Laboratorio di Linguaggi Formali e Traduttori
Corso di Studi in Informatica
A.A. 2015/2016

Ferruccio Damiani e Jeremy Sproston
Dipartimento di Informatica — Università degli Studi di Torino

Versione del 16 dicembre 2015

Sommario

Questo documento descrive le esercitazioni di laboratorio e le modalità d'esame del corso di *Linguaggi Formali e Traduttori* per l'A.A. 2015/2016.

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 entro la sessione d'esame successiva a quella in cui è stato 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).
- 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 2016.

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.

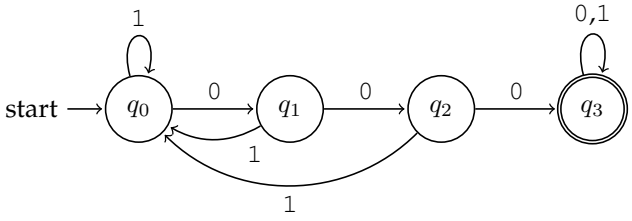


Figura 1: DFA che riconosce stringhe con 3 zeri consecutivi.

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 un DFA che riconosca il linguaggio delle costanti numeriche in virgola mobile. Esempi di tali costanti sono:

```
123    123.5    .567    +7.5    -.7    67e10    1e-2    -.7e2
```

Realizzare il DFA in Java seguendo la costruzione vista in Figura 2, assicurarsi che l'implementazione riconosca il linguaggio desiderato.

In base al particolare stato finale in cui si trova l'automa al termine del riconoscimento, cosa si può dire della costante numerica riconosciuta?

Esercizio 1.3. Modificare l'automa dell'esercizio precedente in modo che riconosca costanti numeriche precedute e/o seguite da sequenze eventualmente vuote di spazi. Modificare l'implementazione Java dell'automa conseguentemente.

Esercizio 1.4. 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 sottolineatura _ che non comincia con un numero e che non può essere composto solo da un _.

Esercizio 1.5. 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 1.6. Progettare e implementare un DFA con l'alfabeto $\{/, \star, a\}$ che riconosca il linguaggio di stringhe che contengono almeno 4 caratteri che iniziano con `/*`, che finiscono con `*/`, e che contengono una solo occorrenza della sequenza `*/`, quella finale. Verificare che il DFA accetti le stringhe `/****/`, `/*a*a*/`, `/*a/**/`, `/**a///a/a**/` e `/**/` ma non `/*/` oppure `/**/***/`.

Esercizio 1.7 (opzionale). Modificare l'automa dell'esercizio precedente in modo che riconosca il linguaggio di stringhe in cui una occorrenza della sequenza `/*` deve essere seguita (anche non immediatamente) da una occorrenza della sequenza `*/` che, a sua volta, deve essere seguita da una sequenza che consiste solo da simboli da $\{a, \star\}$. Le stringhe del linguaggio possono non avere nessuna occorrenza della sequenza `/*`. Ad esempio, il DFA deve accettare le stringe `aaa/****/aa`, `aa/*a*a*/`, `aaaa`, `/****/`, `/*aa*/`, `*/a` e `a/**/***a`, ma non `aaa/*/aa`, `a/**/***/a` oppure `aa/*aa`. Implementare l'automa seguendo la construzione vista in Figura 2.

Esercizio 1.8. Costruire il DFA equivalente al ϵ-NFA in Figura 3, e implementare il DFA seguendo la construzione vista in Figura 2.

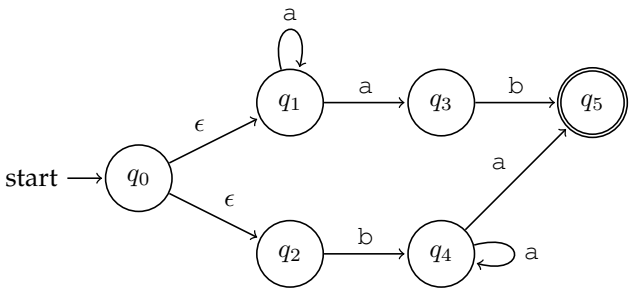


Figura 3: ϵ-NFA di Esercizio 1.8.

2 Analisi lessicale

Consideriamo un semplice linguaggio di programmazione, dove la sintassi è descritta dalla seguente grammatica:

$$
\begin{array}{rcl}
\langle prog \rangle & ::= & [\ \langle decl \rangle\ \texttt{;}\ ]^*\ \langle stmt \rangle\ \texttt{EOF} \\
\langle decl \rangle & ::= & \texttt{var}\ \texttt{ID}\ [\ \texttt{,}\ \texttt{ID}\ ]^*\ \texttt{:}\ \langle type \rangle \\
\langle type \rangle & ::= & \texttt{integer} \mid \texttt{boolean} \\
\langle expr \rangle & ::= & \langle andExpr \rangle\ [\ \texttt{||}\ \langle andExpr \rangle\ ]^* \\
\langle andExpr \rangle & ::= & \langle relExpr \rangle\ [\ \texttt{\&\&}\ \langle relExpr \rangle\ ]^* \\
\langle relExpr \rangle & ::= & \langle addExpr \rangle\ [\ \texttt{==}\ \langle addExpr \rangle \mid \texttt{<>}\ \langle addExpr \rangle \\
 & & \quad \mid \texttt{<=}\ \langle addExpr \rangle \mid \texttt{>=}\ \langle addExpr \rangle \\
 & & \quad \mid \texttt{<}\ \langle addExpr \rangle \mid \texttt{>}\ \langle addExpr \rangle\ ]^? \\
\langle addExpr \rangle & ::= & \langle mulExpr \rangle\ [\ \texttt{+}\ \langle mulExpr \rangle \mid \texttt{-}\ \langle mulExpr \rangle\ ]^* \\
\langle mulExpr \rangle & ::= & \langle unExpr \rangle\ [\ \texttt{*}\ \langle unExpr \rangle \mid \texttt{/}\ \langle unExpr \rangle\ ]^* \\
\langle unExpr \rangle & ::= & \texttt{not}\ \langle unExpr \rangle \mid \langle primary \rangle \\
\langle primary \rangle & ::= & \texttt{(}\ \langle expr \rangle\ \texttt{)} \mid \texttt{ID} \mid \texttt{NUM} \mid \texttt{true} \mid \texttt{false} \\
\langle stmt \rangle & ::= & \texttt{ID}\ \texttt{:=}\ \langle expr \rangle \\
 & \mid & \texttt{print}\ \texttt{(}\ \langle expr \rangle\ \texttt{)} \\
 & \mid & \langle stmt \rangle\ [\ \texttt{;}\ \langle stmt \rangle\ ]^*
\end{array}
$$

I terminali `ID` corrispondono all'espressione regolare (a..z|A..Z)(a..z|A..Z|0..9)*, e i terminali `NUM` corrispondono all'espressione regolare 0..9(0..9)*.

Il linguaggio permette di scrivere programmi costituiti da due sezioni (la prima delle quali può essere omessa):

1. Dichiarazioni di variabili, dove l'elenco di variabili è preceduto dalla parola chiave `var` e seguito da `:` e un tipo, `integer` oppure `boolean`. Esempi:
   - `var x,y:integer;`
   - `var z:boolean;`

2. Sequenze di comandi, con due tipologie distinte di comando.

   - Assegnamento di un valore valori ad una variabile, usando la sintassi `:=`. Esempi: `x:=5;`, `y:=x+7;` e `z:=x&&y;`.
   - Scrittura sul terminale, usando la parola chiave `print`. Esempi: `print(x)`, `print(x||y)` e `print((5+2)*3)`.

Le espressioni che possiamo scrivere in commandi di assegnamento e `print` sono 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 `)`, operatori relazionali `<`, `<=`, `==`, `>=`, `>`, e `<>`, operatori logici `&&`, `||` e `not`. Esempi:

- `34+26-5`
- `(34+26)-5`
- `x == (y+26)*5`
- `x && y == false`
- `x <= y+10`
- `not x || y == true`

Ad esempio, il seguente programma calcola la velocita media di un viaggio dati il numero di metri percorsi `s` e i secondi impiegati `t`:

```
var s, t: integer;
var datiOK: boolean;

s:=300000000;
t:=1;

datiOK:= (s >=0) && (t > 0);
print(datiOK);

print(s/t)
```

Esercizio 2.1. Si scriva in Java un analizzatore lessicale che legga da tastiera comandi scritti in questo linguaggio e per ciascuna comando stampi una sequenza di token.

- I token che corrispondono a numeri prenderanno la forma $\langle \text{NUM}, \text{valore} \rangle$, dove valore è un intero non negativo. Ad esempio, il token che corrisponde al numero 12 sarà $\langle \text{NUM}, 12 \rangle$
- I token che corrispondono agli identificatori prenderanno la forma $\langle \text{ID}, \text{“lessema”} \rangle$. Ad esempio, il token che corrisponde a `x` sarà $\langle \text{ID}, \text{“x”} \rangle$, e il token che corrisponde a `temp` sarà $\langle \text{ID}, \text{“temp”} \rangle$.
- I token che corrispondono agli elementi della sintassi che consistono di un solo carattere (ad esempio, `'('`, `'+'` e `':'`) prenderanno la forma $\langle \text{nome} \rangle$, dove il nome è il codice ASCII del carattere. Ad esempio, il token che corrisponde a `'('` sarà $\langle ' (' \rangle$, e il token che corrisponde a `':'` sarà $\langle ' : ' \rangle$.
- I token che corrispondono agli elementi della sintassi che consistono di più caratteri (ad esempio, `&&`, `<>`, `print`, `integer` e `:=`) prenderanno la forma $\langle \text{nome}, \text{“lessema”} \rangle$. Ad esempio, il token che corrisponde a `:=` sarà $\langle \text{ASSIGN}, \text{“:=”} \rangle$, e il token che corrisponde a `print` sarà $\langle \text{PRINT}, \text{“print”} \rangle$.

Definiamo una classe `Tag` in Listing 1, utilizzando un insieme opportuno di costanti intere per rappresentare il nome dei token. (Si nota che non è assolutemente necessario definire tali costanti per tutti i token: per quelli che corrispondono a un solo carattere, si può utilizzare il codice ASCII del carattere.)

Listing 1: Classe `Tag`

```
public class Tag {
    public final static int
            EOF = -1,
            NUM = 256,
            ID = 257,
            AND = 258,
            OR = 259,
            VAR = 260,
            INTEGER = 261,
            BOOLEAN = 262,
            ASSIGN = 263,
            EQ = 264,
            GE = 265,
            LE = 266,
            NE = 267,
            TRUE = 268,
            FALSE = 269,
            NOT = 270,
            PRINT = 271;
}
```

Questa scelta ha la conseguenza che l'output del nostro programma sarà della forma $\langle 271, \text{“print”}\rangle$ $\langle 40\rangle$ $\langle 257, \text{“x”}\rangle$ $\langle 259, \text{“||”}\rangle$ $\langle 257, \text{“y”}\rangle$ $\langle 41\rangle$ $\langle 59\rangle$ per il comando `print(x||y);` anzichè $\langle \text{PRINT}, \text{“print”}\rangle$ $\langle '('\rangle$ $\langle \text{ID}, \text{“x”}\rangle$ $\langle \text{OR}, \text{“||”}\rangle$ $\langle \text{ID}, \text{“y”}\rangle$ $\langle ')'\rangle$ $\langle ';'\rangle$

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

- `5+)`
- `(34+26( - (2+15-( 27`
- `var 5 := print < boolean`

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 `@`. Per semplicità, si può utilizzare un carattere particolare, come `$`, per segnalare la fine del input (e quindi produrre un token con nome EOF che corrisponde alla fine del input).

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

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
            comma = new Token(','),
            colon = new Token(':'),
```

```
            semicolon = new Token(';'),
            lpar = new Token('('),
            rpar = new Token(')'),
            plus = new Token('+'),
            minus = new Token('-'),
            mult = new Token('*'),
            div = new Token('/'),
            lt = new Token('<'),
            gt = 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
            and = new Word(Tag.AND, "&&"),
            or = new Word(Tag.OR, "||"),
            eq = new Word(Tag.EQ, "=="),
            le = new Word(Tag.LE, "<="),
            ne = new Word(Tag.NE, "<>"),
            ge = new Word(Tag.GE, ">="),
            assign = new Word(Tag.ASSIGN, ":=");
}
```

Gestione delle parole chiave. Abbiamo bisogno di un meccanismo per memorizzare le parole chiave (`var`, `integer`, `boolean`, `not`, `true`, `false` e `print`), in modo tale che possiamo distinguirle da eventuali identificatori. Un modo è di utilizzare una *tabella hash*. La classe `Hashtable` è utilizzata per immagazzinare coppie che consistono di una *chiave* (*key*) e un *valore*. La classe `Hashtable` mette a disposizione due metodi, `put` e `get`, che servono rispettivamente per:

1. inserire associazioni (chiave, valore) nella tabella;
2. recuperare il valore associato a una chiave dalla tabella.

Nel contesto di nostra implementazione di un'analizzatore lessicale, possiamo utilizzare la classe `Hashtable` nel modo seguente.

```
Hashtable<String,Word> words = new Hashtable<String,Word>();
void reserve(Word w) { words.put(w.lexeme, w); }
...
reserve(new Word(Tag.VAR, "var"));
...
Word w = (Word)words.get(s);
...
```

In questo frammento di codice, l'operazione `words.put(w.lexeme, w)` è utilizzata per associare la chiave `w.lexeme` al valore `w`. Quindi la chiamata del metodo `reserve` corrisponde all'associazione della chiave `"var"` al oggetto creato da `new Word(Tag.VAR, "var")`.

Dato una stringa `s`, `(Word)words.get(s)` corrisponde al oggetto del tipo `Word` che è associato alla chiave `s` nella tabella hash `words`. Se la chiave `s` non è stata inserita nella tabella, si otterrà come risultato `null`.

La tabella hash può essere utilizzata anche per memorizzare gli identificatori già visti durante l'analisi lessicale.

Classe `Lexer`. Una possibile struttura del programma (ispirato al testo [1, Appendice A.3]) è la seguente:

Listing 4: Analizzatore lessicale di comandi semplici

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

public class Lexer {

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

    Hashtable<String,Word> words = new Hashtable<String,Word>();
    void reserve(Word w) { words.put(w.lexeme, w); }
    public Lexer() {
        reserve( new Word(Tag.VAR, "var"));

                        // ... gestire le altre parole chiave ... //

    }

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

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

        switch (peek) {
            case ',':
                peek = ' ';
                return Token.comma;

                        // ... gestire gli altri casi ... //

            case '&':
                readch();
                if (peek == '&') {
                    peek = ' ';
                    return Word.and;
                } else {
                    System.err.println("Erroneous character"
                            + " after & : "  + peek );
                    return null;
                }

                        // ... gestire gli altri casi ... //

            default:
                if (Character.isLetter(peek)) {
                    String s = "";
```

```
                    do {
                        s+= peek;
                        readch();
                    } while (Character.isDigit(peek) ||
                            Character.isLetter(peek));
                    if ((Word)words.get(s) != null)
                        return (Word)words.get(s);
                    else {
                    Word w = new Word(Tag.ID,s);
                    words.put(s, w);
                    return w;
                    }
                } else {

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

                        if (peek == '$') {
                            return new Token(Tag.EOF);
                        } else {
                        System.err.println("Erroneous character: "
                                + peek );
                        return null;
                        }
                }
            }
    }

    public static void main(String[] args) {
        Lexer lex = new Lexer();

        Token tok;
        do {
            tok = lex.lexical_scan();
            System.out.println("Scan: " + tok);
        } while (tok.tag != Tag.EOF);
    }

}
```

Esercizio 2.2. Consideriamo la seguente versione modificata della produzione per $\langle stmt \rangle$:

$$
\begin{array}{rcl}
\langle stmt \rangle & ::= & \texttt{ID} \texttt{ :=} \langle expr \rangle \\
& | & \texttt{print (} \langle expr \rangle \texttt{)} \\
& | & \texttt{if} \langle expr \rangle \texttt{then} \langle stmt \rangle [\texttt{else} \langle stmt \rangle]^? \\
& | & \texttt{while} \langle expr \rangle \texttt{do} \langle stmt \rangle \\
& | & \texttt{begin} \langle stmt \rangle [\texttt{;} \langle stmt \rangle]^* \texttt{end}
\end{array}
$$

Estendere l'implementazione del analizzatore lessicale per gestire comandi di programmi scritti nel linguaggio modificato (cioè nel linguaggio con la versione modificata della produzione per $\langle stmt \rangle$).

Esercizio 2.3. Consideriamo la seguente nuova definizione di identificatori (`ID`): un identificatore è composto da una sequenza non vuota di lettere, numeri, ed il simbolo di sottolineatura _ che non comincia con un numero e che non può essere composto solo da _. Più precisamente, i terminali `ID` corrispondono all'espressione regolare $((\texttt{a..z}|\texttt{A..Z})|(\_(\_)^*(\texttt{a..z}|\texttt{A..Z}|\texttt{0..9})))(\texttt{a..z}|\texttt{A..Z}|\texttt{0..9}|\_)^*$. Estendere il metodo `lexer_scan` per gestire identificatori che corrispondono alla nuova definizione. **Suggerimento:** utilizzare la soluzione del Esercizio 1.4 come punto d'inizio.

Lettura da un file. La lettura di un programma da un file, anzichè dalla tastiera come in Listing 4, può essere realizzata nel modo illustrato in Listing 5. Il metodo `main` crea un oggetto della classe `BufferedReader`, che poi è passato come parametro al metodo `lexical_scan`, e a sua volta a `readch`.

Listing 5: Lettura da un file

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

public Token lexical_scan(BufferedReader br) {
            // ... sostituire readch() con readch(br) ... //
            // ... //
            if (peek == (char)-1) {   // sostituisce if (peek == '$')
                  return new Token(Tag.EOF);
            }
            // ... //
}

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();}
}
```

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:

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

Il programma deve fare uso dell'analizzatore lessicale sviluppato in precedenza. Si nota che l'insieme di token corrispondente alla grammatica di questa sezione è un sottoinsieme dell'insieme di token corrispondente alla grammatica 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]). Consideriamo due varianti: nella prima (in Listing 6), l'input è letto dalla tastiera; invece nella seconda (in Listing 7), l'input è letto da un file.

Listing 6: Analizzatore sintattico di espressioni semplici: lettura dalla tastiera

```
import java.io.*;

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

    public Parser(Lexer l) {
        lex = l;
        move();
    }

    void move() {
        look = lex.lexical_scan();
        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() {  // la procedura start puo' essere estesa (opzionale)
        expr();
        match(Tag.EOF);
```

```
    }

    private void expr() {  // la procedura expr puo' essere estesa (opzionale)
        term();
        exprp();
    }

    private void exprp() {
        switch (look.tag) {
        case '+':
            match('+');
            term();
            exprp();
            break;

        case '-':
            // ... gestire gli altri casi ... //
        }
    }

    private void term() {
        // ... riempire ... //
    }

    private void termp() {
        switch (look.tag) {
        case '*':
            match('*');
            fact();
            termp();
            break;

        // ... gestire gli altri casi ... //
    }

    private void fact() {
        switch (look.tag) {
            // ... gestire tutti i casi ... //
        }
    }

    public static void main(String[] args) {
        Lexer lex = new Lexer();
        Parser parser = new Parser(lex);
        parser.start();
    }
}
```

Listing 7: Analizzatore sintattico di espressioni semplici: lettura da un file (`Lexer_extended` è la versione estesa di `Lexer` per gestire lettura da un file, descritto in Sezione 2)

```
import java.io.*;

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

    public Parser(Lexer_extended l, BufferedReader br) {
```

```
        lex = l;
        pbr = br;
        move();
    }

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

            // ... come per il caso della lettura dalla tastiera ... //

    public static void main(String[] args) {
        Lexer_extended lex = new Lexer_extended();
        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();}
    }
}
```

4 Traduzione diretta dalla sintassi

Esercizio 4.1 (Valutatore di espressioni semplici)**.** Modificare l'analizzatore sintattico di 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 nota che un indice (cioè 1) è usato per distinguere due diverse occorrenze dello stesso non-terminale (ad esempio, $\langle exprp \rangle$) nella stessa produzione. Inoltre, si nota 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 consigliato 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 Token look;
    // se l'input e' letto da un file, creare un campo del tipo BufferedReader,
    // come in Esercizio 3.1

    public Valutatore(Lexer l) {
        lex = l;
        move();
        // se l'input e' letto da un file, creare un oggetto del tipo
        // BufferedReader, come in Esercizio 3.1
    }

    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;

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

        System.out.println(expr_val);
    }
        // la procedura start puo' essere estesa
        // come in Esercizio 3.1 (opzionale)

    private int expr() {
        int term_val, exprp_val;

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

        return exprp_val;
    }
        // la procedura expr puo' essere estesa
        // come in Esercizio 3.1 (opzionale)

    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;

        case '-':
            // ... completare ... //
    }

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

    private int termp(int termp_i) {
        int fact_val, termp_val;

        switch (look.tag) {
        case '*':
            match('*');
            fact_val = fact();
            termp_val = termp(termp_i * fact_val);
            break;

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

    private int fact() {
        int fact_val;

        switch (look.tag) {
        case Tag.NUM:
            fact_val = // ... completare ... //
        }

        return fact_val;
    }
}
```

Il main per eseguire il tutto è simile a quelli dei Listing 6 e Listing 7.

Esercizio 4.2 (opzionale). (Type checker). Si scriva un programma per controllare i tipi di un'espressione aritmetica-logico semplice (cioè per segnalare se l'espressione contiene un'incompatibilità rispetto ai tipi delle sue sotto-espressioni), facendo riferimento alla grammatica seguente:

$$
\begin{array}{rcl}
\langle start \rangle & ::= & \langle andExpr \rangle \texttt{ EOF} \\
\langle andExpr \rangle & ::= & \langle andTerm \rangle \ \langle andExprp \rangle \\
\\
\langle andExprp \rangle & ::= & \texttt{\&\&} \ \langle andTerm \rangle \ \langle andExprp \rangle \\
& | & \varepsilon \\
\\
\langle andTerm \rangle & ::= & \langle sumExpr \rangle \ \langle andTermp \rangle \\
\\
\langle andTermp \rangle & ::= & == \langle sumExpr \rangle \\
& | & \varepsilon \\
\\
\langle sumExpr \rangle & ::= & \langle sumTerm \rangle \ \langle sumExprp \rangle \\
\\
\langle sumExprp \rangle & ::= & + \ \langle sumTerm \rangle \ \langle sumExprp \rangle \\
& | & \varepsilon \\
\\
\langle sumTerm \rangle & ::= & \texttt{(} \ \langle andExpr \rangle \ \texttt{)} \\
& | & \texttt{NUM} \\
& | & \texttt{true} \\
& | & \texttt{false}
\end{array}
$$

Più precisemente, dato un'espressione della grammatica di input, il programma deve stampare sul terminale:

- `Numerical` se il valore l'espressione è numerico (ad esempio, per le espressioni `3`, `1+2` oppure `((1+2)+(4+3))`),
- `Boolean` se il valore l'espressione è Booleano (ad esempio, per le espressioni `true`, `false`, `true && false`, `true == false`, `(3+4)==7` oppure `(3==7) && true`).

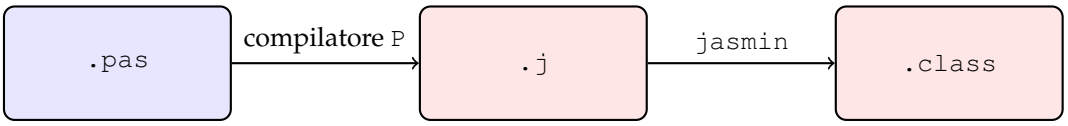
Invece se l'espressione contiene un'incompatibilità rispetto ai tipi delle sue sotto-espressioni (ad esempio, `true + 6`, `3 && 4`, `true==7`, `7 && false` oppure `(3+ true) && (2==(1+1))`), un errore deve essere segnalato, indicando la procedura in esecuzione quando l'errore è stato individuato. **Suggerimento**: prima di implementare il programma, scrivere uno schema di traduzione diretto dalla sintassi adatto al problema.

5 Generazione del bytecode

L'obiettivo di quest'ultima parte di laboratorio è di quello di realizzare un traduttore per i programmi (ben tipati) scritti nel linguaggio di programmazione semplice, che si chiama `P`, visto nella parte teorica del corso. I file di programmi del linguaggio `P` hanno l'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 Sezione 5.2, 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`.

5.1 Esempi di compilazione

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

Listing 9: `A.pas`

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

Listing 10: Bytecode di `A.pas`

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

Listing 11: `B.pas`

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

Listing 12: Bytecode di `B.pas`

```
  ldc 10
  ldc 20
  ldc 30
  imul
  if_icmplt L0
  ldc 0
  goto L1
L0:
  ldc 1
L1:
  invokestatic Output/printBool(I)V
```

Listing 13: `C.pas`

```
integer x, y, z;
begin
  x := 10;
  y := 20;
  z := 30;
  print (x + y * z)
end
```

Listing 14: 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/printInt(I)V
```

5.2 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 delle espressioni aritmetiche/booleane).

Listing 15: 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 , 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 16: 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/printInt(I)V" + "\n";
               else
                   temp =" invokestatic " + "Output/printBool(I)V" + "\n";break;
            case imul : temp = " imul " + "\n"; break;
            case idiv : temp = " idiv " + "\n"; break;
            case ineg : temp = " ineg " + "\n"; break;
            case isub : temp = " isub " + "\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 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 17: 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 printBool(I)V\n"
```

```
        + " .limit stack 3\n"
        + " getstatic java/lang/System/out Ljava/io/PrintStream;\n"
        + " iload_0 \n"
        + " bipush 1\n"
        + " if_icmpeq Ltrue\n"
        + " ldc \"false\"\n"
        + " goto Lnext\n"
        + " Ltrue:\n"
        + " ldc \"true\"\n"
        + " Lnext:\n"
        + " invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V\n"
        + " return\n"
        + ".end method\n"
        + "\n"
        + ".method public static printInt(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 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";
}
```

Notare che le espressioni logiche possono essere tradotte in modo analogo alle espressioni aritmetiche, rappresentando il valore `false` con 0 ed il valore `true` con 1.

Per tenere traccia degli identificatori, che in base alla grammatica del linguaggio `P` possono essere dichiarati solo all'*inizio* del programma, occorre predisporre una *tabella dei simboli*. Per effettuare il controllo di tipo, si introduce una enumerazione per distinguere i due tipi di dato dei programmi `P`:

Listing 18: Rappresentazione dei tipi `P`

```
public enum Type { INTEGER, BOOLEAN }
```

Segue una possibile implementazione della classe `SymbolTable`:

Listing 19: Una possibile implementazione della classe `SymbolTable`

```
public class SymbolTable {
     Map <String, Type> TypeMap = new HashMap <String, Type>();
     Map <String, Integer> OffsetMap = new HashMap <String,Integer>();

     public void insert( String s, Type t, int address ) {
        if( !TypeMap.containsKey(s) ) TypeMap.put(s,t);
        else throw new IllegalArgumentException("Variabile gia' dichiarata.");
        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 Type lookupType ( String s) {
       if( TypeMap.containsKey(s) ) return TypeMap.get(s);
       throw new IllegalArgumentException("Variabile sconosciuta ." + s );
    }

    public int lookupAddress ( String s ) {
       if( OffsetMap.containsKey(s) ) return OffsetMap.get(s);
       throw new IllegalArgumentException("Variabile sconosciuta.");
    }
}
```

5.3 Esercizi

Negli esercizi seguente, possiamo utilizzare il lexer sviluppato in Esercizio 2.2.

Esercizio 5.1. (Parte principale.) Si scriva un traduttore per programmi ben tipati scritti nel frammento del linguaggio `P` che permette di stampare sul terminale il valore di un'espressione aritmetica-logico. Il frammento del linguaggio `P` è descritto dalla grammatica seguente:

$$
\begin{array}{rcl}
\langle prog \rangle & ::= & \texttt{print (} \langle orE \rangle \texttt{) EOF} \\
\langle orE \rangle & ::= & \langle andE \rangle \, \langle orE\_p \rangle \\
\langle orE\_p \rangle & ::= & \texttt{||} \, \langle andE \rangle \, \langle orE\_p \rangle \\
 & | & \varepsilon \\
\langle andE \rangle & ::= & \langle relE \rangle \, \langle andE\_p \rangle \\
\langle andE\_p \rangle & ::= & \texttt{\&\&} \, \langle relE \rangle \, \langle andE\_p \rangle \\
 & | & \varepsilon \\
\langle relE \rangle & ::= & \langle addE \rangle \, \langle relE\_p \rangle \\
\langle relE\_p \rangle & ::= & == \langle addE \rangle \mid <> \langle addE \rangle \mid <= \langle addE \rangle \\
 & | & >= \langle addE \rangle \mid < \langle addE \rangle \mid > \langle addE \rangle \\
 & | & \varepsilon \\
\langle addE \rangle & ::= & \langle multE \rangle \, \langle addE\_p \rangle \\
\langle addE\_p \rangle & ::= & + \, \langle multE \rangle \, \langle addE\_p \rangle \\
 & | & - \, \langle multE \rangle \, \langle addE\_p \rangle \\
 & | & \varepsilon \\
\langle multE \rangle & ::= & \langle fact \rangle \, \langle multE\_p \rangle \\
\langle multE\_p \rangle & ::= & * \, \langle fact \rangle \, \langle multE\_p \rangle \\
 & | & / \, \langle fact \rangle \, \langle multE\_p \rangle \\
 & | & \varepsilon \\
\langle fact \rangle & ::= & \texttt{(} \, \langle orE \rangle \, \texttt{)} \mid \texttt{NUM} \mid \texttt{true} \mid \texttt{false}
\end{array}
$$

Un frammento del codice di una possibile implementazione (che riguarda la gestione di == e +) può essere trovato in Listing 20 (si nota che `code` è un oggetto della classe `CodeGenerator`).

Listing 20: Un frammento del codice di una possibile implementazione del programma di Esercizio 5.1

```
// ...
private Type relE_p( /* ... eventuali parametri ... */ ) {
   // ...
   switch(look.tag) {
      case Tag.EQ:
         match(Tag.EQ);
         addE_type = addE();
         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);
         // ... type checking ...
         break;
   // ... altri casi ...
   }
}
// ...
private Type addE_p( /* ... eventuali parametri ... */ ) {
// ...
   switch(look.tag) {
      case '+':
      match('+');
      multE_type = multE();
      code.emit(OpCode.iadd);
      // ... type checking ...
      break;
   // ... altri casi ...
   }
}
// ...
```

(Parte opzionale.) Prendiamo in considerazione la seguente modifica alla grammatica:

$$\begin{array}{rcl} \langle relE\_p \rangle & ::= & \langle oprel \rangle \ \langle addE \rangle \\ & | & \varepsilon \end{array}$$

$$\langle oprel \rangle \quad ::= \quad == \mid <> \mid <= \mid >= \mid < \mid >$$

Modificare il programma del esercizio precedente in modo da tradurre programmi della versione nuova dalla grammatica, evitando di duplicare codice il più possibile nella procedura associato con il non-terminale $\langle relE\_p \rangle$.

Esercizio 5.2. Modificare il programma del esercizio precedente in modo da tradurre programmi ben tipati scritti nel frammento del linguaggio `P` che permette non solo il comando `print` e le espressione aritmetiche-logici ma anche l'utilizzo di variabili. Più precisamente, il frammento permette (1) la dichiarazione di variabili, (2) l'assegnazione di valori di espressioni a variabili, e (3) l'utilizzo di variabili in espressioni. Il frammento del linguaggio `P` è descritto dalla grammatica seguente (si nota che la grammatica è incompleta perché mancano le produzioni per $\langle orE \rangle$,

come spiegato nei commenti dopo la grammatica):

$$
\begin{array}{rcl}
\langle prog \rangle & ::= & \langle declist \rangle\ \langle stat \rangle\ \texttt{EOF} \\
\langle declist \rangle & ::= & \langle dec \rangle\ \texttt{;}\ \langle declist \rangle \\
& | & \varepsilon \\
\\
\langle dec \rangle & ::= & \langle type \rangle\ \texttt{ID}\ \langle idlist \rangle \\
\langle idlist \rangle & ::= & \texttt{,}\ \texttt{ID}\ \langle idlist \rangle \\
& | & \varepsilon \\
\\
\langle type \rangle & ::= & \texttt{integer} \\
& | & \texttt{boolean} \\
\\
\langle stat \rangle & ::= & \texttt{ID :=}\ \langle orE \rangle \\
& | & \texttt{print (}\ \langle orE \rangle\ \texttt{)} \\
& | & \texttt{begin}\ \langle statlist \rangle\ \texttt{end} \\
\\
\langle statlist \rangle & ::= & \langle stat \rangle\ \langle statlist\_p \rangle \\
\\
\langle statlist\_p \rangle & ::= & \texttt{;}\ \langle stat \rangle\ \langle statlist\_p \rangle \\
& | & \varepsilon
\end{array}
$$

Per il non-terminale $\langle orE \rangle$, utilizziamo le produzioni già presentate in Esercizio 5.1, tranne quelle per $\langle fact \rangle$, in cui aggiungiamo la produzione $\langle fact \rangle ::= \texttt{ID}$. Più precisamente, le produzioni associate con il non-terminale $\langle fact \rangle$ sono descritte nel modo seguente:

$$\langle fact \rangle \quad ::= \quad \texttt{(}\ \langle orE \rangle\ \texttt{)} \mid \texttt{ID} \mid \texttt{NUM} \mid \texttt{true} \mid \texttt{false}$$

Esercizio 5.3. Modificare il programma del esercizio precedente in modo da tradurre programmi ben tipati scritti nel linguaggio `P`, cioè il linguaggio completo: rispetto al linguaggio preso in considerazione in Esercizio 5.2, il linguaggio di questo esercizio permette cicli di esecuzione, utilizzando il comando `while ... do`, e comandi condizionali, utilizzando il comando `if ... then ... else`. Più precisamente, modifichiamo la grammatica del Esercizio 5.2 nel modo seguente (si nota l'introduzione del non-terminale $\langle exp \rangle$, che è utilizzato nella grammatica completa di `P` per rappresentare un'espressione aritmetica-booleano):

$$
\begin{array}{rcl}
\langle stat \rangle & ::= & \texttt{ID :=}\ \langle exp \rangle \\
& | & \texttt{print (}\ \langle exp \rangle\ \texttt{)} \\
& | & \texttt{begin}\ \langle statlist \rangle\ \texttt{end} \\
& | & \texttt{while}\ \langle exp \rangle\ \texttt{do}\ \langle stat \rangle \\
& | & \texttt{if}\ \langle exp \rangle\ \texttt{then}\ \langle stat \rangle \\
& | & \texttt{if}\ \langle exp \rangle\ \texttt{then}\ \langle stat \rangle\ \texttt{else}\ \langle stat \rangle \\
\\
\langle exp \rangle & ::= & \langle orE \rangle \\
\\
\langle fact \rangle & ::= & \texttt{(}\ \langle exp \rangle\ \texttt{)} \mid \texttt{ID} \mid \texttt{NUM} \mid \texttt{true} \mid \texttt{false}
\end{array}
$$

Per tutti gli altri non-terminali, utilizziamo le produzioni già presentate in Esercizio 5.1 e Esercizio 5.2.

Esercizio 5.4 (opzionale). Alcune espressioni logiche hanno un valore determinato pur non essendo noto il valore di *tutte* le loro sottoespressioni. Per esempio, l'espressione

$$\texttt{false}\ \texttt{\&\&}\ E$$

è falsa a prescindere dal valore di E, dal momento che `false` è assorbente per la congiunzione logica. In modo duale, l'espressione

$$\texttt{true}\ \texttt{||}\ E$$

è vera a prescindere dal valore di E.

Seguendo queste considerazioni, è possibile tradurre una espressione logica in modo tale che la sua valutazione si arresti non appena ne è noto il valore. Questo particolare schema di traduzione prende talvolta il nome di "valutazione corto-circuitata". A titolo di esempio, il programma qui sotto contiene un'espressione la cui valutazione completa darebbe luogo a un errore di divisione per zero. Grazie alla valutazione corto-circuitata, tuttavia, esso può essere compilato nella sequenza di istruzioni mostrata a lato, che consente la terminazione con successo del programma:

```
print ( 10 == 20 && 30 / 0 > 15 )
```

```
  ldc 10
  ldc 20
  if_icmpeq L0
  ldc 0
  goto L1
L0:
  ldc 1
L1:
  ldc 0
  if_icmpeq L2
  ldc 30
  ldc 0
  idiv
  ldc 15
  if_icmpgt L3
  ldc 0
  goto L4
L3:
  ldc 1
L4:
  goto L5
L2:
  ldc 0
L5:
  invokestatic And/printBool(I)V
```

In particolare, terminata la valutazione della sottoespressione `10 == 20` (corrispondente alle istruzioni fino all'etichetta `L1`), il bytecode controlla se la sottoespressione è falsa e, in tal caso, salta all'etichetta `L2` scavalcando interamente la valutazione della sottoespressione `30 / 0 > 15` (corrispondente alle istruzioni da `L1` a `L4`).

Modificare il compilatore in modo da effettuare la valutazione corto-circuitata degli operatori `&&` e `||`.

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*, 2015.

[3] Java class file. `http://en.wikipedia.org/wiki/Java_class_file` *Wikipedia*, 2015.

[4] Java bytecode. `http://en.wikipedia.org/wiki/Java_bytecode` *Wikipedia*, 2015.

[5] Java bytecode instruction listings. `http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings` *Wikipedia*, 2015.