Appelli di settembre 2006 e gennaio 2007

SymbolicManipulator

Laurea triennale in Comunicazione Digitale
Laboratorio di Programmazione

1 – Descrizione

Il progetto consiste nell'implementare in JAVA un semplice linguaggio di manipolazione simbolica, in cui sia possibile costruire espressioni complesse a cui applicare degli operatori. Le espressioni più semplici sono:

- I **simboli**, individuati da un nome (una sequenza di caratteri che non contiene spazi) ed eventualmente (ma non necessariamente) associati a un valore numerico
- I **valori numerici**, che per semplicità implementeremo utilizzando solamente il tipo `double` di JAVA (cfr. la definizione della classe `NumericValue`).

A partire da queste semplici espressioni è possibile generare nuove epsressioni considerando ad esempio la derivata o la semplificazione di un'espressione esistente, oppure la somma o il prodotto di due espressioni esistenti. Deve essere inoltre possibile appoggiarsi a una memoria per poter associare ad alcuni simboli delle espressioni.

2 – Le classi da realizzare

E' richiesto di risolvere il progetto descritto nella sezione precedente realizzando in JAVA le seguenti classi:

- `Expression`, che descrive una generica espressione. La classe dovrà contenere:
  - la definizione del metodo `Expression eval()`, da chiamare per valutare l'espressione (nel testo che segue verrà descritto come implementare la valutazione di alcune espressioni quando questo procedimento non sarà ovvio);
  - la definizione del metodo `Object clone()`, che restituisce un nuovo oggetto che rappresenta una **copia** dell'espressione;
  - la definizione del metodo `String toString()`, che restituisce una descrizione testuale dell'istruzione (nel testo che segue verranno indicate in dettaglio le descrizioni dei vari tipi di espressione: **è richiesto di attenersi alla lettera ai formati che**

verranno indicati, senza aggiungere ad esempio spazi o caratteri di tabulazione);
  - la definizione del metodo `boolean equals(Expression e)`, che ritorna `true` se l'espressione su cui è chiamata è uguale all'espressione passata come argomento, e `false` altrimenti. Le implementazioni di questo metodo devono assumere che due espressioni sono uguali se sono dello stesso sottotipo di `Expression` e se le loro variabili di istanza hanno gli stessi contenuti (suggerimento: utilizzare l'operatore `istanceof` di Java).
  - la definizione del metodo `Expression derive(Symbol s)`, che ritorna la derivata dell'espressione rispetto al simbolo indicato come argomento (cfr. la descrizione della classe `Symbol`);
  - la definizione del metodo `Expression simplify()`, che restituisce un'espressione equivalente a quella su cui è chiamato il metodo, possibilmente resa più semplice a seguito di un processo di semplificazione (cfr. la descrizione delle classi seguenti).

- `Symbol`, che descrive un simbolo. La classe dovrà contenere:
  - La variabile d'istanza `String name`, contenente il nome del simbolo. Rappresentano nomi validi tutte le sequenze di caratteri non contenenti lo spazio.
  - L'implementazione del metodo `eval`, che deve ritornare l'espressione associata al simbolo se questa espressione esiste (cfr. la seguente classe `Memory`), e il simbolo stesso altrimenti.
  - L'implementazione del metodo `simplify`, che deve ritornare l'espressione associata al simbolo se questa espressione esiste, e il simbolo stesso altrimenti.
  - L'implementazione del metodo `toString`, che deve ritornare una stringa contenente un carattere di parentesi tonda aperta, seguito dal nome del simbolo, seguito a sua volta da un carattere di parentesi tonda chiusa.
- `NumericValue`, che descrive un valore numerico. La classe dovrà contenere:
  - La variabile d'istanza `double value`, contenente il valore.
  - L'implementazione del metodo `toString`, che deve ritornare una stringa contenente un carattere di parentesi tonda aperta, seguito dal valore numerico, seguito a sua volta da un carattere di parentesi tonda chiusa.
- `Plus`, che descrive la somma di due o più espressioni. La classe dovrà contenere:
  - La variabile d'istanza `Expression[] addends`, che contiene le espressioni da addizionare.
  - L'implementazione del metodo `equals`, in cui due istanze di `Plus` sono da considerare uguali solo se in ogni posizione i corrispondenti array `addends` contengono due espressioni equivalenti.
  - L'implementazione del metodo `simplify`, che deve semplificare i vari addendi ed eliminare quelli uguali a `0`.

- L'implementazione del metodo `toString`, che deve ritornare una stringa contenente un carattere di parentesi tonda aperta, seguito dalle descrizioni delle espressioni contenute nell'array `addends`, separate dal carattere `'+'`, seguite a loro volta da un carattere di parentesi tonda chiusa.

- `Times`, che descrive il prodotto di due o più espressioni. La classe dovrà contenere:
  - La variabile d'istanza `Expression[] factors`, che contiene le espressioni da moltiplicare.
  - L'implementazione del metodo `equals`, in cui due istanze di `Times` sono da considerare uguali solo se in ogni posizione i corrispondenti array `factors` contengono due espressioni equivalenti.
  - L'implementazione del metodo `simplify`, che deve semplificare i vari fattori, eliminando quelli uguali a 1 e ritornando il valore numerico corrispondente a `0.` nel caso uno dei fattori sia uguale a `0.`.
  - L'implementazione del metodo `toString`, che deve ritornare una stringa contenente un carattere di parentesi tonda aperta, seguito dalle descrizioni delle espressioni contenute nell'array `factors`, separate dal carattere `'*'`, seguite a loro volta da un carattere di parentesi tonda chiusa.
- `Exp`, che descrive l'elevamento a potenza utilizzando la base dei logaritmi naturali (e). La classe dovrà contenere:
  - La variabile d'istanza `Expression arg`, che contiene l'esponente a cui elevare la base e.
  - L'implementazione del metodo `equals`, in cui due istanze di `Exp` sono da considerare uguali solo se le corrispondenti variabili di istanza sono equivalenti.
  - L'implementazione del metodo `simplify`, che deve ritornare il valore numerico corrispondente a `1.` in caso l'esponente sia uguale a `0.`.
  - L'implementazione del metodo `toString`, che deve ritornare una stringa contenente un carattere di parentesi tonda aperta, seguito dalla stringa `"Exp"`, da un carattere di parentesi quadra aperta, dalla descrizione dell'espressione contenuta nella variabile d'istanza, da un carattere di parentesi quadra chiusa e da un carattere di parentesi tonda chiusa.
- `Power`, che descrive l'elevamento di una generica base a una potenza intera. La classe dovrà contenere:
  - La variabile d'istanza `Expression base`, che contiene la base da elevare a potenza.
  - La variabile d'istanza `int exponent`, che contiene l'esponente a cui elevare la base.
  - L'implementazione del metodo `equals`, in cui due istanze di `Power` sono da considerare uguali solo se le corrispondenti basi e i corrispondenti esponenti sono equivalenti.
  - L'implementazione del metodo `simplify`, che deve ritornare la base nel caso in cui l'esponente sia uguale al valore numerico

corrispondente a `1.` e il valore numerico `1.` nel caso in cui l'esponente sia uguale a `0.`.
- L'implementazione del metodo `toString`, che deve ritornare una stringa contenente un carattere di parentesi tonda aperta, seguito dalla descrizione dell'espressione contenuta nella variabile d'istanza `base`, da un carattere '^', dal valore della variabile d'istanza `exponent` e da un carattere di parentesi tonda chiusa.

- `Assign`, che descrive l'assegnamento di un'espressione ad un simbolo. La classe dovrà contenere:
  - La variabile d'istanza `Symbol s`, che contiene il simbolo cui assegnare un'espressione.
  - La variabile d'istanza `Expression e`, che contiene l'espressione da associare al simbolo.
  - L'implementazione del metodo `eval`, che deve scrivere nella memoria (cfr. la seguente descrizione della classe `Memory`) l'espressione contenuta in `e` in corrispondenza del nome del simbolo `s`, e poi ritornare l'espressione `e`.
  - L'implementazione del metodo `toString`, che deve ritornare una stringa contenente un carattere di parentesi tonda aperta, seguito dalla descrizione del simbolo contenuto in `s`, dal carattere `'='`, dalla descrizione dell'espressione contenuta in `e` e da un carattere di parentesi tonda chiusa.
- `Derive`, che descrive la derivata di una generica espressione. La classe dovrà contenere:
  - La variabile d'istanza `Expression e`, che contiene l'espressione da derivare;
  - La variabile d'istanza `Symbol s`, che contiene il simbolo rispetto a cui calcolare la derivata.
  - L'implementazione del metodo `eval`, che deve calcolare e ritornare la derivata dell'espressione `e` rispetto al simbolo `s`.
  - L'implementazione del metodo `toString`, che deve ritornare una stringa contenente un carattere di parentesi tonda aperta, seguito dalla stringa `'Der'`, da un carattere di parentesi quadra aperta, dalla descrizione dell'espressione contenuta in `e`, da un carattere `','`, dalla descrizione del simbolo contenuto in `s`, da un carattere di parentesi quadra chiusa e da un carattere di parentesi tonda chiusa.
- `Simplify`, che descrive l'espressione ottenuta semplificando una generica espressione. La classe dovrà contenere:
  - La variabile d'istanza `Expression e`, che contiene l'espressione da semplificare;
  - L'implementazione del metodo `eval`, che deve ritornare l'epsressione semplificata.
  - L'implementazione del metodo `toString`, che deve ritornare una stringa contenente un carattere di parentesi aperta, seguito dal carattere `'Sim'`, da un carattere di parentesi quadra aperta, dalla descrizione dell'espressione contenuta in `e`, da un carattere di parentesi quadra chiusa e da un carattere di parentesi tonda chiusa.

- `Memory`, che descrive la memoria utilizzata per leggere e scrivere le espressioni associate ai simboli. La classe dovrà contenere:
  - La variabile d'istanza `Hashtable state`, in cui memorizzare e leggere le espressioni associate ai simboli;
  - Il metodo `void write(Symbol s, Expression e)`, che memorizza nella hashtable `state` l'espressione `e` **in corrispondenza del nome** del simbolo `s`.
  - Il metodo `Expression read(Symbol s)`, che ritorna l'espressione memorizzata nell'hashtable in corrispondenza del nome del simbolo `s`. L'implementazione di questo metodo dovrà prevedere la possibilità che ad un simbolo non sia associata nessuna espressione: in tal caso il metodo deve ritornare il simbolo `s`.
  - Il metodo `void dump(String s)`, che scrive il contenuto della memoria su un file di testo il cui nome è contenuto nella stringa passata come argomento. Il formato di stampa è il seguente: per ogni coppia (simbolo, espressione) contenuta nella hashtable va scritta una riga nel file, contenente il nome del simbolo, seguito dal carattere `'='` e dalla descrizione testuale dell'espressione. **Il carattere '=' non deve essere preceduto né seguito da alcuno spazio o tabulazione**.
- `Notebook`, che descrive un elenco di espressioni contenute su file. La classe dovrà contenere:
  - Il costruttore `Notebook(String fn)`, che apre il file di testo, il cui nome è contenuto nell'argomento `fn`, che contiene l'elenco di espressioni.
  - Il metodo void process(), che legge il file di testo riga per riga interpretando ogni riga come un'espressione, e valuta le espressioni nella sequenza in cui sono state lette e infine chiude il file di testo.

  Per semplificare il lavoro di conversione delle righe di testo in espressioni è possibile utilizzare la classe `ExpressionReader` descritta nella sezione seguente.

A parte quanto espressamente richiesto, è lasciata piena libertà sull'implementazione delle singole classi e sull'eventuale introduzione di classi aggiuntive, a patto di seguire correttamente le regole del paradigma di programmazione orientata agli oggetti ed i principi di buona programmazione. Si suggerisce di porre particolare attenzione alla scelta dei modificatori relativi a variabili d'istanza e metodi, alla creazione di metodi di accesso alle variabili di istanza quando questi siano ritenuti necessari, alla definizione di metodi astratti dove questi risultino opportuni, nonché alla dichiarazione e alla gestione delle eccezioni che possono venire lanciate dai vari metodi e alle relazioni di ereditarietà tra le varie classi. Si ricorda altresì di rispettare alla lettera i formati descritti per i metodi `toString` e `dump`.

Non è richiesto l'utilizzo di particolari modalità grafiche di visualizzazione: è sufficiente una qualunque modalità di output basata sull'uso dei caratteri.

E' invece **espressamente richiesto** di non utilizzare package non standard di Java (si possono quindi utilizzare `java.util`, `java.io` e così via).

3 – La classe ExpressionReader

Per facilitare la lettura da file di un'espressione è possibile utilizzare la classe `ExpressionReader`, che, tra gli altri, implementa i seguenti metodi:

- Il costruttore `ExpressionReader(String fn)`, dove l'argomento rappresenta il nome di un file di testo contenente un elenco di espressioni separate dal carattere di "a capo". La chiamata di tale costruttore può lanciare l'eccezione `IOException`;
- Il metodo `Iterator iterator()`, che ritorna un iteratore contenente tutte le espressioni scritte sul file di testo descritto nel punto precedente, automaticamente convertite in oggetti della classe `Expression`. La chiamata di questo metodo può lanciare le eccezioni `IOException` e `InvalidExpressionException`: quest'ultima eccezione viene lanciata quando una riga del file di testo contiene una stringa che non può essere convertita in un'espressione valida. La stampa dell'eccezione causa la stampa della riga che non può essere convertita.
- Il metodo `void close()`, che chiude il file contentente l'elenco di espressioni. La chiamata di questo metodo può lanciare l'eccezione `IOException`.

I file ExpressionReader.class e InvalidExpressionException.class possono essere scaricati dal sito web del corso. Sullo stesso sito sono disponibili alcuni file di testo che possono essere utilizzati per testare il corretto funzionamento del programma. Non saranno accettate le sottoposizioni che non funzionino correttamente rispetto a questi casi paradigmatici.

4 – Esempio di esecuzione

A titolo di esempio, si riporta un elenco di descrizioni di espressioni:

```
((x)=(0))
((z)=((y)+(x)))
((h)=(Sim[(z)]))
((r)=(Sim[((q)*(0))]))
((r)=(Sim[((1)+(e)+((q)*(0)))]))
((q)=(Sim[(Der[(r),(e)])]))
```

```
((f)=(Exp[(q)]))
((l)=((x)^3))
```

Dopo avere valutato questo elenco di espressioni e partendo da una memoria completamente vuota, l'esecuzione del metodo `dump` descritto nella sezione precedente dovrebbe scrivere su un file di testo il seguente contenuto:

```
x=(0.0)
l=((0.0)^3)
h=((y))
r=((e)+(1.0))
f=(Exp[((1.0))])
q=((1.0))
z=((0.0)+(y))
```

5 – Modalità di consegna

Il progetto può essere svolto al massimo da tre persone che intendono sostenere l'intero esame di Fondamenti di Architettura e Programmazione - Laboratorio di Programmazione negli appelli di Settembre 2006 o Gennaio 2007, e deve essere consegnato **entro la mezzanotte tra domenica 17 e lunedì 18 settembre 2007**, inviando un messaggio di posta elettronica al proprio docente (Dario Malchiodi per il turno 1 e Walter Cazzola per il turno 2: gli indirizzi sono indicati in calce a questo documento) contenente: 1) tutti i sorgenti che permettono di compilare ed eseguire correttamente il progetto, allegati al messaggio (eventualmente all'interno di un archivio ZIP; altri formati di compressione non saranno accettati); 2) l'indicazione dell'appello (settembre 2006 o gennaio 2007) in cui ogni singolo componente del gruppo intende discutere il progetto. Nel caso il progetto venga svolto da più di una persona, dovrà essere fatta in ogni caso **una sola sottoposizione**, indicando chiaramente in un commento all'inizio dei sorgenti consegnati nome, cognome e matricola dei vari componenti del gruppo.

Non saranno accettate le sottoposizioni i cui sorgenti contengano errori rilevati in fase di compilazione o durante l'esecuzione dei casi paradigmatici descritti nella sezione precedente.

E' inoltre richiesto di consegnare, **entro martedì 19 settembre 2006**, una copia cartacea della stampa del codice sorgente prodotto in portineria del DSI o nella casella di posta del docente, indicando chiaramente nome, cognome e numero di matricola di tutti i componenti del gruppo, nonché il turno e il docente di riferimento, unitamente a un **breve** documento che descriva il modo in cui interfacciarsi con il programma e illustri le principali scelte implementative e le strategie utilizzate per svolgere il progetto. Nel caso si voglia consegnare questo documento **anche** tramite il sito di sottoposizione, sarà necessario utilizzare **un formato non proprietario** (**sono accettabili pdf, rtf e txt**).

E' richiesto di indicare chiaramente all'inizio di tutti i documenti consegnati nome, cognome e matricola dei vari componenti del gruppo.

6 – Valutazione

La discussione con i singoli studenti verterà sulle modalità di implementazione adottate e sulla padronanza di alcuni dei concetti necessari per preparare il progetto e/o spiegati a lezione. La valutazione del progetto sarà fatta in base alla

- conformità dell'implementazione scelta per risolvere il problema con il paradigma di programmazione a oggetti;
- conformità del codice presentato alle regole di buona programmazione;
- adeguatezza del manuale utente presentato a descrivere il modo in cui un utente può utilizzare il programma;
- assenza di errori nel programma;

Dario Malchiodi
Dipartimento di Scienze dell'Informazione
Via Comelico 39/41 20135 Milano
Stanza T304 – Tel. +39 02 503 16338
eMail malchiodi@dsi.unimi.it

Walter Cazzola
Dipartimento di Informatica e Comunicazione
Via Comelico 39/41 20135 Milano
Stanza S233 – Tel. +39 0103536637
eMail cazzola@dico.unimi.it