Mini Compiler

Overview

The specification required that a program be produced which is able to turn Logo programming language into postscript (`PS-Adobe-3.0`). A document

Assumptions

Here are rules that were extrapolated from the provided Logo Grammar and examples of Logo Code.

1. Indentation is not important in the Logo language. This was extrapolated from `spiral.t`. So when parsing file indentation can be ignored, as it purposes are for convention only.
2. All commands must be separated by new lines.
3. All code in Logo language is ran inside of a `PROC` method. So the first word in any logo file must then be `PROC`.
4. Parameters are always in circular brackets when declaring method and calling method in statement.
5. All Logo files have a `PROC MAIN (VOID)` from which where all the code executes.
6. All calculation expressions run in brackets. Extrapolated from examples.
7. Logo doesn't support complex comparison since it doesn't support any `and` or `or` operators. Logo can only support `{mathematical expression} comparison_op {mathematical expression}`. So If statements can only contain one comparison operator.
8. Methods cannot return values.
9. Logo only supports identifiers in the form `a-z` or `A-Z` . So Logo only support identifiers in consistent casing.
10. `IF` statements are types of statements and `IF` statements can contain statements therefore Logo must s
11. You cannot have multiple `PROC` methods with the same name. Just extrapolated from conventional programming.
12. Logo can only support `PROC` methods with one parameter.
13. Since declared by the logo grammar there are only two types of expressions: `primary` and `binary`. Primary expression are: `identifier`, `num` whereas binary expressions have the form: `expression op expression`. This there for means logo supports complex mathematical expressions such as following: `( 3 * 9 + 4 * 2 ) - ( 4 * 2 + 6 * 5) * 4`. Recursive structure as binary expressions can contain binary expressions.

Problem Decomposition

The functionally In which the program requires has been split in to 3 separate sections.

Lexer

- Lexical Analysis
- Parsing

A Token is a element of a programming language. When a token is identified, certain characters or patterns of characters are expected afterwards. It would be unnecessary to create classes for every single possible type of token as certain tokens would automatically imply others. Such as `PROC` is always followed by an identifier naming the procedure. So using this we can then directly parse the input when certain Tokens are found.

All Logo Statements run inside of `PROC` so All statements must be children of there `PROC` tokens. So when a `PROC` declaration is found it following the all the proceeding code until it finds another `PROC` token belongs to that token. From this we can then have all `PROCToken` contain `StatementToken`. Methods Calls and `LEFT`, `RIGHT`, `FORWARD` behave similarly in Logo. e.g. they method has a name and expression that it takes in. So there behaviour can be generalised into a class. The `IF` statements in Logo behaviour is different from that of Method calls and `LEFT`, `RIGHT`, `FORWARD`. Since If statements contain statements and have conditions. So there behaviour would have to be a separate class. When a line begins with `PROC` it automatically identifies a `PROC` and then following input will be assume to be part of `PROC` declaration and then following code until another `PROC` is found shall be stored inside of this `PROCToken`. When a line begins with `IF` it shall automatically identify a IF token and expect a `THEN` to complete the statement and everything inside is assumed to be the conditions which are required for `IF` statement to run. Everything comes after is assumed to be part of that if statements of the `IF` token if all the conditions are met. When the program shall find a `ELSE` token. All the input proceed shall be assumed to be part of `IFToken` if condition has failed. The `IFToken` would then be close when `ENDIF` is found.

So in the Lexer Converts the Logo code to stream of `PROC` tokens. The `PROC` tokens contain the code which runs the method with it. Each type of logo command shall be made into a type of `Token`. This section of code shall also be responsible for the checking for that the logo file has the correct syntax as well. The lexer shall also be able to parse a logo file in which all the commands have not been separated by new lines.

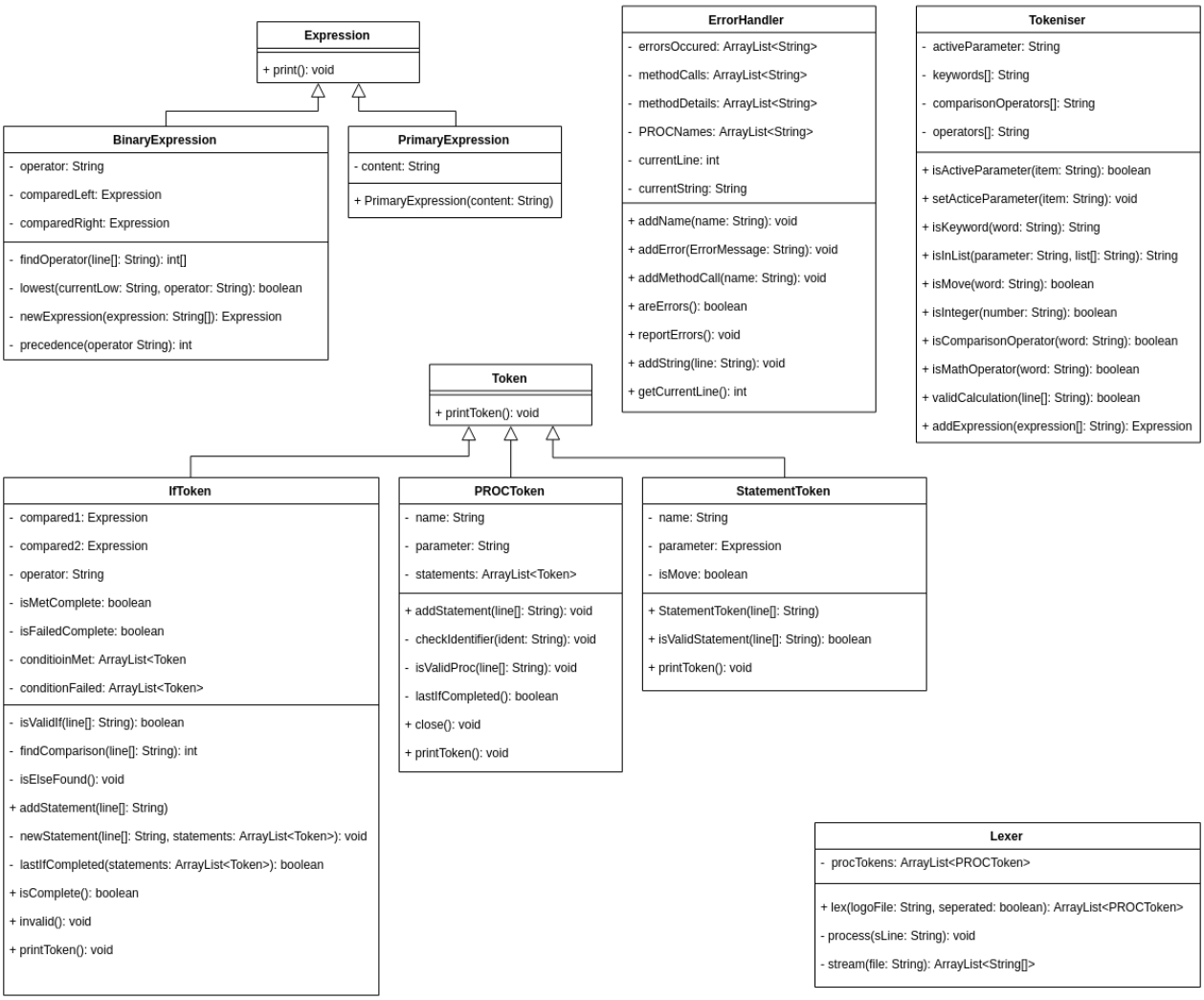
Parsing

- Code Generation

Gets all the commands that are required to be printed to the file from all the tokens which have been created. The steam of `PostScript` commands are printed to a `.ps` file with same name as inputted to program.

Design

Lexer



The following responsibilities have been assigned to the lexer package:

- Syntax Checking.
- Determining If `If statements` have been closed.
- identifier format checking.
- Check identifiers being used have been declared.
- `Tokenisation` of line.
- Binary Expression Tree Implementation.
- Determining if calculations are in the correct format.

Syntax Checking

When a specific token is identified all preceding characters much has a corresponding value or else it. So all these tokens which then allow for identification of preceding input were made into Token Classes:

- `IFToken`
- `PROCToken`
- `StatementToken`

From there the proceeding input could then be checked to see if it meets expected. The Error would be notified when there is deviation from this expectation.

Semantic Analysis

The program is able to determine if identifiers used in statements were actually declared before use. This would using the active parameter in the `Tokeniser` class. The class stores the `Identifier` name of the parameter which was declared in the `PROC` statement and if this so anything which does not have the same name or isn't a number must be incorrect placed in code.

Every method called inside all the `PROCToken` have their names stored in a list of method calls and details about that method are also stored. e.g. line number, the actual input. The list of method calls is then compared to the names of all the `PROCToken` and it is not in the list then a method which has not been declared is being ran.

Binary Expression Tree

The binary expression tree is used to 'break down' expressions that are present in the given LOGO code. The operator with the lowest precedence is made the root of the tree. The number before this operator and the operator of next highest precedence are made children of the root operator.

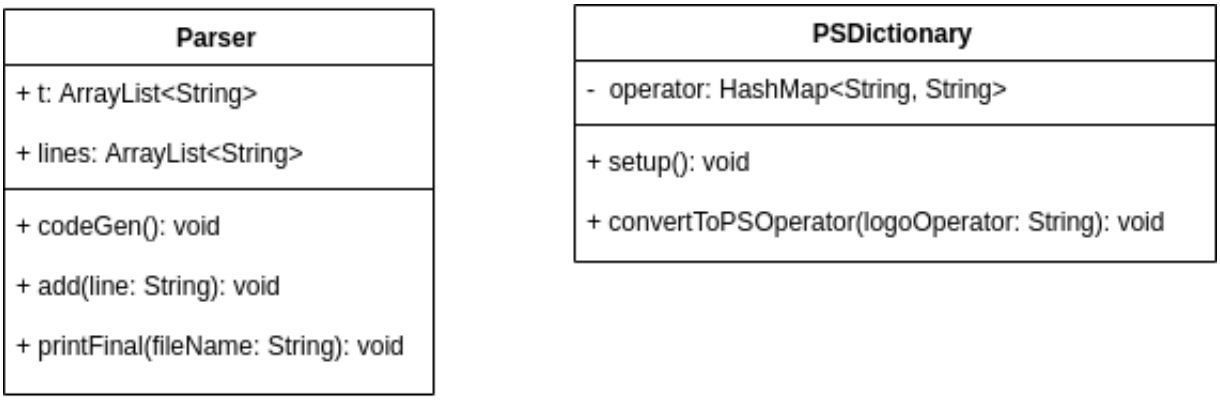
For example, if the expression was `(6 + 3 * 2)` the number `6` and the operator `'` would be made children of the root operator `*`.

The number, in this case 6, is the primary expression. The operator is then made a new expression that 'searches' for the operator of next highest precedence, in this case `*`. From this either another operator is found or the two remaining numbers are made children of the operator and the end of the binary tree is reached.

The Algorithm was also expanded to factor for in the event that a number is enclose in brackets

e.g. `( ( ( 1 ) ) )`

Parser



The following responsibilities have been assigned to the parser package:

- Initialising the conversion of `Logo` to `PostScript`.
- Storing all the `PostScript` that has to be printed to the `PostScript` File.
- Outputting the code to the `PostScript`.

Testing

Test Case 1:

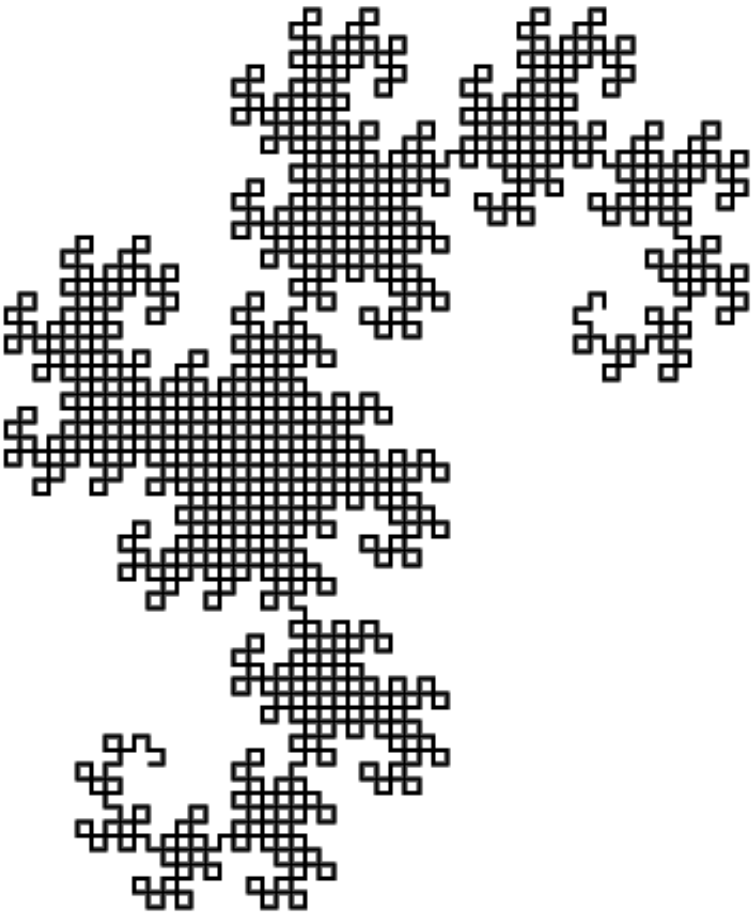
The first test was used to ensure that the Mini-Compiler program successfully converts Logo code into a valid post script file. In this case we used the `dragon.ps` file.

Expected Output

We expected a valid `.ps` file to be produced. When opened in a text editor it valid post script code should be shown. When opened in a file viewer a picture that represents a 'dragon' should be visible.

Actual Output

```
     @pc3-053-l:~/Documents/Semester-2/CS1006/Mini-Compiler/src $ javac LogoPSCompiler.java
     @pc3-053-l:~/Documents/Semester-2/CS1006/Mini-Compiler/src $ java LogoPSCompiler data/dragon.t
     @pc3-053-l:~/Documents/Semester-2/CS1006/Mini-Compiler/src $ cat data/dragon.ps
%!PS-Adobe-3.0
/Xpos    { 300 } def
/Ypos    { 500 } def
/Heading { 0   } def
/Arg     { 0   } def
/Right   {
Heading exch add Trueheading
/Heading exch def
} def
/Left {
Heading exch sub Trueheading
/Heading exch def
} def
/Trueheading {
360 mod dup
0 lt { 360 add } if
} def
/Forward {
dup  Heading sin mul
exch Heading cos mul
2 copy Newposition
rlineto
} def
/Newposition {
Heading 180 gt Heading 360 lt
and { neg } if exch
Heading  90 gt Heading 270 lt
and { neg } if exch
Ypos add /Ypos exch def
Xpos add /Xpos exch def
} def
/LDRAGON {
Arg
0
eq
{
5
Forward
} {
Arg
Arg
1
sub
/Arg exch def
LDRAGON
/Arg exch def
90
Left
Arg
Arg
```



As you can see from the above outputs, the program successfully produces a valid post script file and thus a picture/fractal of, in this case, a 'dragon'.

Test Case 2:

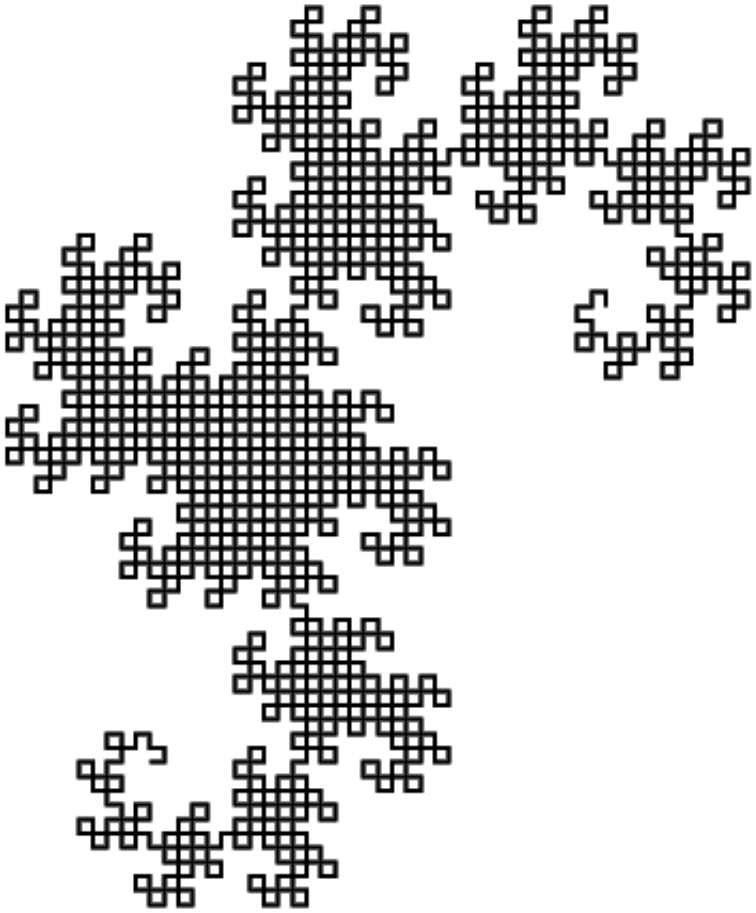
This test was used to ensure that the Mini-Compiler could 'handle' Logo code if it was presented in a single line.

Expected

The program should produce a valid `.ps` file and thus a picture/fractal of a dragon.

Actual

```
     @pc3-053-l:~/Documents/Semester-2/CS1006/Mini-Compiler/src $ java LogoPSCompiler data/dragonLine.t -n
     @pc3-053-l:~/Documents/Semester-2/CS1006/Mini-Compiler/src $ cat data/dragonLine.ps
%!PS-Adobe-3.0
/Xpos    { 300 } def
/Ypos    { 500 } def
/Heading { 0   } def
/Arg     { 0   } def
/Right   {
Heading exch add Trueheading
/Heading exch def
} def
/Left {
Heading exch sub Trueheading
/Heading exch def
} def
/Trueheading {
360 mod dup
0 lt { 360 add } if
} def
/Forward {
dup  Heading sin mul
exch Heading cos mul
2 copy Newposition
rlineto
} def
/Newposition {
Heading 180 gt Heading 360 lt
and { neg } if exch
Heading  90 gt Heading 270 lt
and { neg } if exch
Ypos add /Ypos exch def
Xpos add /Xpos exch def
} def
/LDRAGON {
Arg
0
eq
{
5
Forward
} {
Arg
Arg
1
sub
/Arg exch def
LDRAGON
/Arg exch def
90
Left
Arg
Arg
1
```



The program successfully produces a valid `.ps` file and, in this case, a picture/fractal of a 'dragon'

Test Case 3:

This test case was used to ensure that the Mini-Compiler program handles errors, produces useful error information, including line numbers, 'recovers' from this error and continues to try and compile the rest of the program, providing further error information if applicable.

Expected

The program should show multiple errors and their relevant information. For each error the following information should be shown: Description of the error, the line number where the error occurs and a 'copy' of the section around the error.

Actual

```
@pc3-053-l:~/Documents/Semester-2/CS1006/Mini-Compiler/src $ java LogoPSCompiler data/flowerError.t
Missing Elements to PROC declaration - LINE 1:  PROC FLOWER ( LEVEL
Unexpected Tokens - LINE 1:     PROC FLOWER ( LEVEL
LOGO only supports one comparison operator - LINE 2:        IF LEVEL === 0 THEN
Invalid Calculation - LINE 3:           FORWARD RIGHT 0
Unexpected Tokens - LINE 4:         ELSE {
Invalid Calculation - LINE 5:           FORWARD 2 LEVEL
Method must be called from within brackets - LINE 7:            LFLOWER   LEVEL -+= 1 )
Unexpected Tokens - LINE 8:         ENDIF }
) - Bracket expected - LINE 9:  PROC LFLOWER ( LEVEL 5 )
Unexpected Tokens - LINE 9:     PROC LFLOWER ( LEVEL 5 )
LOGO only supports one comparison operator - LINE 10:       IF LEVEL = 0 THEN
Invalid Calculation - LINE 11:          FORWARD - 0
Unexpected Tokens - LINE 12:        ELSE {
Invalid Calculation - LINE 13:          FORWARD LEVEL 2
Method must be called from within brackets - LINE 15:           RFLOWER ( LEVEL - 1
Unexpected Tokens - LINE 16:        ENDIF }
Missing Elements to PROC declaration - LINE 17:         PROC RFLOWERs LEVEL
Unexpected Tokens - LINE 17:    PROC RFLOWERs LEVEL
IF statement must end in then - LINE 18:            IF LEVEL == 0
Method must be called from within brackets - LINE 19:           FORARD 0
Method must be called from within brackets - LINE 23:           FLOWER  ( LEVEL - 1
Unexpected Tokens - LINE 24:        ENDIF )
IF Statements Must contain ELSE command - LINE 24:          ENDIF )
Missing Elements to PROC declaration - LINE 25:         PROC MAIN (VOID
Unexpected Tokens - LINE 25:    PROC MAIN (VOID
Method must be called from within brackets - LINE 26:      FLOWER   140 )
   @pc3-053-l:~/Documents/Semester-2/CS1006/Mini-Compiler/src $
```

The program successfully displays errors and their relevant information as described above.

Test Case 4:

This test was used to ensure that the binary expression tree worked correctly. The `dragon.t` file was edited and used to perform this test. The following lines were altered:

```
LDRAGON (LEVEL - 1 ) was replaced with
LDRAGON ( LEVEL -  ( 1  * 5 ) + 4 + 10 - ( 2 * 5 ) )

RDRAGON ( LEVEL - 1 ) was replaced with
RDRAGON ( LEVEL - ( 1 + 1 - 1 + 1  - 5 * 2 + 9) )

LDRAGON ( LEVEL -1 ) was replaced with
LDRAGON ( LEVEL - ( ( ( ( ( 1 ) ) ) ) ) )
```

Expected

If the binary expression tree works correctly, the `.ps` file that the compiler produces should be the same as the file produced by the original `dragon.t` file

Actual

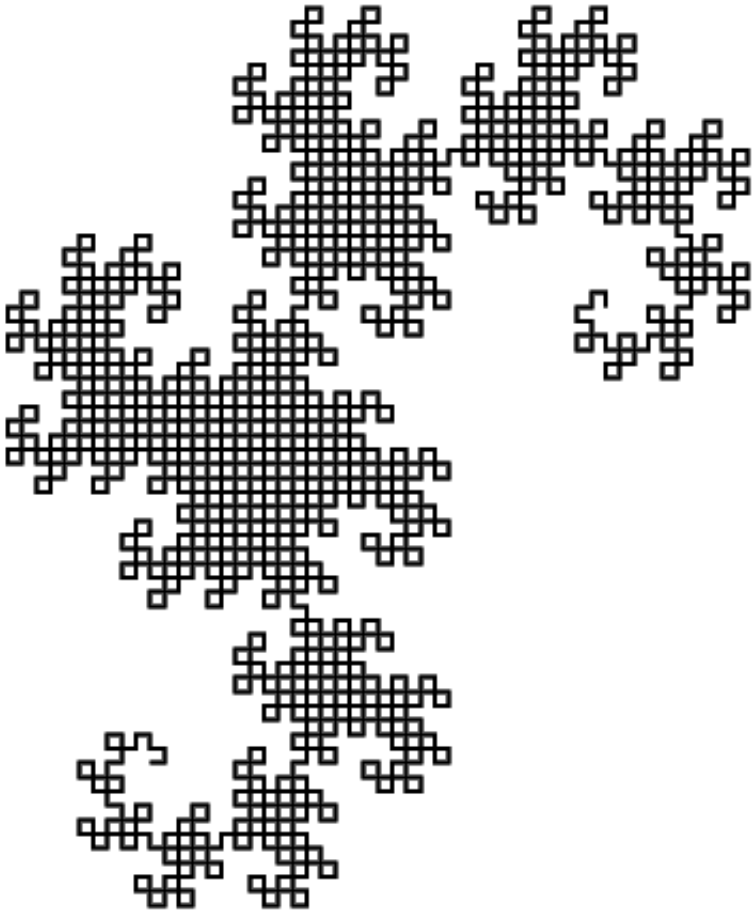
Edited LOGO program with complex mathematical expressions:

```
@pc3-053-l:~/Documents/Semester-2/CS1006/Mini-Compiler/src/data $ cat calc.t

   PROC LDRAGON ( LEVEL )
       IF LEVEL == 0 THEN
           FORWARD 5
       ELSE
           LDRAGON ( LEVEL -  ( 1  * 5 ) + 4 + 10 - ( 2 * 5 ) )
           LEFT 90
           RDRAGON ( LEVEL - ( 1 + 1 - 1 + 1  - 5 * 2 + 9) )
       ENDIF


   PROC RDRAGON ( LEVEL )
       IF LEVEL == (5 * 7 - 35) THEN
           FORWARD (50 - 45)
       ELSE
           LDRAGON ( LEVEL - ( ( ( ( ( 1 ) ) ) ) ) )
           RIGHT 90
           RDRAGON ( LEVEL - (50 - 49 * 1) )
       ENDIF

   PROC MAIN (VOID)
      LDRAGON ( 12 - (50 - 49) )
@pc3-053-l:~/Documents/Semester-2/CS1006/Mini-Compiler/src/data $
```

Post Script produced by the 'Mini-Compiler':

```
     @pc3-053-l:~/Documents/Semester-2/CS1006/Mini-Compiler/src $ javac LogoPSCompiler.java
     @pc3-053-l:~/Documents/Semester-2/CS1006/Mini-Compiler/src $ java LogoPSCompiler data/calc.t
     @pc3-053-l:~/Documents/Semester-2/CS1006/Mini-Compiler/src $ cat data/calc.ps
%!PS-Adobe-3.0
/Xpos    { 300 } def
/Ypos    { 500 } def
/Heading { 0   } def
/Arg     { 0   } def
/Right   {
Heading exch add Trueheading
/Heading exch def
} def
/Left {
Heading exch sub Trueheading
/Heading exch def
} def
/Trueheading {
360 mod dup
0 lt { 360 add } if
} def
/Forward {
dup  Heading sin mul
exch Heading cos mul
2 copy Newposition
rlineto
} def
/Newposition {
Heading 180 gt Heading 360 lt
and { neg } if exch
Heading  90 gt Heading 270 lt
and { neg } if exch
Ypos add /Ypos exch def
Xpos add /Xpos exch def
} def
/LDRAGON {
Arg
0
eq
{
5
Forward
} {
Arg
Arg
1
5
mul
sub
4
10
2
5
mul
```

Picture/Fractal produced by the 'Mini-Compiler'



As you can see. The fractal that the program produces is identical to that of the original `dragon.t` file. This shows that the binary expression tree works correctly.

Evaluation

The specification required that a LOGO to PostScript compiler be produced that correctly translates LOGO to PostScript. As you can see from the above (tests one and two), our program successfully translates LOGO code to PostScript. The compiler makes use of a single register (Arg) and utilises a stack. The program contains a gode generator which correctly 'prints' the translated LOGO code to a valid `.ps` file. The program also contains an error reporting system which produces useful error information when syntactically incorrect code is found. The program recovers from these errors and continue to parse the code until the end of the file as shown in the third test case.

As demonstrated above, the program that has been produced corresponds to the advanced deliverable as set out in the specification.

Conclusion

The provided code package was successfully used alongside the specification to produce a 'Mini-Compiler' that translates LOGO into PostScript. This involved successfully implementing a lexer, used to convert the LOGO program into a series of tokens, and a parser, used to 'parse' the generated tokens into the appropriate data structure, and a code generator that produces the appropriate PostScript code. An error handler was also successfully implemented that produces insightful information when syntactically incorrect LOGO code is 'entered'.

Difficulties

As we had to use a single register the value of arg had to be placed on the stack before we called a method. So Arg would be called then the parameters which are required to be inputed to the method would then be called and then the value of `Arg` is changed with `Arg exch def` then the method is called and final `Arg exch def` sets `Arg` to its original value which sits at the bottom of this stack.

With More Time

- Could have created our own fractals to test the program with.
- Implementation of For Loops in Logo could have been produced.