ECOTE – Final Documentation

Semester:

Author: Robert Dwornik

Subject: Write a program that converts a code with lambda expression into a code without lambda expressions. The code is subset of Java.

General overview and assumptions

Write a program that converts a code with lambda expression into a code without lambda expressions. The code is subset of Java.

The programs aim is to recognize lambda expression and convert it into new file with either anonymous inner class or simple function with auto-generated name. The programs should recognize types of lambda assignment according to Java 8.

In our project we use ANTLR v4 tool to parse input according to our grammar.

Programs should recognizes incorrect syntax and inform user about it with proper error message.

Program on output should be possible to compile after conversion as it was on input.

To make our project possible to implement we have to make few assumptions.

- We assume that the name of the function is the same as the name of the instance identifier otherwise we are not sure if program on output will work.
- We also assume that interface was somewhere implemented in library or package or in the same file otherwise we are not sure if program on output will work.
- All parameters in lambda will be converted into Object type.

Functional requirements

(with algorithms at design level, detailed syntax of an input language if appropriate, transformation rules if appropriate, etc.)

File with *.java extension which will be converted is passed as argument in command line. File is processed firstly to check if it has no syntax errors before the conversion. Only one file can be passed as argument.

Syntax of input language should be acceptable according to Java 1.7 grammar. Our program will recognize one type of lambada assignment. The one with curly brackets. Other types of lambda assumption will be considered as an error.

Example
* *correct no error* *
(type variable) -> { * *function body* * };
* *error missing missing '{' at '}'* *
() -> 123.456;

Implementation

General architecture

1. Grammar

The most important file in our project is “Java.g4”. In this file we define grammar of our language. Parser rules start with lowercase letters, lexer rules with uppercase.

Example
1. This example of lexer rule accepts all possible characters and digits including ‘$’ and ‘_’.

```
JavaLetterOrDigit
    :   [a-zA-Z0-9$_]
;
```

2. This example of Parser Ruler matches **lambdaParameters** and **lambdaBody** rule separated by ‘->’. Every rule must finally much with the lexer.

```
lambdaExpression
    : lambdaParameters arrow lambdaBody

;
arrow
    : '->'
    ;
```

3. We use Java 1.7 grammar downloaded from Internet. This grammar does not handles lambdas it is why part was added by the programmer.

```
lambdaExpression
    : lambdaParameters arrow lambdaBody
    ;

arrow
    : '->'
    ;

lambdaParameters
    : '(' parameters? ')'
    ;

lambdaBody
    : block
    ;

parameters
    : param (folloParam)*
    ;

param
    : Identifier
    ;
folloParam
    : ',' Identifier
    ;
```

2. Classes

1. **ConvertLambdaExpr**
   Main class of the program. Handles all errors during conversion process. Takes input file from command line. Prints and writes the output int new file. It handles 3 important exceptions.
   1. Wrong file name.
   2. File is not runnable it means that user provides file that is not runnable by JVM.
   3. Errors thrown by the grammar.

   1.1 Variables

   a) *inputFile* - stores file name
   b) *cs* – stores character stream of input file
   c) *lexer* – stores a sequence of characters as a sequence of tokens
   d) *parser* – parser of our grammar
   e) *tree* – assign beginning rule for parsing
   f) *walker* - trigger listeners waiting for event
   g) *converter* – instance of ConvertLambdaExpListener
   h) *str* – instance of StrinBuilder which converts filename with *.java extenstion to *.txt extension (used to write errors to text file)

   1.2 Methods

a) *void runProcess(String inputFile)*- runs input file and throws error if process failed to run.
b) *void writeToFile(String data, String fileName)* – wirtes “data” to file

2. **ConvertLambdaExpListener** – Class is derives from JavaBaseListener and overrides it method according to our needs. Listener methods are phrase elements matched by the rules. We can override each method rule we want and it will be triggered by the walker. Methods are listeners they are triggered when match rule appears in parse tree.

2.1 Variables

a) *rewriter* – treats all of the manipulation methods as instruction and queues them when traversing the token stream to render it back as text. Rewriter executes those instruction every time we call getText().
b) *static String typeName* – stores object type as string.
c) *static String variableName* – stores instance name
d) *static StringBuilder parameters* – stores parameters in lambda

2.2 Methods

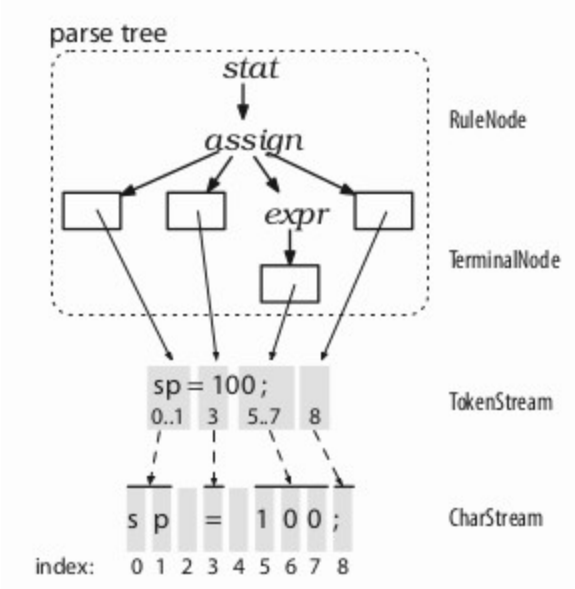
a) *ConvertLambdaExpListener(TokenStream tokens)* – constructor of the class creates *rewrite* instance from TokenStream of our input file.
b) *enterLambdaExpression(JavaParser.LambdaExpressionContext ctx)* – in this method we add interface initialization in place of '→', add closing bracket at and of anonymous method declaration, delete open and closing parenthesis.
c) *public void enterVariableDeclaratorId(JavaParser.VariableDeclaratorIdContext ctx)* - we enter this rule to get object type stored in this context and assign tho variable typeName.
d) *void enterClassOrInterfaceType(JavaParser.ClassOrInterfaceTypeContext ctx)* - we enter this rule to get instance name stored in this context and assign to variable variableName.
e) *void enterLambdaBody(JavaParser.LambdaBodyContext ctx)* – we add anonymous method declaration and clear parameters buffer so we not append parameters to previous ones
f) *public void enterParam(JavaParser.ParamContext ctx)* – append parameter to string buffer and delete it from the file.
g) *public void enterFolloParam(JavaParser.FolloParamContext ctx)* – append following parameter with coma to string buffer and delete it from the file.
h) *public void enterClassName(JavaParser.ClassNameContext ctx)* – append to main class Name “Converted” string.

3. **NotRunnableProcess –** Class extends exception class and it's implementing toString() method which informs about error during file compilation error.
   3.1. Variables
   a) *proc* **–** holds proccess which is was run from input file. We use it to get the error message.
   3.2. Methods
   a) *ToString()* **–** returns string with error message

4. **ThrowingErrorListener –** class extends BaseErrorListner class, which handles by default error in grammar, but it doesn't throw exception that we could catch. It is why we implement this class and which handler errors generated by parser and throws exception.
   4.1. Variables
   a) INSTANCE – stores "this" created instance
   4.2. Methods
   a) *void syntaxError(Recognizer<?, ?> recognizer, Object offendingSymbol, int line, int charPositionInLine, String msg, RecognitionException e)*
   – clears all buffers for safety and throws ParseCancellationException which contains the string with error message.

Data structures

According to compiler structure, lexers process characters and pass tokens to the parser, which in turn checks syntax and creates a parse tree. The corresponding ANTLR classes are CharStream , Lexer , Token , Parser , and ParseTree . The "pipe" connecting the lexer and parser is called a TokenStream. The diagram below illustrates how object of these types are connected to each other in memory.



Each non-terminal symbol (in our case each rule in grammar) is an context object, they record everything we know about the recognition of a phrase by a rule. Each context object knows the start and stop tokens for the recognized phrase. Since ANTLR uses tree data structure provides access to all of the elements of that phrase.

ANTLR provides support for two tree-walking mechanisms in its runtime library. One uses listeners second visitor design pattern. In my project I am using event listener mechanism and this one I will describe in this section.

To walk a tree and trigger calls into a listener, ANTLR's runtime provides class ParseTreeWalker . The methods in a listener are just callbacks. To make a language application, we build a ParseTreeListener implementation containing application-specific code that typically calls into a larger surrounding application. ANTLR generates a ParseTreeListener subclass specific to each grammar with enter and exit methods for each rule. As the walker encounters the node for rule assign , for example, it triggers enterAssign() and passes it the AssignContext parse-tree node. After the walker visits all children of the assign node, it triggers exitAssign() .

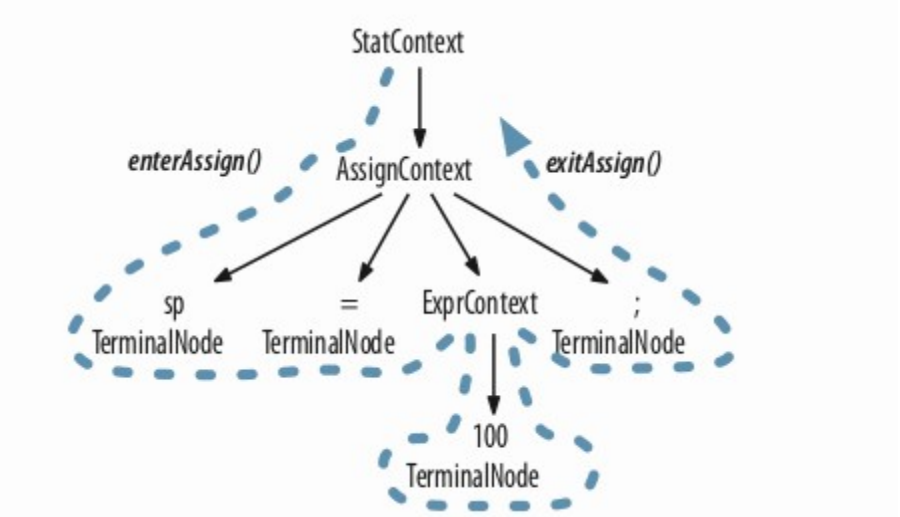
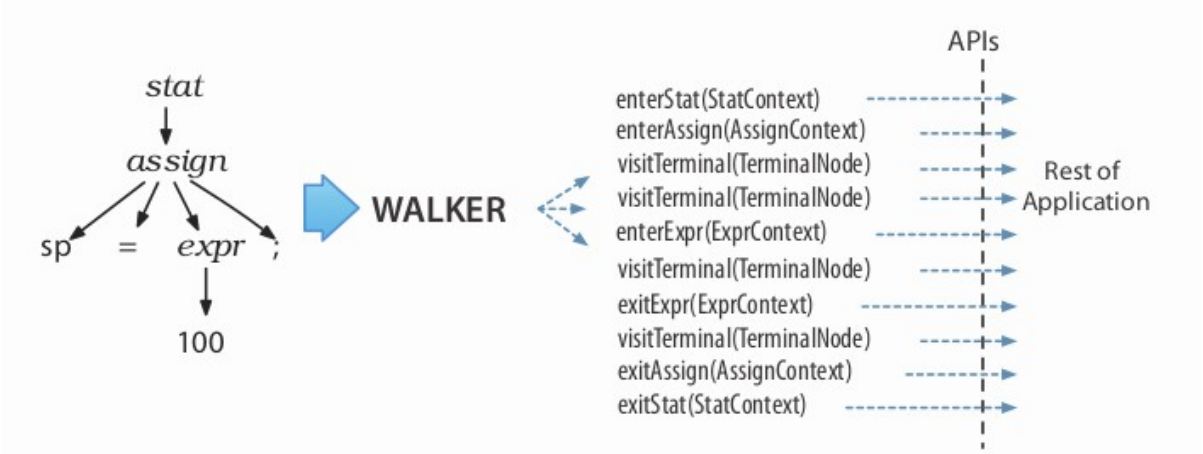


Diagram below shows sequence of triggered methods by the walker.



Module descriptions

When we run ANTLR on a grammar it automatically generates for us Parser and Lexer that recognizes sentences in the language described by the grammar. Additionally it creates also automatically listeners interface. ANTLR stores all of this in 5 files.

Name of the files is combination of name of the grammar and parts of our compiler.

JavaParser.java - This file contains the parser class definition specific to grammar Java.g4 that recognizes our array language syntax.
It contains a method for each rule in the grammar as well as some support code.

JavaLexer.java - ANTLR automatically extracts a separate parser and lexer specification from our grammar.

Java.tokens - ANTLR assigns a token type number to each token we define and stores these values in this file. It's needed when we split a large grammar into multiple smaller grammars so that ANTLR can synchronize all the token type numbers

JavaListener.java JavaBaseListener.java - By default, ANTLR parsers build a tree from the input. By walking that tree, a tree walker can trigger events (callbacks) to a listener object that we provide. JavaListener is the interface that describes the callbacks we can implement. JavaBaseListener is a set of empty default implementations which we inherit from. This class makes it easy for us to override just the callbacks we're interested in, otherwise we would have to implement all methods in JavaListeners interface.

Input/output description

1. Input

To run our program we have to start JVM by typing "java" command followed by our main class and input file.

Example

java ConvertLambdaExpr Demo.java

2. Output

On output will we obtain the content of generated ConvertedDemo.java file

Example

Terminal Output after running program:

```
public class ConvertedDemo{
    public static void main(String[] args) {
            Runnable r = new Runnable() {
                  public void run() {
                        System.out.println("Hello"); }
    }
}
```

Terminal Output after running program “ls” command:

Demo.class Demo.java Java.g4 src

Others

- To run our program it is important to have installed **antlr-4.7.2-complete.jar** library and put in the same folder with our main class.
- Reference:
  The Definitive ANTLR 4 Reference Author: Terence Parr