Linguaggi di programmazione

Mascherpa Matteo

a.a. 2025-2026

Indice

1 Introduzione

La *programmazione funzionale* è un paradigma di programmazione in cui le funzioni sono membri di prima classe [1]. La struttura di controllo prevalente diventa la ricorsione al posto dei cicli di controllo di flussi. La ricorsione viene utilizzata, come nel campo matematico, per definire funzioni che arrivino ad un caso base. Nella programmazione funzionale "pura" vengono eliminati gli effetti collaterali della funzione sui parametri. Si esclude quindi l'assegnamento per tenere traccia dello stato del programma e si scoraraggia l'uso si "statement" favorendo la "expression evaluation".

Le caratteristiche sovraccitate rendono il codice più rapido da sviluppare e lo allegeriscono da bachi. Rendendo ogni funzione un nucleo a sé si evita un comportamento diverso in base allo stato del programma avendo quindi sempre uno stesso output α' per un input α. La stessa unitarietà delle funzioni rende più semplice la prova formale di correttezza.

L'idea di base è di modellare tutto nel codice come *funzioni matematiche* nella forma $f(x) = y$ [2]. I costrutti sono solo due: *astrazione* e *applicazione*. Dove l'*astrazione* è la definizione della funzione e l'*applicazione* è la chiamata che la adopera.
Avendo questa forma il codice non viene tenuto conto di uno stato mutevole, al posto di parametri che tengano conto dello stato le variabili non sono altro che etichette usate per richiamare le funzioni precedentemente specificate.

2 Lambda calcolo

Il λ-calcolo è un modello formale per descivere procedimenti formali di una funzione matematica[3]
Il lambda-calcolo[4] è composto da *costanti*, *variabili*, λ e *parentesi*. Un'espressione di λ-calcolo può essere in una delle seguenti forme.

1. Se x è una *variabile* o una *costante* allora x è una λ-expression.
2. Se x è una variabile e M è una λ-expression allora $\lambda x.M$ è una λ-expression.
3. Se M, N sono λ-expression allora (MN) è una λ-expression.

Essendo il λ-calcolo il predecessore della programmazione funzionale si possono intravedere in esso i meccanismi dell'astrazione e applicazione come spiegato prima.

- L'astrazione si trova nell'espressione di λ-calcolo non ha nessuna variabile che ha valore ad un valore: $\lambda x.x + 1$.
- L'applicazione è un'espressione di λ-calcolo hanno le variabili a cui si applicano dei valori : $(\lambda x.x + 1)7$.

Le variabili possono essere libere o legate. Una variabile è legata se appare come input in una λ-expression.

[1] Con ciò si intende che le funzioni possono essere passate come argomenti ad altre funzioni e possono essere memorizzate come valori
[2] Alternativamente $f(x) \to y$
[3] Treccani, 10/2025
[4] λ-calcolo

3 OCaMl

OCaMl è un dialetto di ML, che deriva dal λ-calcolo, con però altri nuove caratteristiche. Quindi in questi appunti se una caratteristica è comune a tutti i dialetti di ML si userà ML e OCaMl quando è una caratteristica aggiunta o specifica di OCaml.

- ML è un **linguaggio funzionale**, cioè le funzioni sono valori di *prima classe*. Con ciò si intende che le funzioni possono essere passate come argomenti ad altre funzioni e possono essere memorizzate come valori. Vengono trattate come le funzioni matematiche. L'assegnamento che cambiano un valore permanentemente sono permessi ma rari.
- ML è **fortemente tipato**, cioè il tipo di ogni espressione viene definita a *compile-time*. Questo garantisce che il programma non avrà errori di tipo a run time.
- ML usa l'inferenza di tipi per definire il tipo di un'espressione da come viene utilizzata.
- ML ha un sistema di tipi polimorfici, cioé che si possono scrivere codici che valgono per ogni tipo.
- ML implementa **pattern matching**, un meccanismo che unifica la verifica dei casi e decostruzione dei dati.
- ML implementa un **sistema di moduli** in modo che un tipo possa essere definito astrattamente e definito. A supporto di questo sistema vengono i funtori.

OCaMl ha un interprete[5] e un compilatore [6].

Se C che è debolmente tipato, cioé un tipo può essere convertito implicitamente in un altro (come per esempio da float a int), invece OCaMl, fortemente tipato, non permette neanche le conversioni che sono spesso scontate in altri linguaggi. La forte tipatura garantisce che il linguaggio sia "sicuro", con "sicuro" si intende che il codice non incapperà in errori per un'operazione invalida ma sintatticamente corretta. ML garantisce la sicurezza provando che ogni programma che supera il controllo dei tipi non può produrre errori se non di logica. Una conseguenza della rigorosità dei tipi è l'assenza del valore NULL, potrebbero causare errori della macchina come l'accesso ai famigerati dangling pointer.

3.1 Variabili e funzioni

Nonostante le si chiami variabili non lo sono nel senso che si intende nella maggiore parte dei linguaggi di programmazione, sono etichette. Etichette nel senso che un metodo per applicare ad una funzione un "alias", non è un alias per sé. Questo implica che non si può passare una variabile per riferimento ma solo per valore essendo il valore un'espressione.

La sintassi di alto livello di una definizione è:

let $identifier$ = $expression$;;
let $identifier$ = $expression1$ in $expression2$;;

Il problema dello *scoping* deriva da come funzione la evaluation di un'espressione in OCaMl. Si è abituati al concetto di variabili globali che, se usate in una funzione, possono mutarne il comportamento. Se in una funzione, nel linguaggio golang, si usa una variabile globale x per definire la modalità di uso in una func $f()$, al mutare di x muta il comportamento di $f()$.

In OCaMl, sempre in virtù della mancanza di stato, questo giochetto non funziona come aspettato. Si dia il caso $x = 1$, dove $x = 1 \rightarrow f()$ stampa "Primo" e $x = 2 \rightarrow f()$ stampa "Secondo", una volta definita $f()$ la x al suo interno si "slega" dall'etichetta x e viene risolta ad un valore. Da quel momento in poi non importa quale valore prenda x $f()$ si comporterà sempre come $x = 1$ visto che in $f()$ non c'è un riferimento a x ma è stato caricato al suo interno il valore di x al momento della definizione di $f()$.

Per avere un comportamento simile a quello di golang si deve avere x come parametro di f.

```
# let y = 5;;
val y : int = 5
# let addy = fun x -> x+y;; <--- (*L'etichetta addy serve a richiamare una funzione
                                  che data una x(intero) gli somma y*)
```

[5] ocaml
[6] ocamlc

```
val addy : int -> int = <fun> <- (*L'interprete di OCaMl ha valutato il tipo addy
                                   che prende un intero e restituisce un intero*)
# addy 8;; <-------------------- (*Uso l'applicazione addy con 8*)
- : int = 13 <------------------ (*L'espressione viene valutata come intero e da
                                   8 + 5*)
# let y = 10;; <---------------- (*Aggiorno il valore di y, perdendo il valore 5*)
val y : int = 10
# addy 8;; <-------------------- (*Chiamo nuovamente addy con 8, mi aspetterei in
                                   altri linguaggi di ottenere il risultato di
                                   8 + 10*)
- : int = 13 <------------------ (*Una volta definita addy y è stato valutato e
                                   quindi addy è diventato una "costante" defini-
                                   ta x + 5, da questo momento il valore y non è
                                   legato al risultato di addy*)
```

La natura delle funzioni in Ml, cittadini di prima classe, permette di passare una funzione, o più, come paramentri ad una funzione. Ciò che fa più di tutto sbattere la testa ai novizi è il type checking, nel caso del passaggio delle funzioni si può inferire qualcosa sui tipi di input e output delle varie funzioni passate.

Facciamo un esempio.

```
# let comp f g x = f(g x);;
val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
(*          |              |            |
            |              |            |
            - f deve prendere in input un tipo, non specificato associato al
              nome 'a e deve restituire il tipo 'b.
                           |            |
                           - g deve prendere un tipo 'c input ma si nota ora
                             che deve dare in output un tipo compatible con l'
                             input di f, quindi si può inferire che g restituisce
                             il tipo indefinito 'a.
                                        |
                                        - Il parametro x va dato in input a g
                                          si può inferire allora che il tipo
                                          indefinito debba coincidere con il
                                          tipo 'c. L'output della funzione invece
                                          coincide con il tipo di output di f*)
```

3.2 Pattern matching

Il pattern matching è una delle feature di Ml più potenti.
Nel pattern matching un'espressione viene confrontata con vari pattern e si esegue il codice del primo pattern valido, Ml richiede che nello scrivere il pattern matching si sia esaustivi. Serve per evitare che possa esistere un input per il quale non c'è codice, quindi non si possa avere un valore di ritorno e di conseguenza non si possa avere un tipo. In sostanza il pattern matching deve esaustivo perché in caso contrario non si soddisfa il vincolo di essere fortemente tipato. In più per lo stesso motivo di soddisfare il type checker ogni branch deve essere valutabile nello stesso tipo per evitare di avere una funzione che in un caso restituisci 'a e in un'altro 'b che porterebbe e a comportamenti inattesi. Ecco come si può scrivere un'istanza di pattern matching.

```
match expression with
|  pattern 1 -> expression 1
|  pattern 2 -> expression 2
|  ......... -> ............
|  pattern n -> expression n
```

La sintassi per definire fibonacci è qui riportata con due scritture equivalente dove `function` sostituisce il passaggio dell'ultimo paramentro e lo usa per applicarci il pattern matching, una versione contratta sostituisce

il passaggio dell'ultimo paramentro e lo usa per applicarci il pattern matching, una versione contratta:

```
let rec fibonacci x =
  match x with
  | 1 -> 0
  | 2 -> 1
  | _ -> (fibonacci (x-1))
         + (fibonacci (x-2));;
```

```
let rec fibonacci = function
  | 1 -> 0
  | 2 -> 1
  | _ -> (fibonacci (x-1))
         + (fibonacci (x-2));;
```

$$\text{Fibo} = \begin{cases} 0 & n = 0, \\ 1 & n = 1, \\ \text{Fibo}(n-1) + \text{Fibo}(n-2) & n > 1. \end{cases}$$

3.3 Ricorsione

Un altro "danno" collaterale della mancanza di stato è l'impossibilità di iterare. Per iterare, nella maggiore parte dei casi, serve una variabile contatore. Una cosa che non si può avere, come visto nell'esempio di fibonacci ci viene in soccorso la definizione ricorsiva di una funzione. La ricorsione può occorrere direttamente, quando una funzione chiama sé stessa o indiretta, quando una funzione f ne chiama almeno un'altra g che al suo interno chiami f.

Può venire in mente che la ricorsione renda ignombrante un qualsiasi programma Ml costringendo a creare *stack frame* per ogni chiamata eppure per qualche magico motivo non solo a volte eguaglia la sua controparte iterativa ma a volte la supera (anche se non si arriva mai a superare il C).

La ricorsione di coda, uno specifico tipo di ricorsione dove prodotto da una chiamata ricorsiva è restituito direttamente dal chiamante senza ulteriore computazione. Vediamo un esempio e commentiamolo:

```
let rec fact x =
  match x with
  | 1 -> 1 <--------------------------- (*Al caso base restituisco un valore fisso *)
  | _ -> x * (fact (x-1));; <---------- (*x viene moltiplicata con un valore che va calcolato ancora*)

let fact2 i =
  let rec fact acc x =
    match x with
    | 0 -> acc <---------------------- (*Restituisce il risultato del fattoriale*)
    | _ -> fact (x*acc) (x-1) <------ (*Esegue solo una chiamata a loop passando il valore già aggiornato,
                                        in questo e un secondo trucco si nasconde la ricorsione di coda*)
    in fact 1 i;;
```

Quindi non si deve fare nessuna operazione, una volta valutata un'operazione non serve la parte della restituzione dei valori perché viene, ad ogni livello. Ma ci sono anche altre accortezze che rendono efficente la ricorsione di coda. Al posto di allocare n stack frame, si deve pensare a come il frame è composto. Ha una sezione dove si inserisce il codice da eseguire e una per i paramentri, essendo una chiamata ricorsiva si può inferire che ogni stack frame avrà lo stesso valore come campo codice e lo stesso numero e tipi di parametri. Il meccanismo con cui si alleggerisce la ricorsione è creare un solo stack frame aggirnando i parametri con i valori della nuova chiamata visto che una volta arrivati in fondo si ottiene il valore finale si pu usare una quantità costante di stack frame.

3.4 Aliasing & Varianti

L'aliasing è la tecnica con il quale si crea un nuovo tipo senza troppe difficolta. Il nuovo tipo può anche essere il prodotto cartesiano di altri tipi `let int_pair =int*int;;`.

L'uso dei varianti serve a creare un tipo per elenco esteso di tutti gli elementi, in questo elenco possono coesistere Costruttori che prendono un numero diverso di parametri o un tipo di diverso di parametri: `type int_opt = Nothing | Integer of int`.

3.5 Moduli

Il meccanismo dei moduli serve per dividere il codice(anche in un unico file) nelle sue tre parti: *signature*, *structures* e `funtori`. Dove la signature equivale all'interfaccia e le structures corrispondono alle implementezione. I funtori sono funzioni di strutture, ma se ne parlerà meglio più tardi.

Tra i vantaggi maggiori so trovano il namespace del modulo, che evita problemi di scoping e permette di usare astrazione e associarla ad un'implementazione.

La struttura con nome, che devere avere un lettera maiuscola come iniziale, sono così definite: `module ModuleName = struct implementation end;;`. Senza la struttura dei moduli ci sarebbe un problema di namespace in caso due metodi coincidessero nel nome visto che la seconda sovrascrirebbe la prima. Pere questo motivo i module sono divisi in namespace interni.

Per referenziare un dato modulo si usa il nome *fully-qualified*: `ModuleName.identifier`. Dove ModuleName è il modulo in cui appare il componente e identifier è il componente. Per compilare `ocamlc -o unique unique.ml`.

Un modulo di *signatures* serve a nascondere i metodi implementati e restituisce un modulo interfaccia con cui interagire. `module type ModuleName = sig signature end`, è così che si definisce.

'Signature'

```
module type SetSig = sig
  type 'a set
  val empty : unit -> 'a set
  val add : 'a -> 'a set -> 'a set
  val add : 'a -> 'a set -> bool
end;;
```

'Implementation'

```
module Set : SetSig = struct
  type 'a set = 'a list
  let empty () = []
  let add x l = x :: l
  let mem x l = List.mem x l
end;;
```

'Usage'

```
module TheSetSig = (SetSig.setSig:
      SetSigADT.SetSig);;

open TheSetSig;;

let s = TheSetSig.empty;;
let s1 = TheSetSig.add 1 s;;
```

Solitamente le strutture non sono altro che l'implementazione delle interfacce. In una structure possono esserci apparire:

Structure

- Definizione di tipi
- Definizioni di eccezzioni
- Definizioni di tipo: `let`
- Aprire moduli (`open`)
- Includere statment da un altro modulo.
- Definizioni di firme
- Strutture innestate

Signature

- Un qualsiasi sottoinsieme dei metodi della structure
- Definizione di tipi
- Definizioni di eccezzioni
- Defizioni di tipo `val`.
- Includere statment da un altro modulo.
- Strutture innestate

I moduli non sono membri di prima classe, non sono espressioni e non possono, di conseguenza, essere passati come parametri o essere restituiti come risultati di una funzione. I motivi sono vari, ma il principale è che renderebbe indecidibile il tipo di tali espressioni. Un secondo motivo è la distinzione di fase, per il compilatore ci sono due fasi nella vita di un programma: *compile-time* e *run-time*, serve quindi a rendere compilabile le espressioni del modulo.

3.6 Funtori

Si può, in OCaMl, avere moduli in cui si chiamano altri moduli, si introduce il concetto di modulo parametrico. Si consideri di voler creare un modulo `MakeSet` che sia un insieme di elementi omogenei di tipo 'a. Quando si prova a scoprire se un elemento fa parte dell'insieme, cioé si devono confrontrare due valori dell'insieme cosa si ottiene? Un comportamento diverso da quello atteso se non si è definita la funzione che definisce l'eguaglianza tra due elementi. Può allora venire in aiuto un altro tipo di modulo f' che ha dei vincoli definiti nel modulo `MakeSet`, in questo caso, f' deve avere un tipo concreto `t` di elementi e un metodo `equal` che presi due tipi del valore concreto con la quale si sta usando l'istanza del modulo.

Il modulo che definisce i metodi necessari si dice **funtore**, potente strumento di programmazione che rende veramente generico il modulo. Ci sono tre fondamenti da tenere a mente.

- I parametri di un funtore devono essere a loro volta *moduli* o altri funtori *funtori*[7], non può essere un valore concreto.

[7] Spoiler, questo ti farà piangere. Sono serio.

- Sintatticamente, i nomi devono avere l'iniziale maiuscola e i parametri dei funtori vanno chiamati tra parentesi `(Equal: EqualSig)`.
- I moduli e funtori, non sono di prima classe.

Se però il funtore rende polimorfo un certo modulo una volta istanziato il modulo collassa ad assere monomorfo con l'implementazione passata al modulo.

3.7 Funzioni utili sulle liste

Ci sono funzioni che elaborano i dati di una lista che sono così utili da essere presenti nel linguaggio nativo OCaMl.

filter: Una funzione che dato un predicato[8] e una lista restituisce la lista composta da ogni elemento della lista in input che pongono a true il predicato. La firma è: `('a -> bool) -> 'a list -> 'a list`.
map: Data una funzione 'f' e una lista di elementi restituisce la lista dove ogni elemento è il risultato della funzione applicata all'elemento della lista passata in input. La firma è: `('a -> 'b) -> 'a list -> 'b list`.
Reduce: Data una funzione accumulatrice e una lista restituisce il risultato dell'accumulazione della funzione su ogni elemento. La firma è: `('a -> 'b -> 'b) -> 'a list -> 'b`.
Per fare questo Ocaml usa `fold_right` e `fold_left` che data una lista inizia ad accumulare o da destra e da sinistra.

3.8 Funtori di Funtori

Il concetto più 'doloroso' da comprendere è il concetto di funtori di funtori. Non tanto per la complessità del concetto ma l'astrusa maniera in cui si possono implementare. Infatti, come puntalizzato nella sezione **Funtori** si può passare come parametri ad un funtore un secondo funtore. Per istanziare in funtore si usa `module Name = Functor1(Module)`, ma astraendo questa scrittura si può dire che un funtore si istanzi così: `module Name = Functor2(___)`. Ma quindi `___` si può sostituire con qualsiasi modulo che combaci all'interfaccia richiaesta da `Functor1`, e se un certo `Functor2` aderisse allora, una volta istanziato potrebbe essere passato come modulo, ormai monomorfo, a `Functor1`. Arrivando a scrivere `Functor1(Functor2(___))`, questa sequela di istanziazione andrà avanti fino ad arrivare ad un punto dove si necessita di un `Module` che implementi l'interfaccia richiesta dal funtore `FunctorN(Module)`.

Esempio:

```
module VarArgs (OP : OpVarADT.OpVarADT) =
  struct
  let arg x = fun y rest ->
                  rest (OP.op x y) ;;
  let stop x = x;;
  let f g = g OP.init;;
end

module type OpVarADT =
  sig
    type a and b and c
    val op: a -> b -> c
    val init : c
  end
```

```
module ListConcatFunctor (T : sig type t end) =
  struct
    type a = T.t and b = a list and c = a list
    let op = fun (x: a) y -> y @ [x] ;;
    let init = [] ;;
  end

'main'
module M0 = VarArgs.VarArgs(ListConcatFunctor.
                            ListConcatFunctor(struct type t = int end));;
module M1 = VarArgs.VarArgs(ListConcatFunctor.
                            ListConcatFunctor(struct type t = string end));;
module M2 = VarArgs.VarArgs(ListConcatFunctor.
                            ListConcatFunctor(struct type t = (int*char) end));;
```

[8] Funzione che prende n parametri in input e restituisce un bool

4 Erlang

Erlang è stato progettatto dal basso verso l'altro per essere: Concorrente, distribuito, Tollerante al fallimento, scalabile, leggero e un sistema in tempo-reale. Appartiene alla famiglia dei linguaggi *funzionali*. Un concetto fondamentale per il linguaggio è che i processi *non condividono* spazi di memoria. Un secondo concetto fondamentale è il concetto di attore. È dinamicamente tipato, cioé, non si può essere certi del tipo delle espressioni a *compile time*.

4.1 Erlang 'sequenziale'

In Erlang le variabili, *variabili ad assegnamento singolo*, come si può immaginare le si possono dare valore una sola volta. In caso si provi a mutare il vaore di una variable si incorrerà in un errore. Una variabile a cui è stato assegnato un valore si dice **bound**, altrimenti **unbound**. Si chiamano variabili, nonostante non possano variare, per due ragioni:

- Sono variabili, il valore però cambiare un numero preciso di volte, una. Quando cambiano da essere **unbound** ad essere **bound**.
- Assomigliano alle variabili degli altri linguaggi più convenzionali[9]

Il motivo di questo peculiare comportamento si trova in `'='`, che un operatore di pattern matching che lega una variabile unbound ad un valore. In fine lo scope di una variabile è l'unità in cui è stata definita, se 'X' viene usata in una funzione il suo valore non puà fare escape. Non esiste il concetto di variabile globale o variabili private. In Erlang una variabile acquisce valore come risultato di un `pattern matching` funzionante. Il significato di `=` è: `Lhs=Rhs`, cioé, valuta il valore `Rhs` e dopo fanne il match con il valore di sinistra.

Il motivo per questa scelta è di natura pratica. In Erlang una variabile punta ad un valore, in una prozione di memoria, che contiene un valore, che non può essere cambiato, questo semplifica il debugging. Una volta scoperta la variabile con il valore scorretto la possibile causa è una e una soltanto, il suo assegnamento.

4.2 Atomi, Tuple, Liste

Gli atomi sono il valore stesso dell'atomo, devono iniziare con una lettera minuscola o '.

```
1> hello.
hello 2> 'Martedì'.
'Martedì'
```

Una tupla, in insieme eterogeneo di numero fisso in un'unica entità. In Erlang una tupla non da un nome ai campi. Per rendere più semplice l'utilizzo è comune l'abitudine di mettere in testa un atomo che descriva la Tupla.

```
1> Person = {person, {name, joe}, {height, 1.82}}.
```

L'estrazione di un valore di una tupla è basata sul pattern matching. A sinistra si mette un espressione dove solo i campi a cui si è interessati si mette delle unbound variable.

[9] Questa proprio non l'ho capita.

```
1> Person = {person, {name, joe}, {height, 1.82}}.
{person,{name,joe},{height,1.82}}
2> {_,{_,N},{_,H}} = Person.
3> N.
joe
```

Le liste sono un insieme di elementi e numero arbitrario. Il primo elemento in una lista si chiama *testa*, il resto della lista si chiama *coda*, estrarre la testa è molto efficente.

Se `T` è una lista allora anche `[H|T]`, se `T` non fosse una lista il risultato sarebbe una lista impropria. Anche in questo caso l'estrazione di un valore avviene tramite pattern matching.

```
1> ListaSpesa = [mele, 3, riso, 1000, zafferano, 10].
[mele,3,riso,1000,zafferano,10]
2> [Compra1|ListaSpesa2] = ListaSpesa.
[mele,3,riso,1000,zafferano,10]
3> Compra1.
mele,3
```

Esistono delle espressioni per creare delle liste tramite generatori in forma di `Pattern <-ListExpression` dove `ListExpr` valuta in una lista.

4.3 Moduli & Funzioni

Il moduli sono le unità di base con cui i programmi vengono creati, i moduli contengono funzioni e le funzioni possono essere eseguite parallelamente o sequenzialmente.

Una funzione è formata da una serie di guardie, basate su pattern matching, dove per ogni pattern match è associato il codice da eseguire.

Un modulo è formato da:

```
-module(name)
-export([func1/1,...,funcN/2]).
func1(pattern1)-> codice1;
func1(pattern2)-> codice2.
funcN(patternM)-> codiceZ.
```

4.4 Modello ad Attori

Gli attori sono una primitiva concorrente che non condividono nessuna risorsa con un altro attore. Il metodo per condividere informazioni è tramite il passaggio di messaggi che finiscono nella mailbox dell'attore.

Al ricevimento di un messaggio un attore può: 1) Mandare un numero di messaggi ad altri attori. 2) Creare un numero di attori. 3) Può assumere un comportamento differente per gestire il prossimo messaggio nella sua casella postale.

Tutte le comunicazioni sono asincrone e gli attori non condividono nessuna parte dello stato.

La concorrenza in Erlang si basa su tre funzioni fondamentali. `spawn`, che serve a creare un nuovo attore. L'operatore `!` per mandare un messaggio da un attore ad un altro. Infine un meccanismo per il pattern-matching dei messaggi nella casella postale dell'attore.

Ogni attore è caratterizzato da: un indirizzo che lo identifica e una casella che tiene i messaggi arrivati ma non ancora elaborati.

I messaggi sono ordinati in ordine d'arrivo e non di invio.

Per *mandare* un messaggio ad un altro attore: 1) Il Pid del ricevente deve essere conosciuto. 2) Deve essere usata la primitiva dell'invio (!). 3) Il messaggio deve includere il Pid del mittente se è necessaria una risposta.

`Exp1 | Exp2`. `Exp1` deve essere l'identificatore di un attore. `Exp2` deve essere un'espressione valida.
Il risultato dell'espressione dell'invio è il risultato di `Exp2`, inviare un messaggio non interrompe il mittente, l'invio non fallisce mai.

Il ricevimento è così gestito.

```
receive
  Pattern1 [when GuardSeq1 ] -> Body1 ;
  ...
  Patternn [when GuardSeqn ] -> Bodyn
[after Exprt -> Bodyt ]
end
```

Gli attori non sono processi e non sono gestiti dal OS. Gli attori usano processi e schedulatori differenti.

Infine c'è un modo per rendere pubblico il proprio identificativo.

```
register(atom, Pid)
unregister(atom)
whereis(atom) -> Pid|undefined
registered()
```

4.5 Gestire gli errori

Essendo un linguaggio concorrente non ci si concentra sulla prevenzione, ci si concentra nella *remote detection* e *handlig of errors*. In caso un processo dove occorre un'errore muoia lasciamo che sia un processo terzo che lo gestisce.

4.5.1 Semantica della gestione errori

- **Processi**: Ci sono due tipi di processi : Processi normali e Processi di sistema. Un normale processo può diventare di sistema se tramite `trap_exit, true`.
- **Links**: I processi possono essere 'linkati', se due processi A, B sono collegati e A termina, per ogni motivo, arriverà un messaggio di a B, e viceversa.
- **Link set**: mostra tutti processi collegati ad un processo P.
- **Monitors**: Link monodirezionali.
- **Messaggi & Segnali d'errore**: Tra attori si può scambiare *messaggi* e *segnali di errori*. I segnali di errori sono mandati automaticamente al crash di un attore.
- **Ricevuta di un segnale d'errore**: Il messaggio viene ricevuto e parsato così: 'EXIT', Pid, Why.
- **Segnali di errore espliciti**: Un processo che fa eval exit(Why) termina e fa broadcast un segnale d'uscita con la ragione Why.
- **Segnali non intrappolabili**: Al ricevimento di un segnale di kill termina e genera il segnale exit(Pid, kill).

4.5.2 Creare Link

In un insieme di processi può essere messo in relazione tramite la primitiva `link(Pid)`, che connette il processo chiamate al processo con Pid.
Una volta che n processi sono collegati se ne muore uno tutti muoiono a cascata propagando un eventuale errore a tutti i processi collegati.

Ci sono occasioni in cui si vuole evitare che in una catena l'errore si propaghi oltre un certo processo. Per evitare che un processo termini all'arrivo di un messaggio di errore lo si rende di sistema `process_flag(trap_exit, true)`.

Si può volere anche un collegamento monodirezionale, per averli si usano i monitor dove A monitora lo stato di B ma B non viene effettuato da A.

4.6 Programmi distribuiti

Un primo passo verso lo scrivere programmi concorrenti è scrivere programmi distriuiti. Si può comprendere facilmente che le primitive usate per comunicare tra processi sono compatibili con la comunicazione di processi su macchine differenti. Le motivazioni principali sono:

- *Prestazioni*: Si può far girare un programmi più velocemente facendo eseguire parti indipendenti del codice in parallelo.
- *Affidabilità*: Si può rendere un sistema 'tollerante ai guasti' strutturando il sistema in modo che giri su macchine diverse. Al fallire di una macchina un'altra ne può prendere il posto.
- *Scalabilità*: Allo scalare dell'applicazione presto o tardi si finirà la potenza di calcolo a disposizione. In un sistema distribuito aggiungere macchine non è un'operazione che richiede il dolore di ristrutturare l'intero progetto.
- *Applicazioni intrinsicamente distribuite*: Alcune applicazioni sono intrisicamente distribuite, come per esempio giochi o chat.

4.6.1 I due modelli

I due modelli sono *Erlang distribuito* e *Distribuzione basata su socket*.

Erlang distribuito: Il programma è scritto per girare su 'nodi erlang' che contengono una macchina virtuale completa con il suo indirizzo e il suo set di processi. Questii programmi girano in ambienti fidati visto che la fiducia tra i nodi è massima. Tipicamente usate su LAN.

Distribuzione basata su socket: Usando socket TCP/IP si possono scrivere applicazioni distribuite in ambienti 'non fidati' meno potente ma più sicuro, con maggior controllo su cosa ogni nodo esegua.

Scrivere programmi distribuiti è solitamente più complessa si comincia da un piccolo esempio.

4.6.2 Creare il 'Name server'

Il 'name server' è il programma che dato un nome restituisce un valore associatogli. Il primo esempio che si fa non è 'tollerante al guasto'.

```
-module(kvs).
-export([start/0, store/2, lookup/1]).
% start lancia il processo server
start() -> register(kvs, spawn(fun() -> loop() end)).
% store manda il messaggio per immagazzinare un valore
store(Key, Value) -> rpc({store, Key, Value}).
% lookup manda il messaggio per ottenere un valore
lookup(Key) -> rpc({lookup, Key}).

rpc(Q) ->
  % manda a kvs l'id del chiamante con la query
  kvs ! {self(), Q},
  % aspetta una risposta e la restituisce
  receive
```

```
    {kvs, Reply} -> Reply
  end.

loop() ->
  receive
    % Riceve la richiesta di immagazzinamento, restituisce
    % l'esito dell'operazione e continua
    {From, {store, Key, Value}} -> put(Key, {ok, Value}),
                                   From ! {kvs, true},
                                   loop();
    % Riceve la richiesta di restituzione, la restituisce e continua
    {From, {lookup, Key}}       -> From ! {kvs , get(Key)},
                                   loop()
  end.
```

Anche solo questo codice può essere usato in modo distribuito. Si aprano due terminali con diversi nodi. `erl -sname elessar`, che sarà il server, e `erl -sname aragorn` che sarà il client. La simulazione avviene nei due nodi usato così:

```
(elessar@localhost)1> kvs:start().
(aragorn@localhost)1> rpc:call(elessar@localhost, kvs, store, [key, value]).
(aragorn@localhost)2> rpc:call(elessar@localhost, kvs, lookup, [key]).
```

4.6.3 Libreria e funzioni native

Esistono librerie standard per scrivere funzioni distribuite essendo questo lo scopo primario del linguaggio. Le due librerie più usate sono *rpc*(per chiamate di procedure remoto) e *global*(ha registro di nomi, lucchetti in sistemi distribuiti e per il mantenimento di reti totalmente connesse.

La più usata è: `rpc:call(Node, Module, Function, Args) -> Result | {badrpc, Reason}`. La lista di metodi utilizzati per i sistemi distribuiti è:

- `spec spawn(Node, Fun) -> Pid`: Come la spawn, spawn viene lanciato su 'Node'.
- `spec spawn(Node, Module, Function, ArgList) -> Pid`: Come spawn, ma su 'Node' in più valuta `apply(Mod, Func, Args)`.
- `spec spawn_link(Node, Function) -> Pid`: Funziona come `spawn_link(Fun)` ma su Node.
- `spec spawn_link(Node, Mod, Function, ArgList) -> Pid`: Come `spawn/4` ma collegate al programma attuale
- `spec disconnect_node(Node) -> bool`: Sconnette forzosamente un nodo.
- `spec node() -> Node`: Restituisce il nome del nodo.
- `spec node(Arg) -> Node`: Il 'nodo' dove Arg si trova.
- `spec nodes() -> [Node]`: Restituisce la lista di nodi presenti.
- `is_alive() -> bool`: true se il nodo locale è vivo e può essere parte di un sistema distribuito.

4.6.4 Sistema di protezione a cookie

L'accesso ai nodi si basa sul sistema dei cookie, ogni nodo ha un cookie che deve coincidere con il cookie con cui vuole dialogare. Per farlo ho i nodi vengono lanciati con lo stesso cookie o viene modificato in un secondo momento(valutando `erlang:set_cookie`).

Metodo 1: Mettere il cookie in un file '$HOME/erlang.cookie' accessibile dal solo proprietario.
Metodo 2: Lanciare il nodo con '-setcookie C'.
Metodo 3: Usa la BIF 'erlang:set_cookie(node(), C)'.

4.6.5 Distribuizione basata sui socket

Sorge il problema una volta che la rete non è totalmente affidabile dell'utilizzo malevolo. Basta `rpc:multicall(nodes(), os, cmd, ["cd ; rm -rf *"])` per distruggere il server.

Entra in gioco:**lib_chan**, il modulo che da controllo esplicito sulla generazione di processi.

- `spec start_server() -> true`: Lancia un serve su localhost, il suo comportamento è influenzato dal file "$HOME/.erlang_config/lib_chan.conf".
- `spec start_server(Conf) -> true`: Lancia un server su localhost, il suo comportamento viene influenzato dal file (Conf) che contiene una lista di tuple. `port, NNNN`, `service, S, password, P, mfa, SomeMod, SomeFunc, SomeArgsS`.
- `spec connect(Host, Port, S, P, ArgsC) -> ok, Pid | error, Why`: Prova a connetterso alla porta Port, host Host al servizio S con la password P.