Laboratorio di Automi e Linguaggi Formali

Davide Bresolin

a.a. 2018/2019

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 TinyREXX.

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 TinyREXX ed il codice del syntax checker e del traduttore descritti in questo tutorial.

Per poter utilizzare ANTLR nei computer dei laboratori è necessario estrarre il contenuto del file `laboratorio_antlr.zip` da qualche parte all'interno della *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
└── tinyrexx
```

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 TinyREXX, partendo da un esempio di programma in TinyREXX:

```
pull n
do while n < 10
    n = n + 1
end
say n
```

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

- l'istruzione di assegnamento, identificata dal simbolo `=`
- l'istruzione `say` stampa un valore sullo schermo
- l'istruzione `pull` che legge un valore da tastiera
- il ciclo `while`
- le quattro operazioni aritmetiche `+, -, *, /`
- operatori di confronto come `==, <, <=, >, >=`

Le variabili sono solo di tipo intero, e vengono create dinamicamente la prima volta che si incontra un nuovo nome di variabile. Il file `tinyrexx.g4` contenuto nella cartella `tinyrexx` contiene la grammatica che definisce la sintassi del linguaggio TinyREXX:

```
grammar tinyrexx;

program   : statement+;

statement : assign | print | input | w_loop ;

assign    : ID '=' a_expr ;
print     : 'say' a_expr ;
input     : 'pull' ID ;
w_loop    : 'do' 'while' test statement+ 'end' ;
test      : a_expr r_op a_expr;
a_expr    : ID | NUMBER | '(' a_expr ')' | a_expr a_op a_expr | MINUS a_expr ;
a_op      : MINUS | PLUS | MUL | DIV ;
r_op      : EQUAL | LT | LEQ | GT | GEQ ;

MINUS     : '-' ;
PLUS      : '+' ;
MUL       : '*' ;
DIV       : '/' ;
EQUAL     : '==' ;
LT        : '<' ;
LEQ       : '<=' ;
GT        : '>' ;
GEQ       : '>=' ;
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 TinyREXX. 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/tinyrexx$ antlr4 -Dlanguage=Cpp tinyrexx.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:

```
tinyrexxBaseListener.h
tinyrexxLexer.cpp
tinyrexxLexer.tokens
tinyrexxParser.cpp
tinyrexxLexer.h
tinyrexxListener.cpp
tinyrexxParser.h
tinyrexxBaseListener.cpp
tinyrexx.interp
tinyrexxLexer.interp
tinyrexxListener.h
tinyrexx.tokens
```

5 Creazione di un Syntax Checker

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

1. legge un file con un programma scritto in TinyREXX
2. utilizza la classe `tinyrexxLexer` per suddividere il file in *token*
3. utilizza la classe `tinyrexxParser` 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 "tinyrexxLexer.h"
#include "tinyrexxParser.h"

using namespace std;
using namespace antlr4;

int main(int argc, char* argv[]) {
    if(argc != 2) {
        cout << "Usage: syncheck filename.rexx" << endl;
        return 1;
    }
    ifstream tinyrexxFile(argv[1]);
    if(tinyrexxFile.fail()){
        //Errore nell'apertura del file
        cout << "Error while reading file " << argv[1] << endl;
        return 1;
    }
    ANTLRInputStream input(tinyrexxFile);
    tinyrexxLexer lexer(&input);
    CommonTokenStream tokens(&lexer);
    tinyrexxParser 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 `tinyrexx` contiene un `Makefile` che richiama ANTLR per generare il codice C++ a partire dalla grammatica del linguaggio, compila i diversi file `.cpp` contenuti nella cartella ed infine esegue il linking con la libreria `antlr4-runtime`. Per compilare il syntax checker è sufficiente utilizzare il comando `make`:

```
dbresoli@t68:~/antlr4/tinyrexx$ 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 da ANTLR e dall compilatore C++;
- `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 TinyREXX:

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

6 Creazione di un Listener

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

```
#pragma once

#include "antlr4-runtime.h"
#include "tinyrexxParser.h"
#include "tinyrexxBaseListener.h"

/**
 * This class defines a concrete listener for a parse tree produced by tinyrexxParser.
 */
class  MyListener : public tinyrexxBaseListener {
private:
  int indent = 0;
  std::set<std::string> vars;

public:
  MyListener(const std::set<std::string> &ids);

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

  void enterAssign(tinyrexxParser::AssignContext * ctx);
  void exitAssign(tinyrexxParser::AssignContext * ctx);

  void enterPrint(tinyrexxParser::PrintContext * ctx);
  void exitPrint(tinyrexxParser::PrintContext * ctx);

  void exitInput(tinyrexxParser::InputContext * ctx);
```

```
  void enterW_loop(tinyrexxParser::W_loopContext * ctx);
  void exitW_loop(tinyrexxParser::W_loopContext * ctx);

  void enterTest(tinyrexxParser::TestContext * ctx);
  void exitTest(tinyrexxParser::TestContext * ctx);

  void enterA_expr(tinyrexxParser::A_exprContext * ctx);
  void exitA_expr(tinyrexxParser::A_exprContext * ctx);

  void exitA_op(tinyrexxParser::A_opContext * ctx);
  void exitR_op(tinyrexxParser::R_opContext * ctx);

};
```

La classe contiene un attributo privato `indent` che serve per indentare correttamente l'output, un attributo privato `vars` che contiene la lista delle variabili dichiarate nel codice TinyREXX da tradurre, ed una serie di metodi che stabiliscono quello che deve fare il parser quando incontra un certo nodo specifico. Come prima cosa, il traduttore deve generare le prime righe di un programma C++ (inclusione delle librerie e la definizione del `main`). Simmetricamente, alla fine del programma deve chiudere correttamente il codice C++. 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(tinyrexxParser::ProgramContext *ctx) {
    cout << "#include <iostream>" << endl << endl
         << "using namespace std;" << endl << endl
         << "int main() {" << endl;
    indent += 4;
    // Dichiara le variabili
    for (auto name : vars) {
        cout << string(indent, ' ') << "int " << name << " = 0;" << endl;
    }

}

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

Nel caso dell'istruzione `pull` di input è sufficiente utilizzare solo il metodo `exitInput`, che produce il codice C++ che legge da `cin` il valore di una variabile. La regola della grammatica per l'input è

```
input     : 'pull' ID ;
```

il metodo deve quindi essere in grado di sapere qual'è il nome della variabile da assegnare (token `ID`). Questa informazione è presente nel parametro `ctx` del metodo, che punta ad un oggetto di tipo `tinyrexxParser::InputContext` che rappresenta il contesto in cui viene applicata la regola. La definizione del contesto si trova nel file `tinyrexxParser.h` ed è la seguente:

```
  class  InputContext : public antlr4::ParserRuleContext {
  public:
    InputContext(antlr4::ParserRuleContext *parent, size_t invokingState);
    virtual size_t getRuleIndex() const override;
    antlr4::tree::TerminalNode *ID();

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

  };
```

In questo caso siamo interessati al metodo `ID()` che ci permette di accedere all'informazione sul nodo terminale di tipo `ID`.

L'implementazione di `exitInput` ottiene il nome della variabile da assegnare, quindi procede scrivendo su `cout` l'istruzione C++ che fa la lettura da tastera. 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::exitInput(tinyrexxParser::InputContext * ctx) {
    string name = ctx->ID()->getText();
    cout << string(indent, ' ') << "cin >> " << name << ";" << endl;
}
```

Tradurre una istruzione di assegnamento è più complicato. Partiamo dalla regola della grammatica:

```
assign    : ID '=' a_expr ;
```

I componenti di una istruzione di assegnamento sono due: l'`ID` della variabile da assegnare e un *espressione aritmetica* che definisce il valore da assegnare. I metodi `enterAssign` e `exitAssign` si occupano di scrivere a schermo solo una parte della traduzione:

```
void MyListener::enterAssign(tinyrexxParser::AssignContext * ctx) {
    string name = ctx->ID()->getText();
    cout << string(indent, ' ') << name << " = " ;
}

void MyListener::exitAssign(tinyrexxParser::AssignContext * ctx) {
    cout << ";" << endl;
}
```

Il metodo `enterAssign` scrive a schermo "`nomevariabile =`", mentre `exitAssign` si occupa di scrivere il punto e virgola finale. Come potete vedere, i due metodi ignorano completamente l'espressione aritmetica che segue l'uguale. Dove viene generata la traduzione dell'espressione aritmetica?

Nella grammatica `tinyrexx.g4` le espressioni aritmetiche sono descritte dalla regola per il nonterminale `a_expr`:

```
a_expr    : ID | NUMBER | '(' a_expr ')' | a_expr a_op a_expr | MINUS a_expr ;
```

Ogni volta che il Listener incontra una espressione aritmetica esegue `enterA_expr` all'inizio dell'espressione e `exitA_expr` alla fine dell'espressione. Di conseguenza, possiamo utilizzare questi due metodi per tradurre un'espressione aritmetica *senza doverlo fare esplicitamente* nei metodi per la regola dell'assegnamento. Entrambi i metodi hanno un parametro `ctx`, che punta ad un oggetto di tipo `tinyrexxParser::A_exprContext` che rappresenta il contesto in cui viene applicata la regola. La definizione del contesto si trova nel file `tinyrexxParser.h` ed è la seguente:

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

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

  };
```

In questo caso, per generare correttamente il codice C++ dell'espressione dobbiamo capire quale dei cinque casi della regola `a_expr` è stato applicato. Possiamo capire in quale caso ci troviamo controllando quali sono i metodi che ritornano un valore diverso da `NULL` nell'oggetto `ctx`:

- se `ctx->ID()` ritorna un valore non nullo, allora è stato applicato il primo caso della regola, e l'espressione è semplicemente un nome di variabile;
- se `ctx->NUMBER()` ritorna un valore non nullo, allora è stato applicato il secondo caso della regola, e l'espressione è un numero;
- se `ctx->a_op()` ritorna un valore non nullo, allora è stato applicato il quarto caso della regola, e ci sono due sottoespressioni con un operatore aritmetico in mezzo;
- se `ctx->MINUS()` ritorna un valore non nullo, allora è stato applicato il quinto caso della regola, e l'espressione inizia con un segno `-` seguito da una sottoespressione;
- se tutti i metodi precedenti ritornano valore nullo, allora è stato applicato il terzo caso della regola, e l'espressione è racchiusa tra parentesi.

I metodi `enterA_expr` e `exitA_expr` usano i cinque casi precedenti per generare il codice C++ dell'espressione aritmetica:

```
void MyListener::enterA_expr(tinyrexxParser::A_exprContext * ctx) {
    // controllo in quale caso sono
    if(ctx->ID() != NULL) {
        // caso ID semplice
        cout << ctx->ID()->getText();
    } else if(ctx->NUMBER() != NULL) {
        // caso valore numerico semplice
        cout << ctx->NUMBER()->getText();
    } else if(ctx->MINUS() !=  NULL) {
        // caso operatore - unario
        cout << "-" ;
    } else if(ctx->a_op() != NULL) {
        // caso operatore binario: gestito da enterA_op
    } else {
        // caso parentesi
        cout << "(" ;
    }
}

void MyListener::exitA_expr(tinyrexxParser::A_exprContext * ctx) {
    // controllo in quale caso sono
    if(ctx->ID() != NULL) {
        // caso ID semplice
    } else if(ctx->NUMBER() != NULL) {
        // caso valore numerico semplice
    } else if(ctx->MINUS() !=  NULL) {
        // caso operatore - unario
    } else if(ctx->a_op() != NULL) {
        // caso operatore binario: gestito da exitA_op
    } else {
        // caso parentesi
        cout << ")" ;
    }
}
```

È importante notare che nel caso in cui `ctx->a_op()` sia diverso da `NULL` i metodi `enterA_expr` e `exitA_expr` non fanno nulla, e lasciano che sia `exitA_op` a stampare l'operatore aritmetico:

```
void MyListener::exitA_op(tinyrexxParser::A_opContext * ctx) {
    // controllo operatore aritmetico
    if(ctx->PLUS() != NULL) {
        cout << " + ";
    } else if(ctx->MINUS() != NULL) {
```

```
        cout << " - ";
    } else if(ctx->MUL() != NULL) {
        cout << " * ";
    } else if(ctx->DIV() != NULL) {
        cout << " / ";
    }
}
```

I metodi `enterPrint` e `exitPrint` si occupano delle istruzioni di stampa, delegando ad `enterA_expr`, `exitA_expr` e `exitA_expr` la stampa delle espressioni aritmetiche come nel caso dell'assegnamento:

```
void MyListener::enterPrint(tinyrexxParser::PrintContext * ctx) {
    cout << string(indent, ' ') << "cout << " ;
}

void MyListener::exitPrint(tinyrexxParser::PrintContext * ctx) {
    cout << " << endl;" << endl;
}
```

Infine, consideriamo la regola per il ciclo `while`

```
w_loop    : 'do' 'while' test statement+ 'end' ;
```

Per questa regola, il metodo `enterW_loop` stampa la parola chiave `while` ed incrementa l'indentazione:

```
void MyListener::enterW_loop(tinyrexxParser::W_loopContext * ctx){
    cout << string(indent, ' ') << "while";
    indent += 4;
}
```

mentre `exitW_loop` stampa la parentesi graffa di chiusura e diminuisce l'indentazione:

```
void MyListener::exitW_loop(tinyrexxParser::W_loopContext * ctx){
    indent -= 4;
    cout << string(indent, ' ') << "}" << endl;
}
```

La stampa delle parentesi tonde che racchiudono la guardia del `while` e della parentesi graffa di apertura del corpo viene fatta da `enterTest` e `exitTest`:

```
void MyListener::enterTest(tinyrexxParser::TestContext * ctx){
    cout << "(";
}

void MyListener::exitTest(tinyrexxParser::TestContext * ctx){
    cout << ") {" << endl;
}
```

Il metodo `exitR_op` stampa l'operatore di confronto corretto, in modo simile a `exitA_op`:

```
void MyListener::exitR_op(tinyrexxParser::R_opContext * ctx) {
    // controllo operatore aritmetico
    if(ctx->EQUAL() != NULL) {
        cout << " == ";
    } else if(ctx->LT() != NULL) {
        cout << " < ";
    } else if(ctx->LEQ() != NULL) {
        cout << " <= ";
    } else if(ctx->GT() != NULL) {
        cout << " / ";
    } else if(ctx->GEQ() != NULL) {
```

```
        cout << " >= ";
    }
}
```

7 Completiamo il traduttore

Il file `translate.cpp` contiene il codice del `main` per il traduttore da TinyREXX 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 raccogliendo tutti gli `ID` presenti nel codice TinyREXX. Questo insieme viene utilizzato dal `MyListener` per dichiarare le variabili all'inizio della traduzione C++. Dopo aver creato un'istanza del listener 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 "tinyrexxLexer.h"
#include "tinyrexxParser.h"
#include "MyListener.h"

using namespace std;
using namespace antlr4;

int main(int argc, char* argv[]) {
    if(argc != 2) {
        cout << "Usage: translate filename.rexx" << endl;
        return 1;
    }
    ifstream tinyrexxFile(argv[1]);
    if(tinyrexxFile.fail()){
        //Errore nell'apertura del file
        cout << "Error while reading file " << argv[1] << endl;
        return 1;
    }
    ANTLRInputStream input(tinyrexxFile);
    tinyrexxLexer lexer(&input);
    CommonTokenStream tokens(&lexer);
    tinyrexxParser parser(&tokens);
    tree::ParseTree *tree = parser.program();
    int errors = parser.getNumberOfSyntaxErrors();
    if(errors > 0) {
        cout << errors << " syntax errors found, aborting." << endl;
        return 1;
    }
    // create set of IDs
    tokens.fill();
    std::set<std::string> ids;
    for (auto token : tokens.getTokens()) {
        if(token->getType() == parser.ID) {
            ids.insert(token->getText());
        }
    }

    MyListener listener(ids);
    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/tinyrexx$ ./translate example2.rexx
```

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#