Laboratorio di Linguaggi Formali e Traduttori
Corso di Studi in Informatica
A.A. 2023/2024

Luigi Di Caro, Viviana Patti e Jeremy Sproston
Dipartimento di Informatica — Università degli Studi di Torino

Versione del 5 dicembre 2023

Sommario

Questo documento descrive le esercitazioni di laboratorio e le modalità d'esame del corso di *Linguaggi Formali e Traduttori* per l'A.A. 2023/2024.

Svolgimento e valutazione del progetto di laboratorio

È consigliato sostenere l'esame nella prima sessione d'esame dopo il corso.

Supporto on-line al corso e forum di discussione

Sulla piattaforma I-learn sono disponibili due forum: il primo è dedicato alla pubblicazioni di annunci e notizie di carattere generale, mentre il secondo è un forum di discussione dedicato per gli argomenti affrontati durante il corso. L'iscrizone al forum annunci è effettuata automaticamente, è possibile disiscriversi ma è consigliabile farlo solo a seguito del superamento dell'esame per poter sempre ricevere in modo tempestivo le comunicazioni effettuate dal docente.

Progetto di laboratorio

Il progetto di laboratorio consiste in una serie di esercitazioni assistite mirate allo sviluppo di un semplice traduttore. Il corretto svolgimento di tali esercitazioni presuppone una buona conoscenza del linguaggio di programmazione Java e degli argomenti di teoria del corso Linguaggi Formali e Traduttori.

Modalità dell'esame di laboratorio

Per sostenere l'esame a un appello è necessario prenotarsi. L'esame di laboratorio è **orale** e **individuale**, anche se il codice è stato sviluppato in collaborazione con altri studenti. Durante l'esame vengono accertati: il corretto svolgimento della prova di laboratorio; la comprensione della sua struttura e del suo funzionamento; la comprensione delle parti di teoria correlata al laboratorio stesso.

Note importanti

- Per poter discutere il laboratorio è *necessario* aver prima superato la prova scritta relativa al modulo di teoria. L'esame di laboratorio deve essere superato nella sessione d'esame in cui viene superato lo scritto, altrimenti lo scritto deve essere sostenuto nuovamente.

- La presentazione di codice "funzionante" non è condizione sufficiente per il superamento della prova di laboratorio. In altri termini, è possibile essere respinti presentando codice funzionante (se lo studente dimostra di non avere adeguata familiarità con il codice e i concetti correlati).
- Il progetto di laboratorio può essere svolto individualmente o in gruppi formati da al massimo 3 studenti. Anche se il codice è stato sviluppato in collaborazione con altri studenti, i punteggi ottenuti dai singoli studenti sono indipendenti. Per esempio, a parità di codice presentato, è possibile che uno studente meriti 30, un altro 25 e un altro ancora sia respinto.
- Dal momento che durante la prova è possibile che venga richiesto di apportare modifiche al codice del progetto, è opportuno presentarsi all'esame con un'adeguata conoscenza del progetto e degli argomenti di teoria correlati.

Calcolo del voto finale

I voti della prova scritta e della prova di laboratorio sono espressi in trentesimi. Il voto finale è determinato calcolando la media pesata del voto della prova scritta e del laboratorio , secondo il loro contributo in CFU (con una eventuale modifica nel caso in cui lo studente ha scelto di sostenere una prova orale), e cioè

$$\text{voto finale} = \frac{\text{voto dello scritto} \times 2 + \text{voto del laboratorio}}{3} \pm \text{eventuale esito orale}$$

La lode può essere attribuita agli studenti con voto finale pari a 30 e che abbiano dimostrato particolare brillantezza nello svolgimento delle esercitazioni di laboratorio.
Orale facoltativo. Dopo la discussione del laboratorio, ovvero al termine delle due prove obbligatorie, è possibile chiedere su appuntamento un'ulteriore prova orale con i docenti di teoria, per migliorare il proprio voto complessivo.

Validità del presente testo di laboratorio

Il presente testo di laboratorio è valido sino alla sessione di febbraio 2025.

1 Implementazione di un DFA in Java

Lo scopo di questo esercizio è l'implementazione di un metodo Java che sia in grado di discriminare le stringhe del linguaggio riconosciuto da un automa a stati finiti deterministico (DFA) dato. Il primo automa che prendiamo in considerazione, mostrato in Figura 1, è definito sull'alfabeto $\{0, 1\}$ e riconosce le stringhe in cui compaiono almeno 3 zeri consecutivi.

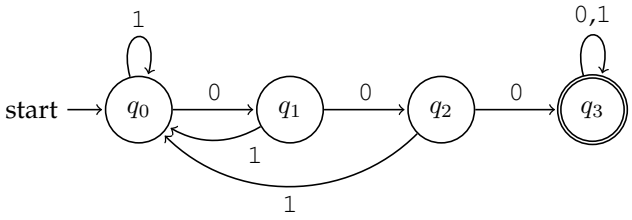


Figura 1: DFA che riconosce stringhe con 3 zeri consecutivi.

L'implementazione Java del DFA di Figura 1 è mostrata in Listing 1. L'automa è implementato nel metodo `scan` che accetta una stringa `s` e restituisce un valore booleano che indica se la

stringa appartiene o meno al linguaggio riconosciuto dall'automa. Lo stato dell'automa è rappresentato per mezzo di una variabile intera `state`, mentre la variabile `i` contiene l'indice del prossimo carattere della stringa `s` da analizzare. Il corpo principale del metodo è un ciclo che, analizzando il contenuto della stringa `s` un carattere alla volta, effettua un cambiamento dello stato dell'automa secondo la sua funzione di transizione. Notare che l'implementazione assegna il valore -1 alla variabile `state` se viene incontrato un simbolo diverso da `0` e `1`. Tale valore non è uno stato valido, ma rappresenta una condizione di errore irrecuperabile.

Listing 1: Implementazione Java del DFA di Figura 1.

```
public class TreZeri
{
    public static boolean scan(String s)
    {
        int state = 0;
        int i = 0;
        while (state >= 0 && i < s.length()) {
            final char ch = s.charAt(i++);

            switch (state) {
            case 0:
                if (ch == '0')
                    state = 1;
                else if (ch == '1')
                    state = 0;
                else
                    state = -1;
                break;

            case 1:
                if (ch == '0')
                    state = 2;
                else if (ch == '1')
                    state = 0;
                else
                    state = -1;
                break;

            case 2:
                if (ch == '0')
                    state = 3;
                else if (ch == '1')
                    state = 0;
                else
                    state = -1;
                break;

            case 3:
                if (ch == '0' || ch == '1')
                    state = 3;
                else
                    state = -1;
                break;
            }
        }
        return state == 3;
    }
```

```
    public static void main(String[] args)
    {
        System.out.println(scan(args[0]) ? "OK" : "NOPE");
    }
}
```

Esercizio 1.1. Copiare il codice in Listing 1, compilarlo e testarlo su un insieme significativo di stringhe, per es. "`010101`", "`1100011001`", "`10214`", ecc.

Come deve essere modificato il DFA in Figure 1 per riconoscere il linguaggio complementare, ovvero il linguaggio delle stringhe di `0` e `1` che **non** contengono 3 zeri consecutivi? Progettare e implementare il DFA modificato, e testare il suo funzionamento.

Esercizio 1.2. Progettare e implementare un DFA che riconosca il linguaggio degli identificatori in un linguaggio in stile Java: un identificatore è una sequenza non vuota di lettere, numeri, ed il simbolo di "underscore" _ che non comincia con un numero e che non può essere composto solo dal simbolo _. Compilare e testare il suo funzionamento su un insieme significativo di esempi.
Esempi di stringhe accettate: "`x`", "`flag1`", "`x2y2`", "`x_1`", "`lft_lab`", "`_temp`", "`x_1_y_2`", "`x___`", "`__5`"
Esempi di stringhe non accettate: "`5`", "`221B`", "`123`", "`9_to_5`", "`___`"

Esercizio 1.3. Progettare e implementare un DFA che riconosca il linguaggio di stringhe che contengono un numero di matricola seguito (subito) da un cognome, dove la combinazione di matricola e cognome corrisponde a studenti del turno 2 o del turno 3 del laboratorio di Linguaggi Formali e Traduttori. Si ricorda le regole per suddivisione di studenti in turni:

- Turno T1: cognomi la cui iniziale è compresa tra A e K, e il numero di matricola è dispari;
- Turno T2: cognomi la cui iniziale è compresa tra A e K, e il numero di matricola è pari;
- Turno T3: cognomi la cui iniziale è compresa tra L e Z, e il numero di matricola è dispari;
- Turno T4: cognomi la cui iniziale è compresa tra L e Z, e il numero di matricola è pari.

Per esempio, "`123456Bianchi`" e "`654321Rossi`" sono stringhe del linguaggio, mentre "`654321Bianchi`" e "`123456Rossi`" no. Nel contesto di questo esercizio, un numero di matricola non ha un numero prestabilito di cifre (ma deve essere composto di almeno una cifra). Un cognome corrisponde a una sequenza di lettere, e deve essere composto di almeno una lettera. Quindi l'automa deve accettare le stringhe "`2Bianchi`" e "`122B`" ma non "`654322`" e "`Rossi`".

Esercizio 1.4. Progettare e implementare un DFA che riconosca il linguaggio delle costanti numeriche in virgola mobile utilizzando la notazione scientifica dove il simbolo `e` indica la funzione esponenziale con base 10. L'alfabeto del DFA contiene i seguenti elementi: le cifre numeriche $0, 1, \ldots, 9$, il segno `.` (punto) che precede una eventuale parte decimale, i segni + (più) e − (meno) per indicare positività o negatività, e il simbolo `e`.

Le stringhe accettate devono seguire le solite regole per la scrittura delle costanti numeriche. In particolare, una costante numerica consiste di due segmenti, il secondo dei quali è opzionale: il primo segmento è una sequenza di cifre numeriche che (1) può essere preceduta da un segno + o meno −, (2) può essere seguita da un segno punto `.`, che a sua volta deve essere seguito da una sequenza non vuota di cifre numeriche; il secondo segmento inizia con il simbolo `e`, che a sua volta è seguito da una sequenza di cifre numeriche che soddisfa i punti (1) e (2) scritti per il primo segmento. Si nota che, sia nel primo segmento, sia in un eventuale secondo segmento, un segno punto `.` non deve essere preceduto per forza da una cifra numerica.
Esempi di stringhe accettate: "`123`", "`123.5`", "`.567`", "`+7.5`", "`-.7`", "`67e10`", "`1e-2`", "`-.7e2`", "`1e2.3`"
Esempi di stringhe non accettate: "`.`", "`e3`", "`123.`", "`+e6`", "`1.2.3`", "`4e5e6`", "`++3`"

Esercizio 1.5. Progettare e implementare un DFA con alfabeto $\{/, *, a\}$ che riconosca il linguaggio di "commenti" delimitati da `/*` (all'inizio) e `*/` (alla fine): cioè l'automa deve accettare le stringhe che contengono almeno 4 caratteri che iniziano con `/*`, che finiscono con `*/`, e che contengono una sola occorrenza della sequenza `*/`, quella finale (dove l'asterisco della sequenza `*/` non deve essere in comune con quello della sequenza `/*` all'inizio).
Esempi di stringhe accettate: "`/****/`", "`/*a*a*/`", "`/*a/**/`", "`/**a///a/a**/`", "`/**/`", "`/*/*/`"
Esempi di stringhe non accettate: "`/*/`", "`/**/***/`"

Esercizio 1.6. Modificare l'automa dell'esercizio precedente in modo che riconosca il linguaggio di stringhe (sull'alfabeto $\{/, *, a\}$) che contengono "commenti" delimitati da `/*` e `*/`, ma con la possibilità di avere stringhe prima e dopo come specificato qui di seguito. L'idea è che sia possibile avere eventualmente commenti (anche multipli) immersi in una sequenza di simboli dell'alfabeto. Quindi l'unico vincolo è che l'automa deve accettare le stringhe in cui un'occorrenza della sequenza `/*` deve essere seguita (anche non immediatamente) da un'occorrenza della sequenza `*/`. Le stringhe del linguaggio possono **non** avere nessuna occorrenza della sequenza `/*` (caso della sequenza di simboli senza commenti). Implementare l'automa seguendo la costruzione vista in Listing 1.
Esempi di stringhe accettate: "`aaa/****/aa`", "`aa/*a*a*/`", "`aaaa`", "`/****/`", "`/*aa*/`", "`*/a`", "`a/**/***a`", "`a/**/***/a`", "`a/**/aa/***/a`"
Esempi di stringhe non accettate: "`aaa/*/aa`", "`a/**//***a`", "`aa/*aa`"

2 Analisi lessicale

Gli esercizi di questa sezione riguardano l'implementazione di un analizzatore lessicale per un semplice linguaggio di programmazione. Lo scopo di un analizzatore lessicale è di leggere un testo e di ottenere una corrispondente sequenza di token, dove un token corrisponde ad un'unità lessicale, come un numero, un identificatore, un operatore relazionale, una parola chiave, ecc. Nelle sezioni successive, l'analizzatore lessicale da implementare sarà poi utilizzato per fornire l'input a programmi di analisi sintattica e di traduzione.

I token del linguaggio sono descritti nel modo illustrato in Tabella 1. La prima colonna contiene le varie categorie di token, la seconda presenta descrizioni dei possibili lessemi dei token, mentre la terza colonna descrive i nomi dei token, espressi come costanti numeriche.

Gli identificatori corrispondono all'espressione regolare:

$$(\mathtt{a} + ... + \mathtt{z} + \mathtt{A} + ... + \mathtt{Z})(\mathtt{a} + ... + \mathtt{z} + \mathtt{A} + ... + \mathtt{Z} + \mathtt{0} + ... + \mathtt{9})^*$$

e i numeri corrispondono all'espressione regolare $\mathtt{0} + (\mathtt{1} + ... + \mathtt{9})(\mathtt{0} + ... + \mathtt{9})^*$.

L'analizzatore lessicale dovrà ignorare tutti i caratteri riconosciuti come "spazi" (incluse le tabulazioni e i ritorni a capo), ma dovrà segnalare la presenza di caratteri illeciti, quali ad esempio `#` o `@`.

L'output dell'analizzatore lessicale dovrà avere la forma $\langle \text{token}_0 \rangle \langle \text{token}_1 \rangle \cdots \langle \text{token}_\text{n} \rangle$. Ad esempio:

- per l'input `assign 300 to d;` l'output sarà $\langle 259, \text{assign} \rangle$ $\langle 256, 300 \rangle$ $\langle 260, \text{to} \rangle$ $\langle 257, \text{d} \rangle$ $\langle 59 \rangle$ $\langle -1 \rangle$;
- per l'input `print(*{d t})` l'output sarà $\langle 267, \text{print} \rangle$ $\langle 40 \rangle$ $\langle 42 \rangle$ $\langle 123 \rangle$ $\langle 257, \text{d} \rangle$ $\langle 257, \text{t} \rangle$ $\langle 125 \rangle$ $\langle 41 \rangle$ $\langle -1 \rangle$;
- per l'input `if (> x y) assign 0 to x else print(y)` l'output sarà $\langle 261, \text{if} \rangle$ $\langle 40 \rangle$ $\langle 258, > \rangle$ $\langle 257, \text{x} \rangle$ $\langle 257, \text{y} \rangle$ $\langle 41 \rangle$ $\langle 259, \text{assign} \rangle$ $\langle 256, 0 \rangle$ $\langle 260, \text{to} \rangle$ $\langle 257, \text{x} \rangle$ $\langle 262, \text{else} \rangle$ $\langle 267, \text{print} \rangle$ $\langle 40 \rangle$ $\langle 257, \text{y} \rangle$ $\langle 41 \rangle$ $\langle -1 \rangle$;

Token	Pattern	Nome
Numeri	Costante numerica	256
Identificatore	Lettera seguita da lettere e cifre	257
Relop	Operatore relazionale (<,>,<=,>=,==,<>)	258
Assegnamento	`assign`	259
To	`to`	260
If	`if`	261
Else	`else`	262
Do	`do`	263
For	`for`	264
Begin	`begin`	265
End	`end`	266
Print	`print`	267
Read	`read`	268
Inizializzazione	`:=`	269
Disgiunzione	`||`	270
Congiunzione	`&&`	271
Negazione	`!`	33
Parentesi tonda sinistra	`(`	40
Parentesi tonda destra	`)`	41
Parentesi quadra sinistra	`[`	91
Parentesi quadra destra	`]`	93
Parentesi graffa sinistra	{	123
Parentesi graffa destra	}	125
Somma	+	43
Sottrazione	−	45
Moltiplicazione	*	42
Divisione	/	47
Punto e virgola	`;`	59
Virgola	`,`	44
EOF	Fine dell'input	-1

Tabella 1: Descrizione dei token del linguaggio

- per l'input `for (dog:=0; dog<=printread) assign dog+1 to dog` l'output sarà $\langle 264, \text{for}\rangle$ $\langle 40\rangle$ $\langle 257, \text{dog}\rangle$ $\langle 269, :=\rangle$ $\langle 256, 0\rangle$ $\langle 59\rangle$ $\langle 257, \text{dog}\rangle$ $\langle 258, <=\rangle$ $\langle 257, \text{printread}\rangle$ $\langle 41\rangle$ $\langle 259, \text{assign}\rangle$ $\langle 257, \text{dog}\rangle$ $\langle 43\rangle$ $\langle 256, 1\rangle$ $\langle 260, \text{to}\rangle$ $\langle 257, \text{dog}\rangle$ $\langle -1\rangle$.

In generale, i token della Tabella 1 hanno un attributo: ad esempio, l'attributo del token $\langle 256, 300\rangle$ è il numero 300, mentre l'attributo del token $\langle 259, \text{assign}\rangle$ è la stringa `assign`. Si noti, però, che alcuni token della Tabella 1 sono senza attributo: ad esempio, il segno "per" ($*$) è rappresentato dal token $\langle 42\rangle$, e la parentesi tonda destra ( )) è rappresentata dal token $\langle 41\rangle$.

Nota: l'analizzatore lessicale non è preposto al riconoscimento della *struttura* dei comandi del linguaggio. Pertanto, esso accetterà anche comandi "errati" quali ad esempio:

- `5+;)`
- `(34+26( - (2+15-( 27`
- `else 5 == print < end`

Altri errori invece, come simboli non previsti o sequenze illecite (ad esempio nel caso dell'input `17&5`, oppure dell'input `|||`), devono essere rilevati.

Classi di supporto. Per realizzare l'analizzatore lessicale, si possono utilizzare le seguenti classi. Definiamo una classe `Tag` in Listing 2, utilizzando le costanti intere nella colonna Nome in Tabella 1 per rappresentare i nomi dei token. Per i token che corrispondono a un solo carattere (tranne < e >, che corrispondono a "Relop", cioè agli operatori relazionali), si può utilizzare il codice ASCII del carattere: ad esempio, il nome in Tabella 1 del segno di somma (+) è 43, il codice ASCII del +.

Listing 2: Classe `Tag`

```
public class Tag {
    public final static int
        EOF = -1, NUM = 256, ID = 257, RELOP = 258,
        ASSIGN = 259, TO = 260, IF = 261, ELSE = 262,
        DO = 263, FOR = 264, BEGIN = 265, END = 266,
        PRINT = 267, READ = 268, INIT = 269, OR = 270, AND = 271;
}
```

Definiamo una classe `Token` per rappresentare i token (una possibile implementazione della classe `Token` è in Listing 3). Definiamo inoltre la classe `Word` derivata da `Token`, per rappresentare i token che corrispondono agli identificatori, alle parole chiave, alle operatori relazionali e agli elementi della sintassi che consistono di più caratteri (ad esempio `&&`). Una possibile implementazione della classe `Word` è in Listing 4. Ispirandosi alla classe `Word`, si può estendere Listing 5 per definire una classe `NumberTok` per rappresentare i token che corrispondono ai numeri.

Listing 3: Classe `Token`

```
public class Token {
    public final int tag;
    public Token(int t) { tag = t;  }
    public String toString() {return "<" + tag + ">";}
    public static final Token
        not = new Token('!'),
        lpt = new Token('('),
        rpt = new Token(')'),
        lpq = new Token('['),
        rpq = new Token(']'),
        lpg = new Token('{'),
        rpg = new Token('}'),
        plus = new Token('+'),
        minus = new Token('-'),
        mult = new Token('*'),
        div = new Token('/'),
        semicolon = new Token(';'),
        comma = new Token(',');
}
```

Listing 4: Classe `Word`

```
public class Word extends Token {
    public String lexeme = "";
    public Word(int tag, String s) { super(tag); lexeme=s; }
    public String toString() { return "<" + tag + ", " + lexeme + ">"; }
    public static final Word
        assign = new Word(Tag.ASSIGN, "assign"),
        to = new Word(Tag.TO, "to"),
        iftok = new Word(Tag.IF, "if"),
        elsetok = new Word(Tag.ELSE, "else"),
        dotok = new Word(Tag.DO, "do"),
        fortok = new Word(Tag.FOR, "for"),
        begin = new Word(Tag.BEGIN, "begin"),
```

```
        end = new Word(Tag.END, "end"),
        print = new Word(Tag.PRINT, "print"),
        read = new Word(Tag.READ, "read"),
        init = new Word(Tag.INIT, ":="),
        or = new Word(Tag.OR, "||"),
        and = new Word(Tag.AND, "&&"),
        lt = new Word(Tag.RELOP, "<"),
        gt = new Word(Tag.RELOP, ">"),
        eq = new Word(Tag.RELOP, "=="),
        le = new Word(Tag.RELOP, "<="),
        ne = new Word(Tag.RELOP, "<>"),
        ge = new Word(Tag.RELOP, ">=");
}
```

Listing 5: Classe `NumberTok`

```
public class NumberTok extends Token {
        // ... completare ...
}
```

Una possibile struttura dell'analizzatore lessicale (ispirata al testo [1, Appendice A.3]) è descritta nella classe `Lexer` in Listing 6.

Listing 6: Analizzatore lessicale di comandi semplici

```
import java.io.*;
import java.util.*;

public class Lexer {

    public static int line = 1;
    private char peek = ' ';

    private void readch(BufferedReader br) {
        try {
            peek = (char) br.read();
        } catch (IOException exc) {
            peek = (char) -1; // ERROR
        }
    }

    public Token lexical_scan(BufferedReader br) {
        while (peek == ' ' || peek == '\t' || peek == '\n'  || peek == '\r') {
            if (peek == '\n') line++;
            readch(br);
        }

        switch (peek) {
            case '!':
                peek = ' ';
                return Token.not;

        // ... gestire i casi di ( ) [ ] { } + - * / ; , ... //

            case '&':
                readch(br);
                if (peek == '&') {
                    peek = ' ';
                    return Word.and;
                } else {
```

```
                    System.err.println("Erroneous character"
                            + " after & : "  + peek );
                    return null;
                }

        // ... gestire i casi di || < > <= >= == <> ... //

            case (char)-1:
                return new Token(Tag.EOF);

            default:
                if (Character.isLetter(peek)) {

        // ... gestire il caso degli identificatori e delle parole chiave //

                } else if (Character.isDigit(peek)) {

        // ... gestire il caso dei numeri ... //

                } else {
                        System.err.println("Erroneous character: "
                                + peek );
                        return null;
                }
         }
    }

    public static void main(String[] args) {
        Lexer lex = new Lexer();
        String path = "...path..."; // il percorso del file da leggere
        try {
            BufferedReader br = new BufferedReader(new FileReader(path));
            Token tok;
            do {
                tok = lex.lexical_scan(br);
                System.out.println("Scan: " + tok);
            } while (tok.tag != Tag.EOF);
            br.close();
        } catch (IOException e) {e.printStackTrace();}
    }

}
```

Esercizio 2.1. Si scriva in Java un analizzatore lessicale che legga da file un input e stampi la sequenza di token corrispondente. Per questo esercizio, si possono utilizzare senza modifica le classi `Tag`, `Token` e `Word`. Invece le classi `NumberTok` e `Lexer` devono essere completate.

Esercizio 2.2. Consideriamo la seguente nuova definizione di identificatori: un identificatore è una sequenza non vuota di lettere, numeri, ed il simbolo di "underscore" _ ; la sequenza non comincia con un numero e non può essere composta solo dal simbolo _. Più precisamente, gli identificatori corrispondono all'espressione regolare:

$$\left(\mathtt{a}+\ldots+\mathtt{Z}\ +\ \left(\_(\_)^*(\mathtt{a}+\ldots+\mathtt{Z}+\mathtt{0}+\ldots+\mathtt{9})\right)\right)\left(\mathtt{a}+\ldots+\mathtt{Z}+\mathtt{0}+\ldots+\mathtt{9}+\_\right)^*$$

(dove $\mathtt{a}+\ldots+\mathtt{Z}$ abbrevia l'espressione regolare $\mathtt{a}+\ldots+\mathtt{z}+\mathtt{A}+\ldots+\mathtt{Z}$). Estendere il metodo `lexical_scan` per gestire identificatori che corrispondono alla nuova definizione.

Esercizio 2.3. Estendere il metodo `lexical_scan` in modo tale che possa trattare la presenza di commenti nel file di input. I commenti possono essere scritti in due modi:

- commenti delimitati con `/*` e `*/`;
- commenti che iniziano con `//` e che terminano con un a capo oppure con EOF.

I commenti devono essere ignorati dal programma per l'analisi lessicale; in altre parole, per le parti dell'input che contengono commenti, non deve essere generato nessun token. Ad esempio, consideriamo l'input seguente.

```
/* calcolare la velocita' */
assign 300 to d; // distanza
assign 10 to t; // tempo
print(* d t)
```

L'output del programma per l'analisi lessicale sarà $\langle 259, \text{assign}\rangle\ \langle 256, 300\rangle\ \langle 260, \text{to}\rangle\ \langle 257, \text{d}\rangle\ \langle 59\rangle\ \langle 259, \text{assign}\rangle\ \langle 256, 10\rangle\ \langle 260, \text{to}\rangle\ \langle 257, \text{t}\rangle\ \langle 59\rangle\ \langle 267, \text{print}\rangle\ \langle 40\rangle\ \langle 42\rangle\ \langle 257, \text{d}\rangle\ \langle 257, \text{t}\rangle\ \langle 41\rangle\ \langle -1\rangle$.

Oltre alle coppie di simboli `/*`, `*/` e `//`, un commento può contenere simboli che non fanno parte del pattern di nessun token (ad esempio, `/*@#?*/` o `/*calcolare la velocita'*/`). Se un commento di forma `/*` ... `*/` è aperto ma non chiuso prima della fine del file (si veda ad esempio il caso di input `assign 300 to d /*distanza`) deve essere segnalato un errore. Si noti che ci possono essere più commenti consecutivi non separati da nessun token, ad esempio:

```
assign 300 to d /*distanza*//*da Torino a Lione*/
```

Inoltre la coppia di simboli `*/`, se scritta al di fuori di un commento, deve essere trattata dal lexer come il segno di moltiplicazione seguito dal segno di divisione (ad esempio, per l'input `x*/y` l'output sarà $\langle 257, \text{x}\rangle\ \langle 42\rangle\ \langle 47\rangle\ \langle 257, \text{y}\rangle\ \langle -1\rangle$). In altre parole, l'idea è che in questo caso la sequenza di simboli `*/` non verrà interpretata come la chiusura di un commento ma come una sequenza dei due token menzionati.

3 Analisi sintattica

Esercizio 3.1. Si scriva un analizzatore sintattico a discesa ricorsiva che parsifichi espressioni aritmetiche molto semplici, scritte in notazione infissa, e composte soltanto da numeri non negativi (ovvero sequenze di cifre decimali), operatori di somma e sottrazione + e −, operatori di moltiplicazione e divisione `*` e `/`, simboli di parentesi `(` e `)`. In particolare, l'analizzatore deve riconoscere le espressioni generate dalla grammatica

$$G_{\texttt{expr}} = (\{\langle start\rangle, \langle expr\rangle, \langle exprp\rangle, \langle term\rangle, \langle termp\rangle, \langle fact\rangle\}, \{+, -, *, /, (,), \texttt{NUM}, \texttt{EOF}\}, P, \langle start\rangle),$$

dove P è il seguente insieme di produzioni:

$$\begin{array}{rcl}
\langle start\rangle & ::= & \langle expr\rangle\ \texttt{EOF} \\
\langle expr\rangle & ::= & \langle term\rangle\ \langle exprp\rangle \\
\langle exprp\rangle & ::= & + \langle term\rangle\ \langle exprp\rangle \\
 & | & - \langle term\rangle\ \langle exprp\rangle \\
 & | & \varepsilon \\
\langle term\rangle & ::= & \langle fact\rangle\ \langle termp\rangle \\
\langle termp\rangle & ::= & * \langle fact\rangle\ \langle termp\rangle \\
 & | & / \langle fact\rangle\ \langle termp\rangle \\
 & | & \varepsilon \\
\langle fact\rangle & ::= & (\ \langle expr\rangle\ )\ |\ \texttt{NUM}
\end{array}$$

Si noti che utilizziamo ::= anziché $\rightarrow$ per indicare una produzione, ad esempio $\langle start \rangle ::= \langle expr \rangle$`EOF` è una produzione con testa $\langle start \rangle$ e corpo $\langle expr \rangle$`EOF`.

Il programma deve fare uso dell'analizzatore lessicale sviluppato in precedenza. Si noti che l'insieme di token corrispondente alla grammatica di questa sezione è un sottoinsieme dell'insieme di token corrispondente alle regole lessicali della Sezione 2. Nei casi in cui l'input corrisponde alla grammatica, l'output deve consistere dell'elenco di token dell'input seguito da un messaggio indicando che l'input corrisponde alla grammatica. Invece nei casi in cui l'input *non* corrisponde alla grammatica, l'output del programma deve consistere di un messaggio di errore (come illustrato nelle lezioni in aula) indicando la procedura in esecuzione quando l'errore è stato individuato.

Segue una possibile struttura del programma (ispirato al testo [1, Appendice A.8]).

Listing 7: Analizzatore sintattico di espressioni semplici

```
import java.io.*;

public class Parser {
    private Lexer lex;
    private BufferedReader pbr;
    private Token look;

    public Parser(Lexer l, BufferedReader br) {
        lex = l;
        pbr = br;
        move();
    }

    void move() {
        look = lex.lexical_scan(pbr);
        System.out.println("token = " + look);
    }

    void error(String s) {
        throw new Error("near line " + lex.line + ": " + s);
    }

    void match(int t) {
        if (look.tag == t) {
            if (look.tag != Tag.EOF) move();
        } else error("syntax error");
    }

    public void start() {
        // ... completare ...
        expr();
        match(Tag.EOF);
        // ... completare ...
    }

    private void expr() {
        // ... completare ...
    }

    private void exprp() {
        switch (look.tag) {
        case '+':
        // ... completare ...
        }
    }
```

```
    private void term() {
        // ... completare ...
    }

    private void termp() {
        // ... completare ...
    }

    private void fact() {
        // ... completare ...
    }

    public static void main(String[] args) {
        Lexer lex = new Lexer();
        String path = "...path..."; // il percorso del file da leggere
        try {
            BufferedReader br = new BufferedReader(new FileReader(path));
            Parser parser = new Parser(lex, br);
            parser.start();
            System.out.println("Input OK");
            br.close();
        } catch (IOException e) {e.printStackTrace();}
    }
}
```

Esercizio 3.2. Seguono le produzioni di una grammatica per un semplice linguaggio di programmazione. Come nell'Esercizio 3.1, le variabili sono denotate con le parentesi angolari (per esempio, $\langle prog \rangle$, $\langle statlist \rangle$, $\langle statlistp \rangle$, ecc.). I terminali della grammatica corrispondono ai token descritti in Sezione 2 (in Tabella 1).

$$
\begin{array}{rcl}
\langle prog \rangle & ::= & \langle statlist \rangle \texttt{ EOF} \\
\langle statlist \rangle & ::= & \langle stat \rangle \ \langle statlistp \rangle \\
\langle statlistp \rangle & ::= & \texttt{;} \ \langle stat \rangle \ \langle statlistp \rangle \ \mid \ \varepsilon \\
\langle stat \rangle & ::= & \texttt{assign} \ \langle assignlist \rangle \\
 & \mid & \texttt{print (} \ \langle exprlist \rangle \ \texttt{)} \\
 & \mid & \texttt{read (} \ \langle idlist \rangle \ \texttt{)} \\
 & \mid & \texttt{for (ID :=} \ \langle expr \rangle \ \texttt{;} \ \langle bexpr \rangle \ \texttt{) do} \ \langle stat \rangle \\
 & \mid & \texttt{for (} \ \langle bexpr \rangle \ \texttt{) do} \ \langle stat \rangle \\
 & \mid & \texttt{if (} \ \langle bexpr \rangle \ \texttt{)} \ \langle stat \rangle \ \texttt{else} \ \langle stat \rangle \ \texttt{end} \\
 & \mid & \texttt{if (} \ \langle bexpr \rangle \ \texttt{)} \ \langle stat \rangle \ \texttt{end} \\
 & \mid & \{ \ \langle statlist \rangle \ \} \\
\langle assignlist \rangle & ::= & \texttt{[} \ \langle expr \rangle \ \texttt{to} \ \langle idlist \rangle \ \texttt{]} \ \langle assignlistp \rangle \\
\langle assignlistp \rangle & ::= & \texttt{[} \ \langle expr \rangle \ \texttt{to} \ \langle idlist \rangle \ \texttt{]} \ \langle assignlistp \rangle \ \mid \ \varepsilon \\
\langle idlist \rangle & ::= & \texttt{ID} \ \langle idlistp \rangle \\
\langle idlistp \rangle & ::= & \texttt{,} \ \texttt{ID} \ \langle idlistp \rangle \ \mid \ \varepsilon \\
\langle bexpr \rangle & ::= & \texttt{RELOP} \ \langle expr \rangle \ \langle expr \rangle \\
\langle expr \rangle & ::= & \texttt{+ (} \ \langle exprlist \rangle \ \texttt{)} \ \mid \ \texttt{-} \ \langle expr \rangle \ \langle expr \rangle \\
 & \mid & \texttt{* (} \ \langle exprlist \rangle \ \texttt{)} \ \mid \ \texttt{/} \ \langle expr \rangle \ \langle expr \rangle \\
 & \mid & \texttt{NUM} \ \mid \ \texttt{ID} \\
\langle exprlist \rangle & ::= & \langle expr \rangle \ \langle exprlistp \rangle \\
\langle exprlistp \rangle & ::= & \texttt{,} \ \langle expr \rangle \ \langle exprlistp \rangle \ \mid \ \varepsilon
\end{array}
$$

Si noti che RELOP corrisponde a un elemento dell'insieme {==, <>, <=, >=, <, >}, NUM corrisponde a una costante numerica e ID corrisponde a un identificatore. Inoltre, si noti che le espressioni aritmetiche sono scritte in *notazione prefissa* o polacca, diversamente da quanto accadeva nell'esercizio precedente dove venivano scritte secondo la notazione infissa (standard). Analogamente le espressioni booleane sono scritte in notazione prefissa, seguendo la convenzione di porre l'operatore relazionale a sinistra delle espressioni. Modificare la grammatica per ottenere una grammatica LL(1) equivalente, e scrivere un analizzatore sintattico a discesa ricorsiva per la grammatica ottenuta.

4 Traduzione diretta dalla sintassi

Esercizio 4.1 (Valutatore di espressioni semplici)**.** Modificare l'analizzatore sintattico dell'esercizio 3.1 in modo da valutare le espressioni aritmetiche semplici, facendo riferimento allo schema

di traduzione diretto dalla sintassi seguente:

$$
\begin{array}{rcl}
\langle start \rangle & ::= & \langle expr \rangle \texttt{EOF} \ \{ \ print(expr.val) \ \} \\
\\
\langle expr \rangle & ::= & \langle term \rangle \ \{ \ exprp.i = term.val \ \} \ \langle exprp \rangle \ \{ \ expr.val = exprp.val \ \} \\
\\
\langle exprp \rangle & ::= & + \ \langle term \rangle \ \{ \ exprp_1.i = exprp.i + term.val \ \} \ \langle exprp_1 \rangle \ \{ \ exprp.val = exprp_1.val \ \} \\
& | & - \ \langle term \rangle \ \{ \ exprp_1.i = exprp.i - term.val \ \} \ \langle exprp_1 \rangle \ \{ \ exprp.val = exprp_1.val \ \} \\
& | & \varepsilon \ \{ \ exprp.val = exprp.i \ \} \\
\\
\langle term \rangle & ::= & \langle fact \rangle \ \{ \ termp.i = fact.val \ \} \ \langle termp \rangle \ \{ \ term.val = termp.val \ \} \\
\\
\langle termp \rangle & ::= & * \ \langle fact \rangle \ \{ \ termp_1.i = termp.i * fact.val \ \} \ \langle termp_1 \rangle \ \{ \ termp.val = termp_1.val \ \} \\
& | & / \ \langle fact \rangle \ \{ \ termp_1.i = termp.i / fact.val \ \} \ \langle termp_1 \rangle \ \{ \ termp.val = termp_1.val \ \} \\
& | & \varepsilon \ \{ \ termp.val = termp.i \ \} \\
\\
\langle fact \rangle & ::= & \texttt{(} \ \langle expr \rangle \ \texttt{)} \ \{ \ fact.val = expr.val \ \} \mid \texttt{NUM} \ \{ \ fact.val = \texttt{NUM}.value \ \}
\end{array}
$$

Si noti che il terminale `NUM` ha l'attributo *value*, che è il valore numerico del terminale, fornito dall'analizzatore lessicale.

Una possibile struttura del programma è la seguente.

Nota: come indicato, è fortemente consigliata la creazione di una nuova classe (chiamata `Valutatore` in Listing 8).

Listing 8: Valutazione di espressioni semplici

```
import java.io.*;

public class Valutatore {
    private Lexer lex;
    private BufferedReader pbr;
    private Token look;

    public Valutatore(Lexer l, BufferedReader br) {
        lex = l;
        pbr = br;
        move();
    }

    void move() {
        // come in Esercizio 3.1
    }

    void error(String s) {
        // come in Esercizio 3.1
    }

    void match(int t) {
        // come in Esercizio 3.1
    }

    public void start() {
        int expr_val;

        // ... completare ...

        expr_val = expr();
        match(Tag.EOF);
```

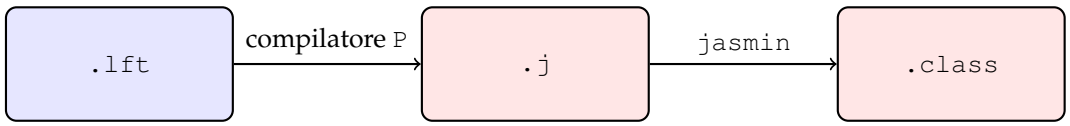
```
        System.out.println(expr_val);

        // ... completare ...
    }

    private int expr() {
        int term_val, exprp_val;

        // ... completare ...

        term_val = term();
        exprp_val = exprp(term_val);

        // ... completare ...
        return exprp_val;
    }

    private int exprp(int exprp_i) {
        int term_val, exprp_val;
        switch (look.tag) {
        case '+':
                match('+');
                term_val = term();
                exprp_val = exprp(exprp_i + term_val);
                break;

        // ... completare ...
        }
    }

    private int term() {
        // ... completare ...
    }

    private int termp(int termp_i) {
        // ... completare ...
    }

    private int fact() {
        // ... completare ...
    }

    public static void main(String[] args) {
        Lexer lex = new Lexer();
        String path = "...path..."; // il percorso del file da leggere
        try {
            BufferedReader br = new BufferedReader(new FileReader(path));
            Valutatore valutatore = new Valutatore(lex, br);
            valutatore.start();
            br.close();
        } catch (IOException e) {e.printStackTrace();}
    }
}
```

5 Generazione del bytecode

L'obiettivo di quest'ultima parte di laboratorio è di quello di realizzare un traduttore per i programmi scritti nel linguaggio di programmazione semplice, che chiameremo `P`, visto nell'esercizio 3.2. I file di programmi del linguaggio `P` hanno estensione `.lft`, come suggerito nelle lezioni di teoria. Il traduttore deve generare bytecode [4] eseguibile dalla Java Virtual Machine (JVM).

Generare bytecode eseguibile direttamente dalla JVM non è un'operazione semplice a causa della complessità del formato dei file `.class` (che tra l'altro è un formato binario) [3]. Il bytecode verrà quindi generato avvalendoci di un linguaggio mnemonico che fa riferimento alle istruzioni della JVM (linguaggio assembler [2]) e che successivamente verrà tradotto nel formato `.class` dal programma assembler. Il linguaggio mnemonico utilizzato fa riferimento all'insieme delle istruzioni della JVM [5] e l'assembler effettua una traduzione 1-1 delle istruzioni mnemoniche nella corrispondente istruzione (*opcode*) della JVM. Il programma assembler che utilizzeremo si chiama Jasmin (distribuzione e manuale sono disponibili all'indirizzo `http://jasmin.sourceforge.net/`).

La costruzione del file `.class` a partire dal sorgente scritto nel linguaggio `P` avviene secondo lo schema seguente:



Il file sorgente viene tradotto dal compilatore (oggetto della realizzazione) nel linguaggio assembler per la JVM. Questo file (che deve avere l'estensione `.j`) è poi trasformato in un file `.class` dal programma assembler Jasmin. Nel codice presentato in questa sezione, il file generato dal compilatore si chiama `Output.j`, e il comando `java -jar jasmin.jar Output.j` è utilizzato per trasformarlo nel file `Output.class`, che può essere eseguito con il comando `java Output`.

Significato delle istruzioni del linguaggio `P`.

`assign [` $\langle expr \rangle_1$ **`to`** $\langle idlist \rangle_1$ **`]`** $\cdots$ **`[`** $\langle expr \rangle_n$ **`to`** $\langle idlist \rangle_n$ **`]`**

Per ogni $i \in \{1, \cdots, n\}$, assegna il valore dell'espressione $\langle expr \rangle_i$ a tutti gli identificatori inclusi nella lista di identificatori $\langle idlist \rangle_i$. Ad esempio, l'istruzione

```
assign [3 to x,y] [+(x,y) to z]
```

corrisponde all'assegnamento del valore 3 a entrambi gli identificatori `x` e `y`, seguito dall'assegnamento della somma del valore di `x` e il valore di `y` all'identificatore `z`.

`print (` $\langle exprlist \rangle$ **`)`**

Stampa sul terminale il valore di tutte le espressioni incluse nella lista di espressioni $\langle exprlist \rangle$. Ad esempio, `print(+(2,3),4])` stampa sul terminale il valore 5 seguito dal valore 4.

`read (` $\langle idlist \rangle$ **`)`**

Il comando `read(`$\mathtt{ID}_1, ..., \mathtt{ID}_n$`)` permette l'inserimento di n numeri interi non-negativi dalla tastiera; l'i-esimo numero inserito dalla tastiera è assegnato all'identificatore $\mathtt{ID}_i$. Ad esempio, il comando `read(a,b)` specifica che l'utente del programma scritto in linguaggio `P` deve inserire due numeri interi non-negativi con la tastiera, il primo del quale è assegnato all'identificatore `a` e il secondo del quale è assegnato all'identificatore `b`.

`for (` $\langle bexpr \rangle$ **`) do`** $\langle stat \rangle$

Permette l'esecuzione ciclica di $\langle stat \rangle$. La condizione per l'esecuzione del ciclo è $\langle bexpr \rangle$. Si noti che $\langle stat \rangle$ può essere una singola istruzione oppure una sequenza di istruzioni racchiusa tra parentesi graffe. Ad esempio, l'istruzione

```
for (> x 0) { assign [- x 1 to x]; print(x) }
```

decrementa e stampa sul terminale il valore dell'identificatore `x`, fino a quando `x` raggiunge il valore 0.

`for (``ID :=` $\langle expr \rangle$ `;` $\langle bexpr \rangle$ **`) do`** $\langle stat \rangle$

Questo comando assomiglia a quello precedente, tranne per il fatto che all'identificatore `ID` è assegnato il valore dell'espressione $\langle expr \rangle$ all'inizio dell'esecuzione del comando, cioè prima del primo controllo della condizione $\langle bexpr \rangle$. Ad esempio, l'istruzione

```
for (y := 10; > y 0) { assign [- y 1 to y]; print(y) }
```

assegna il valore 10 all'identificatore `y`, poi decrementa e stampa sul terminale il valore di `y`, fino a quando `y` raggiunge il valore 0.

`if (` $\langle bexpr \rangle$ **`)`** $\langle stat \rangle$ **`end`**

È un comando condizionale che assomiglia al comando `if` (senza `else`) di Java: se la condizione $\langle bexpr \rangle$ risulta valutata come vera allora $\langle stat \rangle$ è eseguito, poi si procede all'istruzione successiva all'`if...end`; invece se la condizione $\langle bexpr \rangle$ risulta valutata come falsa si procede direttamente all'istruzione successiva all'`if...end`.

`if (` $\langle bexpr \rangle$ **`)`** $\langle stat_1 \rangle$ **`else`** $\langle stat_2 \rangle$ **`end`**

È un comando condizionale che assomiglia al comando `if...else...` di Java: se la condizione $\langle bexpr \rangle$ risulta valutata come vera allora $\langle stat_1 \rangle$ è eseguito, poi si procede all'istruzione successiva all'`if...else...end`; invece se la condizione $\langle bexpr \rangle$ risulta valutata come falsa allora $\langle stat_2 \rangle$ è eseguito, poi si procede all'istruzione successiva all'`if...else...end`.

`{` $\langle statlist \rangle$ **`}`**

Permette di raggruppare una sequenza di istruzioni. Ad esempio, un blocco di istruzioni che legge in input un numero e successivamente stampa sul terminale il numero incrementato di 1 può essere scritto nella maniera seguente: `{read(x); print(+(x,1))}`.

Si noti l'utilizzo della notazione *prefissa* per le espressioni aritmetiche. Inoltre, le operazioni di somma e moltiplicazione possono avere un numero n di argomenti con $n \geq 1$. Invece, le operazioni di sottrazione e divisione hanno esattamente due argomenti. Ad esempio, l'espressione `*(2,3,4)` ha valore 24, l'espressione `-*(2,4)3` ha valore 5, l'espressione `+(2,- 7 3)` ha valore 6, l'espressione `+(/ 10 2, 3)` ha valore 8, e l'espressione `+(5, - 7 3, 10)` ha valore 19.

Classi di supporto. Per la realizzazione del compilatore utilizziamo le seguenti classi.

La classe `OpCode` è una semplice enumerazione dei nomi mnemonici (si veda [5] per un elenco dei nomi, ma utilizziamo solo quelli utili per la traduzione del linguaggio `P`). La classe `Instruction` verrà usata per rappresentare singole istruzioni del linguaggio mnemonico. Il metodo `toJasmin` restituisce l'istruzione nel formato adeguato per l'assembler Jasmin.

Listing 9: Una possibile implementazione della classe `OpCode`

```
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 }
```

Listing 10: Una possibile implementazione della classe `Instruction`

```
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 idiv : temp = " idiv " + "\n"; break;
            case isub : temp = " isub " + "\n"; break;
            case ineg : temp = " ineg " + "\n"; break;
            case istore : temp = " istore " + operand + "\n"; break;
            case ior : temp = " ior " + "\n"; break;
            case iand : temp = " iand " + "\n"; break;
            case iload : temp = " iload " + operand + "\n"; break;
            case if_icmpeq : temp = " if_icmpeq L" + operand + "\n"; break;
            case if_icmple : temp = " if_icmple L" + operand + "\n"; break;
            case if_icmplt : temp = " if_icmplt L" + operand + "\n"; break;
            case if_icmpne : temp = " if_icmpne L" + operand + "\n"; break;
            case if_icmpge : temp = " if_icmpge L" + operand + "\n"; break;
            case if_icmpgt : temp = " if_icmpgt L" + operand + "\n"; break;
            case ifne : temp = " ifne L" + operand + "\n"; break;
            case GOto : temp = " goto L" + operand + "\n" ; break;
            case dup : temp = " dup" + "\n" ; break;
            case pop : temp = " pop" + "\n" ; break;
            case label : temp = "L" + operand + ":\n"; break;
        }
    return temp;
    }
}
```

La classe `CodeGenerator` ha lo scopo di memorizzare in una struttura apposita la *lista delle istruzioni* (come oggetti di tipo `Instruction`) generate durante la parsificazione. I metodi `emit` 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.

Listing 11: Una possibile implementazione della classe `CodeGenerator`

```
import java.util.LinkedList;
import java.io.*;

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();
    }

    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 java/util/Scanner/next()Ljava/lang/String;\n"
        + " invokestatic java/lang/Integer.parseInt(Ljava/lang/String;)I\n"
        + " ireturn\n"
        + ".end method\n"
```

```
        + "\n"
        + ".method public static run()V\n"
        + " .limit stack 1024\n"
        + " .limit locals 256\n";

    private static final String footer = " return\n"
        + ".end method\n"
        + "\n"
        + ".method public static main([Ljava/lang/String;)V\n"
        + " invokestatic Output/run()V\n"
        + " return\n"
        + ".end method\n";
}
```

Per tenere traccia degli identificatori e loro indirizzi, occorre predisporre una *tabella dei simboli* (*symbol table*). L'indirizzo associato con un identificatore può essere utilizzato come argomento dei comandi `iload` oppure `istore`. Segue una possibile implementazione della classe `SymbolTable`:

Listing 12: Una possibile implementazione della classe `SymbolTable`

```
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;
    }
}
```

Esempi di generazione di bytecode. Seguono alcuni semplici programmi `P` affiancati da due esempi corrispondenti di bytecode JVM. Si noti che possono essere diversi modi di tradurre correttamente lo stesso programma `P`.

Listing 13: Programma `A.lft`

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

Listing 14: Esempio di bytecode di `A.lft`

```
 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:
```

Listing 15: Programma `B.lft`

```
assign [10 to x]
       [20 to y]
       [30 to z];
print(+(x,*(y,z)))
```

Listing 16: Esempio di bytecode di `B.lft`

```
 ldc 10
 istore 0
 ldc 20
 istore 1
 ldc 30
 istore 2
 goto L1
L1:
 iload 0
 iload 1
 iload 2
 imul
 iadd
 invokestatic Output/print(I)V
 goto L2
L2:
 goto L0
L0:
```

Esercizio 5.1. Si scriva un traduttore per programmi scritti nel linguaggio `P` (utilizzando uno dei lexer sviluppati per gli esercizi di Sezione 2). Si ricorda che la grammatica del linguaggio `P` è definita nell'esercizio 3.2.

Un frammento del codice di una possibile implementazione può essere trovato in Listing 17 (si noti che `code` è un oggetto della classe `CodeGenerator`).

Listing 17: Un frammento del codice di una possibile implementazione del programma di Esercizio 5.1

```
import java.io.*;

public class Translator {
    private Lexer lex;
    private BufferedReader pbr;
    private Token look;

    SymbolTable st = new SymbolTable();
    CodeGenerator code = new CodeGenerator();
    int count=0;
```

```
public Translator(Lexer l, BufferedReader br) {
    lex = l;
    pbr = br;
    move();
}

void move() {
    // come in Esercizio 3.1
}

void error(String s) {
    // come in Esercizio 3.1
}

void match(int t) {
    // come in Esercizio 3.1
}

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 ...
}

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

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 ...
    }
}
```

```
    private void expr( /* completare */ ) {
        switch(look.tag) {
        // ... completare ...
            case '-':
                match('-');
                expr();
                expr();
                code.emit(OpCode.isub);
                break;
        // ... completare ...
        }
    }

// ... completare ...
}
```

Riferimenti bibliografici

[1] Aho, Alfred V., Lam, Monica S., Sethi, Ravi, and Ullman, Jeffrey D. Compilatori: Principi, tecniche e strumenti. *Pearson*, 2019.

[2] Assembly language. `http://en.wikipedia.org/wiki/Assembly_language` *Wikipedia*, 2023.

[3] Java class file. `http://en.wikipedia.org/wiki/Java_class_file` *Wikipedia*, 2023.

[4] Java bytecode. `http://en.wikipedia.org/wiki/Java_bytecode` *Wikipedia*, 2023.

[5] Java bytecode instruction listings. `http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings` *Wikipedia*, 2023.