TIPLPA Multi-Paradigm Assignment Project

Group 3:

“Jenny” Jung Min Kim (201210736), Rasmus Bækgaard (10893), Janne Elisabeth Walsted (jw68266)

# **Introduction**

This project concerns a multi-paradigm assignment of creating a graphing application for mathematical functions.

A multi-paradigm program is a program that uses more than one programming paradigm. This application consists of a user interface program making use of the object-oriented programming paradigm and a program for interpreting mathematical expressions making use of the functional programming paradigm.

**Assignment Description**

Create a graphing application for mathematical functions.

Application requirements:

* A user should be able to:
  + input a continuous function over numbers as a Scheme procedure,   
    e.g. (lambda (x) (\* x x)
  + specify the region that the graph will display (min and max for x and y)
  + specify whether scales are linear or logarithmic
  + specify how many data points should be plotted
  + choose to display multiple functions in the same chart
  + choose to display both the function and its derivative in the same chart
  + choose to display the area under the curve for a given range of a function using the “rectangle method”
  + choose the number of rectangles to be used to approximate the area
* Real numbers must be dealt with (but not complex numbers)
* Interface/GUI elements must be done in Java or C# (application interface)
* Mathematical calculations must be done in Scheme (application engine)
* The input function must be treated as an opaque string in the Java runtime
* All interpretation of the meaning of the input function must be handled by the Scheme runtime

The application must be tested for both the Scheme and the Java/C# part.

# **Architecture**

The graphing application for mathematical functions consists of two parts - a program containing the user interface and a program interpreting the mathematical expressions that the user inputs.

For interpreting mathematical expressions, Scheme, which is not purely functional but based on the functional programming paradigm, is used. Scheme is a dialect of the programming language List Processing (Lisp).

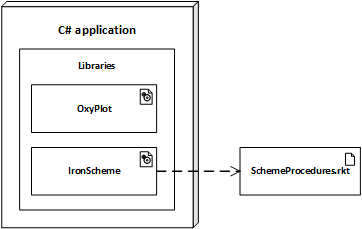
In Scheme the algorithms for interpreting the mathematical expressions are defined as procedures. All the procedures for the graphing application is gathered in a file that is used by the user interface program.

The user interface program is written in a C# application that runs on Microsoft’s .NET framework - this program is based on the object-oriented programming paradigm.

The program has a graphical user interface that allows the user to input information to the program. After having received input the program uses the file with the Scheme procedures, which do the calculations and returns the processed information. The processed information is then displayed to the user as a graph.

In order for the C# application to utilise the Scheme procedures a package called IronScheme is included in the application. IronScheme is an implementation of Scheme and runs on top of the .NET runtime.

Likewise a package called OxyPlot is included in the application. This package is an open source plotting library for .NET and enables the C# application to draw graphs when the Scheme script has returned the processed information.



# **Functionality**

# 1. Scheme Engine

The Scheme engine processes the input from the user. The main input argument is a Scheme procedure, e.g. (lambda (x) (\* x x)). It also receives other arguments, e.g. minimum and maximum values for the regions (axis boundaries) of which area the output graph should be display. The output from the Scheme engine will be the datasets of (x . y) coordinates, for which the zip implementation from a previous assignment has been used.

For the implementation of the Scheme engine there is four main areas:

### Mathematical functions

These take care of pure mathematics, mainly logarithmic, derivative, integral (there is also a linear function but it doesn’t require mathematical calculation. It returns function itself as y value in response to the right x value). Several utility (sub)functions have been implemented for the mathematical computations.

### Procedures as arguments

This is also requirement for Scheme engine. Some procedures are provided as arguments (input). The idea and intention behind this way of implementation is to generalize patterns using higher-order procedures. In this way it is possible to have one pattern for e.g. linear functions which can represent different linear mathematical formulas (equations) using the same pattern, e.g.

|  |
| --- |
| code : (define linearFunc  (lambda (xMin xMax count func) ;;“func” will receive “lambda (x) (....)” as input  ………. |

### Tail recursion

The functions must be displayed in the GUI in a certain region with minimum and maximum axis value specified by the user, therefore Scheme needs to process recursive logic to return the dataset for the given region. Tail recursion is used for the purpose through the functions (but not the mathematical subfunctions).

Fx tail recursion in linearFunc:

(define linearFunc

(lambda (xMin xMax interval func)

(letrec ( (helper-linear

(lambda (x i y)

(if (> i xMin)

(helper-linear (cons i x)

(- i interval)

(cons (func i ) y))

(zip x y))))

)

(helper-linear '() xMax '()))))

### Misc.

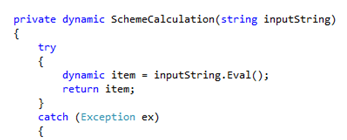
Return values for the dataset of coordinates uses the “zip” function from a previous assignment. Subfunctions for pure math calculations for logarithmic, derivative, rectangle method for integral is new implementations.

# 2. C# Interface

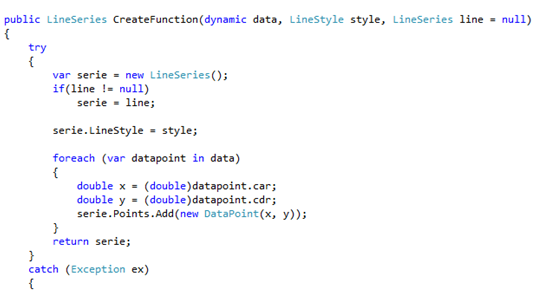
The C# application does no mathematical processing. The mathematical processing is done dynamically by Scheme, i.e. runtime when the user inputs the Scheme procedure in the graphical user interface provided by the C# application.

In order to do the actual calculations it must be specified which procedure is going to be processed, which arguments are going to be evaluated by the procedure, and where the description of the procedure is placed. The C # application does this by creating a string in which the information about what expression and which arguments is to be evaluated:  

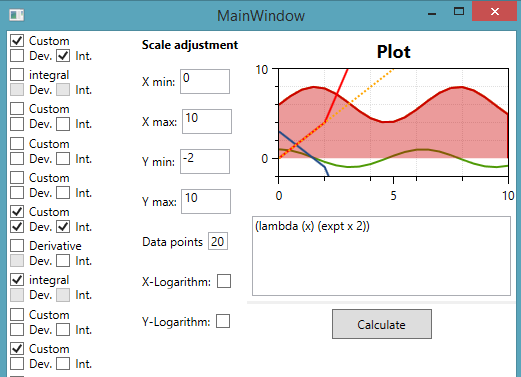

When the string has been created, a dynamic object for the Scheme engine return values is created. The string containing the information of which expression with corresponding arguments is evaluated and the result is stored in the dynamic item:



Finally the results contained in the dynamic item is plotted by making use of the classes in the OxyPlot library:



The output can be seen in the following figure. Here multiple functions has been plotted, sinus, cosinus, cubed, derivatives and integrals. They are all plotted with 20 datapoint:

:

# **Test Strategy**

Two main tests can be performed: unit test and integration test. Unit tests verify smaller bits of functionality that the logic of function is actually working as suppose.

### Unit Test

During the application development the Scheme procedures have been tested in the DrRacket programming environment to make sure that the procedures worked correctly.

An example of a test could be verifying whether the ‘zip’ procedure actually takes two lists and returns a single list of the corresponding pairs of elements from the input lists.

The test would be inputting “(zip '(1 2 3) '(a b c))” into DrRacket and observing whether the output actually is “((1 . a) (2 . b) (3 . c))”.

The above explained testing strategy for testing the Scheme procedures is unit testing, where each procedure is tested to make sure that it returns the correct result when it is executed.

To prevent unintentionally changing the already written procedures, the tests of these procedures could be gathered in a single file. This file should include the files containing the actual procedures. When adding new procedures or modifying the existing ones the tests could be evaluated to make sure that no damaging changes had been made to the original procedures.

In C# there is done very few unit tests, however most of the code is wrapped in try-catch clauses which prevents the program from failing. It must be checked whether these try-catch clauses will return a null-value. When they do not return a null value, they will forward the error message from IronScheme.

### Integration test

The Scheme and C# program should have their interfaces output tested to make sure that the programs can function together.

In the C# program a mockup of the Scheme procedures could be constructed, so when the C# program sends a string for evaluation to Scheme a mock (instead of the actual Scheme engine) would evaluate whether it actually did receive a string and if so send back a list of coordinates for the C# application to print. In that way, it fulfills integration test.

### Challenges

When writing tests for a WPF-program it’s common to use testing plugins, like NUnit and NSubstitute. These are excellent to simulate data in the program through interfaces. But due to an unknown error we couldn’t get the MVVM-pattern to work with the plugin Oxyplot which forced us to write in code-behind. This eliminated almost all good patterns for interfaces and unit testing, and as a result not many test has been written.

When creating the code-behind file everything is linked to the .xaml-file and back again. So in order to create data bindings the spelling of property names must be double checked, since there’s no Intellisense to correct wrong code.

When having adapted to the MVVM-pattern the structure of the file in the code behind is also different. This requires compromises in the code-behind - it introduces small hacks and big methods which are harder to split into smaller and more intelligent sizes.

When using IronScheme the result returned is not known in advance and the return type is set as a ‘dynamic’, which will, on runtime, get the values and methods the object provides.

This causes a lot of debugging to find the exact match of methods to be used.

# **Conclusion**

The intention of this project is gaining experience with implementing multi-paradigm programming. A graphing application for mathematical functions has been created by applying multiparadigm programming. The application consists of two programs paradigms - a C# program providing a graphical user interface (as imperative programming paradigm) and a Scheme program providing computations based on the user input (as functional programming paradigm).

The biggest difference between imperative (C#) and functional (Scheme) programming is, that functional programming is more focused on what you want done, rather than specifying how you want (imperative) something done. It's probably best understood in contrast to imperative programming, e.g. it is much easier to express mathematical computations in Scheme than other imperative programming languages like C. In this project it also showed that all the mathematical calculations were easier to implement, since the implementing method was free.

Therefore, through this project, real practice has been obtained and the concrete concepts for multi-paradigm programming and the flexibility of implementation has been explored (i.e. applying the paradigms for which cases they were better fit).