Project Proposal

Danny Klein, Austin Osborn

0.1 Introduction

As physics and math students, we have a lot of experience using graphing tools like Mathematica and Desmos. We want to create a language that can graph functions using intuitive input and customizable output. Our language will be able to graph complicated equations in whatever color or style the user desires. Often these complicated equations, like $2x^{(sin(x^2))}$ are incredibly difficult and tedious to graph without the help of a computer, and our language will be able to turn these into easy to read graphs.
This should have it's own programming language because it would be incredibly tedious to create these graphs directly in a language not designed for visual output, like F #, or for graphing complicated equations, like SVG. Examples like Desmos and Mathematica show just how useful this language could be. Our language would make it very easy for users to graph functions, with simple and intuitive programs.

0.2 Design Principles

The primary goal of this language is to provide a method of graphing that is intuitive to use, but still maintains the most important use cases for graphing. When discussing our goals for this project, we brought up several graphing programs that we have used in the past, and features which we appreciated about them as well as features we wished they had with the intent of figuring out a unique space for our language to fall in the landscape of programs we have used. The two primary ones we compared it to were: Desmos and Mathematica. Both of these are substantial programs with complexity of such a degree that it would be near impossible to approximate their functionality in the time we have left this semester, but drawing inspiration from them helped guide our design principles. We seek to make a language which is more intuitive to use than Mathematica (since Mathematica can basically do whatever the user wants as long as the user is able to wade through the dense syntax), but able to do some things that Desmos cannot (since it seems Desmos is on the other end of the spectrum with characteristics fully geared towards intuitiveness, which leaves some customization possibilities at the wayside).

0.3 Examples

1. `Plot sin(2x), dashed, red. x from -10 to 10.`

2. `Plot 2x, dotted, green. Plot x^(2)+1, solid, blue. Plot 3^(sin(x)), dashed, red. x from 0 to 5.`

3. `Plot 1, solid, black. x from 0 to 1.`

0.4 Language Concepts

The user would not need to understand much to write programs in our language. The user would need to be able to write equations (combining forms) in terms of numbers and variables (primitives), and represent their desired output with a linetype (primitive) and a color (primitive). A user who knows how to write equations, understands colors, and can tell the difference between a solid line and a dashed line, would be able to use our language easily. Our language takes primitives, the numbers and variables, and uses other combining forms such as exponents and trig functions to represent complex equations (combining form), and creates a graphical output (combining form) using the equations along with the color and linetype.

0.5 Syntax

```
<Graph>    ::= <Plot>^+ <Domain>.
<Plot>     ::= Plot <Func>, <Linetype>, <Color>.

<Func>     ::= <Val>
             | <Trig>
             | <Op>
             | <Parens>

<Val>      ::= <Num>
             | <Var>
<Num>      ::= <Digit>^+ | -<Digit>^+
<Digit>    ::= 0 | ... | 9
<Var>      ::= a | ... | z

<Trig>     ::= <Sin> | <Cos> | <Tan>
<Sin>      ::= Sin <Func>
<Cos>      ::= Cos <Func>
<Tan>      ::= Tan <Func>

<Op>       ::= <Plus>
             | <Minus>
             | <Times>
             | <Div>
             | <Exp>
<Plus> = (<Func> + <Func>)
<Minus> = (<Func> - <Func>)
<Times> = (<Func> * <Func>)
<Div> = (<Func> / <Func>)
<Exp> = (<Func> ^ <Func>)
<Parens> = (<Func>)

<Linetype> ::= Dashed | Dotted | Solid
<Color>    ::= Blue | Green | Red
             | Yellow | Purple | Orange
             | Black | Gray | Pink
             | RGB(<Num>, <Num>, <Num>)

<Domian>   ::= <Var> from <Num> to <Num>.
```

0.6 Semantics

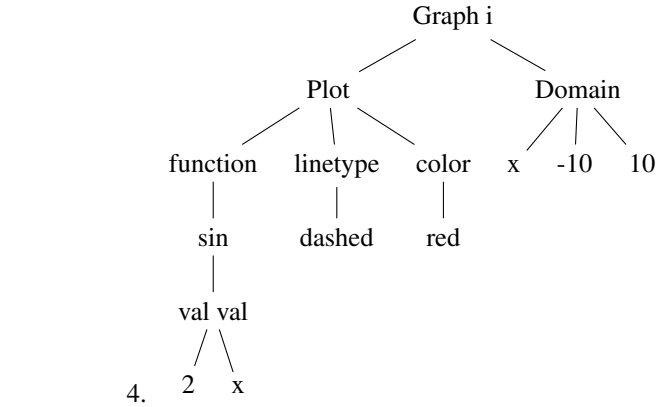
1. In our first version of the graph language, we will use: variable, number, color, and linetype as primitives with the ability to expand into more customization options later on if the scope of the project expands. Var is a variable like x, or y; num is some positive or negative integer; color is a color which is either one of the basic colors which we have manually defined, or can be customized by the user using RGB(R,G,B); linetype can specify how the line is drawn i.e. solid, dashed, or dotted to start.

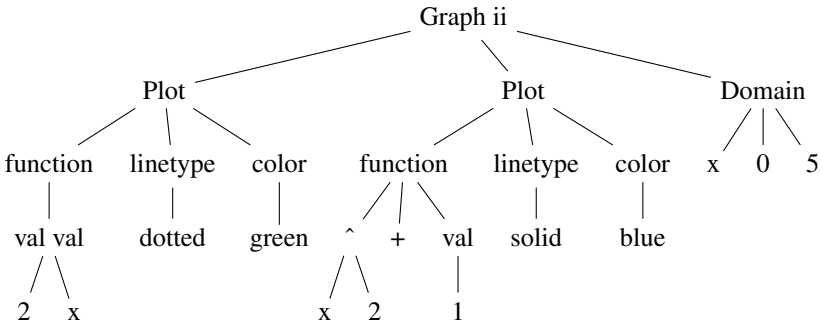
2. The combining forms which we will use to start will be: application (functions like *), plus (+), minus (-), divide (/), exponent (^), all types of functions (sin, cos, tan), plot, domain, and graph (this list will likely grow as the scope of the language grows). The mathematical operations (app, +, -, /, ^) each take the two functions which the operation will be performed on with the operator between them. Some other functions only take in one input

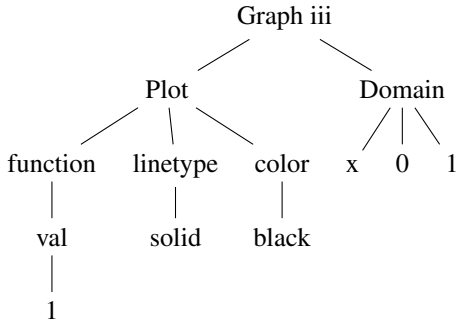
value, currently the only functions like this in our language are the trig functions (sin, cos, tan), but this could change with the scope. A function is a combining form that can combine any number of smaller functions, values, or variables using the operations and functions defined. A plot will combine the combining form of the function with the primitives linetype and color to represent what the plot will look like. The domain will combine the variable we are graphing with a lower and upper bound on the domain. A graph is a combination of at least one plot and one domain. It will draw the stylized functions on the given domain all on one set of axes.

3. Our program is represented as a graph. A graph is a series of plots and a domain. A plot consists of a function, a linetype, and a color. A function is recursive, and can be a val, a trig, or an operation(+ ,- ,* ,/ ,^) on two functions. A val is a number or a variable. A trig is sin, cos, or tan of another function. A linetype, color, num, and var are primitives.

4.







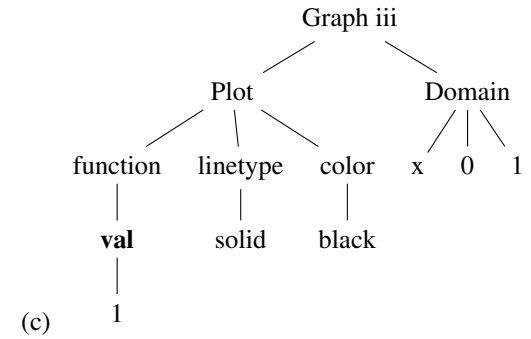
5. (a) Yes!Any input should be able to read, but only inputs which are correct syntactically will give a graphical output.

   (b) Programs in our language will be read through the command line. The user will give the command line input of the graph it wants to create, similar to how we evaluated the derivative calculator in lab 8. The output of the program will be a string that represents an svg program, which would translate to the given graph. This is how our trees will get evaluated:

      i. We read in the program through the command line

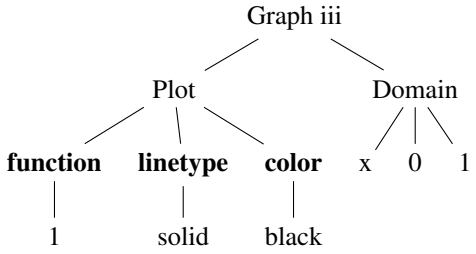
ii. A string that is an SVG program, representing a plot of the given function

iii. Draw another tree

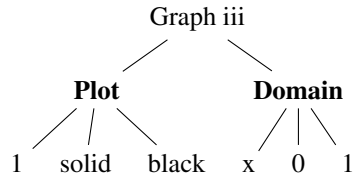


(c)

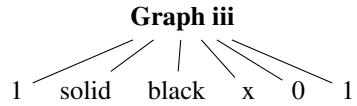
We start with the AST for example iii shown above in part 4.



First step is to evaluate val to be equal to 1 since it is lowest, leftmost node



Next, can evaluate function to be equal to 1, linetype to equal solid, and color to equal black



Next, can evaluate plot as a plot of function 1, with linetype solid, and color black, and can evaluate domain as x varying from 0 to 1

Finally, the graph can be evaluated and outputted with these properties as code for a SVG file