UNIVERSITY OF THE WEST INDIES
Department of Computing
COMP3652–Language Processors
Sem I, 2019
Lecturer(s): Prof. Daniel Coore

Assignment 2

Due Tuesday, November 26, 2019

Introduction

For this assignment, you will extend some given starter code (adapted from `arithexp`) to produce an interpreter for FnPlot. In addition to the capabilities of the `arithexp` evaluator, by the time you are finished with this assignment, your interpreter should also be able to:

- Handle real numbers (i.e. floating point numbers). For example, your interpreter should be able to evaluate any of the following expressions:

```
1.2 + 2 * 3.4
0.005 * (2 + 5)
```

- Handle negative numbers (integers and floating point numbers) such as:

```
-1.2 * 3.4
-1 + 2 * 3
1 + -2 * 3
```

- Correctly handle function definitions, such as:

```
f = fun (x, y) -> 3*x^2 + 2*x + y
g = fun (x) -> 4*x^3 - 3*x
```

- Plot a function correctly on the given graphics display, provided all of its variables have values; e.g.:

```
plot g(x) for x in [-5:5]
let y = 2 in {plot f(x,y) for x in [-10: 10]}
plot f(3, y) for y in [-8 : 5]
```

If you are using an IDE to do this assignment, you may find it convenient to adapt its build process to also incorporate the necessary calls to `jflex` and to `cup`. The necessary configuration file for NetBeans, called `build.xml` has been provided for you as a starting point, if you use the NetBeans IDE. (You will still have to make some small adjustments to the file that I have provided to adjust the paths to match your installation of `jflex` and `cup`.)

Suggested Approach: The questions are structured along the phases of interpretation, but you do not have to implement your assignment in that way. You might find it useful to implement one language feature at a time, all the way through to the evalauator, and be able to test it, before trying to add the next language feature. To get this right, there might be times when you will need to create placeholder methods or classes that permit the project to compile, so that you can test the features that you are currently focused on. I recommend that you take this approach, because it will be less error prone, if you proceed in a disciplined way, but you may ignore this recommendation if you feel strongly about implementing each phase in order, with all of the language features being done at the same time.

Problem 1: FnPlot *Lexer* **[10]**
Modify the lexer specification (`FnPlotLexer`) to return tokens for:

- real numbers
- negative numbers
- the new keywords: `fun, mapsto, plot, for, clear`
- the new punctuation symbols: colon, left- and right- (square) brackets

Problem 2: FnPlot *Parser* **[25]**
Modify the `FnPlotParser.cup` file to contain new rules for the given special forms. Specifically, you will have to add production rules (and declarations) to support:

- real and negative numbers in arithmetic expressions
- function definition (which also requires arbitrarily long sequences of identifiers)
- function call (or application) (which also requires arbitrarily long sequences of argument expressions)
- the `plot` statement
- the `clear` statement

Problem 3: FnPlot *Intermediate Rep* **[10]**
Implement any new intermediate representation classes that you might need, and update the `Visitor` interface appropriately. Specifically, at a minimum, this will require creating classes to represent, and adding methods to handle, the forms for:

- function definition. The class `ExpFunction` in the `fnplot.syntax` package is to be used to represent this. (The implementation of `FnPlotFunction` which you will need to use for the next question refers to this class by this name.) So, all you need to do further is complete two methods in that class.
- function call
- the `plot` statement

- the `clear` statement

Depending on how you decide to go about representing some of those forms, you might need to implement one or more (hopefully only a few) additional supporting classes.

Problem 4: FnPlot *Semantics* **[25]**
Modify the `Evaluator` class to make it conformant to the new `Visitor` interface that you have defined, and so that it implements the specified semantics of each new form. Ensure that the former semantics for evaluating arithmetic expressions remains in tact, even after your modifications to handle real and negative numbers. Specifically, this will require you to (at a minimum):

To handle function definition, you should make sure that evaluating a function definition yields an instance of `FnPlotFunction`. That class was created specifically to represent functions in a statically scoped language. (You may, of course, make changes to that class if you deem them necessary, but I do not anticipate that you will need to do that.)

To handle graph plotting, the `Evaluator` class has already been modified for you to contain a member that is an instance of `GraphPlotter`, which implements the `Plotter` interface, previously described (in the accompanying language specification document for FnPlot).

You should ensure that your implementation of the method to handle the `plot` command respects the specified semantics by making the appropriate calls to the `GraphPlotter` local instance of your evaluator (you must first call `sample` to obtain the sample input values in an array, compute the corresponding output values in a separate array, and then call `plot` with the pair of arrays). If you get it right, executing a `plot` command in the provided `FnPlotFrame` interface should cause the appropriate graph to be drawn.