Laboratorio di Linguaggi Formali e Traduttori
Corso di Studi in Informatica
a.a. 2013/2014

Luca Padovani e Jeremy Sproston
Dipartimento di Informatica — Università degli Studi di Torino
c.so Svizzera, 185 — I-10149 Torino (Italy)

Versione del 27 aprile 2014

Sommario

Queste pagine descrivono il laboratorio per il corso di *Linguaggi Formali e Traduttori* per l'anno accademico 2013/2014.

Modalità di svolgimento e consegna del 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. Per ognuno dei forum disponibili è disponibile un canale *RSS* che riporta gli ultimi dieci post effettuati.

Modalità dell'esame di laboratorio

L'esame è *orale* e *individuale* anche se il laboratorio è svolto in collaborazione con altri studenti. Per sostenere l'esame a un appello è obbligatorio prenotarsi. L'esame ha l'obiettivo di verificare il corretto svolgimento della prova di laboratorio, la comprensione della sua struttura e del suo funzionamento, nonché 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.
- La presentazione di codice "funzionante" non è condizione necessaria né sufficiente per il superamento della prova di laboratorio. In altri termini, è possibile essere promossi presentando codice non funzionante (se relativamente completo e se lo studente dimostra di aver acquisito le conoscenze richieste) così come è possibile essere respinti presentando codice funzionante (se lo studente dimostra di avere scarsa o nulla 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.

L'ultima sessione utile per sostenere la prova orale di laboratorio sarà la sessione di febbraio-marzo 2015.

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

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

La lode è attribuita nel caso si sia mostrata particolare brillantezza durante la parte di teoria e il laboratorio.

Validità del presente testo di laboratorio

Il presente testo di labortorio è valido sino alla sessione di febbraio-marzo 2015.

I Valutatore di espressioni semplici

I.1 Analizzatore lessicale

Consideriamo un linguaggio (sorgente) per specificare 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 `)`. Esempi:

a) $34 + 26 - 5$

b) $(34 + 26) - 5$

d) $(34 + 26) * 5$

Si scriva in Java un analizzatore lessicale che legga da tastiera queste espressioni e per ciascuna espressione stampi una sequenza di coppie $\langle$token, "lessema"$\rangle$. Per esempio, per le prime due espressioni elencate sopra si devono ottenere le sequenze:

- $\langle$NUM, "34"$\rangle$ $\langle$PLUS, "+"$\rangle$ $\langle$NUM, "26"$\rangle$ $\langle$MINUS, "−"$\rangle$ $\langle$NUM, "5"$\rangle$
- $\langle$LPAR, "("$\rangle$ $\langle$NUM, "34"$\rangle$ $\langle$PLUS, "+"$\rangle$ $\langle$NUM, "26"$\rangle$ $\langle$RPAR, ")"$\rangle$ $\langle$MINUS, "−"$\rangle$ $\langle$NUM, "5"$\rangle$

Nota: l'analizzatore lessicale non è preposto al riconoscimento della *struttura* delle espressioni. Pertanto, esso accetterà anche espressioni aritmetiche errate quali ad esempio:

- $5+)$
- $(34 + 26(-(2 + 15 - (27$

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 `@`.

Suggerimento: definire una classe `Tag` con un insieme opportuno di costanti intere per rappresentare il nome dei token come NUM, PLUS, ecc.; definire una classe `Token` (ed eventualmente classi derivate) per rappresentare i token. Una possibile struttura del programma (ispirato al testo [1, Appendice A.3]) è la seguente:

Listing 1: Analizzatore lessicale di espressioni semplici

```
import java.io.*;

public class Lexer {
    public static int line = 1;
    private char peek = ' ';

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

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

        switch (peek) {
        case '(':
            peek = ' ';
            return new Token(Tag.LPAR, "(");

        // ... gestire gli altri caratteri ... //

        default:
            if (Character.isDigit(peek)) {
                String s = "";
                do {
                    s += peek;
                    readch();
                } while (Character.isDigit(peek));
                return new Token(Tag.NUM, s);
            } else
                // ... gestire caratteri illeciti ... //
        }
    }

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

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

NOTA: per inviare il segnale di "fine file" da tastiera usare `CTRL-D` nei sistemi Unix-like (Linux e OS X) e `CTRL-Z` nei sistemi Windows.

I.2 Analizzatore sintattico

Si scriva un analizzatore sintattico a discesa ricorsiva che parsifichi le espressioni semplici discusse precedentemente. In particolare, l'analizzatore deve riconoscere le espressioni generate dalla grammatica che segue:

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

Il programma deve fare uso dell'analizzatore lessicale sviluppato in precedenza. Una possibile struttura del programma (ispirato al testo [1, Appendice A.8]) è la seguente:

Listing 2: Analizzatore sintattico di espressioni semplici

```
import java.io.*;

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

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

    void move() {
        look = lex.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() {
        expr();
        match(Tag.EOF);
    }

    private void expr() {
```

```
        term();
        exprp();
    }

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

        case Tag.MINUS:
            // ... gestire gli altri casi ... //
        }
    }

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

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

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

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

Il main per eseguire il tutto ha una forma tipo la seguente:

Listing 3: Main per l'analizzatore sintattico di espressioni semplici

```
import java.io.*;

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

I.3 Implementazione del valutatore

Modificare il precedente analizzatore sintattico in modo da valutare le espressioni:

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

Una possibile struttura del programma (ispirato al testo [1, Appendice A.8]) è la seguente:

Listing 4: Valutazione di espressioni semplici

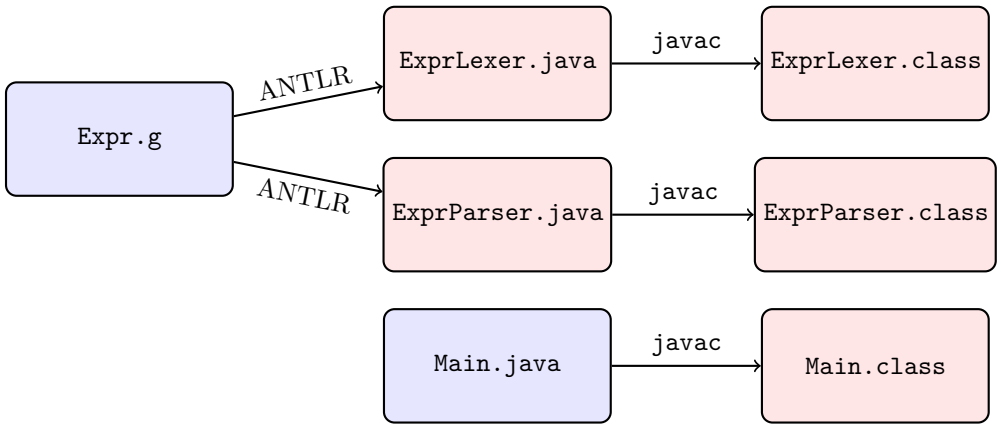
```
import java.io.*;

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

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

    void move() {
        look = lex.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() {
        int expr_val;

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

        System.out.println(expr_val);
    }

    private int expr() {
```

```
        int term_val, exprp_val;

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

        return exprp_val;
    }

    private int exprp(int exprp_i) {
        int term_val, exprp_val;

        switch (look.tag) {
        case Tag.PLUS:
            match(Tag.PLUS);
            term_val = term();
            exprp_val = exprp(exprp_i + term_val);
            break;

        case Tag.MINUS:
            // ... completare ... //
    }

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

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

        switch (look.tag) {
        case Tag.TIMES:
            match(Tag.TIMES);
            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 = Integer.parseInt(look.text);
            // ... completare ... //
        }

        return fact_val;
    }
}
```

Il main per eseguire il tutto è lo stesso del listato 3.

II Valutatore di espressioni semplici con ANTLR

Il tool ANTLR [2] consente la generazione automatica del codice corrispondente agli analizzatori lessicale e sintattico a partire dalla specifica di una grammatica.



II.1 Analizzatore lessicale e sintattico in ANTLR

Realizzare un parsificatore a discesa ricorsiva per la grammatica della Sezione I utilizzando lo strumento ANTLR.

Listing 5: Grammatica ANTLR di espressioni semplici

```
grammar Expr;

start
  : expr EOF
  ;

expr
  : term exprP
  ;

exprP
  :
  | '+' term exprP
  | '-' term exprP
  ;

term
  : fact termP
  ;

termP
  :
  | '*' fact termP
  | '/' fact termP
  ;

fact
  : '(' expr ')'
  | NUM
  ;

NUM : '0'..'9'+ ;
```

```
WS : (' ' | '\t' | '\r' | '\n')+ { skip(); } ;
```

Listing 6: Classe `Main`

```
import org.antlr.runtime.*;

public class Main {
    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        ExprLexer lexer = new ExprLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ExprParser parser = new ExprParser(tokens);
        parser.start();
    }
}
```

Per invocare il tool ANTLR usare il comando:

```
java -cp antlr-3.5-complete.jar org.antlr.Tool Expr.g
```

Per compilare i file `.java`:

```
javac -cp .;antlr-3.5-complete.jar *.java
```

Per eseguire il programma:

```
java -cp .;antlr-3.5-complete.jar Main
```

NOTA: gli elementi del classpath Java vanno separati da `:` nei sistemi Unix-like (Linux e OS X) e da `;` nei sistemi Windows.

II.2 Uso degli attributi sintetizzati ed ereditati in ANTLR

Implementare il valutatore di espressioni aritmetiche annotando opportunamente la specifica ANTLR della sezione II.1.

Listing 7: Attributi sintetizzati ed ereditati in ANTLR (estratto da completare)

```
exprP[int i] returns [int val]
  :                                  { $val = $i; }
  | '+' term e=exprP[$i + $term.val]  { $val = $e.val; }
  | '-' term e=exprP[$i - $term.val]  { $val = $e.val; }
  ;

fact returns [int val]
  : '(' expr ')'                     { $val = $expr.val; }
  | NUM                              { $val = Integer.parseInt($NUM.text); }
  ;
```

II.3 Uso delle estensioni EBNF di ANTLR

ANTLR riconosce costrutti speciali per indicare parti opzionali o iterate nelle produzioni di una grammatica. In particolare, la grammatica mostrata nella Sezione I può essere riscritta in maniera più compatta usando questi costrutti come mostrato qui sotto:

$$\langle start \rangle \quad ::= \quad \langle expr \rangle \ \texttt{EOF}$$

$$\langle expr \rangle \quad ::= \quad \langle term \rangle \ [\ \texttt{+} \ \langle term \rangle \mid \texttt{-} \ \langle term \rangle \ ]^*$$

$$\langle term \rangle \quad ::= \quad \langle fact \rangle \ [\ \texttt{*} \ \langle fact \rangle \mid \texttt{/} \ \langle fact \rangle \ ]^*$$

$$\langle fact \rangle \quad ::= \quad \texttt{(} \ \langle expr \rangle \ \texttt{)} \mid \texttt{NUM}$$

In sintassi ANTLR:

Listing 8: Specifica ANTLR con estensioni EBNF (estratto da completare)

```
expr
  : term ( '+' term | '-' term )*
  ;
```

II.4 Uso degli attributi in ANTLR con costrutti EBNF

Implementare il valutatore di espressioni aritmetiche annotando opportunamente la specifica ANTLR della sezione II.3.

Listing 9: Attributi in ANTLR con EBNF (estratto da completare)

```
expr returns [int val]
  : t0=term { $val = $t0.val; } ( '+' t=term { $val += $t.val; }
                                | '-' t=term { $val -= $t.val; }
                                )*
  ;
```

Oltre all'operatore `*`, che consente di descrivere la ripetizione "zero o più volte" di un costrutto sintattico, è possibile usare anche l'operatore `+` per descrivere la ripetizione "una o più volte" (già usato nell'espressione regolare che descrive il token `NUM`) così come l'operatore `?`, che indica la presenza *opzionale* di un costrutto sintattico.

III Compilatore di un semplice linguaggio imperativo

In quest'ultima parte svilupperemo un compilatore per un semplice linguaggio imperativo sottoinsieme del Pascal, che chiameremo linguaggio `P`, e che include, oltre al sotto-linguaggio delle espressioni aritmetiche trattato fino ad ora, le espressioni logiche, la dichiarazione di variabili, il comando di assegnamento, i comandi condizionali e iterati.

III.1 Sintassi del linguaggio `P`

La sintassi del linguaggio `P`, è descritto dalla seguente grammatica EBNF.

$$
\begin{aligned}
\langle prog \rangle &::= [\ \langle decl \rangle\ \texttt{;}\ ]^{*}\ \langle stmt \rangle\ \texttt{EOF} \\
\langle decl \rangle &::= \texttt{var ID}\ [\ \texttt{,}\ \texttt{ID}\ ]^{*}\ \texttt{:}\ \langle type \rangle \\
\langle type \rangle &::= \texttt{integer} \mid \texttt{boolean} \\
\langle expr \rangle &::= \langle andExpr \rangle\ [\ \texttt{or}\ \langle andExpr \rangle\ ]^{*} \\
\langle andExpr \rangle &::= \langle relExpr \rangle\ [\ \texttt{and}\ \langle relExpr \rangle\ ]^{*} \\
\langle relExpr \rangle &::= \langle addExpr \rangle\ [\ \texttt{=}\ \langle addExpr \rangle \mid \texttt{<>}\ \langle addExpr \rangle \\
&\qquad \mid \texttt{<=}\ \langle addExpr \rangle \mid \texttt{>=}\ \langle addExpr \rangle \\
&\qquad \mid \texttt{<}\ \langle addExpr \rangle \mid \texttt{>}\ \langle addExpr \rangle\ ]^{?} \\
\langle addExpr \rangle &::= \langle mulExpr \rangle\ [\ \texttt{+}\ \langle mulExpr \rangle \mid \texttt{-}\ \langle mulExpr \rangle\ ]^{*} \\
\langle mulExpr \rangle &::= \langle unExpr \rangle\ [\ \texttt{*}\ \langle unExpr \rangle \mid \texttt{/}\ \langle unExpr \rangle\ ]^{*} \\
\langle unExpr \rangle &::= \texttt{+}\ \langle unExpr \rangle \mid \texttt{-}\ \langle unExpr \rangle \mid \texttt{not}\ \langle unExpr \rangle \mid \langle primary \rangle \\
\langle primary \rangle &::= \texttt{(}\ \langle expr \rangle\ \texttt{)} \mid \texttt{ID} \mid \texttt{NUM} \mid \texttt{true} \mid \texttt{false} \\
\langle stmt \rangle &::= \texttt{ID :=}\ \langle expr \rangle \\
&\mid \texttt{print (}\ \langle expr \rangle\ \texttt{)} \\
&\mid \texttt{if}\ \langle expr \rangle\ \texttt{then}\ \langle stmt \rangle\ [\ \texttt{else}\ \langle stmt \rangle\ ]^{?} \\
&\mid \texttt{while}\ \langle expr \rangle\ \texttt{do}\ \langle stmt \rangle \\
&\mid \texttt{begin}\ \langle stmt \rangle\ [\ \texttt{;}\ \langle stmt \rangle\ ]^{*}\ \texttt{end}
\end{aligned}
$$

Di seguito sono riportati alcuni esempi di programmi P:

Listing 10: Algoritmo di Euclide per il calcolo del massimo comun divisore

```
var m, n, tmp : integer;
begin
   m := 123456;
   n := 30;
   while m <> 0 do begin
      if m < n then begin
         tmp := m;
         m := n;
         n := tmp
      end;
      m := m - n
   end;
   print(n)
end
```

Listing 11: Test di primalità di un numero

```
var n, i, r : integer;
var primo : boolean;
begin
   n := 1237;
   i := 2;
   primo := true;
   while i < n / 2 and primo do begin
      r := n - (n / i) * i;
```

```
      print(r);
      primo := r <> 0;
      i := i + 1
   end;
   print(primo)
end
```

Listing 12: Calcolo della sequenza di Fibonacci

```
var i, k, m, n, res : integer;
begin
   k := 10;
   if k = 0 then res := 0
   else begin
      i := 1;
      m := 0;
      n := 1;
      while i <= k - 1 do begin
         n := n + m;
         m := n - m;
         i := i + 1
      end;
      res := n
   end;
   print(res)
end
```

Scrivere una grammatica ANTLR per riconoscere programmi scritti nel linguaggio `P` e verificarne il buon funzionamento con i programmi qui sopra ed altri a piacere.

III.2 Type checker

Prima di tradurre un programma scritto nel linguaggio `P`, il compilatore deve verificarne la *correttezza di tipo* secondo i seguenti criteri:

- Gli operatori aritmetici possono essere applicati esclusivamente ad operandi di tipo `integer`. Ad esempio, l'espressione `5 + true`, che usa l'operatore aritmetico `+` applicato a un operando booleano, è sintatticamente ma non semanticamente corretta e dunque non può essere tradotta.
- Gli operatori logici possono essere applicati esclusivamente ad operandi di tipo `boolean`.
- Gli operatori d'ordine `<`, `>`, `<=` e `>=` possono essere applicati esclusivamente ad operandi numerici, mentre gli operatori `=` e `<>` sono *overloaded*, ovvero possono essere applicati indifferentemente ad operandi di tipo numerico o booleano, purché entrambi gli operandi siano dello stesso tipo.
- I comandi di assegnamento sono corretti se il tipo dell'espressione a destra dell'operatore `:=` è uguale al tipo dell'identificatore a sinistra dell'operatore `:=`.
- I comandi condizionali e iterativi contengono un'espressione che deve essere di tipo booleano.

Inoltre, ci sono altri due errori non legati ai tipi che il compilatore deve gestire:

- Ogni identificatore deve essere dichiarato prima del suo utilizzo.
- Un identificatore non può essere dichiarato più di una volta.

Per effettuare il controllo di tipo, si introduce una enumerazione per distinguere i due tipi di dato dei programmi `P`:

Listing 13: Rappresentazione dei tipi `P`

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

È necessario aggiungere attributi sintetizzati per tenere traccia del tipo delle espressioni. Ad esempio, per le produzioni di ⟨*primary*⟩ possiamo avere un codice ANTLR come il seguente:

Listing 14: Verifica di correttezza di espressioni primarie (frammento)

```
primary returns [Type type]
    :  NUM     { $type = Type.INTEGER; }
    | 'true'   { $type = Type.BOOLEAN; }
    | ...
    ;
```

mentre per ⟨*relExpr*⟩ possiamo avere:

Listing 15: Verifica di correttezza di espressioni relazionali (frammento)

```
relExpr returns [Type type]
    : e1=addExpr { $type = $e1.type; }
        ( '=' e2=addExpr
          { if ($e1.type != $e2.type)
              throw new IllegalArgumentException("Type error in =");
            $type = Type.BOOLEAN;
          }
        | '<' ...
        | ...
        )?
    ;
```

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*. Segue una possibile implementazione:

Listing 16: Implementazione della tabella dei simboli

```
import java.util.*;

public class SymbolTable {
    private Map<String, Type> typeMap = new HashMap<String, Type>();

    public void insert(String s, Type t) {
        if (!typeMap.containsKey(s))
            typeMap.put(s, t);
        else
            throw new IllegalArgumentException("Variabile duplicata " + s);
    }

    public Type lookupType(String s) {
        if (typeMap.containsKey(s))
            return typeMap.get(s);
        else
            throw new IllegalArgumentException("Variabile sconosciuta " + s);
    }
}
```

Per fare in modo che la classe generata da ANTLR contenga una istanza della tabella dei simboli è necessario informare ANTLR attraverso la direttiva `@members`:

Listing 17: Generazione del parser con la tabella dei simboli

```
grammar P;
```

```
@members {
        SymbolTable st = new SymbolTable();
        ...
}

...
```

Tutto ciò che è racchiusto tra le parentesi graffe viene copiato così com'è nella classe di parsing generata da ANTLR. È utile sfruttare questo meccanismo per dichiarare metodi ausiliari, per esempio per implementare frammenti di codice che effettuano la verifica di correttezza di tipo e che vengono applicati invariati in numerosi punti del compilatore.

La tabella dei simboli verrà utilizzata così:

Listing 18: Uso della tabella dei simboli

```
declaration
    :   'var' ID ':' type ';'
        { st.insert($ID.text, $type.type); }
    ;

...

stmt
    : ID ':=' expr
        {
            if (st.lookupType($ID.text) != $expr.type)
                throw new IllegalArgumentException("errore di tipo");
        }
...
```

NOTA: le dichiarazioni del linguaggio `P` consentono di specificare più identificatori separati da virgole di seguito alla parola chiave `var`, pertanto l'azione semantica associata alla produzione mostrata qui sopra relativa al non-terminale `declaration` va opportunamente generalizzata.

Riferimenti bibliografici

[1] Aho, Alfred V., Lam, Monica S., Sethi, Ravi, and Ullman, Jeffrey D. Compilers: Principles, Techniques, and Tools. *Addison Wesley Longman*, 2007.

[2] ANTLR versione 3. `http://www.antlr3.org/`

[3] Assembly language. `http://en.wikipedia.org/wiki/Assembly_language` *Wikipedia*, 2012.

[4] Java class file. `http://en.wikipedia.org/wiki/Java_class_file` *Wikipedia*, 2012.

[5] Java bytecode. `http://en.wikipedia.org/wiki/Java_bytecode` *Wikipedia*, 2012.

[6] Java bytecode instruction listings. `http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings` *Wikipedia*, 2012.