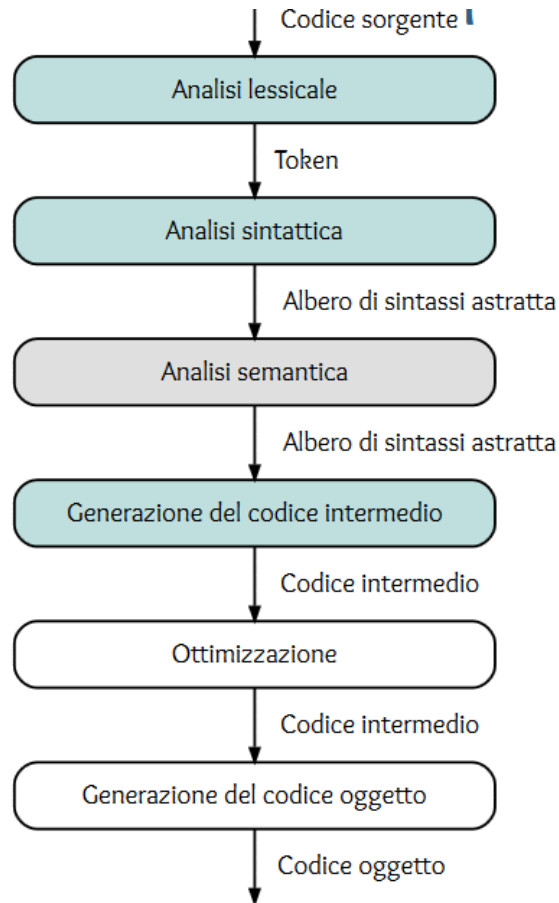


Laboratorio di Linguaggi Formali e Traduttori
LFT lab T4, a.a. 2019/2020

Generazione del bytecode

Generazione codice intermedio



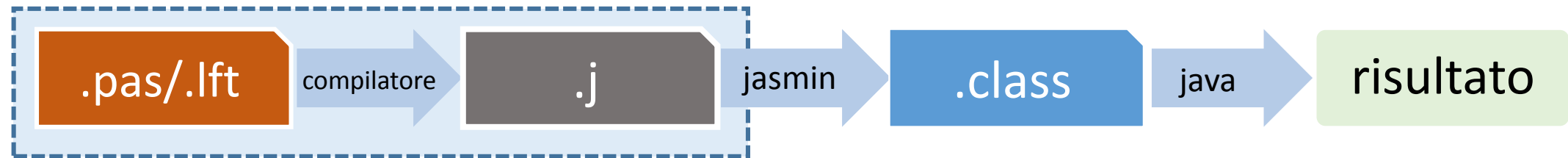
- Generazione codice intermedio:
 - Fase successiva a quelle dell'analisi lessicale e dell'analisi sintattica (l'analisi semantica non è stato affrontato in questo corso).
 - Traduzione di un programma di un linguaggio (sorgente) a un altro (oggetto).
 - Nostro caso:
 - Sorgente: il linguaggio dell'esercizio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

Generazione del bytecode

- Utilizzo tipico del JVM:

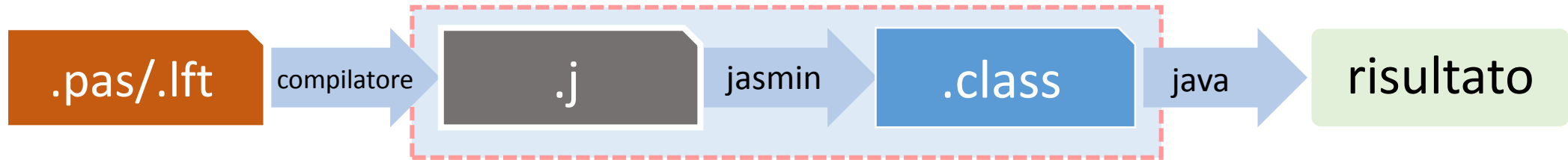


- Obiettivo: realizzare un compilatore per il linguaggio P dove il linguaggio oggetto è bytecode JVM in formato mnemonico.



- 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
= ID <expr>	Assegnamento del valore di un'espressione ad un identificatore	(= x 3)
print <exprlist>	Stampare sul terminale i valori di un elenco di espressioni	(print x 3 (+ x 1))
read ID	Legge un input dalla tastiera	(read y)
do <statlist>	Composizione sequenziale: raggruppa un elenco di comandi	(do (read x) (print (* x 3)))

Comandi del linguaggio



Comando	Significato	Esempio del comando
<code>cond <bexpr> <stat></code>	Comando condizionale: se una condizione booleana è vera, eseguire un comando (versione «senza else»)	<code>(cond (> x 0) (print x))</code>
<code>cond <bexpr> <stat> (else <stat>)</code>	Comando condizionale: se una condizione booleana è vera, eseguire un comando, altrimenti eseguire un altro comando (versione «con else»)	<code>(cond (== x y) (print 0) (else (print 1)))</code>
<code>while <bexpr> <stat></code>	Ciclo: se una condizione è vera, eseguire un comando, poi ripetere	<code>(while (<> x 0) (do (read x) (print x)))</code>

Esempio di traduzione

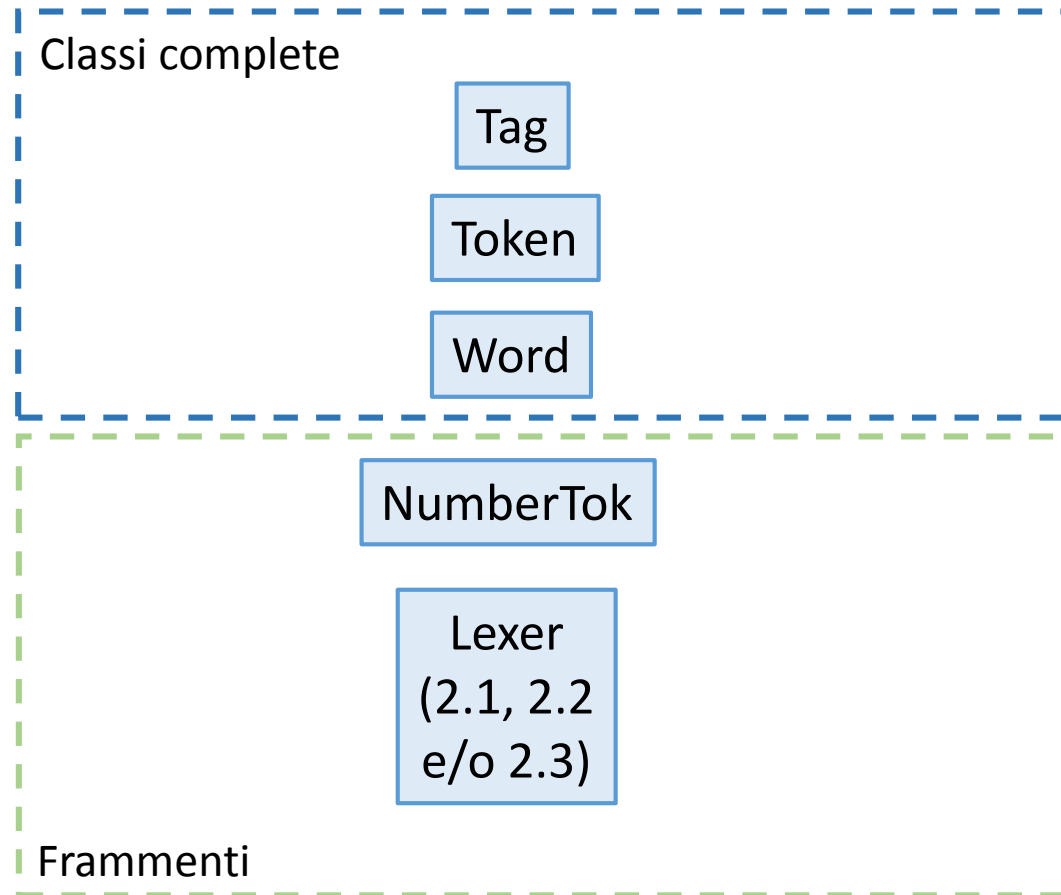
Programma .pas/.lft

```
(do
  (read a)
  (print (+ a 1))
)
```

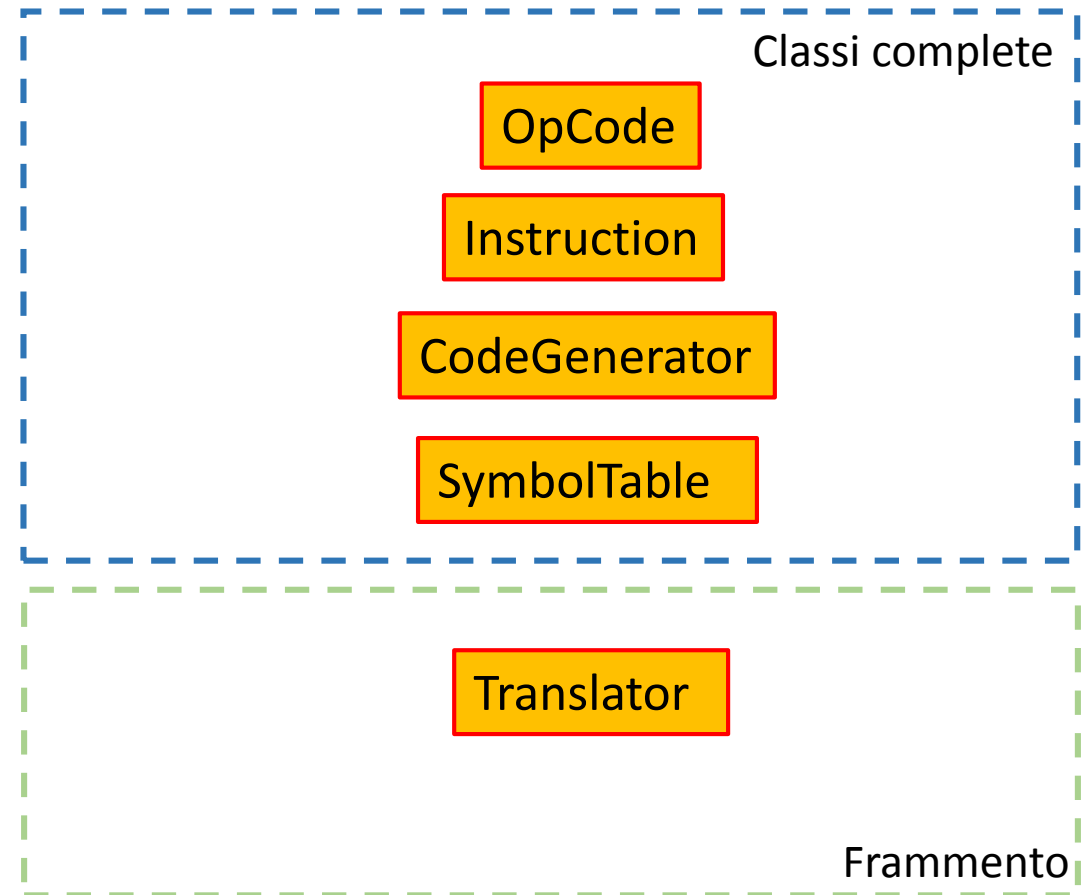
Traduzione del programma .pas/.lft
(frammento di Output.j)

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

Classi del generatore di bytecode



Classi relativi al lexer



Classi relativi 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 ...
```

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.
 - 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 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.
 - 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 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.
 - 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 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;  
    }  
}
```


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 «5.3 Codice intermedio e traduzione di espressioni aritmetiche»), e sui comandi («Grammatica dei comandi», «Assegnamento», «Comandi condizionali», «Comandi iterativi», «Composizione sequenziale» delle slide «5.4 Traduzione di espressioni logiche e comandi»).

Classe Translator

- Metodo `prog`:

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

$\langle prog \rangle ::= \{ stat.next = newlabel() \} \langle stat \rangle \{ emitlabel(stat.next) \} EOF$

```
public void prog() {  
    // ... completare ...  
    int lnext_prog = code.newLabel();  
    stat(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`:

$\langle prog \rangle ::= \{ stat.next = newlabel() \} \langle stat \rangle \{ emitlabel(stat.next) \} EOF$

- Prima azione da fare: creare una nuova etichetta (`stat.next` nel SDT, `lnext_prog` nel codice).
- Il valore della etichetta è assegnata ad un attributo ereditato associato con (il nodo nell'albero di parsificazione di) `<stat>`.
- Dopo `<stat>`, l'etichetta `stat.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();  
    stat(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`:

$\langle prog \rangle ::= \{ stat.next = newlabel() \} \langle stat \rangle \{ emitlabel(stat.next) \} EOF$

- Prima azione da fare: creare una nuova etichetta (`stat.next` nel SDT, `lnext_prog` nel codice).
- Il valore della etichetta è assegnata ad un **attributo ereditato** associato con (il nodo nell'albero di parsificazione di) `<stat>`.
- Dopo `<stat>`, l'etichetta `stat.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();  
    stat(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`:

$\langle prog \rangle ::= \{ stat.next = newlabel() \} \langle stat \rangle \{ emitlabel(stat.next) \} EOF$

- Prima azione da fare: creare una nuova etichetta (`stat.next` nel SDT, `lnext_prog` nel codice).
- Il valore della etichetta è assegnata ad un **attributo ereditato** associato con (il nodo nell'albero di parsificazione di) `<stat>`.
- Dopo `<stat>`, l'etichetta `stat.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();  
    stat(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`:

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

$\langle prog \rangle ::= \{ stat.next = newlabel() \} \langle stat \rangle \{ emitlabel(stat.next) \} EOF$

```
public void prog() {  
    // ... completare ...  
    int lnext_prog = code.newLabel();  
    stat(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 `exprp` (per sottrazione):
 - Come ultima azione da fare rispetto alla produzione associata con sottrazione, emettere un comando di sottrazione (`isub`).

$\langle exprp \rangle$::=	+	
		-	$\langle expr \rangle \langle expr \rangle \{ emit(isub) \}$
		*	
		/	

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

Classe Translator

- Metodo statp (per read):

`read ID { emit(invokestatic(read)) } { emit(istore(id.addr)) }`

- Se ID è già nella tabella dei simboli, recuperare l'indirizzo associato con l'ID.
- Se ID non è stato inserito nella tabella dei simboli, inserire un nuovo elemento nella tabella (utilizzando count per garantire che ogni ID è associato con un indirizzo diverso).
- Successivamente, chiamare il metodo per read (invokestatic con argomento 0)...
- ... e memorizzare il valore letto dalla tastiera nella parte di memoria indicato dall'indirizzo identificato nei passi precedenti.

```
public void statp(int lnext) {
    switch(look.tag) {
        // ... completare ...
        case Tag.READ:
            match(Tag.READ);
            if (look.tag==Tag.ID) {
                int read_id_addr = st.lookupAddress(((Word)look).lexeme);
                if (read_id_addr==--1) {
                    read_id_addr = count;
                    st.insert(((Word)look).lexeme, count++);
                }
                match(Tag.ID);
                code.emit(Opcode.invokestatic, 0);
                code.emit(Opcode.istore, read_id_addr);
            }
            else
                error("Error in grammar (stat) after read with " + look);
            break;
        // ... completare ...
    }
}
```


Classe Translator

- Metodo statp (per read):

```
read ID { emit(invokestatic(read)) } { emit(istore(id.addr)) }
```

- Se ID è già nella tabella dei simboli, recuperare l'indirizzo associato con l'ID.
- Se ID non è stato inserito nella tabella dei simboli, inserire un nuovo elemento nella tabella (utilizzando count per garantire che ogni ID è associato con un indirizzo diverso).
- Successivamente, chiamare il metodo per read (invokestatic con argomento 0)...
- ... e memorizzare il valore letto dalla tastiera nella parte di memoria indicato dall'indirizzo identificato nei passi precedenti.

```
public void statp(int lnext) {
    switch(look.tag) {
        // ... completare ...
        case Tag.READ:
            match(Tag.READ);
            if (look.tag==Tag.ID) {
                int read_id_addr = st.lookupAddress(((Word)look).lexeme);
                if (read_id_addr== -1) {
                    read_id_addr = count;
                    st.insert(((Word)look).lexeme, count++);
                }
                match(Tag.ID);
                code.emit(Opcode.invokestatic, 0);
                code.emit(Opcode.istore, read_id_addr);
            }
            else
                error("Error in grammar (stat) after read with " + look);
            break;
        // ... completare ...
    }
}
```

Classe Translator

- Metodo statp (per read):

read ID { emit(invokestatic(read)) } { emit(istore(id.addr)) }

- Se ID è già nella tabella dei simboli, recuperare l'indirizzo associato con l'ID.
- Se ID non è stato inserito nella tabella dei simboli, inserire un nuovo elemento nella tabella (utilizzando count per garantire che ogni ID è associato con un indirizzo diverso).
- Successivamente, chiamare il metodo per read (invokestatic con argomento 0)...
- ... e memorizzare il valore letto dalla tastiera nella parte di memoria indicato dall'indirizzo identificato nei passi precedenti.

```
public void statp(int lnext) {  
    switch(look.tag) {  
        // ... completare ...  
        case Tag.READ:  
            match(Tag.READ);  
            if (look.tag==Tag.ID) {  
                int read_id_addr = st.lookupAddress(((Word)look).lexeme);  
                if (read_id_addr== -1) {  
                    read_id_addr = count;  
                    st.insert(((Word)look).lexeme, count++);  
                }  
                match(Tag.ID);  
                code.emit(OpCodes.invokestatic, 0);  
                code.emit(OpCodes.istore, read_id_addr);  
            }  
            else  
                error("Error in grammar (stat) after read with " + look);  
            break;  
        // ... completare ...  
    }  
}
```

Classe Translator

- Metodo statp (per read):

read ID { emit(invokestatic(read)) } { emit(istore(id.addr)) }

- Se ID è già nella tabella dei simboli, recuperare l'indirizzo associato con l'ID.
- Se ID non è stato inserito nella tabella dei simboli, inserire un nuovo elemento nella tabella (utilizzando count per garantire che ogni ID è associato con un indirizzo diverso).
- Successivamente, chiamare il metodo per read (invokestatic con argomento 0)...
- ... e memorizzare il valore letto dalla tastiera nella parte di memoria indicato dall'indirizzo identificato nei passi precedenti.

```
public void statp(int lnext) {
    switch(look.tag) {
        // ... completare ...
        case Tag.READ:
            match(Tag.READ);
            if (look.tag==Tag.ID) {
                int read_id_addr = st.lookupAddress(((Word)look).lexeme);
                if (read_id_addr==--1) {
                    read_id_addr = count;
                    st.insert(((Word)look).lexeme, count++);
                }
                match(Tag.ID);
                code.emit(OpCodes.invokestatic, 0);
                code.emit(OpCodes.istore, read_id_addr);
            }
            else
                error("Error in grammar (stat) after read with " + look);
            break;
        // ... completare ...
    }
}
```