Laboratorio di Automi e Linguaggi Formali
Terza Esercitazione

Davide Bresolin

a.a. 2020/2021

Il file `lab03.zip` contiene la grammatica completa del linguaggio Pascal, il codice del syntax checker visto nella seconda lezione di laboratorio, ed il codice dell'interprete descritto in questo tutorial. Anche in questo caso è incluso un `Makefile` per compilare il codice. Per maggiori informazioni sull'installazione di ANTLR v4, del runtime C++ e sul funzionamento del `Makefile` si può fare riferimento al tutorial della seconda esercitazione.

1 Grammatica completa

Il file `pascal.g4` contiene la grammatica che definisce il sottoinsieme del linguaggio Pascal rilevante per l'interprete:

```
grammar pascal;

start     : 'program' ID ';' 'var' decl_list main_code EOF ;

decl_list : decl | decl decl_list ;
decl      : id_list ':' 'integer' ';' ;
id_list   : ID | ID ',' id_list ;

main_code : 'begin' st_list 'end' '.' ;
code_block: statement | 'begin' st_list 'end' ;
st_list   : statement ';' | statement ';' st_list  ;

statement : assign | branch | out | loop | in ;

assign    : ID ':=' expr ;
out       : 'writeln' '(' expr ')' | 'writeln' '(' STRING ')' ;
in        : 'readln' '(' ID ')' ;
branch    : 'if' guard 'then' code_block
          | 'if' guard 'then' code_block 'else' code_block;
loop      : 'repeat' st_list 'until' guard ;
expr      : NUMBER | ID | expr PLUS expr | expr MINUS expr
          | expr MULT expr | expr DIV expr | expr MOD expr | '(' expr ')';
guard     : relation | NOT guard | guard AND guard | guard OR guard | '(' guard ')' ;
relation  : expr LT expr | expr LEQ expr | expr EQ expr
          | expr NEQ expr | expr GEQ expr | expr GT expr ;

PLUS      : '+' ;
MINUS     : '-' ;
MULT      : '*' ;
DIV       : '/' ;
MOD       : '%' ;

AND       : 'and' ;
OR        : 'or' ;
NOT       : 'not' ;
```

```
EQ        : '=' ;
LT        : '<' ;
LEQ       : '<=' ;
GT        : '>' ;
GEQ       : '>=' ;
NEQ       : '<>' ;
ID        : [a-z]+ ;
NUMBER    : [0-9]+ ;
STRING    : '\'' .*? '\'';
R_COMMENT : '(*' .*? '*)' -> skip ;     // .*? matches anything until the first */
C_COMMENT : '{' .*? '}' -> skip ;       // .*? matches anything until the first }
LINE_COMMENT : '//' ~[\r\n]* -> skip ;  // ~[\r\n]* matches anything but \r and \n
WS        : [ \n\t\r]+ -> skip;
ErrorChar : . ;
```

2 Creazione di un Visitor

L'interprete sfrutta un *visitor* per visitare l'albero sintattico del programma Pascal ed eseguirlo. Il visitor è implementato nella classe `runtimeVisitor` che estende l'interfaccia definita dalla classe `pascalBaseVisitor` creata da ANTLR. La definizione è contenuta nel file `runtimeVisitor.h`:

```
#pragma once

#include "antlr4-runtime.h"
#include "pascalVisitor.h"
#include "pascalBaseVisitor.h"

/**
 * This class provides a concrete visitor which only needs to handle a subset of the available metho
 */
class  runtimeVisitor : public pascalBaseVisitor {
protected:
    std::map<std::string,int> vars;

public:

  antlrcpp::Any visitId_list(pascalParser::Id_listContext *ctx);

  antlrcpp::Any visitAssign(pascalParser::AssignContext *ctx);

  antlrcpp::Any visitOut(pascalParser::OutContext *ctx);

  antlrcpp::Any visitIn(pascalParser::InContext *ctx);

  antlrcpp::Any visitBranch(pascalParser::BranchContext *ctx);

  antlrcpp::Any visitLoop(pascalParser::LoopContext *ctx);

  antlrcpp::Any visitExpr(pascalParser::ExprContext *ctx);

  antlrcpp::Any visitGuard(pascalParser::GuardContext *ctx);

  antlrcpp::Any visitRelation(pascalParser::RelationContext *ctx);

};
```

La classe contiene un attributo privato `vars` che serve per gestire le variabili usate nel programma Pascal. `vars` è una mappa che ha come chiave una stringa (`ID` della variabile) e come valore un intero (valore assegnato alla variabile). I metodi stabiliscono quello che deve fare il visitor quando visita un certo nodo nell'albero sintattico del programma, e corrispondono alle regole della grammatica. L'implementazione dei metodi si trova nel file `runtimeVisitor.cpp`.

La dichiarazione di una variabile viene implementata nel metodo `visitId_list`, che corrisponde alla regola

```
id_list   : ID | ID ',' id_list ;
```

della grammatica. Il metodo deve essere in grado di sapere qual'è il nome della variabile da definire (token `ID`). Questa informazione è presente nel parametro `ctx` del metodo, che punta ad un oggetto di tipo `pascalParser::Id_listContext` che rappresenta il contesto in cui viene applicata la regola `id_list`. La definizione del contesto si trova nel file `pascalParser.h`, generato automaticamente da ANTLR, ed è la seguente:

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


    virtual antlrcpp::Any accept(antlr4::tree::ParseTreeVisitor *visitor) override;

  };
```

In questo caso siamo interessati al metodo `ID()` che ci permette di accedere all'informazione sul nodo terminale di tipo `ID`, ed al metodo `id_list()` che ritorna un puntatore al resto della lista di identificatori (nel caso di dichiarazioni multiple).

L'implementazione di `visitId_list` ottiene il nome della variabile da assegnare, quindi procede aggiungendo un nuovo elemento alla mappa `vars`, usando come chiave il nome della variabile e come valore `0` (si assume che una variabile abbia valore `0` al momento della dichiarazione). Se la variabile è già presente nella mappa `vars`, il metodo stampa un messaggio d'errore sullo schermo e termina l'esecuzione dell'interprete. Se il puntatore ritornato da `id_list()` non è `NULL`, si prosegue richiamando ricorsivamente `visitId_list` sul resto della lista di identificatori, altrimenti si termina ritornando un valore `NULL`.

```
antlrcpp::Any runtimeVisitor::visitId_list(pascalParser::Id_listContext *ctx) {
    // Prima variabile nella lista di identificatori
    string varname = ctx->ID()->getText();
    // Resto della lista
    pascalParser::Id_listContext *tail = ctx->id_list();
    // controllo che la variabile non sia già stata dichiarata
    if(this->vars.find(varname) != this->vars.end()) {
        cerr << "Error: Duplicate variable declaration '" << varname << "'" << endl;
        exit(EXIT_FAILURE);
    }
    this->vars[varname] = 0;
    // Continua sul resto della lista di identificatori, se presente
    if(tail != NULL)
        return visitId_list(tail);
    return NULL;
}
```

Il metodo `visitAssign` si occupa della regola per l'assegnamento di una variabile:

```
assign    : ID ':=' expr ;
```

In maniera analoga al caso precedente, il nome della variabile da assegnare si può ottenere da `ctx->ID()`. Se la variabile non è stata dichiarata, il metodo stampa un messaggio d'errore sullo schermo e termina l'esecuzione dell'interprete. Il valore da assegnare alla variabile è dato dal valore dell'espressione posta dopo il simbolo di uguale. Per ottenere questo valore, il metodo `visitAssign` richiama il metodo `visitExpr` sul contesto `ctx->expr()` e poi aggiorna `vars` con il nuovo valore della variabile.

```
antlrcpp::Any runtimeVisitor::visitAssign(pascalParser::AssignContext *ctx) {
    string varname = ctx->ID()->getText();
    int value = visitExpr(ctx->expr());
    this->vars[varname] = value;
    return NULL;
}
```

Il metodo `visitOut` richiama `visitExpr` per ottenere il valore da stampare sullo schermo:

```
antlrcpp::Any runtimeVisitor::visitOut(pascalParser::OutContext *ctx) {
    // verifico se devo stampare intero o stringa
    if(ctx->expr() != NULL) {
        // caso stampa intero
        int value = visitExpr(ctx->expr());
        cout << value << endl;
    }
    // TODO: implementare il caso stampa di una stringa
    return NULL;
}
```

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

```
expr      : expr MOD expr | expr DIV expr | expr MULT expr | expr MINUS expr | expr PLUS expr |
          | '(' expr ')' | NUMBER | ID ;
```

Il metodo `visitExpr` si occupa di valutare una espressione aritmetica. A differenza dei metodi precedenti, ritorna un valore di tipo `int` con il valore calcolato per l'espressione aritmetica. Il calcolo del valore procede ricorsivamente per casi, secondo l'operatore principale dell'espressione. Possiamo capire in quale caso della regola ci troviamo controllando quali sono i metodi che ritornano un valore diverso da `NULL` nell'oggetto `ctx`:

- se `ctx->NUMBER()` ritorna un valore non nullo, allora è stato applicato il primo caso della regola, e il valore dell'espressione è un numero costante;
- se `ctx->ID()` ritorna un valore non nullo, allora è stato applicato il secondo caso della regola, e il valore dell'espressione è il valore della variabile identificata da `ctx->ID()->getText()`. Se la variabile non è stata dichiarata, il metodo stampa un messaggio d'errore sullo schermo e termina l'esecuzione dell'interprete;
- se `ctx->PLUS()` ritorna un valore non nullo, allora è stato applicato il quarto caso della regola, e l'espressione è una somma di due sottoespressioni. Si procede ricorsivamente richiamando `visitExpr` sulle due sottoespressioni e ritorndano la loro somma;
- i casi in cui `ctx->MINUS()`, `ctx->MULT()`, `ctx->DIV()` o `ctx->MOD()` sono non nulli vengono gestiti come la somma, cambiando l'operazione da fare;
- se tutti i metodi precedenti ritornano valore nullo, allora è stato applicato l'ultimo caso della regola, e l'espressione è racchiusa tra parentesi. Si procede ricorsivamente richiamando `visitExpr` sulla espressione senza le parentesi.

```
antlrcpp::Any runtimeVisitor::visitExpr(pascalParser::ExprContext *ctx) {
    // controllo in quale caso sono
    int value = 0;
    if(ctx->ID() != NULL) {
        // caso ID
```

```
        string varname = ctx->ID()->getText();
        if(this->vars.find(varname) == this->vars.end()) {
            cerr << "Error: Undefined variable '" << varname << "'" << endl;
            exit(EXIT_FAILURE);
        }
        value = this->vars[varname];
    } else if(ctx->NUMBER() != NULL) {
        // caso NUMBER
        string numtext = ctx->NUMBER()->getText();
        value = std::stoi(numtext);
    } else if(ctx->PLUS() != NULL) {
        // caso expr + expr
        // calcolo il valore della prima expr
        int left = visitExpr(ctx->expr(0));
        // calcolo il valore della seconda expr
        int right = visitExpr(ctx->expr(1));
        // il risultato finale è la somma
        value = left + right;
    } else if(ctx->MINUS() != NULL) {
        // caso expr - expr
        // calcolo il valore della prima expr
        int left = visitExpr(ctx->expr(0));
        // calcolo il valore della seconda expr
        int right = visitExpr(ctx->expr(1));
        // il risultato finale è la differenza
        value = left - right;
    } else if(ctx->MULT() != NULL) {
        // caso expr * expr
        // calcolo il valore della prima expr
        int left = visitExpr(ctx->expr(0));
        // calcolo il valore della seconda expr
        int right = visitExpr(ctx->expr(1));
        // il risultato finale è la moltiplicazione
        value = left * right;
    } else if(ctx->DIV() != NULL) {
        // caso expr / expr
        // calcolo il valore della prima expr
        int left = visitExpr(ctx->expr(0));
        // calcolo il valore della seconda expr
        int right = visitExpr(ctx->expr(1));
        // il risultato finale è la divisione
        value = left / right;
    } else if(ctx->MOD() != NULL) {
        // caso expr % expr
        // calcolo il valore della prima expr
        int left = visitExpr(ctx->expr(0));
        // calcolo il valore della seconda expr
        int right = visitExpr(ctx->expr(1));
        // il risultato finale è il modulo
        value = left % right;
    } else if(ctx->expr().size() == 1) {
        // caso parentesi
        value = visitExpr(ctx->expr(0));
    }
    return value;
}
```

Il metodo `visitBranch` si occupa di implementare il costrutto `if`. Richiama `visitGuard` per stabilire se la guardia è vera o falsa. In caso positivo chiama `visitCode_block` per eseguire il ramo "then",

identificato da `ctx->code_block(0)`.

```
antlrcpp::Any runtimeVisitor::visitBranch(pascalParser::BranchContext *ctx) {
    // stabilisce il valore della guardia
    bool guard = visitGuard(ctx->guard());
    if(guard) {
        // se guardia vera, esegue ramo then
        visitCode_block(ctx->code_block(0));
    }
    // TODO: implementa l'esecuzione del ramo else (se presente) quando la guardia è falsa
    return NULL;
}
```

Possiamo notare che il metodo `visitCode_block` non è definito esplicitamente dalla classe `runtimeVisitor`. Di conseguenza, il codice esegue il metodo `visitCode_block` ereditato dalla classe padre `pascalBaseVisitor`:

```
  virtual antlrcpp::Any visitCode_block(pascalParser::Code_blockContext *ctx) override {
    return visitChildren(ctx);
  }
```

Questo metodo predefinito visita ricorsivamente tutti i figli del nodo `code_block`, eseguendo i metodi di visita corrispondenti, ed eseguendo il blocco di codice corrispondente al ramo then.

I metodi `visitGuard` e `visitRelation` non sono implementati e si limitano a ritornare il valore `true`. Allo stesso modo, il metodo `visitLoop` ritorna `NULL`.

3 Completiamo il traduttore

Il file `pascal.cpp` contiene il codice del `main` per l'interprete TinyC. 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 creando un'istanza del visitor e richiamando il metodo `visitStart` per iniziare l'esecuzione del programma.

```
#include <iostream>
#include <fstream>
#include <string>
#include <exception>
#include "antlr4-runtime.h"
#include "pascalLexer.h"
#include "pascalParser.h"
#include "runtimeVisitor.h"

using namespace std;
using namespace antlr4;

int main(int argc, char* argv[]) {
    if(argc != 2) {
        cout << "Usage: pascal filename.tc" << endl;
        return 1;
    }
    ifstream pascalFile(argv[1]);
    if(pascalFile.fail()){
        // File open error
        cout << "Error while reading file " << argv[1] << endl;
        return 1;
    }
    ANTLRInputStream input(pascalFile);
    pascalLexer lexer(&input);
    CommonTokenStream tokens(&lexer);
    pascalParser parser(&tokens);
```

```
    pascalParser::StartContext *tree = parser.start();
    int errors = parser.getNumberOfSyntaxErrors();
    if(errors > 0) {
        cout << errors << " syntax errors found." << endl;
        return 1;
    }

    cout << "Running program " << argv[1] << endl;
    runtimeVisitor visitor;
    visitor.visitStart(tree);
    return 0;
}
```

Dopo aver compilato l'inteprete con il comando `make pascal` possiamo provare l'esecuzione sui programmi di esempio:

```
dbresoli@t68:~/pascal$ ./pascal tests/valid.pas
```

ottenendo l'output seguente:

```
Running program tests/valid.pas
12
4
```

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#
- `https://tomassetti.me/getting-started-antlr-cpp/` una introduzione all'uso di ANTLR con C++ dallo stesso autore del tutorial precedente.