

Laboratorio di Linguaggi Formali e Traduttori

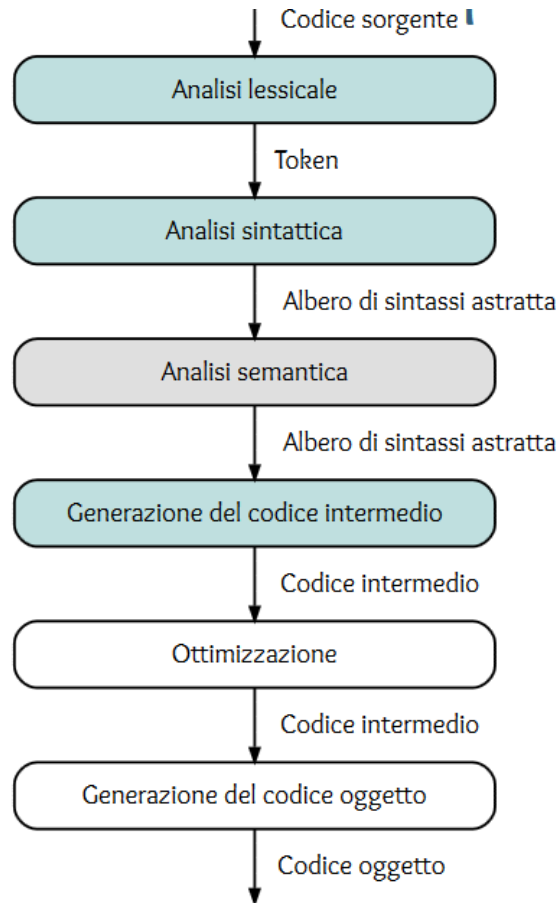
LFT lab T2, a.a. 2021/2022

Generazione del bytecode

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 di codice).
 - La classe Translator deve implementare parsing a discesa ricorsiva, e non solo traduzione (come nell'Esercizio 4.1, dove il programma da ottenere si occupa sia del parsing a discesa ricorsiva che la valutazione di espressioni aritmetiche).
 - Concetti/codice della soluzione dell'Esercizio 3.2 sono da utilizzare.
- Scrivete un SDT “on-the-fly”, ispirandosi dagli esempi di SDT “on-the-fly” delle slide di teoria.
 - Leggere con attenzione le slide sulle espressioni aritmetiche (file «comp_e.pdf»/«5.4 Traduzione di espressioni aritmetiche»), sulle espressioni booleane (file «comp_b.pdf»/«5.5 Traduzione di espressioni logiche») e sui comandi (file «comp_s.pdf»/«5.6 Traduzione di comandi»).

Generazione codice intermedio



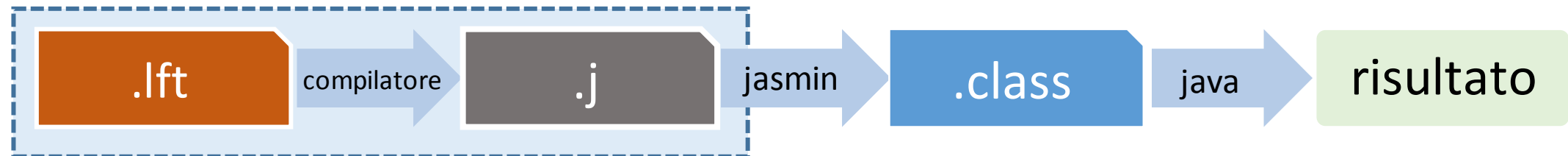
- Generazione codice intermedio:
 - Fase successiva a quelle dell'analisi lessicale e dell'analisi sintattica (l'analisi semantica non è stato affrontato in questo corso).
 - Traduzione di un programma di un linguaggio (sorgente) a un altro (oggetto).
- Nostro caso:
 - Sorgente: il linguaggio dell'Esercizio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

Generazione del bytecode

- Utilizzo tipico del JVM:

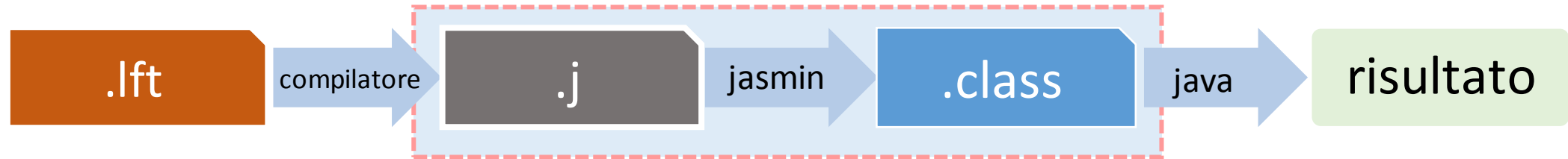


- Obiettivo: realizzare un compilatore per il linguaggio P dove il linguaggio oggetto è bytecode JVM in formato mnemonico.



- 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`.

Comandi del linguaggio

Comando	Significato	Esempio del comando
<code>assign <expr> to <idlist></code>	Assegnamento del valore di un'espressione a uno o più identificatori	<code>assign 3 to x,y</code>
<code>print (<exprlist>)</code>	Stampare sul terminale i valori di un elenco di espressioni	<code>print (+(x,3) , 4)</code>
<code>read (<idlist>)</code>	Legge un o più input dalla tastiera	<code>read(x,y)</code>
<code>{ <statlist> }</code>	Composizione sequenziale: raggruppa un elenco di comandi	<pre>{ read(x); print(*(x,3)) }</pre>

Comandi del linguaggio

Comando	Significato	Esempio del comando
<code>if (<bexpr>) <stat> end</code>	Se la condizione <bexpr> risulta valutata come vera allora <stat> è eseguito	<code>if (> x 0) {print(x); assign 3 to x} end</code>
<code>if (<bexpr>) <stat₁> else <stat₂> end</code>	Se la condizione <bexpr> risulta valutata come vera allora <stat ₁ > è eseguito, altrimenti <stat ₂ > è eseguito	<code>if (> y 0) print(y) else print(x) end</code>
<code>while (<bexpr>) <stat></code>	Ciclo: se la condizione booleana <bexpr> è vera, eseguire un comando, poi ripetere	<code>while (<> x 0) {read(x); print(x)}</code>

Esempi di traduzione

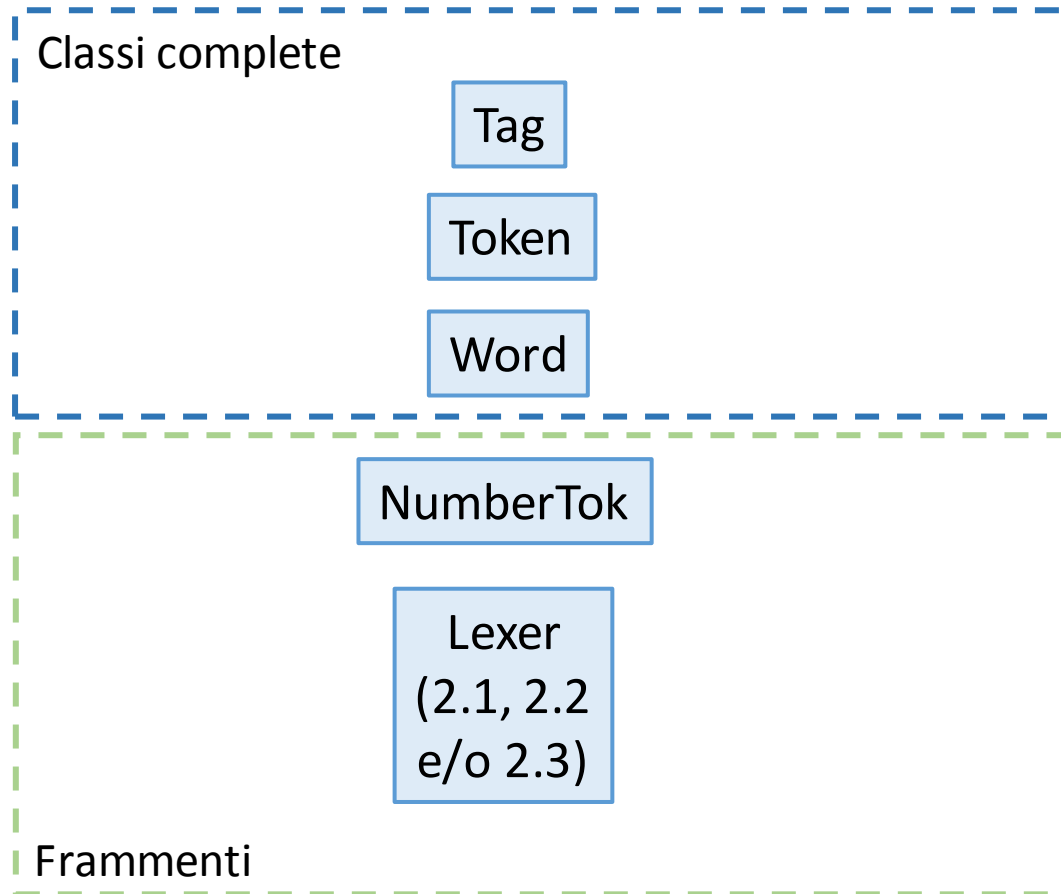
Programma .lft

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

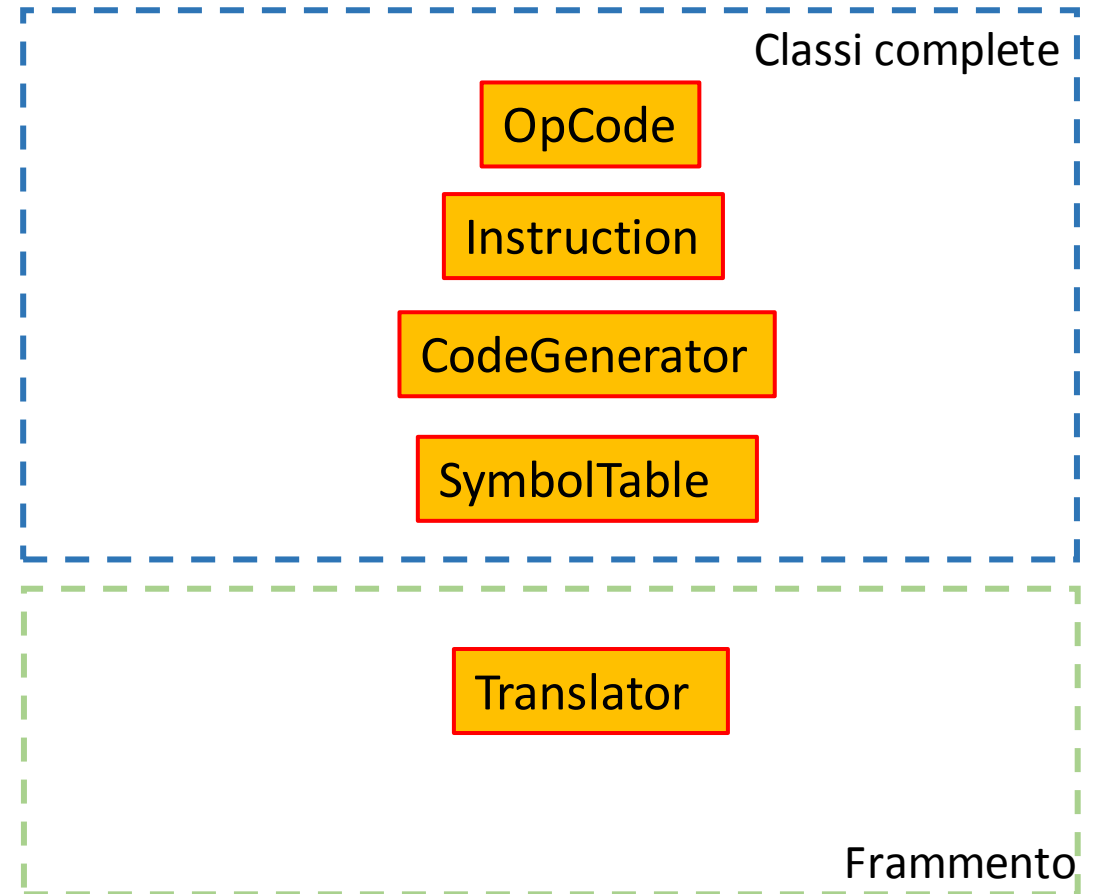
Esempio di traduzione del programma .lft
(frammento di Output.j)

```
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:
```


Classi del generatore di bytecode



Classi relative al lexer



Classi relative alla generazione del 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, 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.

```
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 ... : temp = " ... " + "\n"; break;
        }
        return temp;
    }
}
```

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();  
    }  
}
```

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();  
    }  
}
```

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();  
    }  
}
```

Classi di supporto

- SymbolTable: tabella dei simboli; per tenere traccia degli identificatori e loro indirizzi.
- Indirizzi: utilizzati come argomento dei comandi `iload` oppure `istore`.
- Metodo `insert`: inserisce un nuovo elemento (coppia lessema/indirizzo) nella tabella, se non esiste già un elemento con lo stesso lessema nella tabella.
- Metodo `lookupAddress`: dato un lessema, `lookupAddress` restituisce l'indirizzo del elemento della tabella che corrisponde al lessema (e restituisce -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 e loro indirizzi.
- Indirizzi: utilizzati come argomento dei comandi `iload` oppure `istore`.
- Metodo `insert`: inserisce un nuovo elemento (coppia lessema/indirizzo) nella tabella, se non esiste già un elemento con lo stesso lessema nella tabella.
- Metodo `lookupAddress`: dato un lessema, `lookupAddress` restituisce l'indirizzo del elemento della tabella che corrisponde al lessema (e restituisce -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 e loro indirizzi.
- Indirizzi: utilizzati come argomento dei comandi `iload` oppure `istore`.
- Metodo `insert`: inserisce un nuovo elemento (coppia lessema/indirizzo) nella tabella, se non esiste già un elemento con lo stesso lessema nella tabella.
- Metodo `lookupAddress`: dato un lessema, `lookupAddress` restituisce l'indirizzo del elemento della tabella che corrisponde al lessema (e restituisce -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;  
    }  
}
```

Classe Translator

- Metodo `prog`:

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

- 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 `prog`:

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

- 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 `prog`:

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

- 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 `prog`:

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

- 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 `prog`:

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

- 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` (produzione per `read`):

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

- 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 è 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 */  
    }  
}
```