Berner Fachhochschule ▸ Technik und Informatik

Logo Parser / Translator

BTI7064 – Automata and Formal Languages – Spring Term 2015

Class I2a, Group #8

- Strub, Thomas Reto (strut1@bfh.ch)
- Weidmann Janick Joschua (weidj1@bfh.ch)

V1.0, Biel/Bienne, 12.06.2015

Table of Contents

Table of Figures

Introduction

We were tasked with writing a parser/translator in JavaCC (Logo.jj), which parses given Logo (*.logo) source code files matching an EBNF grammar which was provided, and parses it to Java. Which, when executed, drew the described picture in the ".logo" file.

For example:

The "square.logo" file (see figure 1) will be output as beautiful square (see figure 2)

```
1  # Displays a square of size 100
2
3  LOGO SQUARE
4
5    # Displays a polygon of size :SIZE with :SIDES sides
6    TO POLY :SIDES :SIZE
7      REPEAT :SIDES [ FD :SIZE LT 360/:SIDES ]
8    END
9
10   POLY 4 100
11
12   HT
13
14 END
```

Figure 1: square logo file

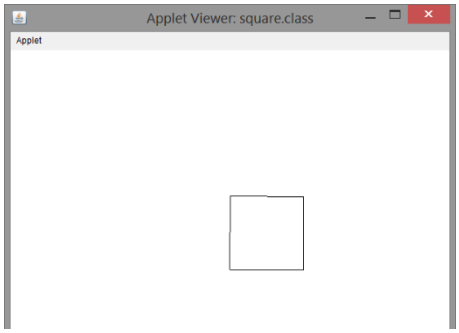

Figure 2: result of square logo file

In order for the parser to successfully interpret and parse the given ".logo" our parser needed to understand the used grammar and commands. Therein lay the challenge of this "mini-project". For example: Commands like `REPEAT :SIDES [ FD :SIZE LT 360/:SIDES ]` needed to be understood as well as parsed into a normal Java class as:

```
for (int i = 1; i <= sides; i++) {
      logo.fd(size);
      logo.lt(360 / sides);
}
```

Grammar

We slightly altered the provided grammar to complete our project. The changes that were made are highlighted in bold.

```
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    = "-" NFactorPos | NFactorPos

NFactorPos = 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
```

We eliminated the redundancy in the "NFactor" routine and created a second one called "NFactorPos"

Solution

We did a quite straightforward implementation by implementing both the parser and the translator combined in one file (Logo.jj). The implementation follows the rules as laid out in the grammar provided.

REPCOUNT

To implement the REPCOUNT identifier, we directly use the loop counter of the Java `for` loops.

Whenever we enter a REPEAT body, we increment a depth counter. When leaving it, we decrement it again. This counter can then be used to determine a unique variable name for up to 18 levels (characters `i` to `z`).

```
c = Character.toChars('i' + ++numRepeat)[0];
                          // no handling for more than 18 repetition levels
if (c > 'z')
  throw new ParseException("No more than " + ('z' - 'i' + 1)
                           + " repetition levels allowed.");
indent();
pw.println("for (int " + c + " = 1; " + c + " <= " + n + ";
                                                 " + c + "++) {");
```

This variable can then be used to represent the value of the REPCOUNT identifier.

```
<REPCOUNT> { // use innermost repeat counter
  return Character.toString(Character.toChars('i' + numRepeat)[0]);
}
```

Warning: Choice Conflict

A choice conflict warning is shown because when calling a subroutine, only the first parameter can be negative. If an additional value starts with a `-` (minus) it is considered to be part of a numeric expression.

We did not handle this warning because this is a problem with the grammar provided. It is simply impossible to determine the difference between an expression and a negative factor. As long as all whitespace is skipped, we can only pass `0 - <NUM>` as a workaround.

Test

We wrote a “master.logo” file, which tests all the required functions:

```
# master test program

LOGO MASTER

  TO WINDOWRECT :WIDTH :HEIGHT # rectangular window

    REPEAT 2 [

      FD :HEIGHT
      RT 90
      FD :WIDTH
      RT 90
    ]

  END

  TO WINDOWOCTA :LENGTH # octagonal window

    REPEAT 8 [

      IFELSE NOT (REPCOUNT <= 3 AND REPCOUNT >= 3
                  OR REPCOUNT < 8 AND REPCOUNT > 6) [
        FD :LENGTH
      ] [
        FD :LENGTH / 2
        RT 90
        FD :LENGTH / 3
        BK :LENGTH / 3
        LT 90
        FD :LENGTH / 2
      ]
      RT 45
    ]

  END

  REPEAT 3 [ # show 'ellipsis' to simulate 'calculation'

    FD (REPCOUNT + 1) * 2.5
    PU
    FD 5
    PD

    IF REPCOUNT != 3 OR FALSE [
      WAIT 125
    ]
    IF TRUE AND REPCOUNT == 3 [
      WAIT 250
      CS
    ]
  ]
```

```
ST # show initial turtle position
HT

PU # move to initial position (bottom-left corner)
FD -150
LT 90
BK 200
PD

FD 300 # draw a simple house
RT 45
FD 212
RT 90
FD 212
RT 45
FD 300
RT 90
FD 300
RT 90

PU # draw door
RT 90
FD 50
LT 90
PD
FD 50
RT 90
FD 25
RT 90
FD 10
BK 10
LT 90
BK 25
LT 90
FD 50
RT 90
FD 50
RT 90
FD 100
PU
LT 90
BK 100
LT 90
PD

PU # draw 'rectangular' windows
RT 90
FD 150
LT 90
FD 50
PD
WINDOWRECT 50 50
PU
RT 90
FD 75
```

```
  LT 90
  PD
  WINDOWRECT 25 50
  PU
  RT 90
  FD 25
  LT 90
  PD
  WINDOWRECT 25 50
  PU
  BK 50
  RT 90
  BK 150 + 75 + 25
  LT 90
  PD

  PU # draw 'round' window
  FD 300 - 25
  RT 90
  FD 165
  RT 180
  PD
  WINDOWOCTA 30
  PU
  RT 180
  BK 165
  LT 90
  BK 300 - 25
  PD

  ST # show final turtle position

END
```

The result draws a lovely little house (see Figure 3).

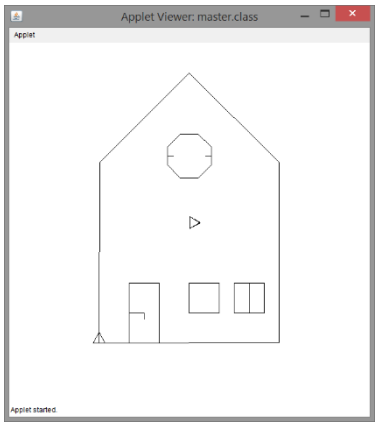

Figure 3: lovely little house

Limitations

Although our solution works with all of your provided examples and our own (hopefully) complete test program, there are certain limitations to it. As already discussed above, no more than 18 levels of nested REPEATs are supported, which should not significantly limit the solution's applicability.

As for the software engineering principles, it is clear we do violate some of them: Our Logo.jj file is responsible for both parsing and translating the LOGO files which is not generally desirable. These two different activities should be separated from each other and preferably not (directly) rely on each other.

Also, mixing the code fragments for these two activities in one file significantly reduces extensibility. The JavaCC syntax is already complicated on itself, but including the logic for both parsing and translating the LOGO language files further complicates reading and understanding the code.

In our opinion, the translation should be segregated from the parser by means of abstraction. It should be possible to use the same parser for different tasks. At least translation into multiple target languages should be supported. Ideally, the files should be executed directly, maybe analysed or even optimised and so on.