POLITECNICO DI BARI

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

TEMA D'ANNO IN

LINGUAGGI FORMALI E COMPILATORI

TRADUTTORE JAVA - ADA

Docente:
Chiar.mo Prof. Giacomo PISCITELLI

Studenti:
Domenico CAFARCHIA
Fabio IVONA

ANNO ACCADEMICO 2010/2011

Sommario

INTRODUZIONE

L'obiettivo di questo progetto è realizzare un traduttore tra due noti linguaggi di programmazione: Java e ADA. Si tratta di un sistema che, ricevuto in ingresso un programma sorgente in Java, ne verifica la correttezza lessicale (sui singoli token) e sintattica (a livello di singoli statement).

La fase di analisi sintattica è affiancata da una serie di azioni semantiche definite attraverso la formulazione di un opportuno set di regole che associano un significato ai singoli costituenti del linguaggio, consentendo pertanto di verificare la validità delle istruzioni.

Soltanto nel caso in cui tali controlli producano un esito soddisfacente, si provvederà ad eseguire la traduzione nella corrispondente sequenza di istruzioni del linguaggio ADA, operazione quest'ultima che richiede la necessità di tener conto anche della semantica del linguaggio finale che può risultare differente da quella analizzata e descritta per il linguaggio di partenza.

Nel caso di studio affrontato si è focalizzata l'attenzione sull'implementazione di un software in grado di svolgere il ruolo di analizzatore sintattico, lessicale e semantico di un linguaggio che rappresenti un sottoinsieme di Java, supportando algoritmi caratterizzati da strutture di selezione, sequenza e di ciclo e in grado di gestire una svariata gamma di operazioni built-in tra tipi predefiniti, rispettando dunque i principi del teorema di Bohm Jacopini. Ciò ovviamente rappresenta una forte restrizione delle potenzialità e delle caratteristiche peculiari del linguaggio Java, che nasce per essere completamente orientato agli oggetti e per offrire caratteristiche quali indipendenza dalla piattaforma, ereditarietà, semplicità nell'accesso e nella gestione della memoria. Si tratta tuttavia di una semplificazione resa necessaria dalle notevoli differenze sintattiche e semantiche esistenti rispetto al linguaggio ADA. In ogni caso la struttura del traduttore, e in particolar modo dei moduli di analisi sintattica e lessicale, è stata volutamente resa flessibile in maniera da lasciare aperte possibilità di sviluppo future in direzione di un'estensione dell'insieme dei costrutti supportati.

Inizialmente sviluppato verso la fine degli anni 70' su iniziativa del Dipartimento della Difesa degli Stati Uniti per lo sviluppo di applicazioni militari, oggi ADA si presenta come un linguaggio *general-purpose* che presenta delle caratteristiche molto evolute in materia di sicurezza del codice e viene impiegato in molti contesti in cui il robusto funzionamento del software è critico.

Fra le caratteristiche peculiari di Ada se ne possono citare alcune che lo avvicinano sensibilmente al

Java:

- programmazione orientata agli oggetti;
- gestione delle eccezioni;
- presenza di tipi di dati astratti.

Nelle fasi di traduzione si dimostrerà tuttavia come la conversione delle varie istruzioni porterà ad ottenere un codice fortemente diverso da quello di partenza.

La relazione è così strutturata: nel secondo capitolo verranno analizzate nel dettaglio le caratteristiche principali di ciascuna fase operativa realizzata, ponendo l'attenzione sulle modalità di funzionamento dei tool di supporto all'analisi lessicale-sintattica adottati. Nel terzo capitolo invece viene esaminata l'effettiva implementazione del sistema e l'analisi critica delle sezioni di codice ritenute più significative oltre che dei vari casi di test affrontati.

1 FASI PER LO SVILUPPO DI UN TRADUTTORE

1.1 ANALISI LESSICALE

Il primo passo per la traduzione è l'analisi lessicale del codice sorgente, realizzata attraverso uno scanner implementato ad hoc.

L'obiettivo è leggere i caratteri di input presenti nel source program e di raggrupparli opportunamente in lessemi, che rispettino determinati pattern predefiniti, producendo come output una sequenza di token.

Questo processo, cosiddetto di tokenizzazione, consente di suddividere i lessemi in categorie a seconda della loro funzione, conferendo loro un significato compiuto: di solito nella programmazione Java, il codice sorgente si può suddividere in 5 classi di token (costanti, identificatori, operatori, parole riservate e separatori.

L'identificazione dei token è realizzata attraverso la definizione di espressioni regolari, sfruttate dallo scanner per realizzare un'efficiente classificazione del flusso di lessemi ricevuto in input; le sequenze non riconducibili a nessuna delle espressioni regolari formulate sono immediatamente rilevate e segnalate come errori lessicali, per i quali è stata predisposta un'opportuna politica di gestione. Infine è opportuno che lo scanner sia in grado di svolgere ulteriori funzioni ausiliarie quali

l’eliminazione degli spazi bianchi e dei commenti (che non saranno sottoposti all’analisi sintattica), la numerazione delle righe di codice e la memorizzazione dell'input per poter localizzare eventuali errori.

Il modo più semplice per realizzare un sistema in grado di garantire le funzionalità appena descritte è implementarlo mediante un automa a stati finiti, generalmente di tipo non deterministico, in cui in ogni stato sono possibili diverse transizioni per uno stesso carattere di input; il tool di generazione dello scanner tuttavia esegue una conversione dell'automa in maniera da restituire in uscita l'implementazione di un automa deterministico. Esistono diversi strumenti per la per la realizzazione di uno scanner: quello adottato nel progetto prevede l’utilizzo di JFlex, un tool per la generazione automatica di analizzatori lessicali scritto in Java (sottoposto alla GNU General Public License) e ottenibile liberamente dall’indirizzo http://jflex.de.

La scelta di questo scanner generator è dovuta alla decisione di implementare l’intero front-end del software in Java, in maniera da renderlo portabile e machine indipendent. L’utilizzo di questo strumento richiede la scrittura di un file (con estensione .lex) contenente le specifiche del linguaggio da analizzare, le quali saranno utilizzate da JFlex per generare l’automa che si occuperà dell’analisi. Nel file vengono indicate le specifiche lessicali del linguaggio mediante un insieme di espressioni regolari e di azioni associate a ciascuna espressione, ottenendo in uscita una classe Java adibita al riconoscimento di tali espressioni.

Un file sorgente per JFlex è composto da tre sezioni distinte separate dal simbolo ‘%%’:

Codice Java definito dall’utente che viene riportato nel file .java generato.

%%

Opzioni e dichiarazioni utili alla definizione del comportamento del tool;

%%

Regole ed azioni da mettere in pratica quando una sequenza di caratteri viene riconosciuta.

1.2 ANALISI SINTATTICA

Come mostrato in figura, lo scopo dell'analisi sintattica è verificare se, data una sequenza di token, tale sequenza rispetta o meno le regole grammaticali del linguaggio sorgente.

La sintassi dunque è costituita da un insieme di regole che definiscono le frasi **formalmente** corrette e allo stesso tempo permettono di assegnare ad esse una struttura che ne indica la decomposizione nei suoi costrutti immediati.

Ciascuna regola viene definita mediante opportune produzioni aventi la seguente forma:

non_terminale ----> non_terminale non terminale…..terminale terminale | ……..

Un simbolo terminale è un elemento (token) dell'alfabeto del linguaggio, mentre un simbolo non-terminale è il risultato di una produzione della grammatica. Inoltre ciascuna regola di produzione di una grammatica context free deve essere definita evitando di generare ambiguità, produzioni ridondanti e non terminali privi di produzione corrispondente.

Una grammatica definisce un linguaggio formale come tutte le frasi costituite dai soli simboli terminali che possono essere raggiunti a partire dal simbolo iniziale (assioma) attraverso opportune derivazioni (sequenze di regole di produzione). Il parser, che esegue l'analisi sintattica, verifica pertanto se il programma sorgente rispetta le regole implicate dalla grammatica context-free di riferimento: in caso affermativo, il risultato di questa fase è la generazione di un **albero sintattico (parse tree)** che rappresenta la struttura del programma di input.

Un analizzatore sintattico è, quindi, un algoritmo che opera su sequenze di token richieste progressivamente allo scanner: se tale sequenza appartiene al linguaggio associato alla grammatica, ne produce una derivazione (raggruppa i token in frasi), altrimenti si ferma ed indica il punto dell'input dove si è presentato l'errore e la tipologia dello stesso.

I parser così definiti possono essere classificati secondo la strategia operativa e le modalità di generazione dell'albero sintattico. In questo senso possiamo distinguere la seguente classificazione:

Parser top-down: Un parser che inizia l'analisi dall'assioma e cerca di giungere alla stringa di input, individuando delle produzioni che consentono progressivamente di giungere dal simbolo iniziale ad uno dei nodi foglia. I parser top-down adottano generalmente una strategia LL (*left to*

right parser, left most derivation): si tratta di analizzatori predittivi nei quali i primi k elementi di un'espressione permettono di scegliere in modo univoco la produzione da utilizzare. L'analisi della sintassi procede da sinistra a destra, con precedenza a sinistra delle espressioni e lettura di k simboli prima della scelta della produzione corretta.

Parser bottom-up: Si tratta di analizzatori che partono dall'input, risalendo sino al simbolo iniziale, adottando solitamente una strategia LR. Un parser di tipo LR(k) (*left-to-right parser, right most derivation*) prende una decisione dopo aver realizzato la corrispondenza della sequenza di simboli con tutti gli elementi a destra dell'espressione più un numero k di altri simboli. Il nome significa un'analisi da sinistra a destra, con precedenza a destra nell'interpretazione delle espressioni e l'analisi di k elementi prima di effettuare un'azione.

L'implementazione del parser viene realizzata con l'ausilio e il supporto di un opportuno tool di generazione automatica. In realtà esistono molteplici parser generator che consentono di ottenere in output un efficiente analizzatore scritto in linguaggio Java: BYACC/J, CUP, JavaCC, etc... si è scelto di adottare CUP (*Java Based Constructor of Useful Parsers*), software open-source (sotto licenza GPL) funzionante su tutte le piattaforme e, soprattutto, in grado di generare delle classi perfettamente compatibili con quelle sviluppate mediante il tool JFlex.

CUP (così come il suo equivalente Bison in C) si presenta come un generatore di parser LALR, abbreviazione che sta per Look-Ahead LR parser, che a partire da un file .cup contenente la definizione delle regole grammaticali in notazione BNF viene generato in output l'intero analizzatore sintattico in codice Java, implementato come un automa a stati finiti deterministico.

Esso utilizza uno stack per tener traccia degli stati che il parser ha attraversato per arrivare a quello corrente e viene guidato nel suo funzionamento da una parsing table che descrive le azioni da intraprendere.

La parsing table è costituita da due parti:

Action table: a partire dallo stato in cima allo stack e dal simbolo contenuto nel buffer di input (chiamato simbolo di lookahead) restituisce l'azione da intraprendere e il nuovo stato da inserire in cima allo stack.

Goto table: consente di ottenere lo stato in cui dovrà transitare il parser quando una riduzione modifica la cima dello stack. Essa è necessaria per realizzare la conversione delle operazioni

realizzate dall'automa a stati finiti deterministico (rappresentato in forma tabellare dall'action table) in azioni equivalenti di un automa a pila. I parser LR possono essere applicati solo a parsing table senza conflitti per assicurarne il determinismo. Un *conflitto* è definito come un'entrata multipla del tipo shift/reduce oppure reduce/reduce nella parte action della parsing table. Più precisamente:

- *conflitto shift/reduce*: significa che nella stessa entrata sono presenti contemporaneamente una azione di shift e una o più azioni di reduce

- *conflitto reduce/reduce*: significa che nella stessa entrata sono presenti contemporaneamente due o più azioni reduce.

Data una grammatica context free esistono tre metodi diversi per costruirne la parsing table per un parser LR, ciascuna delle quali genera un'omonima grammatica:

- *tecnica SLR*

- *tecnica LALR*

- *tecnica LR canonica*

Un' importante distinzione formale che caratterizza queste famiglie di grammatiche è la cosiddetta *gerarchia di grammatiche LR*, che ne stabilisce l'ampiezza: la famiglia più grande è quella LR canonica che include quella LALR che a sua volta include quella SLR. Pertanto a livello di potere riconoscitivo, la tecnica LR canonica è quella più potente. Oltre che in potere riconoscitivo, le tre tecniche di costruzione parsing table si differenziano anche in termini di dimensione della tabella prodotta. Le tecniche SLR e LALR producono di norma parsing table più piccole rispetto alla tecnica LR canonica applicata alla stessa grammatica.

Ad esempio, si pensi che per la grammatica del Pascal le tabelle SLR e LALR presentano un numero di stati dell'ordine delle centinaia, mentre quella LR canonica dell'ordine delle migliaia.

In conclusione, il miglior compromesso tra potere riconoscitivo e dimensioni della tabella prodotta è offerto dalla tecnica LALR che risulta infatti quella più usata nella pratica ed è adottata dal tool CUP impiegato in questo progetto.

Un file di specifica CUP è di solito così strutturato:

Specifica del package di riferimento e delle importatzioni

componenti di codice utente: una serie di dichiarazioni opzionali attraverso le quali è possibile includere codice utente all'interno del file java che verrà generato automaticamente;

listato dei simboli terminali e non terminali;

dichiarazioni delle relazioni di precedenza: sezione opzionale per la definizione di eventuali relazioni di precedenza tra i simboli terminali. Si tratta di operazioni utili nell'ambito di grammatiche ambigue e che consentono di risolvere alcuni conflitti shift-reduce;

grammatica: definizione di tutte le produzioni che descrivono la grammatica presa in considerazione definite in Backus Naur Form.

1.3 ANALISI SEMANTICA

Le valutazioni che si possono condurre attraverso una sintassi libera sono limitate e spesso insufficienti: la sintassi non basta a definire le frasi corrette del linguaggio di programmazione, perché una stringa sintatticamente corretta potrebbe risultare priva di significato. In questo senso è bene specificare che il significato di una frase è ottenuto per mezzo di un **processo compositivo**, unendo il significato dei suoi costituenti. Ad esempio è noto che la sintassi non può esprimere le seguenti condizioni:

- In un programma ogni identificatore di variabile deve essere dichiarato prima di essere utilizzato;
- Nelle operazioni di assegnamento e/o di calcolo è necessario effettuare opportuni controlli sui tipi delle variabili e/o costanti coinvolte;
- Le variabili definite localmente all'interno di procedure, istruzioni condizionali o istruzioni di ciclo non sono visibili all'esterno di tali costrutti;

Condizioni, quelle precedenti, che sono garantite attribuendo un **significato** agli elementi sintattici e fissando delle **regole semantiche** che vengano richiamate al momento opportuno.

Data la complessità della definizione della semantica di un linguaggio, questa fase non può essere automatizzata, come invece avviene per l'analisi lessicale e sintattica. Pertanto, la definizione delle

azioni semantiche è stata realizzata manualmente mediante l'implementazione di una serie di metodi associati ai singoli costrutti presenti nella grammatica. Si è scelto, quindi, di definire una vera e propria **grammatica ad attributi,** estensione del formalismo delle grammatiche non contestuali ottenuto introducendo le nozioni di:

1. **attributi:** proprietà associate ai simboli (terminali e non terminali) della grammatica che possono essere lette o assegnate dalle azioni semantiche;

2. **azioni semantiche** e **regole semantiche:** procedure che affiancano le regole sintattiche della Grammatica.

Essendo questo progetto basato su un parser bottom-up, a ciascun simbolo risultano associati **attributi sintetizzati** il cui valore dipende solo dagli attributi presenti nel sottoalbero del nodo dell'albero sintattico a cui sono associati.

La gestione dei simboli e degli attributi ad essi associati può essere realizzata efficacemente attraverso l'utilizzo di un'apposita **tabella dei simboli:** una struttura di dati, usata dal traduttore per tener traccia dei costrutti del source program e, in particolare, della semantica degli identificatori; essa contiene un record per ogni identificatore, con campi per i suoi vari attributi, ad es. la stringa di caratteri (lessema) che lo identifica, il tipo e lo *scope*.

1.4 TRADUZIONE NEL LINGUAGGIO TARGET

Tramite le azioni semantiche, è possibile condurre la fase di traduzione in parallelo al riconoscimento della struttura sintattica del codice sorgente. Si tratta di un'operazione estremamente delicata, la cui complessità è legata principalmente alle notevoli differenze esistenti tra il linguaggio sorgente (Java) e quello destinazione (ADA). Si tratta di differenze non solo di natura sintattica, particolarmente evidenti ad esempio in istruzioni quali il ciclo for, ma anche di natura semantica (legate ad esempio a un differente approccio nella compatibilità tra tipi).

Il compilatore pertanto controlla la correttezza sintattica e semantica delle stringhe ricevute in input e ne esegue immediatamente la conversione nel linguaggio target prestabilito, provvedendo al tempo stesso ad applicare eventuali correzioni atte a risolvere alcuni problemi semantici.

1.5 TRATTAMENTO DEGLI ERRORI

E’ stata adottata una strategia di gestione degli errori che interrompa il parsing solo in presenza di un errore di natura lessicale o sintattica, provvedendo, in tal caso, a riportare l’indicazione della riga sulla quale tale errore è stato effettivamente rilevato.

Al contrario eventuali errori di natura semantica non causano il blocco dell’esecuzione del processo di compilazione ma sono comunque segnalati mediante opportuni messaggi di warning o errore.

La gestione degli errori è eseguita implementando opportune procedure associate alle produzioni critiche. La segnalazione degli errori è realizzata invece attraverso un override delle routine standard generate da CUP contestualmente alla creazione dell’analizzatore sintattico.

2 ANALISI CRITICA DEL SISTEMA IMPLEMENTATO

E' bene precisare innanzitutto come lo sviluppo dei file java.lex e java.cup che saranno descritti nei paragrafi che seguono è stata realizzata con l'ausilio del pacchetto CLE (CUP/LEX Eclipse plug-in, **http://cup-lex-eclipse.sourceforge.net/update**). Esso garantisce la possibilità di creare i file di specifica .lex e .cup direttamente all'interno dell'IDE Eclipse, occupandosi della rigenerazione del codice per lo scanner e il parser qualora i file di specifica vengano modificati.

2.1 DESCRIZIONE DEL FILE DI SPECIFICA JAVA.LEX

Un file sorgente per JFlex è suddiviso in 3 parti separate dal simbolo %%.

```
package oggetti;

import java_cup.runtime.*;
import java.io.IOException;

import oggetti.JavaSym;
import static oggetti.JavaSym.*;
```

La prima sezione include l'identificativo del package (*oggetti*) per il file sorgente (.lex) e per quello destinazione (.java) che verrà generato e una serie di direttive import necessarie al funzionamento dell'analizzatore lessicale. In particolare è indispensabile importare il package *java_cup.runtime.** che che garantisce la compatibilità con l'analizzatore sintattico CUP.

La seconda sezione consiste innanzitutto di un set di opzioni che definiscono il funzionamento dello scanner.

```
%%

%public

%class Scanner

%unicode

%line
%column
```

```
%cupsym JavaSym
%cup
%cupdebug
```

%public assegna il modificatore di accesso *public* alla classe Scanner.

%class Scanner determina il nome della classe che verrà generata.

%unicode specifica la codifica del file di testo di input.

%line e **%column** garantiscono la possibilità di abilitare il conteggio e l'identificazione rispettivamente del numero di linea e di colonna per i caratteri ricevuti in ingresso.

%cup abilita la compatibilità con il parser CUP

%cupsym JavaSym si limita a specificare il nome della classe che memorizza i token rilevati.

%cupdebug rende attive le opzioni di debug dello scanner.

Segue quindi una sezione (compresa tra ***%{*** e ***}%***) che verrà trasferita interamente all'interno della classe Scanner: essa viene introdotta per definire opportuni metodi di supporto per lo sviluppo del sistema.

In particolare, le due implementazioni del metodo *symbol* restituiscono entrambe un oggetto di classe ***java_cup.runtime.Symbol*** che verrà passato al parser durante la fase di analisi sintattica.

```
%{

  StringBuffer string = new StringBuffer();
  String rigaprec;

  private Symbol symbol(int type) {
    return new Symbol(type, yyline+1, yycolumn+1);
  }

  private Symbol symbol(int type, Object value) {
    return new Symbol(type, yyline+1, yycolumn+1, value);
  }
```

La funzione *getRigaCorrente( )*, impiegata principalmente nell'ambito della rilevazione degli errori, restituisce il contenuto della riga di input in analisi al momento della chiamata. In particolare, essa

agisce in due modi distinti, a seconda della posizione corrente raggiunta all'interno del buffer di input; informazione, quest'ultima, fornita dalla variabile *zzCurrentPos*.

- Errori sintattici quali, ad esempio, la mancanza del ";" al termine di un'istruzione di assegnamento, vengono rilevati quando l'analizzatore è ormai passato alla scansione della riga di source code successiva, separata da quella precedente mediante le sequenze di escape \n \r;
- Alternativamente può succedere che l'errore è rilevato all'interno della riga attualmente sotto analisi.

Di seguito si mostrano le strategie implementate per l'acquisizione della riga contenente un errore nei due scenari appena descritti:

```
public String getRigaCorrente(){
      String riga;
      int posizione = zzCurrentPos;
      int indice = posizione-1;

      if(zzBuffer[indice]=='\t' || zzBuffer[indice]==' '){
      while(zzBuffer[indice]=='\t' || zzBuffer[indice]==' ')
            indice--;
      while(zzBuffer[indice]=='\n' || zzBuffer[indice]=='\r' || zzBuff-
er[indice]=='\t' || zzBuffer[indice]==' ')
            indice--;

      while(zzBuffer[indice+1]!='\n' && zzBuffer[indice+1]!='\r'){
            indice++;
            System.out.println(zzBuffer[indice]);
      }


      StringBuffer buf = new StringBuffer("");
      while(zzBuffer[indice]!='\n' && zzBuffer[indice]!='\r'){
            buf.append(zzBuffer[indice]);
            indice--;
      }
      riga = buf.reverse().toString();

      }else{

            riga = "";
            for(int i=zzCurrentPos-yycolumn+1; !riga.contains("\n") && !ri-
ga.contains("\r") ; i++){
                  riga += zzBuffer[i];
            }
      }
      return riga.replace("\t", "").replace("\n", "").replace("\r", "");
  }
```

La funzione *getLinea( )*, di norma richiamata contestualmente a *getRigaCorrente( )*, restituisce il numero di riga in cui è stato individuato l'errore.

```
public int getLinea(){
      int indice = zzCurrentPos-1;
      int offset = 1;
      while(zzBuffer[indice]=='\t' || zzBuffer[indice]==' ')
            indice--;
      if(zzBuffer[indice]=='\n' || zzBuffer[indice]=='\r')
            offset--;
      while(zzBuffer[indice]=='\n' || zzBuffer[indice]=='\r' || zzBuff-
er[indice]=='\t' || zzBuffer[indice]==' '){
            indice--;
            if(zzBuffer[indice]=='\n')
                  offset--;
      }

      return yyline+offset;
  }
%}
```

La seconda sezione del file .lex consiste nelle dichiarazioni delle macro: si tratta di abbreviazioni di espressioni regolari impiegate per rendere le specifiche lessicali più semplici da leggere e trattare.

```
/* main character classes */
LineTerminator = \r|\n|\r\n
InputCharacter = [^\r\n]

WhiteSpace = [\t\f]

/* comments */
Comment = {TraditionalComment} | {EndOfLineComment} |
          {DocumentationComment}

TraditionalComment = "/*" [^*] ~"*/" | "/*" "*"+ "/"
EndOfLineComment = "//" {InputCharacter}* {LineTerminator}?
DocumentationComment = "/*" "*"+ [^/*] ~"*/"

/* identifiers */
Identifier = [:jletter:][:jletterdigit:]*

/* integer literals */
DecIntegerLiteral = 0 | [1-9][0-9]*

/* floating point literals */
DoubleLiteral = ({FLit1}|{FLit2}|{FLit3}) {Exponent}?

FLit1    = [0-9]+ \. [0-9]*
FLit2    = \. [0-9]+
FLit3    = [0-9]+
Exponent = [eE] [+-]? [0-9]+

/* string and character literals */
```

```
StringCharacter = [^\r\n\"\\]

%state STRING
```

Come appena mostrato, una dichiarazione di macro di solito è così strutturata:

```
macroidentifier = regular expression
```

E' bene specificare che le macro non possono essere ricorsive o mutualmente ricorsive, in ossequio al fatto che esse sono descrittive di linguaggi definiti su grammatiche regolari. Cicli di definizioni di macro sono pertanto individuati e opportunamente segnalati in fase di generazione automatica da parte di JFlex.

La macro `Identifier = [:jletter:][:jletterdigit:]*` (utilizzata per riconoscere i token identificatori) individua tutte quelle stringhe che cominciano con un carattere di classe *jletter* seguito da zero o più ripetizioni di caratteri di classe *jletterdigit*, dove *jletter* e *jletterdigit* sono classi di caratteri predefinite relative a tutti i possibili caratteri che possono trovarsi, rispettivamente all'inizio e all'interno di un identificatore. La sezione si conclude con la dichiarazione dello stato lessicale *STRING* ottenuta richiamando l'opzione *%state*. Tale stato agisce come una sorta di condizione iniziale: se lo scanner, durante la sua evoluzione, giunge nello stato lessicale *STRING*, allora verranno analizzate tutte e sole le espressioni regolari precedute dalla condizione iniziale <*STRING*>. Di default lo stato lessicale *YYINITIAL* è quello a partire dal quale l'analizzatore comincia l'attività di analisi. Nel caso in cui vi siano espressioni regolari prive di condizione iniziale, allora esse verranno verificate per tutti gli stati lessicali.

Infine, il file .lex prosegue con la sezione dedicata alla definizione delle regole lessicali, ovvero delle azioni (espresse mediante opportuno codice Java racchiuso tra "{" e "}") che lo scanner deve eseguire nel momento in cui incontra le espressioni regolari ad esse associate. Durante la scansione del file ricevuto in ingresso, lo scanner tiene traccia di tutte le espressioni regolari presenti nel file di specifica e attiva l'azione associata all'espressione che ha riconosciuto la sequenza più lunga. Nel caso in cui la stessa sequenza di caratteri venga identificata da due espressioni regolari distinte, lo scanner seleziona l'azione associata all'espressione che appare per prima nel file di specifica: si tratta di convenzioni adottate per risolvere eventuali ambiguità lessicali.

Di seguito è mostrato il contenuto della terza sezione del file java.lex:

```
%%

<YYINITIAL> {

  /* keywords */
  "boolean"                       { return symbol(BOOLEAN); }
  "class"                         { return symbol(CLASS); }
  "double"                        { return symbol(DOUBLE); }
  "else"                          { return symbol(ELSE); }
  "int"                           { return symbol(INT); }
  "new"                           { return symbol(NEW); }
  "if"                            { return symbol(IF); }
  "public"                        { return symbol(PUBLIC); }
  "private"                       { return symbol(PRIVATE); }
  "void"                          { return symbol(VOID); }
  "static"                        { return symbol(STATIC); }
  "while"                         { return symbol(WHILE); }
  "for"                          { return symbol(FOR); }

  /* boolean literals */
  "true"                          { return symbol(BOOLEAN_LITERAL, new
Boolean(true)); }
  "false"                         { return symbol(BOOLEAN_LITERAL, new
Boolean(false)); }

  /* null literal */
  "null"                          { return symbol(NULL_LITERAL); }


  /* separators */
  "("                             { return symbol(LPAREN); }
  ")"                             { return symbol(RPAREN); }
  "{"                             { Sym.aumentaAnnidamento(); return
symbol(LBRACE); }
  "}"                             { Sym.riduciAnnidamento(); return
symbol(RBRACE); }
  "["                             { return symbol(LBRACK); }
  "]"                             { return symbol(RBRACK); }
  ";"                             { return symbol(SEMICOLON); }
  ","                             { return symbol(COMMA); }

  /* operators */
  "="                             { return symbol(EQ); }
  ">"                             { return symbol(GT); }
  "<"                             { return symbol(LT); }
  "!"                             { return symbol(NOT); }
  "=="                            { return symbol(EQEQ); }
  "<="                            { return symbol(LTEQ); }
  ">="                            { return symbol(GTEQ); }
  "!="                            { return symbol(NOTEQ); }
  "&&"                            { return symbol(ANDAND); }
  "||"                            { return symbol(OROR); }
  "++"                            { return symbol(PLUSPLUS); }
  "--"                            { return symbol(MINUSMINUS); }
  "+"                             { return symbol(PLUS); }
  "-"                             { return symbol(MINUS); }
  "*"                             { return symbol(MULT); }
  "/"                             { return symbol(DIV); }
  "+="                            { return symbol(PLUSEQ); }
  "-="                            { return symbol(MINUSEQ); }
```

```
  "*="                           { return symbol(MULTEQ); }
  "/="                           { return symbol(DIVEQ); }

  /* string literal */
  \"                             { yybegin(STRING); string.setLength(0); }


  /* numeric literals */

  /* This is matched together with the minus, because the number is too big to
     be represented by a positive integer. */
  "-2147483648"                  { return symbol(INTEGER_LITERAL, new
Integer(Integer.MIN_VALUE)); }

  {DecIntegerLiteral}            { return symbol(INTEGER_LITERAL, new
Integer(yytext())); }
   {DoubleLiteral}                { return symbol(FLOATING_POINT_LITERAL, new
Double(yytext())); }
  {DoubleLiteral}[dD]            { return symbol(FLOATING_POINT_LITERAL, new
Double(yytext().substring(0,yylength()-1))); }


  /* comments */
  {Comment}                      {/* ignore */}


  /* whitespace */
  {WhiteSpace}                   {/* ignore */}
  {LineTerminator}               {/* ignore */}

  /* identifiers */
  {Identifier}                   { return symbol(IDENTIFIER, yytext()); }
}

<STRING> {
  \"                             { yybegin(YYINITIAL); return
symbol(STRING_LITERAL, string.toString()); }

  {StringCharacter}+             { string.append( yytext() ); }

  /* escape sequences */
  "\\b"                          { string.append( '\b' ); }
  "\\t"                          { string.append( '\t' ); }
  "\\n"                          { string.append( '\n' ); }
  "\\f"                          { string.append( '\f' ); }
  "\\r"                          { string.append( '\r' ); }
  "\\\""                         { string.append( '\"' ); }
  "\\'"                          { string.append( '\'' ); }
  "\\\\"                         { string.append( '\\' ); }

  /* error cases */
  \\.                            { System.out.println("errore lessicale"); throw
new RuntimeException("Illegal escape sequence \""+yytext()+"\""); }
  {LineTerminator}               { throw new RuntimeException("Unterminated
string at end of line"); }
}

/* error fallback */
```

```
.|\n                            { throw new RuntimeException("Illegal character
\""+yytext()+
                                                              "\" at line
"+yyline+", column "+yycolumn); }
<<EOF>>                         { return symbol(EOF); }
```

E' possibile notare che, quando viene identificata in input una keyword, un operatore o un separatore e l'analizzatore si trova nello stato YYINITIAL, viene restituito in output un oggetto CUP di classe Symbol recante l'indicazione del tipo e della posizione dello specifico token processato ed opportunamente classificato a seconda dell'espressione regolare (pattern) che viene riconosciuta. Una procedura simile è seguita per il riconoscimento dei letterali booleani o numerici per i quali si provvede a generare oggetti Symbol che tengono traccia del tipo ma anche dello specifico valore.

Nel momento in cui lo scanner identifica un doppio apice in ingresso quando si trova nello stato lessicale *YYINITIAL*, il metodo *yybegin* fa transitare l'analizzatore nello stato *STRING* abilitando il riconoscimento delle espressioni regolari ad esso associate mentre il contenuto della stringa verrà opportunamente accumulato nello StringBuffer apposito istanziato nella sezione precedente.

Ad esempio l'espressione regolare:

```
{StringCharacter}+              { string.append( yytext() ); }
```

Fa si che, se l'analizzatore si trova nello stato STRING e vengono riconosciuti caratteri ammissibili relativamente ad una stringa, tutto il contenuto del testo incontrato venga inserito nello StringBuffer attraverso un'opportuna chiamata della funzione *yytext()*. In caso di riconoscimento di un carattere illegale è lanciata un'apposita eccezione, contenente il riferimento al carattere errato. In questo senso, la sezione **/*error cases*/** è dedicata alla gestione di errori lessicali incontrati durante il riconoscimento di una stringa.

La sezione **/*error fallback*/ presenta** una sola regola lessicale (sempre attiva) utilizzata per segnalare la presenza di caratteri illegali, ovvero di caratteri che non corrispondono a nessuna della espressioni regolari precedentemente definite. Quest'ultima regola non crea conflitti con le altre perché ha la priorità più bassa in assoluto.

Una volta arrivato al termine del file di input, l'analizzatore provvederà ad attivare la regola

lessicale associata a <<*EOF*>>, notificando al parser il termine del flusso di token.

E' opportuno soffermarsi sulle azioni definite in corrispondenza del riconoscimento delle parentesi graffe:

```
"{"                         { Sym.aumentaAnnidamento(); return symbol(LBRACE); }
"}"                         { Sym.riduciAnnidamento();  return symbol(RBRACE); }
```

Oltre all'opportuna generazione di oggetti Symbol, in questi due casi vengano richiamate delle funzioni (*aumentaAnnidamento( )* e *riduciAnnidamento( )*), appartenenti alla classe Sym, dedicate al calcolo dello scope delle variabili.

2.2 DESCRIZIONE DEL FILE DI SPECIFICA JAVA.CUP E DELLE CLASSI AUSILIARIE

Di seguito si descrive il contenuto di ciascuna delle sezioni del file generato:

Sezione 1: Definisce il package di riferimento per la generazione automatica del parser e le direttive di importazione.

```
package oggetti;
import java_cup.runtime.*;
import oggetti.JavaParser;
```

Sezione 2: E' dedicata alla definizione di metodi e di variabili ricopiate poi integralmente all'interno della classe JavaParser. Questa sezione è stata sfruttata nella realizzazione di un opportuno override delle routine generate automaticamente dal tool per la gestione degli errori sintattici. Partendo dal presupposto che la definizione della grammatica è affiancata dall'implementazione delle regole semantiche e delle azioni di traduzione nel linguaggio target, si è scelto di reimplementare le procedure di default *report_error()* e *report_warning()*:

```
parser code  {:
  public void report_error(String message, Object info) {

    StringBuffer m = new StringBuffer("Error:   ");
      String listato = "";

    if (info == null)
        listato = ((Scanner)this.getScanner()).getLinea() + ": " + ((Scan-
ner)this.getScanner()).getRigaCorrente();
    else if (info instanceof java_cup.runtime.Symbol)
      listato = ((Scanner)this.getScanner()).getLinea() + ": " + in-
fo.toString();
    else if (info instanceof O)
```

```
      listato = Integer.toString(((O)info).linea) + ": " + ((O)info).riga;

    m.append(message + " (Riga " + listato + ")");



      String messaggio = m.toString();
      if(!JavaParser.conteggioerrori.containsKey(messaggio)){
            JavaParser.listaerrori.add(messaggio);
            JavaParser.conteggioerrori.put(messaggio, 1);

            if (info instanceof O)
                  Sorgente.setError(((O)info).linea);
            else
                  Sorgente.setError(((Scanner)this.getScanner()).getLinea());
    }else{
      int occorrenze = JavaParser.conteggioerrori.get(messaggio);
      occorrenze++;
      JavaParser.conteggioerrori.put(messaggio, occorrenze);
    }
    JavaParser.errori++;

  }

  public void report_error(String message){
      report_error(message, null);
  }

  public void report_warning(String message, Object info) {

    StringBuffer m = new StringBuffer("Warning: ");
    String listato = "";

    if (info == null)
        listato = ((Scanner)this.getScanner()).getLinea() + ": " + ((Scan-
ner)this.getScanner()).getRigaCorrente();
    else if (info instanceof java_cup.runtime.Symbol)
      listato = ((Scanner)this.getScanner()).getLinea() + ": " + in-
fo.toString();
    else if (info instanceof O)
      listato = Integer.toString(((O)info).linea) + ": " + ((O)info).riga;

    m.append(message + " (Riga " + listato + ")");


      String messaggio = m.toString();
    if(!JavaParser.conteggiowarning.containsKey(messaggio)){
            JavaParser.listawarning.add(messaggio);
            JavaParser.conteggiowarning.put(messaggio, 1);

      if (info instanceof O)
            Sorgente.setWarning(((O)info).linea);
      else
            Sorgente.setWarning(((Scanner)this.getScanner()).getLinea());
    }else{
      int occorrenze = JavaParser.conteggiowarning.get(messaggio);
      occorrenze++;
      JavaParser.conteggiowarning.put(messaggio, occorrenze);
    }
    JavaParser.warning++;
```

```
}

public void report_warning(String message){
    report_warning(message, null);
}
```

La logica di funzionamento delle due procedure è del tutto analoga: presentano come parametri formali il messaggio di specifica della tipologia di errore occorso ed eventualmente il nodo dell'albero presso il quale si è verificato. La rilevazione e la gestione degli errori è realizzata attraverso i seguenti passi (per la gestione dei warning vale un discorso analogo):

1. Generazione del messaggio da mostrare all'utente contornato dalla specifica del numero e dal contenuto della riga sul quale esso è occorso;
2. Accesso alla struttura *conteggioerrori (o conteggiowarning)*: si tratta di una HashMap< String, Integer> che consente di tener traccia dei messaggi di errore generati e del numero di occorrenze per ciascuno di essi. Pertanto, se il messaggio di errore generato al passo 1 non è presente in *conteggioerrori*, lo si inserisce al suo interno; altrimenti ci si limita ad incrementare di un'unità il conteggio delle occorrenze.
3. Chiamata al metodo *Sorgente.setError(int riga)* che si occupa di formattare correttamente la visualizzazione del sorgente all'interno dell'interfaccia grafica dell'applicazione, consentendo all'utente di individuare facilmente i punti del programma in cui si sono verificati dei problemi. In particolare tale metodo, ricevuto in input il numero della riga sulla quale si è verificato l'errore, impartisce che il testo della stessa venga mostrato all'utente in rosso (in caso di warning, il testo sarà mostrato in arancione).

Le due procedure di gestione si differenziano per la tipologia di errori che sono chiamate a segnalare:***report_error*** si occupa degli errori di natura semantica propri del Java, che impediscono il corretto completamento del processo di compilazione (variabili duplicate, indici di array non interi o fuori dal range di quelli consentiti, condizioni di costrutti non booleane etc..); ***report_warning*** invece è impiegata per segnalare errori semantici tipici del linguaggio ADA, per i quali il sistema tenta delle procedure di correzione automatica.

Il contenuto della classe Sorgente viene riportato di seguito:

```
package oggetti;

import java.util.ArrayList;
```

```
public class Sorgente {
      private static ArrayList<String> testo = new ArrayList<String>();

/* Accumula le righe del source program all'interno dell'ArrayList testo*/
      public static void addRiga(String riga) {
      testo.add(riga.replace(" ", " ").replace("<", "&lt;").replace(">",
"&gt;"));
      }

/* Svuota l'ArrayList*/
      public static void svuota() {
      testo = new ArrayList<String>();
      }

/* ricevuto in input il numero di linea del warning, fa si che il testo della
stessa venga riportato in arancio*/
      public static void setWarning(int riga) {
      if(riga<testo.size()) {
            riga--;
            if(!testo.get(riga).contains("<font color="))
                  testo.set(riga, "<font color=\"#FE9A2E\">" + testo.get(riga) +
"</font>");
      }
      }

/* ricevuto in input il numero di linea dell'errore, fa si che il testo della
stessa venga riportato in rosso*/

      public static void setError(int riga) {
      if(riga<testo.size()) {
            riga--;
            if(!testo.get(riga).contains("<font color="))
                  testo.set(riga, "<font color=\"red\">" + testo.get(riga) +
"</font>");
            else if(testo.get(riga).contains("<font color=\"#FE9A2E\">"))
                  testo.set(riga, testo.get(riga).replace("<font
color=\"#FE9A2E\">", "<font color=\"red\">"));
      }
      }

      public static String getTesto() {
      String output = "<PRE>";

      for(int i=0; i<testo.size(); i++) {
            output += i+1 +":\t" + testo.get(i) + "<br>";
      }

      return output + "</PRE>";
      }
}
```

La seconda sezione del file .cup si conclude con la definizione della funzione ***report_fatal_error*** adibita alla rilevazione degli errori lessicali/sintattici che determinano l'immediata interruzione del processo di compilazione.

```
public void report_fatal_error(String message, Object info) {
```

```
    report_error("Errore Sintattico/Lessicale, impossibile continuare la tradu-
zione");
    throw new RuntimeException("Fatal Syntax Error");
  }

  public void report_fatal_error(String message){
      report_fatal_error(message, null);
  }
  %}
```

Sezione 3: Si tratta di una sezione obbligatoria nella quale è necessario fornire un listato dei simboli terminali e non terminali che occorrono complessivamente all'interno delle varie produzioni. Tali dichiarazioni sono responsabili di un'opportuna denominazione e dell'assegnazione di un tipo per ogni simbolo che compare all'interno della grammatica: ciascuno di essi viene rappresentato come un oggetto di classe Symbol. Nel caso dei terminali, essi vengono ritornati dallo scanner e inseriti direttamente nello stack del parser; nel caso dei simboli non terminali invece, essi rimpiazzano la cima dello stack durante un'azione di tipo *reduce*.

La dichiarazione dei simboli della grammatica viene effettuata sfruttando la seguente sintassi:

```
terminal classname name1, name2, ...;
non terminal classname name1, name2, ...;
```

Per i simboli terminali classname rappresenta il tipo del valore associato al simbolo (se nessun classname viene specificato, allora in tal caso il simbolo avrà un'etichetta con l'indicazione di tipo NULL).

```
terminal BOOLEAN; // primitive_type
terminal INT; // integral_type
terminal DOUBLE; // floating_point_type
terminal LBRACK, RBRACK; // array_type
terminal SEMICOLON, MULT, COMMA, LBRACE, RBRACE, EQ, LPAREN, RPAREN;
terminal PUBLIC, PRIVATE; // modifier
terminal STATIC; // modifier
terminal CLASS; // class_declaration
terminal VOID; // method_header
terminal NEW; // array_creation
terminal IF, ELSE; // if_then_statement, if_then_else_statement
terminal WHILE; // while_statement
terminal FOR; // for_statement
terminal PLUSPLUS; // postincrement_expression
terminal MINUSMINUS; // postdecrement_expression
terminal PLUS, MINUS, NOT, DIV;
terminal LT, GT, LTEQ, GTEQ; // relational_expression
terminal EQEQ, NOTEQ; // equality_expression
terminal ANDAND; // conditional_and_expression
terminal OROR; // conditional_or_expression
terminal MULTEQ, DIVEQ, PLUSEQ, MINUSEQ; // assignment_operator
terminal NULL_LITERAL;
```

Come detto in precedenza, tutti i letterali sono preceduti dalla specifica del *classname* che consente di tener traccia del **tipo** associato al valore di ciascun simbolo terminale.

```
terminal java.lang.Number INTEGER_LITERAL;
terminal java.lang.Number FLOATING_POINT_LITERAL;
terminal java.lang.Boolean BOOLEAN_LITERAL;
terminal java.lang.String STRING_LITERAL;
terminal java.lang.String IDENTIFIER; // name


// 19.2) The Syntactic Grammar
non terminal O goal;
// 19.3) Lexical Structure
non terminal O literal;
// 19.4) Types, Values, and Variables
non terminal O type, primitive_type, numeric_type;
non terminal O integral_type, floating_point_type;
non terminal O reference_type;
non terminal O array_type;
// 19.5) Names
non terminal O name, simple_name;
// 19.6) Packages
non terminal O compilation_unit;
non terminal O type_declarations_opt, type_declarations;
non terminal O type_declaration;
// 19.7) Productions used only in the LALR(1) grammar
non terminal O modifiers_opt, modifiers, modifier;
// 19.8.1) Class Declaration
non terminal O class_declaration;
non terminal O class_body;
non terminal O class_body_declarations, class_body_declarations_opt;
non terminal O class_body_declaration, class_member_declaration;
// 19.8.2) Field Declarations
non terminal O field_declaration, variable_declarators, variable_declarator;
non terminal O variable_declarator_id, variable_initializer;
// 19.8.3) Method Declarations
non terminal O method_declaration, method_header, method_declarator;
non terminal O method_body;
// 19.8.4) Static Initializers
non terminal O static_initializer;
// 19.10) Arrays
non terminal O array_initializer;
non terminal O variable_initializers;
// 19.11) Blocks and Statements
non terminal O block;
non terminal O block_statements_opt, block_statements, block_statement;
non terminal O local_variable_declaration_statement, local_variable_declaration;
non terminal O statement, statement_no_short_if;
non terminal O statement_without_trailing_substatement;
non terminal O empty_statement;
non terminal O labeled_statement;
non terminal O expression_statement, statement_expression;
non terminal O if_then_statement;
non terminal O if_then_else_statement, if_then_else_statement_no_short_if;
```

```
non terminal O while_statement, while_statement_no_short_if;
non terminal O for_statement, for_statement_no_short_if;
non terminal O for_init_opt, for_init;
non terminal O for_update_opt, for_update;
non terminal O statement_expression_list;

// 19.12) Expressions
non terminal O primary, primary_no_new_array;
non terminal O array_creation_expression;
non terminal O dim_exprs, dim_expr, dims_opt, dims;
non terminal O array_access;
non terminal O postfix_expression;
non terminal O postincrement_expression, postdecrement_expression;
non terminal O unary_expression, unary_expression_not_plus_minus;
non terminal O preincrement_expression, predecrement_expression;
non terminal O multiplicative_expression, additive_expression;
non terminal O relational_expression, equality_expression;
non terminal O conditional_and_expression, conditional_or_expression;
non terminal O conditional_expression, assignment_expression;
non terminal O assignment;
non terminal O left_hand_side;
non terminal O assignment_operator;

non terminal O expression, expression_opt
```

Per i simboli non-terminali, *classname* identifica la classe di riferimento degli oggetti che rappresenteranno a tutti gli effetti i singoli nodi dell'AST. La classe di default per i nodi (Object) è stata sostituita da un'implementazione più funzionale alle necessità del progetto. Pertanto ciascun nodo dell'AST è descritto come un oggetto di classe *O* avente, tra i suoi attributi, il valore testuale del nodo e il tipo ad esso associato, oltre che il numero e il contenuto della riga del source code sul quale l'oggetto è stato identificato durante il processo di riduzione.

La necessità di questi due ultimi attributi associati a ciascun nodo verrà chiarita successivamente.

```
public class O {
	public String testo;
	public String tipo;
	public String riga;
	public int linea;


	public O(String testo, String tipo) {
		this.riga = ((Scanner)JavaParser.p.getScanner()).getRigaCorrente();
		this.linea = ((Scanner)JavaParser.p.getScanner()).getLinea();
		this.testo = testo;
		this.tipo = tipo;

	}

	public O(String testo) {
		this.riga = ((Scanner)JavaParser.p.getScanner()).getRigaCorrente();
		this.linea = ((Scanner)JavaParser.p.getScanner()).getLinea();
```

```
            this.testo = testo;
            this.tipo = "";
      }


}
```

Sezione 4: Il file di specifica si chiude con la descrizione di tutte le produzioni della grammatica[1]. Il primo simbolo della stessa è specificato attraverso l'istruzione start with *goal*; essa indica l'assioma della grammatica, ovvero il nodo nel quale devono necessariamente terminare le varie derivazioni affinché non vengano generati errori. Ogni produzione della grammatica è definita in *Backus Naur Form* e risulta caratterizzata, pertanto, da un lato sinistro e uno destro opportunamente separati dal simbolo **"::="**.

Sul lato sinistro (testa della produzione) è indicato lo specifico simbolo non-terminale risultante dalla riduzione, mentre sul lato destro (corpo della produzione) possono comparire simboli terminali, non terminali o una combinazione di entrambi. Le produzioni relative ad un medesimo non-terminale devono essere specificate contemporaneamente e opportunamente separate mediante il simbolo "|"). Implementazione delle regole semantiche e traduzione nel linguaggio target sono svolte mediante la definizione di opportune righe di codice e inserite tra i simboli {: e :}. Al fine di richiamare gli elementi che costituiscono una produzione, è possibile etichettare ciascun simbolo non terminale in modo da garantire un accesso agevole ai suoi parametri. Il valore del non-terminale verso il quale una regola riduce è implicitamente definito dall'etichetta *RESULT*: essa rappresenta un riferimento diretto al valore assunto dal lato sinistro di una produzione e ha lo stesso tipo del simbolo non-terminale cui fa riferimento.

Di seguito verranno mostrate soltanto alcune produzioni dell'intera grammatica con lo scopo di sottolineare le principali soluzioni adottate per la realizzazione dei controlli semantici e delle operazioni di traduzione (si rimanda all'appendice la consultazione dell'intero set delle regole sintattiche).

[1]*Le produzioni utilizzate per la descrizione della sintassi del sottoinsieme di Java (vd. Appendice B) sono il risultato di un'opera di revisione e restrizione dell'intera grammatica Java disponibile e completamente scaricabile dalle documentazioni di CUP.*

2.2.1. Scope delle variabili

L'analisi della validità delle variabili è gestita attraverso un'opportuna classe di supporto *Sym* che si occupa di gestire opportunamente la symbol table, tenendo traccia del livello di annidamento raggiunto all'interno del source program, provvedendo a eliminare le variabili divenute irraggiungibili.

Partendo dal presupposto che lo scope di una variabile determina:

– il blocco di codice in cui è accessibile;

– il momento in cui viene distrutta dal garbage collector;

La posizione in cui si dichiara una variabile ne determina il suo scope: variabili istanziate, ad esempio, all'interno di istruzioni condizionali o di ciclo, saranno distrutte non appena il flusso di controllo uscirà da tali costrutti. In Java l'annidamento è definito dall'apertura e la chiusura di parentesi graffe: istruzioni condizionali o di ciclo contenenti esclusivamente la dichiarazione di una variabile, e quindi teoricamente prive di parentesi graffe, sono riconosciute come prive di senso e pertanto segnalate con un messaggio di errore. Si tiene traccia, quindi, del livello di annidamento di una variabile attraverso un conteggio delle parentesi "{" (graffa sinistra) che non risultano ancora chiuse quando la variabile viene istanziata.

In presenza di una "}", la funzione `riduciAnnidamento()` provvede a decrementare il contatore del livello di annidamento e ad eliminare dalla tabella dei simboli tutte le variabili non più accessibili.

Di seguito è illustrato il contenuto dell'intera classe *Sym*:

```
package oggetti;

import java.util.Iterator;


public class Sym {
	static int annidamento = 0;

	public static void aumentaAnnidamento() {
		annidamento++;
	}

	public static void riduciAnnidamento() {
```

```
		annidamento--;

		Iterator<String> it = JavaParser.tabella.keySet().iterator();
		while(it.hasNext()) {
			String nome = it.next();
			if(JavaParser.tabella.get(nome).annidamento>annidamento) {
				JavaParser.tabella.remove(nome);
				it = JavaParser.tabella.keySet().iterator();
			}
		}
	}

	public static void add(String nome, String tipo) {
		if(!JavaParser.tabella.containsKey(nome))
			JavaParser.tabella.put(nome, new Simbolo(tipo));
		else
			JavaParser.tabella.get(nome).tipo = tipo;
	}

	public static void add(String nome) {
		if(!JavaParser.tabella.containsKey(nome))
			JavaParser.tabella.put(nome, new Simbolo(""));
	}

	public static String getTipo(String nome) {
		if(JavaParser.tabella.containsKey(nome)) {
			String tipo = JavaParser.tabella.get(nome).tipo;
			return tipo;
		}
		else {
			return "NULL";
		}
	}

	public static boolean esiste(String nome) {
		if(nome.contains(" "))
			return true;
		if(nome.contains("(") && nome.contains(")"))
			nome = nome.substring(0, nome.indexOf("("));

		if(JavaParser.tabella.containsKey(nome)){
			if(JavaParser.tabella.get(nome).tipo.equals(""))
				return false;
			else
				return true;
		}
		else{
			return false;
		}
	}
```

Tale classe presenta non solo i metodi necessari a tener traccia dello scope ma anche una serie di funzioni di supporto per la gestione della tabella dei simboli che, come sarà mostrato nel capitolo successivo, è implementata per mezzo di una HashMap :

- `add` consente l'aggiunta in tabella di un nuovo elemento non ancora presente, noti il valore ed eventualmente il tipo;

- `getTipo` restituisce una stringa di specifica del tipo di un elemento del quale viene passato il nome;
- `esiste` verifica la presenza in tabella di un elemento noto il suo identificativo.

2.2.2. Riduzione dei letterali

```
literal ::=    INTEGER_LITERAL:letterale
   {:
         RESULT = new O(Integer.toString((Integer)letterale), "INTEGER");
         Sym.add(Integer.toString((Integer)letterale), "NUMERIC");
   :}
   |     FLOATING_POINT_LITERAL:letterale
   {:
         RESULT = new O(Double.toString((Double)letterale), "FLOAT");
         Sym.add(Double.toString((Double)letterale), "FLOAT");
   :}
   |     BOOLEAN_LITERAL:letterale
   {:
         RESULT = new O(letterale?"true":"false", "BOOLEAN");
         Sym.add(letterale?"true":"false", "BOOLEAN");
   :}

   |     STRING_LITERAL:letterale
   {:
         RESULT = new O(letterale.replace("\"", "\\\""), "STRING");
         Sym.add(letterale.replace("\"", "\\\""), "STRING");

   :}
   |     NULL_LITERAL:letterale {:RESULT = new O("null", "");:}
   ;
```

I letterali, sono trattati come dei simboli terminali, la cui occorrenza abilita la produzione del non-terminale *literal* costruito come un oggetto di classe O al quale viene attribuito il valore del letterale e il tipo ad esso associato.
Si provvede a memorizzare l'occorrenza del letterale all'interno della symbol table definendo una nuova entry in cui vengono registrati il valore e il tipo del terminale analizzato.
La scelta di posticipare la popolazione della symbol table in corrispondenza dell'analisi sintattica è giustificata proprio dal fatto che, in questo modo, è possibile registrare immediatamente ciascun simbolo con gli attributi ad esso associati.

2.2.3. Dichiarazione delle variabili

La dichiarazione di variabili distingue inizialmente le dichiarazioni di array da quelle di tipi primitivi. In ciascuno dei due rami l'analisi è ulteriormente suddivisa in dichiarazioni semplici e con assegnamento. La principale problematica riscontrata è che i singoli oggetti dello statement sono definiti a livelli differenti dell'albero sintattico; pertanto, all'interno del codice utente si effettua un'ulteriore tokenizzazione, consentita dalla struttura nota dell'oggetto *variable_declarators* del nodo immediatamente inferiore. Tale operazione consente di agevolare sensibilmente l'operazione di traduzione nel linguaggio target compensando le difficoltà causate da una struttura sintattica, per questo costrutto, notevolmente differente da quella del linguaggio sorgente.

La produzione che regola la dichiarazione di variabili locali (sia per gli array che per i tipi primitivi) prevede che venga specificato il tipo della variabile seguito dall'identificatore della stessa (nel caso di dichiarazione semplice) ed, eventualmente, anche dallo specifico valore con cui è inizializzata (nel caso di dichiarazione con assegnamento). Motivo per il quale i principali controlli necessari a realizzare un'efficiente traduzione di questo tipo di statement riguardano il non terminale *variable_declarators* così definito:

```
variable_declarator ::=
            variable_declarator_id:dichiaratore {:RESULT = dichiaratore;:}
      |     variable_declarator_id:dichiaratore EQ va-
riable_initializer:assegnamento
            {:
                  RESULT = new O(dichiaratore.testo + ":=" + assegnamento.testo,
assegnamento.tipo);

            :}
      ;
```

Particolarmente significativo è il controllo della presenza di variabili duplicate: nel momento in cui si incontra la dichiarazione di una variabile si verifica se questa è già presente o meno nella tabella dei simboli; in caso affermativo viene generato un errore, altrimenti l'id e il tipo della variabile appena dichiarata vengono salvati in una nuova entry della tabella.

Nel caso della dichiarazione di un vettore, si provvede a memorizzare la dimensione dello stesso: questo accorgimento consente di poter realizzare controlli semantici sul range di validità degli indici di un vettore nell'analisi di statement successivi (si rimanda al paragrafo successivo la

trattazione dei controlli sugli array).

```
local_variable_declaration ::= type:tipo variable_declarators:dichiarazione
{:
      if(tipo.tipo.endsWith("[]")){  //Gestione degli array
         if(!Tool.controllaTipo(tipo, dichiarazione))
            parser.report_error("il tipo del vettore non è compatibile con l'as-
segnamento", dichiarazione);

            int numerodieq =  dichiarazione.testo.replaceAll("[^:=]",
"").length()/2;
               if(numerodieq==0){
                           parser.report_error("dichiarazione di vettore non
supportata", dichiarazione);
                     }
               else if(numerodieq==1){
                        // se è una dichiarazione semplice
                  String identificatore = dichiarazione.testo.split(":=")[0];
                  String vettore = dichiarazione.testo.split(":=")[1];

                        if(Sym.esiste(identificatore)){
                             parser.report_error("variabile duplicata", dichia-
razione);
                              }

                             RESULT = new O("declare\n" + identificatore + " : "
+ vettore + ";\nbegin");
                             JavaParser.blocchi++;
                             Sym.add(identificatore, tipo.tipo);
                              if(vettore.contains("..")){
                                    int indice = Inte-
ger.parseInt(vettore.substring(vettore.indexOf("..")+2, vettore.indexOf(")")));
                                    Sym.setIndiceMassimo(identificatore, indi-
ce);
                              }
             }else if(numerodieq == 2){
            // se è una dichiarazione con assegnamento
            //Type checking
                              if(!Tool.controllaTipo(tipo, dichiarazione)){
            parser.report_error("il tipo dell'array non corrisponde a quello
istanziato", dichiarazione);
                              }
            String identificatore = dichiarazione.testo.split(":=")[0];
            String vettore = dichiarazione.testo.split(":=")[1];
            String assegnamento = dichiarazione.testo.split(":=")[2];

                              if(Sym.esiste(identificatore)){
                                    parser.report_error("variabile duplicata",
dichiarazione);
                              }

                              RESULT = new O("declare\n" + identificatore + " :
" + vettore + ";\nbegin\n" + identificatore + " := " + assegnamento + ";");
                              JavaParser.blocchi++;
                              Sym.add(identificatore, tipo.tipo);
                              if(vettore.contains("..")){
                        /*si provvede a identificare la dimensione massima del
                  vettore appena istanziato*/
                                    int indice = Inte-
ger.parseInt(vettore.substring(vettore.indexOf("..")+2, vettore.indexOf(")")));
```

```
                                    Sym.setIndiceMassimo(identificatore, indi-
ce);
                              }

                        }else{
                              //ERRORE
                              parser.report_error("dichiarazione non valida",
dichiarazione);
                        }

            }else{
                        if(!dichiarazione.testo.contains(":=")){ // se è una di-
chiarazione semplice


                              if(Sym.esiste(dichiarazione.testo)){
                                    parser.report_error("variabile duplicata",
dichiarazione);
                              }

                              Sym.add(dichiarazione.testo, tipo.testo);

                              JavaParser.dichiarazioniVariabili += dichiarazio-
ne.testo + " : " + tipo.testo + ";\n";
                              RESULT = new O("");
                        }else{      // se è una dichiarazione con assegnamento
                              String identificatore = dichiarazio-
ne.testo.split(":=")[0];
                              String assegnamento = dichiarazio-
ne.testo.split(":=")[1];

                              if(Sym.esiste(identificatore)){
                                    parser.report_error("variabile duplicata",
dichiarazione);
                              }

                              if(Tool.controllaTipo(tipo, dichiarazione)){
                                    JavaParser.dichiarazioniVariabili += identi-
ficatore + " : " + tipo.testo + ";\n";
                                    RESULT = new O(identificatore + " := " + as-
segnamento + ";");
                              }else{
                                    parser.report_warning("tipi non compatibili,
verrà tentata una conversione per risolvere il problema", dichiarazione);

                                    try{
                                          O result =
Tool.convertiTipiNumerici(new O(identificatore, tipo.testo), new O(":="), new
O(assegnamento, dichiarazione.tipo));
                                          String variabile = re-
sult.testo.split(" := ")[0];
                                          String assegnamentocast = re-
sult.testo.split(" := ")[1];

                                          JavaParser.dichiarazioniVariabili +=
variabile + " : " + tipo.testo + ";\n";
                                          RESULT = new O(variabile + " := " +
assegnamentocast + ";");
                                    }catch(Exception e){
```

```
                                        parser.report_error("impossibile ef-
fettuare la conversione da " + dichiarazione.tipo + " a " + tipo.testo, dichia-
razione);
                                        JavaParser.dichiarazioniVariabili +=
identificatore + " : " + tipo.testo + ";\n";
                                        RESULT = new O(identificatore + " := "
+ assegnamento + ";");
                                    }
                                }

                                Sym.add(identificatore, tipo.testo);
                            }
                        }
                :}
        ;
```

2.2.4. Dichiarazione e gestione degli array

Gli *array* sono definiti come l'implementazione del concetto matematico di vettore-dimensionale, ossia di un oggetto con una struttura complessa le cui componenti (elementi) sono tutte dello stesso tipo. In ADA si distinguono due tipi di array:

- *array vincolati* (*constrained array*), le cui dimensioni sono impostate in fase di definizione del tipo di dato;

  Sintassi ADA : `type NOME_TIPO is array(a..b) of TIPO_COMPONENTE;`

- *array non vincolati* (*unconstrained array*), dei quali non si predefinisce la dimensione (ciò consente di trattare oggetti di dimensione diversa).

  Sintassi ADA: `type NOME_TIPO is array(INTEGER range <>) of TIPO_COMPONENTE;`

E' bene precisare che la gestione degli array varia notevolmente da Java ad ADA: in ADA sia nella soluzione vincolata che in quella non vincolata, una volta definita la dimensione di un array, questa non può essere più modificata: l'array non vincolato, consente soltanto di posticipare la definizione della dimensione del vettore rispetto al momento in cui esso viene dichiarato. In Java, al contrario, gli array possono essere manipolati in maniera decisamente più flessibile. Tenendo conto delle dovute restrizioni, si è deciso di convertire gli array dichiarati in Java in array non vincolati ADA.

Ogniqualvolta è identificata la dichiarazione di un array nel source program, essa viene opportunamente tradotta in ADA e l'occorrenza della stessa è memorizzata attraverso la creazione di una nuova entry nella tabella *dichiarazioniVettori*: essa, così come la symbol table, è strutturata come una HashMap e ha il compito di tenere traccia di tutti i tipi di array utilizzati per poi consentirne la dichiarazione in testa al codice target.

```
array_type ::=    primitive_type:tipo dims:dimensioni
      {:
            if(!JavaParser.dichiarazioniVettori.containsKey("ARRAY_" + ti-
po.testo)){
```

```
                JavaParser.dichiarazioniVettori.put("ARRAY_" + tipo.testo,
"type " + "ARRAY_" + tipo.testo + " is array(INTEGER range <>) of " + tipo.testo
+ ";");
            }
            RESULT = new O("ARRAY_" + tipo.testo, tipo.testo + "[]");
      :};
```

Ma non solo: come accennato nel paragrafo precedente, nel momento in cui è istanziato un vettore, è necessario effettuare una serie di controlli semantici sull'indice dello stesso.

In Java un array viene creato usando l'operatore new, che alloca l'array in memoria e restituisce il reference dello stesso. Ad esempio la seguente istruzione crea un nuovo array e ne inserisce il reference nella variabile unArray:

int [ ] unArray = **new int**[100];

L'istruzione new int[100] alloca un array di 100 celle, ciascuna di tipo intero, all'interno della memoria heap.

Per accedere alla cella di un array, come in altri linguaggi, si utilizza una sintassi che consente di specificare l'indice (ovvero la posizione nell'array) di tale cella (ad es. unArray[0] = 1).
Gli indici devono essere sono numeri interi consecutivi a partire da 0 (indice della prima cella): pertanto, quando si effettua l'allocazione o l'accesso agli elementi di un vettore, viene sempre effettuato un opportuno controllo semantico volto a controllare che vengano utilizzati esclusivamente indici interi.
Alternativamente è possibile allocare in memoria l'array specificando, contestualmente, l'insieme degli elementi che dovranno essere memorizzati al suo interno:

int[ ] unArray = **new int[ ]** { 3, 5, 0, 1 };

Per garantire la compatibilità con il codice ADA, in ognuno dei due casi appena descritti è importante verificare che, durante l'accesso alla cella di un array precedentemente allocato, venga utilizzato un indice valido, ovvero un intero positivo non superiore rispetto alla dimensione del vettore.

Pertanto, quando avviene l'allocazione di un vettore, si provvede a salvare nella symbol table un ulteriore attributo (*indiceMassimo*) per tener traccia del massimo elemento indicizzabile: motivo questo per il quale, come mostrato di seguito, ciascun elemento della tabella dei simboli è descritto come un oggetto di classe *Simbolo* avente tra i suoi attributi il tipo, il contatore del livello di annidamento dell'oggetto e l'identificativo *indiceMassimo*, che ovviamente verrà settato soltanto in corrispondenza dell'allocazione di array.

```
public class Simbolo {
	public String tipo;
	public int annidamento;
	public int indiceMassimo;


	public Simbolo(String tipo) {
		this.tipo = tipo;
		annidamento = Sym.annidamento;
		indiceMassimo = 0;
	}

	public void setTipo(String tipo) {
		this.tipo = tipo;
		indiceMassimo = 0;
	}
}
```

Pertanto, all'accesso ad una cella di un vettore si provvede a verificare che l'indice specificato non sia superiore al valore memorizzato in tabella dei simboli.

Inoltre si è implementato un ulteriore controllo semantico legato all'impossibilità, al contrario di Java, di dichiarare ed utilizzare in ADA array monodimensionali.

In questo caso, pertanto, verrà generato un messaggio di errore specificando l'impossibilità di completare il processo di traduzione.

Il settaggio del valore di *indiceMassimo* e la successiva lettura dello stesso vengono realizzati attraverso opportune chiamate dei metodi `setIndiceMassimo(String nome, int indice)` e `int getIndiceMassimo(String nome)` entrambi implementati nella classe Sym.

Infine, nel caso di allocazione con inizializzazione, è importante verificare che il tipo di ciascuno degli elementi contenuti tra parentesi graffe coincida con il tipo specifico dell'array (si rimanda al paragrafo successivo l'analisi delle procedure che eseguono il type checking).

```
array_creation_expression ::=
		NEW primitive_type:tipo dim_exprs:dimensioni
dims_opt:dimensioniopzionali
		{:
			RESULT = new O("ARRAY_" + tipo.testo + "(0.." + dimensio-
ni.testo + ")", tipo.testo + "[]");
		:}
	|	NEW primitive_type:tipo dims array_initializer:iniz
		{:
			if(!Tool.controllaTipo(tipo, iniz))
				parser.report_error("l'inizializzazione del vettore pre-
senta incompatibilità  nel tipo dei suoi elementi", iniz);
			int dimensioni = iniz.testo.replaceAll("[^,]", "").length();
			if(dimensioni==0)
				parser.report_error("il linguaggio target non supporta
array di dimensione unitaria", iniz);
			RESULT = new O("ARRAY_" + tipo.testo + "(0.." + Inte-
ger.toString(dimensioni) + "):=" + iniz.testo, tipo.testo + "[]");
```

```
            :}
      ;
dim_exprs ::=     dim_expr:dimensioni {:RESULT = dimensioni;:}
      |     dim_exprs dim_expr
      ;
dim_expr ::=      LBRACK expression:dimensioni RBRACK
      {:
            if(!Tool.controllaTipo(dimensioni, new O("INTEGER", "INTEGER"))){
                        parser.report_error("la dimensione di un array deve es-
sere un numero intero", dimensioni);
                  }
            RESULT = dimensioni;
      :}
      ;
// gestisce l'accesso all'elemento di un array
array_access ::=
            name:array LBRACK expression:espressione RBRACK
            {:
                  if(!Tool.controllaTipo(espressione, new O("INTEGER", "INTE-
GER"))){
                        parser.report_error("l'indice di un array deve essere un
numero intero", espressione);
                  }

                  try{
                        int indice = Integer.parseInt(espressione.testo);
                        if(indice>Sym.getIndiceMassimo(array.testo) &&
Sym.esiste(array.testo))
                              parser.report_error("l'indice dell'array supera la
dimensione del vettore", espressione);
                  }catch(Exception e){
                  }

                  RESULT = new O(array.testo + "(" + espressione.testo + ")",
array.tipo.replace("[]", ""));
            :}
      |     primary_no_new_array LBRACK expression RBRACK
      ;
```

2.2.5. Gestione del type checking

Uno dei controlli semantici più significativi e ricorrenti è il **type checking**. Per agevolarne l'implementazione è stato messo a punto un apposito Tool attraverso una classe statica contenente una routine di funzioni richiamabili quando necessario.

Di seguito è mostrato e descritto il contenuto del file Tool.java

```
public class Tool {

	public static String calcolaTipo(O sinistra, O destra) {
		String tiposinistra = sinistra.tipo;
		String tipodestra = destra.tipo;

		if(Sym.getTipo(sinistra.testo).equals("NUMERIC") && (tipode-
stra.equals("INTEGER") || tipodestra.equals("FLOAT"))) {
			return tipodestra;
		}

		if(Sym.getTipo(destra.testo).equals("NUMERIC") && (tiposini-
stra.equals("INTEGER") || tiposinistra.equals("FLOAT"))) {
			return tiposinistra;
		}
		if(tiposinistra.equals("FLOAT") && tipodestra.equals("FLOAT"))
			return "FLOAT";
		if(tiposinistra.equals("INTEGER") && tipodestra.equals("INTEGER"))
			return "INTEGER";
		if(!tiposinistra.equals(tipodestra) && (tiposinistra.equals("FLOAT")
|| tiposinistra.equals("INTEGER")) && (tipodestra.equals("FLOAT") || tipodes-
tra.equals("INTEGER")))
			return "FLOAT";
		if(tiposinistra.equals(tipodestra))
			return tiposinistra;


		return "";
	}
```

La funzione *calcolaTipo(O sinistra, O destra)* riceve in ingresso due oggetti *O* e ha come obiettivo quello di calcolare dinamicamente il tipo relativo all'oggetto risultante da un'operazione numerica. *NUMERIC* è un tipo generico che indica un qualsiasi tipo di dato numerico senza distinguere tra Integer, Float, Double, associato generalmente agli indicatori letterali numerici.

La funzione realizza i seguenti controlli:

- Se uno dei due oggetti ha tipo NUMERIC mentre l'altro INTEGER (o FLOAT), allora sarà restituito un oggetto dello stesso tipo dell'elemento INTEGER (o FLOAT);
- Se gli oggetti presentano lo stesso tipo, il calcolo è immediato e viene restituito lo stesso tipo di partenza;
- Se un oggetto è di tipo FLOAT e l'altro di tipo INTEGER, l'operazione è consentita e l'oggetto risultante risulterà essere di tipo FLOAT.
- In caso di incompatibilità tra gli operandi il metodo restituirà una stringa vuota.

A differenza di *calcolaTipo*, che si limita a calcolare il tipo dell'oggetto risultante da un'operazione, `controllaTipo(O sinistra, O destra)` è una funzione booleana che verifica unicamente la compatibilità tra due operandi.

```
	public static boolean controllaTipo(O sinistra, String tipo) {
		return controllaTipo(sinistra, new O(tipo, tipo));
	}

	public static boolean controllaTipo(O sinistra, O destra) {
		if((Sym.getTipo(sinistra.testo).equals("NUMERIC") || sini-
stra.tipo.equals("NUMERIC")) && (destra.tipo.equals("INTEGER") || de-
stra.tipo.equals("FLOAT"))) {
			return true;
		}

		if((Sym.getTipo(destra.testo).equals("NUMERIC") || de-
stra.tipo.equals("NUMERIC")) && (sinistra.tipo.equals("INTEGER") || sini-
stra.tipo.equals("FLOAT"))) {
			return true;
		}


		if(destra.tipo.equals(sinistra.tipo))
			return true;


		return false;
	}
```

La funzione `controllaOperatore(String tipo1, String operatore)` presenta in ingresso due stringhe: il tipo di un operando e il valore dell'operatore. La funzione restituisce **true** soltanto nel caso in cui l'operatore è effettivamente compatibile con il tipo dell'operando (di cui è stata già verificata la compatibilità con il tipo del secondo operando).

```
public static boolean controllaOperatore(String tipo1, String operatore) {
		if(operatore.equals("+") ||operatore.equals("-")
||operatore.equals("*") ||operatore.equals("/")  || operatore.equals("=") ||
operatore.equals("+=") || operatore.equals("-=") || operatore.equals("*=") ||
operatore.equals("/=")) {
			if(tipo1.equals("INTEGER") || tipo1.equals("FLOAT"))
				return true;
		}


		if(operatore.equals("+") && tipo1.equals("STRING"))
			return true;

		if(operatore.equals(":="))
			return true;

		return false;
	}
```

Oltre alle funzioni di analisi e calcolo del tipo degli operatori, la funzione

convertiTipiNumerici(O sinistra, O operatore, O destra) è sfruttata dai controlli semantici sul tipo propri esclusivamente dell'ADA che, come detto in precedenza, prevede una tipizzazione decisamente più rigida rispetto al linguaggio Java. La funzione riceve in input 3 oggetti di classe O (operandi e operatore) e genera in output un opportuno oggetto O risultante da una traduzione coerente in ADA dell'intera istruzione.

Trattandosi di controlli che riguardano esclusivamente il linguaggio target, essi prevedono che, in caso di rilevamento errori, questi vengano segnalati (attraverso dei warning) e opportunamente corretti dove possibile.

Nelle operazioni di assegnamento si procede come segue:

- Se a un oggetto di tipo FLOAT viene assegnato un valore di tipo INTEGER, quest'ultimo viene opportunamente convertito in FLOAT;
- Se viceversa a un oggetto di tipo INTEGER viene assegnato un valore di tipo FLOAT, si provvede a trasformare quest'ultimo in INTEGER mediante un'opportuna operazione di arrotondamento.

Queste operazioni di conversione vengono svolte mediante opportune chiamate della funzione *convertiTipo* che sarà mostrata in coda al paragrafo.

```
      public static O convertiTipiNumerici(O sinistra, O operatore, O destra)
throws Exception {

      if(operatore.testo.contains("=") && !operatore.testo.equals("=") && !oper-
atore.testo.equals("/=")) {
                  if(sinistra.tipo.equals("INTEGER") && des-
tra.tipo.equals("FLOAT")) {
                        return new O(sinistra.testo + " " + (operato-
re.testo.equals(":=")?operatore.testo:":= " + sinistra.testo + " " + operato-
re.testo.substring(0,1)) + " " + convertiTipo(destra, "INTEGER").testo, sini-
stra.tipo);
                  }else if(sinistra.tipo.equals("FLOAT") && des-
tra.tipo.equals("INTEGER")) {
                        return new O(sinistra.testo + " " + (operato-
re.testo.equals(":=")?operatore.testo:":= " + sinistra.testo + " " + operato-
re.testo.substring(0,1)) + " " + convertiTipo(destra, "FLOAT").testo, sini-
stra.tipo);
                  }else{
                        return new O(sinistra.testo + " " + (operato-
re.testo.equals(":=")?operatore.testo:":= " + sinistra.testo + " " + operato-
re.testo.substring(0,1)) + " " + destra.testo, sinistra.tipo);
                  }
```

In presenza di operazioni aritmetiche il tipo degli operandi viene convertito in quello meno "restrittivo" presente nell'istruzione: pertanto in operazioni aritmetiche tra FLOAT e INTEGER, l'oggetto di tipo INTEGER verrà sempre convertito in FLOAT.

```
      }else if(operatore.testo.equals("+") || operatore.testo.equals("-") ||
operatore.testo.equals("*") || operatore.testo.equals("/")) {

                  if(sinistra.tipo.equals("INTEGER") && des-
tra.tipo.equals("FLOAT")) {
                        return new O(convertiTipo(sinistra, "FLOAT").testo + " "
+ operatore.testo + " " + destra.testo, destra.tipo);
                  }else if(sinistra.tipo.equals("FLOAT") && des-
tra.tipo.equals("INTEGER")) {
                        return new O(sinistra.testo + " " + operatore.testo + "
" + convertiTipo(destra, "FLOAT").testo, sinistra.tipo);
                  }
```

Discorso a parte va fatto per gli operatori relazionali, per i quali il linguaggio ADA impone ulteriori restrizioni per quel che concerne il tipo delle variabili coinvolte:

- Gli operatori di uguaglianza ("=") e di disuguaglianza ("/=") necessitano che gli operandi siano o entrambi dello stesso tipo;
- Gli operatori ">", "<" , ">=", "<=" invece possono essere applicati esclusivamente ad operandi scalari.

```
}else if(operatore.testo.equals("<") || operatore.testo.equals(">") || operato-
re.testo.equals("<=") || operatore.testo.equals(">=")){

      if(sinistra.tipo.equals("INTEGER") && destra.tipo.equals("FLOAT")) {
         return new O(sinistra.testo + " " + operatore.testo + " " + convertiTi-
      po(destra, "INTEGER").testo, "BOOLEAN");

      }else if(sinistra.tipo.equals("FLOAT") && destra.tipo.equals("INTEGER")) {
         return new O(convertiTipo(sinistra, "INTEGER").testo + " " + operato-
      re.testo + " " + destra.testo, "BOOLEAN");

      }else if(sinistra.tipo.equals("FLOAT") && destra.tipo.equals("FLOAT")) {
         return new O(convertiTipo(sinistra, "INTEGER").testo + " " + operato-
      re.testo + " " + convertiTipo(destra, "INTEGER").testo, "BOOLEAN");
                  }

      }else if(operatore.testo.equals("=") || operatore.testo.equals("/=")){


           if(sinistra.tipo.equals("INTEGER") && destra.tipo.equals("FLOAT")) {
                  return new O(convertiTipo(sinistra, "FLOAT").testo + " " +
      operatore.testo + " " + destra.testo, "BOOLEAN");
if(controllaTipo(sinistra, destra))
                  return new O(sinistra.testo + " " + operatore.testo + " " +
destra.testo, sinistra.tipo);
            else
                  throw new Exception();
      }
```

Come accennato in precedenza, la funzione *convertiTipo* si occupa di realizzare tutte le conversioni alla base delle procedure descritte in precedenza.
Essa riceve in input un oggetto di classe O e una stringa recante l'identificativo di uno specifico tipo e fa si che venga restituito in output un nuovo oggetto O risultante dalla conversione.
Conversioni del tipo FLOAT di un oggetto O in tipo INTEGER si traducono in operazioni di arrotondamento del valore dell'oggetto; viceversa conversioni dal tipo INTEGER di un oggetto O in tipo FLOAT si traducono nell'aggiunta della parte decimale al valore dell'oggetto.

```
      public static O convertiTipo(O var, String tipodest){

            if(var.tipo.equals("FLOAT") && tipodest.equals("INTEGER")){
                  try{
                        int valore = (int) round(Double.parseDouble(var.testo));
                        if(var.testo.contains("."))
                              JavaParser.p.report_warning("nella conversione è
stato apportato un arrotondamento");
                        var.testo = Integer.toString(valore);

                  }catch(Exception e){
                        var.testo = "Integer(" + var.testo + ")";
                  }
                  var.tipo = "INTEGER";

            }else if(var.tipo.equals("INTEGER") && tipodest.equals("FLOAT")){
                  try{
                        int valore = Integer.parseInt(var.testo);
                        var.testo = Integer.toString(valore) + ".00";
                  }catch(Exception e){
                        var.testo = "Float(" + var.testo + ")";
                  }
                  var.tipo = "FLOAT";
            }

            return var;
      }
```

2.2.6. Traduzione dei cicli for

La sintassi del ciclo for ADA è la seguente:

```
 for PARAMETRO in TIPO range MINIMO..MASSIMO loop
    istruzione_1;
    istruzione_2;
    ...
    istruzione_n;
  end loop;
```

Oss: TIPO range è un segmento del tutto opzionale.

PARAMETRO è un identificatore dichiarato automaticamente (trattato come una costante nella sequenza delle istruzioni) mentre MINIMO e MASSIMO devono essere dello stesso tipo.

E' evidente che il ciclo for ADA presenta una sintassi decisamente differente che comporterebbe un notevole lavoro di tokenizzazione e riconversione dell'intero statement.

Inoltre in Java si potrebbe utilizzare come condizione iniziale il valore assunto da una variabile intera istanziata in precedenza, cosa che invece non è ammissibile in ADA, la cui sintassi prevede che venga utilizzato un indice di incremento interno al for e che, pertanto, verrà automaticamente distrutto all'uscita dallo stesso.

Inoltre, in un caso simile:

```
...
   N : INTEGER;
   ...
   N := 10;
   for i in INTEGER range 1..N loop
      Ada.Integer_Text_IO.Put(Item => i, Width => 0);
      Ada.Text_IO.New_Line;
      N := 5;
   end loop;
   …
```

A differenza del Java, ADA prevede che il ciclo for ... loop venga ripetuto comunque 10 volte (ossia il valore di N quando inizia il ciclo), anche se N viene modificato all'interno del ciclo.

A seguito di tali considerazioni si è scelto di tradurre i cicli for Java in cicli while ADA, che presentano la seguente sintassi:

```
while condizione loop
istruzione_1;
istruzione_2;
...
istruzione_N;
end loop;
```

La logica di funzionamento garantisce la possibilità di utilizzare variabili di incremento istanziate in precedenza ed elimina il problema di non poter modificare dinamicamente il valore del contatore all'interno del ciclo come mostrato dal seguente esempio.

```
...
  i : INTEGER;
  ...
  i := 0;
  while i < 6 loop
    Ada.Integer_Text_IO.Put(Item => i);
    i := i + 2;
  end loop;
```

…

Di seguito vengono mostrate le produzioni relative ai cicli for Java e le azioni adottate (oltre a quelle semantiche di type checking) per realizzare un'efficiente traduzione:

```
for_statement ::=
      FOR LPAREN for_init_opt:inizializzazione SEMICOLON
   expression_opt:condizione SEMICOLON
            for_update_opt:aggiornamento RPAREN statement:istruzioni
      {:
            if(!Tool.controllaTipo(condizione, new O("", "BOOLEAN"))
   && !condizione.testo.equals("")){
                  parser.report_error("E' necessario inserire un espressione
   booleana come condizione di un while");
            }
            RESULT = new O( inizializzazione.testo + "\n" +
   (!condizione.testo.equals("")?("while " + condizione.testo + " "):"") +
   "loop\n" + istruzioni.testo + "\n" + aggiornamento.testo + "\nend loop;" );
      :}
      ;

   for_statement_no_short_if ::=
      FOR LPAREN for_init_opt:inizializzazione SEMICOLON
   expression_opt:condizione SEMICOLON
            for_update_opt:aggiornamento RPAREN statement_no_short_if:istruzioni
      {:
            if(!Tool.controllaTipo(condizione, new O("", "BOOLEAN"))
   && !condizione.testo.equals("")){
                  parser.report_error("errore: é necessario inserire un
   espressione booleana come condizione di un while");
            }
            RESULT = new O( inizializzazione.testo + "\n" +
   (!condizione.testo.equals("")?("while " + condizione.testo + " "):"") +
   "loop\n" + istruzioni.testo + "\n" + aggiornamento.testo + "\nend loop;" );
      :}
      ;
   for_init_opt ::= {:RESULT = new O("");:}
      |     for_init:iniz {:RESULT = iniz;:}
      ;
   for_init ::=   statement_expression_list
      |     local_variable_declaration:iniz {:RESULT = new O(iniz.testo +
   ";");:}
      ;
   for_update_opt ::= {:RESULT = new O("");:}
      |     for_update:upd {:RESULT = upd;:}
      ;
   for_update ::= statement_expression_list:upd {:RESULT = upd;:}
      ;
   statement_expression_list ::=
      statement_expression:expr {:RESULT = new O(expr.testo + ";");:}
      |     statement_expression_list:expr1 COMMA statement_expression:expr2
      {:
            RESULT = new O(expr1.testo + "\n" + expr2.testo + ";");
      :}
      ;
```

2.3 ANALISI E DESCRIZIONE DELLA CLASSE JAVAPARSER.JAVA

La classe *JavaParser* è strutturata in maniera estremamente semplice, presentando al suo interno esclusivamente una chiamata al metodo `parse()`.
Pertanto è bene focalizzare l'attenzione sugli attributi e sull'unico metodo che caratterizzano questa classe:

```
package oggetti;

import java.io.FileReader;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;


public class JavaParser {
      public static StringBuffer buffer;
      public static HashMap<String, Simbolo> tabella;
      public static HashMap<String, String> dichiarazioniVettori;
      public static String dichiarazioniVariabili;
      public static HashMap<String, Integer> conteggioerrori;
      public static ArrayList<String> listaerrori;
      public static HashMap<String, Integer> conteggiowarning;
      public static ArrayList<String> listawarning;
      public static int blocchi = 0;
      public static int errori;
      public static int warning;
      public static String output;
      public static String messaggi;
      public static JavaCup p;
```

Nel seguito è mostrato il funzionamento del metodo *`parse(String[] argv)`*.
Innanzitutto si osserva che essa richiede in input un vettore di stringhe, ciascuna delle quali rappresenta il path dello specifico file .java che si desidera venga opportunamente analizzato e tradotto.

```
      public static String[] parse(String[] argv) {
            conteggioerrori = new HashMap<String, Integer>();
            listaerrori = new ArrayList<String>();
            conteggiowarning = new HashMap<String, Integer>();
            listawarning = new ArrayList<String>();
            errori = 0;
            warning = 0;
            blocchi = 0;
            tabella = new HashMap<String, Simbolo>();
            dichiarazioniVettori = new HashMap<String, String>();
            dichiarazioniVariabili = "";
            output = "";
            messaggi = "";
            buffer = new StringBuffer("");
```

Per ciascun file .java di input, la funzione avvia l'analisi e la progressiva traduzione in linguaggio ADA. A tale scopo viene innanzitutto creato un oggetto di classe Scanner che si occupa della tokenizzazione del testo presente nel file letto; quindi l'output di tale oggetto viene utilizzato per la creazione del parser (oggetto di classe JavaCup).
Il metodo *parse()* di classe *JavaCup* avvia l'analisi sintattica, semantica e l'attività di traduzione.

```
for (int i = 0; i < argv.length; i++) {

            try {
                  System.out.println("***Parsing ["+argv[i]+"]***");
                  messaggi += "***Parsing ["+argv[i]+"]***\n\n";
                  output += "---["+argv[i]+"]---\n\n";
                  Scanner s = new Scanner(new FileReader(argv[i]));
                  p = new JavaCup(s);


                  p.parse();
```

Si è già ampiamente detto come l'attività di traduzione venga svolta contestualmente alla fase di analisi sintattico-semantica; tuttavia è bene osservare che un source program ADA di solito presenta un'impostazione strutturale differente da quella di un programma scritto in linguaggio Java, differenze, queste ultime, che non possono essere assolutamente trascurate affinché il processo di traduzione porti alla generazione di file .adb compilabili.

La struttura generale di un programma ADA può essere così schematizzata:

```
procedure <nome> (<lista_parametri>) is
- sequenza di dichiarazioni
begin
-sequenza di istruzioni
[exception -gestori delle eccezioni-]
end [<nome>]
```

ADA prevede che ogni source program sia introdotto da un'apposita sezione dichiarativa all'interno della quale effettuare la dichiarazione di tutte le variabili che verranno effettivamente impiegate, a differenza del Java che invece garantisce all'utente un più elevato livello di flessibilità in questo senso.
Segue quindi la sequenza di istruzioni in cui si articola il programma: qui si possono eventualmente individuare veri e propri *blocchi di istruzioni* costituiti da opportune sequenze di istruzioni posizionate in una qualunque parte del programma ed eventualmente contenenti una parte dichiarativa introdotta dalla parola riservata **declare**; le istruzioni del blocco sono racchiuse tra **begin** e **end** come di seguito mostrato:

```
blocco::= declare
```

```
  dichiarazioni
begin
  istruzioni
end;
```

E' importante distinguere tra *variabile globale* e *variabile locale*: nel primo caso la variabile ha valore in tutto il programma, nel secondo caso solo all'interno del blocco in cui essa è stata dichiarata.
La costruzione del programma in linguaggio target prevede quindi innanzitutto la definizione di una sezione dichiarativa.
L'impiego di array non vincolati tuttavia, fa si che nella sezione dichiarativa globale ci si limiti a dichiarare soltanto il tipo degli array (e non la dimensione degli stessi) e degli elementi in esso contenuti mediante la seguente sintassi:

```
type TIPO_VETTORE is array(INTEGER range <>) of TIPO_ELEMENTI;
```

In questo modo l'allocazione vera e propria degli array può essere posticipata consentendo all'utente di rimandare la scelta della dimensione, fatto questo importante se si tiene conto che il linguaggio ADA non consente successive variazioni del numero di elementi di un array.
Pertanto, ogniqualvolta si incontra la dichiarazione di un array nel source program Java, la traduzione nel codice ADA avviene attraverso i seguenti due passi:

- Definizione del tipo dell'array nella parte dichiarativa globale (se questa non è stata ancora realizzata);
- Apertura di un nuovo blocco nella sezione istruzioni nel quale poter realizzare la dichiarazione locale del vettore con la specifica della dimensione dello stesso.

Questa soluzione consente inoltre di risolvere una significativa differenza tra i due linguaggi, che genera un problema nel caso in cui sia istanziato un array la cui dimensione è definita da una variabile. Il codice Java seguente:

```
int i = 2;
i = 7;
double[] vettore = new double[i];
```

potrebbe essere tradotto, in maniera semplicistica, in questo modo:

```
procedure MAIN is
begin
      type ARRAY_DOUBLE is array(INTEGER range <>) of INTEGER;
declare
```

```
      i : INTEGER := 2
      vettore : ARRAY_INTEGER(0..i);

begin

i := 7;

end;
end MAIN;
```

Dal codice di output è evidente che una traduzione del genere è inaccettabile, poiché il vettore verrebbe definito prima che la variabile `i` sia effettivamente impostata al valore 7. Sfruttando l'annidamento di molteplici blocchi, invece, si ha la possibilità di posticipare la definizione dell'array fino al momento in cui è reso disponibile l'effettivo valore della variabile `i`:

```
procedure MAIN is
begin
      type ARRAY_DOUBLE is array(INTEGER range <>) of INTEGER;
declare
      i : INTEGER := 2

begin

      i := 7;

declare
      vettore : ARRAY_INTEGER(0..i);
begin

[...]

end;

end;
end MAIN;
```

Infine il metodo `scriviMessaggi()` si occupa della scrittura di tutti i messaggi di errore e/o di warning (accumulati rispettivamente negli ArrayList *listaerrori* e *listawarning*) con l'indicazione del numero di occorrenze per ciascuno di essi (riportate nel caso in cui uno stesso errore relativo ad una specifica riga viene rilevato un numero di volte > 1).

```
private static void scriviMessaggi(){
      for(int i=0; i<listaerrori.size(); i++){
            int occorrenze = conteggioerrori.get(listaerrori.get(i));
            messaggi += listaerrori.get(i) + (occorrenze>1?(" (x" + occorrenze +
            ")"):"") + "\n";
      }
      for(int i=0; i<listawarning.size(); i++){
            int occorrenze = conteggiowarning.get(listawarning.get(i));
```

```
            messaggi += listawarning.get(i) + (occorrenze>1?" (x" + occorrenze +
            ")":"") + "\n";
      }
}
```

3 CASI DI TEST

3.1 DICHIARAZIONE DI VARIABILI

3.1.1 Caso 1

Input:

```
public class Prova {

      public static void main() {

      double a = 1.5;
      boolean d;

      int c = 2.5;

      double b = .5;
      boolean e = true;

      int z;

      double i = 1;

      int[] vettore = new int[10];
      double[] vettoredouble = new double[2];
      double[] vettoredouble2 = new double[]{1.5,6.3};
      d = false;
  }

}
```

Output:

```
--[C:\Users\Domenico\workspace\Progetto\prova2.java]--

Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 8: int c = 2.5;)
Warning: nella conversione è stato apportato un arrotondamento (Riga 8: int c =
2.5;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 15: double i = 1;)
Traduzione terminata, sono stati rilevati 3 warning

procedure MAIN is
begin
declare
type ARRAY_INTEGER is array(INTEGER range <>) of INTEGER;
type ARRAY_FLOAT is array(INTEGER range <>) of FLOAT;
a : FLOAT;
d : BOOLEAN;
c : INTEGER;
```

```
b : FLOAT;
e : BOOLEAN;
z : INTEGER;
i : FLOAT;

begin


a := 1.5;

c := 3;
b := 0.5;
e := true;

i := 1.00;
declare
vettore : ARRAY_INTEGER(0..10);
begin
declare
vettoredouble : ARRAY_FLOAT(0..2);
begin
declare
vettoredouble2 : ARRAY_FLOAT(0..1);
begin
vettoredouble2 := (1.5,6.3);
d := false;

end;

end;

end;

end;
end MAIN;
```

3.1.2 Caso 2

Input:

```
public class Prova {

      public static void main() {

      double a = 1.5;

      double b = 3.5;

      double c = 3;

      int z = 50;

      double[] vettore = new double[7];

      double[] d = new double[]{1};

      int *d;

       }
```

```
}
```

Output:

```
--[C:\Users\Domenico\workspace\Progetto\prova2.java]--


Traduzione interrotta alla riga 17, sono stati rilevati:
2 errori
1 warning

Error:   l'inizializzazione del vettore presenta incompatibilità  nel tipo dei
suoi elementi (Riga 15: double[] d = new double[]{1};)
Error:   Errore Sintattico/Lessicale, impossibile continuare la traduzione (Riga
17: int)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 9: double c = 3;)
```

3.1.3 Caso 3

Input:

```
public class Prova {

      public static void main() {

      int[] vettore = new int[10];
      int b = 0;
      double a = 1.5;

      double c = b/3;

      int d = (a*2)+(b*10.5)/c-4;


        }
}
```

Output:

```
--[C:\Users\Domenico\workspace\Progetto\prova2.java]--
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 9: double c = b/3;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 11: int d = (a*2)+(b*10.5)/c-4;) (x3)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 11: int d = (a*2)+(b*10.5)/c-4;)
Traduzione terminata, sono stati rilevati 5 warning

procedure MAIN is
begin
declare
type ARRAY_INTEGER is array(INTEGER range <>) of INTEGER;
b : INTEGER;
a : FLOAT;
```

```
c : FLOAT;
d : INTEGER;

begin


declare
vettore : ARRAY_INTEGER(0..10);
begin
b := 0;
a := 1.5;
c := Float(b / 3);
d := Integer(( a * 2.00 ) + ( Float(b) * 10.5 ) / c - 4.00);

end;

end;
end MAIN;
```

3.1.4 Caso 4

Input:

```
public class Prova {

  public static void main() {

            double a = 3.5;


            double[] vettoredouble = new int[3];
            int[] vet = new int[10];
            int b = 0;
            double c = 1.5;

            vet[15] = a;
            vect[2] = 10;

            double d = b/3;

            boolean a = false;
            int[] vettore = new int[2.5];

  }

}
```

Output:

```
--[C:\Users\Domenico\workspace\Progetto\prova2.java]--

Error:   il tipo del vettore non è compatibile con l'assegnamento (Riga 8:
double[] vettoredouble = new int[3];)
Error:   l'indice dell'array supera la dimensione del vettore (Riga 13: vet[15]
= a;)
Error:   E' stata usata una variabile non dichiarata (Riga 14: vect[2] = 10;)
Error:   variabile duplicata (Riga 18: boolean a = false;)
Error:   la dimensione di un array deve essere un numero intero (Riga 19: int[]
vettore = new int[2.5];)
```

```
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 13: vet[15] = a;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 14: vect[2] = 10;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 16: double d = b/3;)

Traduzione interrotta alla riga 19, sono stati rilevati:
5 errori
3 warning
```

3.1.5 Caso 5

Input:

```
public class Prova {

      public static void main() {
      double a = 3.5;
      int b = 0;
      double c = 1.5;
      double d = b/3 * c--;
      int[] vettore = new int[10];
      vettore[0] = (4+a)*c+(b-10);
      c++;
      b +=10;
      d -= 25;
      vettore[0]*= 12;
      a /= 3.5;
      vettore[2] = vettore[0]/vettore[1]*c;
        }
}
```

Output:

```
--[C:\Users\Domenico\workspace\Progetto\prova2.java]--

Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 8: double d = b/3 * c--;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 10: vettore[0] = (4+a)*c+(b-10);) (x2)
Warning: tipi non compatibili, verrà  tentata una conversione per risolvere il
problema (Riga 10: vettore[0] = (4+a)*c+(b-10);)
Warning: tipi non compatibili, verrà  tentata una conversione per risolvere il
problema (Riga 13: d -= 25;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 16: vettore[2] = vettore[0]/vettore[1]*c;)
Warning: tipi non compatibili, verrà  tentata una conversione per risolvere il
problema (Riga 16: vettore[2] = vettore[0]/vettore[1]*c;)

Traduzione terminata, sono stati rilevati 7 warning

procedure MAIN is
begin
declare
type ARRAY_INTEGER is array(INTEGER range <>) of INTEGER;
a : FLOAT;
b : INTEGER;
```

```
c : FLOAT;
d : FLOAT;

begin


a := 3.5;
b := 0;
c := 1.5;
d := Float(b / 3) * c - 1.00;
declare
vettore : ARRAY_INTEGER(0..10);
begin
vettore(0) := Integer(( 4.00 + a ) * c + Float(( b - 10 )));
c := c + 1.00;
b := b + 10;
d := d - 25.00;
vettore(0) := vettore(0) * 12;
a := a / 3.5;
vettore(2) := Integer(Float(vettore(0) / vettore(1)) * c);

end;

end;
end MAIN;
```

3.2 COSTRUTTI CONDIZIONALI

3.2.1 Caso 1

Input:

```
public class Prova {

  public static void main() {

      double a = 1.5;

      int c = 0;

      int b;

      boolean[] vettore = new boolean[10];

      if(a < c){
      vettore[c] = true;
      c++;
      }
      else if(a > 1 && c != 0){
            a++;
            b = 20;}
            else if(c >= 0 || b == 10)
                  c = 20;

        }

}
```

Output:

```
--[C:\Users\Domenico\workspace\Progetto\prova2.java]--

Warning: operatore non compatibile verrà tentata una conversione per risolvere
il problema (Riga 13: if(a < c){)
Warning: operatore non compatibile verrà tentata una conversione per risolvere
il problema (Riga 17: else if(a > 1 && c != 0){)
Traduzione terminata, sono stati rilevati 2 warning

procedure MAIN is
begin
declare
type ARRAY_BOOLEAN is array(INTEGER range <>) of BOOLEAN;
a : FLOAT;
c : INTEGER;
b : INTEGER;

begin


a := 1.5;
c := 0;

declare
vettore : ARRAY_BOOLEAN(0..10);
begin
if Integer(a) < c then
vettore(c) := true;
c := c + 1;
else
if Integer(a) > 1 and c /= 0 then
a := a + 1.00;
b := 20;
else
if c >= 0 or b = 10 then
c := 20;
end if;
end if;
end if;

end;

end;
end MAIN;
```

3.2.2 Caso 2

Input:

```
public class Prova {

      public static void main() {

      double a = 1.5;

      int c = 0;

      int b;
```

```
      if(a < c){
            int d = 10;
            d++;}
            else if(a > 1 && c == 0){
                  a++;
                  d--;
                  b = 20;}
                  else if(c >= 0 || b == 10)
                        c = 20;
       }
}
```

Output:

```
--[C:\Users\Domenico\workspace\Progetto\prova2.java]--

Error:   E' stata usata una variabile non dichiarata (Riga 16: d--;)
Warning: operatore non compatibile verrà tentata una conversione per risolvere
il problema (Riga 11: if(a < c){)
Warning: operatore non compatibile verrà tentata una conversione per risolvere
il problema (Riga 14: else if(a > 1 && c == 0){)
Traduzione interrotta alla riga 16, sono stati rilevati:
1 errori
2 warning
```

3.2.3 Caso 3

Input:

```
public class Prova {

  public static void main() {

      double a = 1.5;

      double c = 0.2;

      double b;
      int[] vettore = new int[]{2,5,10};

      if(a < c){
      c++;
      double d = a*5.5;}
      else{
      a++;
      b += c;
      c /= vettore[2];
          }
        }

}
```

Output:

```
--[/home/domenico/Scrivania/Compilatore/prova2.java]—
```

```
Warning: operatore non compatibile verrà  tentata una conversione per risolvere
il problema (Riga 11: if(a < c){)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 17: c /= vettore[2];)
Traduzione terminata, sono stati rilevati 2 warning

procedure MAIN is
begin
declare
type ARRAY_INTEGER is array(INTEGER range <>) of INTEGER;
a : FLOAT;
c : FLOAT;
b : FLOAT;
d : FLOAT;

begin


a := 1.5;
c := 0.2;

declare
vettore : ARRAY_INTEGER(0..2);
begin
vettore := (2,5,10);
if Integer(a) < Integer(c) then
c := c + 1.00;
d := a * 5.5;
else
a := a + 1.00;
b := b + c;
c /= Float(vettore(2));
end if;

end;

end;
end MAIN;
```

3.2.4. Caso 4

Input:

```
public class Prova {

  public static void main() {

        double a = 1.5;

            int c = 0;

            double b = 20;

            int[] vet = new int[10];

            if( c + b ){
                  vet[c] = b++;
                  c++;
                  a *= vet[c];
            }
```

```
  }
}
```

Output:

```
--[/home/domenico/Scrivania/Compilatore/prova2.java]--


Error:   E' necessario inserire un espressione booleana come condizione di un if
(Riga 13: if( c + b ){)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 9: double b = 20;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 13: if( c + b ){)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 14: vet[c] = b++;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 16: a *= vet[c];)
Impossibile completare la traduzione, sono stati rilevati:
1 errori
4 warning
```

3.3 COSTRUTTI ITERATIVI

3.3.1 Caso 1

Input:

```
public class Prova {

	public static void main() {

	int a = 1.5;
	a += 3.5;

	int c = 0.5;

	int b;

	//int z = 7 + a;

	int[] vettore = new int[7];

	//vettore = new int[2];

	double[] vettoredouble = new double[]{1.5,4.8};
	double[] vettoredouble2 = new double[4];


	for(int z=0; z < 3.5; z++){
	for(int i=0;i<7;i++, b--){

		b = c;

		vettore[i] = c + vettoredouble[i];

	}
```

```
      while(b<8){
            b = 1 + a;
            a++;
            vettoredouble2[b] = 10;
      }

      vettore[3] = z + 4 + vettoredouble[1] * vettoredouble[1];

        }
       }
}
```

Output:

```
--[C:\Users\Domenico\workspace\Progetto\prova2.java]--

Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 5:   int a = 1.5;)
Warning: nella conversione è stato apportato un arrotondamento (Riga 5: int a =
1.5;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 6: a += 3.5;)
Warning: nella conversione è stato apportato un arrotondamento (Riga 6: a +=
3.5;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 8: int c = 0.5;)
Warning: nella conversione è stato apportato un arrotondamento (Riga 8: int c =
0.5;)
Warning: operatore non compatibile verrà tentata una conversione per risolvere
il problema (Riga 22: for(int z=0; z < 3.5; z++){)
Warning: nella conversione Ã¨ stato apportato un arrotondamento (Riga 22:
for(int z=0; z < 3.5; z++){)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 27: vettore[i] = c + vettoredouble[i];)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 27: vettore[i] = c + vettoredouble[i];)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 34: vettoredouble2[b] = 10;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 37: vettore[3] = z + 4 + vettoredouble[1] * vettoredouble[1];)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 37: vettore[3] = z + 4 + vettoredouble[1] * vettoredouble[1];)
Traduzione terminata, sono stati rilevati 13 warning


procedure MAIN is
begin
declare
type ARRAY_INTEGER is array(INTEGER range <>) of INTEGER;
type ARRAY_FLOAT is array(INTEGER range <>) of FLOAT;
a : INTEGER;
c : INTEGER;
b : INTEGER;
z : INTEGER;
i : INTEGER;

begin


a := 2;
```

```
a := a + 4;
c := 1;

declare
vettore : ARRAY_INTEGER(0..7);
begin
declare
vettoredouble : ARRAY_FLOAT(0..1);
begin
vettoredouble := (1.5,4.8);
declare
vettoredouble2 : ARRAY_FLOAT(0..4);
begin
z := 0;
while z < 4 loop
i := 0;
while i < 7 loop
b := c;
vettore(i) := Integer(Float(c) + vettoredouble(i));
i := i + 1;
b := b - 1;
end loop;
while b < 8 loop
b := 1 + a;
a := a + 1;
vettoredouble2(b) := 10.00;
end loop;
vettore(3) := Integer(Float(z + 4) + vettoredouble(1) * vettoredouble(1));
z := z + 1;
end loop;

end;

end;

end;

end;
end MAIN;
```

3.3.2 Caso 2

Input:

```
public class Prova {

      public static void main() {

      int a = 1.5;
      a += 3.5;

      int c = 0.5;

      int b;

      int z = 7 + a;

      int[] vettore = new int[7];

      vettore = new int[2];
```

```
      double[] vettoredouble = new double[]{1.5,4.8};
      double[] vettoredouble2 = new double[4];


      for(int z=0; z < 3.5; z++){
      for(int i=0;i<7;i++, b--){

            b = c;

            vettore[i] = c + vettoredouble[i];

      }

      while(b<8){
            b = 1 + a;
            a++;
            vettoredouble2[b] = 10;
      }

        }
       }
}
```

Output:

```
--[C:\Users\Domenico\workspace\Progetto\prova2.java]--

Error:   il linguaggio target non supporta l'allocazione a posteriori del
vettore (Riga 16: vettore = new int[2];)
Error:   variabile duplicata (Riga 22: for(int z=0; z < 3.5; z++){)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 5: int a = 1.5;)
Warning: nella conversione è stato apportato un arrotondamento (Riga 5: int a =
1.5;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 6: a += 3.5;)
Warning: nella conversione è stato apportato un arrotondamento (Riga 6: a +=
3.5;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 8: int c = 0.5;)
Warning: nella conversione è stato apportato un arrotondamento (Riga 8: int c =
0.5;)
Warning: operatore non compatibile verrà tentata una conversione (Riga 22:
for(int z=0; z < 3.5; z++){)
Warning: nella conversione è stato apportato un arrotondamento (Riga 22: for(int
z=0; z < 3.5; z++){)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 27: vettore[i] = c + vettoredouble[i];) (x2)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 34: vettoredouble2[b] = 10;)


Impossibile completare la traduzione, sono stati rilevati:
2 errori
11 warning
```

3.3.3 Caso 3

Input:

```
public class Prova {

	public static void main() {

	double a = 1.5;

	int c = 0;
	boolean flag = false;

	double b = 20;

	for(int i=0.5 ;i<7.9;i++, b--){
	b--;
	while(b > 8){
		b = 1 + a;
		a++;
		}
	if(b != 10){
	flag = true;
	c++;}
	}
}

}
```

Output:

```
--Parsing [/home/domenico/Scrivania/Compilatore/prova2.java]--

Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 9: double b = 20;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 11: for(int i=0.5)
Warning: nella conversione è stato apportato un arrotondamento (Riga 11: for(int
i=0.5)
Warning: operatore non compatibile verrà tentata una conversione (Riga 11:
for(int i=0.5 ;i<7.9;i++, b--){)
Warning: nella conversione è stato apportato un arrotondamento (Riga 11: for(int
i=0.5 ;i<7.9;i++, b--){)
Warning: operatore non compatibile verrà tentata una conversione (Riga 13:
while(b > 8){)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 14: b = 1 + a;)


Traduzione terminata, sono stati rilevati 7 warning

procedure MAIN is
begin
declare
a : FLOAT;
c : INTEGER;
flag : BOOLEAN;
b : FLOAT;
i : INTEGER;

begin
```

```
a := 1.5;
c := 0;
flag := false;
b := 20.00;
i := 1;
while i < 8 loop
b := b - 1.00;
while Integer(b) > 8 loop
b := 1.00 + a;
a := a + 1.00;
end loop;
if b /= 10.00 then
flag := true;
c := c + 1;
end if;
i := i + 1;
b := b - 1.00;
end loop;

end;
end MAIN;
```

3.3.4 Caso 4

Input:

```
public class Prova {

      public static void main() {

      double a = 1.5;

      int c = 0;

      double b = 20;

      int[] vet = new int[10];


      for(int i=0;i<7;i++, b--){
      b--;
      double[] vettore = new double[7];
      while(b > 8){
            int d = c + 20;
            b = 1 + a;
            a++;
            vet[12] = 5;
            int[] vettore = new int[2.5];
            vettore[0] = b;
            }
      d++;
      vettore[i] = 2.5 + i;
      c++;}
       }

}
```

Output:

```
--[C:\Users\Domenico\workspace\Progetto\prova2.java]--

Error:   l'indice dell'array supera la dimensione del vettore (Riga 21: vet[12]
= 5;)
Error:   la dimensione di un array deve essere un numero intero (Riga 22: int[]
vettore = new int[2.5];)
Error:   variabile duplicata (Riga 22: int[] vettore = new int[2.5];)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 9: double b = 20;)
Warning: operatore non compatibile verrà tentata una conversione per risolvere
il problema (Riga 17: while(b > 8){)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 19: b = 1 + a;)
Traduzione interrotta alla riga 22, sono stati rilevati:
3 errori
3 warning
```

3.3.5 Caso 5

Input:

```
public class Prova {

      public static void main() {

      double a = 1.5;

      int c = 0;
      double i = 2.5;
      int b = 20;


      for(; i<7; i++, b--){
      b--;
      double[] vettore = new double[7];
      vettore[i] = i++;
      for(;b < 8;b++){
            int d = c + 20;
            b = 1 + a;
            a++;
            double[] vettore = new double[15];
            vettore[0] = b;
            d++;
            }
      c++;}
       }
}
```

Output:

```
--[C:\Users\Domenico\workspace\Progetto\prova2.java]--

Error:   l'indice di un array deve essere un numero intero (Riga 14: vettore[i]
= i++;)
Error:   variabile duplicata (Riga 19: double[] vettore = new double[15];)
```

```
Warning: operatore non compatibile verrà tentata una conversione per risolvere
il problema (Riga 11: for(; i<7; i++, b--){)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 17: b = 1 + a;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 17: b = 1 + a;)
Warning: tipi non compatibili, verrà tentata una conversione per risolvere il
problema (Riga 20: vettore[0] = b;)
Impossibile completare la traduzione, sono stati rilevati:
2 errori
4 warning
```

BIBLIOGRAFIA

GIACOMO PISCITELLI, *Dispense del corso di "Linguaggi formali e compilatori"*, A.A. 2010-11

Copyright ©1998-2009 by Gerwin Klein, ***JFlex User's Manual*** *Version 1.4.3, January 31,2009* http://jflex.de/manual.html

Frank Flannery, C. Scott Ananian, Dan Wang, Andrew W. Appel, MichaelPetter, ***CUP User's Manual,*** March 2006(v0.11a) http://www2.cs.tum.edu/projects/cup/manual.html

ISO/IEC 8652:2007(E) Ed. 3 **Ada Reference Manual,** Language and Standard Libraries