Journal of Electrical Engineering, Electronics, Control and Computer Science – JEEECCS, Volume 5, Issue 18, pages 1-6, 2019

Interpreting a basic turtle graphics programming language: hand-coded vs JavaCC based implementations

Florin-Marian Birleanu
Department of Electronics, Computers and Electrical Engineering
Faculty of Electronics, Telecommunications and Computer Science
University of Pitesti, Romania
florin.birleanu@upit.ro

Abstract – **Interpretation of custom programming languages is a challenging task for many software programmers. This paper presents an easy to follow approach to this problem by using as input language a small programming language for generating turtle graphics. First, we present a step by step almost automatic manual implementation in JavaScript of an interpreter for this input language. Then, we automate the implementation even more, by using the JavaCC code generator. In this second implementation, instead of directly interpreting the abstract syntax tree in order to generate a drawing in a Canvas element, we translate it to an SVG graphics file. Besides their usefulness as case studies, the resulting applications can be used as tools for teaching basic programming concepts.**

Keywords – formal languages; regular expressions; grammars; scanning; parsing; parser generators; interpretation; translation; turtle graphics

I. INTRODUCTION

Programming language interpretation and translation are a form of metaprogramming, which is why they scare many programmers. Even those that are familiar with regular expressions and use them on a daily basis may find it difficult to parse JSON or HTML data, for instance. Fortunately, there are libraries for performing these particular tasks. However, if one wants a different functionality or a faster implementation, the solution is to learn the details of parsing [1, 2]. This is surprisingly easy to implement in a mostly automatic manner, as we show in this paper, for a certain class of input languages. We illustrate this for the case of a simple input language that allows its user to easily program a virtual turtle to draw line-based drawings [3, 4, 5].

The input language used here is described in Section 2 and the generic approachh to language interpretation and translation is shown in Section 3. In Section 4 we present our JavaScript based implementation of a web interpreter that shows the output of the turtle program in an HTML5 Canvas element. Another implementation of the same input language is presented in Section 5. This second implementation uses the Java programming language and is even more automatic than the previous one, as it is based on a software tool (*i.e.*, JavaCC) that generates a large part of the source code for the interpreter. In this case the output of the turtle program is an SVG graphics file (that can be visualized in a web browser, as well).

II. THE TURTLE GRAPHICS LANGUAGE

The language that we used as input language for our interpreter is inspired by the LOGO pogramming language [6] and consists in a small set of instructions that command the movement of a virtual turtle that is able to move and draw lines on a virtual paper. The instructions are very simple and require each only one parameter, as they rely on a relative coordinate system. More precisely, the turtle is able only to go forward a certain number of steps, or to turn right or left a certain number of degrees. Two similar instructions are available for moving the turtle forward (relative to its current position and orientation): the "go" instruction (that moves the turtle by leaving a trail between its current position and its new position) and the "jump" instruction (that moves the turtle without leaving any trail on the paper). In order to make the language a little more interesting, we also introduced an instruction that allows the repetition of an instruction block for a certain number of times.

Expressed formally as a context-free grammar in EBNF notation [7], our turtle graphics language looks like in Table I. The regular expressions for the lexical atoms of the language are shown in Table II.

TABLE I. THE GRAMMAR OF OUR TURTLE LANGUAGE

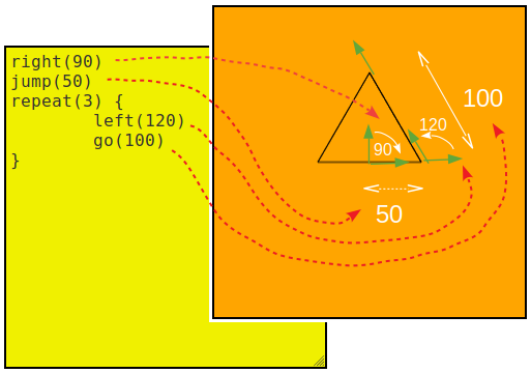
Variable		Body of production
Program	→	(Instruction)*
Instruction	→	Go | Jump | TurnLeft | TurnRight | Repeat
Go	→	<GO> <PO> <NUM> <PC>
Jump	→	<JMP> <PO> <NUM> <PC>
TurnLeft	→	<LFT> <PO> <NUM> <PC>
TurnRight	→	<RGT> <PO> <NUM> <PC>
Repeat	→	<REP> <PO> <NUM> <PC> <BO> (Instruction)* <BC>

TABLE II. THE REGEXES FOR THE LANGUAGE TOKENS

Token name		Regular expression
<GO>	:	go | GO
<JMP>	:	jump | JUMP
<LFT>	:	left | LEFT
<RGT>	:	right | RIGHT
<REP>	:	repeat | REPEAT
<PO>	:	(
<PC>	:	)
<NUM>	:	[0 - 9]+
<BO>	:	{
<BC>	:	}

That means that a simple program for drawing an equilateral triangle pointing up (North) would look like in Fig. 1.

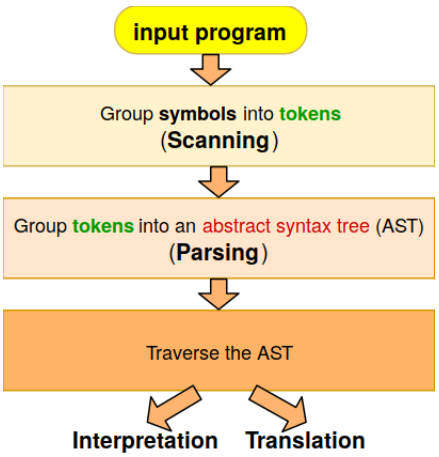
Figure 1. A simple program for drawing a triangle using our turtle graphics programming language.



III. LANGUAGE INTERPRETATION AND TRANSLATION

A program like the one in Fig. 1 is just a string of characters that needs to be understood as a program in our turtle graphics language. Whether one wants to do interpretation or translation, the three steps required are similar – see Fig. 2.

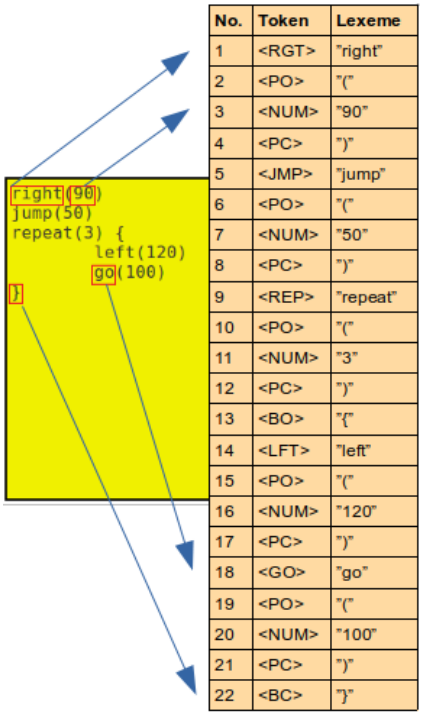
Figure 2. The generic approach to language interpetation or translation of a (programming) language.



In the lexical analysis (or scanning) step, the symbols (characters) in the input program are grouped into lexemes, that correspond each to a token type. For instance, the lexeme ”(” corresponds to the token type <PO>, while the token type <GO> can be associated to different lexemes, *i.e.*, ”go” and ”GO”.

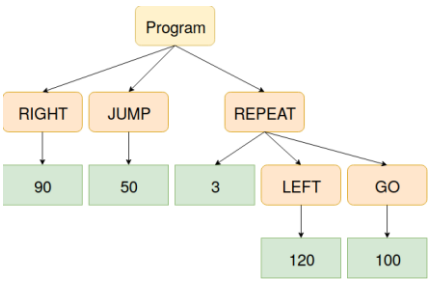
We choose to call token an object containing two fields: a token type (such as <GO>, <NUM>, <BO>) and a token value (a lexeme such as ”go”, ”100”, ”{”). Hence, the output of the lexical analysis step is a list of token objects. In order to make things more clear, Fig. 3 shows the list of tokens obtained after performing lexical analysis on the program in Fig. 1. (The token types from Table II were searched for, while the blanks, tabs and newline markers were ignored.)

Figure 3. The list of tokens resulted by scanning the program in Fig. 1.



While scanning (or lexical analysis) groups *letters* into *words*, parsing (or syntax analysis) groups *words* into *phrases*. The output of the parser is an abstract syntax tree (AST), which is a hierarchical representation of the tokens. For the list of tokens in Fig. 3, the AST might look like in Fig. 4.

Figure 4. The AST for the list of tokens in Fig. 3.



As it can be noticed, some of the tokens were redundant (such as <PO> and <PC>) and can be ignored when constructing the tree.

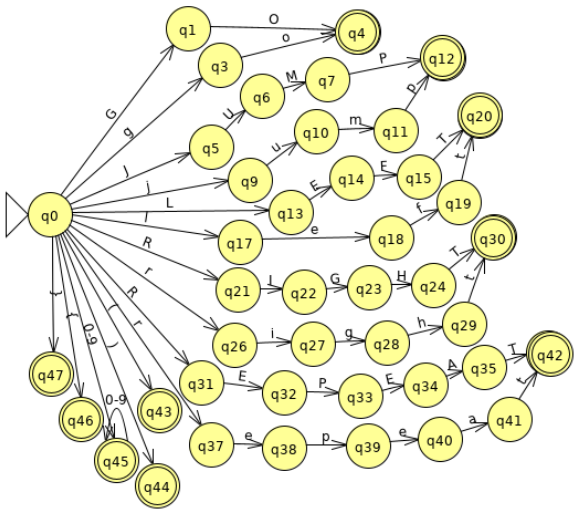
This tree representation of the program is far more manageable from a programming perspective than the string of characters from Fig. 1. One can easily see from it that the program consists in three instructions and that the third instruction is a repeat instruction for a block composed of two instructions. By traversing this tree in depth from left to right, each instruction can be directly interpreted or translated to a different language or representation.

IV. THE HAND CODED JAVASCRIPT BASED INTERPRETER

We present in this section the steps required in order to implement a web browser based implementation of an interpreter for programs written in our turtle graphics language.

As shown before, we need to perform three steps: scanning, parsing, and tree traversing. For implementing the scanning step we start from the regular expressions (from Table II) for the different token types that can appear in our language. Starting from these regexes we construct a finite automaton based on which we construct our scanner – see Fig. 5.

Figure 5. The finite automaton for our scanner.



For implementing the parser we use the fact that our grammar (see Table I) is an LL(1) grammar [1, 2], which means that its productions lack left recursion and, hence, can be easily implemented by turning each variable (from the head of the productions) into a function whose body is constructed based on the body of the production as follows:

- a variable turns into the call of the function that corresponds to that variable
- a token type turns into the call of a function that verifies that the current token from the parser input corresponds to that token type.

Whenever there are more than one production bodies to choose from, we can make a decision based on the current token type from the input of the parser. (This is guaranteed by the ”1” in LL(1), which is a subclass of context-free grammars that can be hand-coded by using the method presented here.)

For instance, three of the functions of our parser might look like in Fig. 6.

Figure 6. The first version (that only performs syntax check) of the JavaScript based parser.

```
function Program()
{ // Program --> ( Instruction )*
    while (CurrentToken() != null) Instruction(); }
function Instruction()
{ // Instruction --> Go|Jump|TurnLeft|TurnRight|Repeat
    var ct = CurrentToken()
    if (ct.tok == GO) Go();
    else if (ct.tok == JMP) Jump();
    else if (ct.tok == LFT) TurnLeft();
    else if (ct.tok == RGT) TurnRight();
    else if (ct.tok == REP) Repeat();
    else SyntaxError(); }
function Go()
{ // Go --> <GO> <PO> <NUM> <PC>
    VerifyToken(GO);
    VerifyToken(PO);
    VerifyToken(NUM);
    VerifyToken(PC); }
...
function Repeat()
{ // Repeat --> <REP><PO><NUM><PC><BO>(Instruction)*<BC>
    VerifyToken(REP);
    VerifyToken(PO);
    VerifyToken(NUM);
    VerifyToken(PC);
    VerifyToken(BO);
    while (CurrentToken()!=null && CurrentToken().tok!=BC)
        Instruction();
    VerifyToken(BC); }
```

However, this version of the parser is only able to check whether the list of tokens produced by the scanner represents a syntactically correct program. (One can do that check by simply calling the function Program(). If no syntax error message appears, the program passed the syntax check.)

A true parser would generate an AST (whose root would be returned by a call to Program()). This can be done by performing some modifications on the code in Fig. 6. See Fig. 7 for the result.

Figure 7. The second version (that generates an abstract-syntax tree) of the JavaScript based parser.

```
function Program() {
    var ret = [];
    while (CurrentToken() != null)
        ret[ret.length] = Instruction();
    return ret; }
function Instruction() {
    var ct = CurrentToken();
    if (ct.tok == GO) return Go();
    else if (ct.tok == JMP) return Jump();
    else if (ct.tok == LFT) return TurnLeft();
    else if (ct.tok == RGT) return TurnRight();
    else if (ct.tok == REP) return Repeat();
    else { SyntaxError(); return null; } }
...
function Repeat() {
    var ret = {type: 0, num: "", instr: []};
    if (!VerifyToken(REP)) return null;
    if (!VerifyToken(PO)) return null;
    if (!VerifyToken(NUM)) return null;
    if (!VerifyToken(PC)) return null;
    if (!VerifyToken(BO)) return null;
    var ret = {type: REP, num: tokenList[index-3].val,
               instr: []}
    while (CurrentToken()!=null && CurrentToken().tok!=BC)
        ret.instr[ret.instr.length] = Instruction();
    if (!VerifyToken(BC)) return null;
    return ret; }
```

In order to interpret the resulting tree, we must first decide how to model internally the position and orientation of the turtle. As it can only move forward and rotate, it is enough to store the current position (*tx*, *ty*) of the turtle on the page and its orientation (*ta*) in degrees measured anticlockwise starting from the *x* axis.

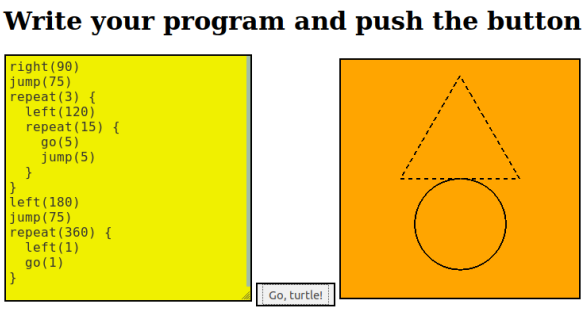
With these in mind, the interpretation of the tree might look like in Fig. 8.

Figure 8. The interpretation of the abstract-syntax tree.

```
function Interpret(ast){
    for (var i=0; i<ast.length; i++)  InterpInstr(ast[i]);
}
function InterpretInstr(instr) {
    var type = instr.type;
    if (type == GO) {
        canvasctx.moveTo(tx, ty);// current turtle position
        tx += instr.num*Math.cos(ta/180*Math.PI); //delta_x
        ty -= instr.num*Math.sin(ta/180*Math.PI); //delta_y
        canvasctx.lineTo(tx, ty); // new position of turtle
        canvasctx.stroke();
    }
    else if (type == JMP) {
        tx += instr.num*Math.cos(ta/180*Math.PI); //delta_x
        ty -= instr.num*Math.sin(ta/180*Math.PI); //delta_y
    }
    else if (type == LFT) {
        ta += instr.num; // increase turtle angle
    }
    else if (type == RGT) {
        ta -= instr.num; // decrease turtle angle
    }
    else if (type == REP) {
        for (var k=0; k<parseInt(instr.num); k++)
            for (var i=0; i<instr.instr.length; i++)
                InterpretInstr(instr.instr[i]);
    }
    else { alert("Unknown instruction type!"); }
}
```

An example of running the interpreter is shown in Fig. 9.

Figure 9. The result of running our JavaScript based interpreter for a test program.



V. The JavaCC based translator

JavaCC (Java Compiler Compiler) is a software tool that generates Java source code for the scanning and parsing stages of the interpreter based on the regular expressions for the tokens and on the EBNF grammar of the source language [7]. The regexes and the grammar are specified using specific constructs in an input file (having the ".jj" extension). This file can also contain the Java code that instantiates and runs the parser and then traverses the AST and interprets it.

Actually, in this version of our application instead of interpreting the AST and showing the generated turtle drawing in a graphical user interface, we translate the AST to an SVG file (that can subsequently be viewed in a web browser).

The JavaCC specifications file starts with the definition of the main class of the translator, placed between PARSER_BEGIN(…) and PARSER_END(…). This class only contains the method main, in which the parser (*i.e.*, the class itself) is instantiated (with System.in as its argument, which means that the input program will be read from standard input in the console) and the generated tree is translated (with the SVG output being written to standard output in the console). In the PARSER... section of the JavaCC specifications file we also put the definition for the Node class, based on which the AST nodes are created. Outside of this section in the ".jj" file we put the regular expressions, followed by the grammar.

The scanner is described with the aid of the SKIP and TOKEN keywords. For instance, SKIP: {" "|"\t"} tells the scanner to ignore blanks and tabs in the input. And TOKEN: {<NUM: (["0"-"9"])+>} describes a token called NUM whose regex is [0-9]+.

The grammar of the parser must be LL(k) and is described with an EBNF-like notation that looks like this Program():{}{(Instruction)*} for the production Program → (Instruction)*. Of course, in order to construct the AST some things must be added, which complicate this structure. The main parts of the JavaCC specifications file are shown in Fig. 10.

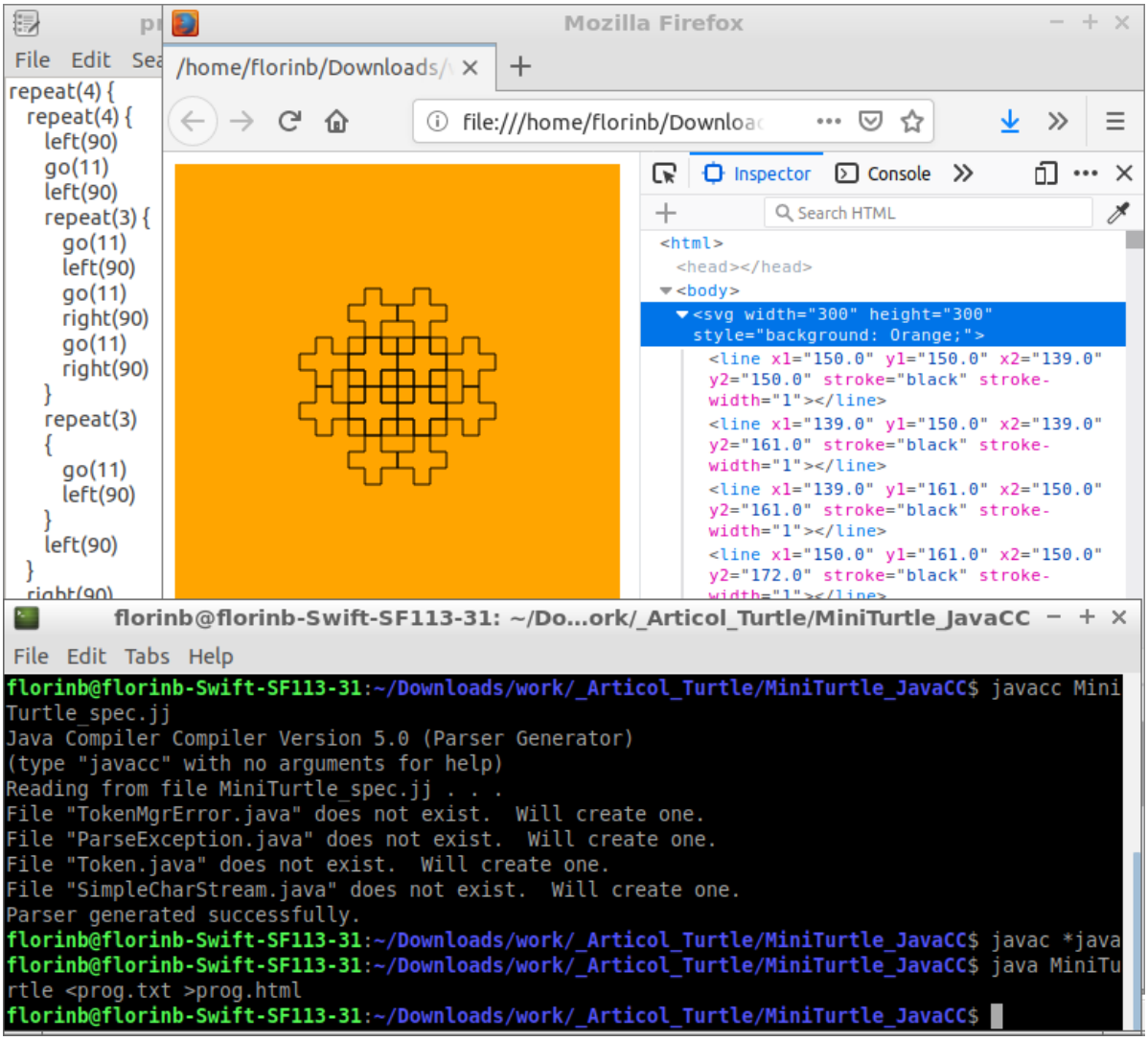
Figure 10. The JavaCC specifications for the turtle language program to SVG image translator.

```
PARSER_BEGIN(MiniTurtle)
    public class MiniTurtle {
        ...// Turtle data
        // Main program
        public static void main(String[] args) throws Exception {
            Node ast = new MiniTurtle(System.in).Program();
            System.out.println(ast);
        }
        // Abstract Syntax Tree node
        public static class Node { ... }
    }
PARSER_END(MiniTurtle)
SKIP : {" " | "\t" | "\n" | "\r\n"}//Ignore space, tab, end of line
TOKEN : { <GO : "GO"|"go"> }
TOKEN : { <NUM : (["0"-"9"])+> }
...
Node Program() : {Node n; Node first;} {
    { n = new Node(); first = n; }
    ( n.next=Instruction() {n=n.next;} )* <EOF>
    { return first; } }
Node Instruction() : {Node n; Token t; Node i;} {
    <GO> <PO> t=<NUM> <PC> //1 - Go
    { n = new Node(); n.type = 1; n.num = Integer.parseInt(t.image);
      return n;}
    | <JMP> <PO> t=<NUM> <PC> //2 - Jump
    { n = new Node(); n.type = 2; n.num = Integer.parseInt(t.image);
      return n;}
    | ...
}
```

The SVG content is output to the standard output in the console, but can be easily redirected to an ".svg" file. It only contains <line> elements, corresponding to the GO commands in the input program. The translation from the AST to the SVG text content is almost identical to the interpretation discussed in the previous section, except that instead of drawing lines in the canvas, we add "<line …>" to the SVG output string.

The ".jj" specifications file is translated by JavaCC (using the "javacc" command in the console) to multiple ".java" files that are then compiled by "javac" to ".class" files. The main class can then be run (with "java") in order to accept input code in our turtle language and generate the corresponding SVG image. Fig. 11 shows an example of use.

Figure 11. Compilation and use of the JavaCC based turtle language to SVG translator.



ACKNOWLEDGEMENT

The author would like to thank his former students Ionela-Florina Rosu and Gabriel-Valentin Gheorghe for their support in the implementation and testing stages of the software applications presented in this paper.

CONCLUSION

Programming language source code interpretation (or translation to another language) is a challenging string processing task. However, this task is manageable if one follows certain well-defined steps. This paper discussed these steps and illustrated them for the case of a didactic language that can be useful for teaching programming basics. First, a manual implementation of an interpreter for this toy programming language was shown, in order to clearly explain the steps required. Then, the implementation of the first two (out of the three) steps was automated with the aid of a source code generator. The resulting applications can be easily modified in order to extend the capabilities of the input language.

REFERENCES

[1] T. Æ. Mogensen, Introduction to Compiler Design, 2nd ed., Springer International Publishing, 2017.

[2] D. Grune and C. J.H. Jacobs, Parsing Techniques. A Practical Guide, 3rd ed., Springer-Verlag New York, 2008.

[3] C. J. Solomon and S. Papert, "A case study of a young child doing turtle graphics in LOGO," in Proceedings of the AFIPS '76 national computer conference and exposition, New York, June 7-10, 1976, pp. 1049–1056.

[4] M. E. Caspersen and H. B. Christensen, "Here, there and everywhere - on the recurring use of turtle graphics in CS1," in Proceedings of the ACSE '00 Australasian conference on Computing education, Melbourne, 2000, pp. 34–40.

[5] R. Goldman, S. Schaefer, and T. Ju, "Turtle geometry in computer graphics and computer-aided design," Elsevier Computer-Aided Design, vol. 36, pp. 1471–1482, December 2004.

[6] W. Feurzeig and G. Lukas, "LOGO – A Programming Language for Teaching Mathematics," Educational Technology, vol. 12, no. 3, pp. 39–46, March 1972.

[7] V. Kodaganallur, "Incorporating language processing into Java applications: a JavaCC tutorial," IEEE Software, vol. 21, pp. 70–77, July-Aug. 2004.