UNIVERSITY OF THE WEST INDIES
Department of Computing
COMP3652–Language Processors
Semester I, 2013
Lecturer: Dr. Daniel Coore

Assignment 2

Due Monday, November 18, 2013

Introduction

For this assignment, you will be completing the implementation of an interpreter for FRACTAL, a language for creating and combining simple fractal shapes. The language specification has been provided in a separate document. You will need to consult that document in order to complete this assignment.

Although graphics are involved in this assignment, you will not need to implement your own graphics routines, nor to create your own classes to manage the drawing of shapes to the screen. A library containing a representation of the turtle (a class called `Turtle` as well as a graphical display class (called `TurtleDisplay` have been provided in a jar file called `cs34q.jar`. The `TurtleDisplay` instance must be registered as a listener to the `Turtle` instance in order for it to display the trace of the turtle, but this will be taken care of for you.

Generically, an interpreter for FRACTAL operates on a context (instance of `FractalState`) and returns a value (an instance of `FractalValue`). The most likely value types that you will encounter have already been implemented for you in the `fractal.values` package. There you will find the abstract class `FractalValue` as well as a number of sub-classes. You may create new ones if you find that you need them.

You interpreter will be implemented in the class `FractalEvaluator` and it should extend the abstract class `AbstractFractalEvaluator`, which takes care of some top level issues for you, such as creating a JFrame instance to display the `TurtleDisplay` and facilitating interaction with the command line. The implementation of `AbstractFractalEvaluator` maintains a persistent state that is affected by each program that it evaluates, and is made available to the next such program. That way, programs can be entered in chunks of top level statements at a time, from the command line.

The `main` entry point is provided in the `Repl` class. It is implemented to take advantage of the persistent state provided in `AbstractFractalEvaluator`. When using `Repl` to test your interpreter, you must tell `Repl` which interpreter class you wish to use. To do this, pass the (fully qualified) name of the evaluator class as your first argument to `Repl`. All subsequent arguments will be treated as file names to be read in by the interpreter before any input is read from standard input. If you pass no arguments, `Repl` will default to using the class `fractal.semantics.FractalEvaluator`, and will immediately read from standard input (since no files were provided). For example, to use the class `FractalEvaluator` as your interpreter to evaluate the file `examples/gosper.fal`, and then read from standard input you would type:

invoke `fractal.sys.Repl fractal.semantics.FractalEvaluator examples/gosper.fal`

Here the word *invoke* has been used to represent a method of launching the Java virtual machine with the necessary classpath. You can use the `java -classpath` command and supply a classpath explicitly, or you can make a `.jar` file, and invoke it with the `java -jar`.

Problem 1: *Syntax* **[20]**

a. Define a JLex specification in a file called `FractalLexer.jflex` that will generate a lexer for FRACTAL. Make sure to place the file in the `fractal/syntax` folder, and ensure that your class declares itself to be in this package so that it can be compiled correctly. **[10]**

b. Define a CUP specification for the grammar of FRACTAL and place it in a file in the same folder as the lexer specification file. When running `cup` on this file, ensure that the class created is called `FractalParser` and that it is in the `fractal.syntax` package. The intermediate representations (abstract syntax tree classes) have (mostly) been defined for you. They are all in the `fractal.syntax` package. **[10]**

Problem 2: *Semantics* **[30]**

a. Complete the class `FractalEvaluator` started for you in the `fractal.semantics` package (it is basically empty, just enough was provided to produce a clean compilation).

You should implement all the missing methods so that it functions as a fully functional FRACTAL interpreter, with the exception of the two fractal composition operations. **For this assignment, you do not have to implement either the fractal sequence (`!`) or the fractal compose (`@`) operations.**

You will find that in order to implement `FractalEvaluator` you will rely on several external classes, most of which have been provided in entirety for you. All of the `ASTNode` subclasses have been provided, as well as a class called `FractalState` to maintain the interpreter's context as it evaluates each form. You are well advised to take a look at these classes, as well as the documentation for the classes in `cs34q.jar` while planning your implementation of `FractalEvaluator`. You should **not** have to edit `AbstractFractalEvaluator`, and you have not been provided with the source code for `Turtle` or `TurtleDisplay`, so you may not edit the behaviour of those classes either. You **may** edit `FractalState` if you find that it is necessary to do so.

You can test your interpreter on the example files provided in the `examples` directory, but you should not limit your tests to only those provided there. **[25]**

b. Use your interpreter to create a new FRACTAL program. See whether you can create something that produces unexpected results. (Have fun!). Submit both your program, as well as the image it produced (take a snapshot of the whole `FractalEvaluator` window). **[5]**

Problem 3: Bonus: *Procedures* **[15]**

As currently defined, FRACTAL contains no support for defining functions or procedures. Add a new special form to FRACTAL that would allow the user to define and call procedures with numerically valued arguments. The syntax of a call should look like the usual infix syntax (e.g. as used in C, Java and Python). The syntax of a procedure definition should be:

`proc` ⟨ *name* ⟩(⟨ *idList* ⟩) ⟨ *stmtList* ⟩ `end`

The list of identifiers may be empty, and the statements that may appear in the body of the procedure may be any that may appear at the top level (i.e. all statements except for `self`). Procedures do not return a useful value, so there is no need for a `return` statement.

Give an example of a procedure definition and its call in action to demonstrate that your implementation works. You might also think about implementing a conditional special form (e.g. `if`) along with a few arithmetic comparator operators to allow for recursive functions. That would open up the possibilities of some very interesting programs indeed!