

Sistemi per l'elaborazione dell'informazione 2

Composti, visitatori e interpreti

Espressioni

Definite una gerarchia di oggetti composti per rappresentare funzioni reali con parametri; tutti gli oggetti che fanno parte di un composto devono implementare l'interfaccia di marcatura **Expression** (si parla di *interfaccia di marcatura* quando un'interfaccia non definisce metodi o costanti, ma serve solo a marcare un insieme di oggetti). Le foglie dei composti sono costanti o variabili (contenenti un **double** o una stringa, rispettivamente), mentre i nodi interni sono somme, prodotti (suggerimento: implementate i costruttori di somme e prodotti in modo che accettino due **Expression** come argomenti. Vi aiuterà nel terzo esercizio), o funzioni predefinite (come logaritmo ed esponenziale); notate che in quest'ultimo caso è possibile che ci sia un solo figlio. In generale, invece, i nodi interni conterranno un vettore di oggetti di tipo **Expression**.

Implementate correttamente **toString()** in maniera che la stampa di un composto dia un risultato esteticamente gradevole. Provate a scrivere un metodo **main()** in cui istanziate manualmente dei composti (tramite dei **new** annidati, per esempio) e li stampate.

Visitatori

Implementate il disegno visitatore nella gerarchia dei composti così creata. Definite nell'interfaccia un metodo **accept(ExpressionVisitor)** che prende come argomento un'istanza di **ExpressionVisitor**, che a sua volta specifica un visitatore per espressioni.

ExpressionVisitor possiede metodi **void preVisit(T)** e **Object postVisit(T, Object[])** per tutti i tipi **T** di nodi interni e **Object visit(T)** per tutti i tipi **T** di foglie..

L'implementazione di **accept()** nei nodi interni deve chiamare **preVisit()**, raccogliere gli **Object** generati dai figli in un vettore e passarlo a **postVisit()**.

Definite un visitatore **EvaluateVisitor** che accetta nel costruttore una mappa (un'istanza di **Map**) da nomi di variabili a valori, e valuta un'espressione utilizzando la mappa data per valutare le variabili. Per ridurre la quantità di forzature di tipo, dichiarate la mappa come

```
Map<String,Double> m = new HashMap<String,Double>;
```

Ragionate su quali dei metodi richiesti da **ExpressionVisitor** sono effettivamente necessari, e chiedetevi perché stiamo utilizzando massicciamente **Object** quando per i nostri scopi sarebbe sufficiente **double**.

Aggiungete nella vostra classe di test la definizione di un **EvaluateVisitor** che vi permetta di valutare le espressioni precedentemente inserite.

Interprete

Per finire, scrivete un interprete per espressioni in forma polacca (cioè gli operandi *seguono* l'operatore: **+ 3 5**) che restituisce un composto, e utilizzatelo per fornire un metodo statico di utilità **eval()** che accetta un'espressione in forma di stringa, una mappa da nomi di variabili a valori, e restituisce il valore dell'espressione.

Per scrivere l'analizzatore sintattico, fissate l'arietà degli operatori (cioè, per esempio, dopo $+$ ci sono sempre due espressioni). In questo modo basta scrivere una funzione ricorsiva `interpret` che accetta un `Reader`, tenta di leggere un operatore, decide il numero k di sottoespressioni seguono e si invoca ricorsivamente k volte. Se $k = 0$ abbiamo a che fare con una costante, e possiamo costruire una foglia, altrimenti i risultati delle chiamate ricorsive vengono inserite in un opportuno nodo interno. Quando volete interpretare una stringa, utilizzate l'adattatore `StringReader`.

Scrivete un metodo `main()` che legge le righe scritte dall'utente: se sono della forma `nomevar=valore` viene aggiunto alla mappa il nome della variabile con il valore corrispondente, altrimenti la riga viene passata all'interprete con la mappa `variable/valore` corrente e il valore risultante viene stampato.

Esercizio facoltativo

Esiste una nozione di *interpretazione astratta* di un'espressione: data un'espressione E e una mappa da variabili a valori M , l'interpretazione astratta di E in M è una nuova espressione in cui la valutazione è stata spinta il più possibile con i valori noti. Per esempio, l'interpretazione astratta di $x + 1$ quando M assegna un valore solo a y è sempre $x + 1$, perché non è possibile semplificare ulteriormente l'espressione, ma se la mappa assegna a x il valore 0 l'interpretazione astratta è 1. Provate a implementare un visitatore restituisce una nuova espressione che rappresenta l'interpretazione astratta di un'espressione data. Notate che, se volete essere particolarmente zelanti, dovete riordinare le operazioni commutative in modo da valutare astrattamente $0 + x + 1$ a $x + 1$ anche se la mappa M è vuota.

Un altro esercizio facoltativo

Nel caso aveste nostalgia delle interfacce grafiche, vale la pena di notare che un modo interessante di utilizzare un visitatore è quello di sfruttarlo al fine di *visualizzare* graficamente un'espressione tramite un albero. Peraltro, questo vi permetterà anche di sperimentare la frustrazione che inevitabilmente si prova quando risulta necessario aggiungere un *diverso* tipo di visita su una struttura già pronta.

Il trucco è relativamente semplice: basta definire i metodi visita in modo che abbiano degli argomenti che servono le coordinate del punto in cui andrà posizionato il testo che descrive l'operazione (il simbolo dell'operatore o il nome della funzione) o gli operandi (il simbolo di una variabile o il valore di una costante). Più precisamente, conviene posizionare mantenere due variabili che contengono l'ascissa sinistra e destra della parte di canvas su cui si sta lavorando, in modo che la radice dell'espressione venga visualizzata nel loro punto medio, mentre i nodi degli operandi risultino equispaziati nella parte sottostante. La gestione delle ordinate, per fortuna, è più facile, basta aggiungere un valore fisso (che vi conviene memorizzare in una costante) ogni volta che si considera una sotto-espressione.

Divertitevi a disegnare i rami che collegano via via i nodi dell'espressione, e, se siete veramente coraggiosi, a disegnare dei cerchi decenti attorno ai nodi.