17 Feb 2016 CS1006 Practical 1
Student ID: Henry Hargreaves (150012773)
Alexandru Ungurianu (150015686)

Mini Compiler

Overview

The first practical required us to write a logo to postscript compile which was required to correctly translate a LOGO program into a PostScript program. The LOGO language is an interpreted language whereas postscript is a stack based language, making the task quite a challenge. The generated PostScript, should then have been able to be interpreted by another application or converted to a PDF. To ensure our program worked correctly we were supplied with three sample logo programs, each of which drew a fractal.

The compiler should implement a lexer and a parser which would parse anything written in the LOGO language. In addition it should also implement code generation functionality which would produce correct code. Finally, if the LOGO code inputted into the compiler had some syntax errors, the compiler should be able to recover from such errors, produce a 'helpful' error message and continue to parse the code until the end if possible.

Design

Lexer

Lexing the program is the first phase of the compilation process. This is done in our program by the *Lexer* class which takes a *FileInputStream* and yields the LOGO-specific tokens from that file.

Each token has it's own class (e.g. *IFToken*, *IdentToken*). Most of the tokens just match to a string, like ELSE or PROC, and these tokens just inherit the *Token* abstract class. This class contains methods for getting the representation of the token in the LOGO language (e.g. PROCToken -> "PROC") and a line number to allow the user to easily track down the error. In addition to those tokens we have *IdentToken* and *NumToken*, which hold identifiers and numbers respectively.

While designing the parser and the error handling for our compiler, we discovered that some of our tokens need to have some kind of flag to show that they can be used in some special cases. This is implemented by creating a list of traits for each token, which take values from the *TokenTrait* enum. These traits are easily accesible using the hasTokenTrait method on a token. This trait system was mostly added to avoid code duplication, as the trivial solution was using an if else chain.

Student ID: Henry Hargreaves (150012773)
Alexandru Ungurianu (150015686)

Currently we have 3 traits:

- VALID_STATEMENT_BEGINNING - which states that a token can be a beginning of a statement (e.g. IF, FORWARD)
- SYNCHRONIZATION_TOKEN - which is used in error recovery
- BOOLEAN_OPERATOR - assigned to boolean operators, this is used to make sure a if condition is valid.

Another special set of tokens are the operator tokens, which inherit from the *OperatorToken* abstract class. Although there are still static tokens (e.g. <=, +, /), they require some extra methods: *precedence*, used for expression parsing and *getPsOpCode* which is used to generate the respective postscript code for each operation.

The precedence levels for the tokens are the following:

- Level 0 for all the operators that aren't involved in expressions. This is set in the *Token* class.
- Level 1 for the boolean operators (e.g. *LessThanToken*, *EqualsToken*)
- Level 2 for addition and subtraction (*PlusOpToken*, *MinusOpToken*)
- Level 3 for multiplication and division (*MultOpToken*, *DivOpToken*)

A special token is the *EOIToken* which represents the end of input, which for our programs is the end of file.

We also added an *ErrorToken* that is mostly placed in unreachable conditions and is used to hold an invalid character or string to be shown to the user for better error recognition, as we considered this to be a better solution that just returning *null*.

Parser

Syntax analysis or parsing is the second phase of the compilation process. Code is examined token by token, to ensure it follows the rules of the language. To do this the parser accepts a stream of tokens generated by the lexer. Following this the parser converts them into their respective syntax trees. Most the data structures created by the parser deal with the non-terminal grammar elements and have appropriate classes.

The parser after initializing the lexer calls the *parse* method, which in turn will return a Prog object, this object being the root of the syntax tree for our program. Each object in the parser package represents a building block of the program and knows how to parse itself; this is enabled through the static *parse* function.

The parser is also where most of the error checking takes place. This is done mainly using if and else statements as well as our own custom built exceptions. This has enabled us to define the way in which exceptions are handled.

Student ID: Henry Hargreaves (150012773)
Alexandru Ungurianu (150015686)

Some of the statements require less complex code to parse as they only take a value from token put into and put it into an identifier. However, other statements such as *IF* or *PROC* are more difficult to parse due to their complex syntax. All the statment classes inherit from the *Stmt* (statement) class, which is an abstract class. This statement class is the parent of all the statement classes (like *Forward*, *Right* or *Procedure Call*). One of the issues we encountered was in identifying a procedure call when parsing statements. To overcome this we decided to choose as an initial token for this statement, the identifier token.

We are also pass the argument that we get in a procedure down the parse tree, so that both procedure calls and identifier expressions can access it, as that argument represents the variable name in the current scope. This allow for recursion handling and for checking used variables against this.

Code generation

After the parsing stage has completed successfully, the generated parse tree is converted into actual PostScript code. The code generation starts in the LogoCompiler class. Firstly an initial block of code is written to the output file which contains the header that identifies the file as being a PostScript file, and some definitions of variables and functions that are used by the actual generated code. After that, the codegen function for the Prog object (the root of the syntax tree) is called, which in turn propagates the code generation down the tree. Most of the statements were easily translated into PostScript code, like IdentExpr, NumExpr and the movement statements, but some presented somewhat of a challenge.

This challenge was given by the fact that our program should handle recursive calls flawlessly. The impediment in doing that is that the definitions dictionary in PostScript is global, and not local to the scope of the procedure. After some research online, we found that one such local dictionary can be created using the PostScript sequence "*1 dict begin*" at the beginning of a procedure, matched by an "*end*" at the end of the procedure. This fix got us through most of the implementation of error handling and we assumed it was correct. But after we generated the dragon fractal and compared it to a sample found online, we realised that our dragon was completely wrong, while the other sample fractals worked with no apparent issues.

After tracking down the problem to recursion handling, we decided to swap the "*1 dict begin*" strategy with a manual one. For each procedure call statement, we push the value of the argument on the stack, right before the evaluation of the expression for the procedure and we pop it as soon as the procedure call was finished which saves the value of the argument in the current call frame. This approach worked great and our dragon was finally generated correctly.

Error handling

Because the parser was required to detect and report any error in the program, we have implemented multiple classes and custom exception handling to add error handling/recovery functionality. In our program when an error is encountered, depending on the type of error,

the parser should be able to handle it and carry on parsing the rest of the input. Although the parser is primarily responsible to check for errors, it is possible for various errors to be encountered at the different stages of the compilation process.

Due to the time constraints we have been unable to implement many complex algorithms to deal with all the errors that could occur during the different stages of compilation. The main types of errors that our program can handle are: Lexical errors, where the name of some identifiers are typed incorrectly and syntactical errors where the structure of the program is invalid. The only semantic error we are currently handling is when a variable name used in a procedure doesn't match with the one in the procedure prototype.

To enable our program to efficiently handle errors we have implemented a strategy known as panic mode. This means that when the parser encounters an error anywhere in the statement, it ignores future tokens until it finds a *synchronization token*. We can also recover from an ELSE token, by defining a separate parsing function that is only used in synchronization.

The parser is then able to skip bypass erroneous code by skipping tokens until a suitable spot to resume parsing is found e.g. a re-synchronizing token. This method for error recovery is an easy way to allow the program to continue parsing even when an error has been found, moreover it prevents the parser from developing infinite loops.

Extending LOGO

While researching about fractals in general, we discovered the L system, which is used to represent line fractals as the ones we were generating so far. While it was pretty easy to translate some of the fractals represented in the L system, like the Koch snowflake, we found that some of the more advanced fractals required a way to somehow save and restore the drawing state, represented by the (x,y) coordinates in the drawing space and the current heading. While trying to recreate those using just vanilla LOGO, using some kind of backtracking, we quickly found that to be error-prone, hard to implement or for all we know, impossible.

Due to these reasons, we changed our focus from LOGO to PostScript, to see if this language allows us to implement this functionality. Fortunately, this isn't the first time we were required to save a state, as this was also used for handling our procedure calls.
As such we added 2 more procedures into the PostScript prologue, *Save* and *Restore* which saves and restores the current state. The save is done by pushing the *Heading*, *PosX* and *PosY* variables to the stack. The restore was a bit more difficult. We have to assign the values from the stack to their respective variables, and also move the drawing position, using the *moveto* builtin. Although that seemed fine in theory, the program drew some of the lines with a strange offset.

In the end, we discovered this was generated by some of the initial code in the definition of the *NewPosition* procedure. Upon removing that, the fractals were generated correctly and we have yet to find any fractals that generate poorly due to the removal of that code.

17 Feb 2016 CS1006 Practical 1
Student ID: Henry Hargreaves (150012773)
Alexandru Ungurianu (150015686)

Testing

To test my program and make sure it works as expected I will create instances of each class and continue by making sure each method from every class is called and works as expected. Because there are not many instances where the user needs to input data into the program, I will not have to do to many verification and validation tests, so for example entering a number instead of a string for a user name. Moreover, we have also incorporated many try and catch error statements which allow my program to catch many other exceptions. However, we will, where possible, enter invalid data in an attempt to break the program, by for example adding invalid LOGO code to the test files and therefore perform both extreme and erroneous tests. If the program works as expected, the program will be able to deal with the exceptions thrown and continue parsing without crashing or returning inaccurate data. Testing the mathematical and logical operands in the program is also very important to ensure the data is being correctly manipulated.

Filename	Console output	Observations
dragon.t	Compilation successful.	The generated postscript file is correct.
hilbert.t	Compilation successful.	The generated postscript file is correct.
triangle.t	Compilation successful.	The generated postscript file is correct.
plant.t	Compilation successful.	The generate postscript file is correct. This files uses the SAVE and RESTORE statements.
wrongidentifier.t	Error at line 1: Expected PROC but got PROC3 Error at line 21: Argument "LEVELdse" not in scope Compilation failed.	Compilation failed, and the postscript file was not created. Here the error messages accurately describe the problem and give the correct line number.

Student ID: Henry Hargreaves (150012773)
Alexandru Ungurianu (150015686)

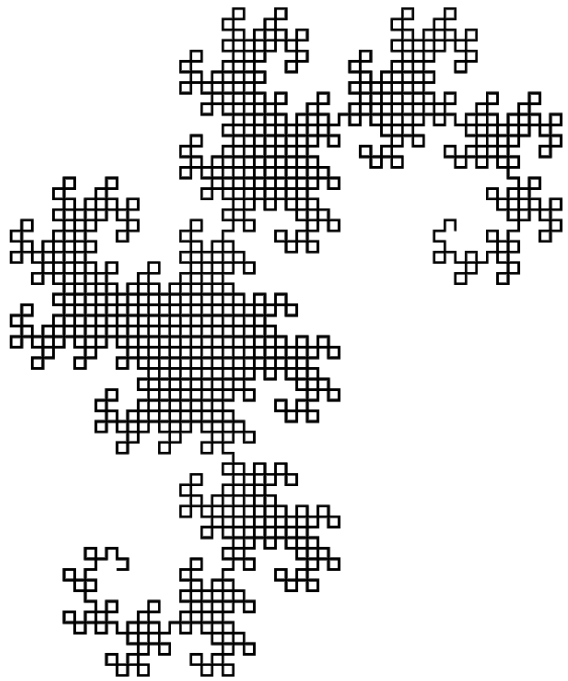
badifgoodelse.t	Error at line 19: Expected (but got LEVEL Error at line 33: Expected number or identifier but got ENDIF Compilation failed.	Compilation failed, and the postscript file was not created. Here the error messages accurately describe the problem and give the correct line number.
badexpressions.t	Error at line 2: Expected a binary expression in if condition Error at line 20: Expected ELSE but got 1 Compilation failed.	Compilation failed, and the postscript file was not created. Here the second error message is less helpful as the program should not be expecting an else. However the line number where the error occurred allowing a user to review the line.
nonbooleancond.t	Error at line 3: Expected a binary expression in if condition Error at line 6: Expected number or identifier but got + Error at line 13: Expected a boolean operator in if condition Error at line 18: Expected) but got 1 Compilation failed.	Compilation failed, and the postscript file was not created. The only ambiguous error is the last number. The message is a bit off point, but it is still helpful as it points to the erroneous line.

17 Feb 2016 CS1006 Practical 1

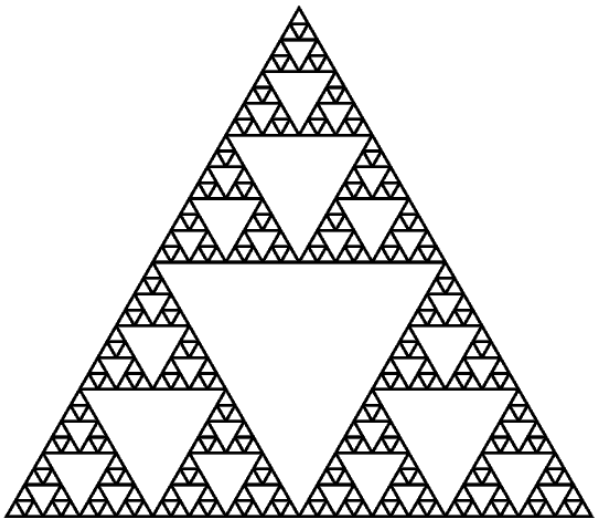
Student ID: Henry Hargreaves (150012773)
Alexandru Ungurianu (150015686)

Examples

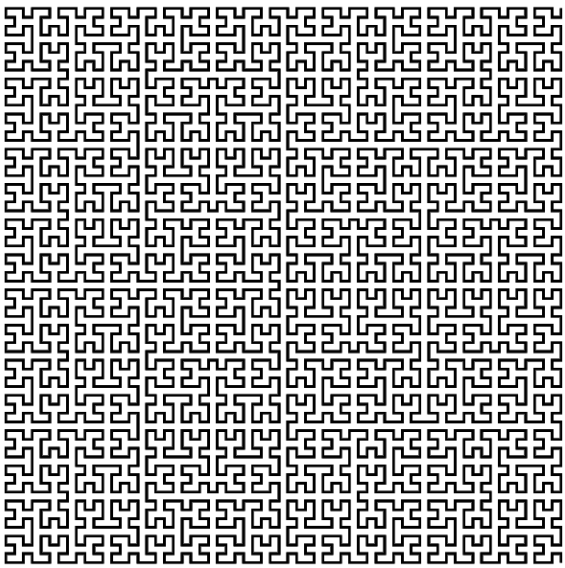
Dragon



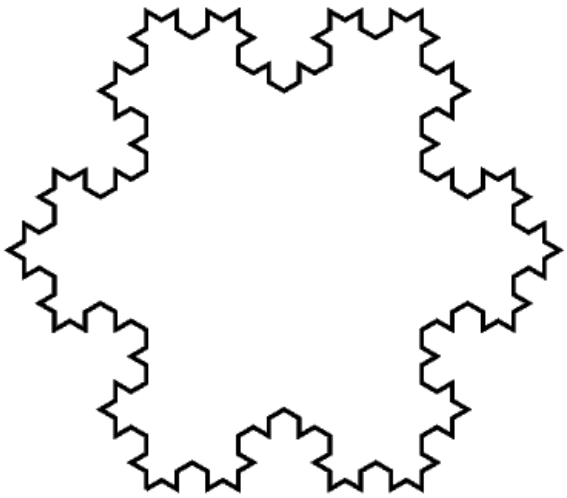
Triangle



Hilbert



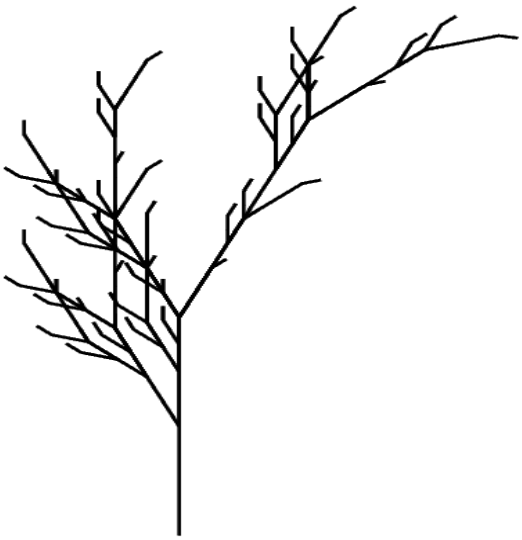
Koch



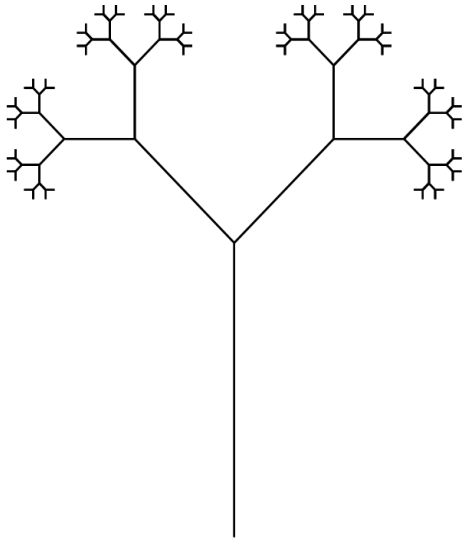
17 Feb 2016 CS1006 Practical 1

Student ID: Henry Hargreaves (150012773)
Alexandru Ungurianu (150015686)

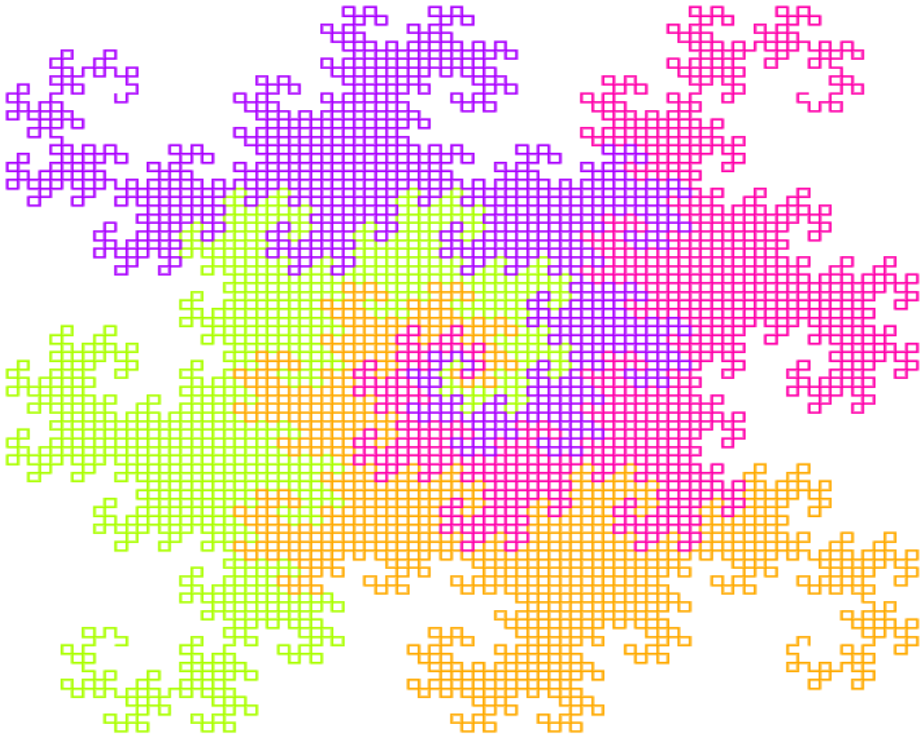
Plant



Pythagora



4 dragon curves rotated 90 degrees



17 Feb 2016 CS1006 Practical 1

Student ID: Henry Hargreaves (150012773)
Alexandru Ungurianu (150015686)

Evaluation

The program we designed meets all the initial requirements successfully as well as incorporating various different extensions. After extensively testing the core functionality of the program, we are pleased to report that the program works as expected, with the lexer and the parser correctly parse anything written in the LOGO language and the code generation functionality producing the correct postscript code. Furthermore our program is able to recover from syntax errors in the input LOGO code and produce an error message, telling the user on which line the error occurred, the statement that the parser expected and the statement the parser received. In the event of an error the parser uses a technique known panic mode, to attempt to recover from the error.

Conclusion

To conclude I am happy with the solution we have created to the given problem. The code is efficient and neat, providing all the required functionality and more. During the development stages, we encountered numerous problems, one of the main ones occured during parsing stage where the issue revolved around recursive calls. Because in the LOGO language all the identifiers are declared in a global dictionary, every call to a function would change a global variable instead of a local one, resulting in stack overflows when we tried to translate the postscript code. To fix that, we pushed the value of the current procedure argument on the stack right before each procedure call, and then popped it off the stack and reassigned it to its identifier after the procedure's execution has ended. This method allowed our program to successfully draw the fractals defined in the LOGO programs.

Individual Contribution

For most of the code we worked together. We used peer programming for the creation core functionality, due to macerial going down we used my partner's logon, hence why the repository looks some what one sided. My partner implanted most of the extension functionality, I helped where I could, but otherwise I was I watching and learning, as the extensions where some what complex and a little outside of my programming knowledge.