Laboratorio di Linguaggi Formali e Traduttori
LFT lab T4, a.a. 2022/2023

Docente: Luigi Di Caro

Generazione del bytecode

Questa lezione (n.8): esercizio principale

Prossime due lezioni: esercizi facoltativi

Questa lezione (n.8): esercizio principale

- Traduzione completa di un programma, secondo una grammatica

Questa lezione (n.8): esercizio principale

- Traduzione completa di un programma, secondo una grammatica
  - (dovete prima progettare una SDT on the fly)

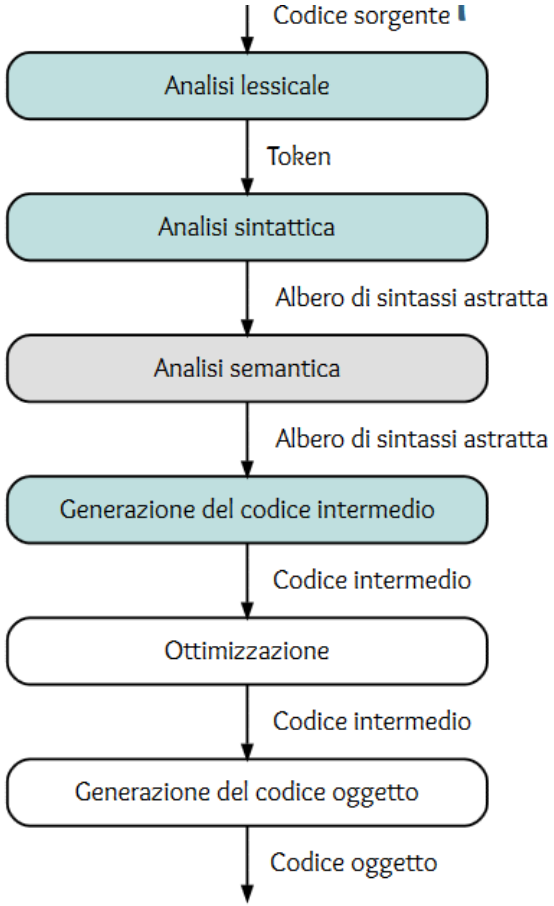
SDT "on-the-fly" per la grammatica LL(1)

Produzioni	Azioni semantiche
$E \to TE'$	
$E' \to \varepsilon$	
$E' \to +T\ E'$	$\{emit(\texttt{iadd})\}$
$E' \to -T\ E'$	$\{emit(\texttt{isub})\}$
$T \to FT'$	
$T' \to \varepsilon$	
$T' \to *F\ T'$	$\{emit(\texttt{imul})\}$
...	
$F \to \texttt{n}$	$\{emit(\texttt{ldc n.}v)\}$
$F \to x$	$\{emit(\texttt{iload \&}x)\}$
$F \to (E)$	

```
private void E'() {
  switch (peek()) {
  case '+': // E' → +TE'
    match('+');
    T();
    emit("iadd");
    E'();
    break;
  case '-': // E' → -TE'
    match('-');
    T();
    emit("isub");
    E'();
    break;
  case ')':
  case '$': // E' → ε
    break;
  default:
    throw error("E'");
  }
}
```

Generazione codice intermedio

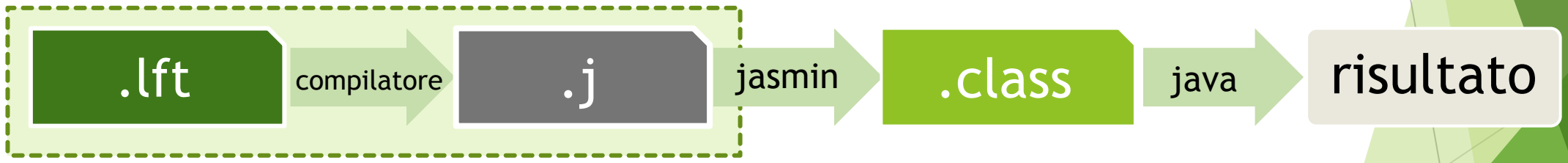
- Generazione codice intermedio:
  - Fase successiva a quelle dell'analisi lessicale e dell'analisi sintattica (l'analisi semantica non è stato affrontato in questo corso).
  - Traduzione di un programma di un linguaggio (sorgente) a un altro (oggetto).
  - Nostro caso:
    - Sorgente: il linguaggio dell'esercizio 3.2.
    - Oggetto: bytecode per la JVM.



Generazione del bytecode

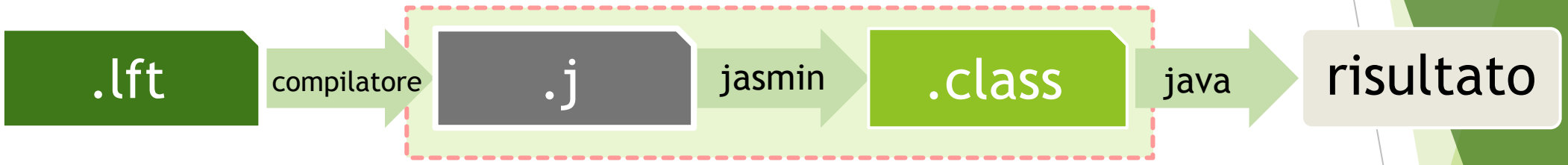
- Utilizzo tipico del JVM:

- Obiettivo: realizzare un compilatore per il linguaggio `P` dove il linguaggio oggetto è bytecode JVM in formato mnemonico.



- File .j: bytecode JVM in formato *mnemonico*.
- File .class: bytecode JVM in formato *binario*.
- Jasmin: programma assembler per tradurre il bytecode dal formato mnemonico al formato binario.

Generazione del bytecode



- jasmin.jar può essere scaricato dalla pagina Moodle/I-learn del laboratorio.
- Per eseguire Jasmin (con il file Output.j come input a jasmin):

```
java -jar jasmin.jar Output.j
```

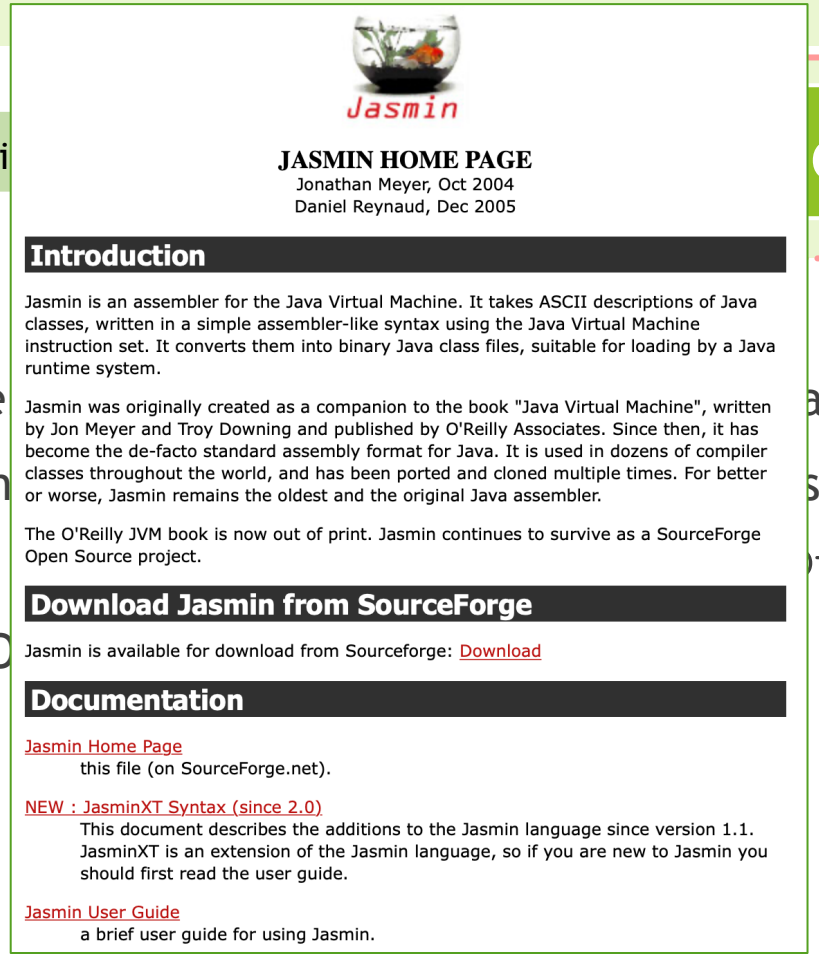
- Jasmin crea il file Output.class.

Generazione del bytecode

.lft

compi

- jasmin.jar può esse
- Per eseguire Jasmin
- Jasmin crea il file O



Jasmin

JASMIN HOME PAGE

Jonathan Meyer, Oct 2004
Daniel Reynaud, Dec 2005

Introduction

Jasmin is an assembler for the Java Virtual Machine. It takes ASCII descriptions of Java classes, written in a simple assembler-like syntax using the Java Virtual Machine instruction set. It converts them into binary Java class files, suitable for loading by a Java runtime system.

Jasmin was originally created as a companion to the book "Java Virtual Machine", written by Jon Meyer and Troy Downing and published by O'Reilly Associates. Since then, it has become the de-facto standard assembly format for Java. It is used in dozens of compiler classes throughout the world, and has been ported and cloned multiple times. For better or worse, Jasmin remains the oldest and the original Java assembler.

The O'Reilly JVM book is now out of print. Jasmin continues to survive as a SourceForge Open Source project.

Download Jasmin from SourceForge

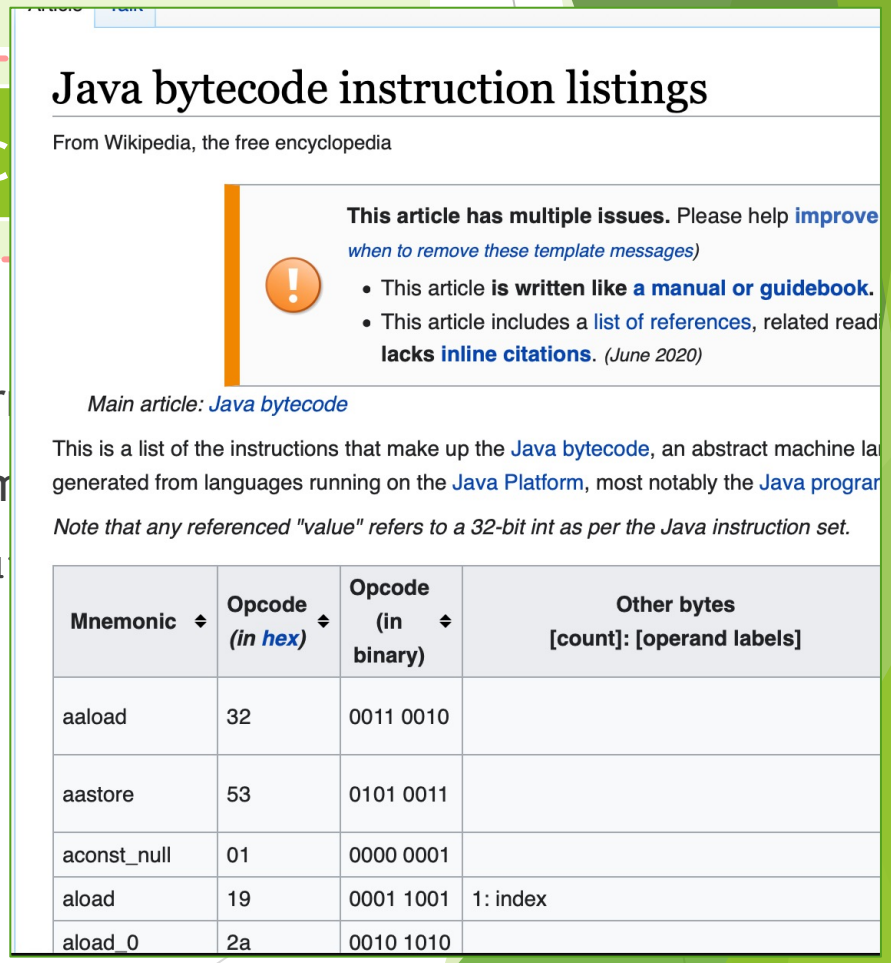
Jasmin is available for download from Sourceforge: Download

Documentation

Jasmin Home Page
this file (on SourceForge.net).

NEW : JasminXT Syntax (since 2.0)
This document describes the additions to the Jasmin language since version 1.1. JasminXT is an extension of the Jasmin language, so if you are new to Jasmin you should first read the user guide.

Jasmin User Guide
a brief user guide for using Jasmin.



Java bytecode instruction listings

From Wikipedia, the free encyclopedia

This article has multiple issues. Please help improve
when to remove these template messages)
- This article **is written like a manual or guidebook.**
- This article includes a list of references, related readi
  lacks inline citations. *(June 2020)*

Main article: Java bytecode

This is a list of the instructions that make up the Java bytecode, an abstract machine la
generated from languages running on the Java Platform, most notably the Java prograr

Note that any referenced "value" refers to a 32-bit int as per the Java instruction set.

Mnemonic	Opcode *(in hex)*	Opcode (in binary)	Other bytes [count]: [operand labels]
aaload	32	0011 0010	
aastore	53	0101 0011	
aconst_null	01	0000 0001	
aload	19	0001 1001	1: index
aload_0	2a	0010 1010	

Comandi del linguaggio

Comando	Significato	Esempio del comando
`assign` *<expr>* `to` *<idlist>*	Assegnamento del valore di un'espressione a uno o più identificatori	`assign 3 to x,y`
`print[`*<exprlist>*`]`	Stampare sul terminale i valori di un elenco di espressioni	`print[+(x,3),4]`
`read[`*<idlist>*`]`	Legge un o più input dalla tastiera	`read[x,y]`

`while(`*<bexpr>*`)` *<stat>*	Ciclo: se la condizione booleana *<bexpr>* è vera, eseguire un comando, poi ripetere	`while (<> x 0)`
		`{read[x];`
		`print[x]}`

`{`*<statlist>*`}`	Composizione sequenziale: raggruppa un elenco di comandi	`{`
		`read[x];`
		`print[*(x,3)]`
		`}`

Comandi del linguaggio

Comando	Significato	Esempio del comando

conditional
[
option (<*bexpr*$_1$>) do <*stat*$_1$>
…
option (<*bexpr*$_n$>) do <*stat*$_n$>
]
end
	Un comando condizionale:
	se la prima espressione da <*bexpr*$_1$>…<*bexpr*$_n$> che risulta valutata come vera è <*bexpr*$_k$> allora <*stat*$_k$> è eseguito, si esce dall'istruzione `cond` e si procede alla prossima istruzione.
		conditional
		[
		option (> x 0) do {
		print[x];
		assign 3 to x
		}
		option (> y 0) do print[y]
		]
		end

conditional
[
option (<*bexpr*$_1$>) do <*stat*$_1$>
…
option (<*bexpr*$_n$>) do <*stat*$_n$>
]
else <*stat*> end
	Estende il comando precedente con il caso else/default: se nessuna espressione nell'elenco <*bexpr*$_1$>…<*bexpr*$_n$> è verificata, viene eseguito lo ⟨*stat*⟩ scritto dopo `else`.	conditional
		[
		option (> x 0) do {
		print[x];
		assign 3 to x
		}
		option (> y 0) do print[y]
		]
		else print[z]

Esempio di traduzione

Programma .lft

```
read[a];
print[+(a,1)]
```

Esempio di traduzione del programma .lft (frammento di Output.j)

```
 invokestatic Output/read()I
 istore 0
 goto L1
L1:
 iload 0
 ldc 1
 iadd
 invokestatic Output/print(I)V
 goto L2
L2:
 goto L0
L0:
```

Esempio di traduzione

```
 invokestatic Output/read()I
 istore 0
L1:
 iload 0
 ldc 1
 iadd
 invokestatic Output/print(I)V
L2:
L0:
```

leggo
memorizzo
etichetta L1
inserisco sulla pila elemento memorizzato
inserisco 1 sulla pila
inserisco somma sulla pila
stampo valore sulla pila
etichette L2 e L0

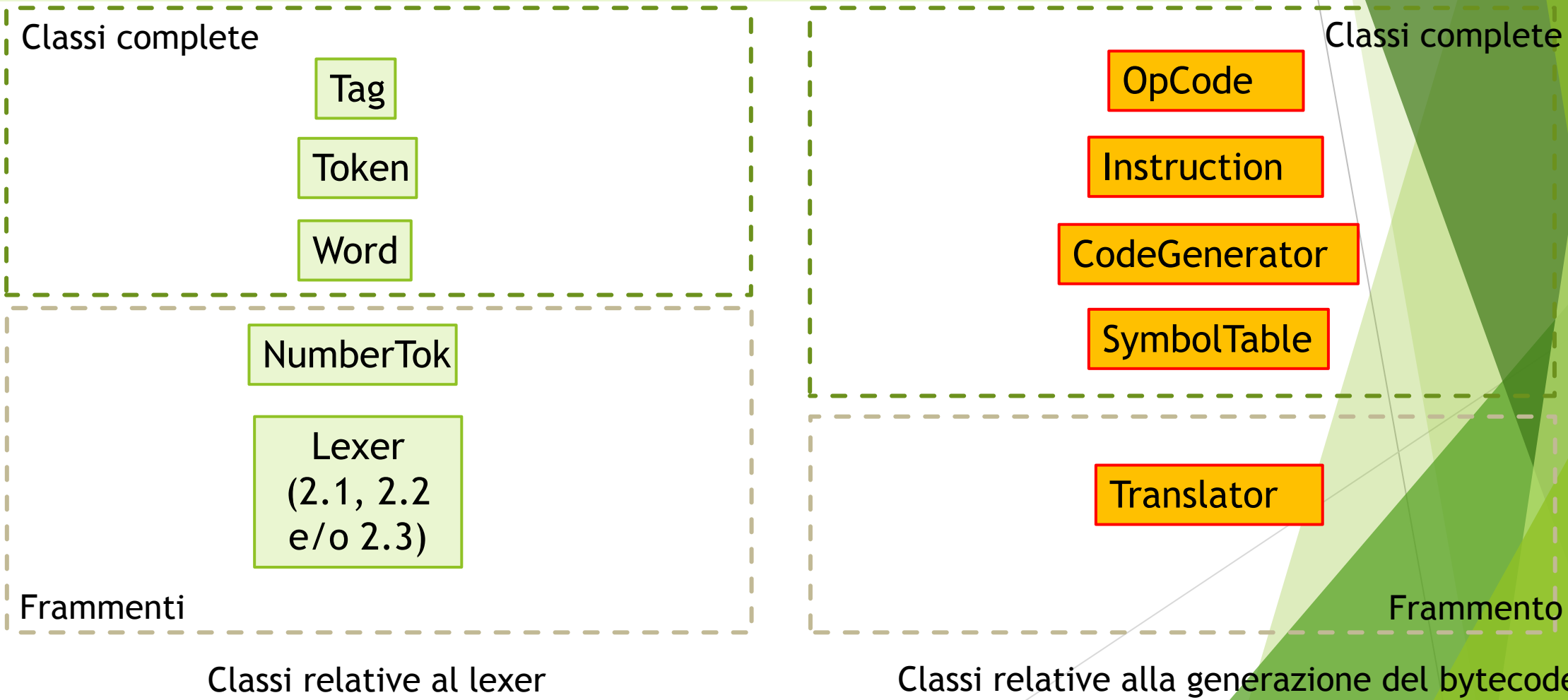
Esempio di traduzione

```
 invokestatic Output/read()I
 istore 0
L1:
 iload 0
 ldc 1
 iadd
 invokestatic Output/print(I)V
L2:
L0:
```

etichette:

- **uso soggettivo!** Non è un problema per l'esame (ma è inutile esagerare)!
- vedremo meglio dopo i vostri primi tentativi, a Gennaio!

Classi del generatore di bytecode



Classi di supporto

- OpCode: semplice enumerazione dei nomi mnemonici delle istruzioni del linguaggio oggetto.

```
public enum OpCode {
    ldc, imul, ineg, idiv, iadd,
    isub, istore, ior, iand, iload,
    if_icmpeq, if_icmple, if_icmplt, if_icmpne, if_icmpge,
    if_icmpgt, ifne, GOto, invokestatic, dup, pop, label }
```

Classi di supporto

- Instruction: verrà usata per rappresentare singole istruzioni del linguaggio mnemonico.
  - Il metodo `toJasmin` restituisce l'istruzione nel formato adeguato per l'assembler jasmin.
  - Per il caso invokestatic usiamo operando 1 per print e 0 per read

```
public class Instruction {
    OpCode opCode;
    int operand;

    public Instruction(OpCode opCode) {
        this.opCode = opCode;
    }

    public Instruction(OpCode opCode, int operand) {
        this.opCode = opCode;
        this.operand = operand;
    }

    public String toJasmin() {
        String temp="";
        switch (opCode) {
            case ldc : temp = " ldc " + operand + "\n"; break;
            case invokestatic :
               if( operand == 1)
                  temp = " invokestatic " + "Output/print(I)V" + "\n";
               else
                  temp = " invokestatic " + "Output/read()I" + "\n"; break;
            case iadd : temp = " iadd " + "\n"; break;
            case imul : temp = " imul " + "\n"; break;
            case
```

(non tutti i comandi hanno operandi)

...

Classi di supporto

- CodeGenerator: ha lo scopo di memorizzare in una struttura apposita la lista delle istruzioni (come oggetti di tipo Instruction) generate.

```
public class CodeGenerator {

    LinkedList <Instruction> instructions = new LinkedList <Instruction>();

    int label=0;

    public void emit(OpCode opCode) {
        instructions.add(new Instruction(opCode));
    }

    public void emit(OpCode opCode , int operand) {
        instructions.add(new Instruction(opCode, operand));
    }

    public void emitLabel(int operand) {
        emit(OpCode.label, operand);
    }

    public int newLabel() {
        return label++;
    }

    public void toJasmin() throws IOException{
        PrintWriter out = new PrintWriter(new FileWriter("Output.j"));
        String temp = "";
        temp = temp + header;
        while(instructions.size() > 0){
            Instruction tmp = instructions.remove();
            temp = temp + tmp.toJasmin();
        }
        temp = temp + footer;
        out.println(temp);
        out.flush();
        out.close();
```

...

Classi di supporto

- CodeGenerator: ha lo scopo di memorizzare in una struttura apposita la lista delle istruzioni (come oggetti di tipo Instruction) generate.
- I metodi `emit/emitLabel` sono usati per aggiungere istruzioni o etichette di salto nel codice.

```
public class CodeGenerator {

    LinkedList <Instruction> instructions = new LinkedList <Instruction>();

    int label=0;

    public void emit(OpCode opCode) {
        instructions.add(new Instruction(opCode));
    }

    public void emit(OpCode opCode , int operand) {
        instructions.add(new Instruction(opCode, operand));
    }

    public void emitLabel(int operand) {
        emit(OpCode.label, operand);
    }

    public int newLabel() {
        return label++;
    }

    public void toJasmin() throws IOException{
        PrintWriter out = new PrintWriter(new FileWriter("Output.j"));
        String temp = "";
        temp = temp + header;
        while(instructions.size() > 0){
            Instruction tmp = instructions.remove();
            temp = temp + tmp.toJasmin();
        }
        temp = temp + footer;
        out.println(temp);
        out.flush();
        out.close();
```

...

Classi di supporto

- CodeGenerator: ha lo scopo di memorizzare in una struttura apposita la lista delle istruzioni (come oggetti di tipo Instruction) generate.
- I metodi `emit/emitLabel` sono usati per aggiungere istruzioni o etichette di salto nel codice.
- Le costanti `header` e `footer` definiscono il preambolo e l'epilogo del codice generato dal traduttore per restituire, mediante il metodo `toJasmin`, un file la cui struttura risponde ai requisiti dell'assembler jasmin.

```
public class CodeGenerator {

    LinkedList <Instruction> instructions = new LinkedList <Instruction>();

    int label=0;

    public void emit(OpCode opCode) {
        instructions.add(new Instruction(opCode));
    }

    public void emit(OpCode opCode , int operand) {
        instructions.add(new Instruction(opCode, operand));
    }

    public void emitLabel(int operand) {
        emit(OpCode.label, operand);
    }

    public int newLabel() {
        return label++;
    }

    public void toJasmin() throws IOException{
        PrintWriter out = new PrintWriter(new FileWriter("Output.j"));
        String temp = "";
        temp = temp + header;
        while(instructions.size() > 0){
            Instruction tmp = instructions.remove();
            temp = temp + tmp.toJasmin();
        }
        temp = temp + footer;
        out.println(temp);
        out.flush();
        out.close();
```

...

header...

```
private static final String header = ".class public Output \n"
        + ".super java/lang/Object\n"
        + "\n"
        + ".method public <init>()V\n"
        + " aload_0\n"
        + " invokenonvirtual java/lang/Object/<init>()V\n"
        + " return\n"
        + ".end method\n"
        + "\n"
        + ".method public static print(I)V\n"
        + " .limit stack 2\n"
        + " getstatic java/lang/System/out Ljava/io/PrintStream;\n"
        + " iload_0 \n"
        + " invokestatic java/lang/Integer/toString(I)Ljava/lang/String;\n"
        + " invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V\n"
        + " return\n"
        + ".end method\n"
        + "\n"
        + ".method public static read()I\n"
        + " .limit stack 3\n"
        + " new java/util/Scanner\n"
        + " dup\n"
        + " getstatic java/lang/System/in Ljava/io/InputStream;\n"
        + " invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V\n"
        + " invokevirtual
```

Classi di supporto

- SymbolTable: tabella dei simboli; per tenere traccia degli identificatori.
  - indirizzi: usati come argomenti di comandi `iload` oppure `istore`

```
public class SymbolTable {

    Map <String, Integer> OffsetMap = new HashMap <String,Integer>();

    public void insert( String s, int address ) {
            if( !OffsetMap.containsValue(address) )
                OffsetMap.put(s,address);
            else
                throw new IllegalArgumentException("Riferimento ad una
                   locazione di memoria gia` occupata da un'altra variabile");
    }

    public int lookupAddress ( String s ) {
            if( OffsetMap.containsKey(s) )
                return OffsetMap.get(s);
            else
                return -1;
    }
}
```

Classi di supporto

- SymbolTable: tabella dei simboli; per tenere traccia degli identificatori.
  - Il metodo `insert` inserisce un nuovo elemento (coppia lessema/indirizzo) nella tabella, se non esiste già un elemento con lo stesso lessema nella tabella.

```
public class SymbolTable {

    Map <String, Integer> OffsetMap = new HashMap <String,Integer>();

    public void insert( String s, int address ) {
            if( !OffsetMap.containsValue(address) )
                OffsetMap.put(s,address);
            else
                throw new IllegalArgumentException("Riferimento ad una
                   locazione di memoria gia` occupata da un'altra variabile");
    }

    public int lookupAddress ( String s ) {
            if( OffsetMap.containsKey(s) )
                return OffsetMap.get(s);
            else
                return -1;
    }
}
```

Classi di supporto

- SymbolTable: tabella dei simboli; per tenere traccia degli identificatori.
  - Il metodo `insert` inserisce un nuovo elemento (coppia lessema/indirizzo) nella tabella, se non esiste già un elemento con lo stesso lessema nella tabella.
  - Dato un lessema, il metodo `lookupAddress` restituisce l'indirizzo del elemento della tabella che corrisponde al lessema (e resituisce -1 se non ci sono elementi della tabella che corrispondono al lessema).

```
public class SymbolTable {

    Map <String, Integer> OffsetMap = new HashMap <String,Integer>();

    public void insert( String s, int address ) {
            if( !OffsetMap.containsValue(address) )
                OffsetMap.put(s,address);
            else
                throw new IllegalArgumentException("Riferimento ad una
                   locazione di memoria gia` occupata da un'altra variabile");
    }

    public int lookupAddress ( String s ) {
            if( OffsetMap.containsKey(s) )
                return OffsetMap.get(s);
            else
                return -1;
    }
}
```

Classi di supporto

- SymbolTable: tabella dei simboli; per tenere traccia degli identificatori.
  - Il metodo `insert` inserisce un nuovo elemento (coppia lessema/indirizzo) nella tabella, se non esiste già un elemento con lo stesso lessema nella tabella.
  - Dato un lessema, il metodo `lookupAddress` restituisce l'indirizzo del elemento della tabella che corrisponde al lessema (e resituisce -1 se non ci sono elementi della tabella che corrispondono al lessema).

```
public class SymbolTable {

    Map <String, Integer> OffsetMap = new HashMap <String,Integer>();

    public void insert( String s, int address ) {
            if( !OffsetMap.containsValue(address) )
                OffsetMap.put(s,address);
            else
                throw new IllegalArgumentException("Riferimento ad una
                   locazione di memoria gia` occupata da un'altra variabile");
    }

    public int lookupAddress ( String s ) {
            if( OffsetMap.containsKey(s) )
                return OffsetMap.get(s);
            else
                return -1;
    }
}
```

Non necessariamente un errore: informazione utile!

Esercizio 5.1

- Si scriva un traduttore per i programmi scritti nel linguaggio `P` (dove la grammatica del linguaggio `P` è quello scritto nel testo dell'esercizio 3.2).
- Classe Translator: frammento di codice da completare (se ritenete opportuno, si può modificare il codice già scritto nel frammento).
- Scrivete un SDT "on-the-fly", ispirandovi agli esempi di SDT "on-the-fly" delle slide di teoria.

Classe Translator

<*prog*> ::= *{statlist.next=newlabel()}* <*statlist*> *{emitlabel(statlist.next)}* *EOF*

- Metodo `prog`:
  - Prima azione da fare: creare una nuova etichetta (*statlist.next* nel SDT, `lnext_prog` nel codice).
  - Il valore della etichetta è assegnata ad un attributo ereditato associato con (il nodo nell'albero sintattico di) <*statlist*>.
  - Dopo <*statlist*>, l'etichetta *statlist.next*/`lnext_prog` è emessa.
  - Dopo il controllo del terminale `EOF`, `code.toJasmin()` è chiamato, per creare il file Output.j.

```
public void prog() {
    // ... completare ...
    int lnext_prog = code.newLabel();
    statlist(lnext_prog);
    code.emitLabel(lnext_prog);
    match(Tag.EOF);
    try {
            code.toJasmin();
    }
    catch(java.io.IOException e) {
            System.out.println("IO error\n");
    };
    // ... completare ...
}
```

Insiemi guida...

Classe Translator

<prog> ::= {statlist.next=newlabel()} <statlist> {emitlabel(statlist.next)} EOF

- Metodo `prog`:
  - Prima azione da fare: creare una nuova etichetta (*statlist.next* nel SDT, `lnext_prog` nel codice).
  - Il valore della etichetta è assegnata ad un attributo ereditato associato con (il nodo nell'albero sintattico di) *<statlist>*.
  - Dopo *<statlist>*, l'etichetta *statlist.next*/`lnext_prog` è emessa.
  - Dopo il controllo del terminale `EOF`, `code.toJasmin()` è chiamato, per creare il file Output.j.

```
public void prog() {
    // ... completare ...
    int lnext_prog = code.newLabel();
    statlist(lnext_prog);
    code.emitLabel(lnext_prog);
    match(Tag.EOF);
    try {
            code.toJasmin();
    }
    catch(java.io.IOException e) {
            System.out.println("IO error\n");
    };
    // ... completare ...
}
```

Classe Translator

<prog> ::= {statlist.next=newlabel()} <statlist> {emitlabel(statlist.next)} EOF

- Metodo `prog`:
  - Prima azione da fare: creare una nuova etichetta (*statlist.next* nel SDT, `lnext_prog` nel codice).
  - Il valore della etichetta è assegnata ad un attributo ereditato associato con (il nodo nell'albero sintattico di) *<statlist>*.
  - Dopo *<statlist>*, l'etichetta *statlist.next*/`lnext_prog` è emessa.
  - Dopo il controllo del terminale `EOF`, `code.toJasmin()` è chiamato, per creare il file Output.j.

```
public void prog() {
    // ... completare ...
    int lnext_prog = code.newLabel();
    statlist(lnext_prog);
    code.emitLabel(lnext_prog);
    match(Tag.EOF);
    try {
            code.toJasmin();
    }
    catch(java.io.IOException e) {
            System.out.println("IO error\n");
    };
    // ... completare ...
}
```

Classe Translator

<*prog*> ::= {*statlist.next=newlabel()*} <*statlist*> {*emitlabel(statlist.next)*} *EOF*

- Metodo `prog`:
  - Prima azione da fare: creare una nuova etichetta (*statlist.next* nel SDT, `lnext_prog` nel codice).
  - Il valore della etichetta è assegnata ad un attributo ereditato associato con (il nodo nell'albero sintattico di) <*statlist*>.
  - Dopo <*statlist*>, l'etichetta *statlist.next*/`lnext_prog` è emessa.
  - Dopo il controllo del terminale `EOF`, `code.toJasmin()` è chiamato, per creare il file Output.j.

```
public void prog() {
    // ... completare ...
    int lnext_prog = code.newLabel();
    statlist(lnext_prog);
    code.emitLabel(lnext_prog);
    match(Tag.EOF);
    try {
            code.toJasmin();
    }
    catch(java.io.IOException e) {
            System.out.println("IO error\n");
    };
    // ... completare ...
}
```

Classe Translator

<prog> ::= {statlist.next=newlabel()} <statlist> {emitlabel(statlist.next)} EOF

- Metodo `prog`:
  - Prima azione da fare: creare una nuova etichetta (*statlist.next* nel SDT, `lnext_prog` nel codice).
  - Il valore della etichetta è assegnata ad un attributo ereditato associato con (il nodo nell'albero sintattico di) *<statlist>*.
  - Dopo *<statlist>*, l'etichetta *statlist.next*/`lnext_prog` è emessa.
  - Dopo il controllo del terminale `EOF`, `code.toJasmin()` è chiamato, per creare il file Output.j.

```
public void prog() {
    // ... completare ...
    int lnext_prog = code.newLabel();
    statlist(lnext_prog);
    code.emitLabel(lnext_prog);
    match(Tag.EOF);
    try {
            code.toJasmin();
    }
    catch(java.io.IOException e) {
            System.out.println("IO error\n");
    };
    // ... completare ...
}
```

Classe Translator

- Metodo `expr` (produzione per sottrazione):
  - Come ultima azione da fare rispetto alla produzione associata con sottrazione, emettere un comando di sottrazione (isub).

```
<expr> ::= + …
     | - <expr> <expr> {emit(isub)}
     | * …
     | / …
     | NUM
     | ID
```

```
case '-':
    match('-');
    expr();
    expr();
    code.emit(OpCode.isub);
    break;
```

Classe Translator

- Metodo `stat` (per `read`):

```
public void stat( /* completare */ ) {
    switch(look.tag) {
    // ... completare ...
        case Tag.READ:
            match(Tag.READ);
            match('(');
            idlist(/* completare */);
            match(')');
    // ... completare ...
    }
}
```

Classe Translator

- Metodo `idlist:`
  - Se l'identificatore è già nella tabella dei simboli, recuperare l'indirizzo associato con l'identificatore
  - Se l'identificatore non è stato inserito nella tabella dei simboli, inserire un nuovo elemento nella tabella (utilizzando **count** per garantire che ogni identificatore sia associato con un indirizzo diverso)

```
private void idlist(/* completare */) {
    switch(look.tag) {
        case Tag.ID:
            int id_addr = st.lookupAddress(((Word)look).lexeme);
            if (id_addr==-1) {
                id_addr = count;
                st.insert(((Word)look).lexeme,count++);
            }
            match(Tag.ID);
            /* completare */
    }
}
```