Laboratorio di Automi e Linguaggi Formali

Davide Bresolin

a.a. 2017/2018

1 Introduzione

Oggi ci sono migliaia di linguaggi informatici disponibili: linguaggi di programmazione, di markup, di query, eccetera, e ne vengono pubblicati di nuovi ogni anno. Ogni informatico, ad un certo punto della sua carriera, si ritrova a dover definire un nuovo linguaggio per qualche scopo particolare. In questo tutorial vedremo come utilizzare alcuni concetti di base visti al corso di Automi e Linguaggi Formali ed il generatore di parser ANTLR v4 per creare un semplice linguaggio di programmazione imperativa che chiameremo SWL (o Simple While Language).

2 Impostazione dell'ambiente di lavoro

Il file `laboratorio_antlr.zip` contiene il generatore di parser ANTLR versione 4.7.1, le librerie di runtime per il linguaggio C++ per linux, la grammatica che definisce i costrutti di base di SWL ed il codice del syntax checker e del traduttore descritti nel seguito tutorial.

Per poter utilizzare ANTLR nei computer dei laboratori è necessario estrarre il contenuto del file `.zip` nella *propria home directory*:

```
dbresoli@t68:~$ unzip laboratorio_antlr.zip
```

L'estrazione del file crea una struttura di cartelle con il necessario per far funzionare ANTLR:

```
antlr4
├── bin
├── include
├── lib
└── swl
```

La cartella `antlr4` contiene lo script `setup.sh` che inizializza l'ambiente di lavoro e che va eseguito all'inizio di ogni sessione di lavoro prima di utilizzare ANTLR:

```
dbresoli@t68:~$ cd antlr4
dbresoli@t68:~/antlr4$ source setup.sh
```

3 Creazione del file con la grammatica

Vediamo ora come possiamo definire il nostro linguaggio di programmazione SWL, partendo da un esempio di programma in SWL:

```
begin
    let a be 5
    let b be 10
    add 3 to b
    add b to a
    add a to b
    print b
    print 3
end
```

Il programma qui sopra mostra le funzionalità di base del linguaggio, che può avere tre tipi di costrutti:

- l'istruzione `let` che definisce una nuova variabile
- l'istruzione `add` che fa la somma
- le istruzioni di input/output `print`

Le variabili sono solo di tipo intero senza segno, ed ogni programma inizia e termina con le parole chiave `begin` e `end`. Il file `swl.g4` contenuto nella cartella `swl` contiene la grammatica che definisce la sintassi del linguaggio SWL:

```
grammar swl;
program   : 'begin' statement+ 'end';

statement : assign | add | print ;

assign    : 'let' ID 'be' (NUMBER | ID) ;
print     : 'print' (NUMBER | ID) ;
add       : 'add' (NUMBER | ID) 'to' ID ;

ID        : [a-z]+ ;
NUMBER    : [0-9]+ ;
WS        : [ \n\t]+ -> skip;
ErrorChar : . ;
```

4 Generazione del Parser e del Lexer

Dopo aver creato il file con la grammatica possiamo utilizzare il comando `antlr4` per creare automaticamente il codice C++ necessario per fare il parsing dei programmi SWL. ANTLR permette di generare codice per diversi linguaggi di target: Java, Pyhton, C++, C#, Swift e Go. In questo tutorial ci utilizzeremo il linguaggio C++. Il linguaggio target va specificato con l'opzione `-Dlanguage`:

```
dbresoli@t68:~/antlr4/swl$ antlr4 -Dlanguage=Cpp swl.g4
```

La sintassi dell'opzione è case-sensitive: è importante fare attenzione alla '`C`' maiuscola. In caso di errore si riceve un messaggio di errore simile al seguente.

```
error(31):  ANTLR cannot generate cpp code as of version 4.7.1
```

L'esecuzione corretta di `antlr4` crea i seguenti file:

```
swlBaseListener.h
swlLexer.cpp
swlLexer.tokens
swlParser.cpp
swlLexer.h
swlListener.cpp
swlParser.h
swlBaseListener.cpp
swl.interp
swlLexer.interp
swlListener.h
swl.tokens
```

5 Creazione di un Syntax Checker

Per testare il corretto funzionamento della grammatica di SWL possiamo scrivere un semplice programma che esegue queste semplici operazioni:

1. legge un file con un programma scritto in SWL
2. utilizza la classe `swlLexer` per suddividere il file in *token*
3. utilizza la classe `swlParser` per creare un *albero sintattico* che rappresenta la struttura del testo
4. controlla il numero di errori di sintassi che il parser ha generato nella costruzione dell'albero e lo riporta all'utente.

Il codice del programma si trova nel file `syncheck.cpp` ed è riportato qui sotto:

```
#include <iostream>
#include <fstream>
#include <string>
#include "antlr4-runtime.h"
#include "swlLexer.h"
#include "swlParser.h"

using namespace std;
using namespace antlr4;

int main(int argc, char* argv[]) {
    if(argc != 2) {
        cout << "Usage: syncheck filename.swl" << endl;
        return 1;
    }
    ifstream swlFile(argv[1]);
    ANTLRInputStream input(swlFile);
    swlLexer lexer(&input);
    CommonTokenStream tokens(&lexer);
    swlParser parser(&tokens);
    tree::ParseTree *tree = parser.program();
    int errors = parser.getNumberOfSyntaxErrors();
    if(errors > 0) {
        cout << errors << " syntax errors found." << endl;
        return 1;
    }
    cout << "No syntax errors found." << endl;
    return 0;
}
```

Per facilitare la compilazione dei programmi la cartella `swl` contiene un `Makefile` che richiama il compilatore C++ con i parametri corretti per i diversi file `.cpp` contenuti nella cartella ed esegue il linking con la libreria `antlr4-runtime`. Per compilare il syntax checker è sufficiente utilizzare il comando `make`:

```
dbresoli@t68:~/antlr4/swl$ make syncheck
```

Oltre al target `syncheck` che compila il controllore sintattico, il `Makefile` mette a disposizione anche i seguenti target:

- `make clean` che elimina gli eseguibili ed i file temporanei creati dal compilatore C++;
- `make distclean` che elimina i file generati da ANTLR a partire dal file con la grammatica;
- `make translate` che compila il codice con il traduttore.

La compilazione di `syncheck.cpp` crea l'eseguibile `syncheck` che può essere utilizzato per controllare la sintassi dei programmi SWL:

```
dbresoli@t68:~/antlr4/swl$ ./syncheck example.swl
No syntax errors found.
```

6 Creazione di un Listener

Il traduttore da SWL a C++ sfrutta un *listener* per visitare l'albero sintattico del programma SWL e generare l'output. Il listener è implementato nella classe `MyListener` che estende l'interfaccia definita dalla classe `swlBaseListener` creata da ANTLR. La definizione è contenuta nel file `MyListener.h`:

```
#pragma once

#include "antlr4-runtime.h"
#include "swlParser.h"
#include "swlBaseListener.h"


/**
 * This class defines a concrete listener for a parse tree produced by swlParser.
 */
class  MyListener : public swlBaseListener {
private:
  int indent = 0;

public:

  void enterProgram(swlParser::ProgramContext *ctx);
  void exitProgram(swlParser::ProgramContext *ctx);

  void exitAssign(swlParser::AssignContext *ctx);

  void exitPrint(swlParser::PrintContext *ctx);

  void exitAdd(swlParser::AddContext *ctx);

};
```

La classe contiene un attributo privato `indent` che serve per indentare correttamente l'output ed una serie di metodi che stabiliscono quello che deve fare il parser quando incontra un certo token specifico. Per esempio, deve sostituire l'istruzione `begin` con le prime righe di un programma C++ (inclusione delle librerie e la definizione del `main`). Simmetricamente, deve sostituire l'istruzione `end` con la fine del programma. Per far questo la classe utilizza i metodi `enterProgram` e `exitProgram`, che vengono eseguiti rispettivamente all'inizio e alla fine della regola `program` presente nella grammatica. L'implementazione dei metodi si trova nel file `MyListener.cpp`.

```
void MyListener::enterProgram(swlParser::ProgramContext *ctx) {
    cout << "#include <iostream>" << endl << endl
         << "using namespace std;" << endl << endl
         << "int main() {" << endl;
    indent += 4;
}

void MyListener::exitProgram(swlParser::ProgramContext *ctx) {
    cout << "}" << endl;
}
```

Nel caso dell'istruzione di assegnamento è sufficiente utilizzare solo il metodo `exitAssign`, che produce il codice C++ che definisce una nuova variabile intera: per esempio, l'istruzione `let a be 0` viene tradotta in `int a = 0;`. La regola della grammatica per l'assegnamento è

```
assign    : 'let' ID 'be' (NUMBER | ID) ;
```

il metodo deve quindi essere in grado di sapere qual'è il nome della variabile da definire (primo token `ID`) e se l'istruzione assegna un `NUMBER` o un `ID`. Questa informazione è presente nel parametro `ctx` del

metodo, che punta ad un oggetto di tipo `swlParser::AssignContext` che rappresenta il contesto in cui viene applicata la regola. La definizione del contesto si trova nel file `swlParser.h` ed è la seguente:

```
class  AssignContext : public antlr4::ParserRuleContext {
  public:
    AssignContext(antlr4::ParserRuleContext *parent, size_t invokingState);
    virtual size_t getRuleIndex() const override;
    std::vector<antlr4::tree::TerminalNode *> ID();
    antlr4::tree::TerminalNode* ID(size_t i);
    antlr4::tree::TerminalNode *NUMBER();

    virtual void enterRule(antlr4::tree::ParseTreeListener *listener) override;
    virtual void exitRule(antlr4::tree::ParseTreeListener *listener) override;

  };
```

In questo caso siamo interessati ai metodi `ID(size_t i)` e `NUMBER()` che ci permettono di accedere all'informazione sui nodi terminali. Poiché la regola usa due terminali di tipo `ID` il metodo `ID(size_t i)` prende come parametro l'indice del terminale a cui siamo interessati: `ID(0)` è il nome della variabile da definire, `ID(1)` è il nome della variabile alla destra di `be` (se esiste). Il metodo `NUMBER()` non ha parametri perché c'è un solo terminale di tipo numerico nella regola.

L'implementazione di `exitAssign` ottiene il nome della variabile da creare, quindi controlla il numero di terminali di tipo `ID` presenti nel contesto: uno se l'istruzione assegna un numero alla variabile, due se l'istruzione assegna un'altra variabile. Quindi procede scrivendo su `cout` l'istruzione C++ corrispondente. La funzione `string(indent, ' ')` genera una stringa composta da un numero di spazi pari al valore di `indent` per allineare correttamente il testo.

```
void MyListener::exitAssign(swlParser::AssignContext *ctx) {
    string name = ctx->ID(0)->getText();
    string val;
    if(ctx->ID().size() > 1) {
        val = ctx->ID(1)->getText();
    } else {
        val = ctx->NUMBER()->getText();
    }
    cout << string(indent, ' ') << "int " << name << " = " << val << ";" << endl;
}
```

Infine, i metodi `exitPrint` e `exitAdd` si occupando delle istruzioni di stampa e di somma:

```
void MyListener::exitPrint(swlParser::PrintContext *ctx) {
    string val;
    if(ctx->ID() != NULL) {
        val = ctx->ID()->getText();
    } else {
        val = ctx->NUMBER()->getText();
    }
    cout << string(indent, ' ') << "cout << " << val << " << endl;" << endl;
}

void MyListener::exitAdd(swlParser::AddContext *ctx) {
    string name;
    string val;
    if(ctx->ID().size() > 1) {
        name = ctx->ID(1)->getText();
        val = ctx->ID(0)->getText();
    } else {
        name = ctx->ID(0)->getText();
        val = ctx->NUMBER()->getText();
```

```
    }
    cout << string(indent, ' ') << name << " += " << val << ";" << endl;
}
```

7 Completiamo il traduttore

Il file `translate.cpp` contiene il codice del `main` per il traduttore da SWL a C++. Il codice è molto simile a quello del syntax checker: come prima cosa si legge il file con l'input, lo si scompone in token e si genera l'albero sintattico. Se non ci sono errori di sintassi si procede con la generazione del codice C++: si crea un'istanza del listener e si richiama la funzione `DEFAULT.walk` che visita l'albero sintattico ed esegue i metodi del listener.

```
#include <iostream>
#include <fstream>
#include <string>
#include "antlr4-runtime.h"
#include "swlLexer.h"
#include "swlParser.h"
#include "MyListener.h"

using namespace std;
using namespace antlr4;

int main(int argc, char* argv[]) {
    if(argc != 2) {
        cout << "Usage: translate filename.swl" << endl;
        return 1;
    }
    ifstream swlFile(argv[1]);
    ANTLRInputStream input(swlFile);
    swlLexer lexer(&input);
    CommonTokenStream tokens(&lexer);
    swlParser parser(&tokens);
    tree::ParseTree *tree = parser.program();
    int errors = parser.getNumberOfSyntaxErrors();
    if(errors > 0) {
        cout << errors << " syntax errors found, aborting." << endl;
        return 1;
    }
    MyListener listener;
    tree::ParseTreeWalker::DEFAULT.walk(&listener, tree);
    return 0;
}
```

Dopo aver compilato il traduttore con il comando `make translate` possiamo provare l'esecuzione sul programma di esempio:

```
dbresoli@t68:~/antlr4/swl$ ./translate example.swl
```

ottenendo l'output seguente:

```
#include <iostream>

using namespace std;

int main() {
    int a = 5;
    int b = 10;
```

```
    b += 3;
    a += b;
    b += a;
    cout << b << endl;
    cout << 3 << endl;
}
```

Riferimenti

Per maggiori informazioni su ANTLR potete far riferimenti ai siti web:

- `http://www.antlr.org/` sito ufficiale di ANTLR v4
- `https://github.com/antlr/antlr4/tree/master/runtime/Cpp` repository github con il codice e la documentazione del runtime C++ per ANTLR
- `https://tomassetti.me/antlr-mega-tutorial/` un tutorial molto esteso sull'uso di ANTLR con molti esempi d'uso in Javascript, Python, Java e C#