Linguaggi di Programmazione

Ivano Salvo

FP4: programmazione su liste (2)
OOP3: Pensare soluzioni a Oggetti

Corso di Laurea in Matematica

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **5**, 27 ottobre 2020

Lezione 5a:

Funzionali su Liste

Schemi di programmi (1)

Molti programmi hanno una struttura comune: ad esempio iterano una stessa funzione su una lista.

Un funzionale può generalizzare questa forma di ricorsione: è chiamato spesso **reduce**, ma in Haskell si chiama `foldl` (vedremo altre versioni simili, come `foldr`).

```
 -- La funzione length
myLength (_:xs) = 1 + myLength xs
myLength   []   = 0
 -- La funzione sum è simile a length
mySum (x:xs) = x + mySum xs
mySum   []   = 0
 -- La funzione produttoria
myProd (x:xs) = x * mySum xs
myProd   []   = 1

 -- Possiamo generalizzare...aka reduce
myFold f g (x:xs) = f x (myFold f g xs)
myFold f g []     = g
myFold :: (t -> t1 -> t1) -> t1 -> [t] -> t1
```

Schemi di programmi (2)

C'è un gusto tutto funzionalista di scrivere funzioni come composizione di altre funzioni (**one-liner**) senza fare ricorsione esplicita decomponente liste o altre strutture dati.

Può a volte aiutare anche il compositore, che esiste già predefinito e si chiama (.)

```
-- e quindi...
> myLength' = myFold (\x y->y+1) 0
> mySum' = myFold (+) 0
> myProd' = myFold (*) 1
 -- ma anche...
> myLength'' = myFold (\x -> (+1)) 0

 -- il funzionale composizione di funzioni
c f g x = f (g x)
c :: (t1 -> t) -> (t2 -> t1) -> t2 -> t
 -- oppure infisso predefinito
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Schemi di programmi (3)

Un famoso funzionale è l'**apply-to-all**, noto soprattutto come `map`, che applica una funzione a tutti gli elementi di una lista.

C'è una versione binaria: `zipWith`.

Volendo c'è pure quella *n*-aria. Vedremo come generalizzare.

```
 -- map applica f a tutti gli elementi di una lista:
myMap f (x:xs) = f x : myMap f xs
myMap f [] = []
myMap :: (t -> t1) -> [t] -> [t1]
 -- ad esempio:
> myMap (+1) [41,36,72]
[42,37,73]
> myMap (\x->[x]) [42,37,73]
[[42],[37],[73]] -- myMap (\x->[x]) . (+1) [41,36,72]

 -- spesso è utile una versione 'binaria' di map
myZipWith f (x:xs)(y:ys)=f x y: myZipWith f xs ys
myZipWith f [] _ = []
myZipWith f _ [] = []
myZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Schemi di programmi (4)

Tra i casi particolari, molto famoso è `zip`, o **cerniera** che accoppia ordinatamente gli elementi di una lista.

Può essere ottenuto facilmente da `zipWith`.

Possiamo anche immaginare di applicare una lista di funzioni a una lista di argomenti…

```
 -- speso è utile una versione ‘binaria’ di map
myZip (x:xs)(y:ys)=(x,y) : myZip xs ys
myZip [] _ = []
myZip _ [] = []
myZip :: [a] -> [b] -> [(a,b)]

 -- ma ovviamente
myZip’ = myZipWith (\x y ->(x,y))

applyList (f:fs)(x:xs)=f x: applyList fs xs
applyList [] _ = []
applyList _ [] = []
applyList :: [t -> t1] -> [t] -> [t1]
```

Una famosa applicazione…

John Backus (progettista del Fortran) nel **1978** scrisse un celeberrimo articolo che rilanciò la programmazione funzionale ponendo l'accento sulle sue **virtù composizionali**.

Fece l'esempio del **prodotto scalare**.

```
 -- prodotto scalare con myFold e myZipWith:
prodottoScalare xs ys =
   myFold (+) 0 (myZipWith (*) xs ys)

 -- ma anche con map e applyList
prodottoScalare' xs ys =
   myFold (+) 0 (applyList (myMap (*) xs) ys)

 -- ma a ben vedere
 -- giocando con la composizione
prodottoScalare'' xs =
      mySum . ((applyList . (myMap (*))) xs)
```

…andiamo oltre: prodotto tra matrici

Immaginiamo una **matrice** come una **lista di liste**, memorizzata per righe.

Dovendo fare il prodotto riga x colonna, e volendo usare la funzione `prodottoScalare`, cominciamo a scrivere una funzione che genera la trasposta.

Dopodichè dovremmo "mappare" per ogni riga r della prima, la funzione `(prodottoScalare r)` nella seconda. Come fare?

```
 -- trasposta di una matrice
 -- caso degenere: la matrice è vuota
trasposta []    = []
 -- vero caso base: la matrice ha una sola riga
trasposta [xs]  = map (\x->[x]) xs
 -- se ho trasposto n-1 righe, mi basterà mettere
 -- tutti gli elementi della prima riga in testa
 -- alle colonne (=righe)…
trasposta [xs:xss]= zipWith (:) xs (traposta xss)
```

miracoli dell'ordine superiore

Ovviamente avendo una riga `r` e una lista di colonne `b`, è facile scrivere il prodotto di `r` per `b`.

Dopo aver astratto su `r` otteniamo una funziona che può essere mappata dentro la trasposta…

A ben vedere (essendo prodottoScalare scritto con una map) le tre map annidate corrispondono a 3 cicli for annidati, ma fa tutt'un altro effetto 😉

```
 -- riga i-esima del prodotto (r è i-esima riga)
 -- b la matrice da moltiplicare
map (prodottoScalare r) b
 -- per fare il prodotto occore mappare questa
 -- funzione dentro la matrice a
 -- avendo cura di astrarre su r
prodMat a b =
  map (\x-> map (prodottoScalare x)
                (trasposta b)) a
 -- miracoli dell'ordine superiore
 -- notare l'utilità di astrarre…
```

Con un poco di zucchero (sintattico)

Due caratteristiche molto amate dagli Haskellioti che permettono di scrivere programmi simili a usuali notazioni matematiche.

1. la clausola **where**: è possibile usare una `**variabile libera**' e poi specificarne il significato.
2. **list comprehension:** liste definite si possono definire come gli insiemi in matematica $\{ x \in A \mid P(x) \}$.

```
 -- curiosamente questo termine tipa... perché?
j = x x where x = \y -> y
 -- vediamo il powerset: distinguiamo gli "insiemi"
 -- che contengono x e quelli che non la contengono
 -- qui where ci evita di ricalcolare lo stesso set
powerset (x:xs) = pws ++ map (x:) pws where
    pws = powerset xs
powerset [] = [[]]
-- map ridefinita per list comprehension
myMapLC f xs = [f x | x<-xs]
-- posso usare predicati, detti guards:
factors n = [x | x<-[1..n], n 'mod' x == 0]
```

Ancora su list comprehension

Si può fare list comprehension, prendendo da due liste… attenzione che si "moltiplicano" i casi…

Possiamo anche "scartare" una lista di liste… vediamo l'esempio dello scioglimento di una lista di liste in una lista concatenando

```
 -- prendendo da due liste è facile costruire
 -- la lista `prodotto cartesiano'
> [(x,y) | x<-[1,2,3], y<-[4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
 -- attenti all'ordine!
> [(x,y) |  y<-[4,5], x<-[1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]

myConcat xss = [x | x<-xs, xs<-xss]
> myConcat [[1,2,3],[4,5],[6,7],[8]]
[1,2,3,4,5,6,7,8]

-- ma anche:
myConcat' = foldl (++) []
```

Un grande classico della propaganda

Questo è un programma da libro di scuola, che mostra quanto sia facile programmare **quicksort** in Haskell.

Ma si tratterà di verità o propaganda?

```
 -- prendendo da due liste è facile costruire
 -- la lista `prodotto cartesiano’
qsort []     = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a<-xs, a<=x ]
      larger  = [b | b<-xs, b>x ]

 -- possiamo tuttavia evitare le due passate
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
   where
      (smaller, larger) = partiziona xs x
      partiziona (x:xs) p =
         if x<=p then (x:s,l) else (s, x:l)
              where (l,s) = partiziona xs
      partiziona [] p = ([], [])
```

Lezione 5b:

Pensare Soluzioni a Oggetti

Problema: giochi da scacchiera

Il problema è costruire un'applicazione che gestisce una **partita a scacchi tra due giocatori** (umani) e che verifica la correttezza delle mosse.

L'obiettivo è di dare una soluzione **il più possibile orientata agli oggetti** e sfruttare le possibilità data dall'OOP di scrivere soluzioni **incrementali**, **modulari** e facilmente **estendibili**.

Cercheremo di evidenziare i vantaggi che possiamo avere in **generalità**, **semplicità** del codice (al prezzo di una certa ridondanza) e **flessibilità** rispetto a future estensioni (**riuso** del codice).

Struttura di base

Un ciclo continuamente richiede la **codifica** (come sequenza di caratteri) **di una mossa** e la invia a una classe `Partita` che risponde in uno dei seguenti modi:

1. la partita **è finita** (patta o vittoria di uno dei due giocatori)
2. un **codice di errore** che dà informazione sul perché la mossa non è legale.
3. **niente da segnalare**, la partita può continuare.

```
int main(){
   Partita bn;
   int res;
   char mossa[6];
   do {  bn.print();
         scanf("%s", mossa);
         if (mossa[0]=='q') break;
         res = bn.muovi(mossa);
         if (res == FINE) break;
         printError(res);
   } while(true);
  return 1;}
```

Obiettivo 1: modularizzare il codice

La classe `Partita` mantiene **lo stato del gioco** e verifica la correttezza delle mosse: se la mossa è legale, modifica lo stato del gioco, invertendo il colore del giocatore di turno e spostando i pezzi.

Verifica inoltre che non si siano verificati **casi di fine della partita**: vittoria o patta.

Se la mossa non è legale, torna un **codice di errore**.

Semplificando, lo **stato del gioco** è costituito da:

1. la **posizione dei pezzi** sulla scacchiera;
2. il giocatore di **turno** (Bianco o Nero)

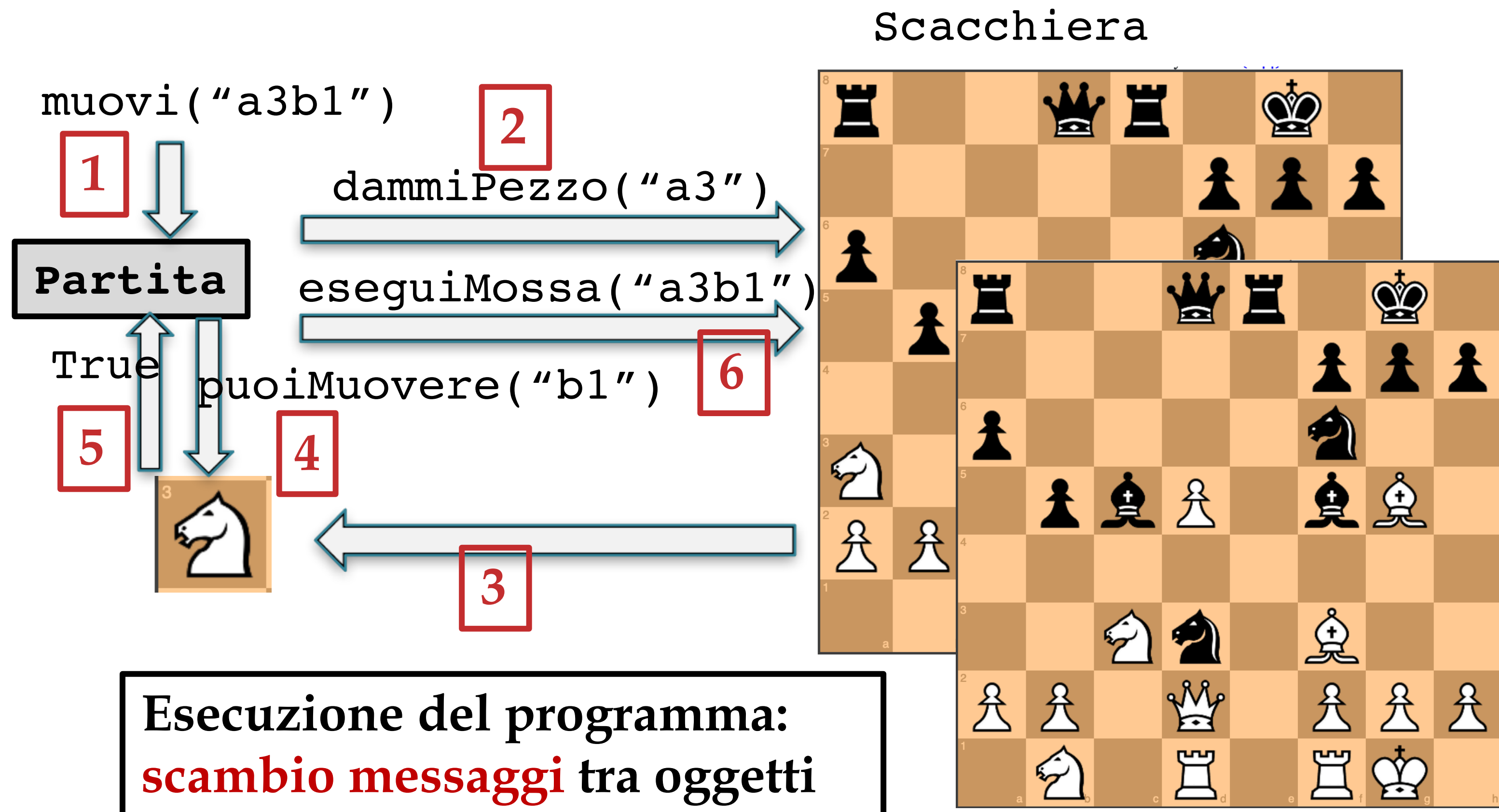
Sono per la verità necessarie **altre informazioni** per valutare la legalità di mosse speciali (arrocco, presa en-passant, etc.).

Può essere inoltre comoda altra **informazione ridondante** per rendere più semplice qualche operazione (vedremo): il prezzo sarà mantenere la **consistenza** di tale informazione.

Obiettivo 1: modularizzare il codice

Per modularizzare il più possibile il codice, scriveremo **una classe per ogni tipo di pezzo**: così facendo, le regole di movimento saranno scritte in piccole classi.

Fissiamo un **protocollo di messaggi tra oggetti** per la verifica della legalità della mossa.



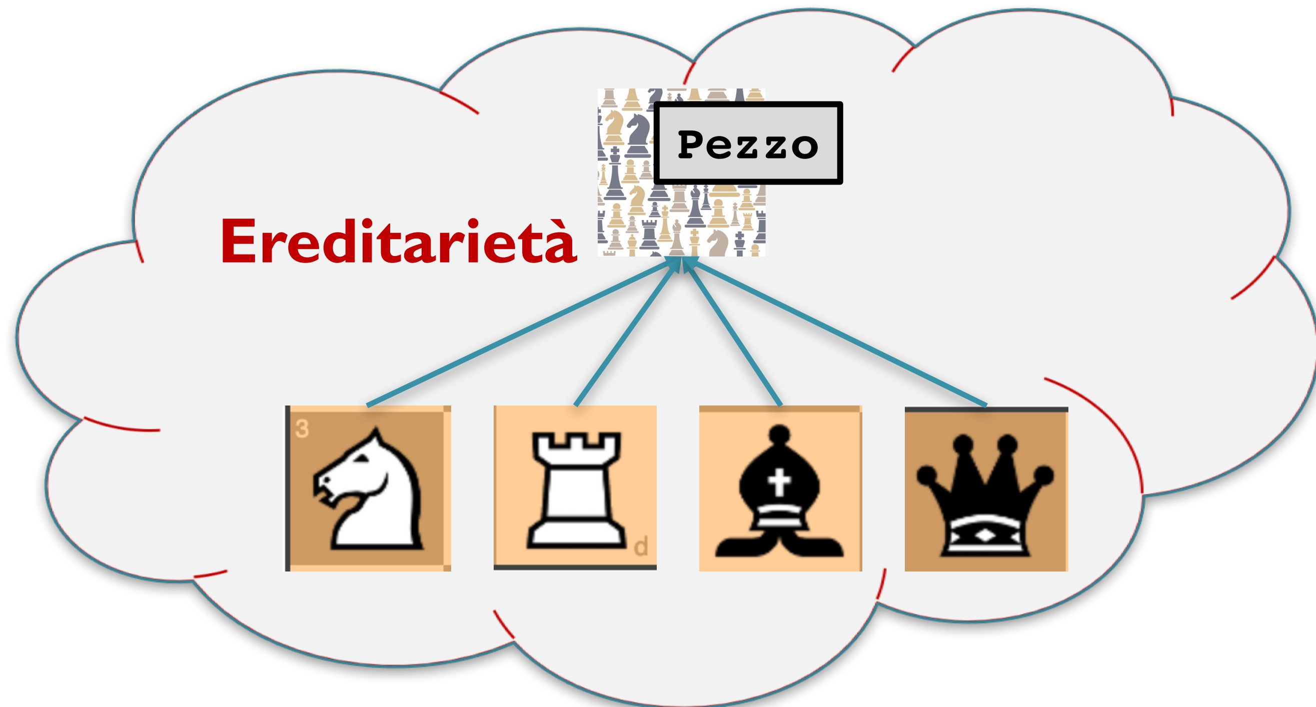
Implementazione: Scacchiera

La scacchiera può chiaramente essere una **matrice di pezzi**.

Meglio: una matrice di **puntatori a pezzi**.

Ma **ho una classe per ogni diverso tipo di pezzo** (Donna, Cavallo, Re, Torre, Alfiere): **come definire la matrice**?

Definire una **superclasse comune**: `Pezzo`.



Protocollo della mossa: classe Partita

La classe partita ha un riferimento a `Scacchiera` e a tutte le altre informazioni che descrivono lo stato del gioco.

Vediamo il protocollo della mossa descritto prima, tralasciando il controllo di mosse speciali (arrocco, presa en-passant etc.).

La funzione `eseguiMossa` aggiorna coerentemente lo stato.

```
int Partita::muovi(Casella da, Casella a){
   Pezzo *p=s->dammiPezzo(da);
   if (p==NULL) return CASELLA_VUOTA;
   if (p->colore()!=turno) return TURNO_ERRATO;
   int res=p->puoiMuovere(da, a, s);
   printError(res);
      /* analisi mosse speciali*/
      /* analisi fine partita*/
   ...
   if (res==LEGALE) Mossa m = eseguiMossa(da,a);
      /* aggiunge m a lista di mosse etc. */
   return res;
}
```

Implementazione: classe Pezzo

Vediamo la parte principale dell'**interfaccia di `Pezzo`**.

Osservate che le regole di movimento dipendono **dal pezzo**, ma ovviamente anche dalla **posizione di altri pezzi**, per cui è necessario passare a `puoiMuovere` la scacchiera.

In alternativa, potevamo mettere dentro `Pezzo` un **riferimento alla scacchiera a cui il pezzo appartiene**.

```
class Pezzo(){
      Colore col;
      Posizione pos;
   public:
      virtual int puoiMuovere(Casella da,
            Casella a, Scacchiera s);
      virtual int puoiMuovere(Casella a,
            Scacchiera s);
      virtual set<Casella> allMoves(Scacchiera s);
      virtual void print()=0;
      void move(Casella a)
}
```

`print` *è un metodo virtuale puro o astratto: il suo codice apparirà solo nelle sottoclassi*

Metodo puoiMuovere in Pezzo

Il metodo `puoiMuovere` può avere un codice significativo già nella classe **astratta** `Pezzo` (astratta nel senso che **non ci saranno oggetti creati dalla classe** `Pezzo`).

Si possono mettere dei controlli che verificano delle regole a cui tutte le mosse degli scacchi devono obbedire:

- un Pezzo **non può rimanere fermo**
- un Pezzo non può mai muovere in **una casella occupata da un pezzo dello stesso colore**.

```
int Pezzo::puoiMuovere(Casella da, Casella a,
                       Scacchiera *s){
   if (da == a) return CASELLA_UGUALE;
   Pezzo *p = s.dammiPezzo(s, a);
   if (p != NULL && p->colore() == col)
        return COLORE_UGUALE;
   return LEGALE;
}
```

Metodo puoiMuovere in Cavallo

Il metodo `puoiMuovere` sarà poi **ridefinito in ogni sottoclasse**: vediamo il Cavallo.

Il metodo comincerà con la chiamata al super-metodo che fa (una volta sola) i controlli generali.

Per scrivere meno codice, memorizzo in un array globale le direzioni del cavallo.

```
 -- memorizzo in variabili globali le direzioni
CavDirX[8]={1,1,2,2,-1,-1,-2,-2};
CavDirY[8]={2,-2,1,-1,2,-2,1,-1};

int Cavallo::puoiMuovere(Casella da, Casella a,
                         Scacchiera *s){
   int res = Pezzo::puoiMuovere(da, a, s);
   if (res != LEGALE) return res;
   for (int i=0; i<8; i++)
     if (a.x==da.x+CavDirX[i] &&
          a.y==da.y+CavDirY[i]) return LEGALE;
   return NON_SALTO_DI_CAVALLO;
}
```

Metodo puoiMuovere in Alfiere

Vediamo il caso dell'Alfiere.

Si comincia sempre con i controlli generali…

È facile verificare che le caselle stiano in una stessa diagonale…

… ma poi devo controllare che sulla strada dell'Alfiere **non ci siano ostacoli** (next slide vediamo linea libera)

```
int Alfiere::puoiMuovere(Casella da, Casella a,
                    Scacchiera *s){
   int res = Pezzo::puoiMuovere(da, a, s);
   if (res != LEGALE) return res;
   if (da.x+da.y!=a.x+a.y && da.x-da.y!=a.x-a.y)
      return NON_STESSA_DIAGONALE;
   if (s->linealibera(da, a))
      return LEGALE
   else return LINEA_NON_LIBERA;
}
```

Funzione linea libera

Confrontando le caselle di partenza e arriva determina la direzione.

Percorre la linea analizzando tutte le caselle lungo una direzione.

```
bool Scacchiera::lineaLibera(Casella da, Casella a){
/* PREC: assume da e a su stessa colonna o diagonale
 * POST: restituice true se la linea tra da e a
 * non contiene ostacoli */
   int dirx = calcolaDirezione(da.x, a.x);
   int diry = calcolaDirezione(da.y, a.y);
   Casella c(da.x+dirx, da.y+diry);
   while (c!=a){
      if (dammiPezzo(c)!=NULL) return false;
      c.move(dirx, diry);
   }
   return true;
}
```

Stampa della Scacchiera

Chiudiamo con un grande classico di **delegation**: il metodo `print()`.

La **scacchiera chiede ai pezzi di stamparsi**. Stampa `'.'` se la casella non contiene alcun pezzo.

Per ora, ogni pezzo semplicemente stampa sé stesso stampando una lettera (`A`(`a`) per alfiere bianco (nero), `C`(`c`) per cavallo bianco(nero) etc.).

Un'eventuale **interfaccia grafica** va scritta sopra a questo codice: **l'interazione grafica produce i messaggi da inviare alle oggetti non grafici**.

```
bool Scacchiera::print(){
   for (int i=0; i<RIG; i++){
      for (int j=0; j<COL; j++)
         if (s[i][j]==NULL) printf(".");
              else s[i][j]->print();
      printf("\n");
   } /* end for */
} /* end print */
```

Lezione 5

That's all Folks…

Grazie per l'attenzione…

…Domande?