Un Compilatore in Java per Tiger

Fausto Spoto

Dipartimento di Informatica
Università di Verona
fausto.spoto@univr.it

17 febbraio 2003

Indice

Introduzione

Ho pensato questa dispensa come un complemento al testo *Modern Compiler Implementation in Java* di Andrew W. Appel. Se infatti tale testo descrive la costruzione di un compilatore in Java per il linguaggio Tiger, è anche vero che tale descrizione è spesso lacunosa e inadatta a un corso di estensione limitata (circa 40 ore). Inoltre è necessario differenziare annualmente il progetto finale del corso di *Compilatori*. Questa dispensa intende quindi descrivere il nucleo centrale funzionante del compilatore Tiger. Il progetto annuale integrerà questa dispensa in nuove direzioni.

La scelta della lingua italiana mi sembra scontata per una dispensa indirizzata a un corso di laurea in Italiano. I nomi degli identificatori nei sorgenti Java sono invece in Inglese, in modo da non interferire con la parte di sorgente già sviluppata da Appel.

Intendo ringraziare tutti gli studenti del corso di *Compilatori* effettuato presso il Dipartimento di Informatica di Verona nell'anno accademico 2001/2002. Parte delle soluzioni adottate in questa dispensa sono state individuate da tali studenti, e i loro commenti e critiche sono sempre stati costruttivi.

Questa dispensa contiene sicuramente errori e imprecisioni. Sarò grato a tutti gli studenti che vorranno segnalarmeli.

Fausto Spoto, Verona, gennaio 2003

Capitolo 1

Analisi Lessicale

1.1 L'analizzatore lessicale

Un analizzatore lessicale è una classe i cui oggetti possono leggere un file sorgente e restituire un token alla volta. Alcuni token hanno un valore lessicale associato, per esempio gli identificatori, le stringhe e i numeri interi. Decidiamo quindi che un analizzatore lessicale sia un oggetto la cui classe implementa la seguente interfaccia:

`Parse/Lexer.java`

```
package Parse;

interface Lexer {
    public java_cup.runtime.Symbol nextToken()
      throws java.io.IOException;
}
```

Come si vede, chiediamo semplicemente che tale analizzatore implementi un metodo `nextToken`, il quale può generare un errore di input/output (se c'è un errore nella lettura del file sorgente) o altrimenti ritornare un oggetto di tipo `java_cup.runtime.Symbol`. Tale classe è usata dall'analizzatore sintattico JavaCup (Capitolo 2) per rappresentare i token. È quindi una buona idea farla ritornare dall'analizzatore lessicale, in modo da interfacciarlo in futuro con l'analizzatore sintattico. Vediamo in dettaglio come è fatto un token:

`java_cup/runtime/Symbol.java`

```
package java_cup.runtime;

public class Symbol {
```

Il numero del token rappresentato

```
  public int sym;
```

Le posizioni di inizio e fine del token nel file sorgente

```
  public int left, right;
```

Un eventuale valore lessicale

```
  public Object value;
```

Un costruttore

```
  public Symbol(int id, int l, int r, Object o) {
    sym = id;
    left = l;
    right = r;
    value = o;
  }

  ...
}
```

Potrebbe sembrare che il numero da associare a ogni token (ovvero il valore del campo `sym` della classe `java_cup/runtime/Symbol.java`) possa essere scelto liberamente: basta che token distinti abbiano associati numeri diversi. Potremmo cioè definire tutti gli identificatori di token tramite una classe del tipo

`Parse/sym.java`

```
package Parse;

public class sym {
  static final int FUNCTION = 42;
  static final int EOF = 0;
  static final int INT = 4;
  static final int GT = 23;
  static final int DIVIDE = 18;
  ...
}
```

Questo è vero se ci limitiamo a una analisi lessicale. Se però vogliamo che il nostro analizzatore lessicale si interfacci con un analizzatore sintattico (JavaCup) occorre che i due analizzatori abbiano la stessa visione del mondo, usino cioè le stesse costanti per identificare i token. La cosa più semplice è di far generare la precedente classe `Parse/sym.java` a JavaCup e utilizzarla anche per l'analizzatore lessicale. Per adesso, quindi, supponiamo di avere scritto noi la classe `Parse/sym.java`, ma si tenga in mente che il realtà tale classe è stata generata da JavaCup.

1.2 Il gestore degli errori

Supponiamo che durante l'analisi lessicale (o sintattica, o semantica, o la generazione del codice…) si verifichi un errore da comunicare all'utente. Sarebbe bello indicare *dove*, nel testo del programma, si trova tale errore. Occorre cioè specificare il token che permette di trovare l'errore. Abbiamo visto che ogni token ha associata la sua posizione nel file sorgente in cui si trova. Questa informazione è molto precisa ma certo poco utile per l'utente che, del token su cui si segnala l'errore, vorrebbe invece conoscere il numero di linea in cui si trova e il numero di caratteri dall'inizio della linea. Decidiamo quindi di memorizzare in una lista a quanti caratteri dall'inizio del file sorgente si trovano i simboli di newline. Usiamo una lista del tipo

`ErrorMsg/LineList.java`

```
package ErrorMsg;

class LineList {
  int head;
  LineList tail;
  LineList(int h, LineList t) {head=h; tail=t;}
}
```

Ogni volta che durante l'analisi lessicale incontriamo un carattere di newline, ne memorizziamo la posizione in testa alla lista (si veda il file `Parse/Tiger.lex`). Questa informazione ci permetterà di ricostruire le coordinate di un token a partire dal numero di caratteri a cui si trova dall'inizio del file sorgente. Questo algoritmo è implementato dalla seguente classe.

`ErrorMsg/ErrorMsg.java`

```
package ErrorMsg;

public class ErrorMsg {
```

Questa è la lista delle posizioni dei newline

```
  private LineList linePos = new LineList(-1,null);
```

Questo contatore indica quante linee compongono il programma sorgente

```
  private int lineNum=1;
```

Il nome del file sorgente

```
  private String filename;
```

Questo flag indica se si sono verificati errori

```
  public boolean anyErrors;
```

Il costruttore

```
  public ErrorMsg(String f) { filename=f; anyErrors = false; }
```

Se si incontra un nuovo carattere di newline, si incrementa il numero di linee del programma sorgente e si aggiunge un elemento in testa alla lista

```
  public void newline(int pos) {
    lineNum++;
    linePos = new LineList(pos,linePos);
  }
```

Questo metodo permette di comunicare un messaggio di errore `msg` che si è verificato al carattere `pos` dall'inizio del file sorgente

```
  public void error(int pos, String msg) {
    int n = lineNum;
    LineList p = linePos;
    String sayPos = "nowhere";
```

Si è verificato un errore!

```
    anyErrors=true;
```

Scorriamo la lista alla ricerca del primo newline che precede la posizione `pos`. Decrementiamo `n` in modo che alla fine esso contenga la linea in cui si trova il carattere `pos`-esimo

```
    for (; p != null && p.head > pos; p = p.tail, n--);
```

`n` è adesso il numero di linea in cui si è verificato l'errore, e `p.head` è il numero di caratteri, dall'inizio del file sorgente, in cui si trova il carattere di newline immediatamente precedente `pos`. La sottrazione ci permette quindi di ricostruire il numero di caratteri dall'inizio della riga in cui si è verificato l'errore

```
    if (p != null) sayPos = n + "." + (pos - p.head);

    System.out.println(filename + "::" + sayPos + ": " + msg);
  }
}
```

1.3 Il generatore di analizzatori lessicali JLex

Il nostro analizzatore lessicale verrà creato dal programma JLex, che è scritto in Java e può quindi essere invocato con una linea di comando del tipo

```
> java JLex.Main Parse/Tiger.lex
```

dove `Parse/Tiger.lex` contiene la descrizione dei token del linguaggio Tiger. Il risultato è un file sorgente Java di nome `Parse/Tiger.lex.java` che contiene una classe, di nome `Yylex`, che implementa l'interfaccia `Parse/Lexer.java`, ovvero un automa capace di riconoscere i token Tiger. Vediamo adesso come possiamo descrivere tali token.

`Parse/Tiger.lex`

Quello che precede il %% finirà testualmente all'inizio del file `Parse/Tiger.lex.java`. Qui stiamo dicendo che tale file appartiene al package `Parse` e può usare la classe definita in `ErrorMsg/ErrorMsg.java`

```
package Parse;
import ErrorMsg.ErrorMsg;
%%
```

Questi *non* sono commenti! Il carattere # identifica delle *direttive* per JLex. Stiamo dicendo che la classe che implementa l'automa regolare dovrà implementare l'interfaccia in `Parse/Lexer.java`. Non è necessario indicare `Parse` grazie alla precedente dichiarazione di `import`. Abbiamo visto che tale interfaccia richiede l'implementazione di un methodo di nome `nextToken` che ritorna un oggetto di tipo `java_cup/runtime/Symbol.java`. Indichiamo anche queste informazione in modo che JLex scriva la funzione che restituisce il prossimo token in maniera compatibile con l'interfaccia `Parse/Lexer.java`

```
%implements Lexer
%function nextToken
%type java_cup.runtime.Symbol
```

Abbiamo bisogno del contatore `yychar`, il quale dice in ogni momento a quale carattere inizia l'ultimo token letto

```
%char
```

Tra %{ e %} possiamo inserire del codice che verrà inserito testualmente all'interno della classe `Yylex`. Decidiamo di inserire dei metodi privati che ci permettono di tenere il conto di quante linee di sorgente sono state viste e di generare dei messaggi di errore, eventualmente specificando la posizione nel sorgente in cui tale errore deve essere riportato. Il metodo `tok` ci permette di generare un `java_cup/runtime/Symbol.java` a partire dal tipo di token riconosciuto e dall'eventuale valore lessicale associato. Basta specificare il carattere di inizio e fine del token

```
%{
private ErrorMsg errorMsg;

private void newline() { errorMsg.newline(yychar); }

private void err(int pos, String s) {
  errorMsg.error(pos,s);
}

private void err(String s) {
  err(yychar,s);
}

private java_cup.runtime.Symbol tok(int kind, Object value) {
  return new java_cup.runtime.Symbol(kind, yychar,
                                     yychar+yylength(), value);
}
```

Il costruttore della classe che contiene l'automa. JLex genera automaticamente un costruttore che richiede solo un `java.io.InputStream` (lo stream da cui leggere). Noi abbiamo però anche una classe `ErrorMsg/ErrorMsg.java` nel nostro automa. Aggiungiamo quindi un costruttore che inizializza entrambi i campi

```
Yylex(java.io.InputStream s, ErrorMsg e) {
  this(s);
  errorMsg=e;
}
```

Queste variabili ci serviranno durante l'analisi lessicale. `commentCount` ci dice la profondità di commento in cui siamo, intesa come il numero di commenti ancora aperti che abbiamo incontrato. `nyNum` and `myString` ci serviranno per calcolare il valore lessicale dei token INT e STRING, rispettivamente

```
int commentCount=0;
int myNum;
String myString="";
%}
```

Tra `%eofval{` ed `%eofval}` possiamo specificare i comandi che devono venire eseguiti quando si incontra la fine del file. Diciamo di inserire uno pseudo-token EOF. Ci preoccupiamo però anche di controllare che tutti i commenti aperti siano stati chiusi, altrimenti diamo errore

```
%eofval{
{
  if (commentCount != 0) err("Unclosed comment");
  else return tok(sym.EOF, null);
}
%eofval}
```

L'automa si trova normalmente in una modalità di default `YYINITIAL`. Definiamo altre due modalità (impropriamente chiamate "stati")

```
%state STRING
%state COMMENT
```

Le espressioni regolari che descrivono i token. Prefissiamo ogni espressione con la modalità in cui è attiva. Se non si usasse nessuna modalità, l'espressione regolare sarebbe valida in *tutte* le modalità definite

```
%%
```

Se leggiamo un carattere di doppio apice entriamo nella modalità `STRING`. Il valore lessicale della stringa viene inizializzato

```
<YYINITIAL>"\""         {myString=""; yybegin(STRING);}
```

Questi caratteri vengono semplicemente scartati

```
<YYINITIAL>[ \t\f]      {}
```

Uno slash seguito da un asterisco indica l'inizio di un commento, che deve essere scartato. Dal momento che i commenti possono essere annidati, non basta arrivare fino alla prima chiusura (asterisco + slash), ma occorre contare quanti commenti sono stati aperti e aspettare che ne vengano chiusi altrettanti. Questo è gestito dalla modalità `COMMENT`

```
<YYINITIAL>"/*"         {commentCount++; yybegin(COMMENT);}
```

Se incontriamo un carattere newline chiamiamo il metodo privato che incrementa il numero di linee lette

```
<YYINITIAL>\n           {newline();}
```

Per ogni parola chiave di Tiger scriviamo una espressione regolare che la riconosce. L'identificatore del token è quello che troviamo nella classe `Parse/sym.java`. Il valore lessicale è `null`

```
<YYINITIAL>"while"      {return tok(sym.WHILE, null);}
<YYINITIAL>"for"        {return tok(sym.FOR, null);}
<YYINITIAL>"to"         {return tok(sym.TO, null);}
<YYINITIAL>"break"      {return tok(sym.BREAK, null);}
<YYINITIAL>"let"        {return tok(sym.LET, null);}
<YYINITIAL>"in"         {return tok(sym.IN, null);}
<YYINITIAL>"end"        {return tok(sym.END, null);}
<YYINITIAL>"function"   {return tok(sym.FUNCTION, null);}
<YYINITIAL>"var"        {return tok(sym.VAR, null);}
<YYINITIAL>"type"       {return tok(sym.TYPE, null);}
<YYINITIAL>"array"      {return tok(sym.ARRAY, null);}
<YYINITIAL>"if"         {return tok(sym.IF, null);}
<YYINITIAL>"then"       {return tok(sym.THEN, null);}
<YYINITIAL>"else"       {return tok(sym.ELSE, null);}
<YYINITIAL>"do"         {return tok(sym.DO, null);}
<YYINITIAL>"of"         {return tok(sym.OF, null);}
<YYINITIAL>"nil"        {return tok(sym.NIL, null);}
```

Questi caratteri vengono trasformati in token senza valore lessicale

```
<YYINITIAL>","          {return tok(sym.COMMA, null);}
<YYINITIAL>":"          {return tok(sym.COLON, null);}
<YYINITIAL>";"          {return tok(sym.SEMICOLON, null);}
<YYINITIAL>"("          {return tok(sym.LPAREN, null);}
<YYINITIAL>")"          {return tok(sym.RPAREN, null);}
<YYINITIAL>"["          {return tok(sym.LBRACK, null);}
<YYINITIAL>"]"          {return tok(sym.RBRACK, null);}
<YYINITIAL>"{"          {return tok(sym.LBRACE, null);}
<YYINITIAL>"}"          {return tok(sym.RBRACE, null);}
<YYINITIAL>"."          {return tok(sym.DOT, null);}
<YYINITIAL>"+"          {return tok(sym.PLUS, null);}
<YYINITIAL>"-"          {return tok(sym.MINUS, null);}
<YYINITIAL>"*"          {return tok(sym.TIMES, null);}
<YYINITIAL>"/"          {return tok(sym.DIVIDE, null);}
<YYINITIAL>"="          {return tok(sym.EQ, null);}
<YYINITIAL>"<>"         {return tok(sym.NEQ, null);}
<YYINITIAL>"<"          {return tok(sym.LT, null);}
<YYINITIAL>"<="         {return tok(sym.LE, null);}
<YYINITIAL>">="         {return tok(sym.GE, null);}
<YYINITIAL>">"          {return tok(sym.GT, null);}
<YYINITIAL>"&"          {return tok(sym.AND, null);}
<YYINITIAL>"|"          {return tok(sym.OR, null);}
<YYINITIAL>":="         {return tok(sym.ASSIGN, null);}
```

Un carattere seguito eventualmente da caratteri o numeri o underscore è un *identificatore*. Questa regola deve essere messa dopo quelle per le parole chiave del linguaggio, altrimenti esse verrebbero considerate identificatori. Si noti che il valore lessicale è il testo letto dall'espressione regolare, ottenibile tramite l'espressione `yytext()`

```
<YYINITIAL>[a-zA-Z][a-zA-Z0-9_]*   {return tok(sym.ID, yytext());}
```

Una o più cifre formano un numero intero. Il testo dell'espressione viene trasformato in un oggetto della classe `Integer`

```
<YYINITIAL>[0-9]+       {return tok(sym.INT, new Integer(yytext()));}
```

Non ci sono altre espressioni regolari. Se del testo non è leggibile tramite una delle espressioni regolari precedenti, diamo un messaggio di errore

```
<YYINITIAL>.            {err("Unmatched Character");}
```

La modalità `STRING`. Dobbiamo gestire tutte le sequenze di escape di Tiger. Tali sequenze iniziano con un backslash. Per esempio, un backslash seguito dal carattere `n` inserisce un newline nella stringa. Dal momento che backslash è usato per le sequenze di escape di JLex, per specificare il carattere di backslash nell'espressione regolare occorre inserirlo come una sequenza di escape per JLex! Ovvero come un doppio backslash. Si noti che nell'azione Java a destra usiamo sequenze di escape Java!

```
<STRING>\\n             {myString+="\n";}
<STRING>\\t             {myString+="\t";}
```

Un backslash seguito da tre cifre inserisce il carattere ascii corrispondente. Con `yytext().charAt(i)` otteniamo il numero ascii dell'i-esimo carattere. Sottraendo 48 otteniamo la cifra i-esima vista come un intero. Una volta calcolato il valore intero del carattere, controlliamo che non sia troppo grande e poi lo convertiamo nel corrispondente carattere tramite un cast

```
<STRING>\\[0-9][0-9][0-9]
                        {myNum=(yytext().charAt(1)-48)*100+
                           (yytext().charAt(2)-48)*10+
                           (yytext().charAt(3)-48);
                         if (myNum>255) err("Overflow in ASCII Code");
                         else myString+=(char)myNum;}
```

Due backslash inseriscono un backslash nella stringa

```
<STRING>\\\\            {myString+="\\";}
```

Un backslash seguito da alcuni caratteri di riempimento e da un altro backslash vengono semplicemente scartati. Questo permette di scrivere stringhe lunghe e di andare a capo

```
<STRING>\\[ \t\f\n]+\\  {}
```

Un backslash seguito da un doppio apice inserisce il doppio apice. Il doppio apice da solo fa tornare nella modalità `YYINITIAL` e restituisce un token con la stringa letta come valore lessicale

```
<STRING>"\\""           {myString+="\"";}
<STRING>"\""            {yybegin(YYINITIAL);
                         return tok(sym.STRING, myString);}
```

Qualsiasi altro carattere viene semplicemente accumulato nel valore lessicale della stringa

```
<STRING>.               {myString+=yytext();}
```

La modalità `COMMENT`. La chiusura di un commento provoca il decremento del contatore `commentCount`, mentre l'apertura di un commento provoca il suo incremento. Quando il contatore torna a zero si ritorna nella modalità `YYINITIAL`. Tutti i caratteri del commento vengono scartati (il commento non genera nessun token)

```
<COMMENT>"*/"           {commentCount--;
                         if (commentCount==0) yybegin(YYINITIAL);}
<COMMENT>"/*"           {commentCount++;}
<COMMENT>\n|.           {}
```

1.4 Usiamo l'analizzatore lessicale

Il file generato da JLex si chiama `Parse/Tiger.lex.java` ma contiene una classe chiamata `Yylex`. Al fine di permetterne la compilazione, dobbiamo dare al file lo stesso nome della classe, ovvero

```
> mv Parse/Tiger.lex.java Parse/Yylex.java
```

Al fine di verificare la funzionalità del nostro analizzatore lessicale, utilizziamo un file `Parse/Main.java` che ne crea un'istanza e la applica a un file sorgente Tiger, stampando i token letti.

`Parse/Main.java`

```
package Parse;

public class Main {
```

Il metodo `main` legge il file il cui nome è passato come argomento, e lo usa per creare una struttura `ErrorMsg/ErrorMsg.java` e uno stream di input

```
  public static void main(String argv[]) throws java.io.IOException {
      String filename = argv[0];
      ErrorMsg.ErrorMsg errorMsg = new ErrorMsg.ErrorMsg(filename);
      java.io.InputStream inp=new java.io.FileInputStream(filename);
```

Creiamo un'istanza dell'analizzatore lessicale

```
      Lexer lexer = new Yylex(inp,errorMsg);
```

Leggiamo un token alla volta e ne stampiamo il nome, il valore lessicale (se esiste) e i caratteri a cui inizia e termina il token

```
      java_cup.runtime.Symbol tok;
      do {
         tok=lexer.nextToken();
         System.out.print(symnames[tok.sym]);
         if (tok.value!=null) System.out.print("(" + tok.value + ")");
         System.out.println(" from " + tok.left + " to " + tok.right);
      } while (tok.sym != sym.EOF);

      inp.close();
  }
```

Diamo un nome stampabile a ogni token

```
  static String symnames[] = new String[100];

  static {
     symnames[sym.FUNCTION] = "FUNCTION";
     symnames[sym.EOF] = "EOF";
     symnames[sym.INT] = "INT";
     ...
   }
}
```

Scriviamo tutto dentro a un makefile, in modo da permettere di compilare il nostro analizzatore lessicale in maniera semplice.

`makefile`

```
JFLAGS=-O
```

L'analizzatore lessicale va ricompilato ogni volta che si modifica `Parse/Tiger.lex`

```
Parse/Yylex.java: Parse/Tiger.lex
```

Il simbolo @ prima di un comando non lo manda in stampa quando viene eseguito. L'eventuale output del comando viene però stampato!

```
        @echo "*** Compiling the lexical analyzer ***"
        java JLex.Main Parse/Tiger.lex
        mv Parse/Tiger.lex.java Parse/Yylex.java
```

Se si richiede la compilazione della shell di test per l'analizzatore lessicale (`make lexical`) verrà eseguita questa regola. La compilazione della shell avverrà se è stato aggiornato l'analizzatore lessicale oppure se il testo della shell è stato modificato o infine se la shell stessa `Parse/Main.class` ha subito modifiche. Per adesso questa è l'unica regola che può modificare la shell stessa, per cui questa condizione non serve. In futuro, però, aggiungeremo altre shell (`make syntactical`, *ecc.*). Dovremo ricompilare la shell lessicale se l'ultima compilazione della shell è più aggiornata del file di annotazione `lexical`, che viene aggiornato ("toccato") alla fine di questa compilazione

```
lexical: Parse/Yylex.java Parse/Main_lexical.java Parse/Main.class
        @echo "*** Compiling the lexical shell ***"
```

Al fine di permettere diverse versioni della stessa classe `Parse/Main.java`, le scriviamo su files distinti. Quello opportuno verrà copiato al posto giusto prima della compilazione

```
        cp Parse/Main_lexical.java Parse/Main.java
        ./javac ${JFLAGS} Parse/Main.java
        touch lexical
```

Con il comando `make clean` cancelliamo tutti i files creati dalle precedenti compilazioni. Indicando `clean` come un *phony target* facciamo in modo che i comandi vengano eseguiti anche se esistesse un file chiamato `clean`. Premettendo i comandi con il carattere `-` chiediamo di non interrompere l'esecuzione sequenziale dei comandi al primo di essi che dà errore. Questo può avvenire se uno dei files di cui si chiede la cancellazione non esiste (o l'asterisco non ha espansioni possibili)

```
.PHONY: clean
clean:
        -rm lexical
        -rm syntactical
        -rm Parse/*.class ErrorMsg/*.class Parse/Yylex.java
            Parse/Grm.java
        touch Parse/Main.class
```

Per compilare e provare il nostro analizzatore lessicale basta dare i seguenti comandi:

```
> make lexical

> java Parse.Main testcases/test1.tig
```

Capitolo 2

Analisi Sintattica

2.1 L'analizzatore sintattico

Un analizzatore sintattico è implementato da una classe descritta nel file `Parse/Grm.java`. La creazione di un'istanza di questa classe richiede di specificare l'analizzatore lessicale da usare per dividere il file sorgente in token, e la struttura di errore da utilizzare per comunicare degli errori all'utente: `new Grm(lexer, errorMsg)`. Una volta creato, possiamo usarlo richiamando il suo metodo `parse()`. Cosa ritornerà tale metodo? La cosa più sensata è di fargli ritornare una descrizione del simbolo a cui il testo del programma si riduce, ovvero un oggetto della classe `java_cup/runtime/Symbol.java` (Sezione 1.1). Si noti che tale classe contiene un campo per l'eventuale *valore lessicale* associato al simbolo. Abbiamo già visto che l'analizzatore lessicale sintetizza il valore lessicale di interi, stringhe e identificatori, ovvero di alcuni terminali della grammatica che stiamo per scrivere per Tiger. Dovrà essere invece l'analizzatore sintattico a preoccuparsi di sintetizzare il valore lessicale dei non terminali della stessa grammatica, usando le classi di sintassi astratta a pag. 103 dell'Appel.

2.2 Il generatore di analizzatori sintattici JavaCup

Non saremo noi a scrivere direttamente l'analizzatore sintattico LALR(1) `Parse/Grm.java`. Lo faremo invece generare automaticamente a un programma Java chiamato (Java)Cup. È possibile eseguirlo con un comando tipo

```
> java java_cup.Main -parser Grm -expect 3 -dump_grammar -dump_states
<Parse/Grm.cup 2>Parse/Grm.err
```

In tale comando specifichiamo che il file Java che conterrà il sorgente dell'analizzatore sintattico dovrà essere chiamato `Grm.java`, che la compilazione può accettare fino a tre conflitti nella tabella LALR(1) (risolti secondo regole di default. Se fossero più di tre l'esecuzione si interromperà con un errore) e che vogliamo mandare sullo standard error (dirottato tramite `>2`) una descrizione della grammatica e degli stati LALR(1) generati per la stessa. Queste informazioni possono servire a fini di debugging, ed è comunque raccomandato che lo studente dia loro un'occhiata. La grammatica da cui partire per generare l'analizzatore viene fornita come standard input.

Vediamo adesso come descriviamo la nostra grammatica Tiger e la sintesi dei valori lessicali per i non terminali della nostra grammatica.

`Parse/Grm.cup`

Iniziamo chiedendo che l'analizzatore sintattico faccia parte del package Java `Parse`

```
package Parse;
```

Questo metodo può essere richiamato dalle azioni semantiche. Il suo scopo è di semplificarci la vita quando vogliamo trasformare il nome di un identificatore in un simbolo che lo rappresenta. Per la classe `Symbol/Symbol.java` si veda anche il testo, capitolo 5

```
action code {:
static Symbol.Symbol sym(String s) {
  return Symbol.Symbol.symbol(s);
}
:};
```

Questo codice viene inserito all'interno della classe `Parse/Grm.java` che viene creata da JavaCup

```
parser code {:
```

Dichiariamo due campi per il nostro analizzatore sintattico. Il primo contiene l'analizzatore lessicale che possiamo usare per leggere il sorgente Tiger, il secondo l'oggetto tramite il quale dare dei messaggi di errore

```
private Lexer lexer;
private ErrorMsg.ErrorMsg errorMsg;
```

Questi metodi ci permettono di comunicare un errore verificatosi mentre si cercava di ridurre il non terminale `current`. Si ricordi che un `java_cup/runtime/Symbol.java` ha un campo `info` che dà la sua posizione dall'inizio del file sorgente Tiger

```
public void syntax_error(java_cup.runtime.Symbol current) {
  report_error("Syntax error", current);
}

public void report_error(String message, java_cup.runtime.Symbol info) {
  errorMsg.error(info.left, message);
}
```

Un costruttore. Arricchiamo il costruttore di default in modo da inizializzare i due campi che abbiamo aggiunto

```
public Grm(Lexer l, ErrorMsg.ErrorMsg err) {
  this();
  errorMsg=err;
  lexer=l;
}
:};
```

Qui specifichiamo come leggere il prossimo token del file sorgente Tiger. Ovvero, chiamando l'analizzatore lessicale

```
scan with {:
return lexer.nextToken();
:};
```

Enumeriamo i terminali. Essi sono tutti quelli che possono essere ritornati dall'analizzatore lessicale, ma anche uno pseudo-terminale `UMINUS` usato per dare massima priorità al meno unario. Per ogni terminale (e, subito dopo, non terminale) possiamo specificare il tipo del valore lessicale che sintetizzeremo per esso

```
terminal String ID, STRING;
terminal Integer INT;
terminal COMMA, COLON, SEMICOLON, LPAREN, RPAREN,
         LBRACK, RBRACK, LBRACE, RBRACE, DOT, PLUS, MINUS,
         TIMES, DIVIDE, EQ, NEQ, LT, LE, GT, GE, AND, OR,
         ASSIGN, ARRAY, IF, THEN, ELSE, WHILE, FOR, TO, DO,
         LET, IN, END, OF, BREAK, NIL, FUNCTION, VAR, TYPE, UMINUS;
```

Enumeriamo i non terminali con il tipo del loro valore lessicale associato

```
non terminal Absyn.Exp exp;
non terminal Absyn.Var lvalue;
non terminal Absyn.ExpList expseq;
non terminal Absyn.ExpList parseq;
non terminal Absyn.FieldExpList fieldinitialseq;
non terminal Absyn.ExpList expseqaux;
non terminal Absyn.Dec dec;
non terminal Absyn.Dec tydec;
non terminal Absyn.Dec vardec;
non terminal Absyn.Dec fundec;
non terminal Absyn.DecList decs;
non terminal Absyn.Ty ty;
non terminal Absyn.FieldList tyfields;
non terminal Absyn.DecList decsaux;
non terminal Absyn.FieldList tyfieldsaux;
```

Specifichiamo le priorità di alcuni terminali. Prima quelli a priorità più bassa. Le due linee seguenti servono a specificare come comportarsi di fronte all'ambiguità: `if...then if...then...else...`. Nel momento in cui leggiamo l'`else`, dobbiamo spostare l'`else` (quindi associandolo con l'ultimo `if`) o ridurre il primo `then` (quindi associando l'`else` al primo `if`)? La scelta ormai consueta in tutti i linguaggi di programmazione è la prima, che in effetti è quella che JavaCup farebbe dal momento che in un conflitto shift/reduce sceglie sempre shift. Con queste direttive di precedenza evitiamo semplicemente che JavaCup segnali il conflitto, poiché specifichiamo che la regola che termina con `ELSE` ha priorità rispetto a quella che termina con `THEN`

```
precedence nonassoc THEN;
precedence nonassoc ELSE;
```

Enumeriamo le altre priorità. Si noti che non accettiamo l'associatività di `ASSIGN`, `EQ` e simili. Quindi non è possibile scrivere `a:=b:=c` né `a=b=c`

```
precedence nonassoc ASSIGN;
precedence left AND, OR;
precedence nonassoc EQ, NEQ, LT, LE, GT, GE;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence left UMINUS;
```

Il non terminale a cui tutto il file sorgente Tiger deve ridursi è `exp`

```
start with exp;
```

Un leftvalue è qualcosa a cui si può assegnare un valore. Si noti come il nome degli identificatori viene trasformato in un `Symbol/Symbol.java`

```
lvalue ::=
     ID:i
     {: RESULT=new Absyn.SimpleVar(ileft,sym(i)); :}
   | lvalue:v DOT:d ID:f
     {: RESULT=new Absyn.FieldVar(dleft,v,sym(f)); :}
```

Questa regola sembra ridondante in quanto è un caso particolare di quella che segue. Si veda la discussione nella Sezione 2.3

```
   | ID:i LBRACK:b exp:e RBRACK
     {: RESULT=new Absyn.SubscriptVar
        (bleft,new Absyn.SimpleVar(ileft,sym(i)),e); :}
   | lvalue:v LBRACK:b exp:e RBRACK
     {: RESULT=new Absyn.SubscriptVar(bleft,v,e); :} ;
```

Si noti che la lista seguente può essere vuota

```
expseq ::=
     {: RESULT=null; :}
   | expseqaux:l
     {: RESULT=l; :} ;
```

Quella che segue, invece, non è mai vuota

```
expseqaux ::=
     exp:h
     {: RESULT=new Absyn.ExpList(h,null); :}
   | exp:h SEMICOLON expseqaux:t
     {: RESULT=new Absyn.ExpList(h,t); :} ;

parseq ::= exp:h
     {: RESULT=new Absyn.ExpList(h,null); :}
   | exp:h COMMA parseq:t
     {: RESULT=new Absyn.ExpList(h,t); :} ;

fieldinitialseq ::= ID:i EQ:q exp:e
     {: RESULT=new Absyn.FieldExpList(qleft,sym(i),e,null); :}
   | ID:i EQ:q exp:e COMMA fieldinitialseq:t
     {: RESULT=new Absyn.FieldExpList(qleft,sym(i),e,t); :} ;

exp ::=
     lvalue:v
     {: RESULT=new Absyn.VarExp(vleft,v); :}
   | NIL:n
     {: RESULT=new Absyn.NilExp(nleft); :}
```

La regola seguente gestisce anche i casi di espressioni del tipo `(exp)` e `()`

```
   | LPAREN expseq:l RPAREN
     {: RESULT=new Absyn.SeqExp(lleft,l); :}
   | INT:i
     {: RESULT=new Absyn.IntExp(ileft,i.intValue()); :}
   | STRING:s
     {: RESULT=new Absyn.StringExp(sleft,s); :}
   | MINUS:m exp:e
     {: RESULT=new Absyn.OpExp(mleft,new Absyn.IntExp(0,0),
                               Absyn.OpExp.MINUS,e); :}  %prec UMINUS
```

Una chiamata di funzione può non avere argomenti. Dare qui due regole è un'alternativa a dare un'unica regola e definire `parseq` in modo da generare anche la lista vuota

```
   | ID:i LPAREN RPAREN
     {: RESULT=new Absyn.CallExp(ileft,sym(i),null); :}
   | ID:i LPAREN parseq:p RPAREN
     {: RESULT=new Absyn.CallExp(ileft,sym(i),p); :}
   | exp:e1 PLUS:p exp:e2
     {: RESULT=new Absyn.OpExp(pleft,e1,Absyn.OpExp.PLUS,e2); :}
   | exp:e1 MINUS:m exp:e2
     {: RESULT=new Absyn.OpExp(mleft,e1,Absyn.OpExp.MINUS,e2); :}
   | exp:e1 TIMES:t exp:e2
     {: RESULT=new Absyn.OpExp(tleft,e1,Absyn.OpExp.MUL,e2); :}
   | exp:e1 DIVIDE:d exp:e2
```

```
      {: RESULT=new Absyn.OpExp(dleft,e1,Absyn.OpExp.DIV,e2); :}
    | exp:e1 EQ:e exp:e2
      {: RESULT=new Absyn.OpExp(eleft,e1,Absyn.OpExp.EQ,e2); :}
    | exp:e1 NEQ:n exp:e2
      {: RESULT=new Absyn.OpExp(nleft,e1,Absyn.OpExp.NE,e2); :}
    | exp:e1 LT:l exp:e2
      {: RESULT=new Absyn.OpExp(lleft,e1,Absyn.OpExp.LT,e2); :}
    | exp:e1 LE:l exp:e2
      {: RESULT=new Absyn.OpExp(lleft,e1,Absyn.OpExp.LE,e2); :}
    | exp:e1 GT:g exp:e2
      {: RESULT=new Absyn.OpExp(gleft,e1,Absyn.OpExp.GT,e2); :}
    | exp:e1 GE:g exp:e2
      {: RESULT=new Absyn.OpExp(gleft,e1,Absyn.OpExp.GE,e2); :}
    | exp:e1 AND:a exp:e2
      {: RESULT=new Absyn.IfExp(aleft,e1,e2,new Absyn.IntExp(0,0)); :}
    | exp:e1 OR:o exp:e2
      {: RESULT=new Absyn.IfExp(oleft,e1,new Absyn.IntExp(0,1),e2); :}
    | ID:i LBRACE RBRACE
      {: RESULT=new Absyn.RecordExp(ileft,sym(i),null); :}
    | ID:i LBRACE fieldinitialseq:a RBRACE
      {: RESULT=new Absyn.RecordExp(ileft,sym(i),a); :}
    | ID:i LBRACK exp:e1 RBRACK OF exp:e2
      {: RESULT=new Absyn.ArrayExp(ileft,sym(i),e1,e2); :}
    | lvalue:l ASSIGN:a exp:e
      {: RESULT=new Absyn.AssignExp(aleft,l,e); :}
```

Le precedenze date sopra per `THEN` e `ELSE` permettono di disambiguare le seguenti due regole

```
    | IF:i exp:e1 THEN exp:e2 ELSE exp:e3
      {: RESULT=new Absyn.IfExp(ileft,e1,e2,e3); :}
    | IF:i exp:e1 THEN exp:e2
      {: RESULT=new Absyn.IfExp(ileft,e1,e2,null); :}
    | WHILE:w exp:e1 DO exp:e2
      {: RESULT=new Absyn.WhileExp(wleft,e1,e2); :}
```

Si noti che un ciclo `for` ha una sintassi astratta un po' diversa dalla sua sintassi concreta, nel senso che dobbiamo creare una dichiarazione di variabile locale

```
    | FOR:f ID:i ASSIGN:a exp:e1 TO exp:e2 DO exp:e3
      {: RESULT=new Absyn.ForExp(fleft,new Absyn.VarDec(aleft,sym(i),
                                      null,e1),e2,e3); :}
    | BREAK:b
      {: RESULT=new Absyn.BreakExp(bleft); :}
```

Si noti che abbiamo fatto in modo che `expseq` possa essere la lista vuota, conformemente alla descrizione del linguaggio data dall'Appel. Inoltre si noti che il non terminale `expseq` ritorna un oggetto della classe `Absyn.ExpList`, mentre una `LetExp` richiede una `Absyn.Exp`. Possiamo trasformare la prima nella seconda costruendo una `SeqExp`

```
    | LET:l decs:d IN expseq:e END
      {: RESULT=new Absyn.LetExp(lleft,d,new Absyn.SeqExp(eleft,e)); :};

decs ::=
      {: RESULT=null; :}
    | decsaux:d
      {: RESULT=d; :} ;
```

In Tiger la mutua ricorsione (`f` chiama `g` che chiama `f`) è ammessa solo fra funzioni (o tipi) dichiarati consecutivamente nel testo del programma. A tal fine la sintassi astratta prevede che una lista di dichiarazioni (una `DecList`) abbia come elementi delle dichiarazioni che, a loro volta, possono essere delle liste. Ogni elemento va interpretato come una sequenza di funzioni (o tipi) dichiarati consecutivamente nel testo del programma. Al fine di creare correttamente tali sottoliste, controlliamo che due dichiarazioni consecutive siano entrambe dichiarazioni di funzioni o entrambe dichiarazioni di tipo. Nel caso lo siano, le "fondiamo" in una sottolista, altrimenti le leghiamo in una `DecList`

```
decsaux ::= dec:h
     {: RESULT=new Absyn.DecList(h,null); :}
   | dec:h decsaux:t
     {: if ((h instanceof Absyn.FunctionDec) &&
            (t.head instanceof Absyn.FunctionDec))
        {
          ((Absyn.FunctionDec)h).next=(Absyn.FunctionDec)(t.head);
          RESULT=new Absyn.DecList(h,t.tail);
        }
        else
        if ((h instanceof Absyn.TypeDec) &&
            (t.head instanceof Absyn.TypeDec))
        {
          ((Absyn.TypeDec)h).next=(Absyn.TypeDec)(t.head);
          RESULT=new Absyn.DecList(h,t.tail);
        }
        else
          RESULT=new Absyn.DecList(h,t); :} ;

dec ::=
     tydec:d
     {: RESULT=d; :}
   | vardec:d
     {: RESULT=d; :}
   | fundec:d
     {: RESULT=d; :} ;

tydec ::=
     TYPE:d ID:i EQ ty:t
     {: RESULT=new Absyn.TypeDec(dleft,sym(i),t,null); :} ;

ty ::=
     ID:i
     {: RESULT=new Absyn.NameTy(ileft,sym(i)); :}
   | LBRACE tyfields:f RBRACE
     {: RESULT=new Absyn.RecordTy(fleft,f); :}
   | ARRAY:a OF ID:i
     {: RESULT=new Absyn.ArrayTy(aleft,sym(i)); :} ;

tyfields ::=
     {: RESULT=null; :}
   | tyfieldsaux:f
     {: RESULT=f; :} ;

tyfieldsaux ::=
     ID:i COLON:c ID:t
```

```
     {: RESULT=new Absyn.FieldList(cleft,sym(i),sym(t),null); :}
   | ID:i COLON:c ID:t COMMA tyfieldsaux:f
     {: RESULT=new Absyn.FieldList(cleft,sym(i),sym(t),f); :} ;

vardec ::=
     VAR:v ID:i ASSIGN exp:e
     {: RESULT=new Absyn.VarDec(vleft,sym(i),null,e); :}
   | VAR:v ID:i COLON ID:t ASSIGN exp:e
     {: RESULT=new Absyn.VarDec(vleft,sym(i),
                              new Absyn.NameTy(tleft,sym(t)),e); :} ;

fundec ::=
     FUNCTION:f ID:i LPAREN tyfields:p RPAREN EQ exp:e
     {: RESULT=new Absyn.FunctionDec(fleft,sym(i),p,null,e,null); :}
   | FUNCTION:f ID:i LPAREN tyfields:p RPAREN COLON ID:t EQ exp:e
     {: RESULT=new Absyn.FunctionDec(fleft,sym(i),p,
                                   new Absyn.NameTy(tleft,sym(t)),
                                   e,null); :} ;
```

2.3 Aggiungere ambiguità!

Abbiamo visto che usiamo una regola per i leftvalue che sembrerebbe ridondante. Supponiamo che tale regola non esista e immaginiamo di voler compilare due programmi Tiger P_1 e P_2 che contengono semplicemente l'espressione `a[5]` e `a[5] of 3`, rispettivamente. Tali programmi sono sintatticamente corretti anche se l'analisi semantica li rifiuterà come errati. Non è comunque difficile immaginare altri programmi che passano anche l'analisi semantica, per esempio

```
let
  type t = array of int
  var a:=t [10] of 0
in
  a[5]
end
```

È però più semplice concentrarci sui programmi `a[5]` e `a[5] of 3`.

Lo stato in cui si trova l'analizzatore lessicale quando sta iniziando a scandire tali programmi contiene i due seguenti item:

`exp`	→.`lvalue LBRACK exp RBRACK`	...
`exp`	→.`ID LBRACK exp RBRACK OF exp`	...
`lvalue`	→.`ID`	`LBRACK`, ...
`lvalue`	→.`lvalue LBRACK exp RBRACK`	...

Leggendo il simbolo `a` (ovvero, un `ID`), finiamo in uno stato che contiene almeno i seguenti item:

`exp`	→`ID`.`LBRACK exp RBRACK OF exp`	...
`lvalue`	→`ID`.	`LBRACK`, ...

A questo punto, vedendo il carattere `[` (ovvero, `LBRACK`), ci troviamo di fronte a un conflitto shift/reduce. Per definizione, JavaCup decide di spostare, per cui finiamo in uno stato che contiene almeno il seguente item

`exp`	→`ID LBRACK`.`exp RBRACK OF exp`	...

Ma questo significa che JavaCup ha dato per scontato che un programma che inizia con `a[` sia qualcosa del tipo `a[5] of 3`! Non abbiamo più alcuna speranza di ricondurre il nostro programma a un leftvalue! La compilazione di P_1 darà errore perché non trova il token `OF`.

Si noti che non sarebbe andata meglio se JavaCup avesse scelto di ridurre. In tal caso, infatto, ci ritroveremmo in uno stato che contiene l'item

```
exp →lvalue.LBRACK exp RBRACK                ...
```

ovvero abbiamo dato per scontato che *non* ci sia il token `OF` dopo la parentesi quadra chiusa. Questa volta, avremmo errore se provassimo a compilare P_2.

Potremmo dire che JavaCup è costretto a scegliere fra spostare e ridurre quando è ancora troppo presto per sapere qual è la scelta giusta. Occorrerebbe aspettare che il compilatore abbia raggiungo il token `RBRACK` e quindi decidere come ridurre la stringa `a[5]` sulla base che segua o meno il token `OF`. Per ottenere questo effetto, aggiungiamo la produzione *ridondante*. A questo punto partiamo da uno stato che contiene gli item:

```
   exp →.lvalue LBRACK exp RBRACK            ...
   exp →.ID LBRACK exp RBRACK OF exp         ...
lvalue →.ID                                  LBRACK, ...
lvalue →.ID LBRACK exp RBRACK                ...
lvalue →.lvalue LBRACK exp RBRACK            ...
```

Leggendo `a` finiamo in uno stato che contiene gli item

```
   exp →ID.LBRACK exp RBRACK OF exp          ...
lvalue →ID.                                  LBRACK, ...
lvalue →ID.LBRACK exp RBRACK                 ...
```

Si noti che a questo punto abbiamo un item in più, il quale ci dà la possibilità di leggere l'espressione fra parentesi quadre *e poi* decidere se ridurre secondo il primo o il terzo degli item dati sopra. Alla fine della lettura del testo `a[5]` ci troveremo quindi in uno stato che contiene gli item

```
   exp →ID LBRACK exp RBRACK.OF exp          ...
lvalue →ID LBRACK exp RBRACK.                ...
```

A questo punto riduciamo con il primo item se segue il token `OF` (P_2), e con il secondo altrimenti (P_1). Entrambi i nostri programmi saranno compilati correttamente.

2.4 Usiamo l'analizzatore sintattico

JavaCup scrive l'analizzatore sintattico nel file `Grm.java`. Inoltre genera il file `sym.java` che contiene l'elenco dei token usati dalla grammatica e il numero unico da associare a ognuno di essi. Spostiamo entrambi questi files dentro la directory `Parse`:

```
> mv Grm.java Parse
> mv sym.java Parse
```

Al fine di usare tale analizzatore sintattico, scriviamo una classe `Parse/Parse.java` che lo esegue e ne stampa il risultato (cioè l'albero di sintassi astratta risultante) e una shell `Parse/Main.java` che ci permette di richiamarlo e applicarlo a un file sorgente Tiger. Dal momento che tale shell ha lo stesso nome di quella dell'analizzatore lessicale (Sezione 1.4). Al fine di non sovrascrivere una shell con l'altra, abbiamo già detto che le memorizzeremo in files diversi che copieremo come `Parse/Main.java` all'occorrenza. Dal momento che anche `Parse/Parse.java` verrà modificato in futuro (per eseguire l'analisi semantica, la generazione del codice, *ecc.*), anch'esso verrà memorizzato in un file `Parse/Parse_syntactical.java` e poi copiato al posto opportuno all'occorrenza.

Parse/Parse_syntactical.java

```
package Parse;

public class Parse {
```

La struttura di errore usata per comunicare gli errori e la variabile che conterrà il risultato dell'analisi sintattica (l'albero di sintassi astratta)

```
  public ErrorMsg.ErrorMsg errorMsg;
  public Absyn.Exp absyn;

  public Parse(String filename) {
    errorMsg = new ErrorMsg.ErrorMsg(filename);
    java_cup.runtime.Symbol symbol;
    java.io.InputStream inp;
```

Apriamo il file specificato dall'utente e diamo errore se ci sono problemi

```
    try { inp = new java.io.FileInputStream(filename); }
    catch (java.io.FileNotFoundException e) {
      System.out.println("File not found: " + filename);
      return;
    }
```

Creiamo l'analizzatore lessicale `Yylex` e l'analizzatore sintattico che lo usa

```
    Grm parser = new Grm(new Yylex(inp,errorMsg), errorMsg);
```

Proviamo a eseguire l'analisi sintattica. Se ci fosse un errore di sintassi o se non si riesce a chiudere il file alla fine, si genererà un'eccezione. In tal caso non abbiamo niente da mostrare all'utente come risultato dell'analisi e quindi terminiamo l'esecuzione

```
    try {
      symbol = parser.parse();
      inp.close();
    }
    catch (Throwable e) { return; }
```

Prendiamo il risultato dell'analisi sintattica e stampiamolo. La classe `Absyn/Print.java` implementa una semplice stampa a discesa ricorsiva dell'albero di sintassi astratta

```
    absyn = (Absyn.Exp)(symbol.value);

    if (absyn != null)
      (new Absyn.Print(System.out)).prExp(absyn,0);
    else
      System.out.println("Attributo semantico pari a null");

    System.out.println("\n\nFine dell'analisi sintattica");
  }
}
```

Vediamo adesso la shell per l'analisi sintattica.

Parse/Main_syntactical.java

```
package Parse;

public class Main {
  public static void main(String argv[]) {
```

Eseguiamo l'analisi sintattica del sorgente Tiger indicato nella linea di comando

```
    String filename = argv[0];
    new Parse(filename);
  }
}
```

Passiamo adesso a vedere come modifichiamo il `makefile`. Indichiamo di seguito solo le modifiche che facciamo a tale file.

`makefile`

Qui indichiamo come creare e compilare l'analizzatore sintattico

```
Parse/Grm.java: Parse/Yylex.java Parse/Grm.cup
        @echo "*** Compiling the syntactical analyzer ***"
        java java_cup.Main -parser Grm -expect 3 -dump_grammar
                -dump_states <Parse/Grm.cup 2>Parse/Grm.err
        mv Grm.java Parse
        mv sym.java Parse
```

Copiamo i files nel posto opportuno e poi compiliamo la shell sintattica

```
syntactical: Parse/Grm.java Parse/Main_syntactical.java \
        Parse/Parse_syntactical.java Parse/Main.class
        @echo "*** Compiling the syntactical shell ***"
        cp Parse/Parse_syntactical.java Parse/Parse.java
        cp Parse/Main_syntactical.java Parse/Main.java
        javac ${JFLAGS} Parse/Main.java
        touch syntactical
```

Questo target diventa un po' più lungo perché abbiamo più roba di eliminare ogni volta

```
.PHONY: clean
clean:
        -rm lexical
        -rm syntactical
        -rm Parse/*.class ErrorMsg/*.class Parse/Yylex.java
            Parse/Grm.java
        -rm Absyn/*.class Symbol/*.class
        touch Parse/Main.class
```

Per compilare e provare il nostro analizzatore sintattico basta dare i seguenti comandi:

```
> make syntactical

> java Parse.Main testcases/test1.tig
```

Capitolo 3

Stack di Attivazione

Durante l'esecuzione di un programma, uno stack contiene i *frame* (o *record di attivazione*) delle procedure. Ogni frame contiene i parametri di ingresso, le variabili locali e i temporanei della procedura a cui fa riferimento. Inoltre i frame sono legati fra di loro in una *catena statica* che permette a una procedura di accedere alle variabili locali dell'ultima istanziazione della procedura che la include sintatticamente nel testo del programma.

Si consideri per esempio il programma in Figura 3.1. La Figura 3.2.(a) mostra lo stack di attivazione al momento dell'inizio dell'esecuzione della funzione `prettyprint`. Lo stack contiene il parametro di ingresso `tree`, un link e il frame per `prettyprint`. Tale frame contiene la variabile locale `output`, dei temporanei e lo spazio per dei parametri di uscita, al momento non utilizzati. La Figura 3.2.(b) mostra come lo stack si evolve al momento della chiamata della procedura `show`. Lo spazio non usato del frame di `prettyprint` viene adesso utilizzato per i due parametri attuali di `show`. Inoltre il link della catena statica permette di accedere alle variabili locali dell'ultima attivazione di `prettyprint`. Questo perché `show` è sintatticamente racchiusa dentro `prettyprint` (Figura 3.1). Anche questa volta, il frame di `show` contiene dello spazio per i parametri in uscita, spazio al momento non utilizzato. La Figura 3.2.(c) mostra l'evoluzione dello stack al momento di una delle due chiamate ricorsive di `show` all'interno di `show`. I parametri sono stati scritti nello spazio a essi riservato, e il puntatore di catena statica punta ancora una volta al frame dell'ultima attivazione di `prettyprint`. La Figura 3.3.(d) mostra come lo stack si evolve al momento della chiamata di `indent` all'interno dell'ultima attivazione di `show`. Questa volta il puntatore di catena statica punta al frame dell'ultima istanziazione di `show`, poiché `indent` è sintatticamente racchiusa dentro `show` (Figura 3.1). Infine, la Figura 3.3.(e) mostra l'evoluzione dello stack al momento della chiamata a `write` all'interno di `indent`. Questa volta `write` è sintatticamente racchiuso dentro `prettyprint`, e quindi il puntatore di catena statica di questa attivazione di `write` deve puntare al frame dell'ultima istanziazione di `prettyprint`.

3.1 Catena statica e coordinate di accesso

In generale, quale puntatore di catena statica deve essere usato al momento della chiamata di una procedura? Occorre far sì che il puntatore di catena statica che precede il frame di una procedura p punti al frame dell'ultima istanziazione della procedura q che sintatticamente racchiude p. A tal fine, si potrebbe pensare di decorare i frame col nome della procedura a cui appartengono. Al momento della chiamata a p, si cerca il frame per la procedura q e si fa puntare a esso il puntatore di catena statica per p. Questa tecnica ha almeno due svantaggi:

1. è necessario cercare il frame di q all'interno dello stack; questa ricerca comporta dei confronti fra stringhe, potenzialmente lenti;
2. due procedure potrebbero avere lo stesso nome: è necessario disambiguarle a tempo di compilazione.

Per questo motivo tale tecnica non è mai usata. Al contrario, si usa la seguente regola dell'*antenato-fratello-figlio*. Si supponga che q chiami p:

```
type tree = {key: string, left: tree, right: tree}

function prettyprint(tree: tree) : string =
  let
    var output := ""

    function write(s: string) =
      output := concat(output,s)

    function show(n: int, t: tree) =
      let function indent(s: string) =
        (for i := 1 to n do write("");
         output := concat(output,s);
         write("\n"))
      in if t = nil then indent(".")
         else (indent(t.key);
               show(n+1,t.left);
               show(n+1,t.right))
      end
  in show(0,tree); output
end
```

Figura 3.1: Un programma Tiger di esempio.

- se p è un figlio di q, allora il puntatore di catena statica per p punta al frame di q (che è in cima allo stack). Questo è esemplificato in Figura 3.2.(b), nel momento cioè in cui `prettyprint` chiama il figlio `show`;
- se p è un fratello di q, allora il puntatore di catena statica di p sarà lo stesso di quello di q. Questo è esemplificato in Figura 3.2.(c), nel momento cioè in cui `show` chiama ricorsivamente se stesso (`show` è un fratello di `show`);
- se p è un antenato k generazioni più vecchio di q, allora si torna indietro sulla catena statica di k passi, e si usa il puntatore di catena statica del record ivi trovato come puntatore di catena statica per p. Questo è esemplificato in Figura 3.3.(e), nel momento cioè in cui `indent` chiama `write`. Dal momento che `write` è un antenato di una generazione più vecchio di `indent`, occorre tornare indietro di un passo sulla catena statica. Finiamo quindi sul frame per la seconda chiamata a `show` (Figura 3.3.(d)). Il suo puntatore di catena statica punta al frame di `prettyprint`, che viene quindi copiato come puntatore di catena statica del frame per `write` (Figura 3.3.(e)).

Come si cerca il valore di una variabile all'interno dello stack? Consideriamo la lettura del valore di `output` alla fine della procedura `prettyprint`. Dal momento che `output` è una variabile locale a `prettyprint`, il suo valore lo troviamo all'interno del frame per `prettyprint`, come mostrato in Figura 3.2.(a). Essendo la prima (e unica) variabile locale di `prettyprint`, sappiamo anche che la troveremo subito sotto l'inizio del frame per `prettyprint`, cioè subito sotto il frame pointer. Consideriamo adesso la lettura del parametro `s` nella procedura `write`. Ancora una volta, la variabile `s` è locale alla procedura `write` ma, questa volta, si tratta di un parametro di input. Concludiamo che il suo valore si trova subito *prima* dell'inizio del frame per `write`, come mostrato in Figura 3.3.(e). Basta saltare lo spazio per il link. Consideriamo infine la lettura della variabile `output` all'interno della procedura `write`. Questa volta stiamo leggendo il valore di una variabile *esterna* a `write`. Tale variabile è definita un livello di scope più all'esterno di `write`. Concludiamo che dobbiamo tornare indietro di un passo sulla catena statica e leggere la prima variabile locale del frame ivi trovato. La Figura 3.3.(e) mostra la correttezza di questo ragionamento.

In conclusione, l'accesso a una variabile richiede la conoscenza di due *coordinate di accesso* (k, n):

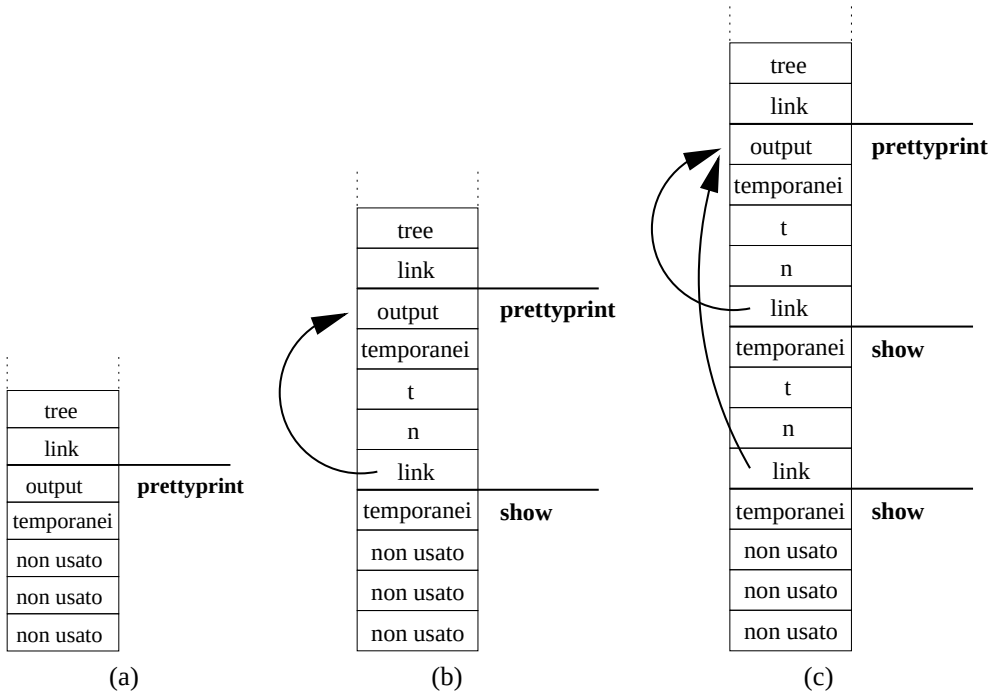

Figura 3.2: Frame per il programma in Figura 3.1.

- k indica di quanti passi tornare indietro sulla catena statica per trovare il frame f in cui la variabile è contenuta; alternativamente, k può identificare il *livello* (o *scope*) in cui una variabile è definita. L'accesso a tale variabile richiederà di tornare indietro di tanti passi quanta è la differenza fra il livello attuale e quello della variabile;
- n indica di quanto muoversi (in più o in meno) dall'inizio di f per trovare il valore della variabile.

Entrambe queste coordinate sono **note a tempo di compilazione**: k è determinabile, in un linguaggio a scope statico, calcolando la differenza fra il livello dell'uso di una variabile e il livello della sua definizione; n è determinabile contando quante variabili locali o parametri sono presenti nella procedura che definisce la variabile, e conoscendo la dimensione di ogni singola variabile.

Si noti in particolare che non serve memorizzare il nome della variabile sullo stack. Questa considerazione non sarebbe più vera in un linguaggio a scope dinamico.

3.2 Frame e livelli per la compilazione di Tiger

Come abbiamo visto nella sezione precedente, il codice generato per un programma Tiger dovrà effettuare le seguenti operazioni:

- calcolare il puntatore di catena statica adeguato a una chiamata di procedura; tale puntatore verrà passato alla procedura come il primo dei suoi parametri;
- accedere al valore di una variabile tramite una sequenza di indirezioni sulla catena statica e uno spostamento finale. La lunghezza di questa sequenza e lo spostamento finale dipendono dalle coordinate di accesso della variabile.

Per permettere la generazione di tale codice, usiamo delle strutture dati ausiliarie che, in fase di compilazione, descrivono la struttura che i frame avranno a tempo di esecuzione. Mentre un oggetto di tipo `Frame/Frame.java` tiene conto dei dettagli implementativi di un frame su una specifica architettura,

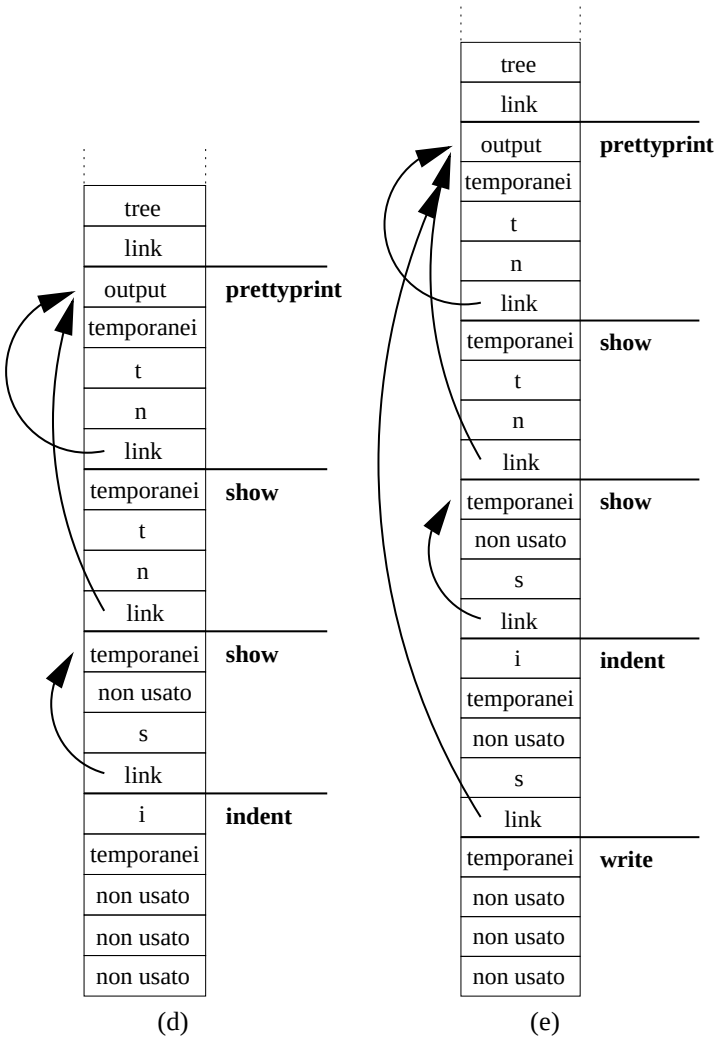

Figura 3.3: Frame per il programma in Figura 3.1.

un oggetto di tipo `Translate/Level.java` descrive, a più alto livello, la relazione di parentela fra i frame delle varie procedure del programma.

Iniziamo col mostrare una classe i cui oggetti implementano *temporanei*, cioè variabili capaci di contenere un valore. Questi temporanei possono essere visti come i regstri di una macchina astratta a infiniti registri.

`Temp/Temp.java`

```
package Temp;

public class Temp  {
  private static int count = 0;
  private int num;

  public String toString() { return "t" + num; }

  public Temp() { num = count++; }
```

```
  public Temp(int n) { num = n; }
}
```

Useremo una classe `Temp/Label.java` che descrive delle *etichette*, cioè dei punti di programma etichettati con un nome.

`Temp/Label.java`

```
package Temp;
import Symbol.Symbol;

public class Label  {
```

Il nome del punto di programma e un contatore usato nel caso in cui si vogliano dei nomi arbitrari ma distinti per `Label` create in successione

```
   private String name;
   private static int count;

   public String toString() {return name;}
```

Vari costruttori

```
   public Label(String n) { name = n; }

   public Label() { this("L" + count++); }

   public Label(Symbol s) {
     this(s.toString() + "_" + count++);
   }
}
```

La classe astratta `Frame/Access.java` descrive una zona di memoria che permette di accedere a un valore.

`Frame/Access.java`

```
package Frame;
import Temp.Temp;
import Temp.Label;

public abstract class Access {
```

Questo metodo genera il codice che permette di accedere al valore a partire dal codice, fornito come parametro, che permette di accedere al frame pointer

```
  public abstract Tree.Exp exp(Tree.Exp base);
}
```

Una lista di `Frame/Access.java` è implementata dalla classe astratta `Frame/AccessList.java`.

La classe astratta `Frame/Frame.java` descrive il frame di una procedura. È possibile creare un frame specificando il numero dei parametri della procedura e indicando per ognuno di essi se sfugge dalla procedura (cioè, se è usato da procedure annidate). Inoltre si possono aggiungere variabili locali. Sarà nostra cura istanziare questa classe con sottoclassi concrete per una specifica architettura.

`Frame/Frame.java`

```
package Frame;

public abstract class Frame {
```

Un frame conosce il nome della procedura a cui fa riferimento e la lista dei suoi parametri formali

```
  public Temp.Label name;
  public AccessList formals;
```

Dal momento che chi crea i frame non deve conoscere l'architettura per la quale tale frame sono stati pensati, usiamo questo pseudo-costruttore. La lista di booleani specifica se i parametri sfuggono o meno

```
  abstract public Frame newFrame(Temp.Label name,
                                 Util.BoolList formals);
```

Questa procedura permette di aggiungere una variabile locale al frame. Il parametro indica se la variabile sfugge o meno

```
  abstract public Access allocLocal(boolean escape);
```

Il frame pointer, cioè il registro in cui tale puntatore è memorizzato

```
  abstract public Temp.Temp FP();
```

Il registro che è usato per contenere il valore di ritorno della funzione il cui metodo è descritto da questo `Frame/Frame.java`

```
  abstract public Temp.Temp RV();
```

Questo metodo genera del codice assembler che permette di dichiarare una zona di memoria etichettata come `lab` che contiene una stringa `lit`. Il risultato è una stringa, ovvero un pezzo di codice assembler. Dal momento che questo codice dipende dalla specifica architettura su cui si compila, questo metodo è inserito dentro `Frame/Frame.java`

```
  abstract public String string(Temp.Label lab, String lit);
```

Questi metodi *avvolgono* il codice del corpo di una funzione in modo che esso si interfacci col sistema di ritorno da procedura usato sulla specifica architettura su cui si compila. Abbiamo due metodi perché alcune funzioni ritornano un valore, altre no

```
  abstract public Tree.Stm procEntryExit1(Tree.Stm body);
  abstract public Tree.Stm procEntryExit1(Tree.Exp body);
```

La dimensione di una parola di memoria

```
  abstract public int wordSize();
}
```

Adesso istanziamo le classi astratte viste sopra per l'architettura Intel. Essenzialmente, tutte le caselle dello stack occupano quattro byte, i parametri di una procedura vengono memorizzati sopra il frame pointer e le variabili locali sotto il frame pointer.

Cominciamo con l'istanziazione della nozione di strumento di accesso a un valore. Dal momento che un valore può essere memorizzato sia sullo stack che in un registro, distinguiamo i due casi con due diverse classi. La classe `Intel/InFrame.java` implementa un valore che viene memorizzato sullo stack. Dal momento che la nozione di link statico non è considerata dalle classi del package `Frame` (verrà recuperata nella classe `Translate/Level.java`), un `Intel/Access.java` contiene solo la coordinata n di accesso alla variabile contenuta nella casella, cioè solo lo spostamento dal frame pointer.

`Intel/InFrame.java`

```
package Intel;
import Temp.Temp;
import Temp.Label;

class InFrame extends Frame.Access {
```

Questo è lo spostamento dal frame pointer, in numero di byte

```
  private int offset;

  public InFrame(int o) { offset=o; }
```

Se `base` è il codice che ci permette di calcolare il valore del frame pointer, aggiungendo lo spostamento `offset` otteniamo il codice che ci permette di accedere al valore presente sullo stack in tale posizione

```
  public Tree.Exp exp(Tree.Exp base) {
    return new Tree.MEM(new Tree.BINOP(Tree.BINOP.PLUS,base,
                                       new Tree.CONST(offset)));
  }
}
```

Un valore può essere anche memorizzato dentro un registro, identificato da un temporaneo. Abbiamo quindi un'altra sottoclasse di `Frame/Access.java`:

`Intel/InReg.java`

```
package Intel;
import Temp.Temp;
import Temp.Label;

class InReg extends Frame.Access {
  private Temp temp;

  public InReg() { temp = new Temp(); }
```

Per accedere al valore di un registro non serve passare attraverso il frame pointer

```
  public Tree.Exp exp(Tree.Exp base) { return new Tree.TEMP(temp); }
}
```

La classe `Intel/Frame.java` istanzia `Frame/Frame.java` implementando i suoi metodi in modo consono all'architettura Intel.

`Intel/Frame.java`

```
package Intel;
import Temp.Temp;
import Temp.Label;

public class IntelFrame extends Frame.Frame {
```

Il numero di parametri e di variabili locali nel frame (in multipli di 4)

```
  private int parCount;
  private int localCount;
```

I registri usati per contenere il frame pointer e il valore di ritorno della funzione il cui frame è descritto da questo `Intel/IntelFrame.java`

```
  private Temp FP;
  private Temp RV;

  public IntelFrame() { this(new Label()); }

  public IntelFrame(Label n) {
```

```
    name = n;
    localCount = 0;
    parCount = 0;
    FP = new Temp();
    RV = new Temp();
  }
```

Lo pseudo-costruttore crea un altro `Intel/IntelFrame.java` e aggiunge tante strutture di accesso ai parametri formali quanti sono i parametri della procedura. Se un parametro sfugge si usa un registro, altrimenti si alloca spazio sul frame

```
  public Frame.Frame newFrame(Label n,Util.BoolList e) {
    Intel.IntelFrame result = new IntelFrame(n);

    Frame.AccessList f = result.formals = null;
    Frame.Access a;

    for (; e!=null; e = e.tail) {
      if (e.head) {
        a = new InFrame(result.parCount);
        result.parCount += 4;
      }
      else a = new InReg();

      if (result.formals == null)
        result.formals = f = new IntelAccessList(a,null);
      else {
        f.tail = new IntelAccessList(a,null);
        f = f.tail;
      }
    }

  return result;
}
```

Le variabili locali si allocano sotto il frame pointer. Le variabili che sfuggono devono essere allocate sullo stack, le altre possono essere allocate nei registri

```
  public Frame.Access allocLocal(boolean escape) {
    if (escape) {
      localCount -= 4;
      return new InFrame(localCount);
    }
    else return new InReg();
  }

  public Temp FP() { return FP; }

  public Temp RV() { return RV; }
```

Il codice assembler che permette di dichiarare una stringa accessibile tramite un'etichetta

```
  public String string(Label label, String value) {
    return label + ": .ascii \"" + value + "\"\n";
  }
```

Se una funzione non ritorna alcun valore, non dobbiamo far nulla al momento del ritorno dalla chiamata. Altrimenti dobbiamo copiare il suo valore di ritorno nel registro usato a tal fine

```
  public Tree.Stm procEntryExit1(Tree.Stm body) { return body; }

  public Tree.Stm procEntryExit1(Tree.Exp body) {
    return procEntryExit1(new Tree.MOVE(new Tree.TEMP(RV),body));
  }

  public int wordSize() { return 4; }
}
```

Adesso aggiungiamo la seconda dimensione alle coordinate di accesso delle variabili. Specifichiamo cioè in quale *livello* si trova una variabile. Un *livello* è una rappresentazione, a tempo di compilazione, del frame per una procedura a tempo di esecuzione.

Translate/Access.java

```
package Translate;

public class Access {
  Level home;
  Frame.Access acc;

  public Access(Level h, Frame.Access a) { home = h; acc = a; }
}
```

Translate/Level.java

```
package Translate;

public class Level {
```

Il livello per una procedura contiene il frame per la procedura e un puntatore al livello della procedura che lo include

```
  Level parent;
  protected Frame.Frame frame;
```

Le coordinate di accesso ai parametri formali della procedura. La coordinata n è il `Frame/Access.java` contenuta fra i `formals` di `frame`

```
  public AccessList formals;

  public Level(Level p, Temp.Label name, Util.BoolList fmls) {
    parent = p;
    frame = parent.frame.newFrame(name,new Util.BoolList(true,fmls));
    formals = null;
    AccessList l = null;
```

Dal punto di vista di un livello, il primo parametro di una procedura è il puntatore di catena statica e quindi non serve avere coordinate di accesso per esso, poiché non è usato esplicitamente nel testo del programma. Quindi qui lo scartiamo. Al contrario, il `frame` non è al corrente della natura di questo parametro, e lo tratterà come un normalissimo parametro per la procedura

```
    Frame.AccessList f = frame.formals.tail;

    for (; f != null; f = f.tail)
      if (formals == null)
```

Le coordinate di accesso dei parametri avranno questo livello come coordinata k

```
        formals = l = new AccessList(new Access(this,f.head),null);
      else {
        l.tail = new AccessList(new Access(this,f.head),null);
        l = l.tail;
      }
  }

  public Level(Frame.Frame f) {
    parent = null;
    frame = f;
    formals = null;
  }
```

Le coordinate di accesso delle variabili locali avranno questo livello come coordinata k

```
  public Access allocLocal(boolean escape) {
    return new Access(this,frame.allocLocal(escape));
  }
}
```

Come preliminare alla generazione del codice, vogliamo associare ad ogni funzione un oggetto della classe `Translate/Level.java` che ne descrive il frame. Inoltre, vogliamo associare a ogni uso di variabile le sue coordinate di accesso, cioè un oggetto della classe `Translate/Access.java`. Il Capitolo 4 descrive l'analisi semantica dei programmi Tiger, e mostra come creare questi oggetti durante tale analisi.

3.3 Il calcolo delle annotazioni di fuga

Abbiamo visto che la decisione di allocare una variabile locale o un parametro sullo stack o in un registro dipende dal fatto che tale variabile o parametro *sfugga* o meno dallo scope che lo definisce. Occorre cioè sapere se esso è utilizzato in funzioni locali allo scope nel quale è definito.

Le annotazioni di fuga vengono scritte nell'apposito campo `escape` di una `Absyn/VarDec.java` (per le variabili locali) o di una `Absyn/FieldList.java` (per i parametri di una funzione). Si definisce una funzione ricorsiva sulla sintassi astratta del programma Tiger, che viene richiamata dal costruttore della classe `FindEscape/FindEscape.java`. Tale discesa ricorsiva utilizza un ambiente che mappa gli identificatori di variabile in oggetti della seguente classe:

`FindEscape/Escape.java`

```
package FindEscape;

abstract class Escape {
```

Questa è la profondità di scope a cui tale variabile è definita. Quando, durante la discesa ricorsiva, incontriamo una variabile, controlliamo se siamo a una profondità maggiore di quella a cui la variabile è definita. In tal caso, la variabile sfugge, e chiamiamo il metodo `setEscape()`

```
  int depth;
  abstract void setEscape();
}
```

Tale classe ha due sottoclassi per variabili locali e parametri formali di una funzione, rispettivamente:

`FindEscape/VarEscape.java`

```
package FindEscape;

class VarEscape extends Escape {
  Absyn.VarDec vd;

  VarEscape(int d, Absyn.VarDec v) { depth=d; vd=v; vd.escape=false; }
  void setEscape() { vd.escape=true; }
}
```

FindEscape/FormalEscape.java

```
package FindEscape;

class FormalEscape extends Escape {
  Absyn.FieldList fl;

  FormalEscape(int d, Absyn.FieldList f) {
    depth=d; fl=f; fl.escape=false;
  }

  void setEscape() { fl.escape=true; }
}
```

La realizzazione della discesa ricorsiva e la verifica della sua funzionalità fanno parte del progetto di quest'anno.

Capitolo 4

Analisi Semantica

4.1 Simboli e tabelle

L'analisi semantica richiede l'utilizzo di simboli per rappresentare gli identificatori del linguaggio e di tabelle che mappano tali simboli in *oggetti*. La reale natura di tali oggetti varia con l'uso che intendiamo fare delle tabelle.

I simboli vengono spesso confrontati per sapere se sono uguali, o chi precede l'altro sulla base di un ordinamento lessicografico. Inoltre ne viene calcolato il valore hash. Occorre fare in modo che tali operazioni dipendano solo dal nome dell'identificatore rappresentato. Facendo in modo che lo stesso identificatore sia rappresentato sempre dallo stesso oggetto di una classe `Symbol/Symbol.java` questa proprietà segue in maniera ovvia.

`Symbol/Symbol.java`

```
package Symbol;

public class Symbol {
```

Il nome dell'identificatore rappresentato

```
  private String name;
```

Al fine di generare lo stesso oggetto ogni volta che si crea un simbolo con un dato nome, memorizziamo tutti gli oggetti creati dentro una tabella hash

```
  private static java.util.Dictionary dict = new java.util.Hashtable();
```

Questo costruttore è privato! Nessuno deve poterlo utilizzare perché non garantisce l'unicità dell'oggetto creato per un dato nome di simbolo

```
  private Symbol(String n) { name = n; }

  public String toString() { return name; }
```

Questo invece è uno pseudo-costruttore pubblico. Si noti che è dichiarato `static` in modo da permetterne l'utilizzo anche quando non è ancora stato creato alcun oggetto di questa classe

```
  public static Symbol symbol(String n) {
```

La funzione `intern` permette di trasformare una stringa in un'altra stringa con lo stesso valore. Garantisce che se due stringhe hanno lo stesso valore allora il loro `intern` è lo stesso oggetto

```
    String u = n.intern();
```

Se abbiamo già creato un simbolo con questo nome lo restituiamo, altrimenti creiamo un nuovo simbolo col costruttore privato, lo inseriamo nella tabella hash e lo ritorniamo

```
    Symbol s = (Symbol)dict.get(u);
    if (s == null) {
      s = new Symbol(u);
      dict.put(u,s);
    }
    return s;
  }
}
```

Per descrivere la gestione delle tabelle e del sistema di apertura e chiusura degli scope, vediamo come implementare la gestione di una tabella che mappa gli identificatori nel loro tipo per il seguente programma Tiger:

```
1: function f(a:int, b:int, c:int) =
2:   (print_int(a + c);
3:    let var j := a + b
4:        var a := "hello"
5:    in print(a); print_int(j);
6:       let var a := b + 1
7:       in print_int(a)
8:       end
9:    end;
10:   print_int(b))
```

Supponiamo che alla linea 1 non ci siano identificatori in scope. La situazione viene rappresentata come in Figura 4.1.(a). Alla linea 2 i tre parametri `a`, `b` e `c` vengono aggiunti allo scope. Questo è rappresentato come in Figura 4.1.(b). Si noti che manteniamo in una variabile `top` l'ultimo simbolo che è stato inserito nel dizionario `dict`. Inoltre ogni simbolo nel dizionario sa qual è il simbolo che era stato inserito subito prima di lui nello scope corrente. Questo significa che partendo da `top` e tornando indietro sul simbolo precedente, si possono scorrere tutti i simboli inseriti nello scope corrente e solo quelli. Alla linea 3 viene creato uno scope locale. Questo ci porta nella situazione rappresentata in Figura 4.1.(c). Abbiamo posto a `null` la variabile `top` e abbiamo copiato in testa alla lista `marks` il valore di `top` prima di questa operazione. Questo valore dovrà essere riabilitato quando usciremo dallo scope locale. Le linee 3 e 4 aggiungono due identificatori allo scope locale. L'aggiunta di `j` è simile alle inserzioni viste in precedenza, e ci porta in Figura 4.1.(d). L'aggiunta di `a`, invece, deve sovrascrivere il vecchio valore di `a`. Al fine di poter riabilitare, in futuro, tale vecchio valore, lo rendiamo raggiungibile legandolo a lista al nuovo valore di `a` (Figura 4.1.(e)). La linea 6 introduce un ulteriore scope locale. Ancora una volta poniamo `top` a `null` e salviamo il vecchio valore di `top` in testa alla lista `marks` (Figura 4.1.(f)). L'aggiunta dell'identificatore `a`, sempre alla linea 6, sovrascrive il vecchio valore che viene comunque legato a lista, come in Figura 4.1.(g). Cosa accadrà alla fine della linea 8, quando lo scope più interno verrà distrutto? Vorremmo tornare nella situazione in Figura 4.1.(e). A tal fine, eliminiamo dalla tabella il simbolo legato alla variabile `top` (cioè `a` in Figura 4.1.(g)). Se tale simbolo ha un simbolo precedente non `null` proseguiamo l'eliminazione con quest'ultimo e così via. Ogni volta che eliminiamo un simbolo dalla tabella, controlliamo se esso ha un vecchio valore legato a lista. In tal caso lo inseriamo al suo posto. Nella Figura 4.1.(g), tutto questo significa sostituire il nuovo valore di `a` col suo vecchio valore. Alla fine eliminiamo la testa della lista `marks` e la copiamo in `top`. Otteniamo esattamente la Figura 4.1.(e). Il ragionamento è simile per la chiusura dello scope alla linea 9: questa volta partiamo dall'identificatore `a` in Figura 4.1.(e) ed eliminiamo `a`, appunto, e `j`, poiché `a` sa di essere stato inserito dopo `j` nello scope. Si noti che il vecchio valore di `a` viene riabilitato. Eliminando la testa della lista `marks` e copiandola in `top` otteniamo esattamente la situazione in Figura 4.1.(b). La chiusura dello scope globale (alla fine della linea 10) ci riporta infine nella situazione in Figura 4.1.(a).

Si faccia attenzione a non confondere le liste di identificatori omonimi in Figura 4.1 con le liste di trabocchi di una tabella hash. L'implementazione della tabella contenuta dentro i quadrati a lati spessi in Figura 4.1 non è significativa per questo algoritmo. Potrebbe anche non essere una tabella hash. Le triple

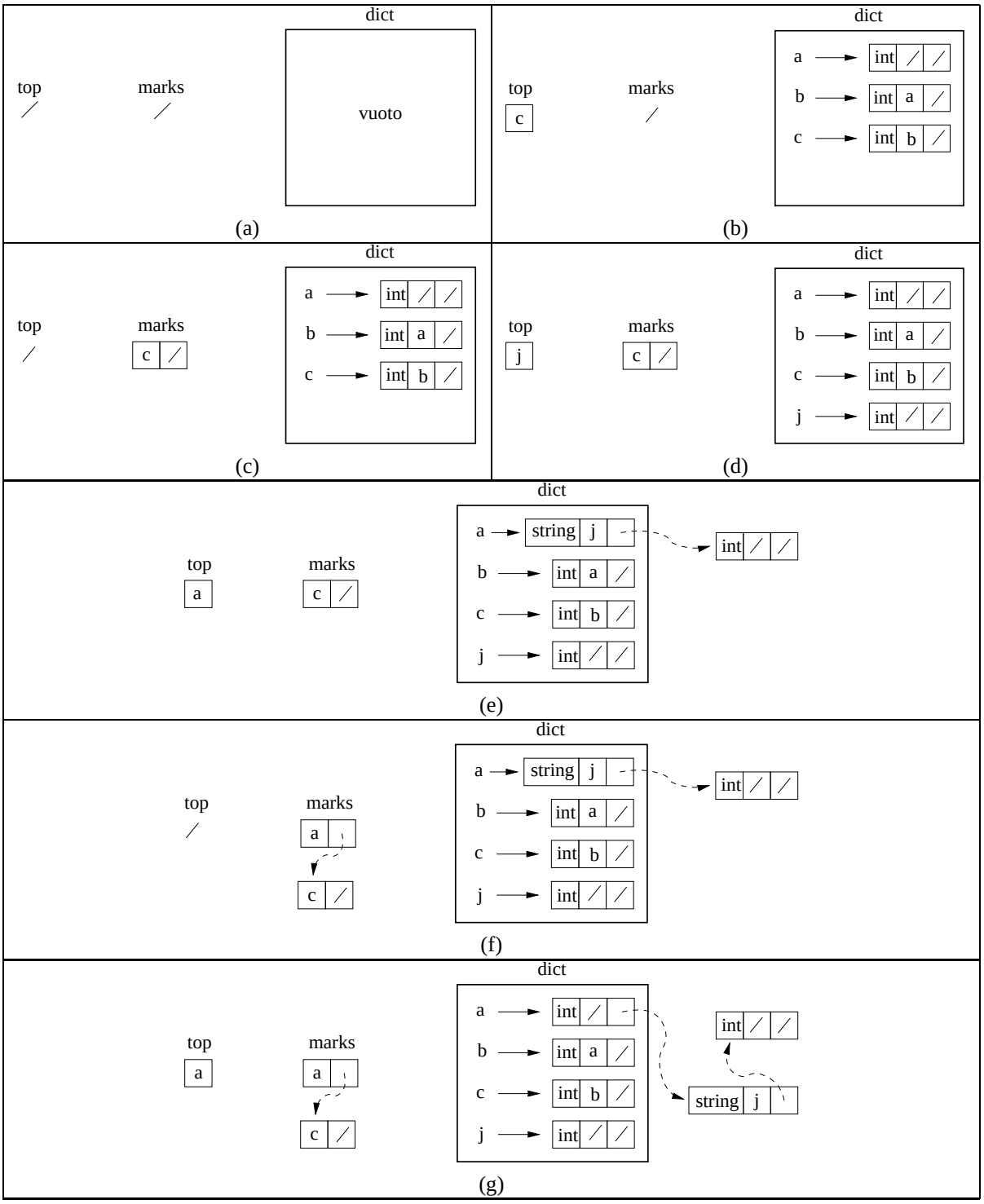

Figura 4.1: La gestione degli scope nelle tabelle di simboli.

esterne a tale tabella solo state sostituite in `dict`, e non ne fanno quindi più parte, neanche nella lista dei trabocchi.

Vediamo adesso come questo algoritmo viene implementato in Java. Le triple a cui sono legati gli identificatori in Figura 4.1 vengono implementate da una classe `Symbol/Binder.java`:

`Symbol/Binder.java`

```
package Symbol;

class Binder {
  Object value;
  Symbol prevtop;
  Binder tail;
  Binder(Object v, Symbol p, Binder t) {
    value = v; prevtop = p; tail = t;
  }
}
```

La lista `marks` richiederebbe un'altra struttura, ma per semplicità decidiamo di utilizzare sempre dei `Binder`s come sopra il cui campo `value` è posto a `null`. Passiamo quindi alle tabelle.

`Symbol/Table.java`

```
package Symbol;

public class Table {
```

Le tre variabili in Figura 4.1. Come già detto, qui potremmo usare qualcosa di diverso da una tabella hash

```
  private java.util.Dictionary dict = new java.util.Hashtable();
  private Symbol top;
  private Binder marks;
```

Il costruttore crea semplicemente la tabella hash

```
  public Table(){}
```

Leggere l'identificatore `key` significa leggerlo nella tabella hash e ritornare il suo valore (il primo campo delle triple in Figura 4.1)

```
  public Object get(Symbol key) {
    Binder e = (Binder)dict.get(key);
    if (e == null) return null;
    else return e.value;
  }
```

La scrittura di un identificatore nella tabella. Lo aggiungiamo in `dict` legando a lista il suo eventuale vecchio valore. Aggiorniamo `top` in modo che contenga sempre l'ultimo identificatore inserito nella tabella

```
  public void put(Symbol key, Object value) {
    dict.put(key, new Binder(value, top, (Binder)dict.get(key)));
    top = key;
  }
```

All'ingresso in un nuovo scope annulliamo `top` e ne salviamo il vecchio valore in testa alla lista `marks`

```
  public void beginScope() {
    marks = new Binder(null,top,marks);
    top = null;
  }
```

All'uscita da uno scope locale eliminiamo tutti gli identificatori inseriti nello scope. Li otteniamo partendo da `top` e tornando indietro verso gli identificatori inseriti subito prima nello stesso scope. Eliminare un identificatore significa decidere se inserire il suo vecchio valore (nel caso in cui esista) o semplicemente rimuoverlo. Alla fine si muove la testa di `marks` in `top`

```
  public void endScope() {
    while (top != null) {
      Binder e = (Binder)dict.get(top);
      if (e.tail != null) dict.put(top,e.tail);
      else dict.remove(top);
      top = e.prevtop;
    }
    top = marks.prevtop;
    marks = marks.tail;
  }
```

Questo metodo ci dà una enumerazione di tutti gli identificatori presenti nella tabella

```
  public java.util.Enumeration keys() { return dict.keys(); }
}
```

4.2 Tipi e tabelle per il controllo dei tipi

La classe da cui tutti i tipi di Tiger sono derivati è `Types/Type.java`:

`Types/Type.java`

```
package Types;

public abstract class Type {
```

Questo metodo ritorna il *vero* tipo rappresentato da un'istanza di questa classe. Normalmente esso ritorna l'oggetto stesso su cui è applicato, ma la classe `Types/NAME.java` (si veda dopo) rappresenta un nome che sta per un tipo, e quindi questo metodo verrà ridefinito

```
   public Type actual() { return this; }
}
```

Le sottoclassi di `Types/Type.java` sono `Types/ARRAY.java`, `Types/INT.java`, `Types/NIL.java`, `Types/RECORD.java`, `Types/VOID.java`, `Types/STRING.java` e `Types/NAME.java`. Vediamone alcune.

`Types/STRING.java`

```
package Types;

public class STRING extends Type {
        public STRING(){}
}
```

`Types/ARRAY.java`

```
package Types;

public class ARRAY extends Type {
```

Il tipo degli elementi dell'array

```
   public Type element;
   public ARRAY(Type e) { element = e; }
}
```

Types/NAME.java

```
package Types;

public class NAME extends Type {
```

Questa classe è usata per rappresentare un tipo di cui si conosce solo il nome `name` e che verrà meglio precisato in futuro. È necessaria per la gestione dei tipi (mutualmente) ricorsivi: prima si inseriscono dei tipi `NAME` nell'ambiente dei tipi, poi si costruiscono le espressioni di tipo a essi legati (le quali possono fare riferimento ad alcuni di questi `NAME`) e infine si sostituiscono le `NAME` col vero tipo a cui fanno riferimento, memorizzandolo nel campo `binding`. Si veda la Sezione 4.4

```
   public Symbol.Symbol name;
   private Type binding;

   public NAME(Symbol.Symbol n) { name=n; }
```

Questo metodo ci permette di controllare se l'identificatore di tipo rappresentato da questa `NAME` è già stato legato al tipo che esso rappresenta o meno

```
   public boolean bound() { return binding != null; }
```

Questo metodo restituisce il tipo legato all'identificatore rappresentato da questa `NAME`

```
   public Type actual() { return binding.actual(); }
```

Questo metodo ci permette di legare l'identificatore rappresentato da questa `NAME` a un tipo

```
   public void bind(Type t) { binding = t; }
}
```

Useremo due tabelle per il controllo dei tipi: la prima, `tenv`, mappa identificatori di tipo in un `Types/Type.java` e la seconda, `venv`, mappa identificatori di variabili nel loro tipo di dichiarazione e identificatori di funzione nella loro segnatura. In genere, diciamo che `venv` mappa identificatori in oggetti della classe `Semant/Entry.java`, la quale ha due sottoclassi `Semant/VarEntry.java` per le variabili e `Semant/FunEntry.java` per le funzioni.

Semant/Entry.java

```
package Semant;

abstract class Entry {}
```

Semant/VarEntry.java

```
package Semant;

class VarEntry extends Entry {
```

Queste sono le coordinate (k, n) di accesso alla variabile

```
  public Translate.Access access;
```

Una variabile viene mappata nel suo tipo di dichiarazione. Inoltre un flag booleano ci dice se su tale variabile è lecito fare assegnamenti. Questa possibilità è utilizzata per le variabili dichiarate da una espressione `for`: l'indice non può essere assegnato all'interno del corpo del `for`. In futuro potrebbe essere utilizzata per aggiungere dichiarazioni di costanti a Tiger

```
  public Types.Type ty;
  public boolean constant;
```

Vari costruttori

```
  public VarEntry(Types.Type t) {
    ty = t; constant = false; access = null;
  }

  public VarEntry(Types.Type t,boolean c) {
    ty = t; constant = c; access = null;
  }

  public VarEntry(Translate.Access a,Types.Type t) {
    ty = t; constant = false; access = a;
  }

  public VarEntry(Translate.Access a,Types.Type t,boolean c) {
    ty =t; constant = c; access = a;
  }
}
```

Semant/FunEntry.java

```
package Semant;

class FunEntry extends Entry {
```

Una funzione ha il suo livello, che contiene il frame per i suoi parametri e per le sue variabili locali

```
  public Translate.Level level;
```

Il codice generato per una funzione inizierà con una etichetta a cui saltare per chiamare la funzione

```
  public Temp.Label label;
```

Il nome di una funzione viene mappato in queste informazioni, che ci dicono qual è il tipo dei suoi parametri e quale quello del suo valore di ritorno (eventualmente `VOID`)

```
  public Types.RECORD formals;
  public Types.Type result;
```

Due costruttori

```
  public FunEntry(Symbol.Symbol f, Types.RECORD f, Types.Type r) {
    level = null; formals = f; result = r;
    label = new Temp.Label(sym);
  }

  public FunEntry(Translate.Level le,Temp.Label la,
                  Types.RECORD f,Types.Type r) {
    level = le; label = la; formals = f; result = r;
  }
}
```

Vediamo adesso la classe che contiene tutte le informazioni che ci permettono di fare il controllo dei tipi di programmi Tiger.

Semant/Env.java

```
package Semant;

class Env {
```

Le due tabelle di cui abbiamo parlato, una per gli identificatori di variabile e funzione e una per gli identificatori di tipo. Segue poi la struttura usata per comunicare errori all'utente durante il controllo dei tipi

```
  public Symbol.Table venv;
  public Symbol.Table tenv;
  public ErrorMsg.ErrorMsg errorMsg;
```

Il costruttore inizializza le due tabelle inserendo in `venv` le funzioni predefinite e in `tenv` i due identificatori di tipo predefiniti di Tiger

```
  public Env(ErrorMsg.ErrorMsg err){
    errorMsg = err;

    venv = new Symbol.Table();
    tenv = new Symbol.Table();

    tenv.put(Symbol.Symbol.symbol("int"),Semant.INT);
    tenv.put(Symbol.Symbol.symbol("string"),Semant.STRING);

    s = sym("print");
    venv.put(s,
      new FunEntry(s,new Types.RECORD(sym("s"),Semant.STRING,null),
                   Semant.VOID));

    s = sym("flush");
    venv.put(s,new FunEntry(s,null,Semant.VOID));

    s = sym("getchar");
    venv.put(s,new FunEntry(s,null,Semant.STRING));

    s = sym("ord");
    venv.put(s,
      new FunEntry(s,new Types.RECORD(sym("s"),Semant.STRING,null),
                   Semant.INT));

    s = sym("chr");
    venv.put(s,
      new FunEntry(s,new Types.RECORD(sym("i"),Semant.INT,null),
                   Semant.STRING));

    s = sym("size");
    venv.put(s,
      new FunEntry(s,new Types.RECORD(sym("s"),Semant.STRING,null),
                   Semant.INT));

    s = sym("substring");
    venv.put(s,
      new FunEntry(s,new Types.RECORD(sym("s"),Semant.STRING,
                     new Types.RECORD(sym("first"),Semant.INT,
                     new Types.RECORD(sym("n"),Semant.INT,null))),
      Semant.STRING));
```

```
    s = sym("concat");
    venv.put(s,
      new FunEntry(s,new Types.RECORD(sym("s1"),Semant.STRING,
                     new Types.RECORD(sym("s2"),Semant.STRING,null)),
      Semant.STRING));

    s = sym("not");
    venv.put(s,
      new FunEntry(s,new Types.RECORD(sym("i"),Semant.INT,null),
                   Semant.INT));

    s = sym("exit");
    venv.put(s,
      new FunEntry(s,new Types.RECORD(sym("i"),Semant.INT,null),
                   Semant.VOID));
  }
```

Questo metodo privato trasforma una stringa in un simbolo univoco che lo rappresenta (si veda `Symbol/Symbol.java`)

```
  private Symbol.Symbol sym(String name) {
    return Symbol.Symbol.symbol(name);
  }
}
```

4.3 Controllo dei tipi

Il controllo dei tipi viene effettuato dalla classe Java `Semant/Semant.java`. Essa inizia con dei metodi ausiliari e poi continua con dei metodi che permettono di fare il controllo dei tipi di espressioni e dichiarazioni. In questa sezione esaminiamo i metodi ausiliari di `Semant/Semant.java`.

Abbiamo bisogno di una classe `Semant/ExpTy.java` che è usata per contenere il risultato del controllo dei tipi delle espressioni. Esso è formato dal tipo dell'espressione e dal codice in cui l'espressione viene compilata, cioè un oggetto della classe `Translate/Exp.java` (si veda il capitolo 5).

`Semant/ExpTy.java`

```
package Semant;
import Translate.Exp;

class ExpTy {
    Exp exp;
    Types.Type ty;

    ExpTy(Exp e, Types.Type t) {exp=e; ty=t;}
}
```

Ecco finalmente la parte iniziale del controllore dei tipi.

`Semant/Semant.java`

```
package Semant;

public class Semant {
```

L'ambiente che contiene tutte le informazioni su identificatori di variabili, funzioni e tipi

```
  private Env env;
```

Il livello corrente, cioè quello che descrive lo scope dentro il quale ci troviamo al momento

```
  private Translate.Level level;
```

Il traduttore nel codice intermedio, che verrà descritto nel Capitolo 5

```
  public Translate.Translate codegen;
```

Il controllo dei tipi include anche il controllo che ogni istruzione `break` sia posta in un punto di programma interno a un ciclo `for` o `while`. A tale fine potremmo utilizzare un flag che indica se un `break` è ammesso o meno. Però, dal momento che in futuro, al momento della generazione del codice, vorremo sapere molto di più, cioè dove esattamente salta il flusso di controllo nel caso in cui si esegua un `break`, decidiamo di usare una label che indica appunto tale punto di programma. È lecito incontrare un `break` se e solo se tale label non è `null`

```
  private Temp.Label _break;
```

Due variabili dichiarate di tipo intero ma in punti di programma diversi sono compatibili, cioè il valore dell'una può essere assegnato all'altra. Al fine di controllare l'uguaglianza di tipo con un semplice ed efficiente test Java `==`, usiamo sempre lo stesso oggetto per rappresentare i tipi di base. Diverse espressioni di tipo composte avranno invece tipo diverso, cioè genereranno oggetti diversi

```
  static Types.Type INT = new Types.INT();
  static Types.Type STRING = new Types.STRING();
  static Types.Type VOID = new Types.VOID();
  static Types.Type NIL = new Types.NIL();
```

Vari costruttori. Creiamo un livello di partenza che usa un frame di tipo `Intel/IntelFrame.java`. Se volessimo cambiare l'architettura di riferimento, dovremmo cambiare solo questa linea. Creiamo anche un traduttore verso il codice intermedio (Capitolo 5)

```
  public Semant(ErrorMsg.ErrorMsg err) { this(new Env(err)); }

  private Semant(Env e) {
    this(e, new Translate.Level(new Intel.IntelFrame()),
         new Translate.Translate(),null);
  }

  private Semant(Env e, Translate.Level l, Translate.Translate c,
                 Temp.Label b) {
    codegen=c; env=e; level=l; _break=b;
  }
```

Questo è il metodo che effettua il controllo dei tipi per l'intero programma, rappresentato dal suo albero di sintassi astratta. Prima del controllo decoriamo l'albero con le annotazioni di fuga per le variabili. Si noti che questo metodo ritorna il codice generato per l'intero programma

```
  public Translate.Exp transProg(Absyn.Exp exp) {
    FindEscape.FindEscape dummy = new FindEscape.FindEscape(exp);
    return transExp(exp).exp;
  }
```

Questa funzione trasforma le annotazioni di fuga per i parametri di una funzione in una lista di booleani

```
  private Util.BoolList extractEscapes(Absyn.FieldList p) {
    if (p == null) return null;

    return new Util.BoolList(p.escape, extractEscapes(p.tail));
  }
```

Controlliamo che il risultato del controllo dei tipi di una qualche espressioni sia intero. Si ricordi l'unicità degli oggetti che rappresentano i tipi di base di Tiger. Si noti che ritorniamo il codice dell'espressione

```
  private Translate.Exp checkInt(ExpTy et, int pos)
  {
    if (et.ty != INT) env.errorMsg.error(pos,"integer required");

    return et.exp;
  }
```

Può un valore di tipo `t2` essere assegnato a una variabile di tipo `t1`?

```
  private void checkCompatible(Types.Type t1,Types.Type t2,int pos)
  {
```

Se i due tipi sono lo stesso la risposta è sì, con la piccola cautela di non permettere l'assegnamento fra due espressioni di tipo `NIL` (caso degenere che non capita mai in Tiger...)

```
    if (t1 == t2 && !(t2 instanceof Types.NIL)) return;
```

È lecito assegnare `nil` a una variabile di tipo record

```
    if (t2 instanceof Types.NIL && t1 instanceof Types.RECORD) return;

    env.errorMsg.error(pos,"incompatible types");
  }
```

Siano `et1` ed `et2` il risultato del controllo dei tipi di due espressioni. È lecito confrontarle con operatori di uguaglianza o disuguaglianza o porle nei due rami di un `if then else`? In caso positivo, ritorna il tipo più generale fra i due. Se il flag `v` è posto a `true` allora si considerano unificabili anche `VOID` con `VOID` e `NIL` con `NIL`

```
  private Types.Type checkUnifiable(ExpTy et1, ExpTy et2, int pos,
                                    boolean v)
  {
```

Se i due tipi sono lo stesso allora la risposta è sì, ma evitando i casi di `NIL` e `VOID`. Tali tipi sono considerati unificabili solo se il flag `v` passato al metodo è posto a `true`

```
    if (et1.ty == et2.ty && (et1.ty == INT || et1.ty == STRING ||
                             et1.ty instanceof Types.RECORD
                             || et1.ty instanceof Types.ARRAY))
       return et1.ty;

    if (v && et1.ty == et2.ty && (et1.ty == VOID || et1.ty == NIL))
       return et1.ty;
```

È anche possibile confrontare `nil` con un record, e in tal caso il record è il tipo più generale

```
    if (et1.ty == NIL && et2.ty instanceof Types.RECORD) return et2.ty;

    if (et2.ty == NIL && et1.ty instanceof Types.RECORD) return et1.ty;

    env.errorMsg.error(pos,"incompatible types");
    return null;
  }
```

4.4 Controllo dei tipi per le dichiarazioni

Le dichiarazioni arricchiscono le tabelle con nuovi identificatori.

Questo metodo effettua il controllo dei tipi per una dichiarazione. Non è altro che un dispatcher che chiama i metodi opportuni sulla base della classe di sintassi astratta della dichiarazione

```
Translate.Exp transDec(Absyn.Dec d) {
  if (d instanceof Absyn.VarDec) return transDec((Absyn.VarDec)d);
  if (d instanceof Absyn.TypeDec) return transDec((Absyn.TypeDec)d);
  if (d instanceof Absyn.FunctionDec)
    return transDec((Absyn.FunctionDec)d);

  throw(new Error("transDec"));
}
```

La dichiarazione di una variabile

```
Translate.Exp transDec(Absyn.VarDec d) {
```

Facciamo il controllo dei tipi dell'espressione di inizializzazione

```
  ExpTy et = transExp(d.init);
  Translate.Access local;
```

Se non c'è un tipo di dichiarazione per la variabile, allora il tipo dell'espressione di inizializzazione non deve essere `NIL`

```
  if (d.typ == null) {
    if (et.ty == NIL) {
      env.errorMsg.error(d.init.pos,
                         "untyped initializer cannot be nil");
      return null;
    }
    else {
```

Aggiungiamo una variabile locale al livello corrente (si noti l'uso dell'annotazione di fuga) e arricchiamo l'ambiente con un nuovo identificatore di variabile. La `VarEntry` per tale identificatore farà riferimento alle coordinate per la variabile locale ritornate da `allocLocal`

```
      local = level.allocLocal(d.escape);
      env.venv.put(d.name,new VarEntry(local,et.ty));

      return codegen.VarDec(local,et.exp,level);
    }
  }
  else {
```

Altrimenti trasformiamo la dichiarazione di tipo nel tipo che essa rappresenta (mai una `NAME`!) e controlliamo che il valore di inizializzazione abbia tipo compatibile col tipo di dichiarazione

```
    Types.Type t = transTy(d.typ).actual();
    checkCompatible(t,et.ty,d.init.pos);
```

Aggiungiamo una nuova variabile locale e arricchiamo l'ambiente delle variabili

```
    local = level.allocLocal(d.escape);
    env.venv.put(d.name, new VarEntry(local,t));

    return codegen.VarDec(local,et.exp,level);
  }
}
```

Identica alla `transDec` ma la variabile viene inserita nell'ambiente col flag che indica che non è possibile usarla alla sinistra di un assegnamento

```
Translate.Exp transDecConstant(Absyn.VarDec d) {
  ExpTy et = transExp(d.init);
  Translate.Access local;

  if (d.typ == null) {
    if (et.ty == NIL) {
      env.errorMsg.error(d.init.pos,
                         "untyped initializer cannot be nil");
      return null;
    }
    else {
      local = level.allocLocal(d.escape);
      env.venv.put(d.name,new VarEntry(local,et.ty,true));

      return codegen.VarDec(local,et.exp,level);
    }
  }
  else {
    Types.Type t = transTy(d.typ).actual();
    checkCompatible(t,et.ty,d.init.pos);

    local = level.allocLocal(d.escape);
    env.venv.put(d.name, new VarEntry(local,t,true));

    return codegen.VarDec(local,et.exp,level);
  }
}
```

La dichiarazione di una sequenza di tipi potenzialmente ricorsivi

```
Translate.Exp transDec(Absyn.TypeDec d) {
  boolean flag = true;
  int Counter = 0;
  Absyn.TypeDec start = d, Temp = null;
  Translate.Exp code = null;
```

Dobbiamo dare errore se lo stesso tipo è definito in due dichiarazioni di questa sequenza

```
  for (; d != null; d = d.next) {
    for (Temp = d.next; Temp != null; Temp = Temp.next)
      if (d.name == Temp.name) {
        env.errorMsg.error(Temp.pos,"Redefinition of " + d.name +
                           " in the same block of definitions");
        return null;
      }
```

Accumuliamo il codice di tutte le dichiarazioni

```
      code = (code == null)? codegen.TypeDec() :
        codegen.Append(code,codegen.TypeDec());
    }
```

Inseriamo i nomi dei tipi definiti in questo gruppo di dichiarazioni come `NAME`. Dopo dovremo legare questi `NAME` al reale tipo che essi rappresentano

```
    for (d = start; d != null; Counter++, d = d.next)
      env.tenv.put(d.name,new Types.NAME(d.name));
```

Finché ci sono ancora NAME non legati al tipo che essi rappresentano…

```
    while (flag) {
      flag = false;
```

Ad ogni iterazione almeno un NAME deve venire legato al reale tipo che esso rappresenta, altrimenti c'è una dichiarazione ciclica. Se quindi, dopo tanti cicli quante dichiarazioni in questa sequenza, ancora ci sono dei NAME da legare al loro tipo, concludiamo che c'è un ciclo in questa sequenza di dichiarazioni

```
      Counter--;
      if (Counter < 0) {
        env.errorMsg.error(start.pos,
                           "illegal cyclic type declarations");
        return null;
      }
```

Scorriamo di nuovo tutte le dichiarazioni in questa sequenza e cerchiamo di legare i nomi dei tipi non ancora legati al tipo che essi rappresentano, purché quest'ultimo sia già stato legato a qualcosa. Si noti che non leghiamo mai un NAME a un altro NAME, grazie all'uso di `actual`

```
      for (d = start; d != null; d = d.next) {
        Types.Type t = transTy(d.ty);

        if (t instanceof Types.NAME && !(((Types.NAME)t).bound()))
          flag = true;
        else
          ((Types.NAME)(env.tenv.get(d.name))).bind(t.actual());
      }
    }

    return code;
  }
```

La dichiarazione di una sequenza di funzioni potenzialmente ricorsive

```
  Translate.Exp transDec(Absyn.FunctionDec d) {
    Types.Type result;
    Absyn.FunctionDec Temp1 = d, Temp2 = null;
    Translate.Level newlevel;
    Translate.Exp code = null;
```

Per ogni funzione dichiarata, inseriamo la sua segnatura nell'ambiente delle variabili e delle funzioni…

```
    for (; d != null; d = d.next) {
```

…controllando che non ci siano due funzioni con lo stesso nome

```
      for (Temp2 = d.next; Temp2 != null; Temp2 = Temp2.next)
        if (d.name == Temp2.name) {
          env.errorMsg.error(Temp2.pos,"Redefinition of " + d.name +
                             " in the same block of definitions");
          return null;
        }
```

Il tipo di ritorno di una funzione può essere assente, e in tal caso si assume che sia VOID

```
      result = (d.result != null) ? transTy(d.result).actual() : VOID;
```

Trasformiamo la lista di espressioni di tipo in una lista di tipi

```
      Types.RECORD formals = transTypeFields(d.params);
```

Creiamo una nuova etichetta per il codice generato per la funzione. Quindi creiamo un nuovo livello per la funzione. Tale livello avrà il livello corrente `level` come padre

```
      Temp.Label name = new Temp.Label(d.name);
      newlevel = new Translate.Level(level,name,
                                     extractEscapes(d.params));
```

Inseriamo la segnatura di una funzione nell'ambiente per variabili e funzioni

```
      env.venv.put(d.name,new FunEntry(newlevel,name,formals,result));
    }
```

Adesso possiamo passare al controllo dei tipi dei corpi delle funzioni

```
    for (d = Temp1; d != null; d = d.next) {
      FunEntry f = (FunEntry)env.venv.get(d.name);
```

Dobbiamo fare il controllo dei tipi di una funzione in un ambiente arricchito con i parametri formali della stessa. Per tali parametri dobbiamo fornire le coordinate di accesso. Esse sono quelle contenute nel livello della funzione

```
      env.venv.beginScope();
      Translate.AccessList al = f.level.formals;

      Absyn.FieldList p = d.params;
      for (; p != null; p = p.tail, al = al.tail) {
        Types.Type temp = (Types.Type)env.tenv.get(p.typ);
```

I tipi dei parametri devono essere stati definiti

```
        if (temp == null) {
          env.errorMsg.error(p.pos,"undeclared type " +
                             p.typ.toString());
          env.venv.endScope();
          return null;
        }
```

Arricchiamo l'ambiente con i parametri formali

```
        env.venv.put(p.name,new VarEntry(al.head,temp));
      }
```

Creiamo un oggetto `Semant/Semant.java` in cui non sia possibile eseguire `break` (a meno di introdurre un ciclo). Quindi lo usiamo per fare il controllo dei tipi del corpo della funzione

```
      Semant newsem = new Semant(env,f.level,codegen,null);

      ExpTy et = newsem.transExp(d.body);
      env.venv.endScope();
```

Controlliamo che il tipo del corpo della funzione sia compatibile col tipo di dichiarazione del risultato

```
      checkCompatible(f.result.actual(),et.ty,d.body.pos);
```

Accumuliamo i codici delle dichiarazioni di funzione

```
      code = (code == null) ?
        codegen.FunctionDec(f.label,et.exp,newsem.level) :
        codegen.Append(code,codegen.FunctionDec
                       (f.label,et.exp,newsem.level));
    }

    return code;
  }
```

Trasformiamo la lista dei parametri formali di una funzione in una lista di tipi

```
  private Types.RECORD transTypeFields(Absyn.FieldList p)
  {
    Types.RECORD result = null, cursor = null;

    for (; p!=null; p=p.tail) {
```

I tipi dei parametri non devono essere indefiniti

```
      Types.Type t = ((Types.Type)(env.tenv.get(p.typ))).actual();
      if (t == null) {
        env.errorMsg.error(p.pos,"undefined type " + p.typ.toString());
        continue;
      }

      if (cursor == null)
        result = cursor = new Types.RECORD(p.name,t,null);
      else {
        cursor.tail = new Types.RECORD(p.name,t,null);
        cursor = cursor.tail;
      }
    }

    return result;
  }
```

Questo metodo trasforma l'albero astratto di una espressione di tipo nel tipo che essa rappresenta

```
  Types.Type transTy(Absyn.Ty t) {
    if (t instanceof Absyn.NameTy) return transTy((Absyn.NameTy)t);
    if (t instanceof Absyn.RecordTy) return transTy((Absyn.RecordTy)t);
    if (t instanceof Absyn.ArrayTy) return transTy((Absyn.ArrayTy)t);

    throw new Error("transTy");
  }
```

I nomi di tipo usati devono già essere stati dichiarati

```
  Types.Type transTy(Absyn.NameTy t) {
    Types.Type Result = (Types.Type)(env.tenv.get(t.name));

    if (Result != null) return Result;

    env.errorMsg.error(t.pos,"undefined type " + t.name.toString());
    return INT;
  }
```

Trasforma un'espressione di un tipo record nel tipo che essa rappresenta

```
  Types.Type transTy(Absyn.RecordTy t) {
    Types.RECORD Result = null, Cursor = null;
    Types.Type temp;

    for (Absyn.FieldList fs = t.fields; fs != null; fs = fs.tail)
    {
```

I nomi di tipo usati nel record devono essere stati dichiarati

```
      temp = (Types.Type)(env.tenv.get(fs.typ));
      if (temp == null) {
        env.errorMsg.error(fs.pos,"undefined type " +
                           fs.typ.toString());
        return INT;
      }

      if (Cursor != null) {
        Cursor.tail = new Types.RECORD(fs.name,temp,null);
        Cursor = Cursor.tail;
      }
      else
        Cursor = Result = new Types.RECORD(fs.name,temp,null);
    };

    return Result;
  }
```

Trasforma un'espressione di un tipo array nel tipo che essa rappresenta

```
  Types.Type transTy(Absyn.ArrayTy t) {
    Types.Type temp = (Types.Type)(env.tenv.get(t.typ));
```

Il nome del tipo degli elementi dell'array deve essere stato definito

```
    if (temp == null) {
      env.errorMsg.error(t.pos,"undefined type " + t.typ.toString());
      return INT;
    }

    return new Types.ARRAY(temp);
  }
}
```

4.5 Controllo dei tipi per i leftvalue

Consideriamo qui la parte di `Semant/Semant.java` che gestisce il controllo dei tipi per i leftvalue.

Questo metodo effettua il controllo dei tipi per un leftvalue. Non è altro che un dispatcher che chiama i metodi opportuni sulla base della classe di sintassi astratta del leftvalue. Il flag `assigned` indica se il leftvalue è usato a sinistra di un assegnamento. L'ultima riga del metodo non dovrebbe mai essere raggiunta. Il tipo ritornato da questo metodo non sarà mai un `Types/NAME.java`

```
  ExpTy transVar(Absyn.Var l, boolean assigned) {
    if (l instanceof Absyn.SimpleVar)
      return transVar((Absyn.SimpleVar)l,assigned);
    if (l instanceof Absyn.SubscriptVar)
      return transVar((Absyn.SubscriptVar)l,assigned);
```

```
    if (l instanceof Absyn.FieldVar)
      return transVar((Absyn.FieldVar)l,assigned);

    throw new Error("transVar");
  }
```

Il controllo dei tipi per un identificatore di variabile controlla che esso sia stato in precedenza dichiarato, e quindi è contenuto in `venv`, ed è effettivamente legato a un oggetto di tipo `Semant/VarEntry.java` (non è una funzione, quindi)

```
  ExpTy transVar(Absyn.SimpleVar v, boolean assigned) {
    Entry x = (Entry)env.venv.get(v.name);

    if (!(x instanceof VarEntry)) {
      env.errorMsg.error(v.pos,"undefined variable " +
                         v.name.toString());
      return new ExpTy(null,INT);
    }

    VarEntry ent = (VarEntry)x;
```

Se questo identificatore si trova a sinistra di un assegnamento, allora deve essere possibile modificarlo

```
    if (assigned && ent.constant) {
      env.errorMsg.error(v.pos,"variable " + v.name.toString() +
                         " cannot be assigned to");
      return new ExpTy(null,INT);
    }
```

Generiamo il codice che legge questo identificatore e restituiamo il tipo che abbiamo trovato in `venv` per questa variabile. Si noti che tramite la chiamata ad `actual` garantiamo che non venga mai restituito un oggetto di classe `Types/NAME.java` (se ci siamo preoccupati di garantire che un `NAME` non sia mai legato a un altro `NAME`, si veda dopo)

```
    return new ExpTy(codegen.SimpleVar(ent.access,level),
      ent.ty.actual());
  }

  ExpTy transVar(Absyn.SubscriptVar v, boolean assigned) {
```

Un leftvalue del tipo `var[index]` richiede il controllo dei tipi per `var` e `index`. Poi si controlla che il tipo di `var` sia un array e quello di `index` sia `INT`

```
    ExpTy array = transVar(v.var,assigned);
    ExpTy index = transExp(v.index);

    if (array.ty instanceof Types.ARRAY) {
      checkInt(index,v.index.pos);
```

Se tutti i controlli hanno avuto buon fine, restituiamo il codice per leggere l'elemento dell'array e il tipo degli elementi dell'array. Si noti di nuovo l'uso di `actual`

```
      return new ExpTy(codegen.SubscriptVar(array.exp,index.exp,level),
                       ((Types.ARRAY)array.ty).element.actual());
    }
    else {
      env.errorMsg.error(v.pos,"array type required");
      return new ExpTy(null,INT);
```

```
    }
  }

  ExpTy transVar(Absyn.FieldVar v, boolean assigned) {
```

Il controllo dei tipi per un leftvalue del tipo `var.f` richiede in primo luogo di effettuare il controllo dei tipi per `var`. Tale leftvalue deve avere tipo `RECORD`

```
    ExpTy var = transVar(v.var,assigned);

    if (var.ty instanceof Types.RECORD) {
```

Scorriamo quindi i campi del record alla ricerca di un campo chiamato `f` (cioè, `v.field`)

```
      int offset = 0;
      Types.RECORD Cursor;

      for (Cursor = ((Types.RECORD)var.ty); Cursor!=null;
        offset++, Cursor = Cursor.tail)
        if (Cursor.fieldName == v.field) break;

      if (Cursor == null) {
        env.errorMsg.error(v.pos,"field " + v.field.toString() +
                           "does not exist");
        return new ExpTy(null,INT);
      };
```

Se siamo riusciti a trovare tale campo, ne restituiamo il tipo (su cui chiamiamo prima `actual`). La variabile `offset` indica a quale campo di questo record stiamo accedendo e servirà in futuro per generare del codice opportuno

```
      return new ExpTy(codegen.FieldVar(var.exp,offset,level),
                       Cursor.fieldType.actual());
    }
    else {
      env.errorMsg.error(v.pos,"record type required");
      return new ExpTy(null,INT);
    }
  }
```

4.6 Controllo dei tipi per le espressioni

Le espressioni sono essenzialmente la quasi totalità della sintassi Tiger…

Ancora un dispatcher

```
  ExpTy transExp(Absyn.Exp e) {
    if (e instanceof Absyn.VarExp) return transExp((Absyn.VarExp)e);
    if (e instanceof Absyn.NilExp) return transExp((Absyn.NilExp)e);
    if (e instanceof Absyn.IntExp) return transExp((Absyn.IntExp)e);
    if (e instanceof Absyn.StringExp) return transExp((Absyn.StringExp)e);
    if (e instanceof Absyn.CallExp) return transExp((Absyn.CallExp)e);
    if (e instanceof Absyn.OpExp) return transExp((Absyn.OpExp)e);
    if (e instanceof Absyn.RecordExp) return transExp((Absyn.RecordExp)e);
    if (e instanceof Absyn.SeqExp) return transExp((Absyn.SeqExp)e);
    if (e instanceof Absyn.AssignExp) return transExp((Absyn.AssignExp)e);
    if (e instanceof Absyn.IfExp) return transExp((Absyn.IfExp)e);
```

```
    if (e instanceof Absyn.WhileExp) return transExp((Absyn.WhileExp)e);
    if (e instanceof Absyn.ForExp) return transExp((Absyn.ForExp)e);
    if (e instanceof Absyn.BreakExp) return transExp((Absyn.BreakExp)e);
    if (e instanceof Absyn.LetExp) return transExp((Absyn.LetExp)e);
    if (e instanceof Absyn.ArrayExp) return transExp((Absyn.ArrayExp)e);

    throw new Error("transExp");
  }
```

Il controllo dei tipi per la lettura di una variabile richiama il metodo `transVar` sulla variabile e ritorna semplicemente il suo tipo

```
  ExpTy transExp(Absyn.VarExp e) {
    ExpTy et = transVar(e.var,false);

    return new ExpTy(codegen.VarExp(et.exp),et.ty);
  }
```

Il tipo delle costanti è facile da determinare

```
  ExpTy transExp(Absyn.NilExp e) {
    return new ExpTy(codegen.NilExp(),NIL);
  }

  ExpTy transExp(Absyn.IntExp e) {
    return new ExpTy(codegen.IntExp(e.value),INT);
  }

  ExpTy transExp(Absyn.StringExp e) {
    return new ExpTy(codegen.StringExp(e.value),STRING);
  }

  ExpTy transExp(Absyn.CallExp e) {
```

Di fronte a una chiamata di funzione, cominciamo col controllare che il tipo dell'identificatore di funzione sia effettivamente una funzione già definita

```
    Entry ent = (Entry)(env.venv.get(e.func));

    if (ent == null || !(ent instanceof FunEntry)) {
      env.errorMsg.error(e.pos,"unknown function identifier "
                         + e.func.toString());
      return new ExpTy(null,INT);
    }

    FunEntry f = (FunEntry)ent;
```

Confrontiamo i parametri attuali con quelli formali: devono avere lo stesso numero e deve essere possibile assegnare i parametri attuali a quelli formali

```
    Absyn.ExpList actuals = e.args;
    Types.RECORD formals = f.formals;
    ExpTy et = null;
    Tree.ExpList args = null, cursor = null;

    for (; actuals != null; actuals = actuals.tail,
         formals = formals.tail) {
```

Se i parametri formali finiscono prima di quelli attuali diamo errore

```
      if (formals == null) {
        env.errorMsg.error(e.pos,"too many arguments");
        break;
      }
```

Facciamo il controllo dei tipi per ogni parametro attuale. Poi controlliamo che il suo tipo sia assegnabile col tipo del corrispondente parametro formale (applichiamo `actual` poiché tale tipo potrebbe essere un tipo definito dall'utente e quindi, potenzialmente, un `NAME`)

```
      et = transExp(actuals.head);
      checkCompatible(formals.fieldType.actual(),et.ty,
                      actuals.head.pos);
```

Accumuliamo in `args` il codice che valuta i parametri attuali uno dopo l'altro

```
      if (args == null)
        args = cursor = new Tree.ExpList(codegen.Arg(et.exp),null);
      else {
        cursor.tail = new Tree.ExpList(codegen.Arg(et.exp),null);
        cursor = cursor.tail;
      }
    }
```

Se sono finiti i parametri attuali ma non quelli formali diamo errore

```
    if (formals != null) env.errorMsg.error(e.pos,"too few arguments");
```

Il tipo della chiamata di funzione è il tipo di ritorno dato nella dichiarazione della funzione. Il generatore di codice intermedio deve sapere se questa funzione ritorna qualcosa o meno. Inoltre forniamo sia il livello corrente `level`, che il livello `f.level` in cui la funzione è definita. Questo permette a `codegen.CallExp()` di implementare la regola antenato-fratello-figlio (Sezione 3.1)

```
    return new ExpTy(codegen.CallExp(f.label,args,f.level,level,
                                     f.result.actual() == VOID),
                     f.result.actual());
  }

  ExpTy transExp(Absyn.OpExp e) {
```

Il controllo dei tipi per una espressione del tipo e1 op e2 comincia con il controllo dei tipi per le due espressioni

```
    ExpTy et1 = transExp(e.left), et2 = transExp(e.right);
```

Quasi tutti gli operatori di confronto richiedono che le due espressioni abbiamo tipo intero, e il risultato è un'espressione intera

```
    if (e.oper != Absyn.OpExp.EQ && e.oper != Absyn.OpExp.NE) {
      checkInt(et1,e.left.pos);
      checkInt(et2,e.right.pos);
      return new ExpTy(codegen.OpExp(et1.exp,e.oper,et2.exp),INT);
    }
    else {
```

Gli operatori di uguaglianza e disuguaglianza possono essere applicati anche a valori che non sono interi. L'importante è che le due espressioni confrontate abbiano un tipo unificabile. Non ammettiamo il confronto di due `nil` o di due espressioni di tipo `VOID`, per cui passiamo `false` a `checkUnifiable`. Il risultato è comunque un intero

```
      checkUnifiable(et1,et2,e.pos,false);

      return new ExpTy(codegen.OpExp(et1.exp,e.oper,et2.exp),INT);
    }
  }
```

Un'espressione del tipo nome-record{campo1 = exp1...campon = expn}

```
  ExpTy transExp(Absyn.RecordExp e) {
```

Il nome del record deve essere un identificatore di un tipo record già definito

```
    Types.Type type = (Types.Type)env.tenv.get(e.typ);

    if (type == null) {
      env.errorMsg.error(e.pos,"undefined type " + e.typ.toString());
      return new ExpTy(null,INT);
    }

    type = type.actual();
    if (!(type instanceof Types.RECORD)) {
      env.errorMsg.error(e.pos,"record identifier required");
      return new ExpTy(null,INT);
    }
```

Controlliamo che ogni expi sia assegnabile al tipo di dichiarazione di campoi

```
    Absyn.FieldExpList a = e.fields;
    Types.RECORD f = (Types.RECORD)type;
```

Gli oggetti della classe `Translate/ExpList.java` sono liste di codici

```
    Translate.ExpList codes = null, cursor = null
    ExpTy et;

    int pos = e.pos;

    for (; a != null; a = a.tail, f = f.tail) {
      pos = a.pos;
```

I campi dell'espressione di record non possono essere di più di quelli dati nella dichiarazione del nome del record

```
      if (f == null) {
        env.errorMsg.error(a.pos,"field "+ a.name.toString() +
                           " is unknown");
        break;
      }
```

I campi dell'espressione di record devono corrispondere posizionalmente a quelli dati nella dichiarazione del nome del record

```
      if (f.fieldName != a.name) {
        env.errorMsg.error(a.pos,f.fieldName.toString() +
                           " field expected");
        break;
      }

      et = transExp(a.init);
      checkCompatible(f.fieldType.actual(),et.ty,a.pos);
```

Accumuliamo i codici in una lista

```
      if (codes == null)
        codes = cursor = new Translate.ExpList(et.exp,null);
      else {
        cursor.tail = new Translate.ExpList(et.exp,null);
        cursor = cursor.tail;
      }
    }
```

I campi dell'espressione di record non possono essere di meno di quelli dati nella dichiarazione del nome del record

```
    if (f != null) env.errorMsg.error(pos,"missing record fields");

    return new ExpTy(codegen.RecordExp(codes),type);
  }

  ExpTy transExp(Absyn.SeqExp e) {
```

Il tipo di una sequenza vuota è `VOID`

```
    Types.Type type = VOID;
    Translate.Exp code = null;
    ExpTy et;
```

Accumuliamo i codici e salviamo in `type` il tipo dell'ultima espressione

```
    for (Absyn.ExpList h = e.list; h != null; h = h.tail) {
      et = transExp(h.head);
      if (code != null)
        code = codegen.Append(code,et.exp);
      else
        code = et.exp;

      type = et.ty;
    }

    return new ExpTy((code == null)? codegen.Nop() : code,type);
  }
```

In un assegnamento bisogna controllare che il rightvalue abbia un tipo compatibile col tipo del leftvalue

```
  ExpTy transExp(Absyn.AssignExp e) {
    ExpTy vt = transVar(e.var,true);
    ExpTy et = transExp(e.exp);

    checkCompatible(vt.ty,et.ty,e.pos);

    return new ExpTy(codegen.AssignExp(vt.exp,et.exp),VOID);
  }

  ExpTy transExp(Absyn.IfExp e) {
    Types.Type temp;

    ExpTy test = transExp(e.test);
    ExpTy thenclause = transExp(e.thenclause);
```

Il ramo else può non esserci

```
    ExpTy elseclause = (e.elseclause != null) ? transExp(e.elseclause)
      : null;
```

La guardia deve avere tipo intero

```
    checkInt(test,e.test.pos);
```

Se l'`else` è presente allora i due rami dell'`if` devono avere tipi unificabili, eventualmente anche entrambi `NIL` o entrambi `VOID` (quindi passiamo `true` a `checkUnifiable`). Si noti che restituiamo il tipo più generale fra quello dei due rami

```
    if (elseclause != null)
      if ((temp = checkUnifiable(thenclause,elseclause,
                                 e.elseclause.pos,true)) != null)
        return new ExpTy(codegen.IfExp(test.exp,thenclause.exp,
                                       elseclause.exp),temp);
      else return new ExpTy(null,VOID);

    if (thenclause.ty == VOID)
      return new ExpTy(codegen.IfExp(test.exp,thenclause.exp),VOID);

    env.errorMsg.error(e.thenclause.pos,"then clause must return void");
    return new ExpTy(null,VOID);
  }

  ExpTy transExp(Absyn.WhileExp e) {
```

La guardia del ciclo deve avere tipo intero

```
    ExpTy test = transExp(e.test);
    checkInt(test,e.test.pos);
```

È possibile eseguire un `break` all'interno del ciclo `while`, per cui creiamo un nuovo oggetto della classe `Semant/Semant.java` con una label non `null` e facciamo il controllo dei tipi del corpo del `while` con tale oggetto

```
    Temp.Label b = new Temp.Label();
    Semant newsem = new Semant(env,level,codegen,b);
    ExpTy body = newsem.transExp(e.body);
```

Il tipo del corpo del `while` deve essere `VOID`

```
    if (body.ty != VOID) {
      env.errorMsg.error(e.body.pos,"while's body must return void");
      return new ExpTy(null,VOID);
    }

    return new ExpTy(codegen.WhileExp(test.exp,body.exp,b),VOID);
  }

  ExpTy transExp(Absyn.ForExp e) {
```

L'indice del ciclo è una variabile locale che deve scomparire all'uscita dal ciclo. Inoltre non deve essere possibile modificarla, per cui la dichiariamo col metodo `transDecConstant` piuttosto che con `transDec` (si veda la Sezione 4.4)

```
    env.venv.beginScope();
    Translate.Exp d = transDecConstant(e.var);
```

La variabile indice e il suo limite superiore devono essere interi

```
    VarEntry v = (VarEntry)(env.venv.get(e.var.name));
    if (v.ty != INT) env.errorMsg.error(e.var.pos,"integer required");

    ExpTy hi = transExp(e.hi);
    checkInt(hi,e.hi.pos);
```

È possibile eseguire un `break` all'interno del ciclo

```
    Temp.Label b = new Temp.Label();
    Semant newsem = new Semant(env,level,codegen,b);
    ExpTy body = newsem.transExp(e.body);
```

Il corpo del `for` deve avere tipo `VOID`

```
    if (body.ty != VOID)
      env.errorMsg.error(e.body.pos,"for's body must return void");

    env.venv.endScope();
    return new ExpTy(codegen.ForExp(d,hi.exp,body.exp,v.access,b),VOID);
  }

  ExpTy transExp(Absyn.BreakExp e) {
```

Se incontriamo un'espressione `break`, dobbiamo controllare che sia lecito eseguirla (cioè, che siamo dentro a un qualche ciclo). Questo è vero se e solo se `_break` non è `null`

```
    if (_break == null) {
      env.errorMsg.error(e.pos,"illegal break position");
      return new ExpTy(null,VOID);
    }
```

`break` ha comunque tipo `VOID`

```
    return new ExpTy(codegen.BreakExp(_break),VOID);
  }

  ExpTy transExp(Absyn.LetExp e) {
    Translate.Exp code = null;
```

Le variabili e i tipi locali devono essere dimenticati all'uscita dal `let`

```
    env.venv.beginScope(); env.tenv.beginScope();
```

Processiamo un gruppo di dichiarazioni mutualmente ricorsive alla volta e accumuliamo i codici

```
    for (Absyn.DecList p = e.decs; p != null; p = p.tail)
      if (code == null)
        code = transDec(p.head);
      else
        code = codegen.Append(code,transDec(p.head));
```

Quindi passiamo al corpo del `let`

```
    ExpTy et = transExp(e.body);

    env.venv.endScope(); env.tenv.endScope();
    return new ExpTy(codegen.LetExp(code,et.exp),et.ty);
  }
```

Un'espressione del tipo `tipo-array[num] of init`

```
  ExpTy transExp(Absyn.ArrayExp e) {
```

Il tipo dell'array deve già essere stato definito e deve effettivamente denotare un tipo ARRAY

```
    Types.Type t = (Types.Type)env.tenv.get(e.typ);
    if (t == null) {
      env.errorMsg.error(e.pos,"undefined array type");
      return new ExpTy(null,INT);
    }

    t = t.actual();
    if (!(t instanceof Types.ARRAY)) {
      env.errorMsg.error(e.pos,"incompatible type");
      return new ExpTy(null,INT);
    }
```

Il numero di elementi dell'array deve essere un intero

```
    ExpTy et1 = transExp(e.size);
    checkInt(et1,e.size.pos);
```

Il valore di inizializzazione deve essere compatibile col tipo di dichiarazione degli elementi dell'array

```
    ExpTy et2 = transExp(e.init);
    checkCompatible(((Types.ARRAY)t).element.actual(),et2.ty,e.init.pos);

    return new ExpTy(codegen.ArrayExp(et1.exp,et2.exp,level),t);
  }
```

4.7 Usiamo l'analizzatore semantico

Dal momento che il nostro analizzatore semantico richiama anche il generatore del codice (Capitolo 5), usiamo per adesso un traduttore che ritorna sempre `null` per ogni categoria di sintassi astratta.

Per richiamare l'analizzatore semantico, facciamo una piccola modifica al file `Parse/Parse.java` come segue:

`Parse/Parse_semantic.java`

```
package Parse;

public class Parse {
  public ErrorMsg.ErrorMsg errorMsg;
  public Absyn.Exp absyn;

  public Parse(String filename) {
      ...
      absyn = (Absyn.Exp)(symbol.value);
      System.out.println("Fine dell'analisi sintattica");
```

Qui controlliamo se l'albero astratto non è `null` ed eventualmente richiamiamo l'analisi semantica su di esso

```
      if (absyn != null) {
          Semant.Semant s = new Semant.Semant(errorMsg);
          s.transProg(absyn);
          System.out.println("Fine dell'analisi semantica");
      }
      else System.out.println("Attributo semantico pari a null");
  }
}
```

Il file `Parse/Main_semantic.java` è invece uguale a `Parse/Main_syntactical.java` ma li teniamo distinti per eventuali future modifiche.

Indichiamo di seguito le aggiunte che facciamo al makefile.

`makefile`

Per compilare il traduttore lo copiamo nel file `Translate/Translate.java` e poi richiamiamo il compilatore Java

```
translate_semantic: Translate/Translate_semantic.java
        @echo "*** Compiling a fake translator ***"
        cp Translate/Translate_semantic.java Translate/Translate.java
        javac ${JFLAGS} Translate/Translate.java
        touch translate_semantic
```

Questa regola ci permette di compilare l'analizzatore semantico

```
Semant/Semant.class: Semant/Semant.java
        @echo "*** Compiling the semantic analyzer ***"
        javac ${JFLAGS} Semant/Semant.java
```

Questo è il target che genera la shell per l'analizzatore semantico

```
semantic: Parse/Grm.java Parse/Main_semantic.java \
        Parse/Parse_semantic.java translate_semantic \
        Semant/Semant.class Parse/Main.class
        @echo "*** Compiling the semantic shell ***"
```

Copiamo i files specifici della shell e compiliamola

```
        cp Parse/Parse_semantic.java Parse/Parse.java
        cp Parse/Main_semantic.java Parse/Main.java
        javac ${JFLAGS} Parse/Main.java
        touch semantic
```

Aggiungiamo altre classi da cancellare

```
.PHONY: clean
clean:
        -rm lexical
        -rm syntactical
        -rm semantic
        -rm translate_semantic
        -rm Parse/*.class ErrorMsg/*.class Parse/Yylex.java
        -rm Parse/Grm.java Absyn/*.class Symbol/*.class
        -rm Semant/*.class Translate/*.class
        -rm Types/*.class Temp/*.class Translate/*.class Intel/*.class
        -rm Frame/*.class rm FindEscape/*.class
        -rm Tree/*.class
        touch Parse/Main.class
```

Per compilare e provare il nostro analizzatore semantico basta dare i seguenti comandi:

```
> make semantic
> java Parse.Main testcases/test1.tig
```

Capitolo 5

Generazione del Codice Intermedio

5.1 Modalità di compilazione

Un pezzo di codice Tiger è compilato in un oggetto della classe `Tree/Exp.java` o `Tree/Stm.java`. Ci sono però *modalità* di compilazione diverse, sulla base dell'*uso* che intendiamo fare del codice. Possiamo cioè compilarlo in una modalità `Ex`, nel caso in cui ci interessa avere un valore ritornato dal codice, oppure in modalità `Nx`, nel caso in cui tale valore non ci interessi, e infine in modalità `Cx`, nel caso in cui vogliamo saltare a due etichette specificate, sulla base per esempio del risultato di un test. Questo conduce alla seguente classe, che è un wrapper generico di codice compilato, e permette di trasformarlo in un codice adatto a una delle modalità appena viste.

`Translate/Exp.java`

```
package Translate;

public abstract class Exp {
  public abstract Tree.Exp unEx();
  public abstract Tree.Stm unNx();
  public abstract Tree.Stm unCx(Temp.Label t, Temp.Label f);
}
```

Potremmo accontentarci della sola classe `Translate/Exp.java`, ma spesso il contesto in cui un codice si trova ci fa preferire una modalità di compilazione piuttosto che un'altra. Per esempio, in `if e then s1 else s2`, la compilazione di `e` deve essere effettuata in modalità `Cx`. Potremmo compilare `e` in una modalità di default (per esempio, la `Ex`), ottenere del codice *avvolto* nel wrapper `Translate/Exp.java` e quindi applicare il metodo `unCx` con due etichette opportune. Questo però ha l'effetto di aggiungere un test di uguaglianza a 0 del valore di `e`, mentre una semplice istruzione `CJUMP` sarebbe stata sufficiente. A fini di ottimizzazione del codice generato, usiamo tre sottoclassi di `Translate/Exp.java` che assumono che il codice che esse avvolgono si comporti di per sé secondo una delle tre modalità viste sopra:

`Translate/Ex.java`

```
package Translate;

class Ex extends Exp {
```

Questo è il codice *avvolto* da questo wrapper. Come si vede, assumiamo che esso sia del codice che ritorna un valore

```
  Tree.Exp exp;

  public Ex(Tree.Exp e) { exp=e; }
```

Grazie all'ipotesi sul tipo di codice *avvolto* da questo wrapper, questa modalità di compilazione è immediata e non aggiunge codice inutile

```
  public Tree.Exp unEx() { return exp; }
```

Se il valore di ritorno non ci serve, lo scartiamo

```
  public Tree.Stm unNx() { return new Tree.EXP(exp); }
```

Se vogliamo trasformarlo in un test, aggiungiamo un controllo di uguaglianza a zero. Prima però controlliamo il caso speciale in cui l'espressione avvolta è una costante. In tal caso (abbastanza frequente) possiamo generare del codice più efficiente che salta a una delle due etichette sulla base del valore della costante, senza bisogno di controllarlo a tempo di esecuzione

```
  public Tree.Stm unCx(Temp.Label t, Temp.Label f) {
    if (exp instanceof Tree.CONST)
      return ((Tree.CONST)exp).value==0 ?
        new Tree.JUMP(f) : new Tree.JUMP(t);

    return new Tree.CJUMP(Tree.CJUMP.EQ,exp,new Tree.CONST(0),f,t);
  }
}
```

Translate/Nx.java

```
package Translate;

class Nx extends Exp {
```

Questa volta il codice non ritorna nulla, quindi è un oggetto della classe `Tree/Stm.java`

```
  Tree.Stm stm;

  public Nx(Tree.Stm s) { stm=s; }
```

Non è possibile che un codice che non ritorna un valore venga compilato in un contesto in cui serve un valore di ritorno

```
  public Tree.Exp unEx() { throw new Error("unEx"); }
  public Tree.Stm unNx() { return stm; }
```

Non è possibile che un codice che non ritorna un valore venga compilato in un contesto in cui serve effettuare un test sulla base di tale valore

```
  public Tree.Stm unCx(Temp.Label t, Temp.Label f) {
    throw new Error("unCx");
  }
}
```

Translate/Cx.java

```
package Translate;
```

Questa classe è ancora astratta! Vedremo dopo un'istanziazione

```
abstract class Cx extends Exp {
```

Per trasformare un test in un'espressione che ritorna un valore, usiamo un registro ausiliario r e compiliamo il tutto come segue:

$$\begin{array}{l} r = 1 \\ unCx(t, f) \\ f : r = 0 \\ t : \\ ritorna\ r \end{array}$$

```
  public Tree.Exp unEx() {
    Temp.Temp r = new Temp.Temp();
    Temp.Label t = new Temp.Label();
    Temp.Label f = new Temp.Label();

    return new Tree.ESEQ(
      new Tree.SEQ(new Tree.MOVE(new Tree.TEMP(r),new Tree.CONST(1)),
        new Tree.SEQ(unCx(t,f),
          new Tree.SEQ(new Tree.LABEL(f),
            new Tree.SEQ(new Tree.MOVE(new Tree.TEMP(r),new Tree.CONST(0)),
             new Tree.LABEL(t))))),
        new Tree.TEMP(r));
  }
```

Per trasformare un test in un codice che non ritorna nulla, compiliamo il codice come $unCx(t, t); t :$

```
  public Tree.Stm unNx() {
    Temp.Label t=new Temp.Label();

    return new Tree.SEQ(unCx(t,t),new Tree.LABEL(t));
  }
```

La compilazione in modalità `Cx` dipende dal tipo di test e deve essere implementata dalle sottoclassi (vedi `Translate/RelCx.java`)

```
  public abstract Tree.Stm unCx(Temp.Label t, Temp.Label f);
}
```

Un esempio di sottoclasse di `Translate/Cx.java` è la seguente, che serve a compilare i test di confronto:

`Translate/RelCx.java`

```
package Translate;

public class RelCx extends Cx {
```

Specifichiamo l'operatore di confronto e il codice che contiene la compilazione dei due operandi

```
  private int oper;
  private Exp left,right;

  public RelCx(int o,Exp l,Exp r) { oper=o; left=l; right=r; }
```

La compilazione in modalità `Cx` richiede di compilare i due operandi in modalità `Ex` (perché devono ritornare un valore) e di generare un salto condizionato alle due etichette specificate

```
  public Tree.Stm unCx(Temp.Label t, Temp.Label f) {
    return new Tree.CJUMP(oper,left.unEx(),right.unEx(),t,f);
  }
}
```

5.2 Frammenti di compilazione

La compilazione di un pezzo di codice Tiger ha come effetto la generazioni di una serie di *frammenti*. Alcuni saranno frammenti di codice intermedio, altri frammenti per le stringhe. Avremo quindi una classe astratta `Translate/Frag.java` e due sottoclassi concrete:

`Translate/Frag.java`

```
package Translate;

abstract public class Frag {
  public Frag next;

  abstract public void print();
}
```

`Translate/ProcFrag.java`

```
package Translate;

public class ProcFrag extends Frag {
```

Il codice contenuto nel frammento e la descrizione del suo frame

```
  public Tree.Stm body;
  public Frame.Frame frame;

  public ProcFrag(Tree.Stm b, Frame.Frame f, Frag n) {
    body = b; frame = f; next = n;
  }

  public void print() {
    System.out.println("code fragment (" + frame.name + "):");
```

Stampiamo il codice contenuto nel frammento

```
    (new Tree.Print(System.out)).prStm(body);
  }
}
```

`Translate/DataFrag.java`

```
package Translate;

public class DataFrag extends Frag {
```

L'etichetta del punto in cui iniziano i dati e la stringa vera e propria

```
  public Temp.Label label;
  public String data;

  public DataFrag(Temp.Label l,String d,Frag n) {
    label = l; data = d; next = n;
  }

  public void print() {
    System.out.println("data fragment (" + label + "):");
    System.out.println(data);
  }
}
```

5.3 La generazione del codice

La generazione del codice Tiger nel linguaggio intermedio viene effettuata da una classe `Translate/Translate.java` che contiene un metodo per ogni classe di semantica astratta. Per compilare le istanze di una classe astratta A, occorre avere già compilato tutte le classi astratte che figurano nella definizione di A.

Mentre effettua la compilazione, la classe `Translate/Translate.java` accumula frammenti di codice in una lista `frags`. Alla fine della compilazione verrà richiamato un metodo che stampa tutti i frammenti accumulati.

`Translate/Translate.java`

```
package Translate;

public class Translate {
```

Queste due etichette dovranno, in fase di assemblaggio, essere linkate a delle routine che stampano un errore e che allocano della memoria, rispettivamente

```
  private Temp.Label error = new Temp.Label("error");
  private Temp.Label malloc = new Temp.Label("malloc");
```

Questa è la lista di frammenti di compilazione che viene generata durante la compilazione di pezzi di programmi Tiger. È possibile leggerla tramite il metodo `getResult()` e stamparla tramite il metodo `printFrags()`

```
  private Frag frags = null;

  public Frag getResult() { return frags; }

  public void printFrags() {
    Frag cursor;

    for (cursor = frags; cursor != null; cursor = cursor.next) {
      cursor.print();
      System.out.println();
    }
  }
```

Questo metodo stampa il codice compilato `code`. Chiama il metodo di stampa opportuno sulla base del fatto che il codice ritorni un valore o no

```
  public static void print(Exp code) {
    if (code instanceof Nx)
      (new Tree.Print(System.out)).prStm(code.unNx());
    else
      (new Tree.Print(System.out)).prExp(code.unEx());
  }
}
```

5.4 La traduzione delle dichiarazioni

Le dichiarazioni hanno un significato quasi esclusivamente per il controllo dei tipi effettuato nel Capitolo 4. Esse quindi generano del codice che non effettua alcuna operazione, tranne per la dichiarazione di variabile che genera del codice per inizializzare la variabile.

Aggiungiamo quindi i seguenti metodi a `Translate/Translate.java`.

Un metodo che restituisce del codice che non fa e non ritorna nulla

```
public Exp Nop() {
  return new Nx(new Tree.EXP(new Tree.CONST(0)));
}
```

Una dichiarazione di funzione genera un codice vuoto. Però deve registrare il codice della funzione fra i frammenti di codice che saranno il risultato della compilazione. Questo è effettuato dal metodo `procEntryExit()`

```
public Exp FunctionDec(Temp.Label name, Exp body, Level l) {
  procEntryExit(l,body);

  return Nop();
}
```

Questo metodo registra il codice di una funzione fra i frammenti generati dal processo di compilazione. Prima di questa registrazione, si preoccupa di chiamare un metodo specifico dell'architettura su cui si compila, che tipicamente espande il codice in modo da copiare il risultato del corpo della funzione su un registro usato come valore di ritorno (si veda `Intel/IntelFrame.java`)

```
public void procEntryExit(Level level, Exp body) {
  Tree.Stm wrappedBody;

  if (body instanceof Ex)
    wrappedBody = level.frame.procEntryExit1(body.unEx());
  else
    wrappedBody = level.frame.procEntryExit1(body.unNx());

  frags = new ProcFrag(wrappedBody,level.frame,frags);
}
```

La dichiarazione di una variabile si compila in del codice che copia il valore di inizializzazione per la variabile in una cella di memoria ottenuta a partire dal frame pointer aggiungendo lo spostamento a cui si trova la variabile (si veda `Frame/Access.java`)

```
public Exp VarDec(Access v, Exp init, Level l) {
  return new Nx(new Tree.MOVE(v.acc.exp(new Tree.TEMP(l.frame.FP())),
                              init.unEx()));
}
```

La dichiarazione di un tipo non ha effetti a livello di compilazione

```
public Exp TypeDec() { return Nop(); }
```

Vediamo alcuni esempi di compilazione. La dichiarazione

```
var a:= 13
```

viene compilata in

```
MOVE(
 MEM(
  BINOP(PLUS,TEMP t0,CONST -4)),
 CONST 13)
```

Si noti che `t0` è il registro usato per contenere il frame pointer del programma, e spostandosi in basso di 4 byte si punta alla prima variabile locale del record di attivazione, cioè `a`. Lo stesso codice viene generato per la dichiarazione

```
var a:int := 13
```

Invece la dichiarazione

```
function a():int = 13
```

è compilata in un codice vuoto:

```
EXP(CONST 0)
```

ma ha il side-effect di aggiungere un `Translate/ProcFrag.java` ai frammenti generati per il programma, il quale contiene il codice

```
MOVE(TEMP t3,CONST 13)
```

Si noti che `t3` è il registro usato per ritornare il risultato della funzione, e che questa istruzione `MOVE` è stata generata dal metodo `procEntryExit1()` di `Intel/IntelFrame.java`.

5.5 La traduzione dei leftvalue

I leftvalue sono i contenitori dell'informazione che può essere letta e scritta durante l'esecuzione del programma. È quindi naturale che questo sia il caso in cui si usa più pesantemente l'accesso allo stack di attivazione.

Aggiungiamo i seguenti metodi a `Translate/Translate.java`.

Il parametro `a` fornisce le coordinate di accesso della variabile a cui vogliamo fare accesso. Il parametro `l` è invece il livello che descrive il record di attivazione della funzione a cui appartiene il codice che stiamo generando. Per accedere alla variabile indicata da `a`, dobbiamo generare del codice che torna indietro sulla catena statica di tanti passi quanta è la differenza fra il livello `l` e il livello in cui è definita la variabile. Poi si aggiunge lo spostamente all'interno del record di attivazione così trovato

```
public Exp SimpleVar(Access a, Level l) {
  Level d = a.home;

  Tree.Exp base = new Tree.TEMP(l.frame.FP());
```

Il primo dei parametri formali del record di attivazione contiene il puntatore di catena statica. Generiamo del codice che legge il suo valore (si veda `Frame/Access.java`)

```
  for (; l != d; l = l.parent) base = (l.frame.formals.head).exp(base);
```

Alla fine generiamo del codice che aggiunge lo spostamento all'interno del record di attivazione in cui è contenuta la variabile

```
  return new Ex(a.acc.exp(base));
}
```

Se vogliamo accedere al campo di un record `e.f`, dobbiamo fornire il codice `record` che permette di accedere al valore di `e`. Questo codice deve essere compilato in modalità `Ex` poiché deve restituirci un puntatore in memoria a cui sommiamo lo spostamento `offset` che ci permette di raggiungere il campo `f`. Se supponiamo che tutti i campi occupino lo stesso spazio di memoria, questo significa sommare `offset`, moltiplicato per la dimensione di una parola, al valore di `e`

```
public Exp FieldVar(Exp record, int offset, Level l) {
  offset *= l.frame.wordSize();

  return new Ex(new Tree.MEM(new Tree.BINOP(Tree.BINOP.PLUS,
    record.unEx(),new Tree.CONST(offset))));
}
```

Se vogliamo accedere a un elemento di un array, dobbiamo compilare in modalità `Ex` l'espressione che ritorna l'array e l'indice a cui accedere. Quindi si somma alla base dell'array il valore dell'indice moltiplicato per la dimensione di una parola. Si noti che mentre nel caso dell'accesso a un campo questa moltiplicazione veniva fatta a tempo di compilazione, qui la facciamo a tempo di esecuzione, dal momento che il compilatore non conosce il valore dell'indice dell'array a cui si fa accesso

```
public Exp SubscriptVar(Exp array, Exp index, Level l) {
  return new Ex(new Tree.MEM(new Tree.BINOP(Tree.BINOP.PLUS,
    new Tree.MEM(array.unEx()),new Tree.BINOP(Tree.BINOP.MUL,
    index.unEx(),new Tree.CONST(l.frame.wordSize())))));
}
```

Vediamo alcuni esempi di compilazione di leftvalue. Si consideri il programma

```
let
  function a(n:int):int =
  let
    function b(m:int):int = m + n
  in
    2
  end
in
  1
end
```

L'espressione `m + n` viene compilata in

```
BINOP(PLUS,
  MEM(
   BINOP(PLUS,TEMP t4,CONST 4)),
  MEM(
   BINOP(PLUS,
    MEM(BINOP(PLUS,TEMP t4,CONST 0)),
    CONST 4)))
```

L'espressione `BINOP(PLUS,TEMP t4,CONST 4)` è il primo parametro della funzione `b`, cioè `m`, mentre l'espressione `BINOP(PLUS,TEMP t4,CONST 0)` è il puntatore di catena statica di tale funzione. Come si vede, il codice dice di leggere il contenuto di tale puntatore e sommare 4 al risultato, in modo da accedere al primo parametro della funzione `a`, cioè `n`. I valori di `m` e di `n` vengono infine sommati. Si noti che l'espressione `BINOP(PLUS,TEMP t4,CONST 0)` potrebbe essere semplificata perché sommare 0 non serve.

Si consideri adesso il programma

```
let
  type list = {head: int, tail:list}
  function a(l:list):int = l.tail.head
in
  1
end
```

L'espressione `l.tail.head` viene compilata in

```
MEM(
  BINOP(PLUS,

   MEM(
```

```
    BINOP(PLUS,
     MEM(BINOP(PLUS,TEMP t2,CONST 4)),
     CONST 4)),

   CONST 0))
```

L'espressione `BINOP(PLUS,TEMP t2,CONST 4)` punta al parametro `l`, per cui con l'espressione `MEM(BINOP(PLUS,TEMP t2,CONST 4))` accediamo al valore di `l`. Aggiungendo 4 puntiamo al campo `tail` di `l`. Quindi leggiamo il valore della parola di memoria così individuata, il che ci dà il valore di `l.tail`. Sommando 0 (operazione superflua) otteniamo un puntatore a `l.tail.head`, e leggendo la parola di memoria puntata otteniamo il valore di `l.tail.head`.

Si consideri infine il programma

```
let
  type t = array of int
  function a(n:t, i:int):int = n[i]
in
  1
end
```

L'espressione `n[i]` viene compilata in

```
MEM(
  BINOP(PLUS,
   MEM(
    MEM(
     BINOP(PLUS,TEMP t2,CONST 4))),
   BINOP(MUL,
    MEM(
     BINOP(PLUS,TEMP t2,CONST 8)),
    CONST 4)))
```

L'espressione `BINOP(PLUS,TEMP t2,CONST 4)` ci permette di accedere al valore del parametro `n`. L'espressione `BINOP(PLUS,TEMP t2,CONST 8)` ci permette invece di accedere al parametro `i`. Come si può vedere, il codice dice di eseguire la moltiplicazione per 4 del valore di `i` (4 è la dimensione di una parola), e di sommare il risultato al valore di `n`. Quindi di accedere al contenuto della cella di memoria così individuata.

5.6 La traduzione delle espressioni

Le espressioni formano la larga maggioranza del codice Tiger. La loro compilazione può ritornare sia del codice compilato in modalità `Ex` che del codice compilato in modalità `Nx`, dal momento che alcune espressioni (come i cicli `while`) non ritornano alcun valore.

Aggiungiamo i seguenti metodi alla classe `Translate/Translate.java`.

Una espressione può essere un leftvalue. In tal caso la compilazione del leftvalue è già la compilazione dell'espressione

```
public Exp VarExp(Exp lvalue) { return lvalue; }
```

La costante `nil` è compilata in una espressione intera che ritorna 0

```
public Exp NilExp() { return new Ex(new Tree.CONST(0)); }
```

Una costante intera è compilata in una espressione che ritorna il valore della costante

```
public Exp IntExp(int value) { return new Ex(new Tree.CONST(value)); }
```

Una stringa viene compilata in una etichetta l. Però aggiungiamo un frammento di compilazione che contiene la dichiarazione di una costante stringa assembler in una posizione etichettata con l

```
public Exp StringExp(String value, Level level) {
  Temp.Label l = new Temp.Label();

  frags = new DataFrag(l,level.frame.string(l,value),frags);

  return new Ex(new Tree.NAME(l));
}
```

La compilazione di una chiamata a una funzione di nome `name`, la compilazione dei cui parametri è fornita in `pars`, chiamata dal livello `caller` e definita nel livello `callee`. Il flag `noResult` indica se la funzione ritorna qualcosa o `void`

```
public Exp CallExp(Temp.Label name, ExpList pars,
                   Level callee, Level caller, boolean noResult){
  Tree.Exp FP;
  Tree.ExpList newPars, cursor;
```

Qui implementiamo la regola del figlio-fratello-antenato. Prima però consideriamo il caso speciale in cui la funzione chiamata non ha alcun livello associato. Questo significa che è una funzione predefinita, che non è interna a nessuno scope. In tal caso, il frame pointer è `null`, cioè la costante 0

```
  if (callee == null) FP = new Tree.CONST(0);
  else {
```

Altrimenti generiamo del codice che preleva il puntatore al frame pointer del chiamante. Questo sarà il codice che ci permette di calcolare il puntatore di catena statica del chiamato nel caso più semplice in cui il chiamato è un figlio del chiamante

```
    FP = new Tree.TEMP(caller.frame.FP());

    if (caller != callee.parent)
```

Altrimenti il chiamante e il chiamato potrebbero essere fratelli. In tal caso il puntatore di catena statica del chiamato si ottiene partendo dal frame pointer del chiamante e leggendo il puntatore di catena statica ivi trovato

```
      if (caller.parent == callee.parent)
        FP = (caller.frame.formals.head).exp(FP);
      else {
```

Infine il caso della chiamata a un antenato. Partiamo dal frame pointer del chiamante e generiamo del codice che torna indietro sulla catena statica fino a raggiungere il livello del chiamato. Quindi ritorna il frame pointer ivi trovato

```
        for (; caller != callee; caller = caller.parent)
          FP = (caller.frame.formals.head).exp(FP);

        FP = (caller.frame.formals.head).exp(FP);
      }
  }
```

I parametri sono una `Translate/ExpList.java`, ovvero una lista di `Translate/Exp.java`. Li trasformiamo in una istanza di `Tree/ExpList.java`, ovvero in una lista di `Tree/Exp.java`. Richiediamo che ogni parametri sia compilato in modalità `Ex`, perché abbiamo bisogno del suo valore. Si noti che aggiungiamo il codice che calcola il puntatore di catena statica come un parametro aggiuntivo della funzione

```
  newPars = cursor = new Tree.ExpList(FP,null);
  while (pars != null) {
    cursor.tail = new Tree.ExpList(pars.head.unEx(),null);
    cursor = cursor.tail;
    pars = pars.tail;
  }
```

Se la funzione non ha alcun valore di ritorno generiamo del codice in modalità `Nx`, altrimenti del codice in modalità `Ex`

```
  if (noResult)
    return new Nx(new Tree.EXP(
      new Tree.CALL(new Tree.NAME(name),newPars)));
  else
    return new Ex(new Tree.CALL(new Tree.NAME(name),newPars));
}
```

Si consideri per esempio il seguente programma Tiger:

```
let
  var s:string := "ciao"
  function a(p:string):string = p
  function b():string = a(s)
in
  b()
end
```

La dichiarazione di `s` viene compilata in

```
MOVE(
 MEM(BINOP(PLUS,TEMP t0,CONST -4)),
  NAME L11)
```

ovvero in del codice che copia nella prima variabile locale l'indirizzo `L11`, che è l'etichetta di una zona di memoria in cui è stata allocata la stringa `ciao`. Infatti la compilazione di tale stringa ha l'effetto di creare il seguente frammento dati:

```
data fragment (L11):
L11: .ascii "ciao"
```

La funzione `b` viene invece compilata in

```
MOVE(
 TEMP t5,
 CALL(
  NAME a_12,
   MEM(BINOP(PLUS,TEMP t4,CONST 0)),
   MEM(
    BINOP(PLUS,
     MEM(BINOP(PLUS,TEMP t4,CONST 0)),
     CONST -4))))
```

ovvero in del codice che copia nel registro di ritorno `t5` il risultato della chiamata ad `a` (`a_12` è l'etichetta usata per il codice di `a`), passando come parametri il puntatore di catena statica del chiamante (perché `a` e `b` sono fratelli) e la prima variabile locale del livello in cui `b` è definita (cioè del corpo del `let`).

Si consideri invece il programma Tiger

```
let
  var s:string := "ciao"
  function a(p:string):string =
  let
    function b():string = a(p)
  in
    b()
  end
in
  a(s)
end
```

La funzione b viene compilata in

```
MOVE(
 TEMP t5,
 CALL(
  NAME a_12,
   MEM(
    BINOP(PLUS,
     MEM(BINOP(PLUS,TEMP t4,CONST 0)),
     CONST 0)),
   MEM(
    BINOP(PLUS,
     MEM(BINOP(PLUS,TEMP t4,CONST 0)),
     CONST 4))))
```

Si noti la differenza col caso precedente. In primo luogo, il primo parametro che è passato ad a, cioè il puntatore di catena statica, è ottenuto tornando indietro di un passo sulla catena statica e quindi leggendo il valore ivi trovato. In secondo luogo, il secondo parametro, cioè p, è ottenuto tornando indietro di un passo sulla catena statica e aggiungendo 4 piuttosto che sottraendo 4, dal momento che qui p è un parametro di a, mentre nell'esempio precedente s è una variabile locale del corpo principale del programma.

Continuiamo ad esaminare i metodi che aggiungiamo a `Translate/Translate.java`.

Un'operazione binaria viene compilata di default in modalità Ex o Cx

```
public Exp OpExp(Exp left, int oper, Exp right) {
  Tree.Exp l = left.unEx(); Tree.Exp r = right.unEx();
  Exp lt = new Ex(l); Exp rt = new Ex(r);

  switch (oper) {
  case Absyn.OpExp.PLUS:
    return new Ex(new Tree.BINOP(Tree.BINOP.PLUS,l,r));
  case Absyn.OpExp.MINUS:
    return new Ex(new Tree.BINOP(Tree.BINOP.MINUS,l,r));
  case Absyn.OpExp.MUL:
    return new Ex(new Tree.BINOP(Tree.BINOP.MUL,l,r));
  case Absyn.OpExp.DIV:
    return new Ex(new Tree.BINOP(Tree.BINOP.DIV,l,r));
  case Absyn.OpExp.EQ: return new RelCx(Tree.CJUMP.EQ,lt,rt);
  case Absyn.OpExp.NE: return new RelCx(Tree.CJUMP.NE,lt,rt);
  case Absyn.OpExp.LT: return new RelCx(Tree.CJUMP.LT,lt,rt);
  case Absyn.OpExp.LE: return new RelCx(Tree.CJUMP.LE,lt,rt);
  case Absyn.OpExp.GT: return new RelCx(Tree.CJUMP.GT,lt,rt);
  case Absyn.OpExp.GE: return new RelCx(Tree.CJUMP.GE,lt,rt);
  default: throw new Error("Translate.OpExp");
```

```
  }
}
```

La creazione di un record viene lasciata agli studenti!

```
public Exp RecordExp(ExpList fields) { return new Ex(new Tree.CONST(1)); }
```

Un assegnamento viene compilato in una istruzione `MOVE` fra il codice per il lato sinistro e quello per il lato destro

```
public Exp AssignExp(Exp lvalue, Exp rvalue) {
  return new Nx(new Tree.MOVE(lvalue.unEx(),rvalue.unEx()));
}
```

Distinguiamo fra il condizionale semplice e quello a due uscite. Nel primo caso creiamo due etichette, compiliamo il test in modalità `Cx` fornendo le due etichette come punti di salto e compiliamo il corpo in modalità `Nx` poiché il condizionale semplice in Tiger non ritorna mai un valore

```
public Exp IfExp(Exp test, Exp thenclause) {
  Temp.Label t = new Temp.Label(); Temp.Label f = new Temp.Label();

  return new Nx(new Tree.SEQ(   test.unCx(t,f),
                new Tree.SEQ(   new Tree.LABEL(t),
                new Tree.SEQ(   thenclause.unNx(),
                                new Tree.LABEL(f)   ))));
}
```

Nel caso del condizionale a due uscite, dobbiamo distinguere il caso in cui i due rami ritornano entrambi `void` (e quindi vanno compilati in modalità `Nx` e il risultato è in modalità `Nx`) da quello in cui invece ritornano qualcosa (e quindi vanno compilati in modalità `Ex` e il risultato è in modalità `Ex`)

```
public Exp IfExp(Exp test, Exp thenclause, Exp elseclause) {
  Temp.Label t = new Temp.Label();
  Temp.Label f = new Temp.Label();
  Temp.Label end = new Temp.Label();

  if (thenclause instanceof Nx)
    return new Nx(new Tree.SEQ(    test.unCx(t,f),
                  new Tree.SEQ(    new Tree.LABEL(t),
                  new Tree.SEQ(    thenclause.unNx(),
                  new Tree.SEQ(    new Tree.JUMP(end),
                  new Tree.SEQ(    new Tree.LABEL(f),
                  new Tree.SEQ(    elseclause.unNx(),
                                   new Tree.LABEL(end)   )))))));
  else {
    Tree.TEMP r = new Tree.TEMP(new Temp.Temp());

    return new Ex(new Tree.ESEQ(
                  new Tree.SEQ(  test.unCx(t,f),
                  new Tree.SEQ(  new Tree.LABEL(t),
                  new Tree.SEQ(  new Tree.MOVE(r,thenclause.unEx()),
                  new Tree.SEQ(  new Tree.JUMP(end),
                  new Tree.SEQ(  new Tree.LABEL(f),
                  new Tree.SEQ(  new Tree.MOVE(r,elseclause.unEx()),
                                 new Tree.LABEL(end)       ))))),r));
  }
}
```

La compilazione del ciclo `while` ci dà modo di comprendere l'uso dell'etichetta `_break` gestita dalla classe `Semant/Semant.java`. Essa è l'etichetta a cui saltare in caso si esegua un'istruzione `break`. Compiliamo il ciclo `while` in maniera tale da etichettare l'istruzione successiva al ciclo con l'etichetta `_break`. Quindi l'istruzione `break`, che viene compilata in un salto incondizionato all'etichetta `_break`, ci permetterà di uscire dal ciclo (si veda il metodo successivo)

```
public Exp WhileExp(Exp test, Exp body, Temp.Label _break) {
  Temp.Label start = new Temp.Label();
  Temp.Label t = new Temp.Label();

  return new Nx(new Tree.SEQ(    new Tree.LABEL(start),
                new Tree.SEQ(    test.unCx(t,_break),
                new Tree.SEQ(    new Tree.LABEL(t),
                new Tree.SEQ(    body.unNx(),
                new Tree.SEQ(    new Tree.JUMP(start),
                                 new Tree.LABEL(_break)))))));
}
```

Un'istruzione `break` viene compilata in un salto incondizionato all'etichetta `_break`

```
public Exp BreakExp(Temp.Label _break) {
  return new Nx(new Tree.JUMP(_break));
}
```

La compilazione del ciclo `for` deve essere scritta dagli studenti!

```
public Exp ForExp(Exp var, Exp hi, Exp body,
                  Access v, Temp.Label _break) { return Nop(); }
```

La compilazione di un'espressione `let` risulta nella compilazione delle dichiarazioni seguita dalla compilazione del corpo

```
public Exp LetExp(Exp decs, Exp body) { return Append(decs,body); }
```

La compilazione di due espressioni in sequenza consiste nella compilazione della prima in modalità `Nx` (il suo valore, ammesso che esista, viene infatti scartato) e della seconda in modalità `Ex` o `Nx` sulla base del fatto che essa ritorni qualcosa o meno

```
public Exp Append(Exp head, Exp tail) {
  if (tail instanceof Nx)
    return new Nx(new Tree.SEQ(head.unNx(),tail.unNx()));
  else
    return new Ex(new Tree.ESEQ(head.unNx(),tail.unEx()));
}
```

La compilazione della creazione di un array

```
public Exp ArrayExp(Exp size, Exp init, Level l) {
  Temp.Temp s = new Temp.Temp(); Temp.Temp i = new Temp.Temp();
  Temp.Temp b = new Temp.Temp(); Temp.Temp bb = new Temp.Temp();
  int k = l.frame.wordSize();

  Temp.Label l1 = new Temp.Label(); Temp.Label l2 = new Temp.Label();
  Temp.Label l3 = new Temp.Label(); Temp.Label l4 = new Temp.Label();
  Temp.Label l5 = new Temp.Label();

  return new Ex(new Tree.ESEQ
```

Salviamo in due temporanei la dimensione dell'array e il suo valore di inizializzazione

```
    (new Tree.SEQ(  new Tree.MOVE(new Tree.TEMP(s),size.unEx()),
     new Tree.SEQ(  new Tree.MOVE(new Tree.TEMP(i),init.unEx()),
```

Se la dimensione è negativa diamo errore

```
     new Tree.SEQ(  new Tree.CJUMP(Tree.CJUMP.LT,new Tree.TEMP(s),
                                   new Tree.CONST(0),error,l1),
     new Tree.SEQ(  new Tree.LABEL(l1),
```

Altrimenti allochiamo `size` parole di memoria e poniamo nel temporaneo `b` l'indirizzo a partire dal quale tali parole sono state allocate. Se `b` è 0 vuol dire che l'allocazione è fallita e dobbiamo dare errore

```
     new Tree.SEQ(  new Tree.MOVE(new Tree.TEMP(b),new Tree.CALL(
                                  new Tree.NAME(malloc),
                                 new Tree.ExpList(new Tree.TEMP(s),null))),
     new Tree.SEQ(  new Tree.CJUMP(Tree.CJUMP.EQ,new Tree.TEMP(b),
                                   new Tree.CONST(0),error,l2),
     new Tree.SEQ(  new Tree.LABEL(l2),
```

`b` sarà il risultato dell'espressione. Per non modificarlo, lo copiamo in un registro `bb` che verrà incrementato a ogni iterazione della dimensione di una parola

```
     new Tree.SEQ(  new Tree.MOVE(new Tree.TEMP(bb),new Tree.TEMP(b)),
     new Tree.SEQ(  new Tree.LABEL(l3),
```

Eseguiamo `size` iterazioni

```
     new Tree.SEQ(  new Tree.CJUMP(Tree.CJUMP.LE,new Tree.TEMP(s),
                                   new Tree.CONST(0),l5,l4),
     new Tree.SEQ(  new Tree.LABEL(l4),
```

Per ogni iterazione copiamo il valore di inizializzazione nella cella di memoria puntata da `bb`, quindi incrementiamo `bb` e decrementiamo `s`

```
     new Tree.SEQ(  new Tree.MOVE(new Tree.MEM(new Tree.TEMP(bb)),
                                  new Tree.TEMP(i)),
     new Tree.SEQ(  new Tree.MOVE(new Tree.TEMP(bb),
              new Tree.BINOP(Tree.BINOP.PLUS,new Tree.TEMP(bb),
                             new Tree.CONST(k))),
     new Tree.SEQ(  new Tree.MOVE(new Tree.TEMP(s),
              new Tree.BINOP(Tree.BINOP.MINUS,new Tree.TEMP(s),
                             new Tree.CONST(1))),
     new Tree.SEQ(  new Tree.JUMP(l3),
                    new Tree.LABEL(l5)  ))))))))))))))),
     new Tree.TEMP(b)));
}
```

Si consideri per esempio il seguente programma Tiger:

```
let
  var i:int := 0
in
  while (i < 10) do i := i + 1
end
```

Esso viene compilato in

```
SEQ(
 MOVE(
  MEM(BINOP(PLUS,TEMP t0,CONST -4)),
  CONST 0),
 SEQ(
  LABEL L12,
  SEQ(
   CJUMP(LT,
    MEM(BINOP(PLUS,TEMP t0,CONST -4)),
    CONST 10,
    L13,L11),
   SEQ(
    LABEL L13,
    SEQ(
     MOVE(
      MEM(BINOP(PLUS,TEMP t0,CONST -4)),
      BINOP(PLUS,
       MEM(BINOP(PLUS,TEMP t0,CONST -4)),
       CONST 1)),
     SEQ(
      JUMP(NAME L12),
      LABEL L11))))))
```

L'espressione `MEM(BINOP(PLUS,TEMP t0,CONST -4))` restituisce il valore della variabile `i`. Inizialmente le si assegna 0. Poi si entra in un ciclo (che inizia alla etichetta `L12`) il quale controlla se `i` è minore di 10. In tal caso esegue il corpo del ciclo, altrimenti esce. Il corpo del ciclo incrementa `i` e salta all'inizio del ciclo.

5.7 Usiamo il generatore di codice intermedio

Il generatore di codice viene chiamato automaticamente da `Semant/Semant.java`. Dobbiamo solo preoccuparci di stampare il risultato della compilazione, cioè il corpo principale del programma e i frammenti di compilazione. Riscriviamo quindi la classe `Parse/Parse.java` come segue:

`Parse/Parse_translate.java`

```
package Parse;

public class Parse {
  public ErrorMsg.ErrorMsg errorMsg;
  public Absyn.Exp absyn;

  public Parse(String filename) {
    ...
    absyn = (Absyn.Exp)(symbol.value);
    System.out.println("Fine dell'analisi sintattica");

    if (absyn != null) {
      Semant.Semant s = new Semant.Semant(errorMsg);
```

Adesso il risultato della compilazione ci serve (si confronti con la sezione 4.7)

```
      Translate.Exp code = s.transProg(absyn);
      System.out.println("Fine dell'analisi semantica");
```

Se non ci sono stati errori, stampiamo il codice del corpo principale del programma e tutti i frammenti generati durante la compilazione

```
      if (!errorMsg.anyErrors) {
        System.out.println("\nmain:");
        s.codegen.print(code);
        System.out.println();
        s.codegen.printFrags();
        System.out.println("Fine della generazione del codice");
      }
    }
    else System.out.println("Attributo semantico pari a null");
  }
}
```

Il file `Parse/Main_translate.java` è sempre uguale a `Parse/Main_syntactical.java`, ma li teniamo distinti per eventuali future modifiche.

Aggiungiamo al makefile un nuovo target che compila il generatore di codice intermedio.

`makefile`

Per compilare il traduttore, lo copiamo come `Translate/Translate.java` e richiamiamo il compilatore Java. Il fatto di non averlo chiamato direttamente con tale nome ci permette di usare traduttori diversi per target diversi del makefile. Si consideri per esempio il caso del traduttore vuoto utilizzato per far funzionare la sola analisi semantica (Sezione 4.7)

```
translate_translate: Translate/Translate_translate.java \
        Translate/Translate.java
        @echo "*** Compiling the translator ***"
        cp Translate/Translate_translate.java Translate/Translate.java
        javac ${JFLAGS} Translate/Translate.java
        touch translate_translate
```

Questo è il target che genera la shell per il generatore di codice

```
translate: Parse/Grm.java Parse/Main_translate.java \
        Parse/Parse_translate.java translate_translate \
        Semant/Semant.class Parse/Main.class
```

Copiamo i files specifici della shell e compiliamola

```
        @echo "*** Compiling the translate shell ***"
        cp Parse/Parse_translate.java Parse/Parse.java
        cp Parse/Main_translate.java Parse/Main.java
        javac ${JFLAGS} Parse/Main.java
        touch translate
```

Aggiorniamo l'elenco delle cose da cancellare ad ogni `make clean`

```
.PHONY: clean
clean:
        -rm lexical
        -rm syntactical
        -rm semantic
        -rm translate_semantic
        -rm translate_translate
        -rm Parse/*.class ErrorMsg/*.class Parse/Yylex.java
        -rm Parse/Grm.java
```

```
        -rm Absyn/*.class Symbol/*.class
        -rm Semant/*.class Translate/*.class
        -rm Types/*.class Temp/*.class Translate/*.class
        -rm Intel/*.class
        -rm Frame/*.class rm FindEscape/*.class
        -rm Tree/*.class
        touch Parse/Main.class
```

Per compilare e provare il nostro generatore di codice intermedio basta dare i seguenti comandi:

```
> make translate

> java Parse.Main testcases/test1.tig
```

Indice analitico