Appelli di Giugno e Luglio 2006	**MyAssembler**
	Laurea Triennale in Comunicazione Digitale Laboratorio di Programmazione

1 Descrizione

Il progetto consiste nell'implementare in Java un'applicazione che preso un programma scritto in un semplice linguaggio di programmazione simil-Java semplificato (solo variabili intere inizializzate a zero di default) lo valuta e lo traduce in assembler.

2 Il Linguaggio Assembler Considerato

La macchina virtuale che prendiamo in considerazione è basata sull'uso dello stack[1]. Tutte le operazioni lavorano sullo stack usando come operandi l'elemento sul top dello stack (in caso di operatori unari) ed eventualmente i due elementi sul top dello stack (in caso di operatori binari).

Le istruzioni che abbiamo a disposizione sono:

```
LOAD   STORE   DECLARE   PRINT   JUMP   JZERO   EQN
AND    OR      NOT       GT      ADD    SUB
```

Ogni istruzione può essere etichettata e le etichette possono essere usate dalle istruzioni di salto per saltare all'istruzione corrispondente. Le etichette hanno tutte forma `Ln` dove n è un numero progressivo (non possono mancare etichette). Quando preposta ad un'istruzione, l'etichetta è seguita da un '`:`', ad es. `L1:`.

Istruzioni di Salto: JUMP e JZERO

Abbiamo due istruzioni di salto:

JUMP ≪*etichetta*≫ **JZERO** ≪*etichetta*≫

[1] Lo stack è una struttura dati, simile all'array, composta da elementi di tipo omogeneo (nel nostro caso interi) che possono essere aggiunti e tolti solo a partire dal primo elemento detto *top*.

rispettivamente per il salto incondizionato e per quello condizionato; l'istruzione di salto condizionato confronta il top dello stack con zero e se uguale salta all'istruzione identificata dall'etichetta. Dopo un'istruzione **JZERO** il top dello stack viene rimosso.

Gestione della Memoria: DECLARE, LOAD e STORE

L'istruzione:

DECLARE ≪*nome*≫

riserva dello spazio di memoria inizializzato a zero per la variabile intera indicata da ≪*nome*≫.

LOAD ≪*nome*≫ **LOAD** ≪*costante*≫ **STORE** ≪*nome*≫

Le prime due istruzioni permettono di caricare il contenuto della variabile ≪*nome*≫ o la costante ≪*costante*≫ sul top dello stack; mentre la terza permette di salvare (rimuovendolo) il contenuto del top dello stack nella variabile identificata da ≪*nome*≫.

Operazioni Aritmetiche e Booleane

Tutte le operazioni (sia aritmetiche che booleane) lavorano sullo stack usando come operandi l'elemento sul top dello stack (in caso di operatori unari) ed eventualmente i due elementi sul top dello stack (in caso di operatori binari).

ADD **SUB** **AND** **OR** **GT** **EQN**

Eseguono l'operazione intesa dal nome recuperando e quindi rimuovendo gli operandi dallo stack e lasciando il risultato sul top dello stesso. **GT** (Greater Than) corrisponde al maggiore e **EQN** corrisponde al test di uguaglianza.

NOT è l'unica istruzione unaria e quindi opera solo sul top dello stack che verrà sostituito dal risultato della negazione logica.

Operazioni di Stampa

Abbiamo due versioni dell'istruzione di stampa:

PRINT **PRINT** ≪*messaggio*≫

La prima permette di stampare il valore contenuto sul top dello stack (dopo il top viene rimosso), la seconda di stampare il messaggio (una stringa) passatole come parametro.

Notare che la traduzione più semplice (e richiesta) dal linguaggio considerato all'assembler considerato è unica.

3 Classi da Realizzare

È richiesto di risolvere il progetto descritto nella sezione precedente realizzando in Java le seguenti classi:

Statement

La classe `Statement` descrive una generica istruzione del linguaggio. La classe dovrà contenere:

- la definizione della `Hashtable state`, deputata a contenere i valori memorizzati nelle variabili che possono essere repecuperati usando come chiave di ricerca i nomi delle stesse; `state` dovrà essere essere **condivisa** e **accessibile** da tutte le istanze delle classi derivate da `Statement` e da `Expression` (vedi dopo);
- la definizione del metodo astratto `void run()`, da chiamare per valutare l'istruzione; durante l'esecuzione di tale metodo potrà essere lanciata l'eccezione `ExecutionException`, rappresentante un generico errore di esecuzione;
- la definizione del metodo astratto `Object clone()` da chiamare per restituire una copia dell'istruzione corrente;
- la definizione del metodo astratto `String toString()`, la cui esecuzione ritorna una descrizione testuale dell codice assembler che realizza l'istruzione (**Nota** un'istruzione ad alto livello potrebbe essere tradotta in diverse istruzioni assembler). La rappresentazione del codice assembler è da intendersi un'istruzione per riga. Inoltre il codice assembler viene scritto a partire dalla colonna 5.

Expression

La classe `Expression` descrive una generica espressione del linguaggio (espressioni aritmetiche, booleane[2] o lettura variabili). La classe dovrà contenere:

- la definizione della `Hashtable state`, deputata a contenere i valori memorizzati nelle variabili che possono essere repecuperati usando come chiave di ricerca i nomi delle stesse; `state` dovrà essere essere **condivisa** e **accessibile** da tutte le istanze delle classi derivate da `Expression` e da `Statement` (vedi prima);
- la definizione del metodo astratto `int eval()`, da chiamare per valutare l'espressione; durante l'esecuzione di tale metodo potrà essere lanciata l'eccezione `ExecutionException`, rappresentante un generico errore di esecuzione;
- la definizione del metodo astratto `Object clone()` da chiamare per restituire una copia dell'espressione corrente;
- la definizione del metodo astratto `String toString()`, la cui esecuzione ritorna una descrizione testuale del codice assembler che realizza l'espressione rappresentata (**Nota** un'espressione può essere tradotta in diverse istruzioni assembler). La rappresentazione del codice assembler è da intendersi un'istruzione per riga. Inoltre il codice assembler viene scritto a partire dalla colonna 5.

[2] Per semplicità le espressioni booleane ritorneranno 1 per vero e 0 per falso.

Sequence

La classe `Sequence` è derivata da `Statement` e descrive una generica sequenza di istruzioni. La classe dovrà contenere:

- la variabile di istanza `Statement list[]`, deputata a contenere l'insieme delle istruzioni da valutare in sequenza;
- il costruttore **`public`** `Sequence(Statement s[])`, che inizializza la variabile di istanza della classe copiando in essa i contenuti dell'array passato come argomento;
- l'implementazione del metodo `run()` della superclasse dovrà essere fatta in modo da eseguire in sequenza tutte le istruzioni contenute nell'array `list`, dalla prima all'ultima posizione;
- l'implementazione del metodo `toString()` della superclasse dovrà essere fatta in modo da concatenare la descrizione testuale della codifica in assembler di tutte le istruzioni contenute in `list` nell'ordine in cui verranno eseguite ed una per riga.

Selection

La classe `Selection` è derivata da `Statement` e descrive l'esecuzione condizionata di un'istruzione sulla base dei contenuti di una o più variabili. La classe dovrà contenere:

- la variabile di istanza `Expression condition` che contiene un'espressione che verrà valutata in 0 o in 1;
- le variabili di istanza `Statement ifStatement` e `elseStatement` che identificano due istruzioni con l'ovvio significato;
- il costruttore

  ```
  public Selection(Expression c, Statement ts, Statement fs)
  ```

  che inizializza le variabili di istanza della classe copiando in essa le istruzioni e l'espressione passate come argomento, tenendo conto del fatto che l'ordine degli argomenti equivale all'ordine in cui sono state presentate le variabili di istanza;
- l'implementazione del metodo `run()` della superclasse dovrà valutare l'espressione booleana contenuta in `condition`; se la valutazione dà come risultato 1 (vero) verrà eseguita l'istruzione identificata da `ifStatement`, mentre quando la condizione è valutata in 0 (falso) verrà eseguita l'istruzione identificata da `elseStatement`;
- l'implementazione del metodo `toString()` dovrà ritornare una stringa che contenga: la codifica dell'espressione `condition` valutata sul top dello stack, seguita da una istruzione di salto condizionato alla codifica di `elseStatement`

se il top dello stack contiene uno zero, altrimenti prosegue con la codifica di `ifStatement` terminata da un'istruzione di salto incondizionato alla fine della codifica dell'istruzione `elseStatement`; attenzione alle etichette dei salti, il loro valore deve essere univoco per tutto il programma e generato in progressione, l'etichette sono inserite a partire dalla colonna 1.

Loop

La classe `Loop` estende `Statement` e descrive l'esecuzione ripetuta di un'istruzione sulla base di un'espressione booleana. La classe dovrà contenere:

- la variabile di istanza `Expression condition` che contiene un'espressione che verrà valutata in 0 o in 1;
- la variabile di istanza `Statement body` che identifica l'istruzione da eseguire;
- il costruttore **`public`** `Loop(Expression c, Statement s)` che inizializza le variabili di istanza della classe copiando in esse l'istruzione e l'espressione passate come argomento;
- l'implementazione del metodo `run()` della superclasse dovrà eseguire l'istruzione identificata da `body` fintantoché la valutazione dell'espressione `condition` da 1 (vero);
- l'implementazione del metodo `toString()` dovrà ritornare una stringa che contenga: la codifica dell'espressione `condition` valutata sul top dello stack, seguita da una istruzione di salto condizionato a dopo la codifica dell'istruzione `body`, seguita quindi dalla codifica di `body` terminata da un'istruzione di salto incondizionato all'inizio della codifica dell'espressione `condition`; attenzione alle etichette dei salti, il loro valore deve essere univoco per tutto il programma e generato in progressione, l'etichette sono inserite a partire dalla colonna 1.

Declare

La classe `Declare` estende `Statement` e descrive la dichiarazione di una variabile (intera). La classe dovrà contenere:

- la variabile di istanza `String name` che contiene il nome della variabile rappresentata;
- il costruttore **`public`** `Declare(String v)` che inizializza la variabile di istanza della classe copiando in essa il valore passato come argomento;
- l'implementazione del metodo `run()` della superclasse dovrà inserire nella hashtable della superclasse una variabile avente nome uguale a quello contenuto in `name` e inizializzata a 0; se la variabile che si sta dichiarando è già stata dichiarata in precedenza, il metodo solleverà l'eccezione `ExistingVariableException`;
- l'implementazione del metodo `toString()` dovrà ritornare una stringa contenente la codifica assembler della dichiarazione della variabile.

Assign

La classe `Assign` estende `Statement` e descrive l'assegnamento del valore di una espressione ad una variabile. La classe dovrà contenere:

- la variabile di istanza `String name` che contiene il nome di una variabile;
- la variabile di istanza `Expression expr` che contiene un'espressione il cui valore verrà assegnato alla variabile indicata da `name`;
- il costruttore **`public`** `Assign(String var, Expression e)`, che inizializza le variabili di istanza della classe copiando in esse i valori passati come argomento;
- l'implementazione del metodo `run()` della superclasse dovrà assegnare il valore ottenuto valutando l'espressione `expr` alla variabile il cui nome è identificato da `name`, cancellandone il contenuto precedente; il metodo dovrà lanciare l'eccezione `NotDeclaredVariableException` qualora `name` non identifichi una variabile precedentemente dichiarata;
- l'implementazione del metodo `toString()` dovrà ritornare una stringa contenente la codifica dell'espressione `expr` valutata sul top dello stack seguita dall'assegnamento del top dello stack alla variabile identificata da `name`.

PrintExpression

La classe `PrintExpression` estende `Statement` e descrive un'istruzione per stampare il valore calcolato di un'espressione. La classe dovrà contenere:

- la variabile di istanza `Expression toBeEvaluated` che contiene l'espressione da calcolare e stampare;
- il costruttore **`public`** `PrintExpression(Expression e)`, che inizializza le variabili di istanza della classe copiando in esse i valori passati come argomento;
- l'implementazione del metodo `run()` della superclasse dovrà valutare l'espressione `toBeEvaluated` e stamparne il valore;
- l'implementazione del metodo `toString()` dovrà ritornare una stringa contenente il codice assembler necessario per valutare l'espressione sul top dello stack e stamparlo.

PrintString

La classe `PrintString` estende `Statement` e descrive l'output a video di un generico messaggio. La classe dovrà contenere:

- la variabile di istanza `String msg` che identifica il messaggio da visualizzare;
- il costruttore **`public`** `PrintString(String msg)` che inizializza la variabile di istanza della classe copiando in essa il valore passato come argomento;

- l'implementazione del metodo `run()` della superclasse dovrà stampare a video la stringa contenuta nella variabile di istanza;
- l'implementazione del metodo `toString()` dovrà ritornare una stringa contenente il codice assembler necessario per stampare il messaggio da visualizzare (delimitato da doppi apici).

Espressioni Booleane

La classe `And` estende `Expression` e descrive l'operazione di and logico tra due sotto espressioni. Notare che le espressioni booleane in realtà sono operazioni aritmetiche il cui risultato è 0 (falso) o 1 (vero), pertanto l'operazione di and logico sarà realizzata dalla moltiplicazione. La classe dovrà contenere:

- le variabili di istanza `Expression firstOp` e `Expression secondOp` che contengono le due sotto espressioni da confrontare;
- il costruttore **`public`** `And(Expression f, Expression s)`, che inizializza le variabili di istanza della classe copiando in esse i valori passati come argomento nell'ordine in cui sono state presentate;
- l'implementazione del metodo `eval()` della superclasse dovrà valutare le due sotto espressioni e ritornare l'and logico delle due;
- l'implementazione del metodo `toString()` dovrà ritornare una stringa contenente il codice assembler che valuta l'espressione `firstOp` sullo stack, seguito dal codice che valuta l'espressione `secondOp` sempre sullo stack, seguito dal codice per calcolare sullo stack l'and logico di top e top-1.

In maniera analoga si devono realizzare le classi `Or`, `Not`, `Equal` e `GreaterThan` dall'ovvio comportamento. Notare che l'implementazione della classe `Not` differisce leggermente dalle altre perché il not è un operatore unario e quindi avrà un solo operando (ed una sola variabile di istanza, chiamata `op`).

Espressioni Aritmetiche

La classe `Add` estende `Expression` e descrive l'operazione di addizione tra due sotto espressioni. La classe dovrà contenere:

- le variabili di istanza `Expression firstOp` e `Expression secondOp` che contengono le due sotto espressioni da sommare;
- il costruttore **`public`** `Add(Expression f, Expression s)`, che inizializza le variabili di istanza della classe copiando in esse i valori passati come argomento nell'ordine in cui sono state presentate;
- l'implementazione del metodo `eval()` della superclasse dovrà valutare le due sotto espressioni e ritornare la loro somma;

- l'implementazione del metodo `toString()` dovrà ritornare una stringa contenente il codice assembler che valuta l'espressione `firstOp` sullo stack, seguito dal codice che valuta l'espressione `secondOp` sempre sullo stack, seguito dal codice per sommare sullo stack il top e il top-1.

In maniera analoga si deve realizzare la classe `Subtract` dall'ovvio comportamento.

Constant

La classe `Constant` estende `Expression` e descrive una costante numerica. La classe dovrà contenere:

- la variabile di istanza `int number` che contiene il valore della costante;
- il costruttore `public Constant(int n)`, che inizializza le variabili di istanza della classe copiando in esse i valori passati come argomento;
- l'implementazione del metodo `eval()` della superclasse dovrà ritornare il valore della costante;
- l'implementazione del metodo `toString()` dovrà ritornare una stringa contenente il codice assembler necessario per mettere la costante sul top dello stack.

ReadVariable

La classe `ReadVariable` estende `Expression` e permette di leggere il contenuto di una variabile. La classe dovrà contenere:

- la variabile di istanza `String name` che contiene il nome della variabile da leggere;
- il costruttore `public ReadVariable(String n)`, che inizializza le variabili di istanza della classe copiando in esse i valori passati come argomento;
- l'implementazione del metodo `eval()` della superclasse dovrà ritornare il valore recuperato in `state` per la variabile indicata da `name`; il metodo solleverà l'eccezione `NotDeclaredVariableException` quando la variabile non è stata inizializzata;
- l'implementazione del metodo `toString()` dovrà ritornare una stringa contenente il codice assembler necessario per mettere il contenuto della variabile sul top dello stack.

Inoltre, andranno implementate le classi `ExecutionException`, `ExistingVariableException` e `NotDeclaredVariableException` definite in modo opportuno.

A parte quanto espressamente richiesto, è lasciata piena libertà sull'implementazione delle singole classi e sull'eventuale introduzione di altre classi, a patto di seguire le regole del paradigma ad oggetti ed i principi di buona programmazione.

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

È invece **espressamente richiesto** di non utilizzare package non standard di Java (si possono quindi utilizzare `java.util`, `java.io` e così via), con l'unica eccezione del package `prog.io` incluso nel libro di testo per gestire l'input da tastiera e l'output a video e di rispettare l'interfaccia fornita.

4 Program.java

La classe `Program` è la classe principale ed avrà un attributo `Hashtable state` che sarà condiviso da tutte le espressioni e le istruzioni.

Inoltre, non verranno presi in considerazione progetti le cui classi non permetteranno almeno di compilare il seguente sorgente:

```
import java.util.*;

public class Program {
   public static Hashtable state = new Hashtable();

   public static void main(String args[]) {
     try {
        Declare d0 = new Declare("a");
        Declare d1 = new Declare("b");
        ReadVariable ra = new ReadVariable("a");
        ReadVariable rb = new ReadVariable("b");
        Constant zero = new Constant(0);
        Constant seven = new Constant(7);
        Assign a0 = new Assign("a", seven);
        Assign a1 = new Assign("b", new Constant(-1));

        PrintString msg1 = new PrintString("a must be greater than zero.");
        PrintString msg2 = new PrintString("b must be greater than zero.");

        Loop l1 = new Loop(
                    new Or(
                      new Equal(rb, zero),
                      new GreaterThan(rb, zero)
                    ), a1);

        Statement sts[] = new Statement[4];
        sts[0] = d0; sts[1] = d1; sts[2] = a0;

        Sequence ss0 = new Sequence(new Statement[]{msg2, new Assign("b", seven)});
        Selection s1 = new Selection(new GreaterThan(rb, zero), l1, ss0);

        Declare d2 = new Declare("ab");
        a1 = new Assign("ab", ra);
        Assign a3 = new Assign("ab", new Add(new ReadVariable("ab"), ra));
        Assign a2 = new Assign("b", new Subtract(rb, new Constant(1)));
        l1 = new Loop(
```

```
            new GreaterThan(rb, zero),
            new Sequence(new Statement[]{a3, a2})
            );
       Sequence ss1 =
           new Sequence(
            new Statement[]{
               new Assign("b", new Constant(-1)),
               s1,
               d2,
               a1,
               l1,
               new PrintExpression(new ReadVariable("ab"))
            }
           );
       sts[3] = new Selection(new Not(new GreaterThan(ra, zero)), msg1, ss1);
       ss1 = new Sequence(sts);
       System.out.println(ss1);
       ss1.run();
    } catch(ExecutionException e) {e.printStackTrace();}
  }
}
```

Naturalmente la discussione del progetto verterà anche sull'analisi dell'algoritmo codificato in `Program.java` e sul peculiare uso delle variabili in esso contenute.

5 Modalità di Consegna

Il progetto deve essere svolto a gruppi di al massimo tre persone che intendono sostenere l'intero esame di Fondamenti di Architettura e Programmazione - Laboratorio di Programmazione negli appelli di Giugno o Luglio 2006, e deve essere consegnato entro mezzanotte di lunedì 5 giugno 2006 via e-mail al docente di laboratorio del vostro turno (cioè a Dario Malchiodi per il turno 1 e a Walter Cazzola per il turno 2), gli indirizzi e-mail sono in calce al documento.

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.

Dovranno essere consegnati tutti i sorgenti Java che permettano al programma di essere eseguito correttamente, compressi in un archivio di tipo ZIP che estragga i file nella directory in cui si trova l'archivio stesso. Nell'archivio dovrà anche essere accluso un breve documento in formato txt o rtf in cui:

- verrà descritto il modo in cui interfacciarsi con il programma;
- saranno illustrate le principali scelte implementative e le strategie utilizzate per svolgere il progetto

Le sottoposizioni che non seguono queste specifiche ed i programmi che non compilano correttamente non verranno valutati.

È inoltre richiesto di consegnare una copia cartacea della stampa del codice sorgente prodotto in portineria del DSI/DICo indicando chiaramente nome, cognome e numero di matricola di tutti i componenti del gruppo, nonché il turno e il docente di

riferimento. Nella copia cartacea indicare anche in quale appello **il gruppo** intende discutere il progetto.

6 Valutazione

Durante la prova orale con i singoli studenti saranno discusse le modalità implementative adottate e la 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;

Walter Cazzola
Dipartimento di Informatica e Comunicazione
Via Comelico 39/41 20135 Milano
Stanza S233 — Tel. +39.010.353.6637
e-mail: `cazzola@dico.unimi.it`

Dario Malchiodi
Dipartimento di Scienze dell'Informazione
Via Comelico 39/41 20135 Milano
Stanza S238 — Tel. +39.02.503.16338
e-mail: `malchiodi@dsi.unimi.it`