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

FnPlot Specifications

Introduction

The language FnPlot is just an extension of the `arithexp` language that we have been using as a case study throughout the course. It introduces (statically scoped) functions along with a special form to permit functions to be "plotted". The actual outcome of a `plot` command is left up to the output of a `plotter` object that will make calls back to the interpreter to obtain the values of the given function at various points in the specified domain. But that will be described in more detail below.

Functions

Functions are created with the special form `fun ... mapsto`, which may be abbreviated as `->`. The syntax is as follows:

`fun (`⟨ *id* ⟩`, ...) mapsto` ⟨ *expr* ⟩

The left hand side of the `mapsto` form specifies the (formal) parameters of the function and the right hand side is a single expression that represents the body of the function (note though that a `let` form is an expression. The value of the function at a point (in the space of all parameter values) is the result of the body evaluated in an environment where its parameters have been bound to the coordinates of the given point.

Functions are named in the same way that ordinary numbers may be named, using the already implemented assignment (or definition) operator, `=`. For example, to define `f` to be a function that adds 1 to its argument, we could do it as follows:

```
f = fun (x) -> x + 1;
```

Note that the semicolon at the end of the line is a statement terminator. It is not required for expressions, except when they are behaving like statements (e.g. in a sequence of statements).

Function Application

The syntax for function application is the typical syntax that is found in most modern infix format programming languages. It is:

⟨ *fn* ⟩`([`arg_1`, ..., `arg_n`])`

Here ⟨ *fn* ⟩ can be any atomic expression, and the argument list could be empty. So the following are all legal syntax for a function call:

```
f(1, 2)
g(x)
g(f(1,2))
h(x + 1, f(1, 2), g(4*f(g(1), x)))
(fun (x, y) -> x + y)(3, 4)
```

Plotting functions

The `plot` command allows us to produce a graph of an expression (usually a function) over a specified range of values of one variable. The syntax for the `plot` command is:

`plot` ⟨ *exp* ⟩ `for` ⟨ *id* ⟩ `in [`⟨ num_{lo} ⟩ `:` ⟨ num_{hi} ⟩`];`

The statement plots the graph of the value of the given expression when evaluated with the given identifier bound to a value drawn from the range inclusive of the two end points supplied. For example, the statement:

```
plot t + 1 for t in [-5 :  5];
```

Should plot a graph of $t + 1$ (y-axis) vs t (x-axis) over the range -5 to 5, inclusive. The actual sampled points within the specified range will be implementation dependent. (See below.)

The command `plot` is intended to plot the graph of a function over a subset of its domain. However, its actual behaviour is determined by the action of an external "plotter", a subclass of `Plotter`. Subclasses of `Plotter` support three methods: `clear`, `sample` and `plot`.

`clear()`**:** clears the plotter instance. In the case of a graphical display, this would erase the graphs that are currently displayed. For a text display, the clear command may not have any visible effect.

`sample(low, hi)`**:** returns an array of sample values between `low` and `hi`.

`plot(points)`**:** display a graph that interpolates the given points. Each point is a pair of (x, y) coordinates.

The plotter will determine, based on its context (e.g. hardware capabilities), how to sample the space of the plot, and return to the interpreter a collection of sample input values to be fed to the function. The interpreter, in turn, should use those input values to compute the function's corresponding output values, and then use the derived pairs of coordinates to call the `plot` method of the plotter to render the graph.

In practice the plotter could even just be a tabulator with no graphics capabilities at all (e.g. see the `TextPlotter` class), it depends on how the plotter implements the `plot` method. The implementation provided to you has a basic plotter implemented in the class `GraphPlotter`. See its implementation to see the methods that it supports, and how they were done, if you want more details.

Implementation

The starting point for the code that has been handed out was the implementation of `arithexp` that has been presented in lectures, and made available to you through OurVLE.

However, there are some notable changes, summarised as:

- All classes have been moved from the default package to appropriate hierarchical packages.
- A new class called `ASTNode` has been created to act as the parent of all intermediate representation classes. The classes `Exp` and `Statement` have been adjusted slightly as a consequence, and some of their subclasses have also been adjusted.
- A new generic class called `FnPlotValue` has been implemented as the parent class of all values that may be stored in a FnPlot variable. (More on this in the next section.)
- A class called `FnPlotFunction` has been provided as a subclass of `FnPlotValue` to represent runtime function objects.
- Classes have been developed to handle a graphical display and an interactive GUI for entering FnPlot code fragments to the interpreter. To get a front end user-interface, launch the `fnplot.gui.FnPlotFrame` class.

Values

The complete declaration of the `FnPlotValue` class is:

```
public class FnPlotValue<T extends FnPlotValue<T>> ...
```

To comprehend this, think of the `FnPlotValue` class as being a parent class and a wrapper class for some native Java types. The `FnPlotValue` class allows us to import a class in Java into the set of values that can be represented in FnPlot. But rather than just using an internal member of type `Object`, which would cause us to lose the type identity of the data member after the object was wrapped, we use specific subclasses to represent each sub-type. For example, the class `FnPlotInt` represents integers in FnPlot. It is declared as:

```
public class FnPlotInt extends FnPlotValue<FnPlotInt> ...
```

The use of the type variable, and its declared restriction of being a subclass of `FnPlotValue` forces the wrapped type to be a subclass of `FnPlotValue`, which means that we have control over what Java types can be imported into FnPlot. In other words, it helps to enforce the discipline that if we want to introduce a new type into FnPlot we must create a subclass of `FnPlotValue` explicitly to represent it. That also means that we are forced to implement certain methods that will ensure that the data type behaves well within the FnPlot world.

You should not need to introduce any new types, because the `FnPlotReal` class has been provided for you to represent real (floating point) values. However, the discussion above should come in handy for you when you have to implement an interpreter for a larger language with more data types in it.

Graphics

The code to support drawing a graph, and managing user interface events is already implemented for you in the `fnplot.gui` package. The primary class through which you interact with your interpreter is `FnPlotFrame`. It will pop up an interface that will receive commands, show a history of previous evaluations, and also show any graphs that you may have plotted. The GUI also contains some control widgets that allow you to scale and pan the graphing area.

The graphing area is an instance of `GraphingPanel`, which is provided in the `cs34q.gfx` package of the `cs34q-utils.jar` library. You do not need to concern yourself with the details of it (unless you are curious), because all of the interaction between the interpreter and the graphing area have been handled in the implementation of the `GraphPlotter` class of the `fnplot.sys.` package.

Summary of Operations and Commands

The following table presents a table of all expressions and commands that should be supported.

Form	**Explanation**
`+`, `-`, `*`, `/`, `^`	Binary operators: addition, subtraction, multiplication, division, exponentiation. These may be used to combine *any* two expressions (infix syntax).
Expressions	
`fun (`⟨ *id* ⟩`, ...) mapsto` ⟨ *expr* ⟩	Create a new function
⟨ $expr_f$ ⟩`(`[⟨ $\texttt{expr}_{a_1}$ ⟩`, ...]`)	Invoke (call) the function that arises from evaluating expression f, on some arguments, $a_1, \ldots$. Each argument expression is evaluated first, before invoking the function on their values.
`let` ⟨ *id* ⟩ `=` ⟨ *expr* ⟩ `[,`⟨ id ⟩ `=` ⟨ expr ⟩`, ...] in` ⟨ *expr* ⟩	Temporarily binds the given identifiers to the values of their associated expressions, and returns the value of the expression within that context.
`{`⟨ *stmt* ⟩`;` ⟨ *stmt* ⟩`; ...}`	Evaluate a sequence of statements or expressions, returning the value of the last one as the value of the sequence.
Statements	
⟨ *id* ⟩ `=` ⟨ *expr* ⟩	Assign the value of the expression to the given identifier (in the global scope).
`plot` ⟨ *exp* ⟩ `for` ⟨ *id* ⟩ `in [`⟨ num_{lo} ⟩ `:` ⟨ num_{hi} ⟩`]`	Plot the expression as a function of `id` over the given range.
`clear()`	Clear the graphics display.