WDGAF : Tutorial/Contribution Aid

Installation

We distribute our utility as a python module over PyPi (The Python Package index) and alias an entry point into our module's initialization script: this is our preferred mode of installation wherein you don't have to worry about the dependencies.

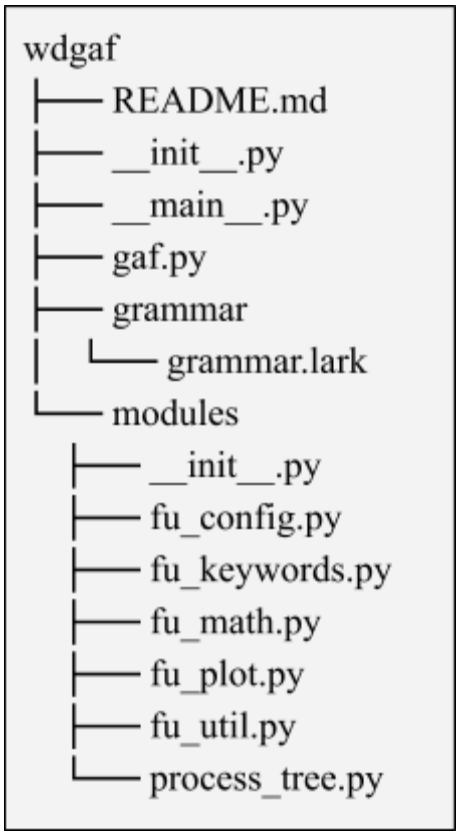
``` bash
*pip install WDGAF*
```

One can build the utility from source if they wish to contribute to or manipulate the utility to suit their own needs. The following section familiarizes the user to the latter.

Module Structure

Cloning the git repository will give you the following.

- *__main__.py* is our module runner and can be accessed by `*python -m wdgaf*` from the parent directory of our root.
- `*gaf.py*` is where we group all the processing (parsing to tree transformation).
- `*grammar/grammar.lark*` is the hackable grammar.
- `*modules*` contains all the submodules that are imported into `gaf.py` before usage.



Installation via pip gives you an executable called `gaf` : get a figure.
If you wish to build from source, use `python -m wdgaf` from the parent directory of our root instead of `gaf` when testing. You will have to deal with dependencies on your own in the latter:
matplotlib and lark-parser are the root ones.

Usage

Proceeding with a pip installation [replace `gaf` with `python -m wdgaf`(path-sensitive) when building from source] :

```bash
gaf <flags> <path/to/source.fu>
```

Flags:

- `-h` : shows help for the utility
- `-t` : prints the parsed Tree from the source according to the Earley algorithm and exits. Note : this will not show any shift/reduce conflicts encountered in the LALR algorithm.
- `-v` : verbose numeric output regarding the figure for debugging purposes.

A figure is then output on a separate window.
Note: an X-window setup is necessary for this step.

Basic Terminology

Keywords: A keyword is a word that is reserved by the language and special meaning to the compiler. WDGAF has the following keywords: ***fig, size, step***

Identifiers: An identifier is a name given to the part of the program such as variables, figures, etc. The following are the rules to define an identifier:

I. An identifier consists of one more alphabets, digits and underscore
II. An identifier cannot start with a digit
III. Keywords cannot be used as identifiers

The following are some valid identifiers:
figure, my_figure, myFigure, MyFigure, figure1, fig2, etc.
The following are some invalid identifiers:
1figure # cannot start with digit
my-figure # ONLY alphabets, digits and underscores
fig # keywords cannot be used as identifiers

Comments: A comment is a programmer-readable explanation or annotation in the source code of a computer program. They are added with the purpose of making the source code easier for humans to understand and are generally ignored by compilers and interpreters.
In WDGAF language a comment start with # and anything following the # is discarded by the compiler e.g.
This is a comment and is ignored by compiler/interpreter
y := x^2 # define y in terms of x as $y = x^2$
x <- (-PI, PI) # x takes values in range $[-\pi, \pi]$

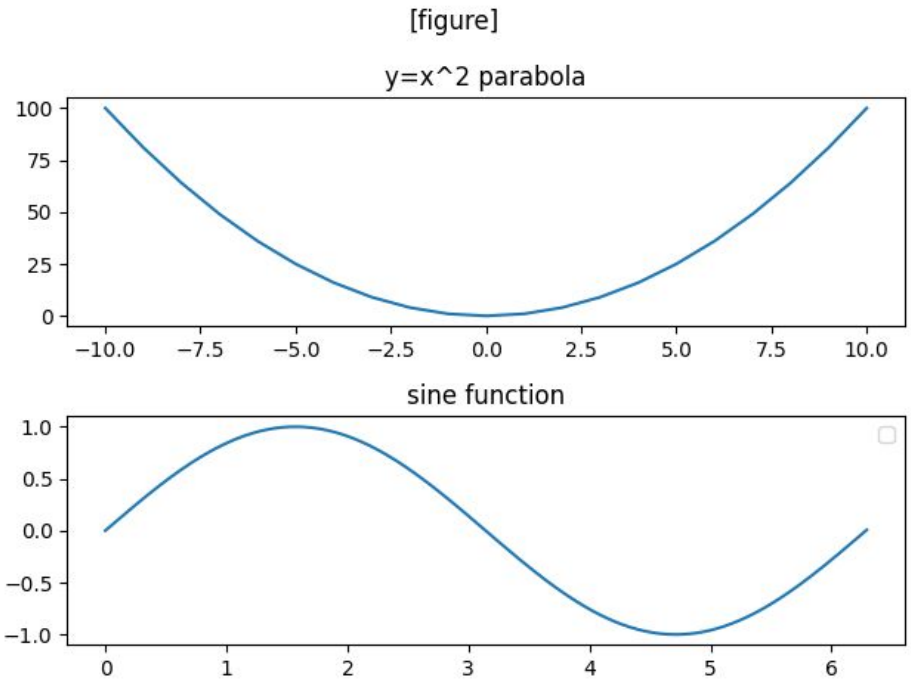
String literals: A string literal can be written by enclosing the text within single quotes (') or double quotes (") e.g. 'hello', "hello", "welcome to 'WDGAF'", etc.
Number literal: A number can be written in integer-notation, decimal-notation or scientific-notation e.g. +10, -20, 1e1, 1.3, 2.5e2, etc.
Color literal: A color literal is used to denote the colors. A color can be written using two formats i.e.

- name of the color e.g. 'red', 'blue', etc.
- hexadecimal color code e.g. '#ffffff', '#00ffee', etc.

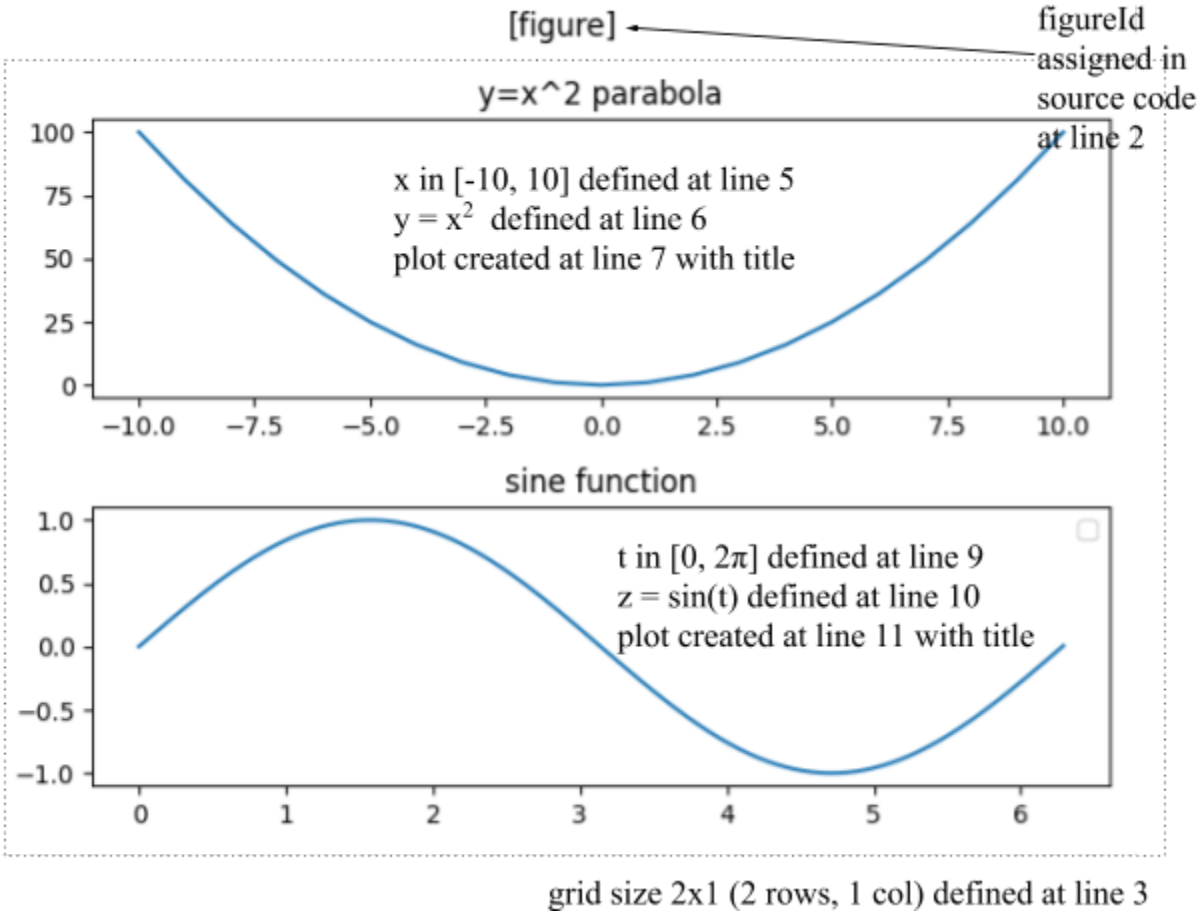
First program in WDGAF

```
1   # create a new figure
2   fig figure{
3           size <- (2, 1) # set the size to be 2x1
4
5           x <- (-10, 10) # x = [-10, -9, ..., 9, 10]
6           y := x^2        # y = x square
7           plot(x, y, {title='y=x^2 parabola'}) # plot x, y
8
9           t <- (0, 2*PI)          # t takes on interval [0, 2π]
10          z := sin(t) # z = sin(t)
11          plot(t, z, {title='sine function'}) # plot t, z
12  }
```

Refer installation and usage instructions provided initially to follow along.
A successful execution will result into following figure:



A closer look at the output shows points the following:



The program is described in the figure itself.

Now we will look at the entire program construction steps by steps considering different types of statements and features available in the language.

The size statement

The size statement configures the size of the figure i.e. number of rows and columns in the figure. By default the size is 1x1 i.e. 1 row and 1 column.

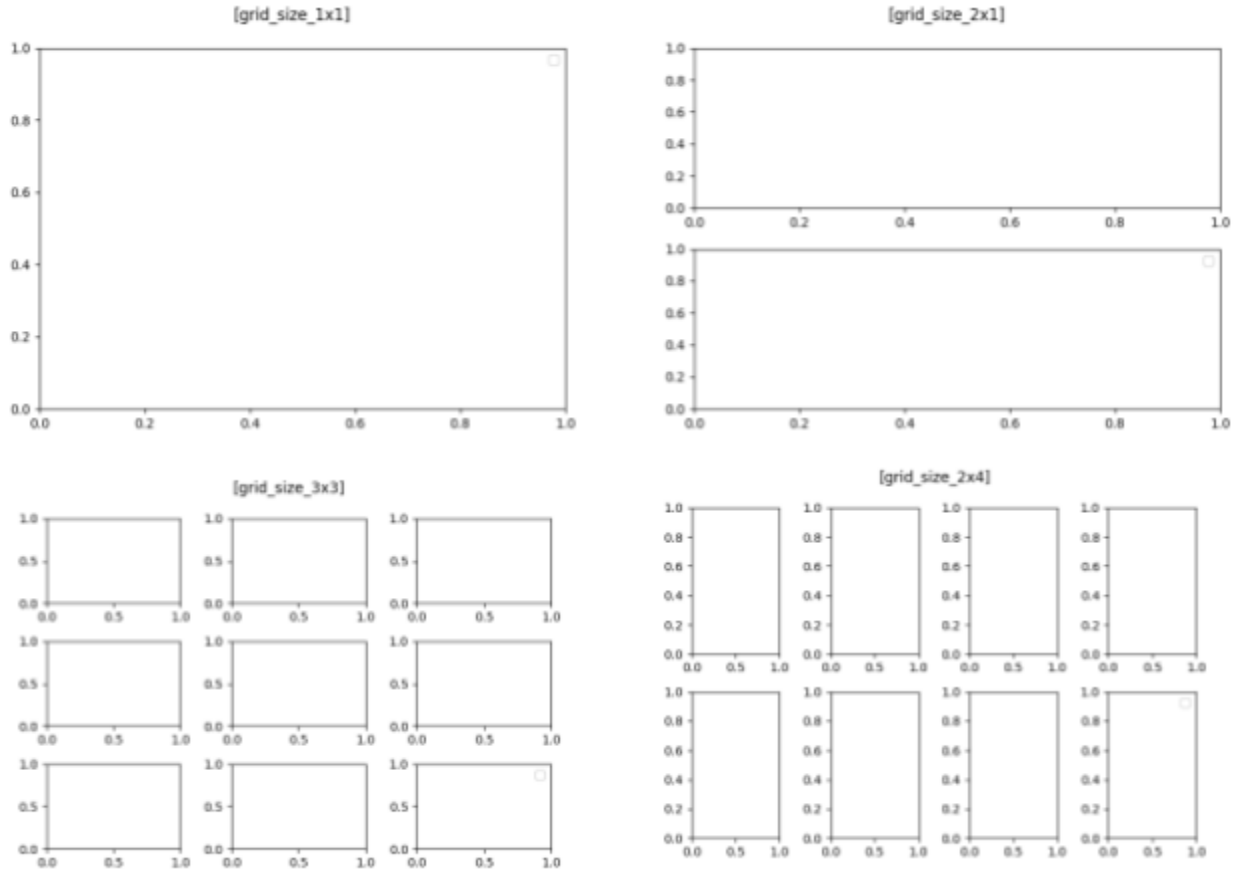
The size statement has the following syntax:

size <- rows [, cols] [, options] where

rows must be a positive integer specifying the number of rows;

cols must be a positive integer specifying the number of columns which is optional and has a default value of 1

options specifies the configurations of the plots created as we will discover soon.



The 1st figure shows the default size configuration i.e. 1 row and 1 column.

The 2nd figure shows the configuration of number of rows which can be achieved using the following command:

size <- 2 # set the number rows to 2 and cols to default (1)

The 3rd figure shows the configuration of number of rows and columns which can be achieved using the following command:

size <- 3, 3 # set the number rows to 3 and cols to 3

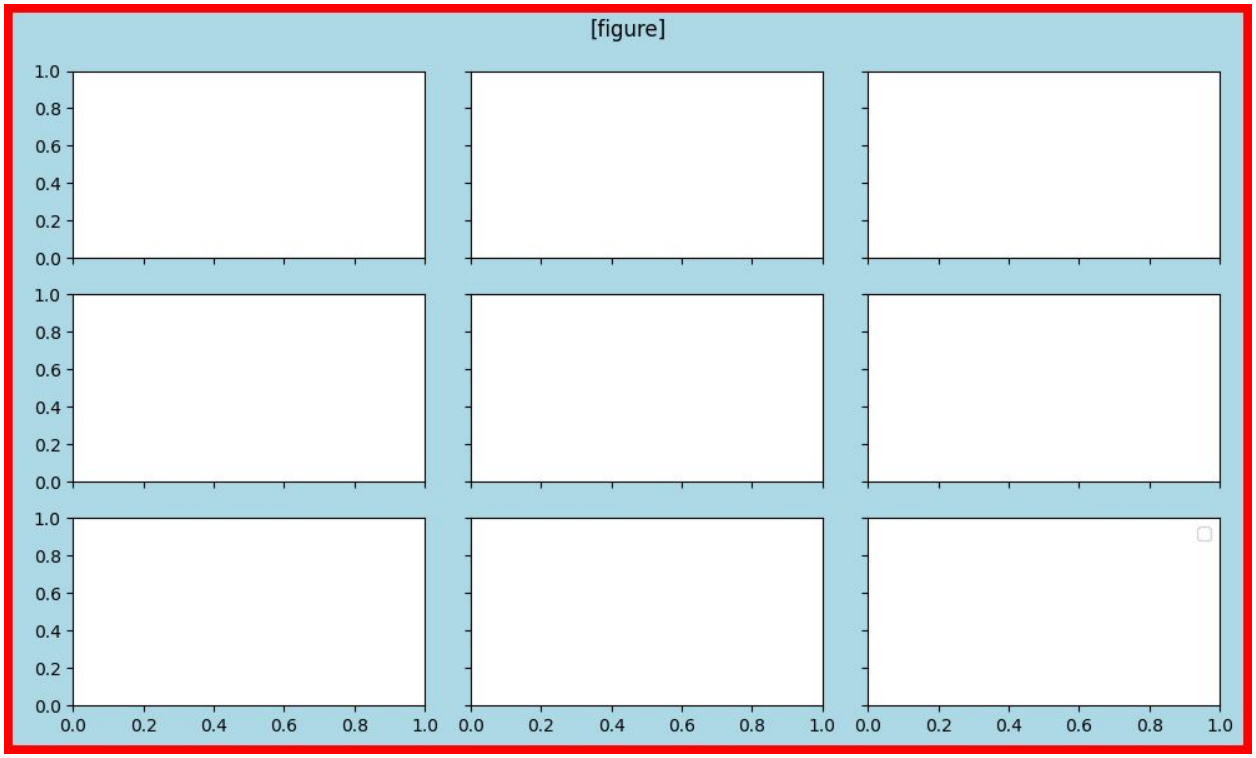
For sake of readability you can optionally wrap the pair within parentheses as:
size <- (3, 3) # set the number rows to 3 and cols to 3
The 4th figure shows the configuration of number of rows and columns which can be achieved using the following command:
size <- 2, 4 # set the number rows to 2 and cols to 4
For sake of readability you can optionally wrap the pair within parentheses as:
size <- (2, 4) # set the number rows to 2 and cols to 4

The size statement accepts the optional configuration options which can be used to configure the figure and the grid cells. The following options are supported:

Option	***Allowed values***	***Effect***
sharex	'true' or 'false'	whether or not the x-axis sharing is allowed
sharey	'true' or 'false'	whether or not the y-axis sharing is allowed
width	positive number	width of figure in *inches*
height	positive number	height of figure in *inches*
facecolor	color string	facecolor of the frame
edgecolor	color string	edgecolor of the frame
linewidth	positive number	linewidth of the frame

The following size statement shows the effects of the options in the next diagram:

```
# This program shows the configuration using size statement
fig figure{
        size <- (3 , 3) { # 3x3 grid
                sharex='true', # allow sharing x-axis
                sharey='true',  # allow sharing y-axis
                width=10, # 10 inches in width
                height=6, # 6 inches in height
                facecolor='lightblue', # facecolor is lightblue
                edgecolor='red', # edgecolor is red
                linewidth=10 # edgewidth 10
        }
}
```



Configuration of the figure using the attributes with size-statement. Try to play around the attributes and observe the generated results.

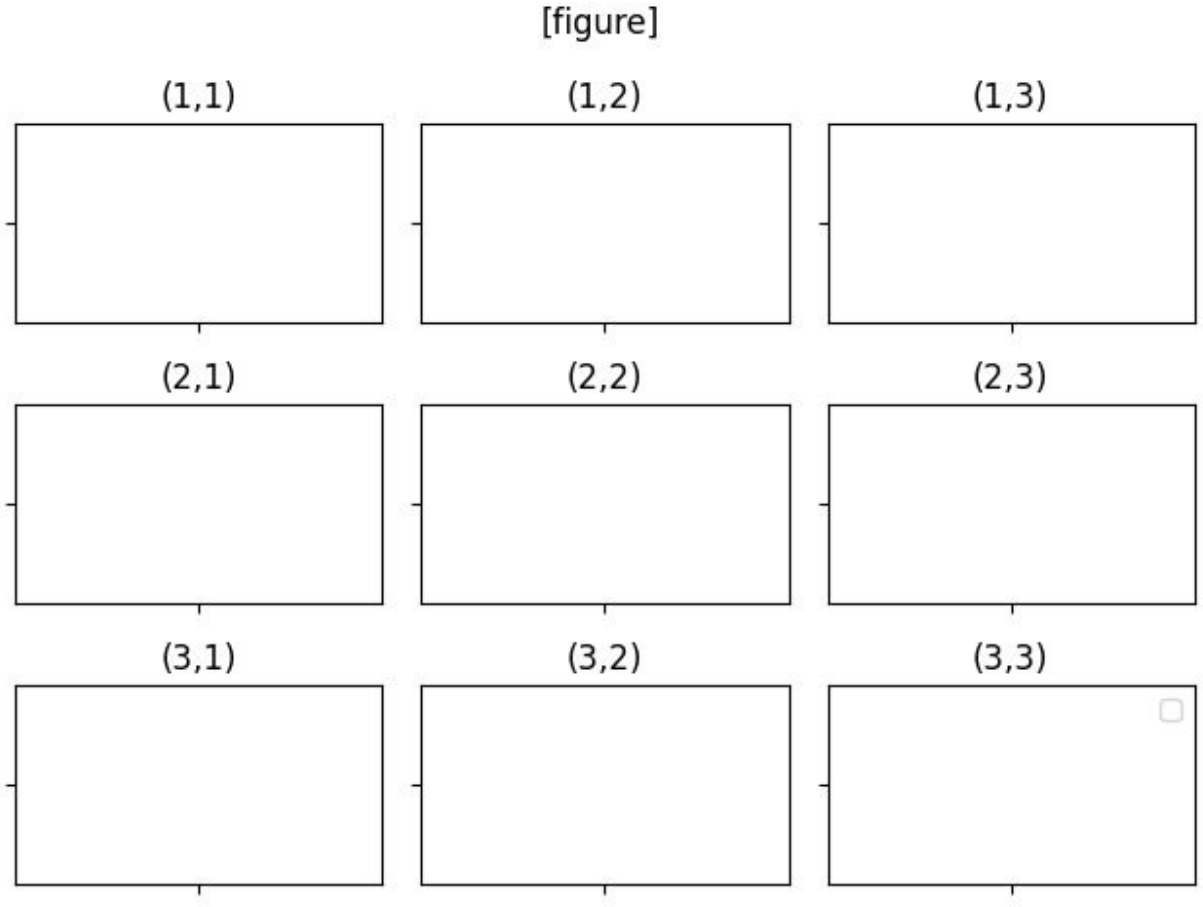
Grid Indexing

One you have defined the size of the figure i.e. number of rows and number of columns you will then index them. For example you might want 1st row and 1st column should contain plot A, 2nd row and 3rd column should contain plot B and so on. WDGAF supports 1 based indexing as we usually do for matrices.

When you create a MxN grid then you will have the following indexing:

(1, 1)	(1, 2)	(1, 3)	...	(1, N)
. . .				
(M, 1)	(M, 2)	(M, 3)	...	(M, N)

The following figure shows the indexing:



Expressions

In this section we will discuss the expressions which form the base of our programming language. As we will discover later WDGAF makes a high use of expressions and therefore we wish to cover them here before starting any other topic. In mathematics, *an expression or mathematical expression is a finite combination of symbols that is well-formed according to rules that depend on the context* e.g. $x + y$, x^2, $1 + 2$, etc.

In short an expression is a combination of variables, constants and operators.

The following table shows the list of supported operators with their priorities:

The operators are shown with decreasing priority.

Operators with the same precedence have associativity from left to right.

Operator
( ) parenthesization
- (Unary negation)
^ (Exponentiation)
/ (True division) // (Floor division) * (Multiplication) %(Modulus or Remainder)
+ (Addition) and - (Subtraction)

When and how to use operator precedence and associativity table?

The precedence and associativity table is used to decide the order of evaluation of operators. Consider the following expression: 2 * 3 ^ 2

Now how do you decide the evaluation result? Is it 6^2 or 2*9

This ambiguity is resolved using the precedence. As we see that exponentiation operator has higher precedence than the multiplication and therefore it should be evaluated as 2 * (3 ^ 2) = 2 * 9 = 18 and not (2 * 3)^2 which is 36

Furthermore if you have more than one operator with same precedence then you will be using the associativity which is left to right which means you will traverse the expression from left to right and whichever operator comes 1st is evaluated provided you are considering only the operator of same priority.

e.g. the expression 2 * 5 // 3 is evaluated as (2 * 5) // 3 and not as 2 * (5 // 3)

The first one evaluates to 3 whereas the second one evaluates to 2

There are often situations where you wish to change these rules e.g. e ^ -1 + x will by default evaluate to (e ^ -(1)) + x i.e. $e^{-1} + x$

However you can use parentheses to change this order as shown below:
e ^ - (1 + x) for $e^{-(1 + x)}$
or for more readability e ^ (-(1 + x)) for $e^{-(1 + x)}$

Constants

Mathematical constants e and Π are encoded as ***EXP*** and ***PI*** in the language and you can use these constants to write expressions
e.g. sin(x + n * PI), EXP ^ sin(x), etc.
Furthermore there are special constants such as ***inf*** (infinity), ***nan*** (NotANumber) which you cannot use in your source code but can these values may arise because of improper arithmetic operations such as division by zero for example.

Mathematical functions

It's often the case that you may wish to write your expression using standard mathematical functions such as trigonometric functions, logarithmic functions, etc.
The following table shows the list of supported mathematical functions:

Type	***Category***	***Functions***
unary	Trigonometric	sin(x), cos(x), tan(x), arcsin(x), arccos(x), arctan(x), sinh(x), cosh(x), tanh(x), arcsinh(x), arccosh(x), arctanh(x) degrees(x), radians(x), deg2rad(x), rad2deg(x)
	Exp and logs	exp(x) [e^x], exp2(x) [2^x], log(x) [$\log_e x$], log2(x) [$\log_2 x$], log10(x) [$\log_{10} x$], expm1(x) [$e^x - 1$], log1p(x) [$\log_e(1+ x)$]
	Others	floor(x), ceil(x), trunc(x), round(x), negative(x), absolute(x), square(x), sqrt(x), reciprocal(x)
Binary	Bitwise	bitwise_and(x, y), bitwise_or(x, y), bitwise_xor(x, y), left_shift(x, y), right_shift(x, y)
	Others	add(x, y), subtract(x, y), multiply(x, y), divide(x, y), floor_divide(x, y), power(x, y), mod(x, y) logaddexp(x, y), logaddexp2(x, y) gcd(x, y), lcm(x, y), maximum(x, y), minimum(x, y) arctan2(x, y), hypot(x, y)

Range statement

The range statement is used to define the independent variables. There are multiple ways to define independent variables as follows:

[1] List of values:

You can define an independent variable using a list of discrete values where the values can either be number or strings but not both.

To define an independent variable using list we have the following syntax:

var_id <- list of values where *var_id* is any valid identifier

```
x <- [1, 2.3, 4.5, 7, 9.8] # list of numbers
y <- ['i', 'ii', 'iii', 'iv', 'v'] # list of strings
z <- [] # empty list
w <- [1, 'ii', 3, 'iv'] # this is invalid - types cannot be mixed
```

[2] Intervals:

It's often the case that we generate plots when variables are defined within some intervals for example plot of sin(x) in $[-\Pi, +\Pi]$. The intervals can be defined in multiple ways and proper care must be taken when defining the intervals.

The following syntax is used to define intervals:

var_id <- (start_expr, end_expr) [step step_expr] where

var_id is any valid identifier

start_expr is an expression that evaluates to a number

end_expr is an expression that evaluates to a number

step_expr is an expression that evaluates to number and the step is optional

There are two cases for defining the intervals:

[2.1] When start_expr, end_expr and step_expr all can be treated as integers:

In this case the values generated are integral values and based on the *start_expr* and *end_expr* the direction is decided. Consider the following examples:

```
x <- (-5, 5) # [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
x <- (5, -5) # [5, 4, 3, 2, 1, 0, -1, -2, -3, -4, -5]
x <- (5, 5)  # [5]
```

By default the two adjacent numbers differ by 1 unit. If you wish to change this behavior then you can specify the steps explicitly as follows:

```
x <- (-5, 5) step 2   # [-5, -3, -1, 1, 3, 5]
x <- (5, -5) step -3 # [5, 2, -1, -4]
```

Special care must be taken when specifying the steps. For example the following statements throws errors:

x <- (-5, 5) step -1 # cannot reach from -5 to +5 in steps of -1

x <- (5, -5) step 1 # cannot reach from 5, -5 in steps of 1

[2.2] When at least one of the start_expr, end_expr or step_expr cannot be treated as integer:

In this case the values are generated with a difference of 0.025 which can be changed using the step parameter.

x <- (-0.5, 0.5) # generates values from -0.5 upto 0.5 with steps of 0.025

x <- (0.5, -0.5) # generates values from 0.5 upto -0.5 with steps of -0.025

x <- (-1, 1) step 0.01 # generates values from -1.0 upto 1.0 with steps of +0.1

Once again special care must be taken when specifying the steps. The following statements throws error:

x <- (-1, 1) step -0.1 # cannot go from -1.0 to +1.0 in steps of -0.1

x <- (1, -1) step +0.1 # cannot go from 1.0 to -1.0 in steps of +0.1

As already mentioned the *start_expr*, *end_expr* and *step_expr* are technically expressions and so you can write them as expressions provided you do not involve any variable in these expressions.

The following range statements are valid:

x <- (-PI/2, PI/2)

x <- (EXP^-1, EXP^1)

The following range statements are invalid:

x <- (-y, y) # defines independent variables and hence cannot contain any variable

Functional expressions

Functional expressions are used to define relationships. The following syntax is used to define the dependent variables:

var_id := functional_expression where

var_id is any valid identifier

functional_expression is any valid expression in terms of pre-defined variables.

e.g. y := x ^ 2 # defines the relationship $y = x^2$

You can write from as simple to as complicated functional expressions as you wish. The following shows some functional expressions:

```
y := x ^ 2
y := (x - 1)^2
y := 2 * x + 3
y := sin(x)
y := sin(n * PI) / PI
```

It is not mandatory for functional expressions to contain variables. They can be purely independent or you can also specify a list of items as shown below:

```
x := [1, 2, 3, 4]
y := [2, 4, 6, 8]
y := []
y := sin(10)
```

To know the reasoning behind this concept checkout the advanced section of this tutoria which explains how the functional expressions are evaluated.

Plot statement

To generate plots we use the *plot* function defined by the language. The plot function has the following syntax:

plot(x, y [, options]) where

x is the variable which occupy the x-axis

y is the variable which occupy the y-axis

options are the optional configurations for the plot

e.g. plot(x, y) plots the values of y against values of x

The following configuration options are supported for the plot statement:

Attribute	***Allowed values***	***Attribute***	***Allowed values***
row	[1, N] where N is the number of rows in the figure	xminorticks	['true', 'false']
col	[1, M] where M is the number of columns in the figure	yminorticks	['true', 'false']
label	string-literal	xticklabel	['true', 'false']
title	string-literal	yticklabel	['true', 'false']
color	color-literal	gridlines	['true', 'false']
xlabel	string-literal	facecolor	color-literal
ylabel	string-literal	marker	[".",",","o","v","^","<",">","1","2","3","4","8","s","p","P","*","h","H","+","x","X","D","d","|","_"]
xscale	['linear', 'log', 'symlog', 'logit']	linestyle	['-', '--', '-.', ':']
yscale	['linear', 'log', 'symlog', 'logit']	linewidth	non-negative number
xmargin	(0.5, ∞]	alpha	[0, 1]

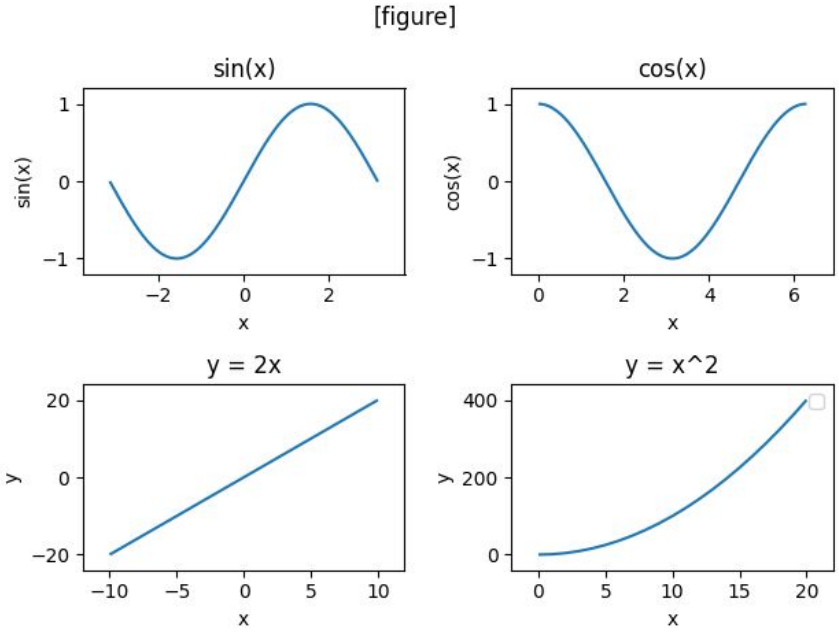
ymargin	(0.5, ∞]	dashcapstyle	['butt', 'round', 'projecting']
xautoscale	['true', 'false']	dashjoinstyle	['miter', 'round', 'bevel']
yautoscale	['true', 'false']	solidcapstyle	['butt', 'round', 'projecting']
xrotation	number	solidjoinstyle	['miter', 'round', 'bevel']
yrotation	number	drawstyle	['default', 'steps', 'steps-pre', 'steps-mid', 'steps-post']
level	['single', multiple']	fillstyle	['full', 'left', 'right', 'bottom', 'top', 'none']
xmajorticks	['true', 'false']	markeredgecolor	color-literal
ymajorticks	['true', 'false']	markeredgewidth	non-negative number
markerfacecolor	color-literal	markersize	non-negative number

Examples

The following program shows the basic working:

```
1   # This program shows the basic working of the language
2   fig figure{
3
4           # set the figure size
5           size <- (2, 2)
6
7           x <- (-PI, PI)
8           y := sin(x)
9           plot(x, y, {title='sin(x)', xlabel='x', ylabel='sin(x)'})
10
11          x <- (0, 2*PI)
12          y := cos(x)
13          plot(x, y, {title='cos(x)', xlabel='x', ylabel='cos(x)'})
14
15          x <- (-10, 10)
16          y := 2*x
17          plot(x, y, {title='y = 2x', xlabel='x', ylabel='y'})
18
19          x <- (0, 20)
20          y := x^2
21          plot(x, y, {title='y = x^2', xlabel='x', ylabel='y'})
22  }
```

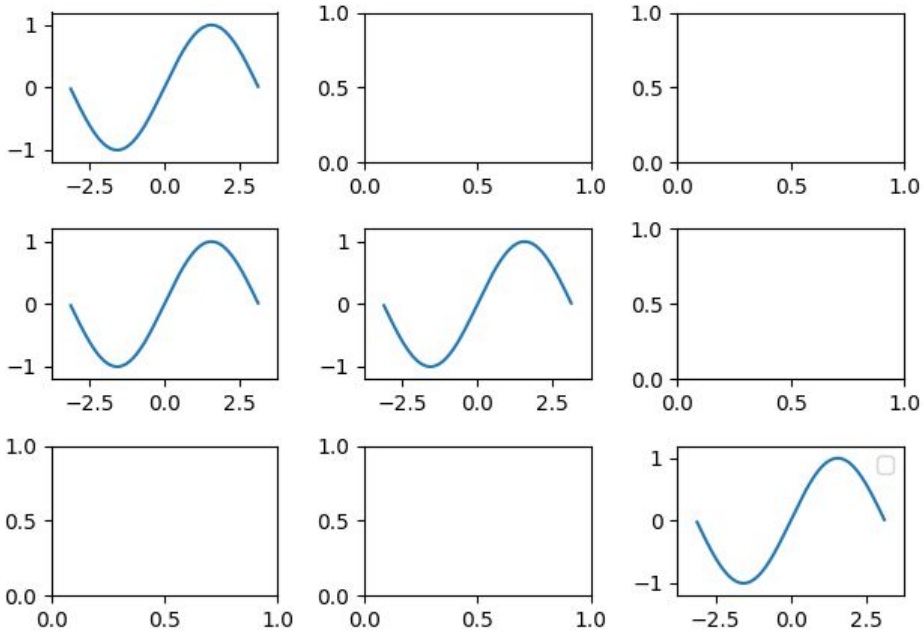
The above program generates the following output:



The following program shows the positioning of plots among cells:

```
1   # This program shows the positioning of plots
2   fig figure{
3
4           # fix the size of the figure
5           size <- (3, 3)
6
7           # define variables
8           x <- (-PI, PI)
9           y := sin(x)
10
11          # by default occupy first
12          # non-empty cell i.e. 1,1
13          plot(x, y)
14
15          # fix the col to be 1 then occupy
16          # 1st non-empty cell in this col
17          plot(x, y, {col=1})
18
19          # fix the row to be 2 then occupy
20          # 1st non-empty cell in this row
21          # now occupy 1st non empty cell
22          plot(x, y, {row = 2})
23
24
25          # fix both the row and the col
26          plot(x, y, {row=3, col=3})
27  }
```

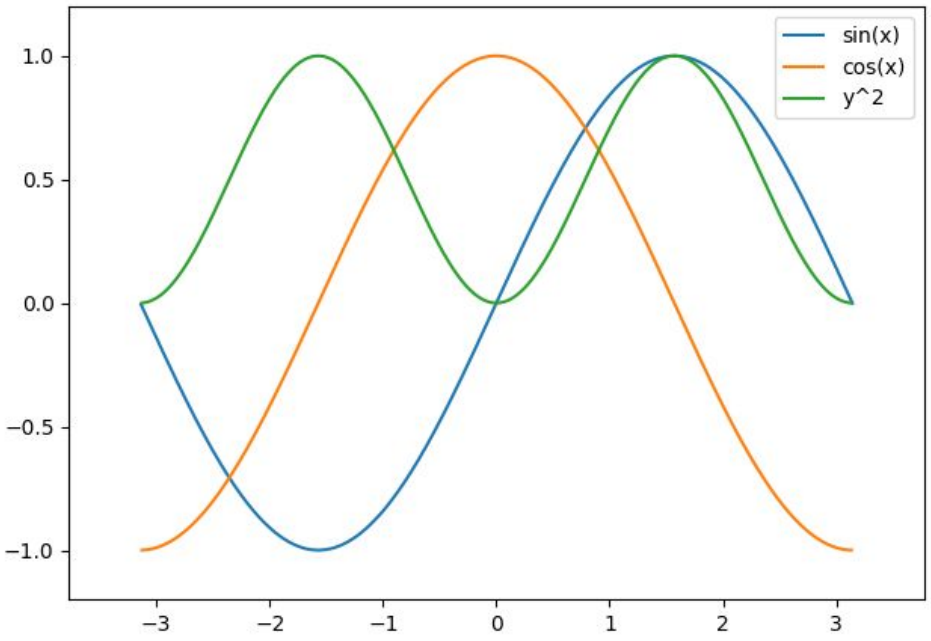
[figure]



The following program shows the multiple plots on the same figure:

```
1   # This programs shows multiple plots on same cell
2   fig figure{
3           x <- (-PI, PI)
4           y := sin(x)
5           z := cos(x)
6           w := y^2
7           plot(x, y, {label='sin(x)'})
8           plot(x, z, {label='cos(x)'})
9           plot(x, w, {label='y^2'})
10  }
```

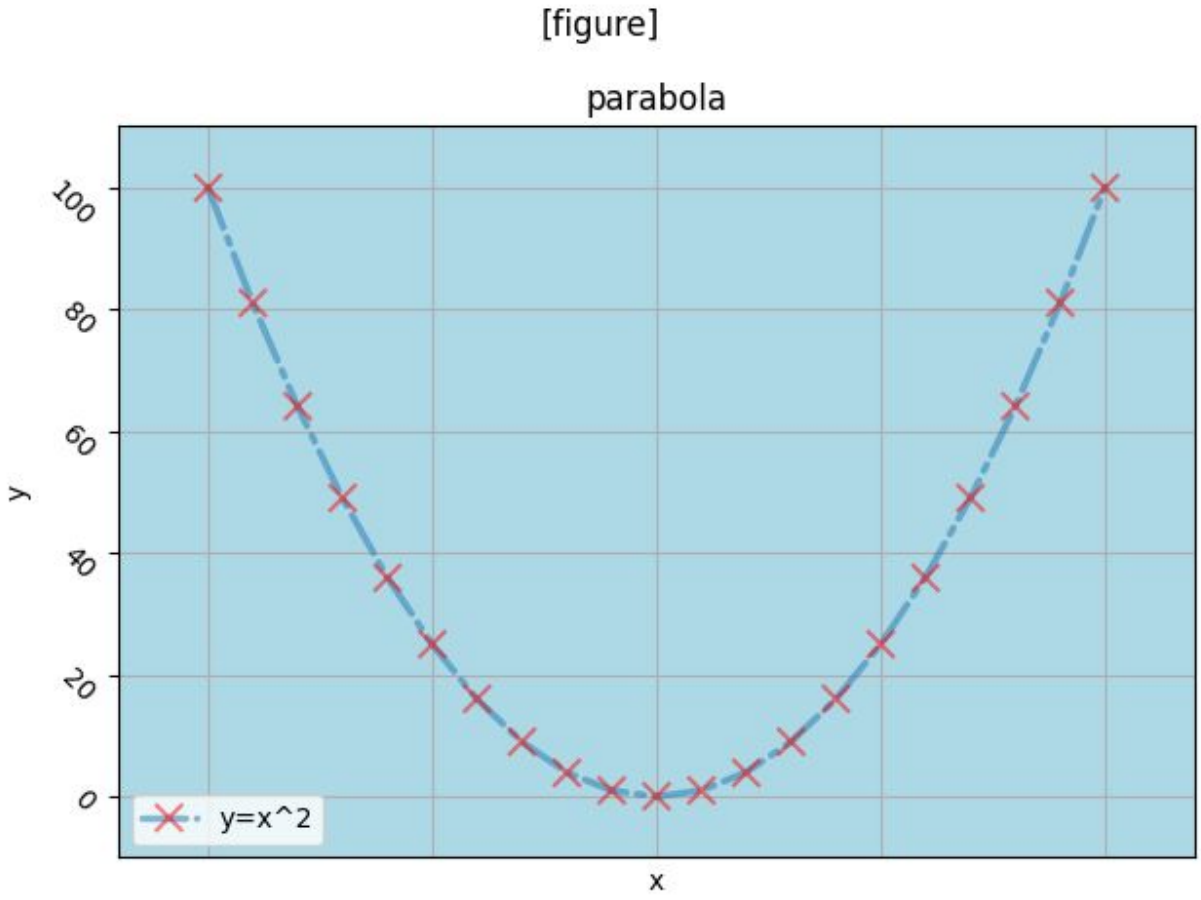
[figure]



The following programs shows the configuration of plot:

```
1  # This programs shows the configuration of plots
2  fig figure{
3          x <- (-10, 10)
4          y := x^2
5          plot(x, y, {
6                  title='parabola',
7                  label='y=x^2',
8                  xlabel='x',
9                  ylabel='y',
10                 xscale='linear',
11                 yscale='linear',
12                 xmargin=0.1,
13                 ymargin=0.1,
14                 xautoscale='true',
15                 yautoscale='true',
16                 xrotation=45,
17                 yrotation=-45,
18                 xmajorticks='false',
19                 xminorticks='true',
20                 xticklabel='false',
21                 ymajorticks='true',
22                 yminorticks='false',
23                 yticklabel='true',
24                 gridlines='true',
25                 facecolor='lightblue',
26                 linestyle='-.',
27                 linewidth=2.5,
28                 fillstyle='full',
29                 alpha=0.5,
30                 dashcapstyle='round',
31                 dashjoinstyle='bevel',
32                 solidcapstyle='round',
33                 solidjoinstyle='bevel',
34                 drawstyle='default',
35                 marker='x',
36                 markeredgecolor='red',
37                 markerfacecolor='green',
38                 markeredgewidth=1.5,
39                 markersize=10.0
40         })
41 }
```

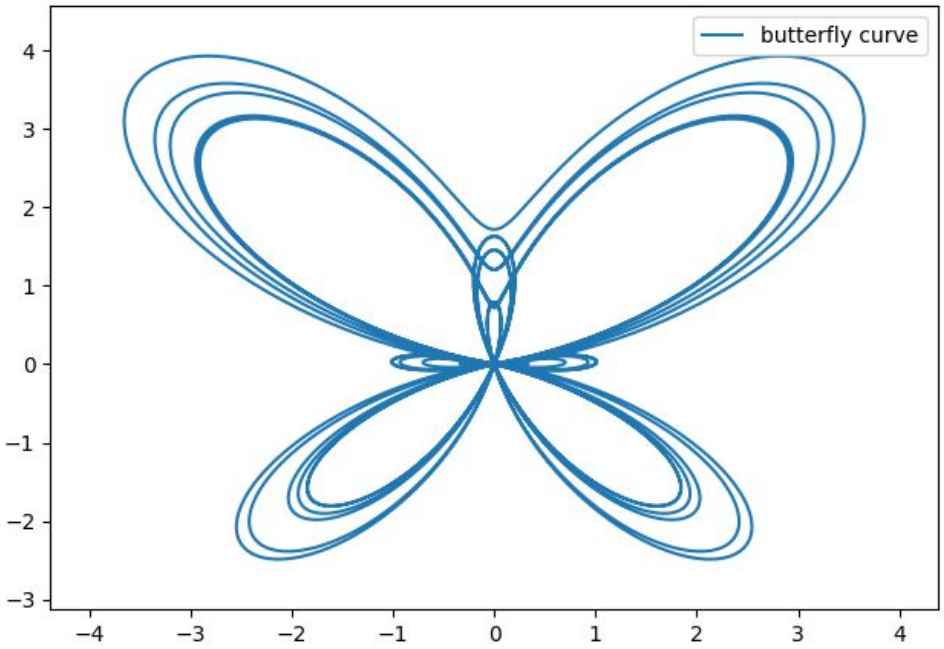
The above program generates the following output:



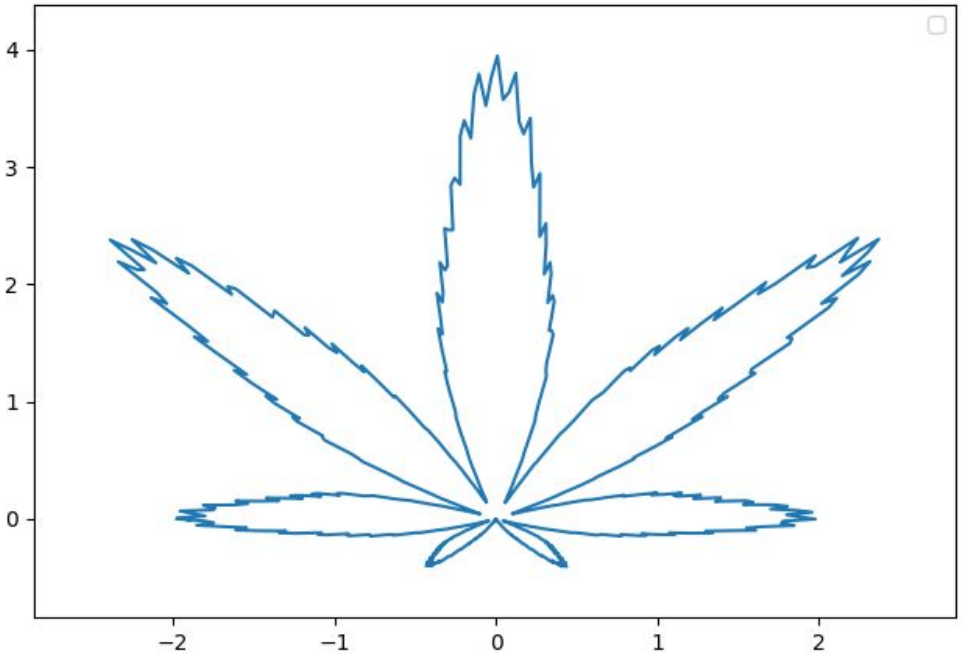
The following programs shows how multiple figures can be generated using a single program:

```
1   # you can write from as simple to as complicated functional expressions
2   fig figure1{
3           t <- (0, 12 * PI)
4           r := EXP^sin(t) - 2 * cos(4*t) + (sin((2*t - PI)/24)) ^ 5
5           x := r * cos(t)
6           y := r * sin(t)
7           plot(x,y, {label='butterfly curve'})
8   }
9
10  fig figure2{
11          t <- (-PI, PI)
12          r := (1 + 0.9*cos(8*t)) * (1 + 0.1*cos(24*t))* (0.9 + 0.05*cos(200*t)) * (1 + sin(t))
13          x := r * cos(t)
14          y := r * sin(t)
15          plot(x,y)
16  }
```

[figure1]



[figure2]



Contour

In addition to normal plots we also have support for contours. Plotting a contour is a two-step process as follows:

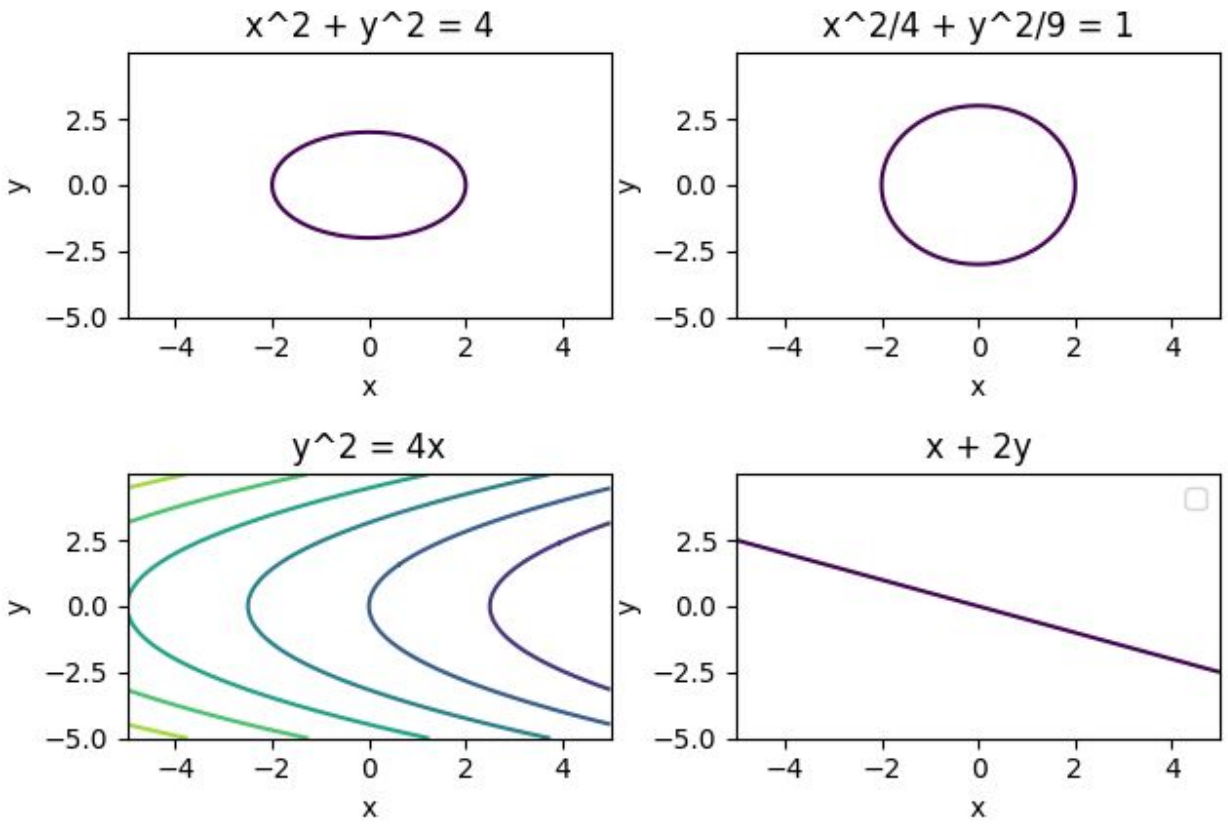
1. First we need to create contour variables using which we can define the equation of the contour. This is done as follows:
   x, y <- contour_grid(start_x, end_x, start_y, end_y) where
   x, y are the identifiers used to create contour variables
   start_x, end_x, start_y, end_y are expressions indicating the starting and ending point for the contour (this basically defines the rectangle where contour is to be plotted)
2. Secondly we define the contour expression and plot it using the contour statement as follows:
   contour(x, y, z, options) where
   x, y are the contour variables created using contour_grid statement
   z is the variable which contains the contour expression
   and options are optional options for contour configurations

The following program shows the plotting of contour:

```
1   # This programs demonstrates the use of contour
2
3   fig figure1{
4           # define the size of the grid
5           size <- (2, 2)
6
7           # define a contour grid and accept the variables
8           x, y <- contour_grid(-5.0, 5.0, -5.0, 5.0)
9           # define the contour using accepted variables
10          # equation of circle x^2 + y^2 = 4
11          contour(x, y, x^2 + y^2- 4,
12                  {title='x^2 + y^2 = 4', xlabel='x', ylabel='y'})
13
14          # equation of ellipse
15          # x^2/4 + y^2/9 = 1
16          contour(x, y, 9*x^2 + 4*y^2 - 36,
17                  {level='single', title='x^2/4 + y^2/9 = 1',
18                  xlabel='x', ylabel='y'})
```

```
19
20          # equation of parabola y^2 = 4x
21          # all set to true generates multiple levels
22          contour(x, y, y^2- 4*x,
23                  {level='multiple', title='y^2 = 4x',
24                  xlabel='x', ylabel='y'})
25
26          # you can also create a contour variable
27          # using the contour variables as shown below
28          z := x + 2*y
29          contour(x, y, z,
30                  {level='single', title='x + 2y',
31                  xlabel='x', ylabel='y'})
32  }
```

[figure1]



Advanced

Technically functional expressions do not define any relationship. It just picks the values of variables, feeds them into the expression and computes the resulting values for the dependent variable. Therefore when you write a functional expression the values are computed and assigned to the variable and hence the functional expression can contain from zero to many variables provided they have the same dimension. The following examples demonstrates the same idea:

```
x <- [1, 2, 3, 4, 5]
y := x^2                    # y = [1, 4, 9, 16, 25]
# at this point x and y are independent
```

```
x <- [1, 2, 3, 4, 5]
y <- [-1, -2, -3, -4, -5]
z := x + y                  # z = [0, 0, 0, 0, 0]
# at this point x, y and z are independent
```

It's not always necessary to write functional expressions that contain variables. You can also write functional expressions that takes constant values e.g.

```
x := [1, 2, 3, 4, 5]        # sets x = [1, 2, 3, 4, 5]
x := -PI/2                  # sets x = [-PI/2]
```

When using more than one variable in the functional expression you need to make sure that variables have the same number of items e.g.

```
x <- [1, 2, 3, 4, 5]
y <- [-1, -2, -3, -4, -5]
z := x + y                  # valid because x and y have the same number of items
```

```
x <- [1, 2, 3, 4, 5]
y <- [-1, -2, -3, -4]
z := x + y                  # INVALID - number of items in x and y are different
```

And now it should make sense that when you plot the variables there is no relationship between them. It's just that their values were computed with the relationship defined earlier and now these variables are replaced by the list and hence the 1:1 mapping of values are plotted and that forces the size of both of the variables to be the same since there is 1:1 mapping for them in the plot statement.
Also it's not mandatory to plot only variables which are related to each other using functional expression. You can plot any two variables as long as they have the same dimension i.e. same number of items.