```java
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@#
#@ SCANNER                                                                    @#
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@#


package it.gabliz.scanner;

import it.gabliz.token.Token;
import it.gabliz.token.TokenType;
import it.gabliz.exception.AcdcLexicalException;
import it.gabliz.util.Logger;
import it.gabliz.exception.TokenConstructorException;

import java.io.*;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;

import static it.gabliz.util.Logger.formatLogNewLine;

/**
 * Classe che gestisce lo scanner del nostro progetto.
 * @author Gabliz
 * @see Token Questa classe rappresenta il singolo oggetto TOKEN che questo scanner usa.
 */
public class Scanner {

    /** Carattere di terminazione file */
    private static final char EOF = (char) -1;

    /** Indicazione della riga su cui lo scanner è attualmente */
    private int riga;

    /** Oggetto per gestire il buffer di caratteri da leggere */
    private PushbackReader buffer;

    @SuppressWarnings("unused")
    @Deprecated
    private String log;

    /** Lista dei caratteri da saltare */
    private final List<Character> skipChars = Arrays.asList(' ', '\n', '\t', '\r', EOF);

    /** Lista dei caratteri che rappresentano lettere */
    private final List<Character> letters = Arrays.asList('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
            'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z');

    /** Lista dei caratteri che rappresentano numeri */
    private final List<Character> numbers = Arrays.asList('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');

    /** Hashmap per associare i tipi di token e i relativi caratteri char
     * @see it.gabliz.token.TokenType */
    private HashMap<Character, TokenType> symbols;

    /** Hashmap per associare i tipi di token e parole chiave del linguaggio.
     * @see it.gabliz.token.TokenType per i tipi di token */
    private HashMap<String, TokenType> keywords;

    /** Il token attualmente in analisi dallo scanner */
    private Token currentToken;

    /** Classe per gestire i log di questa classe.
     * @see Logger per gestione log. */
    private final Logger logger;

    /** Dichiarazione path di base di tutti i file che leggerà questo scanner */
    private static final String PATH_ROOT = "./res/";
```

```java

    /** Numero minimo di cifre di un numero float */
    private static final int FLOAT_MIN_DIGIT = 1;

    /** Numero massimo di cifre di un numero float */
    private static final int FLOAT_MAX_DIGIT = 4;

    /** Abilitazione dei log di tutte le chiamate di {@link #peekChar()} o {@link #readChar()} */
    private static final Boolean ENABLE_PEEK_READ_CHAR_LOGGIN = false;


    /**
     * Costruttore di uno scanner.
     * @param fileName il nome del file da passare allo scanner.
     * @throws FileNotFoundException Se il file non viene trovato.
     * @throws UnsupportedOperationException Se il file non può essere letto.
     */
    public Scanner(String fileName) throws FileNotFoundException, UnsupportedOperationException {

        /* stampo informazioni */
        logger = new Logger(this.getClass().getSimpleName());
        logger.i("Creata nuova istanza di uno scanner per il file \"" + fileName + "\".");

        /* controllo il file */
        File f = new File(PATH_ROOT + fileName);
        if(!f.exists()) throw new FileNotFoundException("Il file \"" + fileName + "\" non è stato trovato in \"" + PATH_ROOT + "\".");
        if(!f.canRead()) throw new UnsupportedOperationException("Il file \"" + fileName + "\" non può essere letto.");

        /* preparazione scanner */
        try {
            this.buffer = new PushbackReader(new FileReader(f));
        }catch (FileNotFoundException e) {
            logger.e("Rilevato errore apertura file per lo scanner. Controllare se path passato è directory.");
            e.printStackTrace();
        }
        logger.i("Il file \"" + fileName + "\" è stato trovato e caricato nello scanner.");
        riga = 1;
        loadSymbolsHashMap();
        loadKeywordsHashMap();
        this.currentToken = null;
    }


    /**
     * Questo metodo serve per recuperare il prossimo token dall'input MA SENZA CONSUMARLO.
     * Quindi una successiva chiamata a questo metodo alla funzione nextToken ritorna ancora
     * il token attuale.
     *
     * @return Il token corrente. Il ritorno di questa funzione dipende dal token attuale:
     * - Se currentToken != null -> la nextToken ha già assegnato il token e lo ritorno.
     * - se currentToken == null -> chiamo la nextToken che assegnerà il nuovo token e lo ritorno.
     * @throws IOException errore di lettura buffer.
     * @throws AcdcLexicalException Se viene rilevato un errore lessicale.
     * @throws TokenConstructorException Se vengono passati parametri errati al token.
     */
    public Token peekToken() throws IOException, AcdcLexicalException, TokenConstructorException {
        logger.d("PeekToken chiamata con currentToken = \"" + currentToken + "\".");
        if(currentToken == null) {
```

```
126             currentToken = nextToken();
127         }
128         return currentToken;
129     }
130 
131     /**
132      * Metodo per recuperare il prossimo token dall'input.
133      * Questo è effettivamente il metodo che può consumare l'input.
134      * Questo metodo può essere visto come una rappresentazione dell'automa
     riconoscitore.
135      * @return il token recuperato.
136      * @throws IOException Per eventuali errori di lettura nel buffer.
137      * @throws AcdcLexicalException Se viene rilevato un errore lessicale.
138      * @throws TokenConstructorException Se vengono passati parametri errati al token.
139      */
140     public Token nextToken() throws IOException, AcdcLexicalException,
     TokenConstructorException {
141 
142         /* stampo log */
143         logger.d("nextToken chiamata con currentToken = \"" + currentToken + "\".");
144 
145         /* Controllo token:
146         * - se è già settato (è != null) vuol dire che una precedente chiamata di
         nextToken lo ha settato.
147         *   Ora lo consumo (lo setto a null) ritornandolo.
148         * - Se non è settato (è == null) vuol dire che è stato consumato
         precedentemente l'input o questa è la prima
149         *   chiamata di nextToken(). Essendo null devo prelevare il prossimo token.
150         *  */
151         if(currentToken != null) {
152             Token token = currentToken;
153             currentToken = null;
154             return token;
155         }
156 
157         /* stampo log */
158         logger.i("Inizio procedura calcolo nuovo token da input.");
159 
160         /* prendo il prossimo carattere dall'input */
161         int nextCharInt = buffer.read();
162         char nextChar = (char) nextCharInt;
163         logger.d("Ho letto dal buffer (consumando) il carattere '" +
         formatLogNewLine(nextChar) + "'.");
164 
165         /* inizio ciclo per controllo caratteri per creare token */
166         while(nextChar != EOF && nextCharInt != -1) {
167 
168             if(nextChar == Token.CHAR_NEW_LINE) {
169                 riga++;
170                 logger.d("La prossima riga da guardare sarà riga " + riga + ".");
171             } else if(!skipChars.contains(nextChar)) {
172                 if(numbers.contains(nextChar)) {
173                     logger.d("Trovato carattere di tipo numerico. Rimetto il
                     carattere nel buffer e procedo controllo numeri successivi.");
174                     buffer.unread(nextChar);
175                     return scanAndGetNumberToken();
176                 }
177                 if(letters.contains(nextChar)) {
178                     logger.d("Trovato carattere di tipo letterale. Rimetto il
                     carattere nel buffer e procedo controllo lettere successive.");
179                     buffer.unread(nextChar);
180                     return scanAndGetIdToken();
181                 }
182                 if(symbols.containsKey(nextChar)) {
183                     return new Token(symbols.get(nextChar), riga).logCreation();
184                 }
185                 throw new AcdcLexicalException("Rilevato errore lessicale dello
                 scanner.");
186             }
187 
188             /* Se arrivati qua è perchè ho trovato un carattere di skip. Leggo
```

```
                prossimo carattere. */
189             char oldChar = nextChar;
190             nextCharInt = buffer.read();
191             nextChar = (char) nextCharInt;
192             logger.d("Trovato carattere di skip ('" + formatLogNewLine(oldChar)
                +"'). " +
193                     "Prossimo carattere in lettura = '" + formatLogNewLine(nextChar)
                        + "'");
194         }
195
196         /* se arrivati a questo punto non ho più altri token da generare e sono
            arrivato a fine file */
197         return new Token(TokenType.EOF, riga).logCreation();
198
199     }
200
201     /**
202      * Se viene chiamato questo metodo è stato trovato carattere numerico. Il
         compito di questo
203      * metodo è quello di generare il token a partire da questo (e potenzialmente
         anche dai successivi) carattere/i
204      * numerici (e decidere se numero INT O FLOAT).
205      * @return il token numero di tipo INT o FLOAT associato al numero/numeri in
         input nel buffer.
206      * @throws IOException se si sono verificati errori di lettura del buffer.
207      * @throws AcdcLexicalException Se vengono trovate più di quattro cifre dopo la
         virgola.
208      * @throws TokenConstructorException Se vengono passati parametri errati al token.
209      */
210     private Token scanAndGetNumberToken() throws IOException, AcdcLexicalException,
        TokenConstructorException {
211         int cifre = 0;
212         StringBuilder number = new StringBuilder();
213
214         while(numbers.contains(peekChar())) {
215             number.append(readChar());
216         }
217
218         if(peekChar() != Token.CHAR_DOT)
219             return new Token(TokenType.INT, riga, number.toString()).logCreation();
220
221         number.append(readChar());
222         while(numbers.contains(peekChar())) {
223             number.append(readChar());
224             cifre++;
225         }
226
227         if (cifre >= FLOAT_MIN_DIGIT && cifre <= FLOAT_MAX_DIGIT) {
228             return new Token(TokenType.FLOAT, riga, number.toString()).logCreation();
229         } else  {
230             if(cifre == 0) {
231                 throw new AcdcLexicalException("Trovato un possibile numero float
                    senza numeri decimali.");
232             } else {
233                 throw new AcdcLexicalException("Trovato un numero float con più di "
                    + FLOAT_MAX_DIGIT + " cifre alla riga: "+ riga + "");
234             }
235         }
236     }
237
238     /**
239      * Se viene chiamato questo metodo è stato trovato carattere alfabetico. Il
         compito di questo
240      * metodo è quello di generare il token a partire da questo (e potenzialmente
         anche dai successivi) carattere/i
241      * alfabetici (e decidere se è parola chiave o no).
242      * @return il token associato agli input del buffer.
243      * @throws IOException se si sono verificati errori di lettura del buffer.
244      * @throws TokenConstructorException Se vengono passati parametri errati al token.
245      */
246     private Token scanAndGetIdToken() throws IOException, TokenConstructorException {
```

```
247             char ch;
248             StringBuilder word = new StringBuilder();
249             while (letters.contains(peekChar())) {
250                 ch = readChar();
251                 word.append(ch);
252             }
253             if (keywords.containsKey(word.toString())) {
254                 return new Token(keywords.get(word.toString()), riga).logCreation();
255             } else return new Token(TokenType.ID, riga, word.toString()).logCreation();
256         }
257 
258         /**
259          * Prendo un singolo carattere dal buffer (int).
260          * @return Il carattere preso.
261          * @throws IOException Per eventuali errori di lettura dal buffer.
262          */
263         @Deprecated
264         @SuppressWarnings("unused")
265         private int simpleReadChar() throws IOException {
266             int c = buffer.read();
267             logger.d("Ho prelevato dal buffer (read) il carattere '" + (char) c + "'.");
268             return c;
269         }
270 
271         /**
272          * Metodo per scansionare i numeri per creare token di tipo intero o float.
273          * @deprecated Questo è il vecchio metodo per scansione i numeri.
274          * Usare {@link #scanAndGetNumberToken()}
275          * @return Il token numerico.
276          * @throws IOException se si sono verificati errori di lettura del buffer.
277          * @throws AcdcLexicalException Se vengono trovate più di quattro cifre dopo la
             virgola.
278          * @throws TokenConstructorException Se vengono passati parametri errati al token.
279          */
280         @Deprecated()
281         @SuppressWarnings("unused")
282         private Token scanNumber() throws IOException, AcdcLexicalException,
             TokenConstructorException {
283             char c;
284             int i = 0;
285             StringBuilder number = new StringBuilder();
286             while(numbers.contains(peekChar())) {
287                 c = readChar();
288                 number.append(c);
289             }
290             c = peekChar();
291             if(c == Token.CHAR_DOT) {
292                 c = readChar();
293                 number.append(c);
294                 if(!numbers.contains(peekChar())) {
295                     buffer.unread(c);
296                     return new Token(TokenType.INT, riga,
                         number.toString()).logCreation();
297                 } else {
298                     while(numbers.contains(peekChar())) {
299                         c = readChar();
300                         number.append(c);
301                         i++;
302                     }
303                     if (i >= FLOAT_MIN_DIGIT && i <= FLOAT_MAX_DIGIT) {
304                         return new Token(TokenType.FLOAT, riga,
                             number.toString()).logCreation();
305                     } else  {
306                         throw new AcdcLexicalException("Trovato un numero float con più
                             di " + FLOAT_MAX_DIGIT + " cifre alla riga: "+ riga + ".");
307                     }
308                 }
309             }
310             else return new Token(TokenType.INT, riga, number.toString()).logCreation();
311         }
312 
```

```
313         /**
314          * Prendo un singolo carattere dal buffer (char).
315          * @return Il carattere preso.
316          * @throws IOException Per eventuali errori di lettura dal buffer.
317          */
318         private char readChar() throws IOException {
319             char c = (char) this.buffer.read();
320             if(ENABLE_PEEK_READ_CHAR_LOGGIN) logger.d("readChar = " +
                formatLogNewLine(c));
321             return (c);
322         }
323 
324         /**
325          * Prendo un singolo carattere dal buffer senza consumarlo.
326          * @return Il carattere preso.
327          * @throws IOException Per eventuali errori di lettura dal buffer.
328          */
329         private char peekChar() throws IOException {
330             char c = (char) buffer.read();
331             if(ENABLE_PEEK_READ_CHAR_LOGGIN) logger.d("peekChar = " +
                formatLogNewLine(c));
332             buffer.unread(c);
333             return c;
334         }
335 
336         /** Metodo per caricare la hashmap dei simboli */
337         private void loadSymbolsHashMap() {
338             symbols = new HashMap<>();
339             symbols.put('+', TokenType.PLUS);
340             symbols.put('-', TokenType.MINUS);
341             symbols.put('*', TokenType.TIMES);
342             symbols.put('/', TokenType.DIV);
343             symbols.put('=', TokenType.ASSIGN);
344             symbols.put(';', TokenType.SEMI);
345         }
346 
347         /** Metodo per caricare la hashmap dei simboli */
348         private void loadKeywordsHashMap() {
349             keywords = new HashMap<>();
350             keywords.put("int", TokenType.TYINT);
351             keywords.put("float", TokenType.TYFLOAT);
352             keywords.put("print", TokenType.PRINT);
353         }
354     }
355 
356     #*****************************************************************************#
357 
358     package it.gabliz.token;
359 
360     import it.gabliz.util.Logger;
361     import it.gabliz.exception.TokenConstructorException;
362 
363     import java.util.*;
364 
365     /**
366      * Classe che rappresenta la singola istanza 'TOKEN'.
367      * @author Gabliz
368      */
369     public class Token {
370 
371         /** Indica la riga su cui si trovava il token */
372         private int riga;
373 
374         /** Indica il tipo di token */
375         private TokenType tipo;
376 
377         /** Indica il valore associato (se associato) */
378         private String val;
379 
380         /** Costante che indica che al token non è associato nessun valore. */
381         public static final String EMPTY_VAL = "";
```

```java

    public static final Character CHAR_NEW_LINE = '\n';
    public static final Character CHAR_DOT = '.';
    private final String CLASS_NAME = this.getClass().getSimpleName().toUpperCase();
    private static final Set<TokenType> tokenTypesWithValues = Set.of(TokenType.ID,
    TokenType.INT, TokenType.FLOAT);

    /**
     * Costruttore con campo valore.
     * @param tipo il tipo di token
     * @param riga la riga su cui si trovava il token
     * @param val il valore associato
     * @throws TokenConstructorException se parametri token errati.
     */
    public Token(TokenType tipo, int riga, String val) throws
    TokenConstructorException {
        if(tipo == null) throw new TokenConstructorException("Il tipo di un token
        non può essere nullo.");
        if(riga <=0 ) throw new TokenConstructorException("La riga non può essere 0
        o negativa.");
        this.riga = riga;
        this.tipo = tipo;
        this.val = val;
        checkTokenConstructorWithVal();
    }

    /**
     * Costruttore senza campo valore.
     * @param tipo il tipo di token
     * @param riga la riga su cui si trovava il token
     * @throws TokenConstructorException se parametri token errati.
     */
    public Token(TokenType tipo, int riga) throws TokenConstructorException {
        if(tipo == null) throw new TokenConstructorException("Il tipo di un token
        non può essere nullo.");
        if(riga <=0 ) throw new TokenConstructorException("La riga non può essere 0
        o negativa.");
        this.riga = riga;
        this.tipo = tipo;
        this.val = EMPTY_VAL;
        checkTokenConstructorWithoutVal();
    }

    /**
     * @return String Una stringa che rappresenta questo token con valore
     */
    private String getStringWithVal() {return "<" + tipo + "," + "r:" + riga + "," +
    val + ">";}

    /**
     * @return String Una stringa che rappresenta questo token senza valore
     */
    private String getStringWithoutVal() {return "<" + tipo + "," + "r:" + riga +
    ">";}

    private void checkTokenConstructorWithVal() {
        if(!tokenTypesWithValues.contains(this.tipo))
            Logger.w(CLASS_NAME, "Il costruttore per il token " + this.tipo + " è
            errato.");
    }

    private void checkTokenConstructorWithoutVal() {
        if(tokenTypesWithValues.contains(this.tipo))
            Logger.w(CLASS_NAME, "Il costruttore per il token " + this.tipo + " è
            errato (controllare presenza/assenza valore).");
    }

    public Token logCreation() {
        Logger.i(CLASS_NAME, "Creato nuovo token: \"" + this + "\".");
        return this;
    }
```

```
443
444         public String toString() {
445             if(!Objects.equals(this.val, EMPTY_VAL)) {
446                 return getStringWithVal();
447             } else {
448                 return getStringWithoutVal();
449             }
450         }
451
452         @Override
453         public boolean equals(Object obj) {
454             if(this == obj) return true;
455             if (!(obj instanceof Token)) {
456                 Logger.e(CLASS_NAME, "Tentativo di equals con due oggetti di classi
                    differenti o non compatibili.");
457                 Logger.e(CLASS_NAME, "I tipi in questione sono : " + getClass() + " e "
                    + obj.getClass());
458                 return false;
459             }
460             Boolean cond1 = this.tipo == ((Token)obj).getTipo();
461             Boolean cond2 = Objects.equals(this.val, ((Token) obj).getVal());
462             Boolean cond3 = Objects.equals(this.riga, ((Token) obj).getRiga());
463             return cond1 && cond2 && cond3;
464         }
465
466         public int getRiga() {
467             return riga;
468         }
469
470         public TokenType getTipo() {
471             return tipo;
472         }
473
474         public String getVal() {
475             return val;
476         }
477
478         public void setRiga(int riga) {
479             this.riga = riga;
480         }
481
482         public void setTipo(TokenType tipo) {
483             this.tipo = tipo;
484         }
485
486         public void setVal(String val) {
487             this.val = val;
488         }
489     }
490
491     #*****************************************************************************#
492
493     package it.gabliz.token;
494
495     public enum TokenType {
496         TYFLOAT,
497         TYINT,
498         PRINT,
499         ID,
500         INT,
501         FLOAT,
502         ASSIGN,
503         PLUS,
504         MINUS,
505         TIMES,
506         DIV,
507         SEMI,
508         EOF
509     }
510
511
```

```
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@#
#@ PARSER                                                                         @#
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@#


package it.gabliz.parser;

import it.gabliz.ast.*;
import it.gabliz.exception.AcdcLexicalException;
import it.gabliz.exception.AcdcSyntaxException;
import it.gabliz.exception.ScannerException;
import it.gabliz.exception.TokenConstructorException;
import it.gabliz.scanner.Scanner;
import it.gabliz.token.Token;
import it.gabliz.token.TokenType;
import it.gabliz.util.*;

import java.io.IOException;
import java.util.ArrayList;

/**
 * Classe che implementa il parser.
 * @author Gabliz
 */
public class Parser {

    /** Variabile scanner fase precedente */
    private Scanner scanner;

    /** Lista dei nodi Dec/St */
    private ArrayList<NodeDecSt> arrayNode;

    /** Classe per gestire i log di questa classe.
     * @see Logger per gestione log. */
    private final Logger logger;

    /**
     * Costruttore di un parser.
     * @param scanner l'oggetto scanner per chiamare {@link Scanner#peekToken()}
     * @throws IllegalArgumentException se parametri costruttore nulli
     */
    public Parser(Scanner scanner) throws IllegalArgumentException {
        if(scanner == null) throw new IllegalArgumentException("L'oggetto scanner non può essere nullo!");
        this.scanner = scanner;
        this.arrayNode = new ArrayList<>();
        logger = new Logger(this.getClass().getSimpleName());
        logger.i("Inizio fase di parsing per il file corrente.");
    }

    /**
     * Questo metodo serve per vedere nel futuro e controllare se il prossimo token è un token corretto
     * (ha un certo tipo). Se ha il tipo che mi aspettavo lo consumo, altrimenti restituisco un errore.
     * @param type il tipo di token che mi aspetto.
     * @throws AcdcSyntaxException se viene rilevato un errore di sintassi.
     */
    private Token match(TokenType type) throws AcdcSyntaxException, TokenConstructorException, IOException, AcdcLexicalException {
        /* TODO : togliere valore di ritorno */
        Token tk = scanner.peekToken();
        if (type.equals(tk.getTipo())) {
            scanner.nextToken();
            return tk;
        }
        else throw new AcdcSyntaxException("Errore di sintassi rilevato durante il match. Il parser si aspettava " +
```

```
578                 "un token di tipo " + type + " ma ha trovato un token di tipo " +
                    tk.getTipo() + "(Riga"
579                 + tk.getRiga() + ").");
580         }
581
582         /**
583          * Metodo pubblico che si occupa di fare tutte le operazioni di parsing.
584          * @return Il nodo programma che contiene l'AST.
585          * @throws ScannerException Se viene rilevato un errore dello scanner.
586          * @throws AcdcSyntaxException Se viene rilevato un errore di sintassi.
587          */
588         public NodeProgram parse() throws ScannerException, AcdcSyntaxException {
589             try {
590                 NodeProgram nodeProgram = parsePrg();
591                 logger.i("Il parser ha restituito : " + nodeProgram);
592                 return nodeProgram;
593             }catch (IOException | TokenConstructorException | AcdcLexicalException e) {
594                 throw new ScannerException("Durante l'esecuzione del parser è stata
                    rilevata un exception " +
595                         "relativa allo scanner: \"" + e.getMessage() + "\".");
596             }
597         }
598
599         /** Parsing della produzione Prg */
600         private NodeProgram parsePrg() throws AcdcSyntaxException,
        TokenConstructorException, IOException, AcdcLexicalException, ScannerException {
601             Token tk = scanner.peekToken();
602
603             switch (tk.getTipo()) {
604                 case TYFLOAT:
605                 case TYINT:
606                 case ID:
607                 case PRINT:
608                     parseDSs();
609                     match(TokenType.EOF);
610                     return new NodeProgram(arrayNode);
611                 default:
612                     break;
613             }
614             throw new AcdcSyntaxException(AcdcSyntaxException.SYNTAX_ERROR_TEMPLATE, tk,
            tk.getRiga());
615         }
616
617         /** Parsing della produzione DSs */
618         private ArrayList<NodeDecSt> parseDSs() throws IOException,
        TokenConstructorException, AcdcLexicalException, AcdcSyntaxException {
619             /* TODO : Togliere valore di ritorno e mettere null */
620             Token token = scanner.peekToken();
621             switch(token.getTipo()) {
622                 case TYFLOAT:
623                 case TYINT:
624                     arrayNode.add(parseDcl());
625                     parseDSs();
626                     return arrayNode;
627                 case ID:
628                 case PRINT:
629                     arrayNode.add(parseStm());
630                     parseDSs();
631                     return arrayNode;
632                 case EOF:
633                     return arrayNode;
634                 default:
635                     break;
636             }
637             throw new AcdcSyntaxException(AcdcSyntaxException.SYNTAX_ERROR_TEMPLATE,
                token, token.getRiga());
638
639         }
640
641         /** parsing della produzione Dcl */
642         private NodeDecl parseDcl() throws IOException, AcdcLexicalException,
```

```
        AcdcSyntaxException, TokenConstructorException {
643         Token token = scanner.peekToken();
644         switch(token.getTipo()) {
645             case TYFLOAT:
646                 match(TokenType.TYFLOAT);
647                 token = match(TokenType.ID);
648                 match(TokenType.SEMI);
649                 return new NodeDecl(new NodeId(token.getVal()), LangType.FLOAT);
650             case TYINT:
651                 match(TokenType.TYINT);
652                 token = match(TokenType.ID);
653                 match(TokenType.SEMI);
654                 return new NodeDecl(new NodeId(token.getVal()), LangType.INT);
655             default:
656                 break;
657         }
658         throw new AcdcSyntaxException(AcdcSyntaxException.SYNTAX_ERROR_TEMPLATE,
            token, token.getRiga());
659     }
660
661     /** parsing della produzione Stm */
662     private NodeStm parseStm() throws IOException, TokenConstructorException,
        AcdcLexicalException, AcdcSyntaxException {
663
664         Token tk = scanner.peekToken();
665         NodeExpr t;
666         switch(tk.getTipo()) {
667             case ID:
668                 tk = match(TokenType.ID);
669                 match(TokenType.ASSIGN);
670                 t = parseExp();
671                 match(TokenType.SEMI);
672                 return new NodeAssign(new NodeId(tk.getVal()), t);
673             case PRINT:
674                 match(TokenType.PRINT);
675                 tk = match(TokenType.ID);
676                 match(TokenType.SEMI);
677                 return new NodePrint(new NodeId(tk.getVal()));
678             default:
679                 break;
680         }
681         throw new AcdcSyntaxException(AcdcSyntaxException.SYNTAX_ERROR_TEMPLATE, tk,
            tk.getRiga());
682     }
683
684     /** parsing della produzione Exp */
685     private NodeExpr parseExp() throws IOException, TokenConstructorException,
        AcdcLexicalException, AcdcSyntaxException {
686         Token tk = scanner.peekToken();
687         NodeExpr temp;
688         switch(tk.getTipo()) {
689             case FLOAT:
690             case INT:
691             case ID:
692                 temp = parseTr();
693                 return parseExpP(temp);
694             default:
695                 break;
696         }
697         throw new AcdcSyntaxException(AcdcSyntaxException.SYNTAX_ERROR_TEMPLATE, tk,
            tk.getRiga());
698     }
699
700     /** parsing della produzione Tr */
701     private NodeExpr parseTr() throws IOException, TokenConstructorException,
        AcdcLexicalException, AcdcSyntaxException {
702         Token tk = scanner.peekToken();
703         NodeExpr temp;
704         switch(tk.getTipo()) {
705             case INT:
706             case FLOAT:
```

```
707                 case ID:
708                     temp = parseVal();
709                     return parseTrP(temp);
710                 default:
711                     break;
712 
713             }
714             throw new AcdcSyntaxException(AcdcSyntaxException.SYNTAX_ERROR_TEMPLATE, tk,
                tk.getRiga());
715         }
716 
717         /** parsing della produzione Val */
718         private NodeExpr parseVal() throws IOException, TokenConstructorException,
                AcdcLexicalException, AcdcSyntaxException {
719             Token tk = scanner.peekToken();
720             switch(tk.getTipo()) {
721                 case INT:
722                     tk = match(TokenType.INT);
723                     return new NodeCost(tk.getVal(), LangType.INT);
724                 case FLOAT:
725                     tk = match(TokenType.FLOAT);
726                     return new NodeCost(tk.getVal(), LangType.FLOAT);
727                 case ID:
728                     tk = match(TokenType.ID);
729                     return new NodeDeref(new NodeId(tk.getVal()));
730                 default:
731                     break;
732             }
733             throw new AcdcSyntaxException(AcdcSyntaxException.SYNTAX_ERROR_TEMPLATE, tk,
                tk.getRiga());
734         }
735 
736         /** parsing della produzione TrP */
737         private NodeExpr parseTrP(NodeExpr left) throws IOException,
                TokenConstructorException, AcdcLexicalException, AcdcSyntaxException {
738             Token tk = scanner.peekToken();
739             NodeExpr temp;
740             switch(tk.getTipo()) {
741                 case TIMES:
742                     tk = match(TokenType.TIMES);
743                     temp = parseVal();
744                     return parseTrP(new NodeBinOp(LangOper.TIMES, left, temp));
745                 case DIV:
746                     tk = match(TokenType.DIV);
747                     temp = parseVal();
748                     return parseTrP(new NodeBinOp(LangOper.DIV, left, temp));
749                 case PLUS:
750                 case MINUS:
751                 case FLOAT:
752                 case INT:
753                 case SEMI:
754                 case ID:
755                 case PRINT:
756                 case EOF:
757                     return left;
758                 default:
759                     break;
760 
761             }
762             throw new AcdcSyntaxException(AcdcSyntaxException.SYNTAX_ERROR_TEMPLATE, tk,
                tk.getRiga());
763         }
764 
765         private NodeExpr parseExpP(NodeExpr left) throws IOException,
                TokenConstructorException, AcdcLexicalException, AcdcSyntaxException {
766             /* TODO VEDERE video 27 aprile per eventiali problemi */
767             /* TODO provare a lasciare solo SEMI nel terzo branch */
768             Token tk = scanner.peekToken();
769             NodeExpr temp;
770             switch(tk.getTipo()) {
771 
```

```
            case PLUS:
                tk = match(TokenType.PLUS);
                temp = parseTr();
                return parseExpP(new NodeBinOp(LangOper.PLUS, left, temp));
            case MINUS:
                tk = match(TokenType.MINUS);
                temp = parseTr();
                return parseExpP(new NodeBinOp(LangOper.MINUS, left, temp));
            case FLOAT:
            case INT:
            case ID:
            case PRINT:
            case SEMI:
            case EOF:
                return left;
            default:
                break;
        }

        throw new AcdcSyntaxException(AcdcSyntaxException.SYNTAX_ERROR_TEMPLATE, tk,
        tk.getRiga());
    }
}





#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@#
#@ ABSTRACT SYNTAX TREE                                                       @#
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@#


package it.gabliz.ast;

public enum LangOper {
    PLUS,MINUS,TIMES,DIV;
}

#******************************************************************************#

package it.gabliz.ast;

public enum LangType { INT, FLOAT }

#******************************************************************************#

package it.gabliz.ast;

import it.gabliz.visitor.IVisitor;

public class NodeAssign extends NodeStm{

    private NodeId id;
    private NodeExpr expr;

    public NodeAssign(NodeId id, NodeExpr expr) {
        this.id = id;
        this.expr = expr;
    }

    public NodeId getId() {
        return id;
    }

    public NodeExpr getExpr() {
        return expr;
    }
    public void setExpr(NodeExpr expr) {
        this.expr = expr;
    }
```

```
842
843         public String toString() {
844             return "ID:  " + id + ", " + "Expr: " + expr;
845         }
846
847         @Override
848         public void accept(IVisitor visitor) {
849             visitor.visit(this);
850         }
851
852     }
853
854     #*****************************************************************************#
855
856     package it.gabliz.ast;
857
858     import it.gabliz.visitor.IVisitor;
859
860     /** Classe astratta che rappresenta un nodo nell'ast */
861     public abstract class NodeAST {
862
863         private TypeDescriptor typeDescriptor;
864
865         public abstract void accept(IVisitor visitor);
866
867         public TypeDescriptor getTypeDescriptor() {
868             return typeDescriptor;
869         }
870
871         public void setTypeDescriptor(TypeDescriptor typeDescriptor) {
872             this.typeDescriptor = typeDescriptor;
873         }
874
875     }
876
877     #*****************************************************************************#
878
879     package it.gabliz.ast;
880
881     import it.gabliz.visitor.IVisitor;
882
883     public class NodeBinOp extends NodeExpr {
884
885         private LangOper op;
886         private NodeExpr left;
887         private NodeExpr right;
888
889         public NodeBinOp(LangOper op, NodeExpr left, NodeExpr right) {
890             this.op = op;
891             this.left = left;
892             this.right = right;
893         }
894
895         public LangOper getOp() {
896             return op;
897         }
898
899         public NodeExpr getLeft() {
900             return left;
901         }
902
903         public void setLeft(NodeExpr left) {
904             this.left = left;
905         }
906
907         public NodeExpr getRight() {
908             return right;
909         }
910
911         public void setRight(NodeExpr right) {
912             this.right = right;
```

```
    }

    @Override
    public String toString() {
        return "OP: " + op + "," + "left: " + left + "," + "right: " + right;
    }

    @Override
    public void accept(IVisitor visitor) {
        visitor.visit(this);
    }
}

#*************************************************************************#

package it.gabliz.ast;

import it.gabliz.visitor.IVisitor;

public class NodeConv extends NodeExpr {

    private NodeExpr n;

    public NodeConv(NodeExpr n) {
        this.n = n;
    }

    @Override
    public void accept(IVisitor visitor) {
        visitor.visit(this);
    }

    public NodeExpr getN() {
        return n;
    }

    public void setN(NodeExpr n) {
        this.n = n;
    }
}

#*************************************************************************#

package it.gabliz.ast;

import it.gabliz.visitor.IVisitor;

public class NodeCost extends NodeExpr {

    private String value;
    private LangType type;

    public NodeCost(String value, LangType type) {
        this.value = value;
        this.type = type;
    }

    public String getValue() {
        return value;
    }

    @Override
    public String toString() {
        return "value = " + value + ", type: " + type;
    }


    public void setValue(String value) {
        this.value = value;
    }

```

```
    public LangType getType() {
        return type;
    }


    public void setType(LangType type) {
        this.type = type;
    }

    @Override
    public void accept(IVisitor visitor) {
        visitor.visit(this);
    }

}

#*****************************************************************************#

package it.gabliz.ast;

import it.gabliz.visitor.IVisitor;

public class NodeDecl extends NodeDecSt {
    private NodeId id;
    private LangType type;

    public NodeDecl(NodeId id, LangType type) {
        this.id = id;
        this.type = type;
    }

    public NodeId getId() {
        return id;
    }

    public LangType getType() {
        return type;
    }

    public void setId(NodeId id) {
        this.id = id;
    }

    public void setType(LangType type) {
        this.type = type;
    }

    @Override
    public String toString() {
        return "ID: " + id + ", Type: " + type;
    }

    @Override
    public void accept(IVisitor visitor) {
        visitor.visit(this);
    }
}

#*****************************************************************************#

package it.gabliz.ast;

/** Classe che rappresenta un nodo di tipo Dec/St */
public abstract class NodeDecSt extends NodeAST { }

#*****************************************************************************#

package it.gabliz.ast;

import it.gabliz.visitor.IVisitor;

```

```
1055  public class NodeDeref extends NodeExpr {
1056      private NodeId id;
1057
1058      public NodeDeref(NodeId id) {
1059          this.id = id;
1060      }
1061
1062      public NodeId getId() {
1063          return id;
1064      }
1065
1066      public String toString() {
1067          return "ID: " + id;
1068      }
1069
1070      @Override
1071      public void accept(IVisitor visitor) {
1072          visitor.visit(this);
1073      }
1074  }
1075
1076  #*****************************************************************************#
1077
1078  package it.gabliz.ast;
1079
1080  public abstract class NodeExpr extends NodeAST {
1081  }
1082
1083  #*****************************************************************************#
1084
1085  package it.gabliz.ast;
1086
1087  import it.gabliz.symboltable.Attributes;
1088  import it.gabliz.visitor.IVisitor;
1089
1090  public class NodeId extends NodeAST {
1091
1092      private String name;
1093      private Attributes definition;
1094
1095      public NodeId (String name) {
1096          this.name = name;
1097      }
1098
1099      public String toString() {
1100          return name;
1101      }
1102
1103      public String getName() {
1104          return name;
1105      }
1106
1107      @Override
1108      public void accept(IVisitor visitor) {
1109          visitor.visit(this);
1110
1111      }
1112
1113      public Attributes getDefinition() {
1114          return definition;
1115      }
1116
1117      public void setDefinition(Attributes definition) {
1118          this.definition = definition;
1119      }
1120
1121
1122  }
1123
1124  #*****************************************************************************#
1125
```

```
package it.gabliz.ast;

import it.gabliz.visitor.IVisitor;

import java.util.ArrayList;

/** Classe che rappresenta il nodo programma */
public class NodeProgram extends NodeAST {

    private final ArrayList<NodeDecSt> n;

    public NodeProgram(ArrayList<NodeDecSt> n) {
        this.n = n;
    }

    public String toString() {
        StringBuilder s = new StringBuilder();
        for(NodeDecSt node : n) {
            s.append("<").append(node.toString()).append(">");
        }

        return s.toString();
    }


    public ArrayList<NodeDecSt> getN() {
        return n;
    }
    @Override
    public void accept(IVisitor visitor) {
        visitor.visit(this);
    }


}

#*****************************************************************************#

package it.gabliz.ast;


import it.gabliz.visitor.IVisitor;

public class NodePrint extends NodeStm {

    private NodeId id;

    public NodePrint(NodeId id) {
        this.id = id;
    }
    public NodeId getId() {
        return id;
    }
    public String toString() {
        return "Print: " + id;
    }
    @Override
    public void accept(IVisitor visitor) {
        visitor.visit(this);

    }



}

#*****************************************************************************#

package it.gabliz.ast;

public abstract class NodeStm extends NodeDecSt{
```

```
}

#******************************************************************************#

package it.gabliz.ast;

public enum TypeDescriptor {
    INT,FLOAT,ERROR,VOID
}





#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@#
#@ SYMBOL TABLE                                                               @#
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@#


package it.gabliz.symboltable;

import it.gabliz.ast.TypeDescriptor;

/**
 * Classe che rappresenta l'istanza 'attributo' per la symbol table
 * @see SymbolTable
 * @author Gabliz
 */
public class Attributes {

    private TypeDescriptor tipo;
    private char register;

    public Attributes(TypeDescriptor tipo) {
        this.tipo = tipo;
    }

    public TypeDescriptor getTipo() {
        return tipo;
    }

    public void setTipo(TypeDescriptor tipo) {
        this.tipo = tipo;
    }

    public char getRegister() {
        return register;
    }

    public void setRegister(char register) {
        this.register = register;
    }
}

#******************************************************************************#

package it.gabliz.symboltable;

import java.util.HashMap;

/**
 * Classe statica che rappresenta la symbol table del nostro progetto.
 * @see Attributes
 * @author Gabliz
 */
public class SymbolTable {
    private static HashMap<String, Attributes> table;

    public static void init() {
        table = new HashMap<>();
    }
```

```
    public static boolean enter(String id, Attributes entry) {
        Attributes value = table.get(id);
        if (value != null)
            return false;
        table.put(id, entry);
        return true;
    }

    public static Attributes lookup(String id) {
        return table.get(id);
    }

    public static String toStr() {
        StringBuilder res = new StringBuilder("symbol table\n=============\n");

        for (HashMap.Entry<String, Attributes> entry : table.entrySet())
            res.append(entry.getKey()).append("   \t").append(entry.getValue())
                    .append("\n");

        return res.toString();
    }

    public static int size() {
        return (table.size());
    }
}





#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@#
#@ VISITOR                                                                    @#
#@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@#


package it.gabliz.visitor;

import it.gabliz.ast.*;
import it.gabliz.symboltable.SymbolTable;

public class CodeGeneratorVisitor implements IVisitor {

    private static char[] reg;
    private StringBuffer codice;
    static int i = 0;

    public CodeGeneratorVisitor() {
        this.codice = new StringBuffer();
        reg = "abcdefghijklmnopqrstuvwxyz".toCharArray();
    }

    @Override
    public void visit(NodeProgram node) {
        SymbolTable.init();
        for(NodeAST currentNode : node.getN())
            currentNode.accept(this);
    }

    @Override
    public void visit(NodeAssign node) {
        NodeId id = node.getId();
        char s = id.getDefinition().getRegister();
        node.getExpr().accept(this);
        codice.append("s").append(s);
        codice.append(" 0 k ");
    }

    @Override
    public void visit(NodeBinOp node) {
```

```
1339            NodeExpr left = node.getLeft();
1340            NodeExpr right = node.getRight();
1341            left.accept(this);
1342            right.accept(this);
1343
1344            switch(node.getOp()) {
1345                case PLUS:
1346                    codice.append("+ ");
1347                    break;
1348                case MINUS:
1349                    codice.append("- ");
1350                    break;
1351                case TIMES:
1352                    codice.append("* ");
1353                    break;
1354                case DIV:
1355                    codice.append("/ ");
1356                    break;
1357                default:
1358                    break;
1359            }
1360        }
1361
1362        @Override
1363        public void visit(NodeCost node) {
1364            codice.append(node.getValue()).append(" ");
1365        }
1366
1367        @Override
1368        public void visit(NodeDecl node) {
1369            NodeId id = node.getId();
1370            id.getDefinition().setRegister(newRegister());
1371        }
1372
1373        private static char newRegister() {
1374            char c = reg[i];
1375            i++;
1376            return c;
1377        }
1378
1379        @Override
1380        public void visit(NodeDeref node) {
1381            NodeId id = node.getId();
1382            char s = id.getDefinition().getRegister();
1383            codice.append("l ").append(s).append(" ");
1384        }
1385
1386        @Override
1387        public void visit(NodeId node) {
1388
1389        }
1390
1391        @Override
1392        public void visit(NodePrint node) {
1393            NodeId id = node.getId();
1394            char s = id.getDefinition().getRegister();
1395            codice.append("l").append(s).append(" p P ");
1396        }
1397
1398        @Override
1399        public void visit(NodeConv node) {
1400            node.getN().accept(this);
1401            codice.append("5 k ");
1402        }
1403
1404        public String toString() {
1405            return codice.toString().trim();
1406        }
1407
1408    }
1409
```

```
#*****************************************************************************#

package it.gabliz.visitor;


import it.gabliz.ast.*;

public interface IVisitor {
    public abstract void visit(NodeProgram node);
    public abstract void visit(NodeAssign node);
    public abstract void visit(NodeBinOp node);
    public abstract void visit(NodeCost node);
    public abstract void visit(NodeDecl node);
    public abstract void visit(NodeDeref node);
    public abstract void visit(NodeId node);
    public abstract void visit(NodePrint node);
    public abstract void visit(NodeConv node);
}

#*****************************************************************************#

package it.gabliz.visitor;


import it.gabliz.ast.*;
import it.gabliz.symboltable.Attributes;
import it.gabliz.symboltable.SymbolTable;
import it.gabliz.util.Logger;

/*
il programma chiama il visitor di tutti i nodi e no nsi ferma al primo errore.
il promamma ha un erroe se ce qualche nodo settato con ERROR:

 */


public class TypeCheckingVisitor implements IVisitor {
    private final Logger logger;

    public TypeCheckingVisitor() {
        logger = new Logger(this.getClass().getSimpleName());
        logger.i("Inizio fase di type checking.");
    }

    @Override
    public void visit(NodeProgram node) {
        SymbolTable.init();
        for(NodeAST currentNode : node.getN()) {
            currentNode.accept(this);
        }
    }

    @Override
    public void visit(NodeAssign node) {
        NodeId id = node.getId();
        NodeExpr exp = node.getExpr();
        id.accept(this);
        exp.accept(this);

        if(id.getTypeDescriptor().equals(TypeDescriptor.INT) &&
        exp.getTypeDescriptor().equals(TypeDescriptor.INT)) {
            node.setExpr(exp);
            node.setTypeDescriptor(id.getTypeDescriptor());
        } else if( compatible(id.getTypeDescriptor(),exp.getTypeDescriptor()) ) {
            node.setExpr(convertExpr(exp));
            node.setTypeDescriptor(id.getTypeDescriptor());
        } else {
            node.setTypeDescriptor(TypeDescriptor.ERROR);
            logger.addTypeCheckingError("{NodeAssign} Impossibile assegnare
            l'esprresione.");
        }
```

```
1479        }
1480
1481        @Override
1482        public void visit(NodeBinOp node) {
1483            NodeExpr left = node.getLeft();
1484            left.accept(this);
1485            NodeExpr right = node.getRight();
1486            right.accept(this);
1487
1488            if(left.getTypeDescriptor() == TypeDescriptor.ERROR ||
                right.getTypeDescriptor() == TypeDescriptor.ERROR) {
1489                node.setTypeDescriptor(TypeDescriptor.ERROR);
1490                logger.addTypeCheckingError("{NodeBinOp} uno dei due operandi ha
                    errore!");
1491            } else if(left.getTypeDescriptor() == right.getTypeDescriptor()) {
1492                node.setTypeDescriptor(left.getTypeDescriptor());
1493            } else {
1494                if(left.getTypeDescriptor() == TypeDescriptor.INT) {
1495                    node.setLeft(convertExpr(left));
1496                } else if (right.getTypeDescriptor() == TypeDescriptor.INT) {
1497                    node.setRight(convertExpr(right));
1498                }
1499                node.setTypeDescriptor(TypeDescriptor.FLOAT);
1500            }
1501        }
1502
1503        @Override
1504        public void visit(NodeConv node) {
1505            NodeExpr n = node.getN();
1506            n.accept(this);
1507            if(n.getTypeDescriptor() != TypeDescriptor.INT) {
1508                node.setTypeDescriptor(TypeDescriptor.ERROR);
1509                logger.addTypeCheckingError("{NodeConv} Rilevata conversione non
                    consentita");
1510            } else {
1511                node.setTypeDescriptor(TypeDescriptor.FLOAT);
1512            }
1513        }
1514
1515        @Override
1516        public void visit(NodeCost node) {
1517            if(node.getType() == LangType.INT) {
1518                node.setTypeDescriptor(TypeDescriptor.INT);
1519            } else if(node.getType() == LangType.FLOAT) {
1520                node.setTypeDescriptor(TypeDescriptor.FLOAT);
1521            } else {
1522                node.setTypeDescriptor(TypeDescriptor.ERROR);
1523                logger.addTypeCheckingError("{NodeConst} la costante " + node.getValue()
                    + " è errata.");
1524            }
1525        }
1526
1527        @Override
1528        public void visit(NodeDecl node) {
1529            NodeId id = node.getId();
1530            String idName = id.getName();
1531            LangType type = node.getType();
1532            if(SymbolTable.lookup(idName) != null) {
1533                node.setTypeDescriptor(TypeDescriptor.ERROR);
1534                logger.addTypeCheckingError("{NodeDecl} l'id " + idName + " è già
                    presente nella symbol table.");
1535            } else {
1536                Attributes att;
1537                if(type.equals(LangType.INT))
1538                    att = new Attributes(TypeDescriptor.INT);
1539                else if(type.equals(LangType.FLOAT))
1540                    att = new Attributes(TypeDescriptor.FLOAT);
1541                else
1542                    att  = new Attributes(TypeDescriptor.ERROR);
1543                SymbolTable.enter(idName, att);
1544                id.setTypeDescriptor(att.getTipo());
```

```
1545                id.setDefinition(att);
1546            }
1547        }
1548
1549        @Override
1550        public void visit(NodeDeref node) {
1551            node.getId().accept(this);
1552            node.setTypeDescriptor(node.getId().getTypeDescriptor());
1553        }
1554
1555        @Override
1556        public void visit(NodeId node) {
1557            String name = node.getName();
1558            if(SymbolTable.lookup(name) == null) {
1559                node.setTypeDescriptor(TypeDescriptor.ERROR);
1560                logger.addTypeCheckingError("{NodeId} la variabile + " + name +" non è
                    dichiarata.");
1561                node.setDefinition(new Attributes(TypeDescriptor.ERROR));
1562            } else {
1563                Attributes att = SymbolTable.lookup(name);
1564                node.setDefinition(att);
1565                node.setTypeDescriptor(att.getTipo());
1566            }
1567        }
1568
1569        @Override
1570        public void visit(NodePrint node) {
1571            node.getId().accept(this);
1572            TypeDescriptor typeDescriptor = node.getId().getTypeDescriptor();
1573            node.setTypeDescriptor(typeDescriptor);
1574        }
1575
1576
1577        private boolean compatible(TypeDescriptor t1, TypeDescriptor t2) {
1578            return (!t1.equals(TypeDescriptor.ERROR) && !t2.equals(TypeDescriptor.ERROR)
                && t1.equals(t2)) || (t1.equals(TypeDescriptor.FLOAT) &&
                t2.equals(TypeDescriptor.INT));
1579        }
1580
1581        private NodeExpr convertExpr(NodeExpr node) {
1582            if(node.getTypeDescriptor() == TypeDescriptor.FLOAT) {
1583                return node;
1584            } else {
1585                NodeConv n = new NodeConv(node);
1586                n.setTypeDescriptor(TypeDescriptor.FLOAT);
1587                return n;
1588            }
1589        }
1590
1591        public String toString() {
1592            return logger.getTypeCheckingLogString();
1593        }
1594    }
1595
```