Laboratorio di Linguaggi Formali e Traduttori
Corso di Studi in Informatica
A.A. 2016/2017

Viviana Patti e Jeremy Sproston
Dipartimento di Informatica — Università degli Studi di Torino

Versione del 20 dicembre 2016

Sommario

Questo documento descrive le esercitazioni di laboratorio e le modalità d'esame del corso di *Linguaggi Formali e Traduttori* per l'A.A. 2016/2017.

Svolgimento e valutazione del progetto di laboratorio

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

Forum di discussione e supporto on-line al corso

Sulla piattaforma I-learn sono disponibili forum di discussione dedicati per gli argomenti affrontati durante il corso e per scambiare opinioni tra i vari gruppi di lavoro e con il docente. L'iscrizione al forum principale 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. Si ricorda che le sessioni d'esame sono tre: (1) gennaio/febbraio, (2) giugno/luglio, e (3) settembre.

- 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).
- 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, e cioè

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

La lode è attribuita agli studenti con voto finale pari a 30 e che abbiano dimostrato particolare brillantezza nello svolgimento delle esercitazioni di laboratorio.

Validità del presente testo di laboratorio

Il presente testo di laboratorio è valido sino alla sessione di settembre 2017.

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.

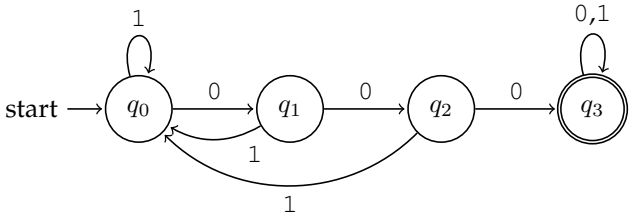


Figura 1: DFA che riconosce stringhe con 3 zeri consecutivi.

L'implementazione Java del DFA di Figura 1 è mostrata in Figura 2. 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

```java
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");
    }
}
```

Figura 2: Implementazione Java del DFA di Figura 1.

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.

Esercizio 1.1. Copiare il codice in Figura 2, compilarlo e testarlo su un insieme significativo di stringhe, per es. "`010101`", "`1100011001`", "`10214`", ecc.

Come deve essere modificato il DFA in Figura 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.

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 1: cognomi la cui iniziale è compresa tra A e K, e il numero di matricola è dispari;
- Turno 2: cognomi la cui iniziale è compresa tra A e K, e il numero di matricola è pari;
- Turno 3: cognomi la cui iniziale è compresa tra L e Z, e il numero di matricola è dispari;
- Turno 4: 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`". Assicurarsi che il DFA sia minimo.

Esercizio 1.4. Modificare l'automa dell'esercizio precedente in modo che riconosca le combinazioni di matricola e cognome di studenti del turno 2 o del turno 3 del laboratorio, dove il numero di matricola e il cognome possono essere separati da una sequenza di spazi, e possono essere precedute e/o seguite da sequenze eventualmente vuote di spazi. Per esempio, l'automa deve accettare la stringa "`654321 Rossi`" e "` 123456 Bianchi `" (dove, nel secondo esempio, ci sono spazi prima del primo carattere e dopo l'ultimo carattere), ma non "`1234 56Bianchi`" e "`123456Bia nchi`". Per questo esercizio, i cognomi composti (con un numero arbitrario di parti) possono essere accettati: per esempio, la stringa "`123456De Gasperi`" deve essere accettato. Modificare l'implementazione Java dell'automa di conseguenza.

Esercizio 1.5. Progettare e implementare un DFA che, come in Esercizio 1.3, riconosca il linguaggio di stringhe che contengono matricola e cognome di studenti del turno 2 o del turno 3 del laboratorio, ma in cui il cognome precede il numero di matricola (in altre parole, le posizioni del cognome e matricola sono scambiate rispetto all'Esercizio 1.3). Assicurarsi che il DFA sia minimo.

Esercizio 1.6. Progettare e implementare un DFA che riconosca il linguaggio dei numeri binari (stringhe di `0` e `1`) il cui valore è multiplo di 3. Per esempio, "`110`" e "`1001`" sono stringhe del linguaggio (rappresentano rispettivamente i numeri 6 e 9), mentre "`10`" e "`111`" no (rappresentano rispettivamente i numeri 2 e 7).
Suggerimento: usare tre stati per rappresentare il resto della divisione per 3 del numero.

Esercizio 1.7. Progettare e implementare un DFA con alfabeto $\{/, \star, a\}$ che riconosca il linguaggio di "commenti" delimitati da `/*` (all'inizio) e `*/` (alla fine): cioé 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,). Quindi l'automa deve accettare le stringhe "`/****/`", "`/*a*a*/`", "`/*a/**/`", "`/**a///a/a**/`", "`/**/`" e "`/*/*/`" ma non "`/*/`", oppure "`/**/***/`".

Esercizio 1.8. 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) 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). Ad esempio, il DFA deve accettare le stringhe "`aaa/****/aa`", "`aa/*a*a*/`", "`aaaa`", "`/****/`", "`/*aa*/`", "`*/a`", "`a/**/***a`", "`a/**/***/a`" e "`a/**/aa/***/a`", ma non "`aaa/*/aa`" oppure "`aa/*aa`". Implementare l'automa seguendo la costruzione vista in Figura 2.

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 $[\mathtt{a}-\mathtt{zA}-\mathtt{Z}][\mathtt{a}-\mathtt{zA}-\mathtt{Z0}-\mathtt{9}]^*$, e i numeri corrispondono all'espressione regolare $[\mathtt{0}-\mathtt{9}][\mathtt{0}-\mathtt{9}]^*$.

Un esempio di programma scritto con questi elementi è il seguente:

```
d:=300;
t:=10;
print(d*t)
```

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 `d:=300;` l'output sarà $\langle 257, \text{d} \rangle \ \langle 262, := \rangle \ \langle 256, 300 \rangle \ \langle 59 \rangle \ \langle -1 \rangle$;
- per l'input `print(d*t)` l'output sarà $\langle 263, \text{print} \rangle \ \langle 40 \rangle \ \langle 257, \text{d} \rangle \ \langle 42 \rangle \ \langle 257, \text{t} \rangle \ \langle 41 \rangle \ \langle -1 \rangle$;
- per l'input `if(x>y) x:=0` l'output sarà $\langle 259, \text{if} \rangle \ \langle 40 \rangle \ \langle 257, \text{x} \rangle \ \langle 258, > \rangle \ \langle 257, \text{y} \rangle \ \langle 41 \rangle \ \langle 257, \text{x} \rangle \ \langle 262, := \rangle \ \langle 256, 0 \rangle \ \langle -1 \rangle$;
- per l'input `while (ifx<=printread) ifx:=ifx+1` l'output sarà $\langle 261, \text{while} \rangle \ \langle 40 \rangle \ \langle 257, \text{ifx} \rangle \ \langle 258, <= \rangle \ \langle 257, \text{printread} \rangle \ \langle 41 \rangle \ \langle 257, \text{ifx} \rangle \ \langle 262, := \rangle \ \langle 257, \text{ifx} \rangle \ \langle 43 \rangle \ \langle 256, 1 \rangle \ \langle -1 \rangle$.

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 $\langle 259, \text{if} \rangle$ è la stringa `if`. 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 graffa destra (}) è rappresentata dal token $\langle 125 \rangle$.

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

Token	Pattern	Nome
Numeri	Costante numerica	256
Identificatore	Lettera seguita da lettere e cifre	257
Relop	Operatore relazionale (<,>,<=,>=,==,<>)	258
If	`if`	259
Else	`else`	260
While	`while`	261
Assegnamento	`:=`	262
Print	`print`	263
Read	`read`	264
Disgiunzione	`||`	265
Congiunzione	`&&`	266
Negazione	`!`	33
Più	`+`	43
Meno	`-`	45
Per	`*`	42
Divisione	`/`	47
Separatore	`;`	59
Parentesi tonda sinistra	`(`	40
Parentesi tonda destra	`)`	41
Parentesi graffa sinistra	{	123
Parentesi graffa destra	}	125
EOF	Fine dell'input	-1

Tabella 1: Descrizione dei token del linguaggio

- `5+)`
- `(34+26( - (2+15-( 27`
- `else 5 := print < if`

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 1, 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", cioé agli operatori relazionali), si può utilizzare il codice ASCII del carattere: ad esempio, il nome in Tabella 1 del segno più (+) è 43, il codice ASCII del +.

Listing 1: Classe `Tag`

```
public class Tag {
    public final static int
                EOF = -1,
                NUM = 256,
                ID = 257,
                RELOP = 258,
                IF = 259,
                ELSE = 260,
                WHILE = 261,
                ASSIGN = 262,
                PRINT = 263,
```

```
                READ = 264,
                OR = 265,
                AND = 266;
}
```

Definiamo una classe `Token` per rappresentare i token (una possibile implementazione della classe `Token` è in Listing 2). Definiamo inoltre la classe `Word` derivata da `Token`, per rappresentare i token che corrispondono agli identificatori, alle parole chiave e agli elementi della sintassi che consistono di più caratteri (ad esempio <= e &&). Una possibile implementazione della classe `Word` è in Listing 3. Ispirandosi alla classe `Word`, si può anche definire una classe `Number` per rappresentare i token che corrispondono ai numeri.

Listing 2: 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('!'),
            plus = new Token('+'),
            minus = new Token('-'),
            mult = new Token('*'),
            div = new Token('/'),
            semicolon = new Token(';'),
            lpt = new Token('('),
            rpt = new Token(')'),
            lpg = new Token('{'),
            rpg = new Token('}');
}
```

Listing 3: 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
        iftok = new Word(Tag.IF, "if"),
        elsetok = new Word(Tag.ELSE, "else"),
        whiletok = new Word(Tag.WHILE, "while"),
        assign = new Word(Tag.ASSIGN, ":="),
        print = new Word(Tag.PRINT, "print"),
        read = new Word(Tag.READ, "read"),
        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, ">=");

}
```

Una possibile struttura dell'analizzatore lessicale (ispirata al testo [1, Appendice A.3]) è descritta nella classe `Lexer` in Listing 4.

Listing 4: 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 la classe `Number` deve essere aggiunta e la classe `Lexer` deve essere completata.

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:

$$\Big([\texttt{a}-\texttt{zA}-\texttt{Z}] \mid (\_(\_)^*[\texttt{a}-\texttt{zA}-\texttt{Z0}-\texttt{9}])\Big)\Big([\texttt{a}-\texttt{zA}-\texttt{Z0}-\texttt{9}] \mid \_\Big)^* .$$

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` */
d:=300; // distanza
t:=10; // tempo
print(d*t)
```

L'output del programma per l'analisi lessicale sarà $\langle 257, \mathrm{d}\rangle\ \langle 262, :=\rangle\ \langle 256, 300\rangle\ \langle 59\rangle\ \langle 257, \mathrm{t}\rangle\ \langle 262, :=\rangle\ \langle 256, 10\rangle\ \langle 59\rangle\ \langle 263, \mathrm{print}\rangle\ \langle 40\rangle\ \langle 257, \mathrm{d}\rangle\ \langle 42\rangle\ \langle 257, \mathrm{t}\rangle\ \langle 41\rangle\ \langle -1\rangle$.

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 `d:=300; /*distanza`) deve essere segnalato un errore. Si noti che ci possono essere due commenti di seguito non separati da nessun token, ad esempio:

`d:=300; /*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 $\langle 257, \mathrm{x}\rangle\ \langle 42\rangle\ \langle 47\rangle\ \langle 257, \mathrm{y}\rangle\ \langle -1\rangle$). 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, 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 che segue:

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

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 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 5: 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.err.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();
            br.close();
        } catch (IOException e) {e.printStackTrace();}
    }
}
```

Esercizio 3.2 (opzionale)**.** Notiamo che le quattro produzioni

$$\begin{array}{rcl} \langle expr \rangle & ::= & \langle term \rangle \ \langle exprp \rangle \\ \\ \langle exprp \rangle & ::= & + \ \langle term \rangle \ \langle exprp \rangle \\ & | & - \ \langle term \rangle \ \langle exprp \rangle \\ & | & \varepsilon \end{array}$$

si possono riunire in una sola utilizzando (in modo ibrido) il linguaggio delle espressioni regolari:

$$\langle expr \rangle \quad ::= \quad \langle term \rangle ((+|-) \ \langle term \rangle)^*$$

Questo suggerisce di realizzare una sola procedura per $\langle expr \rangle$ mediante un'iterazione in cui si riconosce un termine e poi:

- se il simbolo in input è + o − si riconosce, iterativamente, un altro termine e così via;
- se il simbolo in input appartiene all'insieme guida di $\langle exprp \rangle := \varepsilon$ si esce dalla procedura.

Analoga considerazione si può fare per $\langle term \rangle$.

Modificare l'analizzatore sintattico sviluppato per l'Esercizio 3.1 in modo tale che:

- i metodi `expr` e `exprp` siano aggregati in un singolo metodo;
- i metodi `term` e `termp` siano aggregati in un singolo metodo.

Esercizio 3.3. Segue una grammatica per un semplice linguaggio di programmazione, dove i terminali della grammatica corrispondono ai token descritti in Sezione 2 (in Tabella 1).

$$\begin{array}{rcl} \langle prog \rangle & ::= & \langle statlist \rangle \ \texttt{EOF} \\ \\ \langle stat \rangle & ::= & \texttt{ID} \ \texttt{:=} \ \langle expr \rangle \\ & | & \texttt{print} \ \texttt{(} \ \langle expr \rangle \ \texttt{)} \\ & | & \texttt{read} \ \texttt{(} \ \texttt{ID} \ \texttt{)} \\ & | & \texttt{if} \ \texttt{(} \ \langle bexpr \rangle \ \texttt{)} \ \langle stat \rangle \\ & | & \texttt{if} \ \texttt{(} \ \langle bexpr \rangle \ \texttt{)} \ \langle stat \rangle \ \texttt{else} \ \langle stat \rangle \\ & | & \texttt{while} \ \texttt{(} \ \langle bexpr \rangle \ \texttt{)} \ \langle stat \rangle \\ & | & \texttt{\{} \ \langle statlist \rangle \ \texttt{\}} \\ \\ \langle statlist \rangle & ::= & \langle stat \rangle \ \langle statlist\_p \rangle \\ \\ \langle statlist\_p \rangle & ::= & \texttt{;} \ \langle stat \rangle \ \langle statlist\_p \rangle \\ & | & \varepsilon \\ \\ \langle bexpr \rangle & ::= & \langle expr \rangle \ \texttt{RELOP} \ \langle expr \rangle \\ \\ \langle expr \rangle & ::= & \langle term \rangle \ \langle exprp \rangle \\ \\ \langle exprp \rangle & ::= & + \ \langle term \rangle \ \langle exprp \rangle \\ & | & - \ \langle term \rangle \ \langle exprp \rangle \\ & | & \varepsilon \\ \\ \langle term \rangle & ::= & \langle fact \rangle \ \langle termp \rangle \\ \\ \langle termp \rangle & ::= & * \ \langle fact \rangle \ \langle termp \rangle \\ & | & / \ \langle fact \rangle \ \langle termp \rangle \\ & | & \varepsilon \\ \\ \langle fact \rangle & ::= & \texttt{(} \ \langle expr \rangle \ \texttt{)} \mid \texttt{NUM} \mid \texttt{ID} \end{array}$$

Si noti che `RELOP` corrisponde a un elemento dell'insieme {==, <>, <=, >=, <, >}.

Scrivere un analizzatore sintattico a discesa ricorsiva per la grammatica.

Esercizio 3.4 (opzionale)**.** La grammatica usata per la generazione delle espressioni aritmetiche nell'esercizio 3.1 è LL(1) ed è stata ottenuta eliminando le ricorsioni sinistre dalla seguente grammatica più intuitiva:

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

Si scriva un analizzatore sintattico bottom-up per la parsificazione delle espressioni aritmetiche basato su quest'ultima grammatica. Come nell'esercizio 3.1 il programma deve usare l'analizzatore lessicale già sviluppato e stampare un messaggio d'errore nel caso in cui l'input non sia una parola del linguaggio.

Suggerimento: in caso di difficoltà, in prima battuta si consiglia di prendere in considerazione una grammatica semplificata senza le due produzioni $\langle expr \rangle ::= \langle expr \rangle - \langle term \rangle$ e $\langle term \rangle ::= \langle term \rangle \ / \ \langle fact \rangle$.

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:

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

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 6).

Listing 6: 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('+');
```

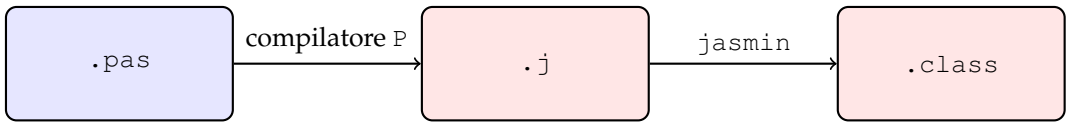
```
                term_val = term();
                exprp_val = exprp(exprp_i + term_val);
                break;

        // ... 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 di qui in avanti `P`, visto nell'esercizio 3.3 e nella parte teorica del corso. I file di programmi del linguaggio `P` hanno estensione `.pas`. Il traduttore deve generare bytecode [4] eseguibile dalla Java Virtual Machine (JVM).

Generare bytecode eseguibile direttamente dalla JVM non è una 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 viene 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`.

Esempi di compilazione. Seguono alcuni semplici programmi `P` affiancati dal bytecode JVM corrispondente.

Listing 7: `A.pas`

```
print (10 + 20 * 30)
```

Listing 8: Bytecode di `A.pas`

```
ldc 10
ldc 20
ldc 30
imul
iadd
invokestatic Output/print(I)V
```

Listing 9: `B.pas`

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

Listing 10: Bytecode di `B.pas`

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

Listing 11: `C.pas`

```
x := 10;
y := 20;
z := 30;
print (x + y * z)
```

Listing 12: Bytecode di `C.pas`

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

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`).

Listing 13: Una possibile implementazione della classe `OpCode`

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

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 14: 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 iadd : temp = " iadd " + "\n"; break;
            case invokestatic :
             if( operand == 1)
                temp = " invokestatic "+ "Output/print(I)V" + "\n";
             else
                 temp =" invokestatic " + "Output/read()I" + "\n"; break;
            case imul : temp = " imul " + "\n"; break;
            case idiv : temp = " idiv " + "\n"; break;
            case isub : temp = " isub " + "\n"; break;
            case istore : temp = " istore " + operand + "\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 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 15: Una possibile implementazione della classe `CodeGenerator`

```
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 occorre predisporre una *tabella dei simboli*. Segue una possibile implementazione della classe `SymbolTable`:

Listing 16: 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;
    }
}
```

Esercizio 5.1. Si scriva un traduttore per programmi scritti nel linguaggio `P` (utilizzando uno dei lexer sviluppati per gli esercizi di Sezione 2).

Un frammento del codice di una possibile implementazione (che riguarda la gestione di `print`, `read`, == e +) si trova 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

```
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 ...
                statlist();
                match(Tag.EOF);
                try {
                    code.toJasmin();
                }
                catch(java.io.IOException e) {
                    System.out.println("IO error\n");
                };
        // ... completare ...
    }

    public void stat() {
        switch(look.tag) {
        // ... completare ...
            case Tag.PRINT:
                match(Tag.PRINT);
                match('(');
                expr();
                code.emit(OpCode.invokestatic,1);
                match(')');
                break;
            case Tag.READ:
                match(Tag.READ);
                match('(');
                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);
                    match(')');
                    code.emit(OpCode.invokestatic,0);
                    code.emit(OpCode.istore,read_id_addr);
                }
                else
                    error("Error in grammar (stat) after read( with " + look);
                break;
        // ... completare ...
        }
```

```
    }

    // ... completare ...

    private void b_expr() {
        // ... completare ...
        expr();
        if (look == Word.eq) {
             match(Tag.RELOP);
             expr();
             int ltrue = code.newLabel();
             int lnext = code.newLabel();
             code.emit (OpCode.if_icmpeq,ltrue );
             code.emit (OpCode.ldc,0);
             code.emit (OpCode.GOto,lnext);
             code.emitLabel (ltrue);
             code.emit (OpCode.ldc,1);
             code.emitLabel (lnext);
        }
        // ... completare ...
    }

    // ... completare ...

    private void exprp() {
        switch(look.tag) {
            case '+':
                match('+');
                term();
                code.emit(OpCode.iadd);
                exprp();
                break;
        // ... completare ...
      }
   }
    // ... 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));
            Translator translator = new Translator(lex, br);
            translator.prog();
            br.close();
        } catch (IOException e) {e.printStackTrace();}
    }
}
```

Nota 1 : il comando indicato nella grammatica con `read(ID)` permette l'inserimento di un numero intero dalla tastiera e l'assegnamento del valore del numero all'identificatore scritto tra parentesi. Ad esempio, il comando `read(a)` specifica che l'utente del programma scritto nel linguaggio `P` deve inserire un numero intero con la tastiera, che poi è assegnato all'identificatore `a`.

Nota 2 : una precisazione per quanto riguarda l'implementazione suggerita in Listing 17 rispetto a quanto avete visto a lezione sulla *traduzione on-the-fly*. Il codice suggerito in questo documento non prevede l'utilizzo dei parametri che invece è suggerito negli schemi "on-the-fly"

discussi a lezione. Sentitevi liberi di sperimentare implementazioni alternative a quella proposta in Listing 17 che siano più aderenti allo schema "on-the-fly" presentato nella parte di teoria (ci riferiamo soprattutto all'implementazione proposta per il metodo `b_expr()`).

Riferimenti bibliografici

[1] Aho, Alfred V., Lam, Monica S., Sethi, Ravi, and Ullman, Jeffrey D. Compilatori: Principi, tecniche e strumenti. *Pearson Paravia Bruno Mondadori S.p.A.*, 2009.

[2] Assembly language. `http://en.wikipedia.org/wiki/Assembly_language` *Wikipedia*, 2016.

[3] Java class file. `http://en.wikipedia.org/wiki/Java_class_file` *Wikipedia*, 2016.

[4] Java bytecode. `http://en.wikipedia.org/wiki/Java_bytecode` *Wikipedia*, 2016.

[5] Java bytecode instruction listings. `http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings` *Wikipedia*, 2016.