Project 1

Creating a GUI by Parsing

Samuel Scalf

CMSC 330 · Alin Suciu

University of Maryland Global Campus

Table of Contents

Process and Lessons Learned

Before even beginning to create classes or methods to parse, I created a single Test class in which I simply created a GUI based directly on the example input file. This was later used to help with creating the Parser class.

Knowing that I would need to read from a file and create tokens, I created a Tokenizer class, which used a StreamTokenizer to split the character stream. I didn't know about StreamTokenizer until I did a search in the Java 8 API documentation. I initially just output each token on a new line with the "TTYPE: token" format. For example, a number would appear as "NUMBER: 200" in the output. This helped me see how tokens were split by the StreamTokenizer. I had to use the ordinaryChar('.') method to mark the period character as a stand-alone character, because it was included in the last "End" word.

Once I got the tokens all sorted out into four categories – NUMBER, WORD, STRING, and PUNCTUATION – I knew I needed to store them somehow, so I created a Token class with String and Type fields to hold the string value and type of token. Here I created a Type enum that just had the four categories. Still, I needed somewhere to store the Tokens, so I quickly made a List-based generic Queue class that stored data in generic Node classes. With the Queue class, I was able to create a custom toString method that output all nodes in the queue without removing any. This was extremely beneficial for troubleshooting and debugging.

At this point, I felt I was ready to begin working on the Parser class. The first thing I did was create a field for the tokenizer and instatiated it in the default constructor. I then created a showTokens() method that output all the tokens and an empty parse() method. I frequently switched between the two methods throughout the rest of the process, especially once I started testing custom errors. In the parse() method, knowing it would be both the initial and final step in the process, I created an if statement that would only show the GUI if the parsing was successful, otherwise it would throw a custom SyntaxError.

I created a SyntaxError class that extended the Exception class, which led to a lot of throws on the methods, so I maybe could have extended RuntimeException instead, but I felt that checked exceptions were the way to go. This way, if anyone were to use my Parser class and attempt to use the parse() method, the compiler would know whether there was an unhandled exception.

For this to work, a new "parseGui()" method would need to be created and return a boolean. It would also need the first token to be in the "currentToken" field. As I was creating a method to evaluate the current token against an expected token, I figured that throwing an exception for an unexpected token would be easier than returning false, and require less lines of code. This also had the added benefit of stopping at the first syntactical error to throw an exception with a message that included both the expected token and the received/found token. Most of the other parse methods returned "void" for this very reason. The methods for parsing widgets, widget, radio_buttons, and radio_button all returned boolean values, which was needed for successful recursing.

When it came to evaluating the tokens, I quickly realized that my four Types would not cut it. I renamed my Type enum to Keyword, added entries for each of the grammar-required tokens (i.e. WINDOW, LAYOUT, etc.) and each of the punctuation (i.e. COMMA, PERIOD, etc.). I then removed "WORD" and "PUNCTUATION" from the list, while retaining STRING and NUMBER. These changes meant a changes to both the WORD and PUNCTUATION cases in the Tokenizer. WORDs were a bit easier, because I just had to match the uppercase version of the token to the Keyword. PUNCTUATION on the other hand required storing the punctuation characters in a list, then targeting the keyword based on its index/ordinal. Since punctuation characters came right after the words in the Keyword enum, I simply added the ordinal for first punctuation to the index of the found character in a List.

While working through the different parse methods, I soon came across my first number. StreamTokenizer did have a number output in the form of nval, but it was a double and my tokens all had just a String value. To resolve the issue, I added an integer "number" field to the Token class and cast the double nval to an int as only integers are used when specifying dimesions for JFrame and GridLayout. I now had two Token constructors: one accepting a String and Type parameter and the other accepting an int and Type parameter.

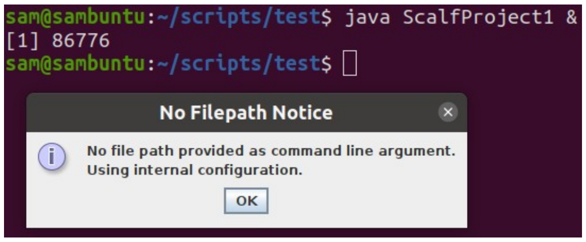
The recursive methods for widgets and radio buttons were very similar. Since at least one would be needed, I decided to create boolean variables to store whether it was needed. This variable was set to true right before calling the initial method (parseWidgets() or parseRadioButtons()) and set to false if the secondary method (parseWidget() or parseRadio()) returned true. This boolean check for whether the token was needed helped with detecting syntax errors as well, because I could check if at least one had been implemented. I might have been able to do this by simply evaluating against an expected token, but that seemed a lot messier when it came to widgets.

I felt I had finally completed the coding, so I went on to test. I started by using the example provided, which was also set as a default for if no command line arguments were given. This turned out successful, so I moved on to making test cases.

Overall, I feel this would have been easier if I had done more planning on paper, especially when it came to the overall process. The method implementations would not have been important that early, but being able to draw lines between methods/classes would have helped me see the flow better. I would have been able to probably see which fields I would need in the classes as well. Both of these would have saved me a lot of time going through code and changing/updating references. I am going to definitely try to use this newfound knowledge on the next project.

Compilation

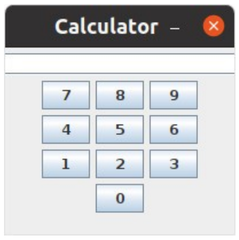
I was able to successfully compile the entire program using the javac command. Note that I am using Java 1.8 update 252.



```
sam@sambuntu:~/scripts/test$ java ScalfProject1 &
[1] 86776
sam@sambuntu:~/scripts/test$
```

No Filepath Notice

No file path provided as command line argument.
Using internal configuration.



Test Cases

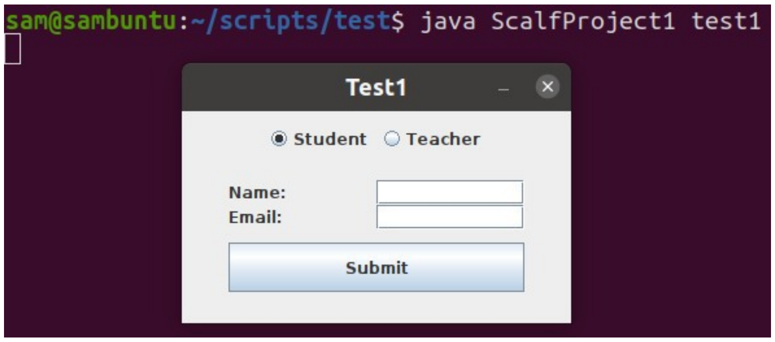
Test Case 1: All Features

My first test case included all layouts, including both options for Grid, and at least one of each widget, including nested panels.

Input File

```
Window "Test1" (300,200) Layout Flow:
     Panel Layout Grid(3,0,10,10):
          Panel Layout Flow:
               Group
                    Radio "Student";
                    Radio "Teacher";
               End;
          End;
          Panel Layout Grid(2,2):
               Label "Name:";
               Textfield 10;
               Label "Email:";
               Textfield 10;
          End;
          Button "Submit";
     End;
End.
```

Results



```
sam@sambuntu:~/scripts/test$ java ScalfProject1 test1
```

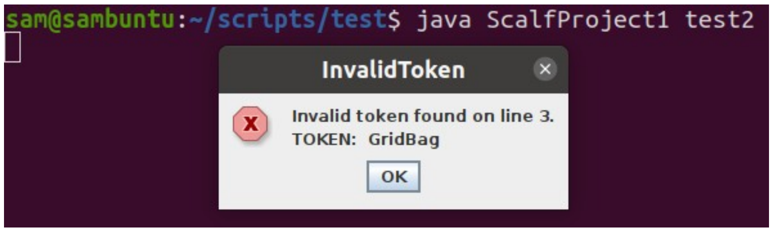
Test Case 2: Invalid Token

My second test case was to test error detection in the event an invalid token was detected (even before parsing for the GUI). In other words, if an unknown word or character existed, then an exception was thrown. This did not check if the token was syntactically correct, just if the combination of characters was a recognized token (e.g. “GridBag” is not a known token in this grammar).

Input File

```
Window "Test2" (200,300) Layout Grid(3,0):
     Label "Invalid Token Test";
     Panel Layout GridBag:
          Label "Name:";
          Textfield 5;
     End;
     Button "Close";
End.
```

Results



```
sam@sambuntu:~/scripts/test$ java ScalfProject1 test2
```

InvalidToken

Invalid token found on line 3.
TOKEN: GridBag

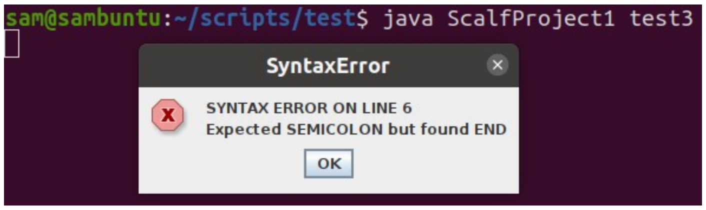
Test Case 3: Multiple Syntax Errors

The third test case was to verify the proper word was found, based on the grammar. The first incorrect token threw a syntax error stating the line of the token, the keyword of the token, and what was expected.

Input File

```
Window "Test3" (200,300) Layout Grid(3,0):
     Label "Multiple Syntax Errors";
     Panel Layout Flow:
          Label "Name:";
          Textfield 5
     End.
     Button "Close";
End.
```

Results



```
sam@sambuntu:~/scripts/test$ java ScalfProject1 test3
```

SyntaxError

SYNTAX ERROR ON LINE 6
Expected SEMICOLON but found END

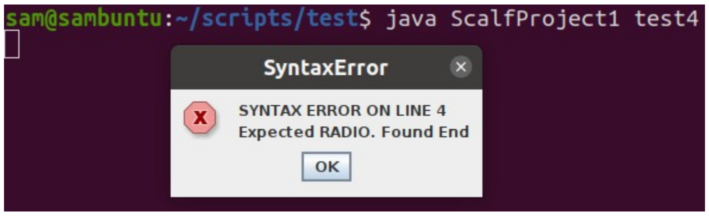
Test Case 4: Expected Radio

This final test case was to verify that the recursive methods functioned properly, ensuring that at least one radio was provided. I very well could have used the widgets for an example, but decided on the sub-method for radio buttons, instead.

Input File

```
Window "Test4" (200,300) Layout Grid(3,0):
     Label "Need At Least One Radio";
     Group
     End;
End.
```

Results



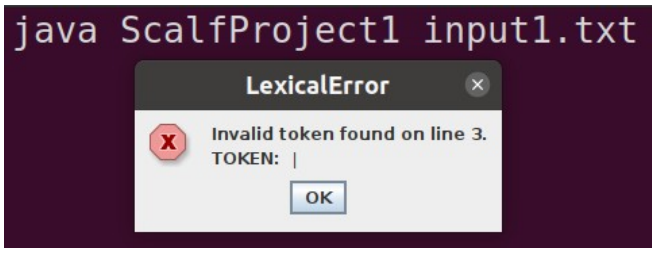
```
sam@sambuntu:~/scripts/test$ java ScalfProject1 test4
```

SyntaxError

SYNTAX ERROR ON LINE 4
Expected RADIO. Found End

Professor's Input Files

input1.txt

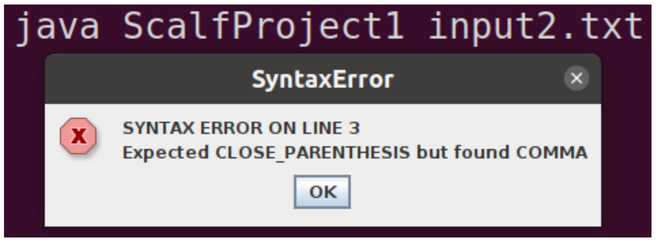


```
java ScalfProject1 input1.txt
```

LexicalError

Invalid token found on line 3.
TOKEN: |

input2.txt

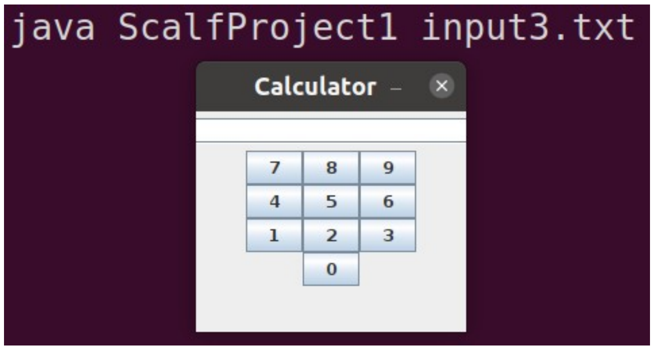


```
java ScalfProject1 input2.txt
```

SyntaxError

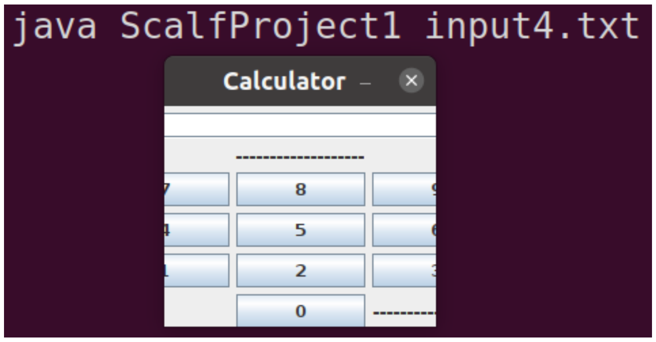
SYNTAX ERROR ON LINE 3
Expected CLOSE_PARENTHESIS but found COMMA

input3.txt



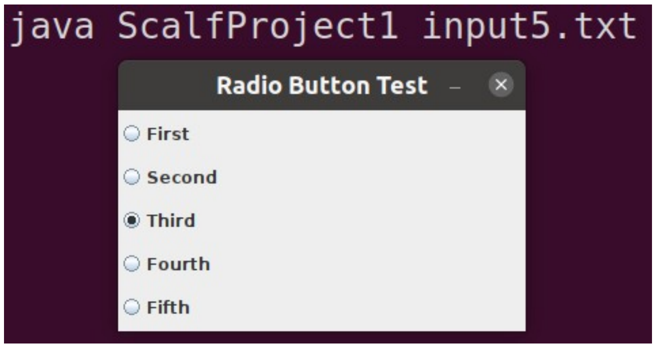
```
java ScalfProject1 input3.txt
```

input4.txt



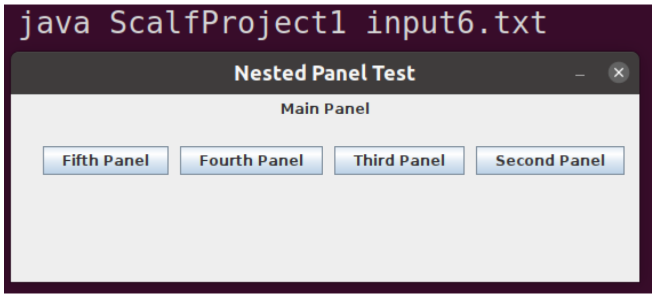
```
java ScalfProject1 input4.txt
```

input5.txt



```
java ScalfProject1 input5.txt
```

input6.txt



```
java ScalfProject1 input6.txt
```