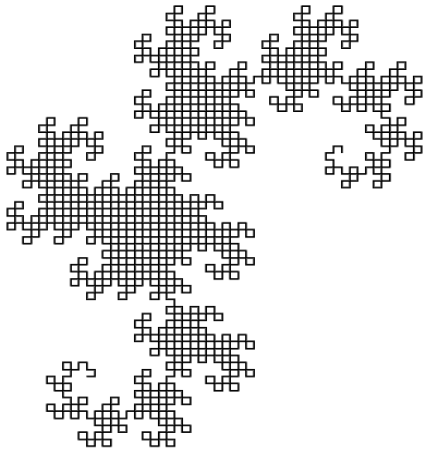
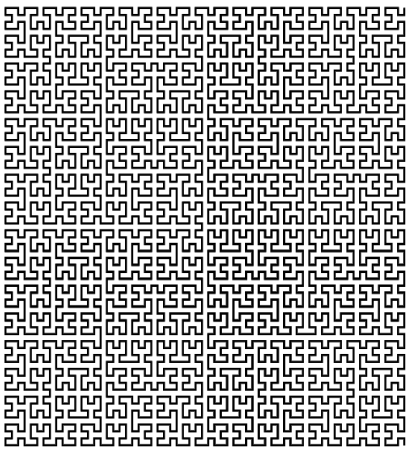
StudentA – 140005087
StudentB – 140001596 Tutor: ap637

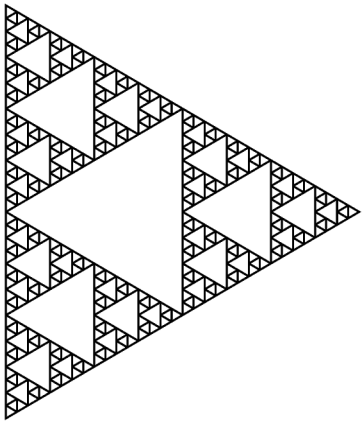
CS1006
Mini-Compiler

Overview:

For this practical, we were tasked with utilising all of the knowledge we've accrued thus far to construct a functioning compiler from logo to postscript. The logo code produces various fractal patterns in the postscript language, which required us to learn about how compilers worked in general. Given the sample code that was supplied, we were given subtle clues as to how to appropriately structure both the lexer and the parser in order for our compiler to produce legal, functional postscript code. To do this we had to first become familiar with the task of each part of the compiler, as well as how to construct an Abstract Syntax Tree (AST), a data structure with which we were previously unfamiliar with. This involved having a syntax analyser in the parser to detect whether or not the supplied code followed appropriate logo grammar. Initially, the logo code was individually split up into tokens by the lexer, which were later parsed into the AST one at a time. Once successfully parsed in accordance with proper logo grammar, the correct postscript code had to be outputted into a readable .ps file. If the postscript grammar was correct, the logo code will have then been successfully compiled into postscript, producing the following images:



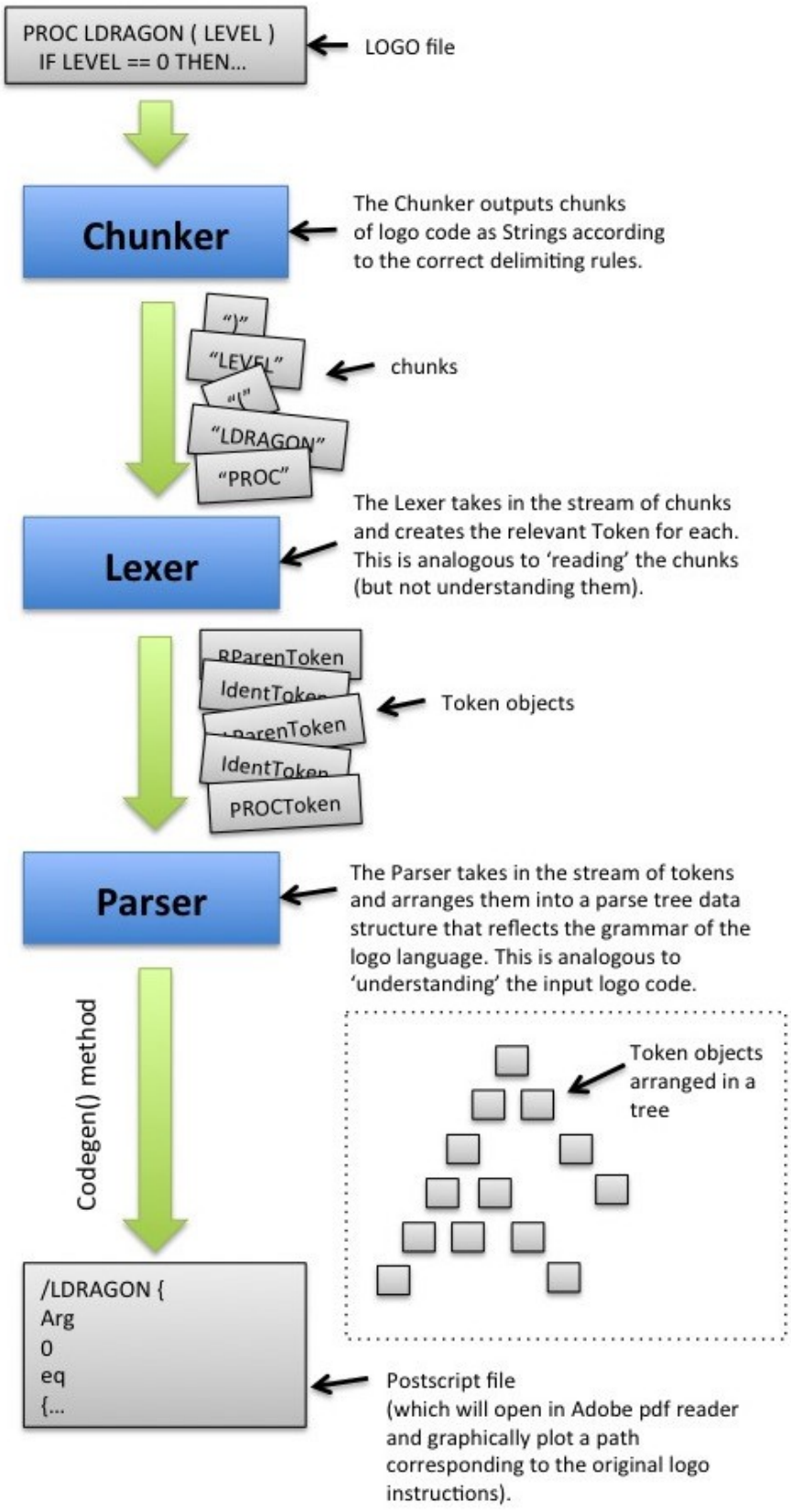




StudentA – 140005087
StudentB – 140001596 Tutor: ap637

Design:

This diagram shows the top level structure of our implementation. We have three main classes called Chunker, Lexer, and Parser. For further details please refer to the source code.



StudentA – 140005087
StudentB – 140001596 Tutor: ap637

Lexer Testing + Error Handling:

Since this practical was relatively involved in what it required us to do, we had to perform comprehensive tests in order to ensure that both our lexer and our parser were functioning as intended. After all, if the tokens were not being correctly produced or were being parsed in the wrong order, our compiler would cease to function. To do this, we were modular in our approach; we tested each bit one at a time before moving on to the next. This ensured that we were understanding the specifications of the practical as well as where errors were occurring, thus making it far easier to debug.

In order to test that the Chunker and Lexer classes were functioning correctly we created a class called test.LexerTest with its own main method. When LexerTest is run it prints to the terminal two columns of text. Below you can see the terminal output from LexerTest when 'Dragon.t' is the input logo file.

```
PROC            PROCToken
LDRAGON         IdentToken:"LDRAGON"
(               LParenToken
LEVEL           IdentToken:"LEVEL"
)               RParenToken
IF              IfToken
LEVEL           IdentToken:"LEVEL"
==              EqToken
0               NumToken:"0"
THEN            ThenToken
FORWARD         ForwardToken
5               NumToken:"5"
ELSE            ElseToken
LDRAGON         IdentToken:"LDRAGON"
(               LParenToken
LEVEL           IdentToken:"LEVEL"
-               SubToken
1               NumToken:"1"
)               RParenToken
LEFT            LeftToken
90              NumToken:"90"
RDRAGON         IdentToken:"RDRAGON"
(               LParenToken
LEVEL           IdentToken:"LEVEL"
-               SubToken
1               NumToken:"1"
)               RParenToken
ENDIF           EndIfToken
PROC            PROCToken
RDRAGON         IdentToken:"RDRAGON"
(               LParenToken
LEVEL           IdentToken:"LEVEL"
)               RParenToken
IF              IfToken
LEVEL           IdentToken:"LEVEL"
==              EqToken
0               NumToken:"0"
THEN            ThenToken
```

The left hand column is all of the chunks from the chunker and the right hand column is all of the tokens from the Lexer.

From this one can verify that the Chunker is correctly delimiting the logo file into the correct chunks and that the Lexer is correctly identifying these and turning them into the Tokens.

If the Lexer is passed a chunk which has an invalid format and it is therefore impossible for the Lexer to know what token to turn it into then a FormatException is thrown. This exception is passed the erroneous chunk on construction so that when the exception is caught at runtime in the main method then the details of that chunk can be shown in the error message.

StudentA – 140005087
StudentB – 140001596 Tutor: ap637

Parser Testing + Error Handling:

In the implementation of the Parser there are many places where the parser looks at the current token and checks (via an 'if' statement) if this is the type of token that is required next. If the current token is not of the required type then the parser throws an UnexpectedTokenException. This exception is passed the erroneous token on construction so that when the exception is caught at runtime in the main method then the details of that chunk can be shown in the error message. See the example below.

```
PROC LDRAGON ( LEVEL )
    IF LEVEL bananas 0 THEN
        FORWARD 5
    ELSE
        LDRAGON ( LEVEL - 1 )
        LEFT 90
        RDRAGON ( LEVEL - 1 )
    ENDIF
```

Here the '==' has been replaced by 'bananas'

```
Unexpected Token: IdentToken:"bananas"
```

When the main method is run this is the error message that is printed to the terminal.

This proved our error handling system worked, as well as proved our syntax tree adhered entirely with logo grammar.

The ultimate test of the Parser was that it successfully created postscript files that can be opened in adobe reader and correctly show the fractal paths.

Evaluation:

Given the requirements of the practical, we believe we achieved all of the specifications that were listed. Not only do our lexer and parser function with all of the supplied logo programs, it will also produce correct postscript code as an output. As an extension, we also implemented a functioning exception handling system which prohibits the use of undefined or unexpected tokens.

Conclusion:

Overall, we feel we have achieved a good deal in this practical. Despite learning about how compilation works, we learned a lot about how to code in java in general. Through the utilisation of exception handling, list iterators and a new data structure, we feel much more confident as programmers. Given how intimidating this project was to begin with, we're ecstatic that we were able to finish on time and with a project we can be proud of. The main challenge we encountered in this practical was a conceptual issue, rather than a technical one. Trying to understand what both the lexer and parser individually were tasked with doing proved to be fairly difficult. However once we overcame that hurdle, the coding structure followed suit relatively easy. Given more time, we would have loved to expand on our exception handling system by writing to the console where a specific error occurred in the logo file - for instance reporting the line number that the erroneous token occured. We also would have liked to write more logo code to develop different fractals. Despite those small additions, we are very happy with the state of our project.