Progetto POO # 1 - A.A. 2017/2018: Valutazione di una espressione aritmetica intera

Mattia Gatto

Matricola:182926

II anno Ingegneria Informatica

**Traccia**

Progetto POO # 1 - A.A. 2017/2018: Valutazione di una espressione aritmetica intera

Si ammettono gli operatori +,-,\*,%,^ e valgono le usuali priorità della matematica, cioè ( priorità):

(^)>(\*,/,%)>(+,-)

A parità di priorità, si assume l’associatività a sinistra. Eventualmente, si possono usare le parentesi per alterare le priorità intrinseche: un’espressione in parentesi () viene sempre valutata prima.

Algoritmo di valutazione Si usano due stack: uno stack di operandi, ed uno stack di caratteri operatori. Quando arriva un operando, lo si inserisce in cima allo stack di operandi. Quando arriva un operatore, sia esso opc (operatore corrente), si procede come segue:  A) Se opc è più prioritario dell’operatore affiorante dallo stack di operatori o tale stack è vuoto, si inserisce opc in cima allo stack degli operatori.  B) Se opc non è più prioritario rispetto alla cima dello stack operatori, si preleva l’operatore op al top dello stack operatori, quindi si prelevano due operandi o2 (top) e o1 (top-1) dallo stack operandi (in caso di eccezioni, l’espressione è malformata). Si esegue o1 op o2. Il risultato è inserito in cima allo stack operandi. Si continua ad eseguire il passo B) se opc risulta ancora non più prioritario dell’operatore affiorante la cima dello stack operatori. Dopo questo, o perché opc è più prioritario dell’operatore in cima allo stack operatori o perché lo stack è vuoto, si applica il caso A). Quando termina una (sotto)espressione, se lo stack operatori non è vuoto, si estraggono uno alla volta gli operatori presenti e si applicano ai rispettivi due operandi prelevati dallo stack operandi, come spiegato al punto B), inserendo ogni volta il risultato in cima allo stack operandi. Quando lo stack operatori è vuoto, allora lo stack operandi dovrebbe contenere un solo elemento che è il risultato dell’espressione. Ogni altra situazione (stack operandi vuoto o con più di un elemento) denota una situazione di espressione malformata. Si suggerisce di dividere il compito della valutazione in due metodi: int valutaOperando(…) e int valutaEspressione(…). Questi due metodi possono ricevere come parametro l’oggetto string tokenizer “acceso” sulla stringa espressione, in modo da ripartire sempre dall’ultima posizione raggiunta. Per il confronto di operatori, introdurre una classe Precedenza che implementa Comparator<Character> il cui metodo compare(…) riceve due caratteri operatori e ritorna il loro confronto secondo le priorità della matematica.

Cosa succede se ci sono parentesi ?

Quando in valutaOperando(…) si incontra una parentesi aperta ‘(‘ anziché un normale operando intero, si invoca ricorsivamente la procedura valutaEspressione(…). Quando in valutaEspressione(…) si incontra una parentesi chiusa ‘)’ come operatore, occorre ritornare l’operando in cima allo stack operandi (un solo elemento o l’espressione è malformata). Attenzione: al termine di un’espressione (o sotto espressione) se esistono operatori sullo stack di operatori, questi vanno ordinatamente processati, uno alla volta. Alla fine il risultato dell’espressione (o sotto espressione) si dovrebbe trovare in cima allo stack operandi (a meno di malformazioni). Il metodo valutaEspressione(…) è opportuno che introduca i due stack come proprie variabili locali, in quanto assistono la valutazione della (sotto) espressione.

Il progetto dovrebbe realizzare un programma valutatore interattivo di espressioni aritmetiche, fornite a fronte di un prompt “>> “. Ogni espressione aritmetica deve essere seguita, sulla linea successiva, dal suo risultato o dalla segnalazione “Espressione malformata!”. Si esce quando l’utente digita una pseudo espressione “.”.

Sviluppi futuri Nel seguito del corso si provvederà ad estendere il progetto prevedendo una minima GUI di interazione con l’utente e la verifica di casi evidenti di malformazioni in una espressione aritmetica mediante l’uso di espressioni regolari.

Inizialmente ho creato una cartella(javaProgect)denominata ProgettoPOO1 all’interno della quale ho creato due package: uno (poo.progetto1) dove ho creato una classe “Espressione” per risolvere l algoritmo si valutazione e un’ altra “FinestraValuta” dove ho esteso una piccola GUI di iterazione con l’utente; un secondo (poo.util) dove ho creato le classi interfaccia , astratta e concreta dello stack da utilizzare come lista per risolvere l’algoritmo.

Nella classe Espressione:

**package** poo.progetto1;

**import** java.util.Comparator;

**import** java.util.StringTokenizer;

**import** javax.swing.JOptionPane;

**import** poo.util.StackConcatenato;

Inizialmente abbiamo importato il Comparator che utilizzeremo per verificare i canoni di precedenza degli operatori, lo StringTokenizer per prendere i singoli elementi di una stringa inserita per verificare se si trattino di operatori o operandi,un JOptionPane che utilizzerò per mostrare all’ utente le istruzioni rispettive ai canoni di validità di una espressione inserita.

**public** **class** Espressione {

StringTokenizer st;

Comparatore precedenza= **new** Comparatore();

**public** Espressione(String s){

st=**new** StringTokenizer(s,"()^%\*/+-",**true**) ;

}

**public** **int** valuta(){

**return** valutaEspressione(st);

}

Qui creiamo inizialmente uno StringTokenizer di nome “st” da utilizzare successivamente per scannerizzare i singoli elementi della Stringa, e un Comparatore di nome “precedenza” che andrà a creare un nuovo comparatore.

Creiamo un metodo Espressione(String s) che riceve una qualsiasi Stringa ‘s’ e restituisce un nuovo StringTokenizer(s,"()^%\*/+-",**true**) che scannerizza la stringa ‘s’ con i delimitatori che sono tra virgolette nella parentesi e li rende visibili grazie al true (li restituisce).

**public** **int** valutaEspressione(StringTokenizer st){

StackConcatenato<Character> operatori= **new** StackConcatenato<>();

StackConcatenato<Integer> operandi =**new** StackConcatenato<>();

operandi.push(valutaOperando(st));

**while**(st.hasMoreTokens()){

**char** c= st.nextToken().charAt(0);

**if**(c==')'){

**while**(operatori.size()>0){

**char** opc=operatori.pop();

**int** o1= operandi.pop();

**int** o2=operandi.pop();

**switch**(opc){

**case**'^':operandi.push((**int**)Math.*pow*(o2,o1));**break**; **case** '+':operandi.push(o2+o1);**break**;

**case** '\*':operandi.push(o2\*o1);**break**;

**case** '/':operandi.push(o2/o1);**break**;

**case** '-':operandi.push(o2-o1);**break**;

**case** '%':operandi.push(o2%o1);**break**;

**default**: **throw** **new** RuntimeException("Espressione malformata");

}

}

**return** operandi.top();

}

**while** (operatori.size()>0&&precedenza.compare(c,operatori.top())<=0){

**int** o1= operandi.pop();

**int** o2=operandi.pop();

**switch**(operatori.pop()){

**case** '^':operandi.push((**int**)Math.*pow*(o2,o1));**break**;

**case** '+':operandi.push(o2+o1);**break**;

**case** '\*':operandi.push(o2\*o1);**break**;

**case** '/':operandi.push(o2/o1);**break**;

**case** '-':operandi.push(o2-o1);**break**;

**case** '%':operandi.push(o2%o1);**break**;

**default** :**throw** **new** RuntimeException("Espressione malformata");

}

}

operatori.push(c);operandi.push(valutaOperando(st));

}

**while**(operatori.size()>0){

**int** o1= operandi.pop();

**int** o2=operandi.pop();

**switch**(operatori.pop()){

**case** '^':operandi.push((**int**)Math.*pow*(o2,o1));**break**;

**case** '+':operandi.push(o2+o1);**break**;

**case** '\*':operandi.push(o2\*o1);**break**;

**case** '/':operandi.push(o2/o1);**break**;

**case** '-':operandi.push(o2-o1);**break**;

**case** '%':operandi.push(o2%o1);**break**;

**default** :**throw** **new** RuntimeException("Espressione malformata");

}

}

**return** operandi.top();

}

Nel metodo creato di nome valutaEspressione che riceve come parametro uno StringTokenizer ‘st’, creiamo un nuovo StackConcatenato<Character> di caratteri chiamato “operatori” che riceverà gli operatori e un nuovo StackConcatenato<Integer> di interi chiamato “operandi” che riceverà i numeri.

Inseriamo ora in cima allo stack grazie al metodo push il ritorno dal metodo valuta operando che restituirà un token di tipo intero.

Ora impostiamo che fino a quando ci sono altri token(caratteri) dalla stringa di partenza, il char c che sarà il prossimo token in posizione 0:

1. se sarà una parentesi tonda chiusa ”)”, vuol dire che è finita una sotto-espresione e quindi fino a quanto lo stack degli operatori avrà una dimensione maggiore di zero(quindi sarà ancora pieno) andremo a prendere il char ‘opc’ che sarà il segno di tipo carattere inserito nello stack degli operatori(grazie al metodo pop che prende e rimuove l’elemento in prima posizione nello stack), o1 che sarà il primo numero di tipo intero inserito nello stack degli operandi e o2 che sarà il prossimo primo numero di tipo intero inserito nello stack degli operandi.  
   Ora analizziamo opc:

* se il segno opc ='^' allora mettiamo in prima posizione nello stack degli operandi il numero intero dato dall’ operazione o2 elevato ad o1;
* se opc ='+' allora mettiamo in prima posizione nello stack degli operandi il numero intero dato dall’ operazione o2 + o1;
* se opc ='\*' allora mettiamo in prima posizione nello stack degli operandi il numero intero dato dall’ operazione o2 \* o1;
* se opc ='/' allora mettiamo in prima posizione nello stack degli operandi il numero intero dato dall’ operazione o2 / o1;
* se opc ='-' allora mettiamo in prima posizione nello stack degli operandi il numero intero dato dall’ operazione o2 - o1;
* se opc ='%' allora mettiamo in prima posizione nello stack degli operandi il numero intero dato dall’ operazione o2 % o1;
* Se opc è diverso dai precedenti casi sarà un “Espressione Malfomata”;  
  Alla fine restituiamo l’elemento in prima posizione nello stack operandi.

1. Ora fino a quanto lo stack degli operatori contiene almeno un elemento (fino a quanto non è vuoto) e il comparatore precedenza fra c e il primo elemento nello stack degli operatori e minore o uguale a zero, ossia se operatori.top a precedenza maggiore rispetto a c, allora prendiamo o1 che sarà il primo numero di tipo intero inserito nello stack degli operandi e o2 che sarà il prossimo primo numero di tipo intero inserito nello stack degli operandi e analizziamo opc come in precedenza.  
   Ala fine mettiamo c in prima posizione nello stack operatori e richiamiamo il metodo valuta operando dello String tokenizer st, sul primo elemento dello stack degli operandi.
2. Alla fine usciti dal while che verifica se ci sono altri token, facciamo un ultimo controllo dicendo che fino a quanto lo stack degli operatori ha una dimensione maggiore di zero prendiamo o1 che sarà il primo numero di tipo intero inserito nello stack degli operandi e o2 che sarà il prossimo primo numero di tipo intero inserito nello stack degli operandi e analizziamo opc come in precedenza.  
   E alla fine restituiamo tramite la funzione top il primo elemento in cima allo stack degli operandi.

**public** **int** valutaOperando(StringTokenizer st){

String c= st.nextToken();

**if**(c.equals("("))**return** valutaEspressione(st);

/\*\*

se volevamo gestire il caso di un "-" all' inizio dell' espressione:

else if(c.equals("-"))return 0-valutaOperando(st);

\*\*/

**return** Integer.*parseInt*(c);

}

Il metodo valutaOperando restituisce un itero e prende come parametro lo StringTokenizer ‘st’, allora qui creiamo una stringa ‘c’ che prende il prossimo token e verifica se c= ad una parentesi aperta”(” in tal caso ritorna il metodo valutaEspressione su ‘st’, altrimenti ritorna c trasformato in intero tramite il metodo Integer.*parseInt.*qui se volevamo verificare il caso in cui come primo elemento o token della stringa era un meno allora potevamo fare un else if nel quale controllavamo se c =”-” e in tal caso restituivamo 0- il metodo valutaOperando con parametro st.(in modo tale che se abbiamo una espressione che inizia con meno la analizziamo come 0- l’ espressione.)

**private** **class** Comparatore **implements** Comparator<Character>{

**public** **int** compare(Character primo, Character secondo){

**switch**(primo){

**case** '^':**return** 1;

**case** '\*':  
**if**(secondo.equals('^'))**return** -1;**if**(secondo.equals('/')||secondo.equals('%'))**return** 0;**return** 1;

**case** '/':

**if**(secondo.equals('^'))**return** -1;**if**(secondo.equals('\*')||secondo.equals('%'))**return** 0;**return** 1;

**case** '%':

**if**(secondo.equals('^'))**return** -1;**if**(secondo.equals('/')||secondo.equals('\*'))**return** 0;**return** 1;

**case** '+':**if**(secondo.equals('-'))**return** 0;**return** -1;

**case** '-':**if**(secondo.equals('+'))**return** 0;**return** -1;

}

**return** 0;

}

}

In questa classe dichiarata privata come inner-class e che estende un Comparator<Character> di tipo carattere, facciamo un metodo che restituisce un intero chiamato compare e riceve come parametri un carattere chiamato “primo” e un altro chiamato “secondo”.

Qui facciamo uno switch riguardo al parametro primo e vediamo che:

* Se primo =’^’ allora restituiamo 1;
* Se primo =’\*’ allora:  
   se secondo=’^’ restituiamo-1;  
   se secondo=’%’o =’/’ allora restituiamo 0;  
   altrimenti restituiamo 1( perché secondo sarà = ‘\*’,o=‘+’);
* Se primo =’/’ allora:  
   se secondo=’^’ restituiamo-1;  
   se secondo=’%’o =’\*’ allora restituiamo 0;  
   altrimenti restituiamo 1( perché secondo sarà = ‘\*’,o=‘+’);
* Se primo =’%’ allora:  
   se secondo=’^’ restituiamo-1;  
   se secondo=’\*’o =’/’ allora restituiamo 0;  
   altrimenti restituiamo 1( perché secondo sarà = ‘\*’,o=‘+’);
* Se primo =’+’ allora:  
   se secondo=’-‘ allora restituiamo 0;  
   altrimenti restituiamo -1;
* Se primo =’-’ allora:  
   se secondo=’+‘ allora restituiamo 0;  
   altrimenti restituiamo -1;  
    
    
    
    
    
    
    
  **public** **static** **void** main (String [] args){

//Primo main con un programma valutatore interattivo di espressioni aritmetiche, fornite a fronte di un prompt “>> “

/\*\*

System.out.println(">>Inserire un espressione da risolvere cosiderando i seguenti punti:");

System.out.println(">> 1) Gli operatori utilizzabili sono: -, +, \*, /, %, ^; ");

System.out.println(">> 2) Si possono usare le parentesi tonde:'(' aperta e')'chiusa per delimitare una sotto-espressione; ");

System.out.println(">> 3) Inserisci '.' per uscire;");

Scanner sc= new Scanner(System.in);

String s=null;

for(;;){

System.out.println(">>Inserire espressione:");

System.out.print(">>");

s=sc.nextLine();

if(s.equals(".")) break;

try{

int risultato=new Espressione(s).valuta();

System.out.println(">>Il risultato dell espressione:\n"+">>"+s+" = "+risultato);

}catch(Exception e){

System.out.println("Espressione malformata");

}

}

sc.close();

System.exit(0);

\*\*/

//un programma valutatore interattivo di espressioni aritmetiche, fornite a fronte di un interfaccia grafica con la

//gestione di casi evidenti di malformazioni in una espressione aritmetica mediante l’uso di espressioni regolari.

JOptionPane.*showMessageDialog*(**null**,"Inserire un espressione da risolvere cosiderando i seguenti punti:\n1) Gli operatori utilizzabili sono: -, +, \*, /, %, ^;\n2) Si possono usare le parentesi tonde:'(' aperta e')'chiusa per delimitare una sotto-espressione;\n3)Premi: VALUTA per sapere il risultato dell' espressione, mentre RESETTA per cancellare l'espressione; ");

FinestraValuta fv= **new** FinestraValuta();

fv.setVisible(**true**);

}

}

Qui alla fine vediamo il metodo main che può essere fatto in due modi, al momento è nella versione in modalità GUI, ma si potrebbe anche fare con un valutatore interattivo di espressioni aritmetiche, fornite a fronte di un prompt “>> “:

* Se lo vogliamo fare con un valutatore interattivo di espressioni aritmetiche, fornite a fronte di un prompt “>> “,allora dopo aver inserito le istruzioni con dei System.out.println, apriamo uno Scanner sc e inizializziamo una String ‘s’ a null.  
  Successivamente apriamo un for(;;) per introdurre un ciclo infinito e qui inseriamo con dei System.out.print le istruzioni su cosa inserire e dopo aver fatto questo, prendiamo la stringa con il comando sc.nextLine() e la memorizziamo in ‘s’, ora se ‘s’ fosse uguale ad un punto usciamo dal ciclo altrimenti apriamo un try dove prendiamo il risultato dell’ espressione data da Espressione(s).valuta() e la memorizziamo in una variabile intera chiamata ”risultato”, altrimenti facciamo catch(Exception e){System.out.println("Espressione malformata");} per gestire un eventuale eccezione; alla fine del ciclo for chiudiamo lo scanner sc con la funzione close() e usciamo dal programma con System.exit(0);
* Se invece vogliamo un programma valutatore interattivo di espressioni aritmetiche, fornite a fronte di un interfaccia grafica con la gestione di casi evidenti di malformazioni in una espressione aritmetica mediante l’uso di espressioni regolari:  
  utilizziamo un JOptionPane.showMessageDialog per inserire le istruzioni del programma e successivamente andiamo a creare una FinestraValuta chhiamata‘fv’ e la rendiamo visibile con la funzione fv.setVisible(true), da cui si andrà nella classe FinestraValuta che estende un JFrame e implementa un ActionListener;

package poo.progetto1;  
**import** java.awt.BorderLayout;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.util.StringTokenizer;

**import** javax.swing.JButton;

**import** javax.swing.JFrame;

**import** javax.swing.JLabel;

**import** javax.swing.JOptionPane;

**import** javax.swing.JPanel;

**import** javax.swing.JTextField;

Richiamati i dovuti import e messo il nome del package nel quale ci troviamo andiamo al passo successivo.

@SuppressWarnings("serial")

**public** **class** FinestraValuta **extends** JFrame **implements** ActionListener{

**private** JTextField espressione,risultato;

**private** JButton valuta,resetta;

**private** StringTokenizer st;

creiamo due JTextField chiamati “espressione” e “risultato” che saranno i dovuti campi di testo e poi due JButton chiamati “valuta”, ”resetta” che serviranno uno per calcolare l’ espressione, l’altro per resettare il campo di testo dell’ espressione.

**public** FinestraValuta(){

setTitle("Valutatore espressioni");

setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

JPanel p=**new** JPanel();

p.add( **new** JLabel("ESPRESSIONE", JLabel.***CENTER***) );

p.add( espressione=**new** JTextField("",20) );

add(p, BorderLayout.***NORTH***);

JPanel l=**new** JPanel();

l.add( **new** JLabel("RISULTATO", JLabel.***CENTER***) );

l.add( risultato=**new** JTextField("",10) );

add(l, BorderLayout.***CENTER***);

JPanel q=**new** JPanel();

q.add( valuta= **new** JButton("VALUTA") );

q.add( resetta= **new** JButton("RESETTA") );

add( q, BorderLayout.***SOUTH*** );

valuta.addActionListener( **this** );

resetta.addActionListener( **this** );

setLocation(300,300);

setSize(550,200);

}

Ora andiamo a creare un metodo chiamato FinestraValuta alla quale :

* Inseriremo come titolo del Frame “Valutatore espressione” tramite la funzione setTitle;
* Andiamo a creare un pannello nuovo chiamato ’p’ con la funzione JPanel p=new JPanel() e lo aggiungiamo ad un nuovo JLabel che verrà chiamato ‘ESPRESSIONE’ e sarà posizionato al centro del nostro frame, qui ora aggiungiamo a ‘p’ il JTextField chiamato “espressione” dichiarato privato sopra, di lunghezza 20 e aggiungiamo ‘p’ in posizione nord del BorderLayout;
* Andiamo a creare un pannello nuovo chiamato ’l’con la funzione JPanel l=new JPanel() e lo aggiungiamo ad un nuovo JLabel che verrà chiamato RISULTATO e sarà posizionato al centro del nostro frame, qui ora aggiungiamo a ‘l’ il JTextField chiamato “risultato” dichiarato privato sopra, di lunghezza 10 e aggiungiamo ‘l’ in posizione centro del BorderLayout;
* Andiamo a creare un pannello nuovo chiamato ’q’ con la funzione JPanel q=new JPanel() e aggiungiamo a ‘q’ un bottone ‘valuta’ di nome “VALUTA” e aggiungiamo a ‘q’ un altro bottone ‘resetta’ di nome “RESETTA” e infine aggiungiamo ‘q’ al BorderLayout in posizione SOUTH;
* Ora prendiamo valuta e gli aggiungiamo un addActionListener di parametro this e prendiamo resetta e gli aggiungiamo un addActionListener di parametro this e settiamo la locazione del frame e la sua dimensione;

**public** **void** actionPerformed(ActionEvent evt){//metodo callback

risultato.setText(**null**) ;

String FORMATO="[\\(\\d+\\)[\\+\\-\\\*\\/\\%\\^]]+";

String e=espressione.getText() ;

st=**new** StringTokenizer(e,"()",**true**);

**int** c=0;

**while**(st.hasMoreTokens()){

**char** s= st.nextToken().charAt(0);

**if**(s==('('))c++;

**if**(s==(')'))c--;

}

**if**( evt.getSource()==valuta){

**if**(e.matches(FORMATO)&&c==0){

**try** {

**double** ris=**new** Espressione(e).valuta();

risultato.setText( String.*format*("%1.0f",ris) );

}**catch**(RuntimeException s) {

JOptionPane.*showMessageDialog*(**null**,"Espressione malformata");

/\*\*

\* se volevamo resettare il campo dell' espressione quando l' espressione è malformata:

\* espressione.setText(null);

\*/

}

}

**else** {

JOptionPane.*showMessageDialog*(**null**,"Espressione malformata");

}

}

**if**( evt.getSource()==resetta ){

espressione.setText(**null**);

}

}

}//FinestraValuta  
Per conclusione andiamo a creare un metodo void actionPerformed di parametro ActionEvent ‘evt’ e inizialmente settiamo il campo di testo di risultato a null, poi creiamo una stringa FORMATO che sarà la nostra regex, prendiamo il testo nel campo di testo ‘espressione’ e lo memorizziamo nella stringa e grazie alla funzione getText, successivamente creiamo uno stringTokenizer di ‘e’ e prendiamo come delimitatori le parentesi tonde aperte e chiuse e le facciamo ritornare , creiamo una variabile intera ‘c’ che sarà il nostro contatore e la poniamo uguale a zero.  
Qui poniamo che fino a quanto ci saranno altri caratteri , mettiamo in un carattere ‘s’ il prossimo token in posizione zero e verifichiamo che, se è uguale ad una parentesi tonda aperta incrementiamo ‘c’ altrimenti se è una parentesi tonda chiusa lo decrementiamo.

Usciti dal while:

* facciamo un if e controlliamo se l’ evento ‘evt ‘è uguale a ‘valuta’ , in tal caso andiamo a fare un ulteriore controllo vedendo se ‘e’ rispecchia i canoni della regex con la funzione e.matches(FORMATO) e controlliamo che ‘c’ sia uguale a zero per verificare che ogni parentesi aperta è stata chiusa, infine se è tutto giusto apriamo un try e mettiamo in una variabile double chiamata’ris’ il metodo valuta di Espressione di ‘e’ e impostiamo con un setText nel campo di testo ‘risultato’il valore di ‘ris’, ma se provando con il try si verifica un eccezione entriamo nel catch(RuntimeException s) che cattura l’eventuale eccezione che abbiamo gestito nei metodi della classe Espressione e manda un messaggio all’ utente con un JOptionPane.showMessageDialog con scritto “Espressione malformata”.
* Facciamo un altro if e controlliamo se l’evento ‘evt ‘è uguale a ‘resetta’, in tal caso settiamo il campo di testo ‘espressione’ a null con la funzione espressione.setText(null);  
    
    
  Ecco come appare nell’ esecuzione:

