Politecnico di Torino

Ingegneria Informatica

Corso di Laurea Specialistica in Ingegneria Informatica

A.A. 2009/2010

Tesina di Linguaggi e Traduttori

Sviluppo del linguaggio TwynCAT con Antlr e traduzione in codice Beckhoff TwinCAT

Luca Belluccini

Angelo Mantellini

La scelta di Antlr

Antlr è un lexer, parser e tree parser generator e fornisce tutti gli strumenti necessari per la scrittura di interpreti o traduttori.
Questo strumento è capace di generare ASTs (Abstract Syntax Trees), eseguire una ri-scrittura online dell'albero di parsing o generare in automatico stringhe formattate a partire da una singola regola (mediante StringTemplate).
Esso può generare codice Java, C#. C++ e Python.
Permette un error handling flessibile ed estendibile, oltre che strumenti per l'error ricovery.
La scelta dei tool da utilizzare per sviluppare questa tesina è ricaduta su Antlr per:

- una buona community a supporto (una mailing list e diversi esempi, anche di uso reale)
- la possibilità di sviluppare Lexer e Parser mediante un unico Framework
- la possibilità di analizzare il comportamento di un parser LL(*)
- la presenza di un debugger grafico

Antlr viene fornito come pacchetto stand-alone (attualmente alla versione 3.2) o in bundle con l'ambiente di sviluppo AntlrWorks (ma solo per la generazione di codice Java e Python).
Quest'ultimo è uno strumento valido anche se un po' rozzo e non adatto a grammatiche di grandi dimensioni.
Antlr è utilizzabile anche via riga di comando, registrandolo opportunamente nel Classpath di sistema.

TwynCAT: idea di base e vantaggi

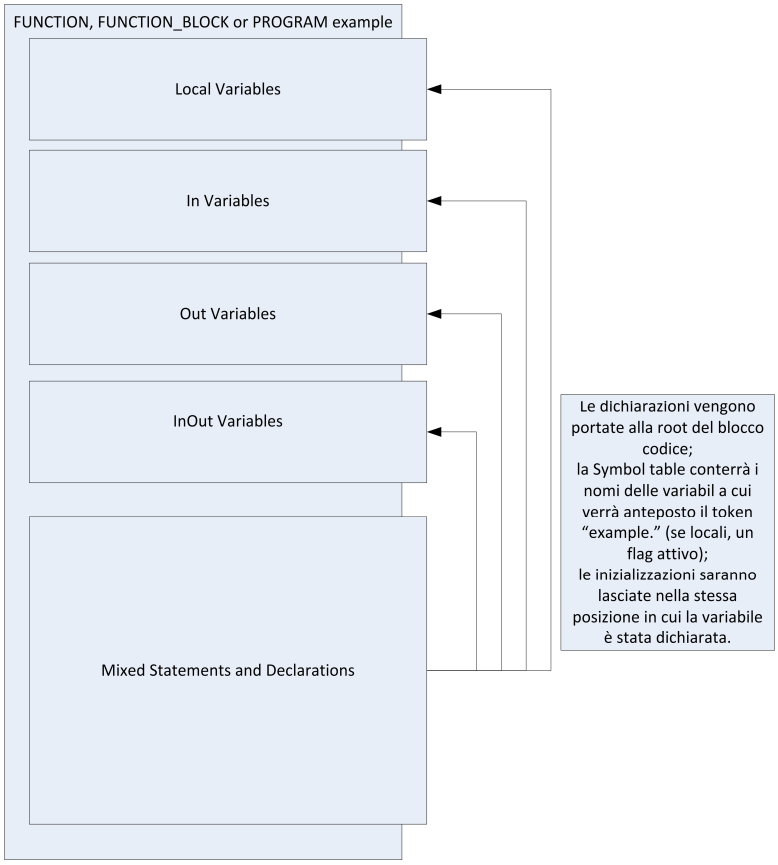
Il linguaggio TwynCAT è ispirato al Python, principalmente per la sintassi.
E' stato possibile avere a disposizione uno stub delle regole di Python.
I blocchi di codice sono delimitati da regole precise di indentazione. Ciò comporta la scrittura di codice più leggibile. Si tratta quindi di un linguaggio che rispetta la così detta off-side rule.
Essendo TwinCAT un linguaggio già molto evoluto, è stato necessario dare la stessa espressività anche al nostro linguaggio.
Per questo, gran parte degli statement originali sono stati trasposti in TwynCAT.
Nella tabella seguente è possibile osservare le differenze nella sintassi.

TwinCAT	TwynCAT

IF <test> THEN
  <statements...>
ELSIF <test> THEN
  <statements...>
ELSE
  <statements...>
END_IF;
	if <test>:
	  <statements...>
	elif <test>:
	  <statements...>
	else:
	  <statements...>

CASE <test> OF <value>: <statements...> <value>: <statements...> ELSE <statements...> END_CASE;	case <test>: <value>: <statements...> <value>: <statements...> default: <statements...>
FOR <index> TO <upper limit> BY <step> DO <statements...> END_FOR;	for <index> in {<start>:<step>:<upper limit>}: <statements...>
WHILE <test> DO <statements...> END_WHILE;	while <test>: <statements...>
REPEAT <statements...> UNTIL <test> END_REPEAT;	repeat: <statements...> until <test>

Un altro vantaggio del nuovo linguaggio è la possibilità di dichiarare le variabili in qualunque punto del codice. Le variabili verranno raggruppate secondo le specifiche di TwinCAT e le inizializzazioni verranno invece lasciate nel punto in cui le variabili in cui esse sono state dichiarate.
Questo punto ha reso complesso il parsing: le variabili sono state man mano raccolte e fatte emergere fino alla radice del blocco di codice in cui esse sono state definite.
In TwinCAT non esiste uno scope a livello di statement, ma a livello di blocco (funzione o programma).



I tipi dato sono stati "modellati" come classi Python "fake". In dettaglio, è possibile distinguere:

[<scope>.][<modifier>.]<type> <variable name> [, <variable name>]*

Dove:

- <scope> può essere "in", "out", "inout"; se assente, la variable è locale
- <modifier> può essere "retain", "constant", "persistent"
- <type> può essere un tipo base, un tipo dato definito dall'utente, il nome di un blocco funzionale
- <variable name> il nome di una variabile (TwinCAT utilizza il formato IEC61131-3)

Da questo esempio è stata esclusa una possibile inizializzazione.
Altri short cut implementati sono gli operatori di assegnazione composti ("+=", "*=",ecc…), l'elevamento a potenza("**") e lo shifting ("<<",">>").
Un difetto dell'ambiente Beckhoff: non è possibile generare codice sorgente ed elaborarlo direttamente nell'ambiente di sviluppo. E' necessario quindi operare manualmente con il codice generato per introdurlo in un file sorgente TwinCAT.

TwynCAT: cosa non è stato implementato

Non sono stati implementati:

- i tipi di dato "aree di memoria"

- i puntatori: la grammatica contiene la regola associata, ma andrebbe integrata non come tipo stand alone (esempio: “pointer.int”), ma come modificatore di un tipo dati comune (esempio: “in.pointer.int pointerToInt”); la traduzione in TwinCAT complica di molto la sua gestione (ha una dichiarazione differente dai tipi dato comuni)
- la dichiarazione di variabili globali e di configurazione: la grammatica contiene la regola associata, ma andrebbero escluse le inizializzazioni dinamiche e implementata la “raccolta e classificazione delle variabili”; questo anche perché esse sono modificabili solo da GUI (non esiste una forma testuale)
- lo short cut disponibile in TwinCAT per l’inizializzazione degli array “<nTimes>(<value>)”
- la possibilità di concatenare le stringhe mediante l’operatore “+” o di poter effettuare il confronto con i comuni operatori: le stringhe non sono di comune utilizzo nei PLC e un appesantimento del Parser per introdurre istruzioni condizionali solo per questo tipo di dato è eccessivo
- il controllo dei range degli array e delle loro dimensioni
- il controllo dei range dei tipi dato base
- escape characters

Sono pronti come classi Java ma non integrati nel Parser gli strumenti per:
- dichiarazione di tipi base (e controllo esistenza)
- dichiarazione di strutture (e controllo esistenza)
- dichiarazione di variabili (di tipo base e strutture)
- controllo delle FUNCTION_BLOCK: deve essere integrato nel Parser come controllo di tutte le variabili passate come argomento alla FB; ad esempio: “NomeFunzione(a:=3, b:=4)” necessita del controllo delle variabili “NomeFunzione.a” e “NomeFunzione.b”
- controllo delle FUNCTION: deve essere completata l’implementazione della lettura del file contenente tutti i prototipi delle funzioni esistenti a causa di un problema con funzioni con numero arbitrario di parametri (è mancante quindi anche l’implementazione della funzione “checkFunction”)

Il linguaggio target TwinCAT: caratteristiche

La versione dimostrativa di Beckhoff TwinCAT PLC è stata sufficiente per capire come è strutturato l’IDE e fare delle piccole prove con del codice sorgente TwinCAT.
La documentazione è reperibile sul sito in formato CHM o HTML.
Non è stato facile scoprire alcuni dettagli riguardo le funzioni e il funzionamento degli scope.
I blocchi di codice possono essere di 3 tipi:
- FUNCTION_BLOCK: è una funzione che non ritorna dati; per essere utilizzata necessita di essere istanziata mediante dichiarazione (esempio: “myFunc: FunctionBlockName”) e successivamente richiamata (esempio: “myFunc(a:=3,b:=8)”) specificando il nome degli argomenti e il valore loro assegnato
- FUNCTION: è una funzione che ritorna un dato; i nomi degli argomenti sono impliciti
- PROGRAM: è un programma

All’inizio di ogni sezione, è presente la sezione delle variabili:
- VAR_INPUT [...] END_VAR, contiene tutte le variabili che verranno utilizzate come Input lines nel FBD
- VAR_OUTPUT [...] END_VAR, contiene tutte le variabili che verranno utilizzate come Output lines nel FBD

- VAR_IN_OUT [...] END_VAR, contiene tutte le variabili che verranno utilizzate come Input-Output lines nel FBD
- VAR [...] END_VAR, contiene tutte le variabili che verranno utilizzate come Local variables, non sono accessibili dal FBD

Le variabili globali, le variabili di configurazione vanno definite in una parte dell'IDE di TwinCAT.
Le sezioni delle variabili possono avere un modificatore: RETAIN, PERSISTENT, CONSTANT.

Lo scope di TwinCAT e il Variable Checking in TwynCAT

I blocchi di codice hanno principalmente 2 tipologie di variabili:

- Locali: potranno essere accedute solo dal blocco di codice in cui sono state definite
- Pubbliche ("IN", "OUT", "IN_OUT"): possono essere accedute da qualunque blocco di codice

Le scelte in TwynCAT, a livello Symbol Table, durante la dichiarazione, sono state le seguenti:

- Il simbolo viene registrato come <Blocco Codice>.<Nome Variabile>
- Se il tipo dato è una struttura, vengono richiamate ricorsivamente tutti i campi della struttura e vengono dichiarati tutti come variabile, ad esempio: "MyFunc.myStructure.a", "MyFunc.myStructure.b", "MyFunc.myStructure.b.size" ognuno con il tipo di dato associato
- Se il simbolo è locale, viene dichiarato imponendo un flag a True: ciò impedirà l'accesso al momento del check della variabile

Durante il controllo di una variabile:

- Si tenta di accedere mediante <Nome Blocco Codice Corrente>.<Stringa Variabile>
    - Se viene individuata, sarà sicuramente accedibile
    - Se non viene individuata, viene rimosso il <Nome Blocco Codice Corrente>: in questo modo viene effettuata una ricerca di <Stringa Variabile>
        - Se il flag Locale è True, non può essere acceduta
        - Se il flag Locale è False, può essere acceduta

In seguito, un esempio con 2 funzioni.

Funzione myFunctionA Variabili Locali: a,b,c Variabili In: inputV Variabili Out: outputV	Funzione myFunctionB Variabili Locali: x,y,z Variabili In: inputV Variabili Out: outputV	Symbol Table: myFunctionA.a (T) myFunctionA.b (T) myFunctionA.c (T) myFunctionA.inputV (F) myFunctionA.outputV (F) myFunctionB.x (T) myFunctionB.y (T) myFunctionB.z (T) myFunctionB.inputV (F) myFunctionB.outputV (F)

Se da "myFunctionA", tentiamo di richiamare tutte le variabili dichiarate al suo interno, avremo un match diretto. Stessa cosa per "myFunctionB".
Se da "myFunctionA" proviamo ad accedere ad "x", verrà cercata "myFunctionA.x": non avremo match perché tale variabile non esiste.
Se da "myFunctionA" proviamo ad accedere a "myFunctionB.x" verrà prima cercata "myFunctionA.myFunctionB.x": non ci sarà match.
In automatico, viene rimosso il primo prefisso e verrà ricercato "myFunctionB.x": ci sarà match, ma verrà controllato il flag di località, che impedirà di poter accedere a tale variabile, che è locale a "myFunctionB".

Questo permette anche di scalare bene tra variabili di blocco e variabili globali.

Approccio alla traduzione e dettagli implementativi

Il primo problema affrontato per il parsing del linguaggio TwynCAT è stato il gestire correttamente l'indentazione.
Abbiamo imposto, per compatibilità, che ogni TAB venga "esploso" in 8 spazi.
L'implementazione è stata possibile mediante l'utilizzo di uno stack che contiene un valore per ogni INDENT effettuato. Il valore è pari al numero di colonne nell'INDENT.
Ogni volta che viene matchato un WHITESPACE alla colonna 0, si controlla il valore contenuto nel top dello stack:

- Se il numero di colonne di WHITESPACE è uguale, non c'è alcuna azione, perché è stato mantenuto il livello di indentazione precedente
- Se il numero di colonne di WHITESPACE è maggiore, si introduce un Token virtuale INDENT nel Lexer e si effettua una push nello stack
- Se il numero di colonne C di WHITESPACE è inferiore, si verifica che C sia contenuto nello stack (viene effettuata una ricerca) e in caso affermativo, si introduce un Token virtuale DEDENT nel Lexer per ogni pop necessaria per raggiungere C. Se C non è presente nello stack, c'è un errore di indentazione

Il problema non è completamente risolto: nel caso in cui un programma termini senza essere ritornati alla colonna 0, è necessario svuotare lo stack delle indentazioni.
Tale problema può essere risolto effettuando modifiche alle funzioni "emit(Token token)" e "nextToken()"del Lexer, assicurandoci che all'arrivo del token EOF, vengano introdotti tutti i DEDENT necessari.

Il secondo problema è dato dalla possibilità di dichiarare variabili in ogni punto del codice. Ciò comporta un riordinamento degli statement, oltre la loro effettiva traduzione in codice TwinCAT.
I tipi di statement possono essere:

- semplici (assegnazioni, chiamate a funzione, dichiarazione variabili, ecc…): istruzioni che non necessitano di una indentazione
- complessi (if, while, for, ecc…): in questo tipo di istruzioni abbiamo un preambolo e successivamente una indentazione obbligatoria seguita da statements

Gli statement complessi contengono sempre all'interno delle regole corrispondenti la regola "codeblock". La regola codeblock corrisponde a:

NEWLINE INDENT statements+ DEDENT

Ogni volta che viene matchata la regola codeblock, dovranno essere raccolte tutte le variabili dichiarate in quel blocco e tutti gli statement nel corretto ordine. Una volta fatto ciò, vengono passate al blocco di codice a cui codeblock appartiene, portando questo bundle sempre più in alto nell'albero di parsing.
Una volta terminato il matching della regola "program" o "function", tutti i dati vengono mergiati in attesa di essere inviati in output dalla regola di root "file".
Onde preservare la possibilità di inizializzare le variabili anche con funzioni o con valori calcolabili solo a runtime, una dichiarazione presente in qualunque punto del codice viene divisa in 2 parti:

- la definizione vera e propria verrà pacchettizzata nel VariableBundle
- l'inizializzazione verrà gestita come statement di tipo assegnazione e introdotto nella lista di statement nel punto di dichiarazione

La gestione delle priorità degli operatori è ispirata a quella di Python, comune a quasi tutti i linguaggi di programmazione. Si tratta di una serie di regole del parser, a partire da “test” che richiamano regole sempre più prioritarie, fino ad arrivare alla regola “atom”.

In alcuni casi, quando la sintassi di TwynCAT differisce molto da quella di TwinCAT, nelle regole del Parser non si hanno tutti gli elementi per effettuare la traduzione.
Per questo, è stato necessario passare alcuni Token o stringhe alle sotto-regole che compongono la “definition”. In seguito, uno stralcio delle regole, cui sono state rimosse le azioni in Java:

```
definition returns [List<String> statements, VariablesBundle vbund]
        : dp=defPurpose? dm=defModifier? (sT=sdt | uT=ID)
        vlist=varList[ ($sT == null?"":$sT.txt) + ($uT == null?"":$uT.text) , ($dp == null) ]
        ;

varList [ String type, boolean isLocal ] returns [ List<String> statements, List<String>
initializations ]
        : vle1=varListElem[ $type, $isLocal ] (',' vleN=varListElem[ $type, $isLocal ])*
        ;

fragment varListElem [String vType, boolean isLocal] returns [String txt, String initialization]
        : ID (trailer)? (arrayModifier ('=' ace=arrayConstantExpression)?
        | ( '[' strl=DECIMALL ']' )? ('=' t=test )? )?
        ;
```

Per poter tradurre correttamente ogni singola variabile in “varListElem” è necessario che la regola “definition” passi a “varList” una String contenente il tipo dato e una boolean che specifica se la variable è locale o meno. Queste informazioni vengono passate di nuovo a “varListElem”.
Quest’ultima le sfrutterà per generare la stringa corrispondente al codice TwinCAT e permetterà di effettuare la dichiarazione come variabile locale o meno nelle azioni in Java.

Antlr: Tutorial e informazioni pratiche

Il primo passo è quello di effettuare il download di Antlr e di AntlrWorks.
Per la generazione dell’albero in forma grafica è necessario installare anche GraphViz.
Successivamente, si deve aggiungere il jar di Antlr al CLASSPATH di sistema o utente.
Per iniziare, è possibile utilizzare AntlrWorks: la GUI ci permette di creare un nuovo file, che può essere uno String Template Group o una Grammatica.
Lo String Template Group è un file che può essere utilizzato in combinazione con Antlr per la generazione di stringhe utilizzando dei Template, molto più velocemente che con concatenazione di stringhe o StringBuilder di Java.
Tramite File – New è possibile creare una grammatica, per ora partiamo da una “Combined Grammar”. Introduciamo un nome e verrà generato il codice.
In realtà è possibile utilizzare anche Eclipse come IDE, ma i 2 plugin disponibili non sono molto aggiornati e/o tendono a crashare troppo spesso. Sono accessibili dal sito di Antlr.
E’ consigliabile avere a portata di mano la reference stampabile dell’ultima versione.
Analizziamo una grammatica molto semplice, come la seguente:

```
grammar T;
/** Match things like "call foo;" */
r : 'call' ID ';' {System.out.println("invoke "+$ID.text);} ;
ID: 'a'..'z'+ ;
WS: (' '|'\n'|'\r')+   {$channel=HIDDEN;} ; // ignore whitespace
```

La regola “r” è una regola del Parser, “ID” e “WS” del Lexer. Questa distinzione viene effettuata da Antlr a seconda del case utilizzato per la definizione della regola.
E’ possibile utilizzare operatori come la chiusura di Kleene o la chiusura positiva (“*” e “+” rispettivamente).

Le azioni sono eseguite all'interno delle parentesi "{ … }". Le azioni verranno eseguite subito dopo il parsing dell'elemento alla sinistra e subito prima dell'elemento alla destra.
Si può fare riferimento agli attributi dei Token (del Lexer) o alle Rules (del Parser) con i rispettivi nomi specificati all'interno della regola preceduti dal simbolo "$". Se c'è ambiguità, è necessario dichiarare un alias.
Esempio: in "rule: ID (, ID)+" per poter accedere ad entrambi gli ID, è necessario scrivere "rule: alias1=ID (, alias2=ID)+".
Gli attributi dei Token sono: text, type, line, index, pos, channel, tree, int.
Gli attributi delle Rules sono: text, start, stop, tree, st.
Per accedere quindi ad "ID", è necessario utilizzare "$ID.text".
Da qui, è consigliabile visionare o provare a modificare alcuni esempi, seguire il debugger.
Alcuni link utili sono:

- Expression evaluator
- Introduzione ad Antlr V3
- Mailing list
- Le ultime grammatiche messe in evidenza, alcune complesse

Analizziamo un set di regole di una grammatica più complessa.

```
exprStm returns [ String txt ]:
t1=test augAssign t2=test { $txt = $t1.txt + " := " + $t1.txt + $augAssign.txt + $t2.txt + ";"; }
| t3=test '=' t4=test { $txt = $t3.txt + " := " + $t4.txt + ";"; }
        ;

augAssign returns [ String txt ]
        : '+=' { $txt = " + "; }
        | '-=' { $txt = " - "; }
        | '*=' { $txt = " * "; }
        | '/=' { $txt = " / "; }
        | '%=' { $txt = " MOD "; }
        | '&=' { $txt = " AND "; }
        | '|=' { $txt = " OR "; }
        ;
```

La regola "exprStm" ritorna come attributo una variabile String di nome "txt": sarà accessibile da altre regole che la contengono come "$exprStm.txt", ovvero come un comune attributo.
Tale regola può essere strutturata in 2 modi:

- "test augAssign test"
- "test = test"

Sono stati introdotti gli alias proprio per eliminare l'ambiguità nel riferirsi ad una delle due regole "test". Le stringhe vengono costruite a seconda dei casi e assegnate all'attributo "txt".

Altra sezione di codice, dove viene utilizzata la direttiva "@init". Qui viene sfruttata per inizializzare i due oggetti di ritorno "statements" e "vbund" prima del loro effettivo utilizzo.

```
codeBlock returns [ List<String> statements, VariablesBundle vbund  ]
@init  {
        $statements = new LinkedList<String>();
        $vbund = new VariablesBundle();
        }
        : stm=simpleStm
        {
        if($stm.statements != null) {
                $statements.addAll($stm.statements); $vbund.addAll($stm.vbund);
        }
        }
        | NEWLINE INDENT ( stms=statement
        {
        if($stms.statements != null) {
                $statements.addAll($stms.statements); $vbund.addAll($stms.vbund);
        }
        })+ DEDENT
        ;
```

Le regole possono essere condizionate, come nel seguente esempio.

```
myRule:
        { value == 0 }?=> ‘expecting’ ‘rule’ number
        | ‘not’ ‘expecting’ number
        ;
```

E’ presente anche la direttiva “@after” per far eseguire azioni al termine del match di una regola. Sono particolarmente importanti perché con il backtrack alcune azioni devono essere eseguite all’inizio o alla fine di una regola.
La grammatica “twyncat.g” contiene esempi sull’overload dei metodi del Lexer per modificare il naturale flow dei Token (per la gestione degli INDENT e DEDENT in questo caso).
Nell’svn sono presenti anche file Java che mostrano come può essere richiamato il parser da una comune applicazione Java.