Transformative Code Generation

Julian Rathke
jr2@ecs.soton.ac.uk
Automated Code Generation

= 
Domain Knowledge Representation 
+ 
Metaprogramming
The Generative Approach

• So far we have focussed on generative code generation
  • Code is assembled from fragments expressed in some (high-level) programming language
  • Representation of the domain knowledge is encoded as code annotations or meta-programming constructs
    • e.g. pointcut descriptors, templates
    • It becomes part of the meta-programming
The Transformative Approach

• Knowledge is represented in some input high-level domain model

• Rules are given to transform that knowledge from the high-level domain model to low(er) level models / source code / software artefacts

• To implement this we need to think about
  • The domain model itself
  • How to express the transform rules
  • How to actually build the transformation engine

• Let’s consider the domain model first
Domain Models

• What actually is a domain model?
  • Fundamentally, it is a data structure that represents the domain knowledge
  • It is often tree-shaped
  • This doesn’t seem that radical - do we actually need any tool support?
  • Let’s think about how the domain model is actually inputted
    • Usually this is done Graphically or Textually
Graphical vs Textual

- Graphical Domain Models
  - The model instances are given using some form of GUI
  - e.g. Eclipse Modelling Framework, SpecWare
  - User is directly building the model data structure in the back-end

- Textual Domain Models
  - Model instances are given using some language or fixed format
  - e.g. XML or JSON, DSLs
  - The model is built via parsing the input
  - Graphical models often have a textual form too
Parsing Support

• So, suppose we create ourselves a textual domain model, with some DSL to input the domain knowledge.
• In order to implement a transformation we will need to build a parser for that DSL
• Compiler writers have spent decades worrying about parsing
  • Yes, really
• We’ve now got some excellent tools to automatically generate parsers.
• We will look at one of them called Antlr (Version 4).
Brief overview of Antlr

• Antlr v4 is a modern, easy to use, parser generator
• Feed it a grammar, it generates you code for a lexer and a parser in a target language of your choice
  • Targets available: Java, C++, C#, Python, Javascript, Go, Swift
• An extra feature is the generation of tree walkers
• You can traverse the parsed data structures in a straightforward way
  • This is the part of Antlr that we’ll look at rather than parsing and grammars
• For more detail on grammar design see the Antlr textbook - *The Definitive Antlr 4 Reference* (Parr)
Starting with Antlr

• You should have installed Antlr 4 on your machines by now
• If not then visit the Antlr website and follow the instructions there
• Make sure you have downloaded the big directory of examples that come with the Antlr book too
• I’ll also assume that you have read the extract from the book called “Let’s get meta”
  • This tells us the basics of what parsing is
  • You should know what the following things mean
    • Tokens, Token Stream, Parse Tree
Antlr Tree Walkers

- There is support for two different approaches to tree walking in Antlr
  - Parse Tree Listeners
  - Parse Tree Visitors
- These are both quite similar in what can be achieved using them but that is not the point (you could write any iterator in the target language)
  - Both are provided because the two approaches are more convenient to use than each other in different scenarios
Parse Tree Listeners

• These use an event-driven model
• What are parsing “events” then?
• For parse rule R (e.g. R : '{' '|' expr '|' '}'), as the parser recognises text matching R it constructs a node in the parse tree and begins to calculate that node (e.g. data in the node, child nodes)
  • This event is called “enterR”
• There is also an event as the parser finishes constructing that node
  • This event is called “exitR”
Parse Tree Listeners

• By default, for each rule R, Antlr generates two methods / functions in the target language (one method for the exitR event and one for enterR).

• If Foo is the name of the input grammar, Antlr also generates interfaces for these methods:
  • a default interface called FooListener
  • that extends a fixed ParseTreeListener interface
  • and a default (adapter) implementation of FooListener called FooBaseListener

• To implement your own functionality, just override the appropriate methods in FooBaseListener
Example Time

• Consider the following simple (partial) grammar

```
stat : assign
    | ifstat
    | whilestat
;
assign : ID '=' expr ';'

expr : INT;
```

These are lexer non-terminals

These are non-terminals in the parser

These are terminals

These are lexer non-terminals

• Let’s look at the classes generated in the Parse Tree data structure for this
Assign Example

• For each non-terminal N in the parser there is a NContext class generated
Listener Traversal

- The Listener support includes a ParseTreeWalker class that iterates depth-first over the whole parse tree by calling methods in the generated classes.
Listener Traversal Method Calls

- These are the methods called and the order in which they are called by the default Walker.
Here is an Antlr grammar for parsing array initialiser values, for example

```
{ 1, 6, 7, 83, 12 } or { 3 } but not {} 
```

```java
grammar ArrayInit;

/** A rule called init that matches comma-separated values between {...}. */
init : '{' value (',' value)* '}' ;  // must match at least one value

/** A value can be either a nested array/struct or a simple integer (INT) */
value : init
    | INT
    ;

// parser rules start with lowercase letters, lexer rules with uppercase
INT :   [0-9]+ ;             // Define token INT as one or more digits
WS  :   [ \t\r\n]+ -> skip ; // Define whitespace rule, toss it out
```
ArrayInit Example

• The idea is to use Listeners to translate the array initialisers into hexadecimal string form - to avoiding a limitation in Java initialisation methods
• What listener methods should we implement?
• Look at the parse tree:

We need to produce the string “\u0001\u0002\u0003” - including the quotes
ArrayInit Example

- We can see there are events `enterInit`, `exitInit`, `enterValue`, `exitValue`

- We need to implement the first three:
  - On `enterInit`, we print double quotes (to open)
  - On `exitInit` we print double quotes (to close)
  - On `enterValue` we print the hex value - this will get called multiple times
The generated files

• If we run Antlr on the ArrayInit grammar then the following files are generated (Java target):

• Have a look at these in the generated code
• We want ArrayInitBaseListener.java
/***
 * Excerpted from "The Definitive ANTLR 4 Reference",
 ... 
 /** Convert short array inits like {1,2,3} to \\\\u0002\\u0003 */
 public class ShortToUnicodeString extends ArrayInitBaseListener {
    /** Translate { to " */
    @Override
    public void enterInit(ArrayInitParser.InitContext ctx) {
        System.out.print('"');
    }

    /** Translate } to " */
    @Override
    public void exitInit(ArrayInitParser.InitContext ctx) {
        System.out.print('"');
    }

    /** Translate integers to 4-digit hexadecimal strings prefixed with \u */
    @Override
    public void enterValue(ArrayInitParser.ValueContext ctx) {
        // Assumes no nested array initializers
        int value = Integer.valueOf(ctx.INT().getText());
        System.out.printf("\u%04x", value);
    }
}

Easy

Let’s look more closely
EnterValue method

```java
/*
 ** Translate integers to 4-digit hexadecimal strings prefixed with \\u */

@Override
public void enterValue(ArrayInitParser.ValueContext ctx) {
    // Assumes no nested array initializers
    int value = Integer.valueOf(ctx.INT().getText());
    System.out.printf("\\u%04x", value);
}
```

Print the hex value

Fetch the child (terminal) node of the Value node

Fetch the string (char stream) content of the node

The node for the current parser match

How would we solve this for nested initialiser values ???
Sledgehammer meet Nut

• We can easily argue that this solution is far too overblown for the problem:
  • A straightforward Java solution would be easy
  • Also, there are problems with nested initialisers using Listeners - how do I identify the exit of the top level init call?
  • Would need to introduce a “depth” counter and suitable delimiters in the strings to indicate depth.
• Fair points - but we can see that no traversal code was required by us - no real knowledge of the parse trees is needed other than the names of the non-terminals in the grammar.
A Java code transformation example

- Consider the following source to source code transformation problem:
  - Automatically generate a Java interface from any given Java source code class definition
- We could use the Java reflection API or some tool
- But we can also use Antlr Listeners to great effect here - with a very concise solution
- Why? Because Antlr is distributed with a full Java grammar! So we can parse Java source code into parse trees and manipulate it.
  - There are plenty of other full grammars available for other languages too
The Java grammar

• File java.g4 in the examples directory (subdirectory tour) contains the full Java grammar in Antlr syntax
• The bits we need to see are

```plaintext
classDeclaration
  :   'class' Identifier typeParameters? ('extends' type)?
     ('implements' typeList)?
     classBody
     ;

methodDeclaration
  :   type Identifier formalParameters ('[' ']')* methodDeclarationRest
  |   'void' Identifier formalParameters methodDeclarationRest
  ;
```
The event methods

• This tells us that there will be event listener methods with the corresponding names for these non-terminals

• In the generated code for JavaListener we have:

```java
public interface JavaListener extends ParseTreeListener {
    void enterClassDeclaration(JavaParser.ClassDeclarationContext ctx);
    void exitClassDeclaration(JavaParser.ClassDeclarationContext ctx);
    void enterMethodDeclaration(JavaParser.MethodDeclarationContext ctx);
    void exitMethodDeclaration(JavaParser.MethodDeclarationContext ctx);
}
```

• We can solve the problem by implementing the first three of these:
  • For the enterClassDeclaration method we simply output the interface declaration, name and block delimiters ‘{‘ and ‘}’ in the exit method of course.
  • For the enterMethodDeclaration we need to get the return type and formal parameters
The Listener Code Pt1

The enterClass / exitClass methods are straightforward. But look at how to extract the class Name.

```java
public class ExtractInterfaceListener extends JavaBaseListener {
    JavaParser parser;
    public ExtractInterfaceListener(JavaParser parser) {this.parser = parser;}
    /** Listen to matches of classDeclaration */
    @Override
    public void enterClassDeclaration(JavaParser.ClassDeclarationContext ctx){
        System.out.println("interface I" + ctx.Identifier() + " {");
    }
    @Override
    public void exitClassDeclaration(JavaParser.ClassDeclarationContext ctx) {
        System.out.println("}");
    }
}
```

We will need to access the token stream.

The Context object has accessor methods for each non-terminal in the parsing rule that is currently matched. i.e. the child nodes in the parse-tree.
@Override
public void enterMethodDeclaration(
    JavaParser.MethodDeclarationContext ctx
)
{
    // need parser to get tokens
    TokenStream tokens = parser.getTokenStream();
    String type = "void";
    if ( ctx.type()!=null ) {
        type = tokens.getText(ctx.type());
    }
    String args = tokens.getText(ctx.formalParameters());
    System.out.println("\t"+type+" " + ctx.Identifier()+args+";");
}
The Visitor Pattern

• Let’s remind ourselves of the well-known Visitor Software Design Pattern

• There is an Visitor interface containing the visit method

• This method calls accept on the appropriate type of node and passes the visitor object for callbacks
Visitor Pattern Example

Diagram from tutorialpoint.com
Visitors in Antlr

- Antlr provides support for using Visitors to traverse Parse Trees.
- When running Antlr using the -visitor flag will cause the Visitor interface for that parse tree to be generated.
- This lists the visit method for each node type
- The interface is parametric in the return type of the visit methods
- Antlr also generates a default visitor implementation class - the BaseVisitor.
  - For each node, this will automatically visit each node in the parse tree and pass the return value back
grammar Expr;

/** The start rule; begin parsing here. */
prog:   stat+ ;

stat:   expr NEWLINE
|   ID '==' expr NEWLINE
|   NEWLINE
;

eexpr:  expr ('*' | '/' ) expr
|   expr ('+' | '-' ) expr
|   INT
|   ID
|   '(' expr ')' 
;

ID  :   [a-zA-Z]+ ;      // match identifiers
INT :   [0-9]+ ;         // match integers
NEWLINE: '\r'? '\n' ;   // return newlines to parser (is end-statement signal)
WS  :   [ \t]+ -> skip ; // remove whitespace
Example: Arithmetic Expressions

- Think about how we might implement a simple calculator for these expressions using a Listener.

- One problem is that there is only one `enterExpr` event for each type of expression - i.e. addition, multiplication etc - and INTs!

- To code this, we will need to dispatch on the operation (`n` or `ID` or `+,-,*,/`) in the `enterExpr` method.
  - You can do this but it isn’t particularly elegant

- Another problem is how do you return the value of subexpressions up to the calculation of the larger expression?

- e.g. `( 5 + 6 ) * (9 - 2) - needs to return 11 and 7 to the multiplication
Example: Arithmetic Expressions

- We can address the latter problem using Visitors - the visitor methods can be used to traverse the tree **and** return values along the way.
- To address the former problem we need a way of identifying the different cases in the grammar so that we can visit (or Listen to them) differently.
- Antlr provides support for this in the form of Labels
- If you label a line in the grammar using #foo then a visit method, or enter/exit method is generated for that sub-rule in the grammar
Example: Labelled Expressions

grammar LabeledExpr; // rename to distinguish from Expr.g4

prog: stat+ ;

stat:  expr NEWLINE          # printExpr
     |  ID '=' expr NEWLINE     # assign
     |  NEWLINE                  # blank
     ;

expr:  expr op=('*'|'/') expr  # MulDiv
     |  expr op=('+'|'-') expr  # AddSub
     |  INT                     # int
     |  ID                      # id
     |  '(' expr ')'           # parens
     ;

MUL :  '*' ; // assigns token name to '*' used above in grammar
DIV :  '/' ;
ADD :  '+' ;
SUB :  '-' ;
ID :  [a-zA-Z]+ ;       // match identifiers
INT :  [0-9]+ ;         // match integers
NEWLINE: '\r'? '\n' ;  // return newlines to parser (is end-statement signal)
WS :  [ \t]+ -> skip ;  // remove whitespace
Now, if we run Antlr on this labelled grammar we get methods for each label rather than each rule - let’s see this for visit methods:

```java
public interface LabeledExprVisitor<T> extends ParseTreeVisitor<T> {

    ...
    T visitMulDiv(LabeledExprParser.MulDivContext ctx);
    T visitAddSub(LabeledExprParser.AddSubContext ctx);
    T visitInt(LabeledExprParser.IntContext ctx);
    T visitId(LabeledExprParser.IdContext ctx);
    T visitParens(LabeledExprParser.ParensContext ctx);

}
```

We can implement these separately by overriding the separate visit methods in the LabeledExprBaseVisitor.
Implementing an Evaluation Visitor

```java
import java.util.HashMap;
import java.util.Map;

public class EvalVisitor extends LabeledExprBaseVisitor<Integer> {
    /** "memory" for our calculator */
    Map<String, Integer> memory = new HashMap<String, Integer>();

    /** ID '=' expr NEWLINE */
    @Override
    public Integer visitAssign(LabeledExprParser.AssignContext ctx) {
        String id = ctx.ID().getText();  // ID is left-hand side of '=
        int value = visit(ctx.expr());   // compute value of expression on right
        memory.put(id, value);           // store it in our memory
        return value;
    }

    /** expr NEWLINE */
    @Override
    public Integer visitPrintExpr(LabeledExprParser.PrintExprContext ctx) {
        Integer value = visit(ctx.expr()); // evaluate the expr child
        System.out.println(value);         // print the result
        return 0;                          // return dummy value
    }
}
```

Return type of visit methods

You must explicitly visit the next node
/** INT */
@Override
public Integer visitInt(LabeledExprParser.IntContext ctx) {
    return Integer.valueOf(ctx.INT().getText());
}

/** ID */
@Override
public Integer visitId(LabeledExprParser.IdContext ctx) {
    String id = ctx.ID().getText();
    if (memory.containsKey(id)) return memory.get(id);
    return 0;
}

/** '(' expr ')' */
@Override
public Integer visitParens(LabeledExprParser.ParensContext ctx) {
    return visit(ctx.expr()); // return child expr's value
}
```java
/** expr op=('*' | '/' ) expr */
@Override
public Integer visitMulDiv(LabeledExprParser.MulDivContext ctx) {
    int left = visit(ctx.expr(0));  // get value of left subexpression
    int right = visit(ctx.expr(1)); // get value of right subexpression
    if ( ctx.op.getType() == LabeledExprParser.MUL ) return left * right;
    return left / right; // must be DIV
}

/** expr op=('+' | '-' ) expr */
@Override
public Integer visitAddSub(LabeledExprParser.AddSubContext ctx) {
    int left = visit(ctx.expr(0));  // get value of left subexpression
    int right = visit(ctx.expr(1)); // get value of right subexpression
    if ( ctx.op.getType() == LabeledExprParser.ADD ) return left + right;
    return left - right; // must be SUB
}
```
Observations on Listeners and Visitors

- Listeners automatically traverse the tree in a fixed order but Visitors are traversed under your control.
- e.g. The JavaExtractInterface example had no need to traverse the tree below the methodDeclaration nodes
- Can be more efficient to use Visitors
- But Listener code can be shorter
- Returning values up the tree is easy with a Visitor
- But only for a single value of a single type!
- And Antlr allows parse tree nodes to store values too
Expressions with a node value

grammar Expr;

/** The start rule; begin parsing here. */
prog:   stat+ ;

stat:   expr NEWLINE
   |   ID '=' expr NEWLINE
   |   NEWLINE
   ;

expr returns [int value]:   expr ('*'|'/') expr
   |   expr ('+'|'-') expr
   |   INT
   |   ID
   |   '(' expr ')' ;

Again, this is restricted to a single value. Another solution is to use a HashMap of nodes to values. In fact, there is a built-in support for this with the ParseTreeProperty<T> class - it hashes nodes to values of generic type T.

This injects a field named value in to the class that represents expr nodes in the Parse Trees i.e. ExprContext

void enterExpr(ExprContext ctx)

and

void exitExpr(ExprContext ctx)

have access to this field via ctx.
Did we forget something in the JavaExtractListener example?

• We should really provide the import declarations that the class uses in the interface also.
  • This isn’t too hard - implement `enterImportDeclaration()` and print out the text for the matching node.

• We should really make the input class declare that it implements the generated Listener.
  • This is trickier - we need to modify the input source code
  • This can be done by modifying the Token Stream during the parsing process
TokenStreamRewriter

- A TokenStreamRewriter object can provide altered views of a token stream
  - It doesn’t actually modify the input stream but just keeps tracks of edits to be made when rendering a token stream back to text
  - You can delete, insert, and replace tokens from the original stream
  - After editing you can call getText to see the altered view of the stream.
  - Inside enterClassDeclaration we can use

```java
JavaParser.ClassBodyContext cb = ctx.classBody();
rewriter.insertBefore(cb.start, "implements I"+ctx.Identifier() + " ");
```

where `rewriter` is a TokenStreamRewriter object for the input Token Stream
Tree to Tree Transformation?

- Although Antlr is great for expressing grammars and generating parsers,
- And great for generating walkers for parse trees
- The support for Transforming from parse trees to ASTs is not really there.
- To generate code from Antlr Listeners or Visitors you essentially are
  - Writing strings, or
  - Rewriting Token Streams, or
  - Manually building AST nodes of the generator target
- The latter could really do with improving!
Summary

- That's about all I wanted to show for Transformative Code Generation
- Support for generalised input and traversal over the model structures is key to this approach
- Antlr is not really a transformation tool in itself but it is meets the above points well for textual domain models
- Another really interesting tool for transforming textual domain models is the Stratego tool (now part of Spoofax).
- For more graphical approaches, such as EMF there are a bunch of model to transformation tools
  - e.g. Acceleo, JET, Epsilon, ATL, Xpand