JavaCC Parser for Logo

Generation of JavaApplet

Project documentation in Module AFL

Field of Study:	Information Technology
Authors:	Emanuel Knecht (knece1@bfh.ch)
	David Aeschlimann (aescd5@bfh.ch)
Professor:	Olivier Biberstein
Date:	12.06.2015
Version:	Version 2.0

Table of Contents

1 Introduction

This document contains a description of the work done during the project for the class Automata and Formal Languages and the results that have been accomplished by our group.

1.1 Problem/Task

The objective of this project is to develop a parser/translator from a given subset of the Logo programming language into Java. The parser must be developed by means of JavaCC. The EBNF grammar was provided, as well as some examples of Logo programs, the primitives of Logo written in Java and a base ant project with a partly implemented JavaCC file. We are allowed to modify the grammar, but not to change the specification of the language. The developed parser/translator implements the provided grammar and must be able to run all the Logo programs provides as examples without modifications.

1.1.1 Logo

Logo is an educational programming language who was first released in 1967. It is mostly used for so called "Turtle Graphics". A turtle can be moved on a screen and controlled to paint objects in two dimensions. Its functionality is quite limited, but that makes it easy to learn and deliver immediate graphical feedback. That is why it is often used in schools or universities.

1.1.2 Deliverables

The deliverables are as follows:

- This written report documenting our work and the result of our project
- The parser/translator written in JavaCC
- A logo program we used to test our parser

1.2 Grammar

We didn't modify the grammar in any way.

```
Program    = "LOGO" Identifier { Subroutine } { Statement } "END"

Subroutine = "TO" Identifier { Parameter } { Statement } "END"

Statement  = "CS" | "PD" | "PU" | "HT" | "ST"
           | "FD" NExpr | "BK" NExpr | "LT" NExpr | "RT" NExpr
           | "WAIT" NExpr
           | "REPEAT" NExpr "[" { Statement } "]"
           | "IF" BExpr "[" { Statement } "]"
           | "IFELSE" BExpr "[" { Statement } "]" "[" { Statement } "]"
           | Identifier { NExpr }

NExpr      = NTerm { ( "+" | "-" )  NTerm }

NTerm      = NFactor { ( "*" | "/" ) NFactor }

NFactor    = "-" ( Number | REPCOUNT | Parameter | "(" NExpr ")" ) |
         Number | REPCOUNT | Parameter | "(" NExpr ")"

BExpr      = BTerm { "OR" BTerm }

BTerm      = BFactor { "AND" BFactor }

BFactor    = "TRUE" | "FALSE" | "NOT" "(" BExpr ")"
       | NExpr ( "==" | "!=" | "<" | ">" | "<=" | ">=" )  NExpr


Comments start with "#" with scope until the newline
Numbers are real numbers
Identifiers start with a letter followed by letters or digits
Parameters are ":" followed by Identifier
Identifiers, parameters, keywords in uppercase only
```

2 Implementation

2.1 Approach

In a first step we tried to develop a working parser. To test the parser we chose the most complicated logo file in the /logofiles folder. After that one by one we wrote a mapping between the different LOGO Primitives into Java code with the help of the given class LogoPrimitives.

2.2 Description of important elements

2.2.1 Comment

According to the grammar comment lines must be skipped by the parser/translator. We had to add a SKIP expression to our Logo.jj file with a regular expression that will fit to a complete line starting with the # character.

SKIP : <"#"(~["\n"])*>

2.2.2 Variables

At first we were on the wrong track with the variables. We thought that it was necessary to evaluate the Variables and not just translate them, which was a lot of extra work. After rereading the description, we decided to just translate everything into Java without the evaluation part, which was then done quite easily.

2.2.3 Loops

Here an explanation of how we implemented the Repeat keyword, because of it's close link with the REPCOUNT keyword. We chose to map it to a for loop, whose variable is a combination of the prefix i with the current number of indentation. This enables us to use nested repeats and still have a unique counter variable.

```
<REPEAT> iterations = nExpr() <LBRA>
{
    indent();
    numIndent++;
    pw.println("for(int i"+numIndent+" = 0;i"+numIndent+"<"+iterations+";i"+numIndent+"++){");
}
```

Also it is not a problem if two loops are on the same level of indentation. They will have the same variable name, but because it is in a different scope, the old variable will be hidden by the newly declared.

2.2.4 REPCOUNT

With this chosen approach for loops, the REPCOUNT becomes really easy to implement.

```
t = <REPCOUNT>  {value = "i"+numIndent;}
```

We just access the counter variable of the loop that we are in right now, by accessing the indentation that is already remembered by our parser. So even in nested loops, we will always access the right counter variable.

2.2.5 Error Warnings

During the course of our project we encountered different errors, here just the most common ones.

- Syntaxtic Error: The parser found an syntaxtic error
- Lexical Error: The lexer has found a bad character sequence
- Compilation Error of the generated Java File

2.3 Tests

We were running our parser/translator against each one of the files that were given to us for testing purposes. Everything worked just fine. Further we also tested the loops with the same indentation in the file /logofiles/testrep.logo. We also tried looking on the internet for some other examples, but many of them use a version of LOGO which contained Keywords or other grammars (like small lettered identifiers) that could not be understood by our parser.

2.4 Limitations

Our parser is only able to check for syntactic and not semantic errors, although those might be discovered during the compilation process of the generated java program. Apart from that we were able to implement all the features described by the grammar. Obviously the language LOGO itself is quite limited.

3 Conclusion

We enjoyed this project a lot, because it was an interesting application of the theory learned in the course Automata and Formal Language. We further had a chance to look into how JavaCC works. It was nice to have a direct visualization at the end of the project. The workload was quite a lot also because working with JavaCC was a bit tricky at first.