SYNTAXTRAIN
User's Guide

Version 1.0

Andreas Leon Aagaard Moth
Technical University of Denmark
DK-2800 Lyngby, Denmark

June 20, 2011

1 Introduction

SYNTAXTRAIN is a tool intended for novices learning to program. It parses the student's program and if a syntax error is found it displays the syntax diagrams of the incorrect construct. Syntax diagrams are visual representations of the grammar of a programming language that are easier to read and understand than BNF. By tracing through the diagrams a student can easily locate and fix syntax errors.

Currently, SYNTAXTRAIN supports programs written in JAVA, but it can be modified to use other languages.

Syntax diagrams were first used in Pascal; see: K. Jensen and N. Wirth. *PASCAL User Manual and Report*, LNCS 18, Springer-Verlag, 1975.

2 Installation and execution

The website for SYNTAXTRAIN is `http://code.google.com/p/syntaxtrain/`.

SYNTAXTRAIN is open-source software under the terms of the GNU General Public License, Version 3.

Students and teachers

- SYNTAXTRAIN requires JAVA JRE 1.5 or above.
- Download and open the archive `syntaxtrain-n-n.zip`. This will create the executable file `SyntaxTrain.jar`, a directory with the documentation and a directory `examples` with the Java source code of the examples.
- To run, double-click the icon for the `jar` file or run the following command from the command line:

  ```
  java -jar SyntaxTrain.jar
  ```

Developers

- SYNTAXTRAIN requires JAVA JDK 1.5 or above.
- Download and extract the archive `syntaxtrain-source-n-n.zip`. This will create the directory `src` with the source and BNF files.
- Section 5 explains how to change the language of SYNTAXTRAIN and the software documentation is in Section 6.

Limitations

SYNTAXTRAIN has some minor limitations due to implementation difficulties.

- The JAVA program must use braces `{ }` around the sequences of statements controlled by an if-statement, even if the sequence consists of a single statement:

```
if (expression) {
  Statement1;
}
else {
  Statement2;
}
```

- A statement consisting only of an identifier is not flagged as an error:

```
int dayInMonth(int month) {
  month;
}
```

3 User interface

The user interface consists of a toolbar and three panes. At the top of the window is a small pane that spans the width of the window; below it, the window is divided into larger left and right panes.

The toolbar contains the following buttons:

- **Open** - Open a file (Ctrl+O).
- **Reload file** - Reads the current file again overwriting all changes since last save (Ctrl+R).
- **Check Syntax** - Revalidates the code and draws syntax diagrams if a syntax error is found (F5).
- **Save** - Save the source code (Ctrl+S).
- **Show/hide syntax components** - Shows/hides a list of all syntax rules for the current language (F10).
- **Help** - Displays help information.
- **About** - Displays the copyright notice.

When a file is opened its source code is displayed in the left pane. If a syntax error is found, SYNTAXTRAIN will highlight the code that did not match the grammar. You can edit and revalidate (F5) your code without having to save it. Once you have corrected all your errors you can save the code and return to using your development environment.

The right pane shows the syntax diagrams for the rules involved in the incorrect code, starting with the rule that was last used, then the rule which called that rule and so on until the first rule for the language is reached. The rules are marked with colors; their meanings are:

- **Black:** This rule is not relevant for your source code.
- **Blue:** This rule has been correctly matched with your source code.
- **Red:** This rule caused an error when parsing your source code.
- **Yellow:** These rules would be legal to write at the position of your error.

The top pane shows a list of the rules involved in the error, starting on the left with the last one.

If `Show/hide syntax components` is selected, a new pane will appear at the right of the window. The pane contains a complete list of the syntax rules of Java.

4 Debugging a syntax error with SYNTAXTRAIN

When you encounter a syntax error that you don't understand, open the source in SYNTAXTRAIN. Look at the highlighted code in the left pane and then at the last rule used—the top rule in the right pane. Since the error might have caused the parser to become confused, look down through the diagrams until you find a diagram that you understand and then work your way back up, looking for the first rule that doesn't match what you intended. Let us look at some examples:

4.1 Example 1

Consider the JAVA code in Figure 1 which computes the number of days in a month (ignoring leap years); Figure 2 shows a screenshot of SYNTAXTRAIN after opening this file.

From the left pane we know that there is a problem with the code that follows the keyword `case`. The right pane shows that the error occurred when parsing `expressionMain` as part of `Expression`. The yellow elements in the first diagram show what the parser was expecting, but there does not seem to be a problem with any expression, so we look down to the syntax diagram for the `switch_statement`, shown magnified in Figure 3.

```
class Dates {
  int dayInMonth(int month) {
    switch (month) {
      case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        return 31;
      case 4: case 6: case 9: case 11:
        return 30;
      case: 2
        return 28;
      default:
        return 0;
    }
  }
}
```

Figure 1: Example 1

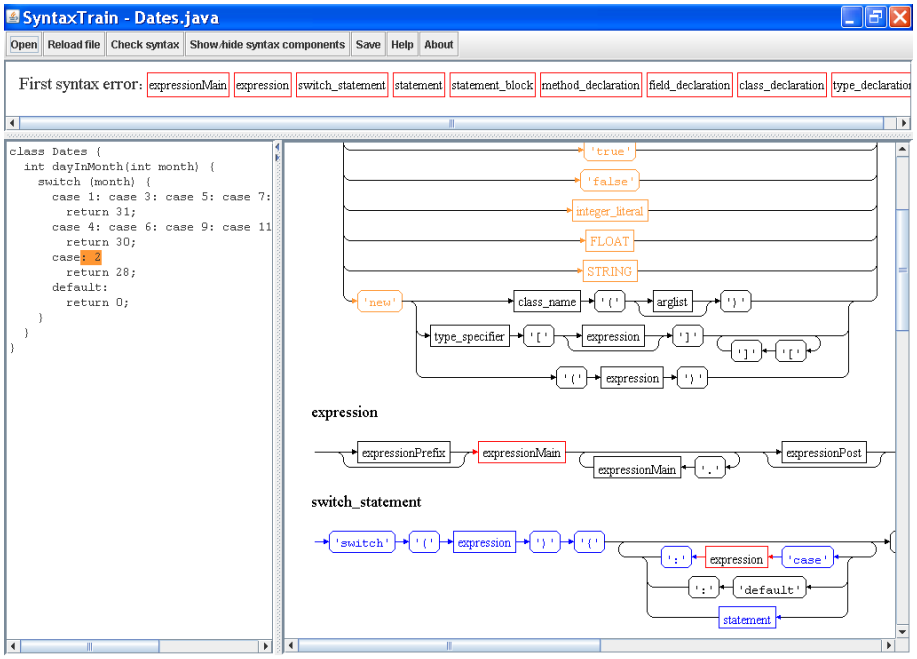

Figure 2: The Date class syntax error displayed by SYNTAXTRAIN

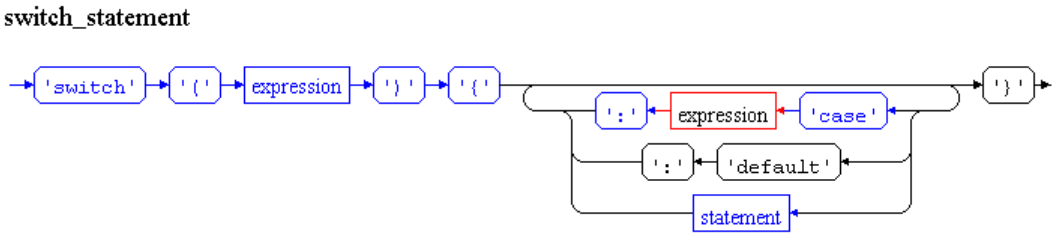

Figure 3: The rule for the `switch` statement

Trace through this diagram; every element is blue until we reach the element after the keyword `case`. Clearly, the parser was expecting an expression, but instead we have written the colon before the expression (the integer literal 2) instead of after it. Modify the source code and revalidate; the error is gone!

4.2 Example 2

Look at the code in Figure 4 which writes a string to a file.

```
class FileWriterHelper {
  import java.io.*;

  public void writeTextToFile( String text, File file ) {
    try {
      BufferedWriter out =
        new BufferedWriter(new FileWriter(file));
      out.write(text);
      out.flush();
    }
    catch (IOException e) {
      System.out.println("Text was not written to file!");
    }
  }
}
```

Figure 4: Example 2

Figure 5 shows a screenshot of the code opened in SYNTAXTRAIN. The left pane shows that the error occurred while parsing the import statement.

In the right pane the top diagram is `class_declaration` (Figure 6). This diagram shows

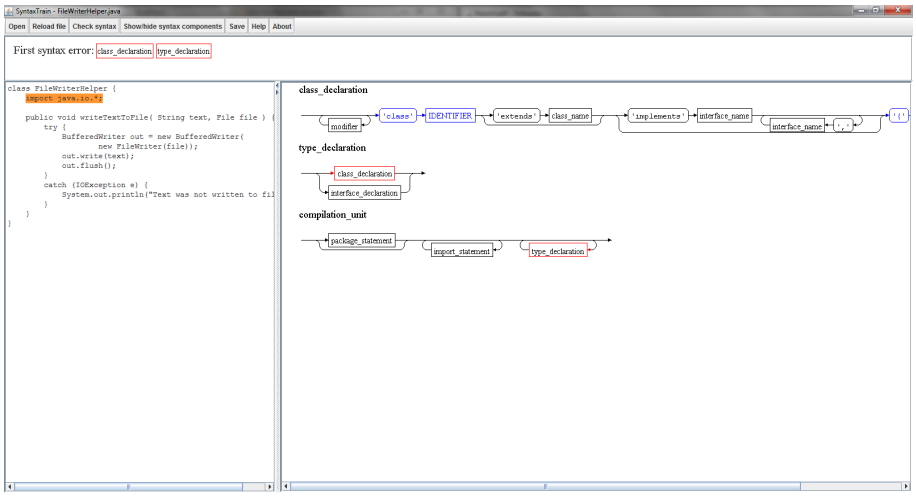

Figure 5: FileWriterHelper opened in SYNTAXTRAIN

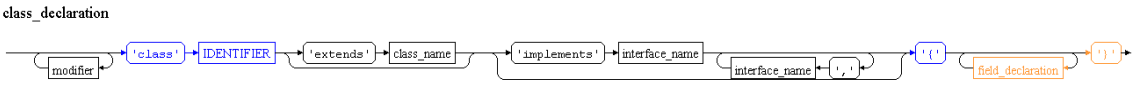

Figure 6: The `class_declaration` statement shown by SYNTAXTRAIN

that only `}` and `field_declaration` are legal to write at this point. The former is not relevant since the parsing has not reached the end of the class.

However, a `field_declaration` must be a method, a constructor, a variable declaration or a static initializer (If you do not remember this, select `Show/hide syntax components` (F10) and click the `field_declaration` component). It follows that `import` is illegal at this point. Looking down through the diagrams, we see that `compilation_unit` can accept an `import_statement`, but before the `type_declaration`, which is an interface or a class. Moving the import statement before the class declaration solves the problem.

4.3 Example 3

Figure 7 shows a fragment of a calculator class, including a method for the operation square root. The method saves the result in the variable `sum`. This code contains no syntax errors.

Suppose now that you forget the semicolon at the end of line 7. Figure 8 shows a screen-shot of SYNTAXTRAIN; it is clear that the semicolon is missing, since expression was matched correctly and semicolon is now the only legal possibility.

Suppose, instead, that the user forgets the left brace { for the `while`-statement at line 6. Figure 9 shows this error displayed in SYNTAXTRAIN. From the highlighted source code in the left pane, it is seen that the error occurred after the assignment to the variable

```
1 class Calculator {
2         int sum;
3
4         public void sqrt() {
5                 int s = 2;
6                 while ((s)*(s) <= sum) {
7                         s = s + 1;
8                 }
9                 sum = s - 1;
10        }
11        public Calculator(int initialValue) {
12                sum = initialValue;
13        }
14 }
```

Figure 7: Example 3

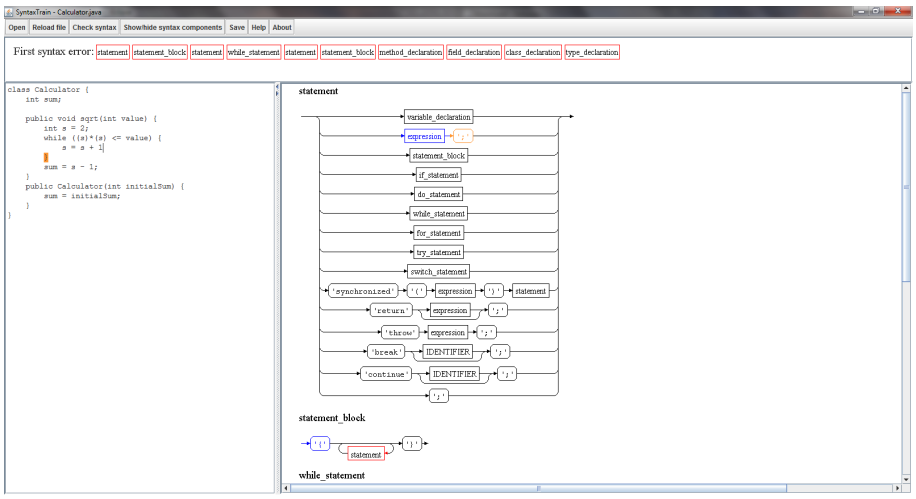

Figure 8: Screenshot of a syntax error where a semicolon is missing

`sum`. The right pane shows that the error occurred while parsing `field_declaration`, which means that the compiler expects a method, constructor, variable declaration or static initializer at this point. What the user intended to write at that point is neither of those, but an expression. Therefore we go on to the next diagram, in order to find something familiar and then backtrack.

The next diagram is `class_declaration`, see Figure 10. This is where the Calculator class is defined. Tracing through this diagram we see that the error occurs while inside the brace and parsing `field_declaration`, which we already knew. Since the error occurred at the level of the `field_declaration`, and not inside the method, that means that for some reason we are *outside* the function. A careful inspection of the code will show that the brace that should have closed the `while`-statement actually closed the function instead.

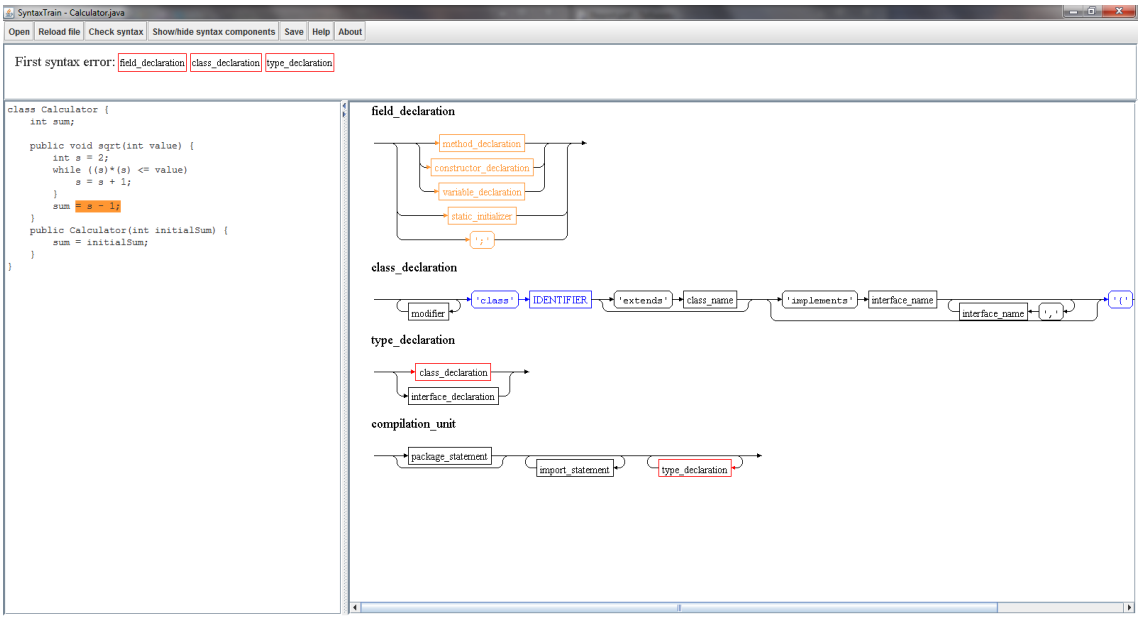

Figure 9: Screenshot of a syntax error where the { is missing

Figure 10: The `class_declaration` statement shown by SYNTAXTRAIN

5 Adapting SYNTAXTRAIN for a new language

The BNF compiler is a tool that reads a text file containing the grammar of a language in BNF and creates a grammar file that can be used by SYNTAXTRAIN. The distribution includes the file `javagrammar.bnf` with the BNF for JAVA. To create the grammar file for SYNTAXTRAIN, run:

```
java -jar BnfCompiler.jar bnf-filename
```

First make sure that the path to your JDK bin directory is specified in the file `options.xml`.

There are several limitations that must be followed when preparing a BNF for use in SYNTAXTRAIN:

- The first rule is the starting rule.
- The BNF may not be left recursive.
- The BNF must be decidable, that is, there must be only one way to match a given string.

The following rules have been pre-programmed into the grammar file:

- IDENTIFIER: any string consisting of letters, underscore and numbers, and starting with a letter or underscore.
- INT: integer literals.
- FLOAT: float literals.
- HEX: hex literals, which must start with a number as in 0FE.
- STRING: strings delimited by quotation marks.
- HEX_DIGIT: a single-digit hex literal.

Additionally the following are automatically ignored:

- Comments: delimited by `/*` and `*/`, or `//` until end of line.
- Whitespace: new lines, line feed, tabs and spaces.

6 Software documentation

SYNTAXTRAIN uses the ANTLR parser generator `http://www.antlr.org/` in order to convert a BNF into a stack of rules. This stack of rules are used to create another ANTLR file that is able to verify the language specified by the BNF and return a stack trace in case there is a syntax error.

CLAPHAM `http://clapham.sourceforge.net/` is used to draw the syntax diagrams. It has been modified to support specifying colors and fonts, along with some minor bug-fixes.

The SYNTAXTRAIN software consists of two major components: the GUI and the kernel, which communicate through two interfaces: `GuiApi` `KernelApi`. The kernel manages the interaction with ANTLR and the modelling part of CLAPHAM; that is, to describe to CLAPHAM how the diagrams are structured and which components should be highlighted.

The kernel consists of the following classes:

- **GrammarBase**: Reads/writes source code and loads grammar files.
- **GrammarCompiler**: Creates syntax nodes used by CLAPHAM, modelled to match the grammar and colored depending on the stack trace.
- **SourceCodeCompiler**: Verifies the source code using ANTLR parser and lexer files, loaded by GrammarBase.
- **GrammarInterface**: Interface to the kernel, to ensure that calls are made in right order.
- **Variables**: Various global variables and constants used by the kernel.

The GUI consists of the following classes (graphical components all starts with a lowercase *g*, with the exception of MainScreen):

- **Controller**: Handles all gui events, except those of the `Syntax components` pane.
- **Dialogs**: Shows about and help dialogs.
- **gErrorTrace**: The top pane that shows a list of rules involved in the error.
- **gGrammarDiagram**: Panel containing the diagrams generated by CLAPHAM, it also calls CLAPHAM to generate these diagrams.
- **gGrammarOptions**: Panel allowing the user to select which diagrams are shown.
- **gGrammarPanel**: Panel containing both `gGrammarDiagram` and `gGrammarOptions`.
- **gSourceCode**: Text pane for showing and highlighting the source code.

- **gToolbar**: Toolbar at the top (also registers shortcuts).
- **MainScreen**: The main frame.
- **Variables**: Various global variables and constants used by the gui.

All grammar files contains a parser that extends `BnfParser`, such that all grammars have a similar interface.

Besides from these classes, a few helper classes are also used. These are `XmlNode` (reads XML), `Lock` (concurrent lock, to avoid race-conditions) and `StdLibrary` (reads files and escapes strings).

The `BnfCompiler` consists of `CommandLineTool`, which manages the entire process of generating a grammar file from a BNF. `CommandLineTool` uses `BnfEvaluatorParser` and `BnfEvaluatorLexer` to read BNF and return a stack of `Link` to describe the language. `BnfEvaluatorParser` and `BnfEvaluatorLexer` are generated using ANTLR, the code can be found in `BnfEvaluator.g`. `Link` is a class, which you can put together with other links, telling that the given link is optional, one of multiple options, a loop running 0+ times, a loop running 1+ times or a simple sequence.