

Overview:

In this practical, we were asked to write a compiler, to translate a small fragment of the Logo programming language into the PostScript language. We successfully translated Logo into PostScript, and generated code in the form of `.gs` files. We also implemented a comprehensive error system, that checks not only for parsing errors, but also for unrecognised variable and procedure names. Finally, we attempted to write our own fractals in Logo, with moderate success.

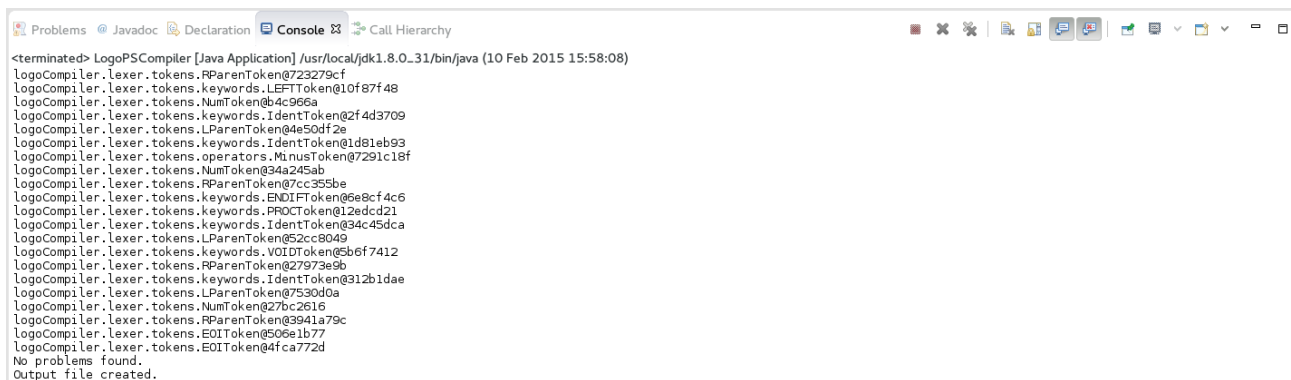
Design:

Lexer

Initially, we read our Logo code character by character, skipping whitespaces and using the first character after the whitespace to search for the token. However, we decided this was not a good implementation. It required us to use many conditionals to check possible configurations without white space, for example `LEVEL>1`, and made the code harder to read, especially since a new character had to be retrieved in multiple locations in the lexer.

Therefore, we decided to process the file, separating special symbols like `(` or `>` with spaces. In this way, we were able to use a Scanner object to simply read one token at a time, without worrying about the problems previously mentioned. The consequence to this was that we could no longer track the line number of the tokens because we ignore new lines together with whitespaces and tabs. To solve this, we replaced all “`\n`” (newline) with our own representation of new line, `N_E_W_L_I_N_E`. Every time the lexer comes across this string, it will ignore it but increase the line number.

In both implementations, we used a switch statement to check whether or not the token was a grammatical part of the language, like `IF` or `PROC` or if it was an identity or number. To distinguish between the two, we tried to parse the string as an integer, returning an integer token if the parse was valid, an identity token otherwise. We created a class for every grammatical token, as it made it easier for the Parser to process different token subclasses.



```
<terminated> LogoP5Compiler [Java Application] /usr/local/jdk1.8.0_31/bin/java (10 Feb 2015 15:58:08)
LogoCompiler.lexer.tokens.RParentToken@723279cf
LogoCompiler.lexer.tokens.keywords.LEFTToken@10f87f48
LogoCompiler.lexer.tokens.NumToken@b4c966a
LogoCompiler.lexer.tokens.keywords.IdentToken@2f4d3709
LogoCompiler.lexer.tokens.LParentToken@4e50df2e
LogoCompiler.lexer.tokens.keywords.IdentToken@1d81eb93
LogoCompiler.lexer.tokens.operators.MinusToken@7291c18f
LogoCompiler.lexer.tokens.NumToken@34a245ab
LogoCompiler.lexer.tokens.RParentToken@7cc355be
LogoCompiler.lexer.tokens.keywords.ENDIFToken@6e8cf4c6
LogoCompiler.lexer.tokens.keywords.PROCToken@12edcd21
LogoCompiler.lexer.tokens.keywords.IdentToken@34c45dca
LogoCompiler.lexer.tokens.LParentToken@52cc8049
LogoCompiler.lexer.tokens.keywords.VOIDToken@5b6f7412
LogoCompiler.lexer.tokens.RParentToken@27973e9b
LogoCompiler.lexer.tokens.keywords.IdentToken@312b1dae
LogoCompiler.lexer.tokens.LParentToken@7530d0a
LogoCompiler.lexer.tokens.NumToken@27bc2616
LogoCompiler.lexer.tokens.RParentToken@3941a79c
LogoCompiler.lexer.tokens.EOIToken@506e1b77
LogoCompiler.lexer.tokens.EOIToken@4fca772d
No problems found.
Output file created.
```

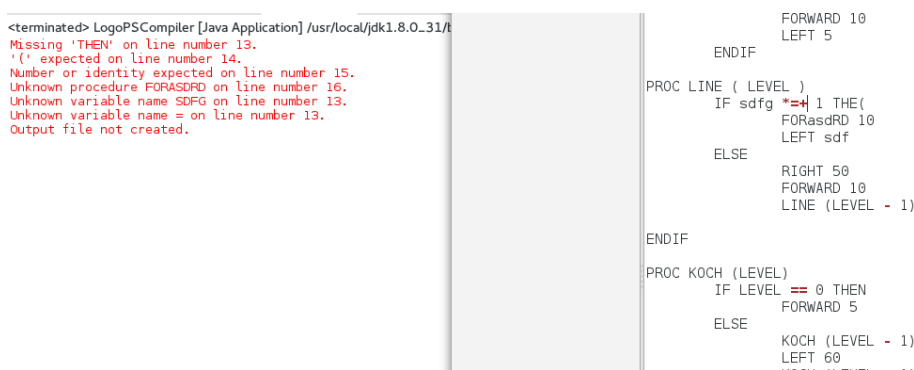
Screenshot 1: Each Token being printed on creation for debugging purposes.

We made a new class called **Dictionary** which stores all the keywords and deals with passing the correct tokens to the lexer. This makes the lexer much cleaner and easier to work with. In addition, if the compiler ever has to be extended in the future, it would be easy to simply go into the dictionary and add new keywords to it.

Parser

We started on the parser by using **if** statements in the **Proc** class to check for instances of certain tokens, so we can easily check if the stream of tokens follows this pattern. If it does not we can give appropriate error messages. We continued to do this for every statement inside **Proc**, going all the way down to expressions. For expressions, we implemented the Fraser and Hanson system of precedence which was provided, to generate code for them in the appropriate order.

To handle errors, we decided to simply print them to the default system error stream, since it's likely that a compiler would be run from the console. We added errors for unexpected and expected tokens, as well as unrecognised variable and procedure names, which are checked after the grammar check.



```

<terminated> LogoPSCompiler [Java Application] /usr/local/jdk1.8.0_31/rt
Missing 'THEN' on line number 13.
'(' expected on line number 14.
Number or identity expected on line number 15.
Unknown procedure FORASDRD on line number 16.
Unknown variable name SDFG on line number 13.
Output file not created.

FORWARD 10
LEFT 5
ENDIF
PROC LINE ( LEVEL )
  IF sdfg *== 1 THE(
    FORASDRD 10
    LEFT sdf
  ELSE
    RIGHT 50
    FORWARD 10
    LINE (LEVEL - 1)
  ENDIF
PROC KOCH (LEVEL)
  IF LEVEL == 0 THEN
    FORWARD 5
  ELSE
    KOCH (LEVEL - 1)
    LEFT 60
  ENDIF

```

Screenshot 2: Error Handling

Code Generation

From looking at the **PostScript** code, we were not sure about how to use the Arg register. Looking online, we first tried to use local variables with **dict**. It worked but we were told to only use the stack, so we tried again with Arg multiple times, but we got a stack overflow error because of the way the **Logo** was recursive.

Searching online, we found sample **PostScript** code and the commands they used. **dup** duplicates the top value of the stack and puts it on the stack. This lets us be able to keep a value on the stack while using another value to be used for the recursive procedures. This makes it so there is still the same variable on the stack that is not lost. For our final submission, we replaced **dup** with the Arg equivalent.

Testing:

To test that the lexer works, we have the tokens print themselves so we see that we are in fact getting the right stream of tokens. We also printed out the string that was read by the scanner so we can see if the right string has been taken by the lexer.

```
<terminated> LogoPSCompiler [Java Application] /usr/local/jdk1.8.0_31/bin/java (10 Feb 2015 16:01:30)
N_E_W_L_I_N_E
LogoCompiler.Lexer.tokens.keywords.PROCToken@12edcd21
MAIN
LogoCompiler.Lexer.tokens.keywords.IdentToken@34c45dca
{
LogoCompiler.Lexer.tokens.LParenToken@52cc8049
VOID
LogoCompiler.Lexer.tokens.keywords.VOIDToken@5b6f7412
}
LogoCompiler.Lexer.tokens.RParenToken@27973e9b
N_E_W_L_I_N_E
LogoCompiler.Lexer.tokens.keywords.IdentToken@312b1dae
{
LogoCompiler.Lexer.tokens.LParenToken@7530d0a
e
LogoCompiler.Lexer.tokens.NumToken@27bc2616
}
LogoCompiler.Lexer.tokens.RParenToken@3941a79c
N_E_W_L_I_N_E
LogoCompiler.Lexer.tokens.EOTToken@506e1b77
LogoCompiler.Lexer.tokens.EOTToken@4fca772d
No problems found.
Output file created.
```

Screenshot 3: Token generation along with the String read by the Scanner

For code generation testing, we had to first try manually translating the **Logo** into **PostScript** ourselves and compare it to what our compiler produced. The difficulty was in understanding **PostScript** ourselves to try and write it. After we got it working, adding the code generation to the compiler was quite easy.

Evaluation:

In regards to the initial specification, we believe we have successfully completed all requirements, and produced valid **PostScript** code that renders the required fractal designs. One small bug that we were unable to fix was that the line numbers written on the errors are sometimes inconsistent with the file numbers of the file.

Given more time, we would add more rigorous error checking, as well as learning to properly utilize the stack to optimize our **PostScript** code.

Conclusion:

Our compiler is able to compile correct **Logo** code into working **PostScript**, it also gives error messages if something is wrong with the **Logo** code and a line number of where the error occurred. We also wrote our own **Logo** programs, which compiled into **PostScript** successfully. We believe this satisfies the initial specification.