiarando un parametro `unit` avremmo, come visto in precedenza, un semplice valore.

Se invece forniamo più argomenti del necessario avremo un messaggio di errore: `Error: ... It is applied to too many arguments ...`

A parte Ocaml e Haskell dove il *currying* è la norma, altri linguaggi ne permettono l'uso come alternativa. In **Scala** le funzioni con parametri multipli non sono automaticamente *curried*:

```
// funzione normale con due argomenti
def sum (a:Int, b:Int) = a + b
// signature: sum: (a: Int, b: Int)Int

sum (1,2)
// result: res0: Int = 3

// funzione curried
def sum_curried (a:Int) (b:Int) = a + b
// signature: sum_curried: (a: Int)(b: Int)Int

sum_curried (1) (2);
// result: res1: Int = 3

// funzione curried
def sum_curried2 (a:Int) = (b:Int) => a + b
// signature: sum_curried2: (a: Int)Int => Int

sum_curried2 (1) (2);
// result: res1: Int = 3
```

In Scala le funzioni possono essere trasformate nell'uno o nell'altro tipo facilmente:

```
val sum_curried3 = (sum _).curried
// signature: sum_curried3: Int => (Int => Int) = <function1>

sum_curried3 (1)(2)
// result: res10: Int = 3

val sum_uncurried = Function.uncurried(sum_curried _)
// signature: sum_uncurried: (Int, Int) => Int = <function2>

sum_uncurried(1,2)
// result: res11: Int = 3
```

Il carattere _ (il *segnaposto* o *placeholder*) è necessario per indicare al compilatore di trattare la funzione come un valore e non come una invocazione. Per poterla convertire va trattata come una *funzione parzialmente applicata*. Bisogna notare che Scala è in grado di convertire le funzioni *curried* fino ad un numero di parametri pari a 5 (al compilatore versione 2.11.5).

```
def curried (a:Int)(b:Int)(c:Int)(d:Int)(e:Int)(f:Int)(g:Int)(h:Int)(i:Int) = a+b+c+d+e+f+g+h+i
// signature: curried: (a: Int)(b: Int)(c: Int)(d: Int)(e: Int)(f: Int)(g: Int)(h: Int)(i: Int)Int

val uncurried = Function.uncurried(curried _)
// signature: uncurried: (Int, Int, Int, Int, Int) => Int => (Int => (Int => (Int => Int))) = <function5>

uncurried(1,2,3,4,5)(6)(7)(8)(9)
// result: res20: Int = 45

uncurried(1,2,3,4,5,6,7,8,9)
/* error: <console>:10: error: too many arguments for method apply:
(v1: Int, v2: Int, v3: Int, v4: Int, v5: Int)Int => (Int => (Int => (Int => Int))) in trait Function5
uncurried(1,2,3,4,5,6,7,8,9) */
```

Scala deve far convivere *mondi* diversi: quello **Java** in OOP e quello funzionale. Però anche se sembra una limitazione, chi mai potrà fare una funzione (in un linguaggio funzionale) con più di cinque argomenti? Nel caso… Ripensate al vostro codice.

Scala ha anche numero massimo di parametri dichiarabili per una funzione: 22.

3.5 Funzioni parzialmente applicate.

Sono e ne abbiamo già parlato, un modo che i linguaggi funzionali permettono per restringere il numero dei parametri cablandoli per l'uso locale. Il concetto è che cablando un certo numero di parametri si ottiene una funzione con i restanti.

```
let sum a b = a + b;;
(* signature: val sum : int -> int -> int = <fun> *)

let sum_four_to = sum 4;;
(* signature: val sum_four_to : int -> int = <fun> *)

sum_four_to 5;;
(* result: - : int = 9 *)
```

Guardiamo questo altro esempio per capire meglio la loro utilità:

```
[1;2;3] |> List.map sum_four_to;;
(* result: - : int list = [5; 6; 7] *)
```

Cosa succede? Abbiamo una lista di `int` poi un operatore detto *pipe*, una chiamata a `List.map` e la nostra funzione parzialmente applicata.

Vediamo di capire meglio. `List.map sum_four_to` è essa stessa una parziale applicazione di `List.map` dato che la sua *signature* è questa:

```
('a -> 'b) -> 'a list -> 'b list = <fun>
```

È una funzione che ha come primo argomento una funzione da applicare ad ogni elemento della lista che prende come secondo argomento. Il valore di ritorno sarà una nuova lista.

```
let sum_four_for_all = List.map sum_four_to;;
(* signature: val sum_four_for_all : int list -> int list = <fun> *)

[1;2;3] |> sum_four_for_all;;
(* result: - : int list = [5; 6; 7] *)
```

In questo esempio è stata esplicitamente creata una nuova *funzione parzialmente applicata* ed applicata alla lista. Come si vede il risultato è identico.

Esageriamo:

```
[1;2;3] |> sum_four_for_all |> sum_four_for_all;;
(* result: - : int list = [9; 10; 11] *)
```

Vediamo l'operatore |> (*pipe*). La *pipe* fa quello che *deve fare* e sarebbe di convogliare il valore di una funzione su un'altra in una sequenza definita. Seguendo il flusso dell'esempio precedente:

1. la lista viene passata alla prima funzione
2. la lista viene elaborata ed un nuova lista è in uscita
3. la nuova lista è passata alla seconda funzione.
4. una nuova lista elaborata è in uscita

Per convenienza visiva, quando le *pipe* sono molte possiamo anche scrivere:

```
[1;2;3]
|> sum_four_for_all
|> sum_four_for_all;;
(* result: - : int list = [9; 10; 11] *)
```

Possiamo così creare delle *pipeline* di elaborazione molto complesse mantenendo nel contempo un *discreta* facilità di lettura.

L'operatore *pipe* è definito in questa maniera (dalla versione 4.00 di OCaml è implementato nativamente):

```
let (|>) x fn = fn x;;
(* signature: val ( |> ) : 'a -> ('a -> 'b) -> 'b = <fun>  *)
```

Direi che è semplice: è una funzione che prende un valore ed una funzione applicando quella funzione a quel valore. Esiste anche una *reverse pipe*:

```
let (@@) fn x = fn x;;
(* signature: val ( @@ ) : ('a -> 'b) -> 'a -> 'b = <fun> *)
```

In **F#** le due *pipe* sono |> e <| rispettivamente. In OCaml sono diverse per un problema di associatività degli operatori: storicamente in OCaml il carattere | come iniziale nelle funzioni infisse fornisce associatività a sinistra mentre @ a destra.

La composizione di funzioni o *function composition* è una interessante applicazione delle funzioni parzialmente applicate.

In **F#** esistono degli operatori di composizione e sono: >> e <<. In OCaml non ci sono nella libreria standard ma occorre un supporto aggiuntivo tramite Batteries[17] o Core[7] (Batteries nel modulo `BatStd` come |- e -| mentre Core nel modulo `Fn` ha la funzione `compose`).

Sono comunque facilmente definibili come:

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

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

Vediamo come farle funzionare:

```
let sum a b = a + b;;
(* signature: val sum : int -> int -> int = <fun> *)

let times v n = v * n;;
(* signature: val times : int -> int -> int = <fun> *)

let sum_one_for_four_times = sum 1 >> times 4;;
(* signature: val sum_one_for_four_times : int -> int = <fun> *)

sum_one_for_four_times 2;;
(* result: - : int = 12 *)
```

[17] http://batteries.forge.ocamlcore.org/

Un altro esempio:

```
let sum_to_and_mult x = (+) x >> ( * );;
(* signature: val sum_to_and_mult : int -> int -> int -> int = <fun> *)

sum_and_mult 1 4 5;;
(* result: - : int = 25 *)

sum_to_and_mult 3 4 5;;
(* result: - : int = 35 *)
```

Interessante no? Il valore 3 è sommato al valore 4 e poi moltiplicato per il valore 5. Notare che si è scritto `( * )` e non `(*)` perché `(*` è l'inizio dei commenti in OCaml.

3.6 Funzioni ricorsive

Le funzioni ricorsive sono molto importanti nella programmazione funzionale che ha dei supporti ottimizzati per utilizzarle. In linea di massima i costrutti imperativi per i cicli sarebbero superflui o quasi in un linguaggio funzionale.

Una funzione ricorsiva in OCaml deve essere esplicitamente dichiarata:

```
let rec factorial x =
  match x with
  | 0 -> 1
  | x -> x * factorial (x - 1)
;;
(* signature: val factorial : int -> int = <fun> *)
```

Con `let rec` si dichiara che una funzione è ricorsiva, nel caso si ometta `rec` il compilatore emetterà un errore:

```
let factorial x =
  match x with
  | 0 -> 1
  | x -> x * factorial (x - 1)
;;
(* Error: Unbound value factorial *)
```

Questo però potrebbe portare a strani comportamenti collegati allo *shadowing*:

```
let rec factorial x =
  match x with
  | 0 -> 1
  | x -> x * factorial (x - 1)
;;

let factorial x =
  match x with
  | 0 -> 1
  | x -> x * factorial (x - 1)
;;
```

in questa forma non saranno sollevati errori (all'inverso invece sì). Portate molta attenzione.

Si possono anche dichiarare funzioni mutualmente ricorsive tramite il costrutto `let rec..and`:

```
let rec is_even x =
  if x = 0 then true else is_odd (x - 1)
and is_odd x =
  if x = 0 then false else is_even (x - 1)
;;
(* signature: val is_even : int -> bool = <fun> *)
(* signature: val is_odd : int -> bool = <fun> *)
```

Le funzioni ricorsive hanno delle importanti ottimizzazioni nei linguaggi funzionali e lo vedremo più avanti con la tail recursion[3.10].

3.7 Funzioni anonime

Le *funzioni anonime* sono funzioni che, ovviamente, non hanno un nome a cui riferirsi. Nei codici presentati precedentemente sono state già incontrate, ma vediamole più in dettaglio. Sono chiamate anche *lambda expression* ed hanno due funzioni principali:

- Sono passate come argomento direttamente ad una *funzione di ordine superiore*.
- Si usano come valori di ritorno.

Hanno avuto origine dai lavori del matematico Alonzo Church[18], il quale teorizzò nel suo *lambda calculus* nel 1936 come le funzioni siano di fatto tutte anonime.

Strutturalmente sono delle funzioni annidate che possono accedere a valori compresi nell'ambito di visibilità che contiene la funzione stessa. Questo significa che devono essere implementate come *closure* (*chiusure*). Essendo anonime non possono essere ricorsive a meno di non usare un *fixpoint operator*.

Un *fixpoint operator* o *fixpoint combinator* è matematicamente una funzione che soddisfa la relazione:

$$y(f) = f(y(f)) \tag{10}$$

ed è chiamata così perché se si imposta $x = y(f)$ rappresenta una soluzione per equazione a *punto fisso*. Per farla decisamente semplice e breve ed uscendo dalla matematica: se consideriamo il *binding* di una funzione su un nome il suo *punto fisso*, essa potrà ruotare intorno al suo nome. Si potrà avere quindi *ricorsione*.

Per risolvere questo problema si utilizza un cosiddetto *Y combinator* e cioè una funzione *senza stato* o *pura* che prenderà come argomento una funzione anch'essa *pura* trasformandola in ricorsiva.

```
type 'a b = Roll of ('a b -> 'a);;
(* signature: type 'a b = Roll of ('a b -> 'a) *)

(* Definizione con la direttiva rectypes (che permette tipi ricorsivi) attivata *)
(* let fixpoint f g = (fun x a -> f (x x) a) (fun x a -> f (x x) a) g *)
(* signature: val fixpoint : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun> *)

let unroll (Roll x) = x;;
(* signature: val unroll : 'a mu -> 'a mu -> 'a = <fun> *)

let fixpoint f =
  (fun x a -> f (unroll x x) a) (Roll (fun x a -> f (unroll x x) a))
;;
(* signature: val fixpoint : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun> *)

let factorial f = function
  | 0 -> 1
  | x -> x * f (x - 1)
;;
(* signature: val factorial : (int -> int) -> int -> int = <fun> *)

let factorial2 x =
  fixpoint (fun f -> function
      | 0 -> 1
      | x -> x * f (x - 1)) x
;;
(* signature: val factorial2 : int -> int = <fun> *)

let () =

  Printf.printf "Recursion (fixpoint) with: fixpoint factorial 5\n";
  Printf.printf "%d" (fixpoint factorial 5);

  Printf.printf "\n";

  Printf.printf "Recursion (fixpoint) with: factorial2 5\n";
  Printf.printf "%d\n" (factorial2 5);
```

[18] http://en.wikipedia.org/wiki/Alonzo_Church

`fixpoint` è il nostro *y-combinator* che rende ricorsiva una funzione *pura* sia in forma anonima che non anonima.

3.7.1 Lambda Calculus

Il lavoro di Church[18] sul *lambda calculus* iniziò in realtà nel 1930 con gli studi sulla *logica compbinatoria* ma fu dimostrato come *logicamente inconsistente* da due altri *logici-matematici*: Stephen Cole Kleene[19] (padre delle *espressioni regolari* e di alcuni teoremi che sono da fondamento alla ricorsione) e John Barkley Rosser[20]; in quello che venne definito il *paradosso di Kleene-Rosser*. Nel 1936 Church pubblicò solo una parte isolata e relativa alla pura computazione, quella che è conosciuta come *untyped lambda calculus* poi evoluta nel 1940 nella *simply typed lambda calculus*.

Il *lambda calculus* è la base per quelle che vengono chiamate *closure* e per il *currying*.

3.8 Organizzare le funzioni.

Ogni linguaggio moderno offre un qualche modo per organizzare le funzioni in gruppi correlati. Un sistema di moduli più o meno sofisticato nella gestione delle relazioni tra di essi. Alcuni dei linguaggi funzionali sono anche di tipo *ibrido* o *multiparadigma*. Scala per esempio ha un supporto totale per la programmazione OOP: *classi*, interfacce (implementabili detti *trait*), *package*, ereditarietà e così via. OCaml ha una estensione per la programmazione orientata agli oggetti, anche se per un tradizionalista OOP può apparire leggermente strana.

I *moduli* sono la base per dare una correlazione logica ad un insieme di funzioni, una scatola con un nome che contiene altre piccole scatole. Un *namespace*.

Prima di dare una occhiata ai *moduli* va ricordato che generalmente in un linguaggio funzionale si possono definire funzioni dentro altre funzioni e questo semplicemente perché sono valori. Non hanno uno speciale trattamento come in molti linguaggi imperativi (alcuni comunque permettono la definizione di funzioni interne).

```
1 let sum_value_in_list n list =
2   let plus = (+) n in
3   List.map plus list
4
5 let () =
6   [1;2;3;4;5]
7   |> sum_value_in_list 5
8   |> List.iter (fun el -> Printf.printf "%d\n" el)
```

Questo è un piccolo programma che evidenzia una funzione interna ad un'altra. Va notato come sia definita all'interno della struttura `let...in`. In questo semplice caso avremmo potuto scrivere semplicemente:

```
1 let sum_value_in_list n list =
2   List.map ((+) n) list
```

In OCaml i *moduli* sono molto importanti. Ogni file è un *modulo* ed il nome del modulo è derivato dal nome del file con la prima lettera maiuscola. I moduli creano uno spazio per i nomi dei valori con la possibilità di avere una visibilità controllata.

Negli esempi abbiamo già utilizzato i moduli, per esempio `List` con `List.map` o `List.iter`.

Di un modulo se ne può controllare *il cosa* far vedere all'esterno tramite un file associato di *interfaccia* (i file in OCaml hanno estensione .ml e le interfacce lo stesso nome con estensione .mli).

Per il nostro programma di cui sopra il file *.mli* conterrà:

```
1 val sum_value_in_list : int -> int list -> int list
```

che è la *signature* della funzione `sum_value_in_list`. Il modulo principale (automaticamente creato dal nome del file) può inoltre contenere dei sotto moduli:

[19] http://en.wikipedia.org/wiki/Stephen_Cole_Kleene
[20] http://en.wikipedia.org/wiki/J._Barkley_Rosser

```
module ToolsSub : sig
  val sum_value_in_list : int -> int list -> int list
end = struct
  let sum_value_in_list n list = List.map ((+) n) list
end

let () =
  [1;2;3;4;5]
  |> ToolsSub.sum_value_in_list 5
  |> List.iter (fun el -> Printf.printf "%d\n" el)
```

Che si può esprimere, forse, in maniera più elegante:

```
module type ToolsSub_t = sig
  val sum_value_in_list : int -> int list -> int list
end

module ToolsSub : ToolsSub_t = struct
  let sum_value_in_list n list = List.map ((+) n) list
end

let () =
  [1;2;3;4;5]
  |> ToolsSub.sum_value_in_list 5
  |> List.iter (fun el -> Printf.printf "%d\n" el)
```

È come se avessimo la coppia di file `toolsSub.ml` e `toolsSub.mli`

OCaml inoltre estende i suoi moduli fino ad essere parametrizzati e diventare quindi dei *functor* come le funzioni. Un modulo funtore è parametrizzato con altri moduli che quindi ne sono l'argomento. Vengono utilizzati per risolvere problemi strutturali come per esempio:

- Iniezione delle dipendenze (implementazioni di componenti sostituibili).
- Estensione del modulo con nuove funzionalità (*Mixin*).
- Istanziare il modulo con uno stato.

Un esempio banale:

```
module type X_int = sig
  val x : int
end

module Increment (M : X_int) : X_int = struct
  let x = M.x + 1
end

module Three = struct
  let x = 3
end

module Four = Increment(Three)

let () =
  print_int (Four.x - Three.x)
```

Si è stabilita una relazione tra i moduli. Il modulo `Increment` ha come parametro un modulo di tipo `X_int` e gli viene passato `Three` che ha la stessa *signature* del tipo considerato. Possiamo quindi passargli ogni altro modulo la cui signature è allineata:

```
module type X_int = sig
  val x : int
end

module Increment (M : X_int) : X_int = struct
```

```
  let x = M.x + 1
end

module Three = struct
  let x = 3
end

module Four = Increment(Three)

module MoreAndText : sig
  val x : int
  val text : bytes
end = struct
  let x = 10
  let text = "ten"
end

module TenAndText = Increment(MoreAndText)

let () =
  Printf.printf "%d\n" (Four.x - Three.x);
  Printf.printf "%d\n" (TenAndText.x - Four.x);
```

I moduli *funturi* non ci sono in F# (ma in SML.NET si quindi non ci dovrebbero essere problemi tecnici, pare però che che a Don Syme[21] non piacciano), come il sistema OOP di OCaml avendo quello *.NET*.

3.9 Pattern matching

Un'altra caratteristica generalmente ritrovabile nei linguaggi funzionali è il *pattern matching* e la decomposizione da strutture complesse in valori semplici. Bisogna pensare sempre al *mapping* come concetto pervasivo ed alla gestione delle strutture dati, in particolare delle liste.

Il *pattern matching* si può vedere come una struttura `switch..case` dei linguaggi imperativi, ma parecchio *pompata*.

```
void print_num(int num) {
  switch(num) {
      case 1:
          printf("1");
          break;
      case 2:
          printf("2");
          break;
      case 3:
          printf("3");
          break;
      default:
          printf("default case.");
  }
}
```

In OCaml l'espressione `match...with` fa la stessa cosa:

```
let print_num num =
  match num with
  | 1 -> print_string "1"
  | 2 -> print_string "2"
  | 3 -> print_string "3"
  | _ -> print_string "default case"
;;
(* signature: val print_num : int -> unit = <fun> *)

print_num 2;;
(* result: 2- : unit = () *)
```

[21] http://research.microsoft.com/en-us/people/dsyme/

```
12
13 print_num 90;;
14 (* result: default case- : unit = () *)
```

Una delle differenze importanti è che quello che nei linguaggi imperativi è uno *statement* (dichiarazione) nei funzionali è una *espressione* e come espressione restituisce un valore.

```
1  let print_num1 num =
2    print_string (
3      match num with
4      | 1 -> "1"
5      | 2 -> "2"
6      | 3 -> "3"
7      | _ -> "default case")
8  ;;
9  (* signature: val print_num1 : int -> unit = <fun>  *)
10
11 let print_num2 num =
12   let n = match num with
13     | 1 -> "1"
14     | 2 -> "2"
15     | 3 -> "3"
16     | _ -> "default case"
17   in
18   print_string n;
19 ;;
20 (* signature: val print_num2 : int -> unit = <fun>  *)
```

Le due impostazioni sono equivalenti.

L'espressione `match..with` ha molteplici usi come per esempio l'estrazione di dati da una lista. Questa funzione somma tutti gli elementi di una lista:

```
1  let rec sum list =
2    match list with
3    | [] -> 0
4    | head :: tail -> head + sum tail
5  ;;
6  (* signature: val sum : int list -> int = <fun> *)
7
8  sum [1;2;3;4];;
9  (* result: - : int = 10  *)
```

In queste poche righe sono evidenziate alcune cose importanti. La prima che di fatto l'espressione `match..with` è usata come un *decostruttore* per la lista, la seconda che si potrebbe ottimizzare.

3.9.1 Altri esempi di pattern matching

Si potrebbe usare al posto di una espressione `if..then..else` per esempio:

```
1  let this_is_that this that =
2    match this = that with
3    | true -> "yes"
4    | false -> "no"
5  ;;
6  (* signature: val this_is_that : 'a -> 'a -> bytes = <fun> *)
```

Passando delle funzioni in caso di `true` o in caso di `false`:

```
1  let this_is_that this that fn1 fn2 =
2    match this = that with
3    | true -> fn1
4    | false -> fn2
5  ;;
6  (* signature: val this_is_that : 'a -> 'a -> 'b -> 'b -> 'b = <fun> *)
7
```

```
this_is_that 1 2 ("yes") ("no");;
(* result: - : bytes = "no" *)

this_is_that 1 1 ("yes") ("no");;
(* result: - : bytes = "yes" *)
```

Il *pattern matching* come corpo della funzione è molto utilizzato e per questo OCaml offre una versione sintattica alternativa e più compatta:

```
let is_upper char =
  match char with
  | 'A' .. 'Z' -> true
  | _ -> false
;;
(* signature: val is_upper : char -> bool = <fun> *)

let is_upper = (fun char ->
    match char with
    | 'A' .. 'Z' -> true
    | _ -> false
  )
;;
(* signature: val is_upper : char -> bool = <fun> *)

let is_upper = function
  | 'A' .. 'Z' -> true
  | _ -> false
;;
(* signature: val is_upper : char -> bool = <fun> *)
```

Le tre versioni sono equivalenti ed a voi usare la forma che più vi aggrada, anche se l'ultima credo sia più leggibile e compatta. La parola chiave `function` genera una espressione che prende un argomento e vi applica un *match*. Si può vedere inoltre, come nei match vengano supportati i *range* di pattern che in questo caso sono le lettere maiuscole.

Una importante caratteristica del *pattern matching* è che spesso i compilatori sono in grado di valutare cosa i casi mancanti. Questo come in altri casi (i compilatori sono generalmente molto *stretti* nella valutazione del codice) ci da un enorme aiuto nello scrivere programmi *corretti*:

```
let num_to_str = function
    | 1 -> "1"
    | 2 -> "2"
    | 3 -> "3"
;;
(* Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: 0 *)
(* signature: val num_to_str : int -> bytes = <fun> *)
```

Ovviamente mancano dei casi, i numeri naturali `int` non si limitano a tre. A noi la scelta a questo punto, inserire i casi mancanti o una caso *predefinito* (*default*) tramite `_ -> qualcosa`, cioè con il carattere di *placeholder*.

I pattern possono avere delle espressioni di *guardia* per filtrare ulteriormente:

```
let rec count = function
    | [] -> 0
    | head :: tail when head = 0 || head = 1 -> 1 + count tail
    | _ :: tail -> count tail
;;
(* signature: val count : int list -> int = <fun> *)

count [1; 2; 3; 0; 1; 2; 3];;
(* result: - : int = 3 *)
```

Il *pattern matching* è senza dubbio una delle caratteristiche più importanti e potenti di OCaml che oltre ad essere un elegante ed efficiente sistema per la diramazione computazionale, è soggetto al controllo statico oltre che al controllo sui tipi, accelerando per questo lo sviluppo ed aumentando la chiarezza del codice e la sua correttezza.

Massimo Ghisalberti - tel.: +39 3939018290 – email: massimo.ghisalberti@gmail.com

3.9.2 Considerazioni sulla velocità

l'espressione `match..when` a qualunque grado di complessità è particolarmente veloce, i compilatori sono in grado di risolvere i *salti* in maniera molto efficiente e senza testare i valori intermedi. Sono molto più veloci di una serie di espressioni `if..then...else...if` ed addirittura del semplice `if..then..else`. Per consiglio, controllare il proprio linguaggio di programmazione sulla efficienza del *match* ne potrebbe valere la pena.

3.9.3 Tipi varianti (*variants*)

Il *pattern matching* diventa molto utile ed interessante nella decomposizione o destrutturazione dei tipi. Un caratteristica di OCaml particolarmente utile sono i *tipi varianti* che seguono una sintassi come segue:

```
type <variant> =
  | <Tag> [ of <type> [* <type>]... ]
  | <Tag> [ of <type> [* <type>]... ]
  | ...
```

Ogni riga rappresenta un caso *variante* con una etichetta ed un possibile tipo associato.

```
type color =
  | RGB of int * int * int
  | GRAY of int
  | HEX of bytes
;;
(* signature:
   type color = RGB of int * int * int | GRAY of int | HEX of bytes *)

let hex_to_int v =
  int_of_string ("0x" ^ v)
;;
(* signature: val hex_to_int : bytes -> int = <fun> *)

let chars_to_string ?sep:(sep = "") chars =
  let strings =
    List.map (fun c -> String.make 1 c) chars
  in
  String.concat sep strings
;;
(* signature: val chars_to_string : ?sep:bytes -> char list -> bytes = <fun> *)

let hex_to_list_of_channels hexstr =
  let patched_str =
    match String.length hexstr with
    | 6 -> hexstr
    (* CSS Short Hexadecimal notation *)
    | 3 ->  chars_to_string [hexstr.[0];hexstr.[0];hexstr.[1];hexstr.[1];hexstr.[2];hexstr.[2]]
    (* Any other case in white*)
    | _ -> "ffffff"
  in
  let hex_value str index =
    ((String.make 1 str.[index]) ^ (String.make 1 str.[index - 1]))
  in
  let rec loop accumulator index =
    match index with
    | index when index < 0 -> accumulator
    | _ -> loop ((hex_to_int (hex_value patched_str index)) :: accumulator) (pred (index - 1))
  in loop [] (String.length patched_str - 1)
;;
(* signature: val hex_to_list_of_channels : bytes -> int list = <fun> *)

let to_rgb color =
  match color with
  | RGB (r,g,b) -> color
  | GRAY (v) -> RGB(v,v,v)
  | HEX (v) -> let parts = hex_to_list_of_channels v in
    RGB((List.nth parts 0),(List.nth parts 1),(List.nth parts 2))
```

```
;;
(* signature: val to_rgb : color -> color = <fun>  *)

let decompose = function
  | RGB(r,g,b) -> (r,g,b)
  | _ -> (255,255,255)
;;
(* signature: val decompose : color -> int * int * int = <fun> *)

let print_value rgb =
  let (r, g, b) = decompose rgb in
  Printf.printf "RGB VALUE: %d %d %d\n" r g b
;;
(* signature: val print_value : color -> unit = <fun> *)

let rgb1 = to_rgb (HEX "aaffcc");;
(* result: - : color = RGB (170, 255, 204) *)

let rgb2 = to_rgb (GRAY 127);;
(* result: - : color = RGB (170, 255, 204) *)

let rgb3 = to_rgb (RGB (170,255,204));;
(* result: - : color = RGB (170, 255, 204) *)

let rgb4 = to_rgb (HEX "a0f");;
(* signature: val rgb : color = RGB (170, 0, 255) *)

let () =
  print_value rgb1;
  print_value rgb2;
  print_value rgb3;
  print_value rgb4;
```

Il fine era quello di definire una funzione `to_rgb` che data una certa rappresentazione la convertisse in valori `RGB`. L'esempio è semplicistico ed ha dei problemi *intrinseci* ma penso che chiarisca abbastanza il concetto. La cosa più importante è come si può usare il `match..with` per discriminare la *varianza* di tipo (da notare anche la somiglianza con le `case class` di Scala).

Alcune cose vanno notate nel codice come per esempio il parametro `sep` della funzione `chars_to_string` che è un parametro *opzionale* e che può avere una inizializzazione. Cosa molto importante è che OCaml è *strettamente tipizzato* ed è evidente come le conversioni e trasformazioni di tipo debbano essere esplicite. È norma quindi, scrivere parecchie piccole funzioni per agevolare questo compito (od utilizzare librerie alternative e molto sofisticate come Core[7] o Batteries[17]). Il `match..with` è pervasivo.

3.10 Tail Call Optimization

Nel vedere il pattern matching ho presentato una funzione `sum` che usa la ricorsione come mezzo per sommare gli elementi di una lista di numeri, vediamo adesso come si potrebbe ottimizzare:

```
let rec sum_rec accumulator list =
  match list with
  | [] -> accumulator
  | head :: tail -> sum_rec (head + accumulator) tail
;;
(* signature: val sum_rec : int -> int list -> int = <fun> *)

let sum = sum_rec 0;;
(* signature: val sum : int list -> int = <fun> *)

sum [1;2;3];;
(* result: - : int = 6  *)
```

o in una forma più *racchiusa*:

```
let sum =
  let rec sum_rec accumulator list =
```

```
  match list with
  | [] -> accumulator
  | head :: tail -> sum_rec (head + accumulator) tail
  in
  sum_rec 0
;;
(* signature: val sum : int list -> int = <fun> *)
```

Qui ho usato una *tail recursion*. Ho modificato il codice per far si che la invocazione alla funzione ricorsiva sia alla fine della funzione stessa, in *tail* o *coda* cioè. Per fare questo ho avuto bisogno di un accumulatore come secondo parametro.

La *tail recursion* è un importante meccanismo nelle ricorsioni ed è spesso usata per esprimere i *loop*. Questo tipo di ricorsione (una funzione che internamente richiama sé stessa) è molto efficiente nei linguaggi funzionali (anche se non in tutti, Scala per esempio che deve fare i conti con la sottostante JVM). La sua caratteristica è che il compilatore non deve allocare lo *stack* e perdere tempo in una nuova invocazione per ogni chiamata (*tail-call optimization*). Sarà difficile se non impossibile in OCaml saturare lo *stack* con una chiamata ricorsiva in *tail recursion*.

Un effetto collaterale potrebbe essere un aumento della velocità di esecuzione (anche se questionabile ed estremamente dipendente dal codice e dal compilatore). Una funzione ricorsiva in *tail recursion* potrebbe in linea di principio essere leggermente più veloce a causa del fatto che vengono oltrepassate le numerose *invocazioni a funzione* che inevitabilmente comporterebbero codice da eseguire e allocazioni sullo *stack* (per contro le frequenti sovrascritture potrebbero invece introdurre rallentamenti).

La funzione `fib` che abbiamo incontrato prima ottimizzata in *tail recursion* sarà:

```
let fib x =
  let rec fib_rec x acc piv =
    match x with
    | 0 -> acc
    | _ -> fib_rec (x - 1) piv (acc + piv)
  in
  fib_rec x 0 1
;;
(* signature: val fib : int -> int = <fun> *)
```

Come vedrete in seguito[3.15] questa versione sarà anche notevolmente più veloce, anche per motivi non imputabili alla stessa *tail call optimization*.

L'uso *sapiente* della *ricorsione a coda* rende pressoché inutili i vari *cicli* dei linguaggi imperativi senza le problematiche delle *funzioni ricorsive* degli stessi.

3.11 *Loop* e cicli - Map e Fold

La programmazione funzionale con i suoi costrutti rende praticamente inutili molte delle strutture cicliche dei linguaggi imperativi. Ogni ciclo può essere espresso in termini di funzioni ricorsive ed è anche per questo motivo che i linguaggi funzionali sono particolarmente ottimizzati per questo genere di cose. A parte la possibilità in sé della ricorsività, nelle librerie base sono presenti tutta una serie di funzioni utilizzabili per iterare all'interno di collezioni ed eseguire operazioni sugli elementi che le compongono.
Il *folding* è una operazione molto comune che consiste nel far *avvolgere* gli elementi di una collezione da una funzione ed accumularne il valore di questo *avvolgimento*.

```
List.fold_left (+) 0 [1;2;3;4;5];;
(* result: - : int = 15 *)
```

La funzione `fold_left` accetta una funzione con cui *avvolgere* ogni elemento della lista `[1;2;3;4;5]`, un accumulatore e la lista stessa. L'applicazione di `(+)` sommerà i valori della lista ponendo il risultato nell'accumulatore. Come se visualmente, sostituisse il + (che è un operatore associativo) al separatore degli elementi della lista `;`.

Utilizzando le caratteristiche imperative di OCaml, si potrebbe scrivere una `sum_list` in questo modo (anche se non direttamente analogo all'uso di `fold_left`):

```
let sum_list list =
  let acc = ref 0 in
  let length = (List.length list) - 1 in
  for index = 0 to length do
    let value = List.nth list index in
    acc := !acc + value
  done;
  !acc
;;
(* signature: val sum_list : int list -> int = <fun> *)
```

Ci sono delle cose interessanti da vedere (a parte quanto possa essere *brutta*): l'utilizzo della parola chiave `ref` e del `!`. Il ciclo `for..to..do..done` credo sia chiaro e simile a qualunque altro linguaggio che lo supporta.

Dichiarare una *variabile* (perché adesso è una variabile) come `let acc = ref 0` significa fare questo (se dovessimo fare tutto a mano):

```
type 'a mut = { mutable contents : 'a };;
(* signature: type 'a mut = { mutable contents : 'a } *)

let mut x = { contents = x };;
(* signature: val mut : 'a -> 'a mut = <fun>  *)

let (!@) r = r.contents;;
(* signature: val ( !@ ) : 'a mut -> 'a = <fun> *)

let (<=) r x = r.contents <- x;;
(* val ( <= ) : 'a mut -> 'a -> unit = <fun> *)

let acc = mut 10;;
(* signature: val acc : int mut = {contents = 10} *)

acc <= 20;;
(* result: - : unit = () *)

!@acc;;
(* signature: - : int = 20 *)
```

Ho definito un tipo `mut` (nella libreria è definito `ref`) che è un record con un campo `contents` *mutabile* (`mutable`). Poi alcune funzioni per gestirlo.
La stessa identica cosa è nella libreria come:

```
type 'a ref = { mutable contents : 'a };;
(* signature: type 'a ref = { mutable contents : 'a; } *)

let ref x = { contents = x };;
(* signature: val ref : 'a -> 'a ref = <fun> *)

let (!) r = r.contents;;
(* signature: val ( ! ) : 'a ref -> 'a = <fun> *)

let (:=) r x = r.contents <- x;;
(* signature: val ( := ) : 'a ref -> 'a -> unit = <fun> *)
```

Se volessimo implementare `sum_list` con una struttura `while..do..done`:

```
let sum_list list =
  let acc = ref 0 in
  let index = ref 0 in
  let length = (List.length list) in
  while !index < length do
    acc := !acc + (List.nth list !index );
    incr index
  done;
    !acc
;;
(* signature: val sum_list : int list -> int = <fun> *)
```

La funzione `fold_left` ha una sua analoga contraria come `fold_right`, il *right* e *left* si riferisce alla direzione del *folding* e la versione *right* è normalmente meno efficiente. Il *fold* è chiamato in molti modi: `reduce`, `accumulate`, `aggregate`, `compress`, `inject`. Lo troverete in tutti i linguaggi funzionali ma non solo in quelli.

Un altro metodo importantissimo è `map`, che applica una funzione ad ogni elemento della collezione restituendo una collezione di risultati.

```
List.map ((+) 2) [1;2;3;4;5];;
(* result: - : int list = [3; 4; 5; 6; 7] *)
```

Applicando una funzione *parzialmente applicata* `((+) 2)` si sommerà il numero 2 ad ogni elemento della lista di interi.

Queste due funzioni: `map` e `fold` hanno delle importanti implicazioni se usate insieme. Forse avrete sentito parlare di `Map-Reduce` soprattutto in merito al motore di ricerca **Google**. Google tramite queste tecniche è riuscita ad aumentare a dismisura la frequenza dei risultati nell'unità di tempo. La caratteristica delle due funzioni di essere senza stato ed effetti collaterali fa in modo che grosse interrogazioni sui database possano essere eseguite in parallelo con conseguente aumento delle prestazioni.

Un esempio di come usare le due funzioni insieme:

```
let find_if_size_is size list =
  List.fold_left
    (fun acc tpl ->
       let name, wsize = tpl in
       if wsize = size then
         acc @ [name]
       else
         acc @ [])
    []
    (List.map (fun x -> (x, String.length x)) list)
;;
(* signature: val find_if_size_is : int -> bytes list -> bytes list = <fun> *)

find_if_size_is 5 ["pippo"; "paperino"; "topolino"; "qui"; "pluto"];;
(* result: - : bytes list = ["pippo"; "pluto"] *)
```

Giusto per esercizio di codifica, come si può notare sono tutti argomenti della funzione `fold_left` (il codice potrebbe essere scritto tutto su una linea).

3.12 Lazy and Strict

I linguaggi si possono dividere anche in *strict* (*severi* e sono la maggior parte) o *lazy* (*pigri*). I derivati dal ML, come OCaml, sono *strict* mentre altri come *Haskell* sono *lazy*. I linguaggi funzionali *puri* sono normalmente *pigri*. La *lazy evaluation* o *call by need* (invocazione a richiesta) è una strategia di valutazione che rimanda il più possibile la computazione di una espressione, almeno finché il suo valore non sia richiesto.

Attraverso la valutazione *lazy* si hanno alcuni vantaggi teorici:

- Aumento delle performance limitando i calcoli al solo momento richiesto, le invocazioni di funzioni o le valutazioni di valori hanno sempre un certo *carico*.
- La possibilità di costruire strutture complesse praticamente infinite.
- Uso di astrazioni di strutture di controllo invece di primitive, ovvero il poterle organizzare in maniera più *dichiarativa* e meno a *statement imperativi*.

Al contrario, nella valutazione *strict* (detta anche *eager* o *greedy*), l'espressione è valutata all'assegnamento od al *binding*. Questo tipo di strategia è spesso associata ai linguaggi imperativi dove l'ordine di esecuzione rispecchia l'organizzazione strutturale del codice sorgente. Un vantaggio è che elimina il tracciamento e la *schedulazione* delle valutazioni delle espressioni permettendo una più facile predizione del flusso logico del codice. Uno svantaggio evidente è che ci saranno computazioni non necessarie a runtime e questo obbligherà il programmatore ad una organizzazione delle istruzioni del codice ottimale.

Oggi, comunque, la maggior parte dei compilatori sono in grado di riorganizzare l'ordine delle valutazioni o di eliminare codice non utilizzato; quindi la distinzione o i pregi dell'una o dell'altra strategia si è decisamente assottigliata.

Anche nei linguaggi *strict* molti *statement* sono valutati in maniera *pigra*, il blocco `if a then b else c` verrà valutato solo se `a = true`. La valutazione a corto circuito (*short circuit evaluation*) delle strutture di controllo booleane lo è nella stessa maniera. Se in una valutazione `if a || b || c then d else e`, a è `true`, b e c non saranno valutate perché non porterebbero ulteriore *informazione*.

Si può simulare la *lazy evaluation* in un linguaggio *severo* tramite l'uso di funzioni che non vengono valutate immediatamente ma solo alla loro invocazione (anche se non se ne può sapere esattamente la riuscita a meno di non conoscere intimamente il comportamento del compilatore. Alcuni hanno delle strategie interne nel caso in cui una funzione sia immediatamente valutabile in fase di compilazione, oltre al fatto che gli argomenti della funzione vengono valutati prima della funzione stessa. Potrebbe non esserci alcun vantaggio in ogni caso.).

In OCaml, che è un linguaggio *strict*, si può avere il supporto per la valutazione *lazy* tramite un modulo: `Lazy`.

```
let div_by_zero = 1/0;;
(* Exception: Division_by_zero. *)

let div_by_zero2 = lazy (1/0);;
(* signature: val div_by_zero2 : int lazy_t = <lazy> *)

Lazy.force div_by_zero2;;
(* Exception: Division_by_zero. *)
```

`div_by_zero` è valutata immediatamente ed ha scatenato una eccezione, mentre `div_by_zero2` no. La seconda funzione è di tipo `lazy_t` e non viene valutata. Non finché non viene forzata esplicitamente con `Lazy.force`.

In OCaml quindi, abbiamo una valutazione *lazy* manuale.

In Scala il meccanismo è lo stesso e per avere dei valori *pigri* dobbiamo dichiararli `lazy`, anche se non c'è bisogno di forzarli venendo valutati alla chiamata:

```
lazy val div_by_zero = 1/0
/* signature: div_by_zero: Int = <lazy> */

div_by_zero
/* exception: java.lang.ArithmeticException: / by zero
```

3.12.1 La divisione per zero

Giusto per informazione vediamo cosa fa Haskell in una divisione per zero:

```
let div_by_zero = 1 `div` 0
-- :t div_by_zero
-- signature: div_by_zero :: Integer

let div_by_zero2 = 1 / 0
-- :t div_by_zero2
-- signature: div_by_zero2 :: Double

let div_by_zero3 = 0 / 0
-- :t div_by_zero3
-- signature: div_by_zero3 :: Double

let div_by_zero4 = 1 / (-0)
-- :t div_by_zero4
-- signature: div_by_zero4 :: Double

div_by_zero
-- exception: *** Exception: divide by zero

div_by_zero2
-- result: -Infinity

div_by_zero3
-- result: -NaN
```

```
25
26 div_by_zero4
27 -- result: -Infinity
```

Questi sono alcuni casi che potrebbero lasciare un poco interdetti. In una divisione *intera* con l'operatore `div` giustamente viene sollevata una eccezione ma nel caso del tipo a virgola mobile? Haskell (come gli altri) segue le specifiche IEEE 754[22] per i numeri in virgola mobile dove una divisione per zero equivale ad `Infinity`. Le eccezioni sono sollevate solo per gli `Integer` dove lo zero ha un valore *certo* mentre non lo è per la matematica approssimata dei `Float`.

Scala non è diverso:

```
1 1.0 / 0
2 /* result: res3: Double = Infinity */
```

Ocaml non è diverso:

```
1 1.0 /. 0.0;;
2 (* result: - : float = inf *)
```

C++ non è diverso:

```
1  #include <cstdio>
2
3  int main() {
4    float zero = 1.0 / 0.0;
5    std::printf("%f\n", zero);
6  }
7
8  /* compilation: gcc div_by_zero.cpp -std=c++11 -o div_by_zero */
9  /* call: ./div_by_zero */
10 /* result: inf */
```

```
1  #include <cstdio>
2
3  int main() {
4    int zero = 1 / 0;
5    std::printf("%d\n", zero);
6  }
7
8  /* compilation: gcc div_by_zero.cpp -std=c++11 -o div_by_zero
9  div_by_zero.cpp: In function ‘int main()’:
10 div_by_zero.cpp:4:19: warning: division by zero [-Wdiv-by-zero]
11    int zero2 = 1 / 0; */
12                   ^
13 /* call:  ./div_by_zero */
14 /* exception: [1]    6373 floating point exception (core dumped)  ./div_by_zero */
```

Per insistere sulla particolarità: la divisione di interi solleverà una eccezione mentre quella tra `float` no ed il valore di ritorno sarà `inf` o `Infinity`. Questo ci fa comprendere come i linguaggi in cui il *casting* (conversione o promozione di tipo) è *implicitamente* eseguito possano essere in alcuni casi molto pericolosi.

OCaml è strettamente tipizzato e nessun *cast* è implicito, anche se potrà sembrare *noioso* ed a volte macchinoso questa peculiarità si dimostrerà un vantaggio nello scovare i possibili *bug* del nostro codice.

Questo succede in C++:

```
1  #include <cstdio>
2
3  int main() {
4    float zero1 = 1.0 / 0;
5    int zero2 = 1.0 / 0;
6    //int zero3 = 1 / 0;
7
8    std::printf("as Float: %f\n", zero1);
9    std::printf("as Int: %d\n", zero2);
```

[22] http://it.wikipedia.org/wiki/IEEE_754

```
  //std::printf("as Int: %d\n", zero3);
}

/* compilation: gcc div_by_zero.cpp -std=c++11 -o div_by_zero
div_by_zero.cpp: In function 'int main()':
div_by_zero.cpp:4:23: warning: division by zero [-Wdiv-by-zero]
   float zero1 = 1.0 / 0;
                       ^
div_by_zero.cpp:5:21: warning: division by zero [-Wdiv-by-zero]
   int zero2 = 1.0 / 0; */

/* call: ./div_by_zero */
/* result:
as Float: inf
as Int: -2147483648 */
```

Se togliamo i commenti nel codice, si vedrà come il compilatore si lamenterà con la variabile `zero3` come aveva fatto prima con la `zero2`:

```
$  source gcc div_by_zero.cpp -std=c++11 -o div_by_zero
div_by_zero.cpp: In function 'int main()':
div_by_zero.cpp:4:23: warning: division by zero [-Wdiv-by-zero]
   float zero1 = 1.0 / 0;
                       ^
div_by_zero.cpp:5:21: warning: division by zero [-Wdiv-by-zero]
   int zero2 = 1.0 / 0;
                     ^
div_by_zero.cpp:6:19: warning: division by zero [-Wdiv-by-zero]
   int zero3 = 1 / 0;
                   ^
$  ./div_by_zero
[1]    6677 floating point exception (core dumped)  ./div_by_zero
```

La variabile `zero3` ha portato ad una eccezione non gestita, ma `zero2` ha restituito un valore negativo (in questo caso il più basso nel range degli interi segnati a 32bit. Se avessimo usato un `Int64_t` avremmo avuto un valore pari a: `-9223372036854775808`).

In OCaml questo non si può fare:

```
1.0 / 0;;
(* Error: This expression has type float but an expression was expected of type int *)

1.0 /. 0;;
(* Error: This expression has type int but an expression was expected of type float *)
```

3.12.2 Tipi *Boxed* e *Unboxed*

Nelle discussioni sulla programmazione funzionale è spesso affrontato questo argomento che pone alcune problematiche specialmente per i linguaggi *lazy*. La distinzione non è molto complessa anche se a volte il loro uso lo è. La sostanza è questa: un tipo `unboxed` ha una rappresentazione *grezza* in memoria, mentre un tipo *boxed* è più complesso.
Il tipo *boxed* è *racchiuso* (inscatolato) in una ulteriore struttura più o meno sofisticata, spesso allocata nella parte di memoria detta *heap*. Accedere ad un valore di questo tipo porta dunque una indirezione inevitabile ed un gioco di *puntatori*. Chi ha programmato in Java, saprà che esistono più rappresentazioni dello stesso tipo, quelli primitivi come `int` e quelli non primitivi: la classe `Integer`.

Il concetto è quello.

Che il linguaggio renda più o meno trasparente il loro uso, si tratta sempre di *puntatori* a zone di memoria che devono essere referenziati e dereferenziati.

La rappresentazione in memoria di un *array* `unboxed`:

mentre come si presenta uno `boxed`:

Appare ovvio, come la gestione dell'`array` non *boxed* sia possibilmente più veloce, nessuna indirezione o allocazione ulteriore o sofisticate *garbage collection*.

Nei linguaggi *lazy* normalmente i valori sono di tipo *boxed* a causa della loro *pigrizia*.
Nel caso in cui noi avessimo una funzione di questo tipo:

`double x = x * x`

in un linguaggio non *strict* `x` potrebbe non essere ancora stata valutata alla chiamata di `double`, nel qual caso verrebbe passato non il valore ma un puntatore ad una *closure* allocata nell'*heap*. Una volta valutata la *closure* il valore sarebbe passato alla funzione e la zona di memoria sovrascritta con il valore stesso. Come conseguenza ogni successiva chiamata ad `x` fornirebbe direttamente il valore. Questa è la *lazy evaluation*.

Alcuni compilatori di Haskell come il GHC cercano di convertire il più possibile i diversi tipi per migliorare le performance, ma con alcuni limiti sull'uso ed accorgimenti; per esempio non possono essere passati così come sono alle funzioni polimorfiche e devono prima essere convertiti.
A parte le considerazioni sulla velocità, i valori *boxed*, hanno una flessibilità maggiore fornendo ulteriori strutture o funzioni per la loro manipolazione che non i tipi *grezzi* (*raw*).

In OCaml i tipi base sono tutti *unboxed*.

3.13 Memoization

La tecnica detta *memoization* (parola coniata nel 1968 da Donald Michie[23] dal latino *memoro, -as, -avi, -atum, -are*) è spesso utilizzata per aumentare le prestazioni immagazzinando i risultati di invocazioni a funzione per computazioni successive. La tecnica è particolarmente utile nei linguaggi *lazy* per i motivi specificati sopra, ma è interessante in molti casi come per esempio il *parsing* di strutture ricorsive o l'analisi sintattica.

Il concetto è quello di costruire una tabella di *lookup* di valori di `input` in relazione ai rispettivi valori di `output`.

Come si dice spesso, l'*hello world* della *memoization* è il calcolo dell'nth elemento della sequenza di Fibonacci[24].

```
let rec fib x =
    match x with
    | 0 -> 0
    | 1 -> 1
    | x -> (fib (x - 1)) + (fib (x - 2))
;;
(* signature: val fib : int -> int = <fun> *)
```

I tempi per una invocazione sono con argomento 20 e 40 per esempio sono notevolmente diversi. Sulla mia macchina questi sono i valori:

- `fib 20` avrà dei tempi intorno a 0.0507832ms per un risultato di 6765
- `fib 40` avrà dei tempi intorno a 632.199ms per un risultato di 102334155

La differenza è enorme.

Usando la *memoization* potremmo accorciare i tempi notevolmente. Per prima cosa va impostato un funtore che si occupi di impostare una tabella di *lookup* e di cercarci di dati.

```
let memoize f =
  let table = Hashtbl.Poly.create () in
  (fun x ->
     match Hashtbl.find table x with
     | Some y -> y
     | None ->
       let y = f x in
       Hashtbl.add_exn table ~key:x ~data:y;
       y
  );;
(* signature: val memoize : ('a -> 'b) -> 'a -> 'b = <fun>  *)
```

Un classico sistema è quello di usare una *tabella Hash* (in questo caso la implementazione disponibile nella libreria Core[7]) per memorizzare i dati. Una versione usando direttamente la libreria standard:

[23] http://en.wikipedia.org/wiki/Donald_Michie
[24] http://it.wikipedia.org/wiki/Successione_di_Fibonacci

```
let memoize f =
  let table = Hashtbl.create 12 in
  (fun x ->
     try Hashtbl.find table x with Not_found ->
       let y = f x in
       Hashtbl.add table x y;
       y
  )
;;
(* signature: val memoize : ('a -> 'b) -> 'a -> 'b = <fun>  *)
```

La quantità 12 passata ad `Hashtbl.create` è la dimensione iniziale della tabella che però potrà ingrandirsi a richiesta. La versione standard di `Hashtbl.find` solleverà una eccezione `Not_found` in caso non trovi la chiave nella tabella, mentre quella contenuta in *Core* un `None`. I tipi `Some` e `None` sono dei tipi comuni nei linguaggi funzionali ed esprimono dei valori (*Options*) che possono o non possono contenere dati (praticamente la risposta funzionale e *sicura* al `Null` di molti linguaggi imperativi).

Il funzionamento non è complesso: `memoize` prende come argomento una funzione ed alloca una *tabella hash* restituendo una nuova funzione; questa quando invocata controlla se un valore sia preesistente, altrimenti chiama la funzione dell'argomento e ne immagazzina il valore.

Per valutare il tempo di invocazione abbiamo questa funzione un po' rudimentale:

```
let time f =
  (* =Time= è un modulo di Core. *)
  let start = Time.now () in
  let x = f () in
  let stop = Time.now () in
  printf "Time: %s\n" (Time.Span.to_string (Time.diff stop start));
  x
;;
(* signature: val time : (unit -> 'a) -> 'a = <fun> *)
```

A questo punto ecco il sorgente completo:

```
open Core.Std

let time f =
  let start = Time.now () in
  let x = f () in
  let stop = Time.now () in
  printf "Time: %s\n" (Time.Span.to_string (Time.diff stop start));
  x
;;
(* signature: val time : (unit -> 'a) -> 'a = <fun> *)

let rec fib x =
    match x with
    | 0 -> 0
    | 1 -> 1
    | x -> (fib (x - 1)) + (fib (x - 2))
;;
(* signature: val fib : int -> int = <fun> *)

let memoize f =
  let table = Hashtbl.Poly.create () in
  (fun x ->
     match Hashtbl.find table x with
     | Some y -> y
     | None ->
       let y = f x in
       Hashtbl.add_exn table ~key:x ~data:y;
       y
  );;
(* signature: val memoize : ('a -> 'b) -> 'a -> 'b = <fun>  *)
```

```
31
32 let fib_memoized = memoize fib;;
33 (* val fib_m : int -> int = <fun> *)
34
35 let () =
36   printf "Called for 20 -> fib 20\n";
37   let r = time (fun () -> fib 20) in
38   printf "Value: %d\n\n" r;
39
40   printf "Called for 40 -> fib 40\n";
41   let r = time (fun () -> fib 40) in
42   printf "Value: %d\n\n" r;
43
44   printf "First call for 20 -> fib_memoized 20\n";
45   let r = time (fun () -> fib_memoized 20) in
46   printf "Value: %d\n\n" r;
47
48   printf "First call for 40 -> fib_memoized 40\n";
49   let r = time (fun () -> fib_memoized 40) in
50   printf "Value: %d\n\n" r;
51
52   printf "Second call for 20 -> fib_memoized 20\n";
53   let r = time (fun () -> fib_memoized 20) in
54   printf "Value: %d\n\n" r;
55
56   printf "Second call for 40 -> fib_memoized 40\n";
57   let r = time (fun () -> fib_memoized 40) in
58   printf "Value: %d\n\n" r;
59 ;;
```

Compilandolo ed eseguendolo si vedrà come le seconde chiamate saranno estremamente più veloci:

```
1  $  ./fib.native
2
3  Called for 20 -> fib 20
4  Time: 0.0419617ms
5  Value: 6765
6
7  Called for 40 -> fib 40
8  Time: 631.014ms
9  Value: 102334155
10
11 First call for 20 -> fib_memoized 20
12 Time: 0.0429153ms
13 Value: 6765
14
15 First call for 40 -> fib_memoized 40
16 Time: 626.977ms
17 Value: 102334155
18
19 Second call for 20 -> fib_memoized 20
20 Time: 0.000953674ms
21 Value: 6765
22
23 Second call for 40 -> fib_memoized 40
24 Time: 0s
25 Value: 102334155
```

Questo è un esempio (un po' fallato e che potrebbe essere ottimizzato ancora) di come le capacità *multiparadigma* di OCaml siano di ausilio nella soluzione di problemi pratici, come in questo caso in cui è servita una *tabella hash modificabile*.

L'esempio ha però un problema: la `memoize` infatti così come è non è efficiente con una funzione ricorsiva. Il motivo è abbastanza semplice da intuire: le susseguenti chiamate nella ricorsione non saranno memorizzate. Ce ne possiamo rendere conto modificando la `memoize` in questo modo:

```
1 let memoize f =
2   let table = Hashtbl.create 12 in
```

```
  (fun x ->
     Printf.printf "find: %d\n" x;
     try
       let r = Hashtbl.find table x in
       Printf.printf "found - get: %d with value: %d\n" x r;
       r
     with Not_found ->
       let y = f x in
       Printf.printf "not found - store: %d with value: %d\n" x y;
       Hashtbl.add table x y;
       y
  )
;;
```

Lanciando il programma si potranno vedere dei commenti di questo tipo:

```
$  ./fib.native

First call for 40 -> fib_memoized 40
find?: 40
not found - store: 40 with value: 102334155
Time: 560.83ms
Value: 102334155

Second call for 40 -> fib_memoized 40
find?: 40
found - get: 40 with value: 102334155
Time: 0.000953674ms
Value: 102334155
```

La seconda chiamata ha trovato un solo valore immagazzinato e non è quello che cercavamo di fare: tutti i risultati dovrebbero essere considerati. Andrebbe ottimizzata meglio, ma non sarà così semplice.
La `memoize`, quindi, funzionerà solo se unita a funzioni non ricorsive a cui però non dobbiamo rinunciare. Innanzi tutto vediamo di eliminare la parola chiave `rec` dalla definizione:

```
let fib_nrec fn x =
    match x with
    | 0 -> 0
    | 1 -> 1
    | x -> (fn (x - 1)) + (fn (x - 2))
;;
(* signature: val fib_nrec : (int -> int) -> int -> int = <fun> *)
```

Quello che si è preparato è un cosiddetto `pattern`, ovvero un *modello computazionale*. Si è aggiunto inevitabilmente un ulteriore parametro `fn` che altro non è che la funzione che andremo poi ad invocare risorsivamente. Per provare come tutto ciò sia vero basta:

```
let rec fib_rec x = fib_nrec fib_rec x;;
```

eseguendo `fib_rec 20` vedremo che sarà la stessa cosa che aver dichiarato `let rec fib_rec x`. Questo modo di scrivere le funzioni ricorsive (funzione non `rec` e *compagna*) porta anche ad alcuni vantaggi, come per esempio la possibilità di scrivere funzioni *compagne* per iniettare messaggi informativi durante la ricorsione senza alterarne il codice principale:

```
let rec fib_rec x =
  Printf.printf "x: %d\n" x;
  fib_nrec fib_rec x
;;
(* signature: val fib_rec : int -> int = <fun>   *)
```

Quello che però noi vogliamo è un modo per applicare una *memoization* alla ricorsività, fare in modo quindi, che il parametro `fn` di `fib_nrec` sia una funzione in grado di ricordare i valori calcolati.
Per alcune problematiche inerenti al meccanismo di risoluzione e identificazione dei tipi in maniera non ambigua del compilatore, non possiamo usare il `rec` come sopra, ma abbiamo bisogno di trasformare il nostro modello

computazionale in una vera funzione ricorsiva.
Il problema è che la funzione da invocare dovrà essere conosciuta *a priori* e per far questo abbiamo bisogno di impostare un *valore funzionale mutabile* e fittizio che verrà sostituito in un secondo tempo.

```
(* funzione iniziale conosciuta a priori *)
let fn = ref (fun _ -> assert false);;
(* signature: val fn : ('_a -> '_b) ref = {contents = <fun>} *)

(* possiamo dichiarare adesso la funzione perché fn è conosciuto *)
let fib_rec x = fib_nrec !fn x;;
(* signature: val fib_rec : int -> int = <fun>*)

(* attiviamo la memorizzazione della funzione non ricorsiva *)
let fib_rec_memo = memoize fib_rec;;
(* signature: val fib_rec_memo : int -> int = <fun>  *)

(* sostituiamo la funzione nel valore mutabile *)
fn := fib_rec_memo
(* signature: - : unit = () *)

fib_rec_memo 20;;
```

Scriverà:

```
find: 20
find: 18
find: 16
find: 14
find: 12
find: 10
find: 8
find: 6
find: 4
find: 2
find: 0
not found - store: 0 with value: 0
find: 1
not found - store: 1 with value: 1
not found - store: 2 with value: 1
find: 3
find: 1
found - get: 1 with value: 1
find: 2
found - get: 2 with value: 1
not found - store: 3 with value: 2
not found - store: 4 with value: 3
find: 5
find: 3
found - get: 3 with value: 2
find: 4
found - get: 4 with value: 3
not found - store: 5 with value: 5
not found - store: 6 with value: 8
find: 7
find: 5
found - get: 5 with value: 5
find: 6
found - get: 6 with value: 8
not found - store: 7 with value: 13
not found - store: 8 with value: 21
find: 9
find: 7
found - get: 7 with value: 13
find: 8
found - get: 8 with value: 21
not found - store: 9 with value: 34
not found - store: 10 with value: 55
find: 11
```

```
find: 9
found - get: 9 with value: 34
find: 10
found - get: 10 with value: 55
not found - store: 11 with value: 89
not found - store: 12 with value: 144
find: 13
find: 11
found - get: 11 with value: 89
find: 12
found - get: 12 with value: 144
not found - store: 13 with value: 233
not found - store: 14 with value: 377
find: 15
find: 13
found - get: 13 with value: 233
find: 14
found - get: 14 with value: 377
not found - store: 15 with value: 610
not found - store: 16 with value: 987
find: 17
find: 15
found - get: 15 with value: 610
find: 16
found - get: 16 with value: 987
not found - store: 17 with value: 1597
not found - store: 18 with value: 2584
find: 19
find: 17
found - get: 17 with value: 1597
find: 18
found - get: 18 with value: 2584
not found - store: 19 with value: 4181
not found - store: 20 with value: 6765
- : int = 6765
```

Abbiamo così la nostra funzione ricorsiva in grado di memorizzare i propri valori.
Generalizzando per funzioni con un solo parametro:

```
let memoize_rec fn_nrec =
  let fn = ref (fun _ -> assert false) in
  let fn_rec_memo = memoize (fun x -> fn_nrec !fn x) in
  fn := fn_rec_memo;
  fn_rec_memo
;;
(* signature: val memoize_rec : ((int -> int) -> int -> int) -> int -> int = <fun> *)

let fib_memoized_rec = memoize_rec fib_nrec;;
(* signature: val fib_memoized_rec : int -> int = <fun> *)
```

La funzione per calcolare la progressione di Fibonacci però non è che sia un esempio poi così utile. Vediamo come lo stesso meccanismo sia invece interessante per calcolare altro come per esempio la *distanza di Levenshtein*. Il calcolo di questa *distanza* serve per determinare quanto due stringhe siano simili ed è stata formulata dal russo Vladimir Levenshtein nel 1965. Può essere applicata a varie problematiche che non siano solo le stringhe ma anche per controllare la similarità tra suoni, immagini o altro. È il *numero minimo di cambiamenti base* a livello di carattere per trasformare una stringa in un'altra:

- cancellazione
- inserimento
- sostituzione

Per esempio se vogliamo trasformare *pippo* in *pappa* dobbiamo sostituire:

- la prima *i* in *a*
- l'ultima *o* in *a*

quindi abbiamo una distanza di 2, perché due sono le operazioni da fare.
La formula di Levenshtein per calcolare la distanza di due stringhe a e b è espressa matematicamente in questo modo:

$$lev_{a,b}(i,j) = \begin{cases} max(i,j) & \text{se } min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{altrimenti} \end{cases} \tag{11}$$

In sé e per sé non è particolarmente complessa ed esprime le tre possibilità indicate sopra: il primo elemento del `min` rappresenta la cancellazione, il secondo l'inserimento ed il terzo la sostituzione nel caso in cui la lettera sia diversa.
L'indicazione:

$$1_{(a_i \neq b_j)} \tag{12}$$

sta a significare che si dovrà aggiungere `1` nel caso in cui le lettere siano diverse o `0` altrimenti (questo viene detto *costo*, *funzione indicatrice* o *funzione caratteristica* e indica l'appartenenza o no di un elemento ad un dato insieme matematico).
Se volessimo implementarla, si potrebbe procedere semplicemente seguendo la formula data:

```
let lev2 a b =
  let minimum x y z = min x (min y z) in
  let cost i j =
    if a.[i - 1] = b.[j - 1] then 0 else 1
  in
  let rec distance i j =
    match (i, j) with
    | (i, 0) -> i
    | (0, j) -> j
    | (i, j) ->
      let deletion = (distance (i - 1) j) + 1 in
      let insertion = (distance i (j - 1)) + 1 in
      let substitution = (distance (i - 1) (j - 1)) + (cost i j) in
      minimum deletion insertion substitution
  in
  distance (String.length a) (String.length b)
```

Si controlla inizialmente che una delle stringhe sia nulla (lunghezza zero) e quindi la distanza sarà inevitabilmente la lunghezza dell'altra. Dopo si applicheranno ricorsivamente le regole indicate per i tre casi, compreso il calcolo del *costo* nella possibile sostituzione.
Questa versione però per motivi abbastanza evidenti (surplus di calcolo anche in caso di lettere uguali) non è particolarmente performante:

```
utop # time (fun () -> lev "levenshtein" "levenshtain");;
Time: 4.502269s
- : int = 1
```

Migliorando l'implementazione:

```
let lev s t =
  let minimum a b c = min a (min b c) in
  let rec distance i j =
    match (i, j) with
    | (i, 0) -> i
```

```
    | (0, j) -> j
    | (i, j) when s.[i - 1] = t.[j - 1] -> distance (i - 1) (j - 1)
    | (i, j) -> 1 + (minimum (distance (i - 1) j) (distance i (j - 1)) (distance (i - 1) (j - 1)))
  in
  distance (String.length s) (String.length t)
```

Questa forma non è cambiata di molto ma il guadagno in velocità è notevole:

```
utop # time (fun () -> lev "levenshtein" "levenshtain");;
Time: 0.011653s
- : int = 1
```

Il nodo problematico era il calcolo del *costo* inserito nel *minimo* che adesso semplicemente evitiamo o meglio mascheriamo col penultimo caso: se le lettere sono uguali passeremo oltre visto che noi stiamo cercando le differenze.

Questa implementazione del calcolo della *distanza di Levenshtein* è un ottimmo candidato per la *memoization*, avendo numerose chiamate ricorsive che elaborano nella maggior parte cose già calcolate, quindi risultati noti in precedenza.
Applicando come abbiamo visto sopra, lo stesso meccanismo usato per la sequenza di Fibonacci, potremmo avere un codice di questo tipo:

```
let lev_memo s t =
  let minimum a b c = min a (min b c) in
  let distance fn i j =
    match (i, j) with
    | (i, 0) -> i
    | (0, j) -> j
    | (i, j) when s.[i - 1] = t.[j - 1] -> fn (i - 1) (j - 1)
    | (i, j) -> 1 + (minimum (fn (i - 1) j) (fn i (j - 1)) (fn (i - 1) (j - 1)))
  in
  let memo fn i j =
    let fn_ref = ref (fun _ -> assert false) in
    let f = memoize (fun i j -> fn !fn_ref i j) 12 in
    fn_ref := f;
    f i j
  in
  (memo distance) (String.length s) (String.length t)
```

Queste saranno le performance tra la versione *normale* e quella *memoized*:

```
utop # time (fun () -> lev "levenshtein" "levenshtain");;
Time: 0.009528s
- : int = 1

utop # time (fun () -> lev_memo "levenshtein" "levenshtain");;
Time: 0.001963s
- : int = 1
```

I tempi, indipendentemente da quello che valgono queste misurazioni, si commentano da soli.

Come suggerimento implementativo dove si è parlato di *funzioni anonime* si è accennato a *y-combinator*, qualche idea? Vediamo un esempio con la sequenza di Fibonacci:

```
type 'a b = Roll of ('a b -> 'a)
let unroll (Roll x) = x

let y f = ((fun g -> g (Roll g)) (fun x n -> f (unroll x x) n))

let memoize f' =
  let table = Hashtbl.create 12 in
  fun f x -> begin
      Printf.printf "find?: %d\n" x;
      try
```

```
        let r = Hashtbl.find table x in
        Printf.printf "found - get: %d with value: %d\n" x r;
        r
      with Not_found ->
        let r = f' f x in
        Printf.printf "not found - store: %d with value: %d\n" x r;
        Hashtbl.add table x r;
        r
    end

let fib' fib x =
  match x with
    | 0 -> 0
    | 1 -> 1
    | _ -> fib (x - 1) + fib (x - 2)

let fib_rec_memo x = y (memoize fib') x
```

Per amor di informazione, l'algoritmo di Levenshtein è implementabile anche in altri modi e sfruttando le caratteristiche *imperative* di OCaml si potrebbe scrivere una cosa simile (ed avrà prestazioni elevate rispetto anche alla forma *memoized*):

```
let lev_iter s t =
  let n = String.length s in
  let m = String.length t in
  let d = Array.make_matrix (n + 1) (m + 1) 0 in
  for i = 1 to n do d.(i).(0) <- i done;
  for j = 1 to m do d.(0).(j) <- j done;
  let minimum x y z = min x (min y z) in
  Array.iteri (fun i a ->
      if i > 0 then
        Array.iteri (fun j _ ->
            if j > 0 then
              begin
                let cost = if s.[i - 1] = t.[j - 1] then 0 else 1 in
                let deletion = d.(i - 1).(j) + 1 in
                let insertion = d.(i).(j - 1) + 1 in
                let substitution = d.(i - 1).(j - 1) + cost in
                d.(i).(j) <- minimum deletion insertion substitution
              end
          ) a;
    ) d;
  d.(n).(m)
```

Con questi tempi:

```
iter version
Execution time: 0.000006s
2
memo version
Execution time: 0.000037s
2
slow version
Execution time: 0.673041s
2
```

3.14 S-Expressions

Le *s-expressions* [25] [26] o *sexpr* sono un metodo per rappresentare in forma testuale dati strutturati (serializzazione dei dati [27]). Sono abbastanza comuni nel mondo funzionale specialmente nella famiglia dei vari **LISP**, il cui codice sorgente è esso stesso una *s-expression*. I vari dialetti LISP infatti hanno questa particolarità.

[25] http://en.wikipedia.org/wiki/John_McCarthy_%28computer_scientist%29
[26] http://people.csail.mit.edu/rivest/Sexp.txt
[27] http://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats

Sono meno comuni in altri linguaggi, specialmente con la disponibilità di **XML** [28], **JSON** [29] o **YAML** [30], ma offono alcuni vantaggi rispetto a questi:

- Una *sexpr* è più semplice da leggere.
- Non c'è bisogno di generare *parser* appositi (usando *lex* o *yacc*).
- Possono essere usate per definire dei mini linguaggi *LISP like* da usare nei programmi.

Le *sexpr* possono essere dei semplici *atomi* (l'atomo in LISP è definito di base come: una sequenza di uno o più caratteri), elementi fondamentali come numeri o *coppie* espresse nella forma `(a . b)` dove `a` e `b` sono *s-expression*.

Vediamo la stessa struttura dati in vari modi.

JSON:

```
{"dict" : {
    "name" : "Pico",
    "surname" : "dePaperis",
    "age" : 65,
    "tags" : ["personaggio","fumetti"]
    "locality" : {
        "address" : "via del Campo",
        "number" : 12,
        "city" : "Paperopoli"
        }
    }
}
```

XML:

```
<dict>
  <name>Pico</name>
  <surname>dePaperis</surname>
  <age>65</age>
  <tags>
    <tag>personaggio</tag>
    <tag>fumetti</tag>
  </tags>
  <locality>
    <address>via del Campo</address>
    <number>12</number>
    <city>Paperopoli</city>
  </locality>
</dict>
```

YAML:

```
---
dict:
  name: Pico
  surname: dePaperis
  age: 65
  tags:
    - personaggio
    - fumetti
  location:
    address: via del Campo
    number: 12
    city: Paperopoli
```

s-expressions:

[28] http://it.wikipedia.org/wiki/XML
[29] http://json.org/
[30] http://yaml.org/

```
(dict
    (name "Pico")
    (surname "dePaperis")
    (age 65)
    (tags (personaggio fumetti))
    (locality (dict
      (address "via del Campo")
      (number 12)
      (city "Paperopoli"))))
```

A voi giudicare quale sia il sistema più semplice da leggere ad occhio, ma ricordate che una *s-expression* è soltanto una lista annidata. A mio parere la *sexpr* e la forma *yaml* sono quelle che hanno meno *rumore* sintattico. La *sexpr* è praticamente un sorgente LISP: `dict` sarà una *keyword* che denota un dizionario, le stringhe sono chiuse tra doppi apici, `tags` è una lista *quotata* di `atoms`, `locality` un ulteriore dizionario.

Per ricapitolare le regole:

- Un `atom` è un simbolo è essenzialmente un valore semplice (*atomico* appunto), come `foo`, `a-bar`, `12`.
- Le liste sono tra parentesi tonde `(..)`, possono essere vuote, contenere `atomi` separati da uno spazio o da un punto o altre liste.
- Le *quote* sono per gli atomi che contengono parentesi o spazi.
- Il *backslash* è il carattere per lo *escape*.
- Il punto e virgola introduce i commenti.

Molti linguaggi funzionali hanno delle librerie di supporto (interne od esterne) per le *s-expressions*, ma comunque non è complicato programmare un parser come potete anche vedere qui: Rosetta Code, s-expressions.

La libreria Core[7] ha un modulo apposito `Sexp` per la loro gestione perché è utilizzato anche come *serializzatore* interno.

```
open Core.Std
open Sexplib.Std

type person_locality = {
  address : string;
  number : int;
  city : string;
} with sexp
;;
(*
type person_locality = { address : bytes; number : int; city : bytes; }
val person_locality_of_sexp : Type.t -> person_locality = <fun>
val sexp_of_person_locality : person_locality -> Type.t = <fun>
*)

type person = {
  name : string;
  surname : string;
  age : int;
  tags : string list;
  locality : person_locality;
} with sexp
;;
(*
type person = { name : bytes; surname : bytes; age : int; tags : bytes list; locality : person_locality; }
val person_of_sexp : Type.t -> person = <fun>
val sexp_of_person : person -> Type.t = <fun>
*)

let person_to_sexp person =
  let atom x = Sexp.Atom x and list x = Sexp.List x in
  list [
```

```
    list [ atom "name"; String.sexp_of_t person.name  ];
    list [ atom "surname"; String.sexp_of_t person.surname];
    list [ atom "age"; Int.sexp_of_t person.age ];
    list [ atom "tags"; List.sexp_of_t String.sexp_of_t person.tags];
    list [ atom "locality";
           list [
             list [ atom "address"; String.sexp_of_t person.locality.address ];
             list [ atom "number"; Int.sexp_of_t person.locality.number ];
             list [ atom "city"; String.sexp_of_t person.locality.city];
           ];
         ];

  ]
;;
(* val person_to_sexp : person -> Type.t = <fun> *)

let sexp_source = "
  ((name \"Pico\")
    (surname \"dePaperis\")
    (age 65)
    (tags (personaggio fumetti))
    (locality (
      (address \"via del Campo\")
      (number 12)
      (city \"Paperopoli\"))))"
;;
let se1 = Sexp.of_string sexp_source;;
(*
val se1 : Type.t =
 ((name Pico)
  (surname dePaperis)
  (age 65)
  (tags (personaggio fumetti))
  (locality (
    (address "via del Campo")
    (number 12)
    (city Paperopoli))))
*)

let se2 = person_to_sexp {
    name = "Pico";
    surname = "dePaperis";
    age = 65;
    tags = ["personaggio"; "fumetti"];
    locality = {
      address = "via del campo";
      number = 12;
      city = "Paperopoli"
    }
  }
;;
(*
val se2 : Type.t =
 ((name Pico)
  (surname dePaperis)
  (age 65)
  (tags (personaggio fumetti))
  (locality (
     (address "via del campo")
     (number 12)
     (city Paperopoli))))
*)

let person_data = {
  name = "Pico";
  surname = "dePaperis";
  age = 65;
  tags = ["personaggio"; "fumetti"];
```

```
  locality = {
    address = "via del campo";
    number = 12;
    city = "Paperopoli";
  }
}
;;
(*
val person_data : person =
{
  name = "Pico";
  surname = "dePaperis";
  age = 65;
  tags = ["personaggio"; "fumetti"];
  locality = {
    address = "via del campo";
    number = 12;
    city = "Paperopoli"
    }
}
*)

let se3 = sexp_of_person person_data;;
(*
val se3 : Type.t =
 ((name Pico)
  (surname dePaperis)
  (age 65)
  (tags (personaggio fumetti))
  (locality (
    (address "via del campo")
    (number 12)
    (city Paperopoli))))
*)

let se3_deser = person_of_sexp se3;;
(*
val se3_deser : person =
{
  name = "Pico";
  surname = "dePaperis";
  age = 65;
  tags = ["personaggio"; "fumetti"];
  locality = {
    address = "via del campo";
    number = 12;
    city = "Paperopoli"
    }
}
*)

let () =

  printf "\nPerson data\n";
  printf "  name: %s\n" se3_deser.name;
  printf "  surname: %s\n" se3_deser.surname;
  printf "  age: %d\n" se3_deser.age;
  printf "  tags: %s\n" (String.concat ~sep:"," se3_deser.tags);
  printf "  locality\n";
  printf "    address: %s\n" se3_deser.locality.address;
  printf "    number: %d\n" se3_deser.locality.number;
  printf "    city: %s\n" se3_deser.locality.city;

  printf "=========================================0\n";

  printf "Sexp from source string:\n";
  printf "\n%s\n" (Sexp.to_string se1);
  printf "=========================================0\n";
```

```
169
170   printf "Sexp programatically from struct:\n";
171   printf "\n%s\n" (Sexp.to_string se2);
172   printf "=========================================0\n";
173
174   printf "Sexp Sexplib emitter from struct:\n";
175   printf "\n%s\n" (Sexp.to_string (se3));
```

In questo esempio si è *serializzata* e *deserializzata* una struttura (*struct*) per dimostrarne il funzionamento.

```
1  ./sexpr_test.native
2
3  Person data
4    name: Pico
5    surname: dePaperis
6    age: 65
7    tags: personaggio,fumetti
8    locality
9      address: via del campo
10     number: 12
11     city: Paperopoli
12 =========================================0
13 Sexp from source string:
14
15 ((name Pico)(surname dePaperis)(age 65)(tags(personaggio fumetti))
16 (locality((address"via del Campo")(number 12)(city Paperopoli))))
17 =========================================0
18 Sexp programatically from struct:
19
20 ((name Pico)(surname dePaperis)(age 65)(tags(personaggio fumetti))
21 (locality((address"via del campo")(number 12)(city Paperopoli))))
22 =========================================0
23 Sexp Sexplib emitter from struct:
24
25 ((name Pico)(surname dePaperis)(age 65)(tags(personaggio fumetti))
26 (locality((address"via del campo")(number 12)(city Paperopoli))))
```

Come ho specificato prima è stata usata Core[7], ma Batteries[17] per esempio ha un supporto analogo. Il modulo `Sexplib` è molto sofisticato e va molto al di là del semplice codice che potrete aver trovato su Rosetta Code; come si può vedere nelle *signature* delle varie parti `Sexplib`, per esempio, genera una buona quantità di funzioni *helper* che facilitano di molto il compito.

3.15 Velocità di esecuzione

I linguaggi funzionali sono spesso accusati di essere lenti. Chi lo fa non lo dice generalmente a ragion veduta e si riferisce alle prime implementazioni *arcaiche* degli interpreti LISP. Linguaggi come OCaml, per esempio, sono estremamente efficienti nonostante la presenza di un *garbage collector*.

In questa sezione vi mostrerò varie versioni dello stesso codice con un *benchmark* casalingo. È chiaro che, come tutti i *benchmark* non abbia un valore assoluto e debba essere preso *con le molle*.

Vediamo una prima implementazione:

```
1  let time f =
2    let start = Sys.time () in
3    let x = f () in
4    let stop = Sys.time () in
5    Printf.printf "Time: %fs\n%!" (stop -. start);
6    x
7  ;;
8
9  let rec fib x =
10   match x with
11   | 0 -> 0
12   | 1 -> 1
13   | x -> (fib (x - 1)) + (fib (x - 2))
```

```
;;

let range a b =
  let rec aux a b =
    if a > b then [] else a :: aux (a + 1) b  in
  if a > b then List.rev (aux b a) else aux a b
;;

let fib_range range =
  let print_value value =
    let r = fib value in
    Printf.fprintf stdout "fib (%d): %d\n" value r;
    flush stdout
  in
  List.iter print_value range
;;

let () =
  time (fun () -> fib_range (range 0 39));
;;
```

In questo codice si cerca di mostrare quanto tempo serva per calcolare dei numeri di Fibonacci per quaranta interi in successione.
Compilando ed eseguendo il programma avremo questo risultato:

```
$  ./fib.native
fib (0): 0
fib (1): 1
fib (2): 1
fib (3): 2
fib (4): 3
fib (5): 5
fib (6): 8
fib (7): 13
fib (8): 21
fib (9): 34
fib (10): 55
fib (11): 89
fib (12): 144
fib (13): 233
fib (14): 377
fib (15): 610
fib (16): 987
fib (17): 1597
fib (18): 2584
fib (19): 4181
fib (20): 6765
fib (21): 10946
fib (22): 17711
fib (23): 28657
fib (24): 46368
fib (25): 75025
fib (26): 121393
fib (27): 196418
fib (28): 317811
fib (29): 514229
fib (30): 832040
fib (31): 1346269
fib (32): 2178309
fib (33): 3524578
fib (34): 5702887
fib (35): 9227465
fib (36): 14930352
fib (37): 24157817
fib (38): 39088169
fib (39): 63245986
Time: 1.039338s
```

Per ogni numero viene invocata la funzione `fib` e se ne stampa il valore, alla fine il tempo occorso. Cambiamo adesso con una versione in cui `fib`, che è una funzione ricorsiva, è implementata in *tail recursion* usando un accumulatore:

```
let time f =
  let start = Sys.time () in
  let x = f () in
  let stop = Sys.time () in
  Printf.printf "Time: %fs\n%!" (stop -. start);
  x
;;

let fib x =
  let rec fib_rec x acc piv =
    match x with
    | 0 -> acc
    | _ -> fib_rec (x - 1) piv (acc + piv)
  in
  fib_rec x 0 1
;;

let range a b =
  let rec aux a b =
    if a > b then [] else a :: aux (a + 1) b  in
  if a > b then List.rev (aux b a) else aux a b
;;

let fib_range range =
  let print_value value =
    let r = fib value in
    Printf.fprintf stdout "fib (%d): %d\n" value r;
    flush stdout
  in
  List.iter print_value range
;;

let () =
  time (fun () -> fib_range (range 0 39));
;;
```

Il codice, come si può vedere, è pressoché identico a parte `fib`. Eseguendo e stupendoci:

```
$  ./fib.native
fib (0): 0
fib (1): 1
fib (2): 1
fib (3): 2
fib (4): 3
fib (5): 5
fib (6): 8
fib (7): 13
fib (8): 21
fib (9): 34
fib (10): 55
fib (11): 89
fib (12): 144
fib (13): 233
fib (14): 377
fib (15): 610
fib (16): 987
fib (17): 1597
fib (18): 2584
fib (19): 4181
fib (20): 6765
fib (21): 10946
fib (22): 17711
fib (23): 28657
fib (24): 46368
```

```
fib (25): 75025
fib (26): 121393
fib (27): 196418
fib (28): 317811
fib (29): 514229
fib (30): 832040
fib (31): 1346269
fib (32): 2178309
fib (33): 3524578
fib (34): 5702887
fib (35): 9227465
fib (36): 14930352
fib (37): 24157817
fib (38): 39088169
fib (39): 63245986
Time: 0.000274s
```

Notato il tempo? Questa versione in *tail call* con accumulatore è decisamente più veloce.
Come nota vi dico che se avessi usato la libreria **Core** [7] avrei avuto prestazioni migliori (almeno nei miei test).
Vediamo OCaml rispetto ad altri linguaggi oggi di moda.

Go [31]

```go
package main

import (
  "fmt"
  "time"
)

func fib(n int) int {
  if n < 2 {
    return n
  }
  return fib(n - 1) + fib(n - 2)
}

func main() {
  start := time.Now()
  for n := 0; n < 40; n++ {
    fmt.Printf("fib (%d) = %d\n", n, fib(n))
  }
  fmt.Printf("Time %s\n", time.Since(start))
}
```

Dopo averlo compilato ed eseguito:

```
$  ./fib
fib (0) = 0
fib (1) = 1
fib (2) = 1
fib (3) = 2
fib (4) = 3
fib (5) = 5
fib (6) = 8
fib (7) = 13
fib (8) = 21
fib (9) = 34
fib (10) = 55
fib (11) = 89
fib (12) = 144
fib (13) = 233
fib (14) = 377
fib (15) = 610
fib (16) = 987
```

[31] http://golang.org/

```
fib (17) = 1597
fib (18) = 2584
fib (19) = 4181
fib (20) = 6765
fib (21) = 10946
fib (22) = 17711
fib (23) = 28657
fib (24) = 46368
fib (25) = 75025
fib (26) = 121393
fib (27) = 196418
fib (28) = 317811
fib (29) = 514229
fib (30) = 832040
fib (31) = 1346269
fib (32) = 2178309
fib (33) = 3524578
fib (34) = 5702887
fib (35) = 9227465
fib (36) = 14930352
fib (37) = 24157817
fib (38) = 39088169
fib (39) = 63245986
Time 1.505562721s
```

In questa versione con `fib` semplicemente ricorsiva OCaml è notevolmente più veloce. Va notata anche una cosa in OCaml non abbiamo un costruttore per dei tipi Range come in **Go** ed è più performante nonostante debba perdere tempo nella costruzione di una lista di interi.

Vediamo il **Go** ottimizzato alla stessa maniera di OCaml:

```go
package main

import (
  "fmt"
  "time"
)

func fib_rec(n int, acc int, piv int) int {
  if n < 1 {
    return acc
  }
  return fib_rec((n - 1), piv, (acc + piv))
}

func fib(n int) int {
  return fib_rec(n, 0, 1)
}

func main() {
  start := time.Now()
  for n := 0; n < 40; n++ {
    fmt.Printf("fib (%d) = %d\n", n, fib(n))
  }
  fmt.Printf("Time %s\n", time.Since(start))
}
```

Compile and run…

```
$  ./fib
fib (0) = 0
fib (1) = 1
fib (2) = 1
fib (3) = 2
fib (4) = 3
fib (5) = 5
fib (6) = 8
fib (7) = 13
```

```
fib (8) = 21
fib (9) = 34
fib (10) = 55
fib (11) = 89
fib (12) = 144
fib (13) = 233
fib (14) = 377
fib (15) = 610
fib (16) = 987
fib (17) = 1597
fib (18) = 2584
fib (19) = 4181
fib (20) = 6765
fib (21) = 10946
fib (22) = 17711
fib (23) = 28657
fib (24) = 46368
fib (25) = 75025
fib (26) = 121393
fib (27) = 196418
fib (28) = 317811
fib (29) = 514229
fib (30) = 832040
fib (31) = 1346269
fib (32) = 2178309
fib (33) = 3524578
fib (34) = 5702887
fib (35) = 9227465
fib (36) = 14930352
fib (37) = 24157817
fib (38) = 39088169
fib (39) = 63245986
Time 733.419us
```

Il tempo riporta 733,419 microsecondi ovvero 0,000733419 secondi contro gli 0.000274 secondi di OCaml. **GO** non supporta a pieno la *tail call optimization* ma in ogni caso implementando la funzione in questa maniera si ottengono notevoli benefici.

Si fa un gran parlare di **Rust** [32] oggi, il nuovo linguaggio di Mozilla[33] approdato da qualche giorno alla versione 1.0.
Rust è un linguaggio molto promettente e multiparadigma con una particolarità: non ha un garbage collector. Utilizza un sofisticato sistema di *appartenenza, prestito e durata (Ownership, Borrowing, Lifetimes)* delle allocazioni. Si propone come un linguaggio *sicuro e veloce*.

Vediamo in Rust:

```
extern crate time;
use time::PreciseTime;
use std::ops::Range;

fn fib(x : u32) -> u32 {
    match x {
        0 => 0,
        1 => 1,
        _ => fib(x - 1) + fib(x - 2)
    }
}

fn fib_range(r : Range<u32>) {
    for i in r {
        println!("fib ({:?}) = {:?}", i, fib(i));
    }
}

```

[32] http://www.rust-lang.org/
[33] https://www.mozilla.org

```
fn exec_and_time<F>(f : F ) where F: Fn() {
    let start = PreciseTime::now();
    f();
    let stop = PreciseTime::now();
    println!("Time: {}s", start.to(stop).num_milliseconds() as f64 / 1000 as f64);
}

fn main() {
    exec_and_time(||{fib_range(0..40)});
}
```

La funzione `fib` è la solita, dopo aver compilato ed eseguito (ometto un po' di numeri che ormai l'output si è capito):

```
$  ./fib`
fib (0) = 0
fib (1) = 1
fib (2) = 1
fib (3) = 2
fib (4) = 3
fib (5) = 5
fib (6) = 8
fib (7) = 13
fib (8) = 21
fib (9) = 34
fib (10) = 55
fib (11) = 89
...
Time: 1.402s
```

Come al solito la versione ottimizzata:

```
extern crate time;
use time::PreciseTime;
use std::ops::Range;

fn fib(x: i32) -> i32 {
    fn _fib(x: i32, acc: i32, piv: i32) -> i32 {
        match (x, acc, piv) {
            (0, _, _) => acc,
            _         => _fib(x - 1, piv, acc + piv)
        }
    }
    _fib(x, 0, 1)
}

fn fib_range(r : Range<i32>) {
    for i in r {
        println!("fib ({:?}) = {:?}", i, fib(i));
    }
}

fn exec_and_time<F>(f : F ) where F: Fn() {
    let start = PreciseTime::now();
    f();
    let stop = PreciseTime::now();
    println!("Time: {}s", start.to(stop));
}

fn main() {
    exec_and_time(||{fib_range(0..40)});
}
```

Produrrà:

```
...
fib (34) = 5702887
```

```
fib (35) = 9227465
fib (36) = 14930352
fib (37) = 24157817
fib (38) = 39088169
fib (39) = 63245986
Time: PT0.000367682Ss
```

La cosa interessante è nella comparazione tra Rust ed OCaml e come le prestazioni siano simili nonostante il *garbage collector* del secondo. Nella versione ricorsiva semplice comunque Rust è decisamente indietro.

Per concludere e non tediarvi troppo vediamo **Nim** [34], che è per qualche verso un diretto concorrente di Rust (a mio parere).

```
import strutils
import times

proc fib(x: int): int {. noSideEffect .} =
  case x
  of 0: 0
  of 1: 1
  else: fib(x - 1) + fib(x - 2)

proc fib_range(r: Slice[int]) =
  for i in r:
    echo "fib ($1): $2" % [i.intToStr, fib(i).intToStr]

proc time(f: proc () ) =
  let start = cpuTime()
  f()
  let stop = cpuTime()
  echo "Time: ", stop - start, "s"

time(proc () = fib_range(0..39))
```

Nella consueta versione ricorsiva semplice produce questi risultati:

```
...
fib (35): 9227465
fib (36): 14930352
fib (37): 24157817
fib (38): 39088169
fib (39): 63245986
Time: 0.6634450000000001s
```

Mentre nella versione ottimizzata:

```
import strutils
import times

proc fib(n: int, acc: int, piv: int): int =
  if n == 0:
    acc
  else:
    fib(n - 1, piv, acc + piv)

proc fib(n: int): int =
  fib(n, 0, 1)

proc fib_range(r: Slice[int]) =
  for i in r:
    echo "fib ($1): $2" % [i.intToStr, fib(i).intToStr]

proc time(f: proc () ) =
  let start = cpuTime()
  f()
  let stop = cpuTime()
```

[34] http://nim-lang.org/

```
  echo "Time: ", stop - start, "s"

time(proc () = fib_range(0..39))
```

```
...
fib (35): 9227465
fib (36): 14930352
fib (37): 24157817
fib (38): 39088169
fib (39): 63245986
Time: 0.000209s
```

Nim è di fatto un generatore di codice **C** molto ottimizzato, quindi ha prestazioni paragonabili.

In un ultimo esempio (è vero avevo già detto di smettere) ci proviamo con **Ruby** [35]. Ruby è un linguaggio interpretato (anche se non è proprio vero) multiparadigma con la nomea della lentezza.

Vediamo la solita implementazione iniziale:

```
def fib(x)
  case x
  when 0
    0
  when 1
    1
  else
    fib(x - 1) + fib(x - 2)
  end
end

def fib_range(r)
  r.each { |n|
    puts "fib (#{n}): #{fib(n)}"
  }
end

def time(&f)
  start = Time.now
  yield f
  stop = Time.now
  puts "Time: #{stop - start}s"
end

if __FILE__ == $0
  time { fib_range(0..39) }
end
```

Risultato:

```
...
fib (34): 5702887
fib (35): 9227465
fib (36): 14930352
fib (37): 24157817
fib (38): 39088169
fib (39): 63245986
Time: 33.736951485s
```

Trentatre secondi e rotti. Decisamente lento no?

Ora ottimizziamo come per gli altri:

```
def fib(x)
  def fib_rec(x, acc, piv)
    case x
    when 0
      acc
```

[35] https://www.ruby-lang.org

```
    else
      fib_rec(x - 1, piv, acc + piv)
    end
  end
  fib_rec(x, 0, 1)
end

def fib_range(r)
  r.each { |n|
    puts "fib (#{n}): #{fib(n)}"
  }
end

def time(&f)
  start = Time.now
  yield f
  stop = Time.now
  puts "Time: #{stop - start}s"
end

if __FILE__ == $0
  time { fib_range(0..39) }
end
```

Signori e Signore, bambini e ragazzi…

```
...
fib (34): 5702887
fib (35): 9227465
fib (36): 14930352
fib (37): 24157817
fib (38): 39088169
fib (39): 63245986
Time: 0.000417195s
```

Decisamente impressionante.

Concludo questa tediosa sfilza di numeri che, pur lasciando il tempo che trova, penso dimostri vagamente come OCaml sia estremamente efficiente (ma anche che il codice si può scrivere in tanti modi).
Non prendete le cose troppo sul serio, però, questi sono giocattoli e non devono avere valenza assoluta.

I *benchmark* sono cose da valutare su larga scala con codici complessi ed a volte *si vince* ed a volte *si perde*: la perfezione non è di questo mondo. Se volete divertirvi ancora, visitate questo sito web: http://benchmarksgame.alioth.debian.org/

Il codice di questi esempi lo trovate qui: https://github.com/minimalprocedure/fib_test

3.16 Conclusioni

In questo testo ho cercato di presentare nel modo più semplice, alcuni dei vantaggi che può offrire il *paradigma funzionale* senza niente togliere agli altri. Come ho detto all'inizio, non voleva essere un corso di programmazione quanto una semplice presentazione, un cercare di convincere a tentare un approccio diverso al solito sviluppo del software.
Molte delle problematiche che ci sono oggi, sono affrontabili in maniera più chiara e maggiormente mantenibile una volta comprese alcune metodiche.
Lo sviluppo e la disponibilità di linguaggi di programmazione multiparadigma offre molte possibilità per poter utilizzare la giusta, o almeno la più adatta, soluzione al problema che ci viene posto. L'orientamento funzionale o quello ad oggetti non sono mutualmente esclusivi; ma se usati bene, complementari per un software che sia mantenibile al meglio nel tempo.

Qualcuno potrà obiettare sicuramente sulla mia scelta di utilizzare OCaml per presentare gli esempi e non altri; come Scala per esempio, che sta godendo di un roseo momento. Senza togliere niente agli altri (personalmente adoro Scala), trovo OCaml particolarmente elegante e conciso, non dimenticando che è compilabile nativamente

con performance del tutto dignitose da non far rimpiangere il C/C++.
Un linguaggio *pragmatico*, estremamente espressivo e potente.

Si lamenterà della mancanza del supporto *nativo* per **Unicode** (che in realtà non è nemmeno propriamente vero), ci sono ottime librerie per questo e non se ne sentirà la (probabile) mancanza. Si criticherà il problema della computazione parallela (ma a presto disponibile[36]), imputabile al suo *garbage collector* particolare ma anche questo facilmente superabile grazie a numerose librerie tra cui LWT[8] e Async[7]. OCaml gode di una comunità molto spesso gentile e disponibile al contrario di altre e questo non è cosa da poco. In OCaml sono scritti linguaggi di programmazione (per citare Haxe[37] o Hack[38] di Facebook, Rust[39] è stato inizialmente scritto in OCaml), risolutori di teoremi matematici, virtualizzatori e software *mission critical*.

È usato da aziende come Citrix[40] (XenServer), Jane Street[41], Bloomberg[42], Dassault Aviation[43] e Dassault Systemes[44], Microsoft e Facebook.

Comunque sia, prendetevi uno dei tanti *funzionali* e cercate di cambiare il modo di pensare la programmazione, ne varrà la pena.

4 Appendici

4.1 Paradigma di programmazione

- In informatica un **paradigma di programmazione** è uno stile fondamentale di programmazione.
- È un insieme di strumenti concettuali forniti dal linguaggio ed una serie di *best practice* da usare per la stesura del codice sorgente di un programma.
- I **paradigmi di programmazione** si differenziano nei concetti e nelle astrazioni usate per rappresentare gli elementi di un programma.
- È possibile identificare un processo ereditario tra i vari paradigmi e quindi un **albero genealogico**.

4.1.1 Linguaggi di programmazione

- Ogni linguaggio di programmazione viene sviluppato seguendo una idea in qualche modo riconducibile ad un particolare paradigma.
- Esistono linguaggi considerati **puri** e linguaggi **impuri** cioè che non seguono in maniera stretta (pura) il paradigma.
- Alcuni linguaggi si ispirano a due o più paradigmi: **linguaggi multiparadigma**.

4.1.2 Paradigmi più comuni

- Esistono molti paradigmi di programmazione che sono stati pensati e sviluppati nel tempo, alcuni hanno avuto più successo di altri.
- Il successo di una *pratica di programmazione* è indipendente dalla sua *bontà intrinseca*, ma deve essere fatto ricadere in più fattori:
  - momento storico
  - capacità di chi la usa.

[36] http://kcsrk.info/multicore/gc/2017/07/06/multicore-ocaml-gc/
[37] http://haxe.org/
[38] http://hacklang.org/
[39] http://www.rust-lang.org/
[40] http://www.citrix.com/
[41] https://www.janestreet.com/
[42] http://www.bloomberg.com/
[43] http://www.dassault-aviation.com/
[44] http://www.3ds.com

  - richieste industriali
  - visione del futuro.
  - management

4.1.2.1 Programmazione procedurale

- **~1960**
- Blocchi di codice esecutivo identificati da un nome.
- I blocchi sono racchiusi in un ambito di visibilità da delimitatori.
- I blocchi possono avere variabili e parametri che sono persi all'uscita.
- I blocchi possono avere valori in uscita.
- Subroutine, procedure, funzioni.
- **Fortran**, **COBOL**.
- I linguaggi moderni permettono la scrittura di procedure e funzioni (alcuni come il Pascal le distinguono, altri come il C no).

4.1.2.2 Programmazione strutturata

- **~1960/1970**
- È la base per altri tipi di programmazione imperativa (es: **Object Oriented Programming (OOP)**).
- Critica del *goto* e *spaghetti code*.
- Nata basandosi sul teorema di Corrado Böhm e Giuseppe Jacopini che afferma come si possa scrivere un programma senza *goto* solo con delle strutture di controllo.
- Strutture di controllo: **sequenza** (una istruzione dopo l'altra, sequenza imperativa), **selezione** (*if-then-else*, *operatori ternari ?:*, *if a tre vie* del FORTRAN 2, *case/switch*), **ciclo** (*while-do*, *do-until*, *for-do*).
- Sviluppo della applicazione di **algoritmi** alla programmazione.
- Linguaggi tipizzati.
- **Pascal**, **Ada**, **Algol**, **C**.

4.1.2.3 Programmazione modulare

- **~1970**
- Suddivisione del programma in parti chiamate moduli.
- I moduli sono indipendenti.
- I moduli sono opachi internamente e possono idealmente vivere a se stanti.
- I moduli hanno una interfaccia per comunicare con l'esterno.
- **Modula**, **Modula-2**, **Oberon**, **CLU**, **Ada**, **Pascal**.
- Linguaggi tipizzati.
- La maggior parte dei linguaggi moderni la permettono in qualche maniera.

4.1.2.4 Programmazione funzionale

- **~1970**
- Incentrata sul concetto di **funzione** nel senso matematico del termine. (dominio (x), codominio (y) e relazione (f): y = f(x)).
- Le **funzioni** sono *valori*.
- **Trasparenza referenziale** (assenza di effetti collaterali, lo stato esterno della funzione non è alterato. Il valore di ritorno dati gli stessi parametri è garantito in qualunque caso).
- La mancanza di effetto collaterale la rende automaticamente *thread-safe*.
- **Ricorsività** e **funzioni ricorsive**, in particolare la *tail recursion* che limita l'allocazione dello stack ed il recupero del blocco esecutivo.
- **Funzioni di ordine superiore (high-order)**, le funzioni possono prendere funzioni come parametri e restituire funzioni come risultato).
- **Funzioni anonime**, **closure**.
- **Tipi parametrici**.
- **Generalized and extended algebraic data type**, permette di creare tipi compositi, tipi varianti.
- **Currying**, funzione con parametri multipli in funzioni con parametri singoli (possibili le **partial application** o funzioni semiprecalcolate).
- **Pattern matching**, *switch/case on steroids*.
- **Decomposizione di strutture complesse**.
- **List/set/for comprehension**, map, filter, reduce, collect, ciclo for parametrico.
- **Strict** o **lazy**, computazione dei valori e parametri prima della funzione o solo a richiesta.
- Linguaggi tipizzati, inferenza di tipo, dinamici, puri (con Monadi), impuri.
- **Lisp**, **Clojure**, **Scheme**, **Racket**, **Scala**, **Standard ML**, **OCaml**, **F#**, **Erlang**, **Elixir**, **Haskell**, **Dylan**, **Rust**, **Julia**, **Ruby** (in parte), **Python** (in parte).
- In forte ascesa. Programmazione parallela più sicura. **Reactive programming**.

4.1.2.5 Programmazione object oriented

- **~1980**
- Estensione della programmazione modulare.
- Definizione di oggetti opachi in grado di interagire tramite messaggi o funzioni (metodi).
- **Incapsulamento** (separazione tra *interfaccia* ed *implementazione*).
- **Ereditarietà** singola o multipla (creazione di gerarchie *genitore* e *figlio* - visibilità dei campi: pubblica o privata, protetta, pubblicata (Object Pascal *property - Java Beans*)).
- **Polimorfismo**, la possibilità di utilizzare una classe dalla sua interfaccia, intercambiabilità di classi con identica interfaccia.
- **Autoreferenza** con le parole chiave *this* o *self*.
- **Classi**, **classi astratte**, **interfacce, istanze**.

Massimo Ghisalberti - tel.: +39 3939018290 – email: massimo.ghisalberti@gmail.com

- **Tipi parametrici**.
- **Simula** (1967), **Smalltalk**, **C++**, **Objective C**, **Object Pascal**, **Java** (linguaggi per la **JVM**), **C#** (linguaggi **.NET**), **Ruby**, **Python**, **Eiffel**, **Ada95/2005/2012**, **Oberon**…
- Linguaggi tipizzati, inferenza di tipo parziale, dinamici.
- È il paradigma sicuramente oggi più utilizzato.

4.1.3 Paradigmi meno comuni

- Programmazione concorrente (**Ada**, **Occam**, **Erlang**, **X10**, oggi presente in molti linguaggi con librerie di supporto o costrutti nativi).
- Programmazione logica (**Prolog**).
- Programmazione event driven.
- **Design by Contract** (**Eiffel**, **D**)
- Programmazione a vincoli (constraint) (**Prolog**).
- Programmazione ad aspetti, AOP (**AspectJ**, **AspectC++**).

4.1.4 Multiparadigma

- Utilizzano due o più dei paradigmi citati.
- La maggior parte dei linguaggi odierni ed in sviluppo.
- **Scala**, **OCaml**, **Elixir**, **Clojure**, **Groovy**, **Ceylon**, **Kotlin**, **Xtend**, **Lua**, **Ruby**, **Python**, **C#**, **Java**, **C++11**, **Rust**, **X10**, **D**…

4.1.5 Il Futuro

- Sviluppo di linguaggi di programmazione **multiparadigma**.
- Il paradigma funzionale sarà onnipresente in qualche maniera.
- Inserimento ed evoluzione del **Reactive programming** e del **paradigma concorrente ed asincrono**.
- **Computazione distribuita**.
- **Polyglot Programming**.
- Evoluzione ulteriore di **DSL** (Domain Specific Language), linguaggi generalmente non *completi* utili per risolvere specifiche problematiche.
- Evoluzione di ulteriori paradigmi (per es.: *programmazione funzionale quantistica*, Haskell-QML `http://sneezy.cs.nott.ac.uk/QML/`).
- **Il limite è solo la fantasia…**

4.1.6 Lista linguaggi di programmazione

- `http://www.levenez.com/lang/`
- `http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm`

Massimo Ghisalberti - tel.: +39 3939018290 – email: massimo.ghisalberti@gmail.com