

La generazione del codice intermedio per Kitten

Fausto Spoto

9 marzo 2005

Generazione del codice dalla sintassi astratta

Aggiungiamo alle classi di sintassi astratta dei **metodi di generazione del codice intermedio**

- Nella classe `Absyn/Expression.java`:

```
protected Bytecode translate()
```

che genera il codice che valuta l'espressione

- Tale metodo è ereditato dalla classe `Absyn/Lvalue.java` che, **in più**, ha anche il metodo:

```
public Bytecode translateForWriting  
    (Expression rvalue)
```

che genera il codice che valuta `rvalue` e ne assegna il valore al `leftvalue`

La generazione del codice per le espressioni

Il principio delle montagne russe

Il codice generato dal metodo `translate` delle espressioni deve comportarsi come segue:

- 1 Lascia immutati gli elementi che trova sullo stack
- 2 Carica in cima allo stack il valore v dell'espressione, del tipo statico dell'espressione



- 3 Non modifica le variabili locali

La generazione del codice per le espressioni

- Il codice generato dal metodo `translate` lascia sullo stack un valore del **tipo statico** dell'espressione
- Ma questo non è necessariamente il tipo che serviva
- Dipende dal **contesto** in cui l'espressione è usata

Esempio: tipo statico `int`, ma serviva un `float`

`3.2 + 5`

Esempio: tipo statico `char`, ma serviva un `int`

`f('a')` se `f` si aspetta un `int`

Esempio: tipo statico `float`, ma serviva un `int`

`3.5 as int`

La generazione del codice per le espressioni

Il metodo `translate` è quindi `protected`. La vera interfaccia per la generazione del codice delle espressioni è fatta dai seguenti metodi pubblici di `Absyn/Expression.java`:

```
public Bytecode translateAsBoolean()  
public Bytecode translateAsCharacter()  
public Bytecode translateAsInteger()  
public Bytecode translateAsFloat()  
public Bytecode translateAsReference()  
public Bytecode translateAs(Types.Type type)
```

La generazione del codice per le espressioni

Le espressioni booleane possono solo essere usate in un contesto in cui ci si aspetta un booleano

```
public Bytecode translateAsBoolean() {  
    return translate(); }  
}
```

Quelle intere in tutti i contesti *numerici*

```
public Bytecode translateAsInteger() {  
    if (staticType == TypeChecker.CHAR_TYPE)  
        // promozione di tipo  
        return translate().append(new C2I(null));  
    else if (staticType == TypeChecker.FLOAT_TYPE)  
        // conversione di tipo  
        return translate().append(new F2I(null));  
    // corrispondenza di tipo  
    else return translate(); }  
}
```

La generazione del codice per le costanti

Dentro `Absyn/Integer.java`

```
protected Bytecode translate() {  
    return new ICONST(value,null); }
```

Dentro `Absyn/Floating.java`

```
protected Bytecode translate() {  
    return new FCONST(value,null); }
```

Dentro `Absyn/True.java`

```
protected Bytecode translate() {  
    return new BCONST(true,null); }
```

Dentro `Absyn/Nil.java`

```
protected Bytecode translate() {  
    return new CONST_NIL(null); }
```

Si noti che il principio delle montagne russe è rispettato!

La generazione del codice per le espressioni binarie

Dentro `Absyn/And.java`

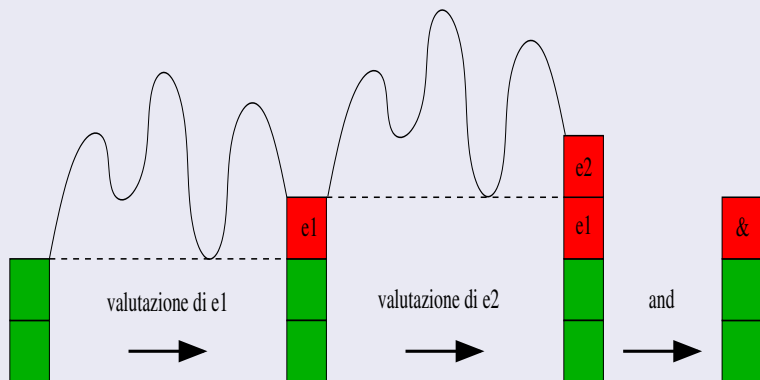
```
protected Bytecode translate() {  
    Bytecode leftBytecode, rightBytecode;  
    leftBytecode = left.translateAsBoolean();  
    rightBytecode = right.translateAsBoolean();  
    return leftBytecode.append(rightBytecode)  
        .append(new AND(null)); }  
}
```

Compilazione di `true & false`

```
bconst true } true.translateAsBoolean()  
bconst false } false.translateAsBoolean()  
and
```

Importanza del principio delle montagne russe

L'esecuzione di e_1 & e_2



Se non fosse rispettato, nessuno ci garantirebbe che il valore di e_1 sia ancora nello stack alla fine della valutazione di e_2

La generazione del codice per le espressioni binarie

Dentro `Absyn/Addition.java`

```
protected Bytecode translate() {  
    Bytecode leftBytecode, rightBytecode;  
    leftBytecode = left.translateAs(staticType);  
    rightBytecode = right.translateAs(staticType);  
    return leftBytecode  
        .append(rightBytecode)  
        .append(new ADD(staticType, null)); }  
}
```

Compilazione di `3 + 5.0` (tipo static `float`)

```
iconst 3  
i2f      } 3.translateAs(float)  
fconst 5.0 } 5.0.translateAs(float)  
add float
```

La generazione del codice per le espressioni binarie

Dentro `Absyn/Equal.java`

```
protected Bytecode translate() {
    Types.Type type = left.staticType
        .leastCommonSupertype(right.staticType);
    Bytecode leftBytecode, rightBytecode;
    leftBytecode = left.translateAs(type);
    rightBytecode = right.translateAs(type);
    return leftBytecode
        .append(rightBytecode)
        .append(new EQ(type, null)); }
```

Compilazione di `3 = 3.0`

```
iconst 3 } 3.translateAs(float)
i2f      }
fconst 5.0 } 5.0.translateAs(float)
eq float
```

La generazione del codice per i leftvalue

Abbiamo già visto che i leftvalue hanno sia il metodo `translate` che il metodo `translateForWriting`

Dentro `Absyn/Variable.java`

```
protected Bytecode translate() {  
    return new LOAD(varNum, staticType, null);  
}
```

`varNum`?

- È il **numero d'ordine** della variabile fra tutte quelle che sono dichiarate dentro il metodo in cui occorre
- È calcolato durante il type-checking
 - quando si dichiara una variabile, la si inserisce nella tabella delle variabili (`putVar`) con un nuovo numero d'ordine
 - quando la si usa, il suo numero d'ordine viene prelevato dalla tabella (`getVar`) e annotato nella sintassi astratta, in modo da poterlo usare quando generiamo il codice

La generazione del codice per i leftvalue

Dentro `Absyn/FieldAccess.java`

```
protected Bytecode translate() {  
    if (field instanceof FieldEntry)  
        return receiver.translateAsReference()  
            .append(new GETFIELD((FieldEntry)field,null));  
    else return receiver.translateAsReference()  
        .append(new POP  
            (new GETCONSTANT((ConstantEntry)field,null)));  
}
```

`studente.età`

```
load 2 Studente  
getfield Studente.età
```

`math.PI`

```
load 3 Math  
pop  
getconstant Math.PI
```

} inutile!

Devo compilare il ricevitore delle costanti: `f(x).PI`, `a[i++].PI`

La generazione del codice per i leftvalue

Dentro `Absyn/ArrayAccess.java`

```
protected Bytecode translate() {  
    return array.translateAsReference()  
        .append(index.translateAsInteger())  
        .append(new ALOAD(staticType,null)); }  
}
```

Compilazione di `a[2 + 5]`, con `a` array di `float`

```
load 3 array of float} a.translateAsReference()  
iconst 2 }  
iconst 5 } 2 + 5.translateAsInteger()  
add int }  
aload float
```

Generazione del codice per la creazione di un oggetto

Dentro `Absyn/NewObject.java`

```
protected Bytecode translate() {
    if (actuals != null)
        return new NEW(constructor.clazz,
            new DUP(actuals.translateAs
                (constructor.parameters)))
            .append(new STATICCALL(constructor,null));
    else
        return new NEW(constructor.clazz,
            new DUP(new STATICCALL(constructor,null))); } }
```

- Si noti che la chiamata al costruttore è **statica**, cioè determinata a tempo di compilazione
- Il costruttore `constructor` è quello che era stato selezionato (*sel*) durante il type-checking
- Senza duplicare l'oggetto, esso scomparirebbe dallo stack dopo la chiamata al costruttore

Generazione del codice per la creazione di una stringa

Dentro `Absyn/_String.java`

```
protected Bytecode translate() {  
    return new NEWSTRING(value,null); }  
}
```

Compilazione di `new Studente("vr012345",45)`

```
new Studente  
dup  
newstring "vr012345" } "vr012345",45  
iconst 45           } .translateAs(String,int)  
staticcall Studente.<init>(String,int)
```

Generazione del codice per la creazione di un array

Dentro `Absyn/NewArray.java`

```
protected Bytecode translate() {  
    return size.translateAsInteger()  
        .append(new NEWARRAY(staticElementType, null));  
}
```

`staticElementType` in `new t[size]` è `[[t]]`

Compilazione di `new Studente[100 + a]`

```
iconst 100  
load 3 int  
add int  
newarray Studente  
} 100 + a.translateAsInteger()
```

Generazione del codice per il cast

Dentro `Absyn/Cast.java`

```
protected Bytecode translate() {  
    if (asWhat instanceof Types.ClassType)  
        // e' un vero e proprio cast  
        return expression.translateAsReference()  
            .append(new CHECKCAST(asWhat,null));  
    else // e' una conversione di tipo  
        return expression.translateAs(asWhat); }  
}
```

`gennaro as Student`

```
load 3 Person } gennaro.translateAsReference()  
checkcast Student
```

`34.5 as int`

```
fconst 34.5  
f2i } 34.5.translateAsInteger()
```

Gener. del codice per la chiamata di un metodo

Dentro `Absyn/MethodCallExpression.java`

```
protected Bytecode translate() {
    if (actuals != null)
        return receiver.translateAsReference()
            .append(actuals.translateAs(method.parameters))
            .append(new VIRTUALCALL(method, null));
    else
        return receiver.translateAsReference()
            .append(new VIRTUALCALL(method, null)); }
```

- Si noti che la chiamata è **virtuale**, cioè determinata a tempo di esecuzione
- L'oggetto `method` contiene solo la segnatura del metodo che si vuole chiamare
- Vedremo anche la `MethodCallCommand`, che scarta il valore di ritorno

Compilazione di `gatto.set(3.2, "pongo", 11)`

```
load 2 Gatto}gatto.translateAsReference()  
fconst 3.2  
newstring "pongo" } 3.2, "pongo", 11  
iconst 11 } .translateAs(float, String, int)  
virtualcall Animale.set(float, String, int)
```

Scrittura in un leftvalue

Abbiamo già visto che i leftvalue hanno sia il metodo `translate` che il metodo `translateForWriting`

Dentro `Absyn/Variable.java`

```
public Bytecode translateForWriting
  (Expression rvalue) {
  return rvalue.translateAs(staticType)
    .append(new STORE(varNum,staticType,null)); }
```

Scrittura in x di `13 + 17`

```
  iconst 13 }
  iconst 17 } 13+17.translateAs(int)
  add int
  store 3 int
```

Scrittura in un leftvalue

Dentro `Absyn/FieldAccess.java`

```
public Bytecode translateForWriting
    (Expression rvalue) {
    return receiver.translateAsReference()
        .append(rvalue.translateAs(staticType))
        .append(new PUTFIELD((FieldEntry)field,null)); } }
```

Scrittura in `studente.età` di `13 + 17`

```
load 3 Studente} studente.translateAsReference()
iconst 13      }
iconst 17      } 13+17.translateAs(int)
add int
putfield Studente.età
```

Scrittura in un leftvalue

Dentro `Absyn/ArrayAccess.java`

```
public Bytecode translateForWriting
  (Expression rvalue) {
  return array.translateAsReference()
    .append(index.translateAsInteger())
    .append(rvalue.translateAs(staticType))
    .append(new ASTORE(staticType,null)); }
```

Scrittura in `a[5]` di `13 + 17`

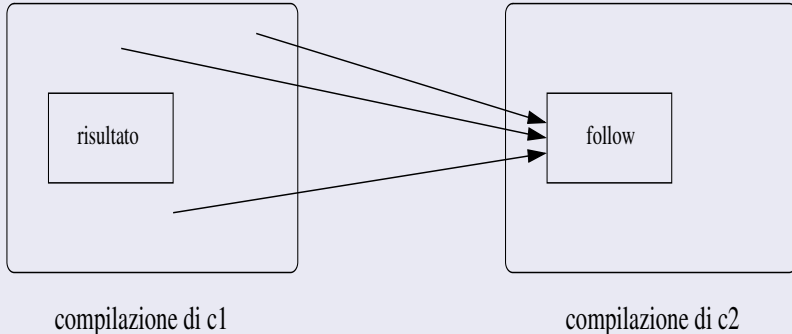
```
load 3 array of int} a.translateAsReference()
iconst 5} 5.translateAsInteger()
iconst 13 }
iconst 17 } 13+17.translateAs(int)
add int
astore int
```

La generazione del codice per i comandi

```
CodeBlock translateAsVoid(CodeBlock follow)
```

- `follow` è ciò che deve essere eseguito eseguito dopo questo comando
- il risultato è il blocco da cui inizia l'esecuzione del comando

La compilazione di `c1 ; c2`



La generazione del codice per l'assegnamento

Dentro `Absyn/AssignmentCommand.java`

```
public CodeBlock translateAsVoid
    (CodeBlock follow) {
    follow = super.translateAsVoid(follow);
    return new LinkedCodeBlock
        (lvalue.translateForWriting(rvalue), follow);
}
```

- Ciò che segue c_1 in $c_1; c_2; c_3; \dots, c_n$ si ottiene compilando ricorsivamente $c_2; c_3; \dots; c_n$, con `follow` come seguito, e usando il risultato come seguito di c_1
- La ricorsione è implementata dalla chiamata alla superclasse
- Si noti che restituiamo un blocco linkato poiché l'assegnamento non contiene scelte

La generazione del codice per una dichiarazione di variabile

Dentro `Absyn/LocalDeclaration.java`

```
public CodeBlock translateAsVoid
    (CodeBlock follow) {
    follow = super.translateAsVoid(follow);
    if (initialiser != null)
        return new LinkedCodeBlock
            (initialiser.translateAs(staticType)
                .append(new STORE(varNum, staticType, null)),
                follow);
    else return new LinkedCodeBlock
        (new INIT(varNum, staticType, null), follow); }
```

- Se c'è un inizializzatore, la compiliamo come un assegnamento
- Altrimenti inizializziamo la variabile al valore di default per il suo tipo

Gener. del codice per la chiamata di un metodo

Dentro `Absyn/MethodCallCommand.java`

```
CodeBlock translateAsVoid(CodeBlock follow) {
    follow = super.translateAsVoid(follow);
    if (actuals != null)
        bytecode = receiver.translateAsReference()
            .append(actuals.translateAs(method.parameters))
            .append(new VIRTUALCALL(method, null));
    else
        bytecode = receiver.translateAsReference()
            .append(new VIRTUALCALL(method, null));
    if (method.returnType != TypeChecker.VOID_TYPE)
        bytecode.append(new POP(null));
    return new LinkedCodeBlock(bytecode, follow); }
```

È come l'espressione di chiamata di un metodo, ma il valore di ritorno viene scartato da una POP

Generazione del codice per il ritorno da metodo

Dentro `Absyn/Return.java`

```
CodeBlock translateAsVoid(CodeBlock follow) {
    Bytecode bytecode;
    if (returned == null)
        bytecode = new RETURN(StaticTypeChecker.VOID_TYPE, null);
    else {
        bytecode = returned.translateAs(staticType);
        bytecode.append(new RETURN(staticType, null));
    }
    return new FinalCodeBlock(bytecode); }
}
```

- Il tipo statico è quello di ritorno del metodo in cui questa istruzione occorre, ed è stato annotato in fase di analisi semantica
- Si noti che restituiamo un blocco finale

La compilazione delle guardie

Le guardie di condizionali e cicli sono espressioni (booleane). Oltre alla normale `translate`, dotiamo quindi le espressioni di un metodo `translateAsTest`

Dentro `Absyn/Expression.java`

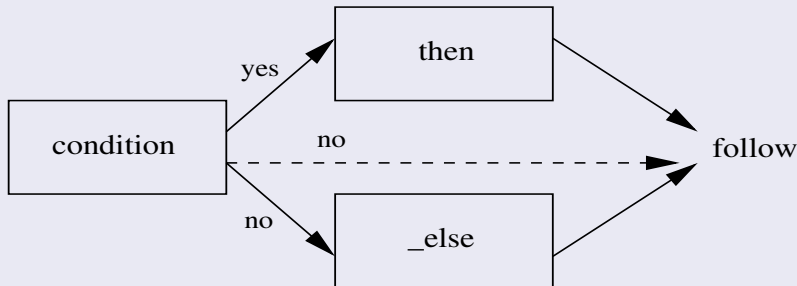
```
public CodeBlock translateAsTest
    (CodeBlock yes, CodeBlock no) {
    return new ConditionalCodeBlock
        (translateAsBoolean(), new IF_TRUE(), yes, no);
}
```

- Si noti che restituisce un blocco condizionale
- Il bytecode condizionale usato è sempre `IF_TRUE`

La generazione del codice per il condizionale

Dentro `Absyn/IfThenElse.java`

```
public CodeBlock translateAsVoid  
    (CodeBlock follow) {  
    follow = super.translateAsVoid(follow);  
    return condition.translateAsTest  
        (then.translateAsVoid(follow),  
         _else != null ?  
         _else.translateAsVoid(follow) : follow); }  
}
```



Codice generato per `if (a & b) then c1 else c2`

```
load 2 boolean  
load 3 boolean  
and  
if_true
```

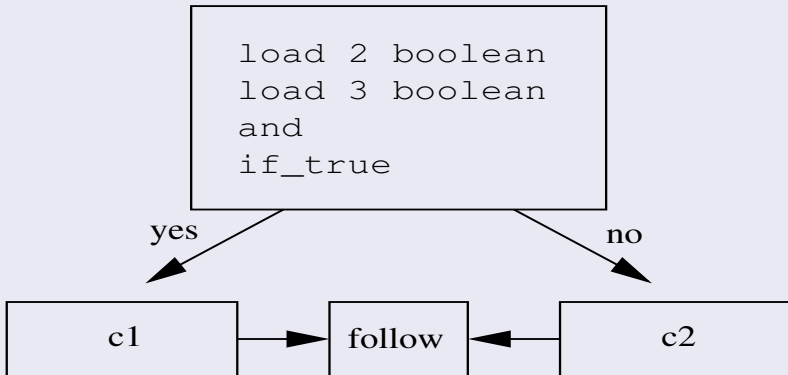
yes

no

c1

follow

c2



Esempio

Codice generato per `if (a >= b) then c1 else c2`

```
load 2 boolean  
load 3 boolean  
ge  
if_true
```

yes

no

c1

follow

c2

Si può fare di meglio!

`ge` e `if_true` possono essere compattati in un unico `if_cmpge`

Ridefiniamo `translateAsTest` dentro

`Absyn/GreaterThanOrEqualTo.java`

```
public CodeBlock translateAsTest
    (CodeBlock yes, CodeBlock no) {
    Types.Type supertype = left.staticType
        .leastCommonSupertype(right.staticType);
    Bytecode leftBytecode =
        left.translateAs(supertype);
    Bytecode rightBytecode =
        right.translateAs(supertype);
    return new ConditionalCodeBlock
        (leftBytecode.append(rightBytecode),
         new IF_CMPGE(supertype), yes, no);
}
```

Esempio ottimizzato

Codice generato per `if (a >= b) then c1 else c2`

```
load 2 boolean  
load 3 boolean  
if_cmpge int
```

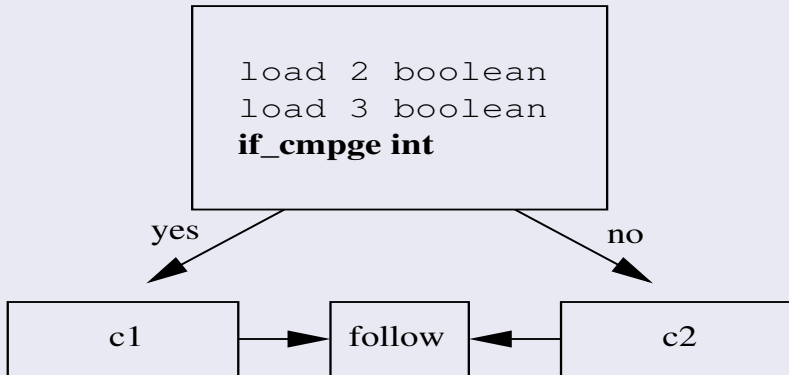
yes

no

c1

follow

c2



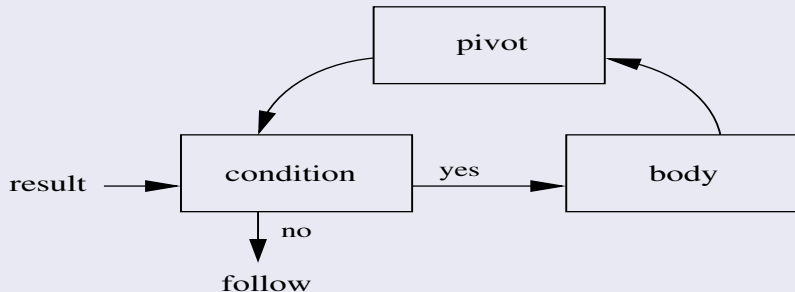
Quando ottimizzare il codice?

- Abbiamo visto un esempio di ottimizzazione durante la stessa generazione del codice
- Altre ottimizzazioni si possono fare dopo, quando tutto il codice è già stato generato
- In genere, è meglio ottimizzare dopo, e preoccuparsi prima della correttezza del codice che si sta generando
- L'ottimizzazione precoce è la madre di tutti i bug!

La generazione del codice per il while

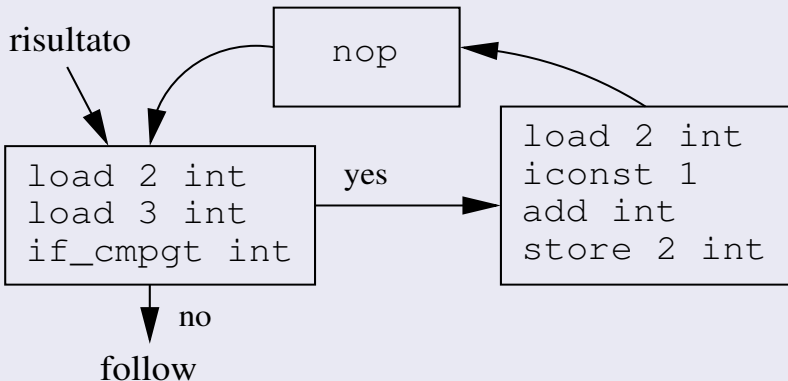
Dentro `Absyn/While.java`

```
CodeBlock translateAsVoid(CodeBlock follow) {  
    LinkedCodeBlock pivot = new LinkedCodeBlock();  
    follow = super.translateAsVoid(follow);  
    CodeBlock result = condition.translateAsTest  
        (body.translateAsVoid(pivot), follow);  
    pivot.linkTo(result); return result; }  
}
```



Esempio

Codice generato per `while (a > b) a := a + 1`

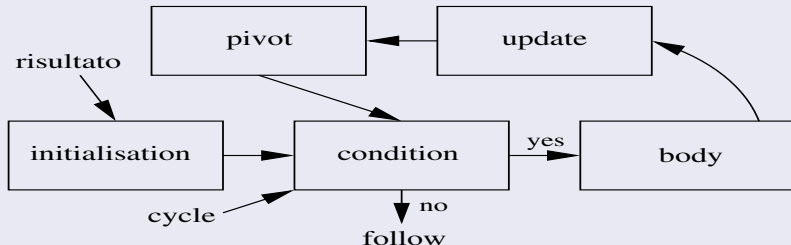


- Il pivot è sempre un blocco vuoto
- Una semplice pulizia del grafo, **dopo** la generazione del codice, lo eliminerà

La generazione del codice per il for

Dentro `Absyn/For.java`

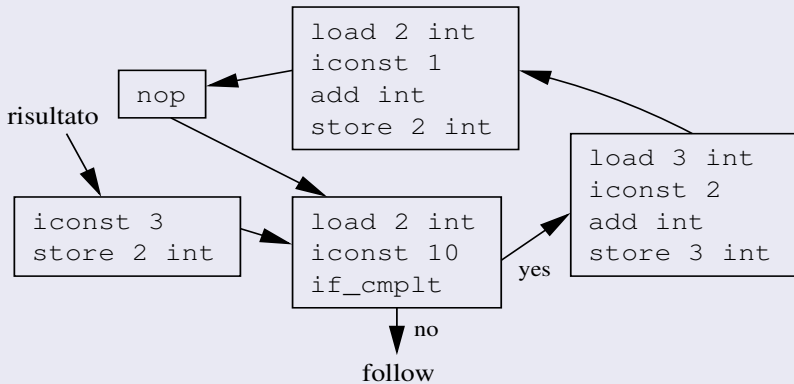
```
CodeBlock translateAsVoid(CodeBlock follow) {  
    LinkedCodeBlock pivot = new LinkedCodeBlock();  
    follow = super.translateAsVoid(follow);  
    CodeBlock cycle = condition.translateAsTest  
        (body.translateAsVoid  
         (update.translateAsVoid(pivot)), follow);  
    pivot.linkTo(cycle);  
    return initialisation.translateAsVoid(cycle);  
}
```



Esempio

Codice generato per

```
for (i := 3; i < 10; i := i + 1) j := j + 2
```



Gestione dei `break` e `continue`

- Il comando `break` provoca un salto alla fine del ciclo in cui occorre
- Il comando `continue` provoca un salto all'inizio del ciclo in cui occorre
- Ma come facciamo a sapere dove saltare?
- Dobbiamo arricchire l'informazione fornita al traduttore:

```
CodeBlock translateAsVoid  
    (CodeBlock follow,  
     CodeBlock _continue,  
     CodeBlock _break)
```

Il ciclo `while` rivisto

```
CodeBlock translateAsVoid(CodeBlock follow,  
    CodeBlock _continue, CodeBlock _break) {  
    ...  
    body.translateAsVoid  
        (pivot,  
         pivot,  
         follow),  
    ...  
}
```

- Usiamo `pivot` (l'inizio del ciclo) per i comandi `continue`
- Usiamo `follow` (ciò che segue il ciclo) per i `break`
- Similmente per il comando `for`

Il ciclo while rivisto

Dentro `Absyn/Continue.java`

```
CodeBlock translateAsVoid(CodeBlock follow,  
    CodeBlock _continue,CodeBlock _break) {  
    return _continue;  
}
```

Dentro `Absyn/Break.java`

```
CodeBlock translateAsVoid(CodeBlock follow,  
    CodeBlock _continue,CodeBlock _break) {  
    return _break;  
}
```

Il generatore di codice

La generazione del codice **per una classe**

- Si crea un'istanza di una struttura dati `Translate/ClassCode.java` che contiene il codice generato per i metodi, i costruttori e le costanti della classe
- Si aggiungono a tale struttura dei metodi di accesso ai campi non `hidden`

La generazione del codice **per un programma**

- Si fa l'analisi semantica del programma, ottenendo un insieme di classi
- Se non ci sono stati errori di tipo, si genera il codice per ciascuna classe, come visto sopra
- È tutto effettuato da un `Translate/Translator.java`, che alla fine fornisce un insieme di `Translate/ClassCode.java`

Il file `Translate/Main_translate.java`

```
public class Kitten {
    public static void main(String[] args) {
        Translator translator;
        translator = new Translator
            (new TypeChecker(new Parser
                (new Lexer(args[0]))));
        translator.translate();
        System.out.println("End of the translation");
        // emette il codice delle classi
        // in formato .dot
    }
}
```