Project 2: gnuplot

Version 4, 12 October 2004

Project handed out on 12 October. Complete Java implementation due on 2 November.

2.1 The task

Implement a utility for plotting graphs of mathematical functions. It will accomplish functionality similar to the `gnuplot` utility's `plot` command in spirit, but much simpler.

2.2 Input

The user can specify mathematical expressions that denote mathematical real functions (i.e., functions in **R** → **R**) to be plotted—details below.

The user must also specify a real interval for the independent variable, given by its lower and upper boundary. The user must also specify the number of interior points to be used for function evaluation for each mathematical expression given (evaluation points are to be spaced uniformly between the lower and the upper boundary of the interval).

2.2.1 Expressions

The input syntax of mathematical expressions is given by the following context-free grammar:

Expr → *IntegerNumeral*
Expr → **e**
Expr → **pi**
Expr → *Variable*
Expr → **let** *Variable* **be** *Expr* **in** *Expr* **end**
Expr → **neg** (*Expr*)
Expr → **add** (*Expr*, *Expr*)
Expr → **sub** (*Expr*, *Expr*)
Expr → **mul** (*Expr*, *Expr*)
Expr → **div** (*Expr*, *Expr*)
Expr → **sin** (*Expr*)
Expr → **cos** (*Expr*)
Expr → **tan** (*Expr*)
Expr → **arctan** (*Expr*)
Expr → **exp** (*Expr*)
Expr → **ln** (*Expr*)
Expr → **pwr** (*Expr*,*Expr*)

where:

IntegerNumeral is a string of characters denoting an integer number (as in JLS2, p.21, but decimal only)

Variable is a string of characters consisting of letters, digits, underscores, and the apostrophe, and beginning with a letter, other than any keyword (keywords are given in boldface above).

White space is allowed everywhere between lexical tokens. To be specific, we define white space as anything for which `Character.isWhiteSpace ()` returns `true`.

Note that a fully parenthesised prefix notation is used for arithmetic, in order to simplify the task of parsing expressions.

2.2.2 Variables

Expressions in our input syntax have both variables and a `let` construct for variable binding. Note that the `let` construct can be nested—it works exactly as in Scheme.

Each occurrence of a variable is either bound by a particular enclosing `let` or it is free. An expression that has no free variable occurrences is said to be closed. In our syntax, non-closed expressions are possible.

The value of an expression is determined by the usual mathematical rules of evaluation, except for the values of variables and `let`-expressions.

The value of a bound occurrence of variable is defined to be the value of the expression that variable was bound to in the binding `let`.

The value of a non-closed expression depends on the interpretation of the values of the free variable occurrences. This interpretation is given by always evaluating an expression *with respect to an environment*; the environment supplies the values for the variables.

The value of a `let`-expression with respect to an environment *e* is the value of its body, evaluated with respect to an environment that adds the `let`-binding to *e*.

2.2.3 Environments

Environments are mappings from variables to floating-point numbers. You will represent an environment as a list of bindings, i.e., variable-value pairs.

However, the input syntax for environments is:

Environment → **empty**
Environment → **bind** (*Variable*, *Expr*, *Environment*)

In other words, an environment is specified as either empty (it maps no variables), or as extending another environment by adding another mapping (binding another variable). In the latter

case, the *Expr* must not have free variables that are not bound in the right-hand-side *Environment*.

Upon reading an environment from the input file, you must transform it into a list of bindings, i.e, variable-value pairs.

2.2.4 More on expression evaluation

If an expression is to be evaluated with respect to an environment, then the environment must bind all variables that occur free in the expression. A closed expression can safely be evaluated with respect to the empty environment.

2.2.5 Input syntax: commands

The syntax of commands is:

Command → **plot** [*Expr*:*Expr*] *PlotItemList*

PlotItemList → **nil**
PlotItemList → **cons** (*PlotItem*, *PlotItemList*)

PlotItem → **item** (*Expr*, *Variable*, *Environment*, *PrecisionSpecification*)

PrecisionSpecification → **with** *IntegerNumeral* **points**

2.2.6 Input syntax: putting it all together

The input file must contain exactly one command.

2.3 Examples

2.3.1

To plot the mathematical functions $x \mapsto e^{-x^2}$ and $t \mapsto A \sin \omega t$ with parameter values $A = \pi$ and $\omega = 0.7$ on the interval $[-\pi, \pi]$, invoke[1]

```
plot [neg(pi):pi] cons (item (exp(neg(pwr(x,2))),
                              x,
                              empty,
                              with 1000 points),
                        cons (item (mul(A,sin(mul(omega,t))),
                              t,
                              bind(A,pi,bind(omega,div(7,10),empty)),
                              with 1000 points),
                        nil))
```

The result should look like the following (Figure 1):

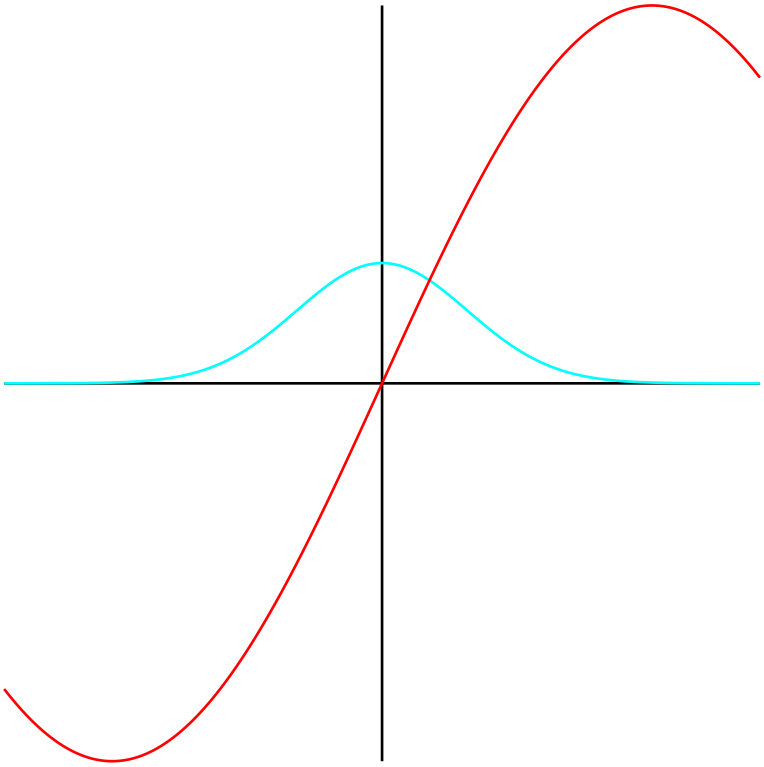


Figure 1: Example output: $x \mapsto e^{-x^2}$ (cyan) and $t \mapsto A \sin \omega t$ where $A = \pi$ and $\omega = 0.7$ (red)

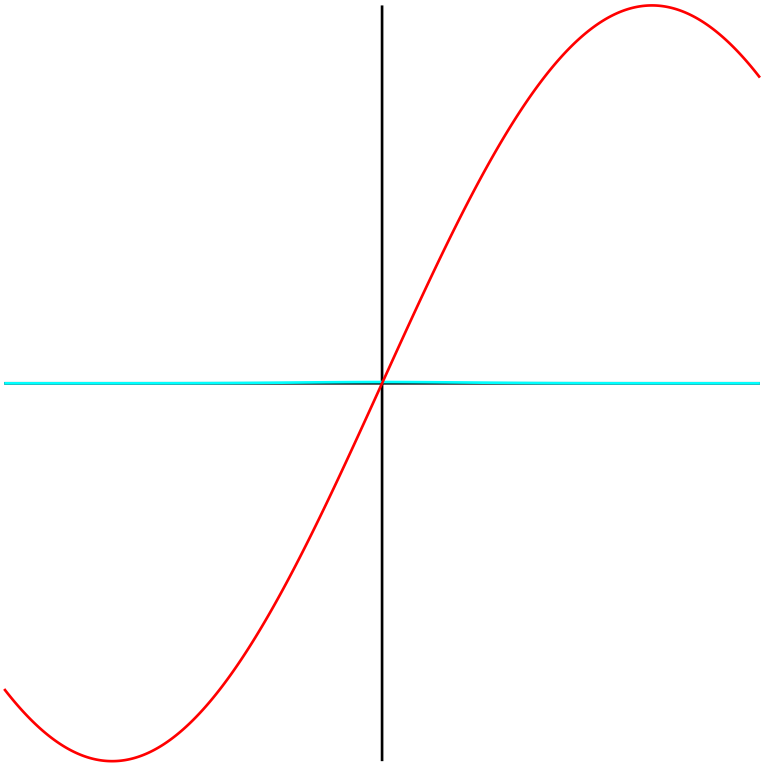
[1] In standard `gnuplot`, this corresponds to: `plot [-pi:pi] exp(-x*x), pi*sin(0.7*x)`.

2.3.2

The input:

```
plot [neg(pi):pi] cons (item (exp(neg(pwr(x,2))),
                              x,
                              empty,
                              with 1000 points),
                        cons (item (mul(A,sin(mul(omega,t))),
                              t,
                              bind(A,300,bind(omega,div(7,10),empty)),
                              with 1000 points),
                        nil))
```

should produce output like the following:

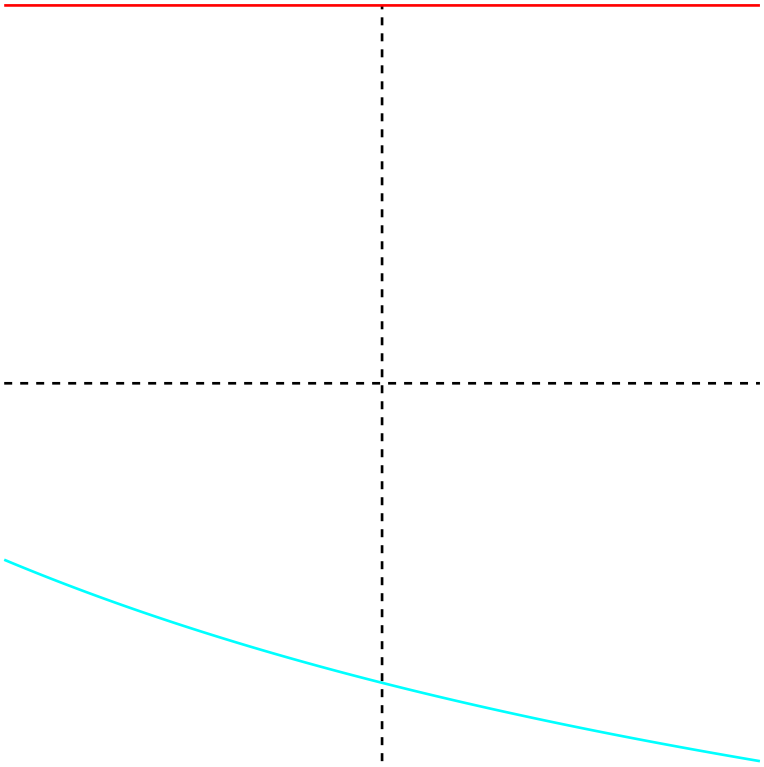


2.3.3

The input:

```
plot [2:pi] cons (item (div (1,x),
                            x,
                            empty,
                            with 999 points),
                  cons (item (1,
                            t,
                            empty,
                            with 1000 points),
                  nil))
```

should produce output like the following:

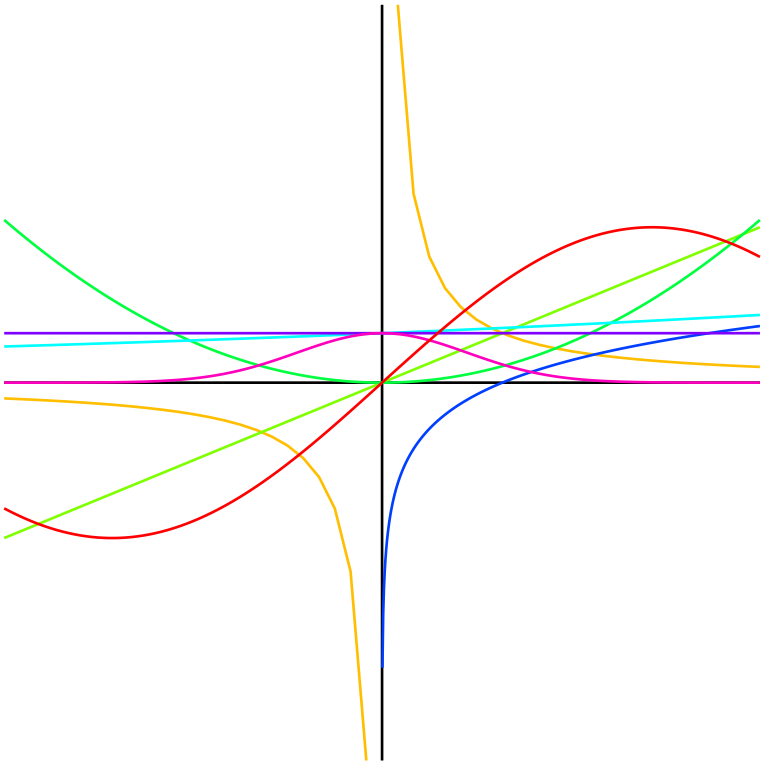


2.3.4

The input:

```
plot [neg(pi):pi]cons (item (div (1, x), x, empty, with 47 points),
                  cons (item (x, x, empty, with 100 points),
                  cons (item (div (pwr (x, 2), 3), x, empty, with 1000 points),
                  cons (item (exp (div (x, 10)), x, empty, with 1000 points),
                  cons (item (ln (x), x, empty, with 1000 points),
                  cons (item (1, x, empty, with 1000 points),
                  cons (item (exp(neg(pwr(x,2))),
                             x,
                             empty,
                             with 1000 points),
                  cons (item (mul(A,sin(mul(omega,t))),
                             t,
                             bind(A,pi,bind(omega,div(7,10),empty)),
                             with 1000 points),
                  nil))))))))
```

should produce output like the following:



2.4 Output

The output must be in the form of a fig file.

The result is a visual display of the plot of the given functions on the given interval, with axes suitably drawn.

The display window should be square. To achieve this, the image should be scaled; in particular, the two dimensions may need to be scaled differently.

The drawing area of the resulting fig program should be *consistent* for functions with arbitrary domains and ranges; to make this precise, we insist that it should be a square 100 mm on the side. The plots should (almost) fully use the drawing area; a 1 mm border is recommended.

Each line graph must be in a different color, but none should be black. The colors must be easily distinguishable from one another (both on the screen and on the printed page).

Determine the region of the plane spanned by the given functions over the given interval. If the x-axis passes through this region, it should be drawn in black. If the x-axis does not pass through this region, draw a dashed black line parallel to the x-axis through the middle of the region.

If the y-axis passes through this region, it should be drawn in black. If the y-axis does not pass through this region, draw a dashed black line parallel to the y-axis through the middle of the region.

The plot file must be a valid fig file. It must be possible to run it through your transfig program, and it must be possible to view the resulting Encapsulated PostScript file with ghostview, include it in LaTeX documents, and print to a PostScript printer. It must also be possible to load and edit the fig file using xfig. (For example, since your version of gnuplot does not place legends or even labels on the plots, you may add those by hand.)

2.5 Structure

The parsing of the input must be done systematically. Follow the pattern suggested in Scott's *Programming Language Paradigms* and define the lexical and syntactic structure of the language and write a scanner and a parser.

You should define a data type for mathematical expressions and environments, which can be distinct from the concrete parse tree data type, and develop an expression evaluator. This can be done as a first step, independent of the rest of the project (i.e., of the drawing tasks). The following expression is recommended as a test case:

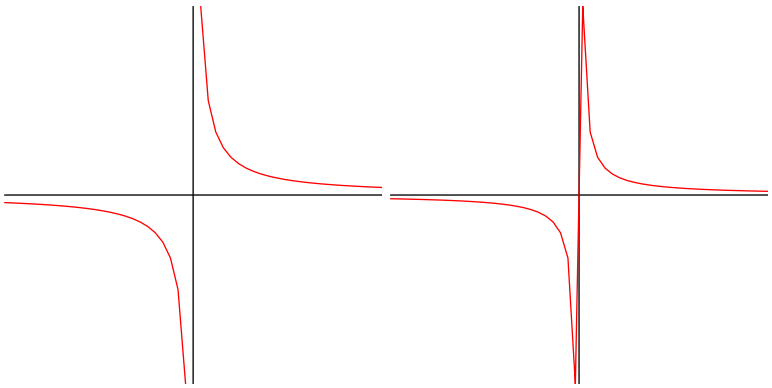
```
let
   x be let y be 3 in mul (y, 2) end
in
   let
      z be sub (x, 1)
   in
      let
         w be pwr (sub (pwr (x, 2), x), 2)
      in
         let
            w be div (pi, arctan (mul (w,
                 div (1, pwr (ln (exp (mul (neg (x),
                 neg (z)))), 2)))))
         in
            let
               g be div (ln (pwr (w, w)), ln (2))
            in
               let
                  u be cos (div (pi, 6))
               in
                  let
                     v be tan (div (pi, pwr (mul (sin (div (pi,
                          mul (div (mul (pwr (u, 2), 2), g),
                          pwr (w, 2)))), 2), 2)))
                  in
                     let
                        p be pwr (e, v)
                     in
                        div (pwr (ln (mul (p, p)), 2), mul (2, x))
                     end
                  end
               end
            end
         end
      end
   end
end
```

2.6 Important details

All floating-point computation must use double-precision floating-point numbers. Your program must handle singularities gracefully and still produce an appropriate plot. For instance, if the input asks for $x \mapsto \frac{1}{x}$ to be plotted over the domain $[-2,2]$ with 199 internal points, it is not acceptable for the program to fail upon attempting to divide 1 by 0 at the 100th internal point.

We suggest two approaches: one is building into the evaluator certain knowledge of the behavior of elementary real functions, such as, e.g., that the function ln is defined only for positive arguments; and the other is relying on the behavior of Java platform mathematics methods (the class `java.lang.Math`) and inspecting their results.

The following two figures illustrate expected output for $x \mapsto \frac{1}{x}$ plotted over the domain $[-1,1]$, with 49 and with 50 internal points, respectively.



The figure on the right, with 50 internal points, contains a line segment that is mathematically meaningless but is acceptable according to this specification.

2.7 Implementation restrictions

There are two restrictions, intended to simulate a real-life situation.

To ensure wide portability, the management has decided that this project must be backwards-compatible with Java version 1.3. Make sure that you do not use any features introduced in later Java versions. This can be accomplished with the `-source` and `-target` options to the `javac` Java compiler. If you have access to a 1.3 JVM, test on it as well as on a 1.4 JVM. For reference, the web site
`http://java.sun.com/j2se/1.4.2/docs/relnotes/features.html`
lists the features that were added between version 1.3 and version 1.4.2 (the current default version on CS machines).

At the same time, the management wants to keep the code upwardly mobile, so various features (especially, various library classes) left over from earlier versions of Java but no longer recommended for use must be avoided. Running the `javac` Java compiler with the `-deprecation`

flag identifies any such features. Thus, when you compile and test your code with version 1.4.2 and the `-deprecation` flag, no warnings must be issued.

There is no need to worry about version 1.5.

2.8 Packaging

Your code must be placed in a package named
`edu.unm.cs.cs351.fall2004.`*yourusername*`.gnuplot`.
The main class of your program must be called `Gnuplot`.

You should use subpackages; for instance a subpackage `syntax` makes sense.

The program accepts one command-line argument, the name of a gnuplot command file. Thus, it is invoked as:
yourfavoritejvm `edu.unm.cs.cs351.fall2004.`*yourusername*`.gnuplot.Gnuplot` *gnuplotfilename*.
The output is placed in *figfilename*, which is obtained by stripping the trailing `.gnuplot` from *gnuplotfilename* and replacing it with `.fig`.

2.9 Optional features

Support an *optional* explicit specification of the *y*-range.

Support an additional kind of plot item for plotting tabulated data from files, with the syntax *PlotItem* → **dataitem** (*Filename*)

Support more palatable syntax: (1) Infix notation for mathematical expressions, with traditional operator symbols, and traditional operator precedence (and parentheses). (2) The plot function list as a comma-separated list instead of a list built using the nil and cons constructors. (3) Environments represented as

$$\{v_1 \to e_1, \ldots, v_n \to e_n\}$$

instead of

$$\texttt{bind}(v_n, e_n, \ldots \texttt{bind}(v_1, e_1, \texttt{empty}) \ldots)$$

(note order!). Supporting this alternative syntax will require you to learn about more powerful parsing techniques than those that suffice for the basic assignment.

Produce labels for the axes, *x* and *y*, or as specified by the user. Produce tick marks on the axes (or the middle lines, as the case may be). Produce numerical markings corresponding to the tick marks on the axes. Produce a figure legend identifying the plot lines. Produce a figure title.

More gracefully handle functions such as $x \mapsto \frac{1}{x}$ around singularities, perhaps by using adaptive selection of evaluation points, to avoid false lines.

Directly output Encapsulated PostScript in addition to fig.

Consult the real `gnuplot` for inspiration on these features, but make sure that they fit nicely in the input syntax.

Any and all optional features that you implement must not be in lieu of the standard features. It is recommended that you submit a package containing an implementation with standard features only, and, separately, a package containing an implementation with optional features.

How to turn in

Same as for Project 1.